Input-Convex Neural Networks and Posynomial Optimization

Spencer Kent SPENCER.KENT@EECS.BERKELEY.EDU
Eric Mazumdar EMAZUMDAR@EECS.BERKELEY.EDU
Anusha Nagabandi NAGABAN2@EECS.BERKELEY.EDU
Kate Rakelly RAKELLY@EECS.BERKELEY.EDU

Abstract

Fitting a model to data is a general problem that arises in many fields. Recently, deep neural networks have been successfully applied to modeling natural images, speech, and language. Posynomial models are another class of models that have been widely applied to problems in engineering design due to their tractability as an optimization problem. Both of these approaches are parametric models for function fitting, though they make different assumptions about the structure of the function being modeled. Neural networks are well-known to be universal function approximators, but this expressiveness comes at the cost of being highly non-convex in their parameters, making them difficult to optimize. By contrast, optimizing posynomial functions can be re-formulated as a convex optimization problem, but these models impose many restrictions on the structure of the learned function. In this paper, we explore connections between deep neural networks and posynomials. First, we explore a network model that trades off expressiveness for more principled optimization. We investigate a novel application of such a network and then we formulate such a model as a posynomial. Finally, we consider the problem of fitting a posynomial function with a neural network and demonstrate fast and accurate posynomial fitting using deep learning machinery.

1. Introduction

Many practical applications require modeling and optimizing over complex system dynamics that are notoriously non-linear and nonconvex functions of the state. Canonical approaches to this problem include assuming simple, known dynamics models for the system or linearizing the dynamics in a region of interest. These approaches are successful in modeling certain systems and allow efficient and interpretable optimization over these systems. However, as we look to analyze more complex systems, these approaches are often too simple to accurately capture the nuances of the underlying system.

To address the challenge of fitting complex models to real systems, robotics researchers have recently applied deep learning techniques to the problem of learning system dynamics (Bansal et al., 2016). Deep neural networks are highly expressive models that have achieved success in applications including visual recognition (Krizhevsky et al., 2012), speech processing (Hinton et al., 2012), and natural language modeling (Collobert & Weston, 2008). These networks derive much of their expressive power from the repeated composition of linear operators with point-wise nonlinearities. In fact, they have been shown to be universal function approximators (Hornik et al., 1989). However, the network function of a standard deep neural network is not convex in its parameters. Thus the optimization of these networks is guided largely by heuristics, and lacks guarantees regarding local optima and convergence rates.

Another set of models that can capture complex system dynamics are posynomial functions. Posynomials are generalized power laws that have long been used to build expressive, non-linear, and nonconvex models. They are widely and successfully applied in various fields, including integrated circuits, aeronautic design, and energy systems. Despite the fact that they are non-convex, the optimization of posynomial models can be cast as a convex optimization problem and solved efficiently. The problem of optimizing over posynomial functions is known as Geometric Programming. More recently, (Calafiore et al., 2014) have shown that the problem of learning posynomial functions from data can be cast as a convex optimization problem.

With this method for learning posynomials, we can view posynomials and neural networks from the perspective of function approximators and compare their benefits and drawbacks. On one hand, neural networks are a much more expressive class of models, but present difficulties for optimization. On the other hand, posynomials are a less expressive class of models, yet Geometric Programming allows us to use the vast literature on convex analysis to optimize over them. In light of these observations, in this paper we analyze the links between posynomial functions and neural networks in an effort to understand the tradeoffs between model expressiveness and ease of optimization.

We begin by reviewing the recent paper (Amos et al., 2016) which presents Input-Convex Neural Networks (ICNNs), a network architecture that trades expressive power for the property of being convex in some of its inputs. We present a novel application of this model to the problem of trajectory following. We then review posynomial models and their reformulation as convex optimization problems. We explore the links between ICNNs and posynomials by first showing that the function captured by an ICNN can be re-written as a posynomial, implying that the prediction step can be solved with geometric programming. Finally, we show that the problem of fitting a posynomial model from data can be approached with neural networks, and that by using gradient descent and deep learning machinery, we are able to estimate posynomial models quickly and with high accuracy.

Final Project for EE227B
Fall 2016

2. ICNN Paper Summary

Practical problems provide additional motivation for models that trade expressivity for principled optimization. For example, deep reinforcement learning (RL) has been recently applied with some success to learning control policies (Mnih et al., 2013), but important challenges remain. In this RL setting, the agent moves between states $\mathbf{s} \in S$ (e.g. (x, y) locations in a grid-world) by taking actions $\mathbf{u} \in U$ (e.g. "move left"). The agent is rewarded for each action according to an arbitrary reward function r. The goal of the popular "Q learning" algorithm is to assign a value to each state-action pair via a Q function $Q(\mathbf{s}, \mathbf{u})$. The agent's policy can then be defined as choosing the action that maximizes $Q(\mathbf{s}, \mathbf{u})$ given its current state. Modeling this Q function as a deep neural network has proven to be a successful approach; however, the action-selection step has proved difficult as it requires finding the minimum of the network function which is non-convex in actions. In light of these motivations, in this section we explore a network model presented in (Amos et al., 2016) that trades expressive power for the property of being convex in some of its inputs. In the reinforcement learning context, input-convexity implies that $Q(\mathbf{s}, \mathbf{u})$ is convex in actions, making the action-selection step a convex optimization. More broadly, input-convex networks can be viewed as an interesting point along the tradeoff curve between expressivity and well-founded optimization. In this section, we reproduce the main theoretical results from (Amos et al., 2016) and analyze some aspects in greater depth than the original paper.

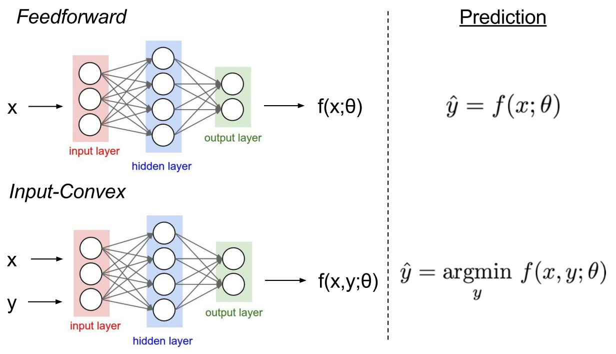

Figure 1. Compare the prediction procedure in an ICNN with that of a traditional feedforward network.

2.1. Problem Setup

In ICNNs, constraints are applied to the general neural network model to make the resulting model convex in some of its inputs. Denote the network function $f(\mathbf{x}, \mathbf{y}; \theta)$ where $\mathbf{x}$ and $\mathbf{y}$ are inputs and θ are the parameters (weights and biases). If all weights applied to the input $\mathbf{y}$ are non-negative, and all activation functions applied to $\mathbf{y}$ are convex and non-decreasing, then f is convex in the input $\mathbf{y}$. This follows from the fact that sums of convex functions are convex, and composing a convex non-decreasing function with a convex function results in a convex function. For brevity of notation, here we write the network equations without the $\mathbf{x}$ input.

$$\mathbf{z}_{i+1} = g_i(\mathbf{W}_i^{(z)}\mathbf{z}_i + \mathbf{W}_i^{(y)}\mathbf{y} + b_i) \text{ , } i = 0, ..., k-1$$
$$f(y; \theta) = \mathbf{z}_k$$

2.2. Prediction

In traditional feed-forward networks, predictions are produced via a single forward pass through the network. In an ICNN, prediction is a convex minimization problem. See Figure 1.

$$\hat{\mathbf{y}} = \operatorname*{argmin}_{\mathbf{y} \in Y} f(\mathbf{x}, \mathbf{y}; \theta) \tag{1}$$

2.2.1. Linear Program Formulation

First, we note that this optimization problem can be written as an LP. (We assume that the activation functions are ReLU.)

$$\begin{aligned} &\min_{\mathbf{y}, \mathbf{z_1}, \ldots, \mathbf{z_k}} \mathbf{z_k} \\ \text{s.t. } &\mathbf{z_{i+1}} \geq \mathbf{W_i}^{(z)}\mathbf{z_i} + \mathbf{W_i}^{(y)}\mathbf{y} + b_i \text{ , } i = 0, ..., k-1 \\ &\mathbf{z_i} \geq 0 \text{ , } i = 0, ..., k-1 \end{aligned}$$

In this formulation, the number of variables is equal to the total number of activations in the network. With a standard deep network model, the large number of activations makes the LP infeasible to solve with standard methods.

2.2.2. Approximation Methods

We would like approximation methods that (1) still exploit the convexity of the problem and (2) take advantage of the fact that we can quickly compute gradients of the network function $\nabla_x f(\mathbf{x}, \mathbf{y}; \theta)$ using backpropogation. In this section, we assume an input $\mathbf{x}$ over which the network has no constraints, and $\mathbf{y}$ over which the network function is convex.

Gradient Descent This approach is straightforward and makes use of gradients, but arguably it is not the best method to exploit the convexity of the problem. Starting with an initial guess for $\hat{\mathbf{y}}$, we perform the update

$$\hat{\mathbf{y}}^{(t+1)} = \hat{\mathbf{y}}^{(t)} - \alpha \nabla_y f(\mathbf{x}, \mathbf{y}; \theta)$$

Choosing an appropriate step size α can be difficult, and since the objective is non-smooth, we can't check for a solution with $\nabla_y f(\mathbf{x}, \mathbf{y}; \theta) = 0$.

Bundle Method A clear drawback of gradient descent is that it uses only the current function evaluation and gradient computation at each iteration. By contrast, the bundle method retains the information gained from all past iterations by building a piecewise linear lower bound on the function. At each iteration this bound is optimized to find the next point of evaluation. This method has three main advantages over gradient descent: (1) it does not require a step size since the next point is given precisely by an optimization, (2) the function value at each iteration is guaranteed to be lower than that of the previous iteration, and (3) we can obtain an estimate of sub-optimality at each iteration.
Let the point evaluated at iteration t be $\mathbf{y}^{(\mathbf{t})}$. At each iteration, we compute a linear approximation to f:

$$f(\mathbf{x}, \mathbf{y}^{(t)}; \theta) + \nabla_y f(\mathbf{x}, \mathbf{y}^{(t)}; \theta)(\mathbf{y} - \mathbf{y}^{(\mathbf{t})})$$

We add this line to our piecewise linear approximation of f, which we denote as $\hat{f}$.

$$\hat{f} = \max_{1 \leq i \leq k} f(\mathbf{x}, \mathbf{y}_i; \theta) + \nabla_y f(\mathbf{x}, \mathbf{y}_i; \theta)(\mathbf{y} - \mathbf{y}_i)$$

Then the next point chosen $\mathbf{y}^{(t+1)}$ will be the minimizer of $\hat{f}$.

$$\mathbf{y}^{(t+1)} = \operatorname*{argmin}_{\mathbf{y}\in Y} \max_{1\le i\le k} f(\mathbf{x}, \mathbf{y}_i; \theta) + \nabla_y f(\mathbf{x}, \mathbf{y}_i; \theta)(\mathbf{y} - \mathbf{y}_i)$$

Letting $\mathbf{G}$ be the matrix with rows $g_i = \nabla_y f(\mathbf{x}, \mathbf{y}_i; \theta)$ and $\mathbf{h}$ a vector with entries $\mathbf{h}_i = f(\mathbf{x}, \mathbf{y}_i; \theta) - \nabla_y f(\mathbf{x}, \mathbf{y}_i; \theta)^T \mathbf{y_i}$, this problem can be re-written as

$$\mathbf{y}^{(t+1)}, t^{(t+1)} = \operatorname*{argmin}_{\mathbf{y}\in Y, t} t \text{ s.t. } \mathbf{G}\mathbf{y} + \mathbf{h} \le t1$$

At each iteration of the bundle method, we must solve this linear program where the number of constraints is equal to the iteration number. As formulated, this problem is still difficult. Since Y is a polytope of n constraints, it would take $n+1$ gradient evaluations to reach the interior of Y, which is too slow in practice.

Entropy Regularization To address the drawbacks of the bundle method, we note that the convex polytope Y can be represented by a set of bounds on each dimension of $\mathbf{y} \in Y$. So we can encode the constraint $\mathbf{y} \in Y$ by placing a barrier function in the objective.

$$\min_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}; \theta) - H(\mathbf{y})$$

$-H(y)$ is a convex function with $\lim_{y\to 0} H(y) = \lim_{y\to 1} H(y) = 0$. It acts as a barrier since its gradients approach ∞ as y approaches 0 or 1, thus ensuring that the solution $\hat{y}$ lies in the unit hypercube. This hypercube can be transformed to Y by scaling. We can now pose the problem from each iteration of the bundle method as

$$\mathbf{y}^{k+1}, t^{k+1} = \operatorname*{argmin}_{\mathbf{y}, t} t - H(\mathbf{y}) \text{ s.t. } \mathbf{G}\mathbf{y} + \mathbf{h} \le t1$$

We find that the Lagrangian dual of this problem is

$$\begin{aligned} &\max_{\boldsymbol{\lambda}} \; (\mathbf{G}\mathbf{1} + \mathbf{h})^T \boldsymbol{\lambda} - \mathbf{1}^T log(1 + exp(\mathbf{G}^T \boldsymbol{\lambda})) \\ &\quad \text{s.t. } \boldsymbol{\lambda} \ge 0 \, , 1^T \boldsymbol{\lambda} = 1 \end{aligned}$$

This is a smooth optimization problem over the unit simplex, which can be solved efficiently with the projected Newton method. Given the optimal solution $\boldsymbol{\lambda}_k$, we can compute $\mathbf{y}_{k+1}$.

$$\mathbf{y}_{k+1} = (1 + exp(\mathbf{G}_k^T \boldsymbol{\lambda}_k))^{-1}$$

To recap, prediction in the ICNN model amounts to finding the minimizer of the convex function $f(\mathbf{x}, \mathbf{y}; \theta)$. Because the LP formulation has too many variables and gradient descent does not fully exploit convexity, (Amos et al., 2016) propose en entropy regularized bundle method which involves repeatedly minimizing a piecewise lower bound on f. This inner loop minimization is performed by solving the dual problem and mapping the dual solution back to the primal problem via the simple expression given above.

2.3. Learning

The objective of learning in the ICNN model is to shape the network function such that optimization of this convex function with respect to $\mathbf{y}$ yields a minimizer $\hat{\mathbf{y}}$ that is close to a true value $\mathbf{y}^\star$. Specifically, for the training data pair $(\mathbf{x}, \mathbf{y}^\star)$ we would like the following:

$$\mathbf{y}^\star \quad \approx \quad \hat{\mathbf{y}} = \arg\min_{\mathbf{y}} \tilde{f}(\mathbf{x}, \mathbf{y}; \theta)$$

Where, to recap, $\tilde{f}(\mathbf{x}, \mathbf{y}; \theta)$ is just the network function augmented with the entropy: $\tilde{f}(\mathbf{x}, \mathbf{y}; \theta) = f(\mathbf{x}, \mathbf{y}; \theta) - H(\mathbf{y})$.

2.3.1. Maximum-Margin Prediction

In the most general classification or regression setting, another way to think of the learning problem is that the network assigns a *score*, $g(\mathbf{x}, \mathbf{y}; \theta)$ given simply as the negative cost $-\tilde{f}(\mathbf{x}, \mathbf{y}; \theta)$, for a given setting of θ and pair $(\mathbf{x}, \mathbf{y})$. This may remind us of the SVM, or, more generally max-margin structured prediction. The objective in these models is to find a decision boundary that separates data points $\{\mathbf{x}\}$ based on the score, some margin around the decision boundary, and slack terms ξ_i that allow some small number of violations of the boundary and margin.

The authors of the original ICNN model briefly discuss the framework of maximum-margin prediction for learning the parameters θ of the network. They frame the problem (in equation (32) of (Amos et al., 2016)) for a dataset $\{(\mathbf{x}_i, \mathbf{y}_i^\star)\}_{i=1}^n$ as the following minimization problem:

$$\begin{aligned} \min_\theta \quad & \frac{\lambda}{2}||\theta||_2^2 + \sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & \tilde{f}(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) - \xi_i \le \tilde{f}(\mathbf{x}_i, \mathbf{y}; \theta) - \ell(\mathbf{y}_i^\star, \mathbf{y}) \quad \forall i, \mathbf{y} \end{aligned} \tag{2}$$

(there is a typo in their paper, $-\xi_i$ should be on the left-hand side of the inequality as shown here). The constraint says that the energy, or cost, of the supervised input pair $(\mathbf{x}_i, \mathbf{y}_i^\star)$ should be lower than the energy of the pair $(\mathbf{x}_i, \mathbf{y})$, minus some margin, for any other value of $\mathbf{y}$ (and allowing for some small number of violations of this constraint). Switching from an *energy* to a *score*, this is equivalent to the constraint:

$$g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) + \xi_i \ge g(\mathbf{x}_i, \mathbf{y}; \theta) + \ell(\mathbf{y}_i^\star, \mathbf{y}) \;\; \forall i, \mathbf{y}$$

which is what we will use herein due to it being more common to the max margin formulation. One difference between max-margin prediction for ICNNs and the traditional problem is that the parameters θ are combined in a nonlinear way to produce the score $g(\mathbf{x}, \mathbf{y}; \theta)$, as opposed to the score being a linear function of θ. For the max margin approach to learning in ICNNs we have the optimization problem:

$$\begin{aligned} \min_\theta \quad & \frac{\lambda}{2}||\theta||_2^2 + \sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) + \xi_i \ge g(\mathbf{x}_i, \mathbf{y}; \theta) + \ell(\mathbf{y}_i^\star, \mathbf{y}) \quad \forall i, \mathbf{y} \end{aligned} \tag{3}$$

Here, ξ_i are slack terms which allow violation of the margin imposed by $\ell(\mathbf{y}_i^\star, \mathbf{y})$ which simply must satisfy $\ell(\mathbf{y}_i^\star, \mathbf{y}) > 0 \; \forall \mathbf{y} \ne \mathbf{y}_i^\star$ and $\ell(\mathbf{y}_i^\star, \mathbf{y}_i^\star) = 0$. In the case of binary classification in the traditional SVM framework this would be the "0-1" loss but in general could be some other loss function. The max margin formulation here can be solved as a quadratic program (using the dual). However, this may be intractable for problems that have large $\mathbf{y}$ because the number of constraints grows exponentially with the number of elements in $\mathbf{y}$. As a result, the original max-margin work of (Taskar, 2004) develops an approximate method called "Structured Sequential Minimial Optimization" to solve the problem. However, subsequent work in (Ratliff et al., 2007) reports that this still may not be efficient enough for certain problems and so they develop a subgradient approach to solving the max margin structured prediction problem. The setup is as follows:

We can process the constraint in (3) by taking the $\max$ over $\mathbf{y}$ and substituting the constraint written for ξ_i into the objective function:

$$\min_{\theta} \quad \frac{\lambda}{2}||\theta||_2^2 - \sum_{i=1}^{n} g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) - \max_{\mathbf{y}} \left(g(\mathbf{x}_i, \mathbf{y}; \theta) + \ell(\mathbf{y}_i^\star, \mathbf{y}) \right) \tag{4}$$

This is the optimization problem solved via the subgradient method developed in (Ratliff et al., 2007) which is what the authors of the ICNN paper considered. They report poor empirical performance of this algorithm and use that as a justification for employing a different, more black-box method for learning, but we believe there may be much more to be explored here. For one, as reported in (Taskar, 2004) and (Ratliff et al., 2007), the particular choice of the loss function $\ell(\mathbf{y}_i^\star, \mathbf{y})$ can have a large impact on the overall expected risk of the model and for making prediction tractable. For this reason, (Taskar, 2004) uses a Hamming distance type loss, which is not something considered by (Amos et al., 2016). Also, this is a subgradient method, which can be notoriously slow to converge in many cases. By slightly rethinking the learning problem it may be possible to acheive better performance. In particular, we propose that a more useful approach, at least in the classification setting, is to think of the learning problem from the perspective of maximum entropy.

2.3.2. Maximum Entropy

Maximum entropy discriminative models are motivated by the task of maximizing the conditional likelihood of the training data under a softmax likelihood model. The setup is

$$P(\mathbf{y}|\mathbf{x};\theta) = \frac{\exp g(\mathbf{x}, \mathbf{y}; \theta)}{\sum_{\mathbf{y}'} \exp g(\mathbf{x}, \mathbf{y}'; \theta)}$$

To maximize the likelihood for the entire dataset we have

$$\begin{aligned} &\max_{\theta} \quad \log \prod_i P(\mathbf{y}_i^\star | \mathbf{x}_i; \theta) \\ &= \max_{\theta} \sum_i \log \left(\frac{\exp g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta)}{\sum_{\mathbf{y}'} \exp g(\mathbf{x}_i, \mathbf{y}'; \theta)} \right) \\ &= \max_{\theta} \sum_i \left(g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) - \log \sum_{\mathbf{y}'} \exp g(\mathbf{x}_i, \mathbf{y}'; \theta) \right) \end{aligned}$$

We can turn this into a minimization problem and add a regularization term for the parameters θ to get

$$\min_{\theta} \frac{\lambda}{2}||\theta||_2^2 - \sum_i \left(g(\mathbf{x}_i, \mathbf{y}_i^\star; \theta) - \log \sum_{\mathbf{y}'} \exp g(\mathbf{x}_i, \mathbf{y}'; \theta) \right) \tag{5}$$

This expression minimizes the “log-loss” on each sample $\mathbf{x}_i$ where the log-loss is an upper bound (up to a constant) on the zero-one loss. This is the unconstrainted optimization problem we would like to solve in Maximum Entropy. Compare equation (5) to equation (4). We've replaced a non-differentiable objective (hence requiring a subgradient approach) with one that is continuously differentiable. Maximum entropy models like this one can thus be fit with general purpose optimizers and are commonly fit using approaches like conjugate gradient or limited-memory BFGS, which could perform much better than the method of (Ratliff et al., 2007) on this problem. Maximum entropy models like kernelized logistic regression are competitive with SVM models in many cases, although the hingle loss of SVM tends to produce sparse solutions, which can be nice. While we did not explore it further in this project, we think a next direction would be to implement prediction in ICNNs as a maximum entropy problem to see how it compares with some of these other methods on real datasets.

2.3.3. Direct Autodifferentiation

The actual approach to learning taken by (Amos et al., 2016) is a little less well-motivated but is nevertheless a common type of approach in neural network models. They take some arbitrary loss function $\ell(\mathbf{y}^\star, \hat{\mathbf{y}})$ imposed on the output of prediction, $\hat{\mathbf{y}}$ and the supervised values $\mathbf{y}^\star$ and compute the gradient of this function with respect to the parameters of the network, θ. The value $\hat{\mathbf{y}}$ is itself the result of the optimization problem (1) solved during the prediction step which is why this is not just a straightforward gradient derivation. For brevity, we won't reproduce the derivation here (it can be found in section 5.3 of (Amos et al., 2016)) but will just include the three key equations involved.

$$\begin{aligned} \nabla_\theta \ell(\hat{\mathbf{y}}(x;\theta), \mathbf{y}^\star) = \sum_{i=1}^{k} \Big(& c_i^\lambda \nabla_\theta f(\mathbf{x}, \mathbf{y}^t; \theta) \\ & + \nabla_\theta \left(\nabla_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}^t; \theta)^\top \left(\lambda_i \mathbf{c}^y + \mathbf{c}_i^\lambda \left(\hat{\mathbf{y}}(\mathbf{x};\theta) - \mathbf{y}^t \right) \right) \right) \Big) \end{aligned} \tag{6}$$

This represents the gradient of some arbitrary loss function with respect to the parameters θ. It is used to take stochastic gradient steps using the first-order accelerated gradient method ADAM (Kingma & Ba, 2014). The complexity of this expression is, again, related to the fact that it accounts for iterates of the inner optimization procedure that takes place in prediction. We have that

$$\begin{aligned} \frac{\partial l}{\partial \theta} &= \frac{\partial l}{\partial \hat{\mathbf{y}}} \left(\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{G}} \frac{\partial \mathbf{G}}{\partial \theta} + \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \theta} \right) \\ &= \left(\frac{\partial l}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{G}} \right) \frac{\partial \mathbf{G}}{\partial \theta} + \left(\frac{\partial l}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}} \right) \frac{\partial \mathbf{h}}{\partial \theta} \end{aligned}$$

Where the quantities $\mathbf{G}$ and $\mathbf{h}$ we have defined above in section 2.2.2 on the Bundle Entropy prediction method but will be restated, namely that the matrix $\mathbf{G}$ is computed in the prediction step, specifically from each iterate $\mathbf{y}^t$. The t^{th} row of $\mathbf{G}$, $\mathbf{g}_t^\top$, is given by

$$\mathbf{g}_t^\top = \nabla_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}^t; \theta)^\top$$

The vector $\mathbf{h}$ keeps track of the relative change in function value at each $\mathbf{y}^t$, namely

$$h_t = f(\mathbf{x}, \mathbf{y}^t; \theta) - \nabla_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}^t; \theta)^\top \mathbf{y}^t$$

This is the first step in the derivation of (6) above for which each of the constituent terms is identified below:

- The term $\mathbf{y}^t$ is the value of $\mathbf{y}$ on the t^{th} iteration of prediction. (The bundle-entropy method keeps track of a $\mathbf{y}$ for each iteration it takes)
- λ is the dual variable solution provided by the bundle entropy method
- The variables $\mathbf{c}^\lambda$ and $\mathbf{c}^t$ come from the solutions to the following linear system:

$$\begin{bmatrix} \text{diag}\left(\frac{1}{\hat{\mathbf{y}}} + \frac{1}{1-\hat{\mathbf{y}}} \right) & \mathbf{G}^\top & 0 \\ \mathbf{G} & 0 & -\mathbf{1} \\ 0 & -\mathbf{1}^\top & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{c}^y \\ \mathbf{c}^\lambda \\ \mathbf{c}^t \end{bmatrix} = \begin{bmatrix} -\nabla_{\mathbf{y}} \ell(\hat{\mathbf{y}}, \hat{\mathbf{y}}) \\ 0 \\ 0 \end{bmatrix} \tag{7}$$

The terms of the form $\nabla_\theta(\nabla_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}^t; \theta)^\top v)$ are what are difficult to analytically compute in (6). However, using the the autodifferentiation tools of modern deep-learning engines, one can numerically compute compute this quantity. The way to think of this is that the term $\nabla_{\mathbf{y}} f(\mathbf{x}, \mathbf{y}^t; \theta)^\top v$ is simply a graph computation which can differentiated itself or can be approximated with a finite difference approximation

2.4. Application: Q-learning for Robotics

Returning to the RL application partially motivating this section, we mention that ICNNs can be mapped easily into the domain of reinforcement learning without all of the details of learning discussed above. The Q function is given by the input-convex network equation (specifically $-\tilde{f}(\mathbf{x}, \mathbf{y}; \theta)$) where $\mathbf{x} = \mathbf{s}$ and $\mathbf{y} = \mathbf{u}$. Thus prediction in ICNNs produces the action that maximizes the Q function which is precisely what we aim to do in reinforcement learning. Learning the Q function amounts to standard backpropogation in the ICNN, as the values of the Q function are directly supervised by the Bellman equation. Thus the prediction optimization procedure is not needed during training, precluding the need for the learning procedure described above. However when the model is deployed, control actions can be selected via convex optimization. For a slightly more detailed discussion please see (Amos et al., 2016).

3. Novel ICNN Extension: Trajectory Following

3.1. The Task

Now that we have introduced the ICNN which exchanges model expressiveness for optimization efficiency and optimality guarantees, we apply it to a robotics task. Consider the common task of generating control inputs to make a robot follow a desired trajectory. When the robot dynamics are known, they can be used to compute what state the robot will end up in as the result of applying a certain control input. In this case, various techniques exist for solving for the optimal controls, such as Trajectory Optimization and Model Predictive Control. These approaches amount to selecting the control input at each time step that minimizes the difference between the desired trajectory and the resulting trajectory (defined by the dynamics).

For certain systems, however, the dynamics are not known. In these cases, possible methods for optimal control include approximating the dynamics with simpler known models, or leveraging deep learning techniques to learn system dynamics (Bansal et al., 2016). For robots such as the VelociRoACH (Zarrouk et al., 2015) (Figure 2), these approaches are still insufficient for a number of reasons. These small, fast, inexpensive, and extremely mobile robots are rapidly prototyped and nontrivial to control. They are under-actuated, only partially observable, and are meant for operation on irregular terrain. Additionally the system dynamics are complex due to discontinuities caused by ground contacts, and they vary between robots due to their lightweight and low-cost manufacturing. We aim to directly learn to generate optimal controls for following a desired trajectory by learning a representation of the system's dynamics.

3.2. Trajectory Following in the ICNN Framework

Note the following notation clarifications to transition from the previous sections (that review ICNNs) to the current section (regarding trajectory following):

$$y \quad \rightarrow \quad \begin{bmatrix} u_0 \\ \vdots \\ u_{T-1} \end{bmatrix}$$

$$x \quad \rightarrow \quad \begin{bmatrix} x_0 \\ x_{1desired} \\ \vdots \\ x_{Tdesired} \end{bmatrix}$$

$$f(x,\ y;\ \theta) \quad \rightarrow \quad f(x_0,\ x_{1:Tdesired},\ u_{0:T-1};\ \theta)$$

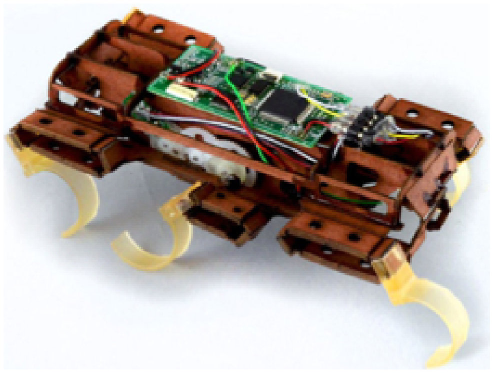
Figure 2. The VelociRoaCH: Small, fast, mobile legged robot from the Biomimetic Systems Lab at UC Berkeley.

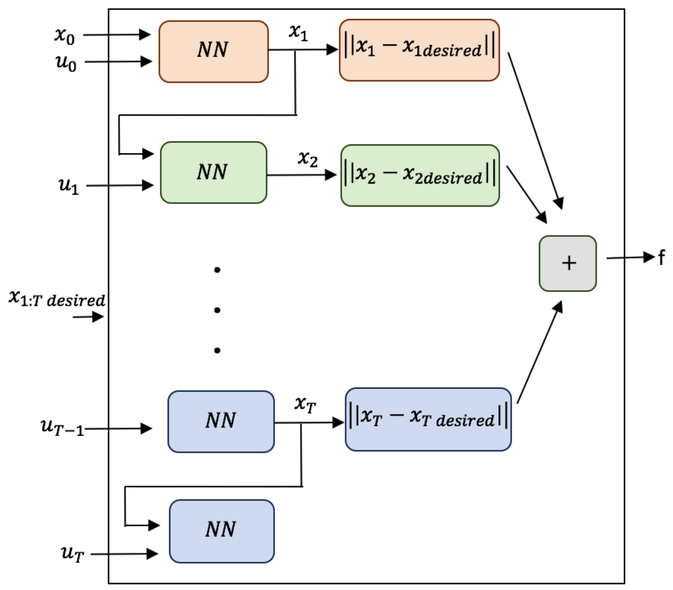

Figure 3. The proposed network architecture for the task of trajectory following. Inputs are starting state, sequence for desired trajectory, and sequence for control inputs. Each NN block is a neural network with two hidden layers.

The goal for this prediction task is that given a starting state (x_0) and the desired trajectory ($x_{1:Tdesired}$), we want to find the optimal controls ($u_{0:T-1}$) to follow that trajectory.

Figure 3 shows our proposed ICNN architecture for this task. Each NN block is a neural network which has two hidden layers and is convex in u. The first NN outputs the learned representation for x_1, the robot state after applying u_0 starting from x_0. This state is then fed into the next NN, which outputs the result of applying u_1 from that x_1. The final output of the network is then $f = ||x_{1:T} - x_{1:Tdesired}||_2^2$. With this definition for f, the prediction step of finding optimal controls is now a convex optimization problem:

$$\hat{u} = argmin_u\ f(x_0,\ x_{1:Tdesired},\ u_{0:T-1};\ \theta) \tag{8}$$

This formulation now provides the intuition that the optimal controls are those which minimize the difference between the resulting trajectory and the desired trajectory.

At each step of training of this network, the prediction step (8) gives us $\hat{u}$ as a result of convex minimization. We can supervise the u values by defining the loss function as $L = ||\hat{u} - u^*||_2$. Following the training procedure outlined in (Amos et al., 2016), network parameters are updated with gradient descent $\theta_{t+1} = \theta_t - \nabla_\theta L$, where the gradients are computed as described in Section 2.3. Denoting the learned network parameters as θ^*, we can solve for optimal controls at any time in the future by solving the prediction problem (8) with the learned network parameters θ^*.

3.3. Training the ICNN on Single Step Trajectories

Consider the simpler version of the task with only one time step, instead of a sequence. Now, we want to know what control input u will get the system from starting state x_0 to desired state x_1. This corresponds to looking at just the first block section of the sequence architecture proposed above in Figure 3. The general input/output layout of this task is shown below in Figure 4. We use the same architecture as proposed in (Amos et al., 2016) which is illustrated in detail in Figure 5.

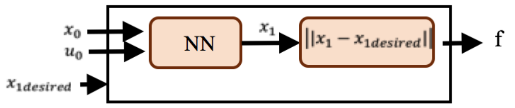

Figure 4. High-level network layout for the simplified task of solving for the control input that gets you to the desired next state. The inputs to the network are starting state, control input, and next desired state. The intermediate output is the learned representation of the next state. The output is the norm difference between that next state and the desired next state.

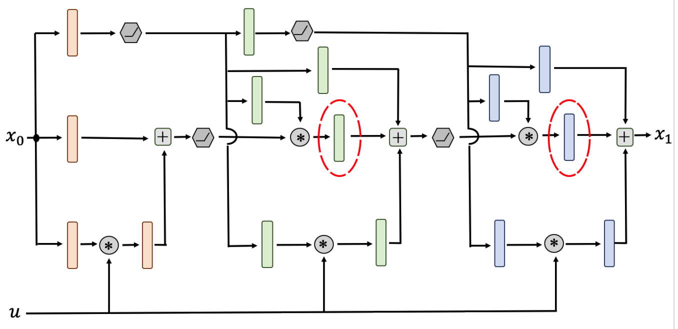

Figure 5. The detailed network architecture for the NN block shown in the previous figure, where the inputs are current state and control input, and the output is the resulting state. Note that each colored rectangle here is a fully connected layer, and the other blocks are addition, multiplication, and the ReLU activation function. The circled layers are restricted to having positive weights to ensure convexity in u.

3.3.1. Implementation

To implement this model, we used GPU-enabled open source numeric computation library TensorFlow (Abadi et al., 2016) and built our code on top of the ICNN implementation provided by (Amos et al., 2016). For simplicity, we trained the network on the task of generating controls for a point mass moving in 2-D. The force F applied to the point mass is interpreted as our control input u, and we hold that input to be constant over a time period dt. The state x of the point mass consists of its position as well as its velocity. Thus, each training example consists of ($x_0 = \begin{bmatrix} p_0 \\ v_0 \end{bmatrix}$, $x_1 = \begin{bmatrix} p_1 \\ v_1 \end{bmatrix}$, $u = F$), and we obtain this data by applying the following equations of motion:

$$a_t = \frac{F_t}{m} \tag{9}$$

$$p_{t+1} = p_t + v_t * dt + \frac{1}{2} a_t * dt^2 \tag{10}$$

$$v_{t+1} = v_t + a_t * dt \tag{11}$$

We implemented the prediction optimization with gradient descent rather than the bundle entropy method, because our objective is not quite convex. Note that the prediction for the next state x_1 is convex with respect to u; however, f is the composition of the 2-norm with this function of u: $f = ||x_1 - x_{1desired}||_2^2 = ||g(u) - x_{1desired}||_2^2$. Thus, f is not convex in u because the 2-norm is not a non-decreasing function. We claim, however, that the composition of a norm and a convex function cannot contain any sub-optimal local minima, see Figure 6. Thus, gradient descent should still find a globally optimal value when minimizing f.

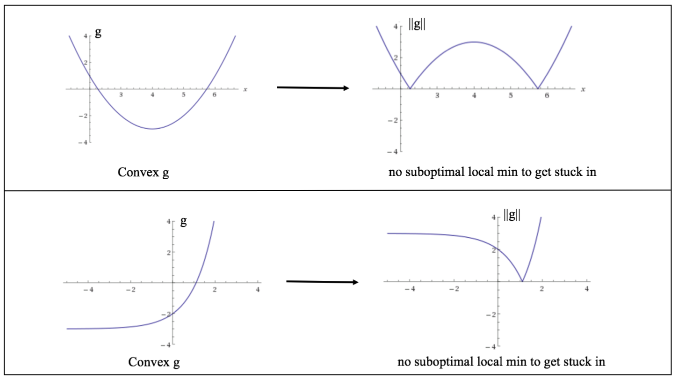

Figure 6. Taking norm of a convex function does not introduce any suboptimal local minima.

During implementation, we found that it was important to use a lot of training data, add Gaussian noise to it, and shuffle it after each epoch of training. We found that model training was highly sensitive to learning rate and momentum. We also found that initializing the network parameters θ with Xavier initialization was helpful, and that using full-batch gradient descent instead of batched stochastic gradient descent resulted in smoother optimization.

3.3.2. Results

First we should note that in the setup described above, the parameters θ are adjusted in order to bring the predicted $\hat{u}$ (which is the minimum of the function $f = ||x - x_{desired}||_2^2$) closer to the desired u^*. Although the problem involves finding the minimum of f, note that over the course of training, this learning procedure does not guarantee the direct reduction of the value of f(see Figure 7). We experimentally verify this intuition in Figure 8 where we plot the magnitude of f on the left and the error in the prediction of u on the right over epochs of training. During training, the magnitude of f stayed constant, indicating that the network did not learn to predict the correct next state. Also note that from that plot, it looks like the network learned to correctly predict u, but this is not actually the

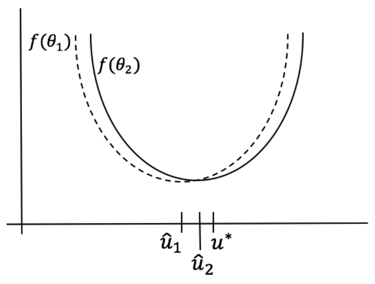

Figure 7. Although u is defined as the min of f, supervising the u (to get closer to u^*) might not lead to f actually decreasing in value.

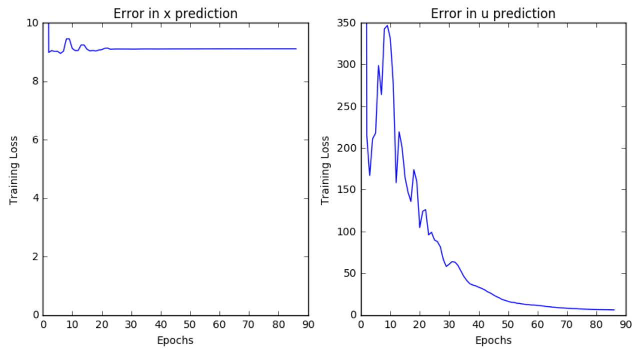

Figure 8. Only u supervised. Left: Training error for the next state x (does not decrease). Right: Training error for control input u (stops decreasing at a value of 5, indicating the inability to learn on our task).

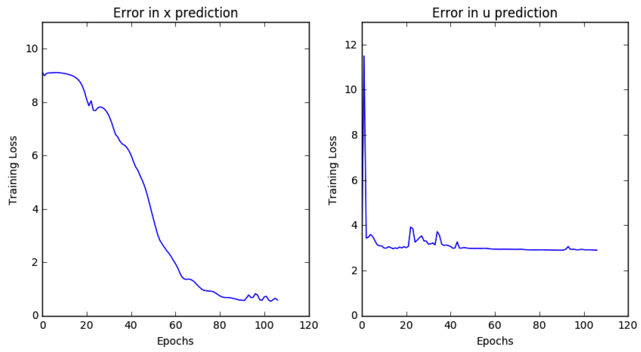

Figure 9. Both u and x supervised. Left: Training error for the next state x. Right: Training error for the control input u (stops decreasing at a value of 2.5, indicating the inability to learn on our task).

case because the loss plateaued at a value of 5: the range of u in the training data was only from -5 to 5, so an error of 5 indicates that nothing meaningful was learned.

To address this issue of f (the x loss) not decreasing during training, we directly supervise x by adding a term to the loss function so that $L = \alpha||\hat{u} - u^*||_2 + \beta||x - x_{desired}||_2$, where α and β are weights. These results are shown in Figure 9. While the error in state prediction becomes small, the error in control input estimation remains high. Intuitively, this is because the loss on u adjusts the parameters so that u is the minimum of $||x - x_{desired}||_2^2$, even when the predicted state x is wildly incorrect at the beginning of training.

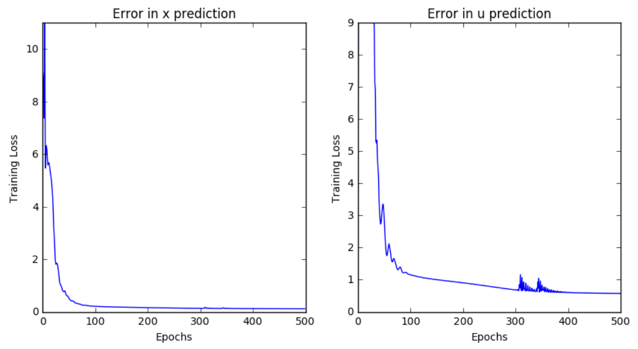

Figure 10. Only x supervised, and correct u^* values used during training. Left: Training error for the next state x. Right: training error for control input u.

This observation suggests that a training approach similar to the approach used for Q learning may be more appropriate. We first want to learn the correct relationship between (x_0, u) and (x_1). We do this by train the network as a function of x_0 and the *true* control input u^*, and by supervising only the state x prediction with $L = ||x - x_{desired}||_2^2$. Once this relationship is learned, the network now knows the function f, where $u*$ is the minimum over u of this function. This is similar to the approach used in Q learning, where the network predicts Q-values which are directly supervised based on the Bellman equation.

This approach worked the best in terms of prediction error for both state and control input. Figure 10 displays the progression of the x and u loss during training, and Figure 11 shows the trajectory that we achieve from applying the controls as predicted by our learned network function. One noteworthy realization during debugging was that the dt timestep value that we used for point mass simulation had a large effect on training. Since $p_1 = ... + u * dt^2$, having $dt = 0.001$ essentially made the next state value independent of u. In this case, we cannot hope to learn any meaningful way to predict u based on the next state. Thus, we increased the dt to 0.1, which physically translates to a time-step of 0.1 seconds and results in a controller that runs at $10Hz$.

We generated 20,000 training points, using the point mass equations 9-11. Training for 500 epochs took a total of 44.05 seconds, and it resulted in 0.10 training loss in x and 0.548 training loss in u. On a validation set of 10,000 points, this achieved a 0.30 loss in u. Note that it is not surprising for this validation loss to be lower than the training loss, because we artificially added noise to the training data. When fitting a function to the true underlying model rather than directly to the given data, we would indeed expect the error on un-noised data to be lower than the error on noised data.

3.4. Alternative Formulation of the Network Architecture

Instead of sequence-to-sequence learning where we output a sequence of optimal controls given the sequence of desired states, we consider on online approach. Rather than learning to predict the next state, we learn only the optimal control given the initial state x_0 and desired state x_1. In this network architecture (Figure 12), inputs are x_0, x_1, and u, and the network function f is an energy function (which will be convex with respect to u). We propose to use our model for trajectory following by performing the prediction $u = min_u \; f(x_0, \; x_1, \; u; \; \theta)$ at every time step along the desired trajectory, using the actual next state as the following step's start-

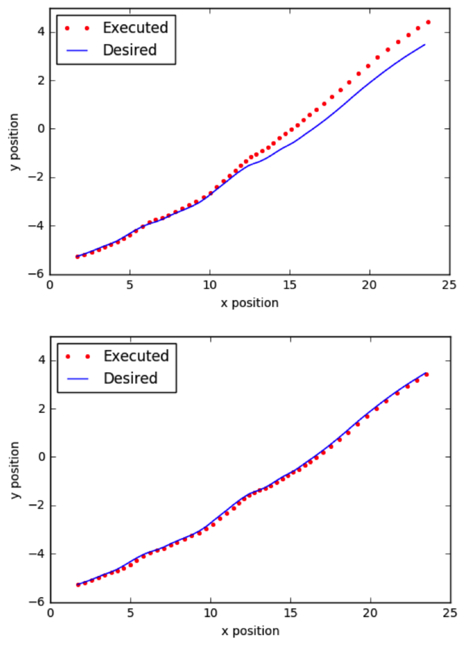

Figure 11. Desired trajectory vs. the trajectory that we achieve from applying the controls as predicted by our learned network function. Shown after training the network for 30 epochs (top) and for 500 epochs (bottom).

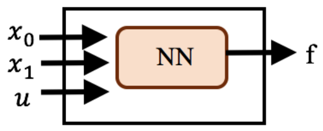

Figure 12. An alternative network architecture. The inputs are starting state x_0, desired next state x_1, and control input u. Now, f is the direct output of the neural network, representing an energy function.

ing state. Since this prediction step can be performed quickly via gradient descent, we can achieve real-time control.

We applied this setup to the same task as before of the point mass moving in 2-D (equations 9- 11), see Figure 13. It is important to note that hyper-parameters (learning rates, momentum values for gradient descent, number of gradient iterations, etc.) during training made a large difference in the results. For this task of generating optimal controls u for a point mass to reach a desired next state x_1, our loss in u during training (Figure 13) was 0.55, offering approximately the same results as the previous formulation in Section 3.3.

3.5. Future Directions

These initial experiments demonstrate the potential of using ICNNs to learn dynamics for optimal control problems; however, much remains to be done to apply this model to complex systems. For many systems, dynamics are not convex in control inputs, so ICNNs cannot be directly applied. We propose to use a conventional neural network to map the control input u into a high-dimensional space. Then it is possible that the dynamics could be convex in the transformed u and thus the ICNN model is appropriate. This approach can leverage the network architecture of the ICNN to train both the conventional neural network and the ICNN model in an end-to-end fashion. Compounding this problem, for robots such as the VelociRoACH the dynamics are discontinuous, implying that a small perturbation in control input u can cause a large difference in the resulting state x. Further investigation is required to develop robust methods for learning such dynamics. Despite these challenges, ICNNs offer a promising step in combining the benefits of deep learning with those of convex optimization. We next investigate the relationship between ICNNs and posynomial models.

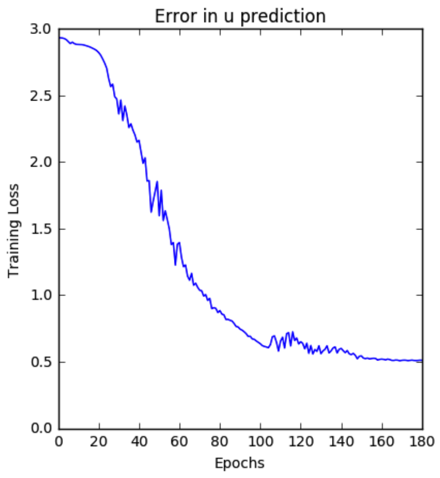

Figure 13. Loss plot for the control input u, with the new setup as shown in Figure 12, and with the point mass dynamics given by equations 9- 11.

4. Posynomials and ICNNs

Input convex neural-networks, in theory, offer us an appealing trade-off between the expressiveness of the model we want to learn and the ease of optimization of said model. Another type of model that has a similar appeal are posynomials. These are generalized power laws that have been used to build expressive, non-linear, and non-convex models which could in turn be optimized over efficiently using a body of work known as *Geometric Programming*. Such models arise in various fields including, but not limited to, integrated circuits, aeronautic design, and energy systems.

A posynomial, is a function $f : \mathbb{R}^n \to \mathbb{R}$, whose domain is $\mathbb{R}^n_{++}$ and has the form:

$$f(\mathbf{x}) = \sum_{i=1}^{K} g_i(\mathbf{x})$$

where each g_i is known as a monomial and has the form:

$$g_i(\mathbf{x}) = c_i \prod_{j=1}^{n} x_j^{\alpha_{ij}}$$

for $c_i \in \mathbb{R}_+$ and $\alpha_i \in \mathbb{R}^n$.

It is clear to see that the space of posynomials is closed under multiplication and addition with positive scalars. Further, the product of any two posynomials is itself a posynomial, and the sum of two posynomials is a posynomial.

We define a generalized posynomial to be any function $F : \mathbb{R}^n \to \mathbb{R}$ which is obtained from posynomials through point-wise maximum, or being raised to a constant *positive* power.

In this section, we first conduct an overview of Geometric Programming methods and the way they let us optimize over posynomials in an efficient way. We then outline connections between posynomials and ICNNs. Finally we show that we can use deep neural networks to learn posynomial models from data, allowing us to leverage the tools and techniques developed for training deep neural networks for learning posynomial models from the data.

4.1. Geometric Programming

The problem of optimizing over posynomial functions, known as *Geometric Programming*, is hard at first glance, due to the fact that posynomials (and consequently generalized posynomials) are generally non-convex functions. For example, the function $f(x) = x^{\frac{1}{2}}$ is a non-convex function in one-dimension. We review the method of reformulating a geometric program in standard form as a convex optimization problem.

The standard form for geometric programs is:

$$\begin{aligned} \min_{\mathbf{x}} \ f_0(\mathbf{x}) :& \\ f_i(\mathbf{x}) \leq 1, &\quad i = 1, ...m \\ g_j(\mathbf{x}) = 1, &\quad j = 1, ...p \end{aligned}$$

Where $f_0, ..., f_m$ are generalized posynomials, and $g_1, ...g_p$ are monomials. Since the objective and constraints are generally non-convex functions, geometric programs in their standard form are not convex. However, they can be reformulated as convex optimization problems through a change of variables.

4.1.1. Writing Posynomials as Convex Functions

We first show that we can rewrite a posynomial as a convex function through variable substitution. We define $\mathbf{y} = log(\mathbf{x})$ and rewrite each monomial g_i as a function of $\mathbf{y}$.

$$\tilde{g}(\mathbf{y}) = c_i e^{{\alpha_i}^\top \mathbf{y}}$$

The posynomial in turn becomes a weighted sum of exponential functions of our new variable $\mathbf{y}$:

$$\tilde{f}(\mathbf{y}) = \sum_{i=1}^{K} c_i e^{{\alpha_i}^\top \mathbf{y}}$$

Since each monomial is a convex function, $\tilde{f}$ is a convex function of $\mathbf{y}$. Further, by minimizing $\tilde{f}$ with respect to $\mathbf{y}$ we minimize f with respect to $\mathbf{x}$. The crux of the argument for this is the fact that e^x is the inverse of $log(x)$, and thus the range of $\tilde{f}$ is the same as that of f.

For numerical reasons, we often optimize over $log(\tilde{f}(\mathbf{y}))$ since the values of a posynomial are sometimes too large to be numerically feasible to optimize over. This still allows us to use convex optimization techniques since the log-sum-exp function is still convex (Boyd & Vandenberghe, 2010).

4.1.2. Convex Form of Geometric Programs

Given that a posynomial can be written as a convex function of a different variable, we now rewrite the standard geometric program as a convex optimization problem. We first show how to write each generalized posynomial as a posynomial.

If f_i for $i \geq 1$ is a generalized posynomial then we can introduce a scalar variable t, and rewrite the constraint $f_i(\mathbf{x}) \leq 1$ as two different constraints:

$$f_i(\mathbf{x}) \leq 1 \iff \begin{cases} t \leq 1 \\ f_i(\mathbf{x}) \leq t \end{cases}$$

If f_i was obtained by raising a posynomial h_i to a constant positive power β, the second constraint becomes:

$$\begin{aligned} (h_i(\mathbf{x}))^\beta &\leq t \\ h_i(\mathbf{x}) &\leq t^{\frac{1}{\beta}} \\ \tilde{f}_i(t, \mathbf{x}) := t^{-\frac{1}{\beta}} h_i(\mathbf{x})) &\leq 1 \end{aligned}$$

$t > 0$ by virtue of the fact that $h_i > 0$ for all $\mathbf{x}$. Thus $\tilde{f}_i$ is a posynomial function of $\mathbf{x}$ and t and the generalized posynomial constraint is now a posynomial constraint:

$$f_i(\mathbf{x}) \leq 1 \iff \begin{cases} t \leq 1 \\ \tilde{f}_i(\mathbf{x}, t) \leq 1 \end{cases}$$

If f_i was obtained by taking a point-wise maximum of two posynomials h_{i1} and h_{i2}, the generalized posynomial constraint is equivalent to the posynomial constraints:

$$f_i(\mathbf{x}) \leq 1 \iff \begin{cases} t \leq 1 \\ h_{i1}(\mathbf{x}) \leq t \\ h_{i2}(\mathbf{x}) \leq t \end{cases}$$

Which can be rewritten as:

$$f_i(\mathbf{x}) \leq 1 \iff \begin{cases} t \leq 1 \\ \tilde{h}_{i1}(\mathbf{x}, t) := t^{-1} h_{i1}(\mathbf{x}) \leq 1 \\ \tilde{h}_{i2}(\mathbf{x}, t) := t^{-1} h_{i2}(\mathbf{x}) \leq 1 \end{cases}$$

If f_0, the objective function we are minimizing over, is a generalized posynomial, we can rewrite the objective as:

$$\begin{aligned} \min_{\mathbf{x}, t} \ t \ : \\ f_0(\mathbf{x}) \leq t \end{aligned}$$

We can then use the same techniques as above to process the new constraint.

Thus, given a geometric program with generalized posynomials in the objective or the constraints, we can reduce it to a geometric program with only posynomial objectives or constraints by adding variables. We can then use the technique outlined in section 4.1.2 to rewrite each posynomial as a convex function of $\mathbf{y} = log(\mathbf{x})$, resulting in a convex optimization problem.

4.2. Links between ICNNs and Posynomials

As discussed above, we can formulate optimization over posynomial functions as a convex optimization problem. We began thinking about whether the problem being solved by an Input Convex Neural Network could be recast as a geometric programming problem by reformulating it in terms of posynomial functions. Our first result is on reformulating the function instantiated by an Input Convex Network as a generalized posynomial function of a re-parameterized version of $\mathbf{y}$. This derivation is shown for a *Fully* Input Convex Network (where we have $f(\mathbf{y}; \theta)$) but the result holds for the more general case of *Partially* Input Convex Networks (which are $f(\mathbf{x}, \mathbf{y}; \theta)$ and what we have been using for maximum generality throughout most of the rest of this paper). We started by examining equation (12) in (Amos et al., 2016) which defines the prediction problem in ICNNs.

$$\min_{\mathbf{y}} \mathbf{z}^{(K)} \;:\; \mathbf{z}^{(k+1)} = g(\mathbf{W}^{z(k)}\mathbf{z}^{(k)} + \mathbf{W}^{y(k)}\mathbf{y} + \mathbf{b}^{(k)}), \quad k = 0, \dots K-1 \tag{12}$$

The constraints are the recursive definition for the activation $\mathbf{z}^k$ of each neural network layer in terms of the layer before it, and thus give the formula for $f(\mathbf{y}; \theta) = \mathbf{z}^{(K)}$.

To begin, we make two small but crucial changes to the original formulation:

1. We replace the rectified linear unit (relu) function with $g(v) = \log(1 + e^v)$, the smooth relaxation of the relu. This is still a convex non-decreasing function, but is now differentiable everywhere.
2. We make a variable substitution for the variable $\mathbf{y}$, (the input we want to minimize over) specifically we take instead $\tilde{\mathbf{y}} = e^{\mathbf{y}}$

The affect of the first change is that the optimization problem is now

$$\min_{\mathbf{y}} \mathbf{z}^{(K)} \;:\; \mathbf{z}^{(k+1)} = \log(1 + e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)} + \mathbf{W}^{y(k)}\mathbf{y} + \mathbf{b}^{(k)}})$$

Where we are applying the exponential function elementwise to the vector $\mathbf{W}^{z(k)}\mathbf{z}^{(k)} + \mathbf{W}^{y(k)}\mathbf{y} + \mathbf{b}^{(k)}$. $\mathbf{z}^{(k+1)}$ is of dimension m, and $\mathbf{y}$ is of dimension n so $\mathbf{W}^{z(k)}$ is $m \times m$ and $\mathbf{W}^{y(k)}$ is $m \times n$. Looking at each element of $\mathbf{z}$, in effect we have m equalities of the form

$$z_i^{(k+1)} = \left(\log(1 + e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)} + \mathbf{W}^{y(k)}\mathbf{y} + \mathbf{b}^{(k)}})\right)_i \quad \forall\, i = 0, \dots, m-1 \tag{13}$$

We can manipulate the constraints like so:

$$\begin{aligned} &\log(1 + e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)} + \mathbf{W}^{y(k)}\mathbf{y} + \mathbf{b}^{(k)}}) \\ =&\log(1 + e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)}} \circ e^{\mathbf{W}^{y(k)}\mathbf{y}} \circ e^{\mathbf{b}^{(k)}}) \end{aligned}$$

Where $\circ$ represents the hadamard (elementwise) product of the vectors. Now, we define the vector $\mathbf{d}^{(k)} := e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)}}$. Notice that $\mathbf{d}^{(k)}$ is nonnegative. We also define $\mathbf{c}^{(k)} := e^{\mathbf{b}^{(k)}}$

Let us now examine the quantity $e^{\mathbf{W}^{y(k)}\mathbf{y}}$. We will consider this for any k so that we can drop the superscript on $\mathbf{W}^{y(k)}$:

$$\begin{aligned} &e^{\mathbf{W}\mathbf{y}} \\ =&e^{y_1\mathbf{w}_1 + y_2\mathbf{w}_2 + \dots + y_n\mathbf{w}_n} \\ =&e^{y_1\mathbf{w}_1} \circ e^{y_2\mathbf{w}_2} \circ \dots \circ e^{y_n\mathbf{w}_n} \\ =&\begin{bmatrix} (e^{y_1})^{w_{11}}(e^{y_2})^{w_{12}} \dots (e^{y_n})^{w_{1n}} \\ (e^{y_1})^{w_{21}}(e^{y_2})^{w_{22}} \dots (e^{y_n})^{w_{2n}} \\ \vdots \\ (e^{y_1})^{w_{m1}}(e^{y_2})^{w_{m2}} \dots (e^{y_n})^{w_{mn}} \end{bmatrix} \\ =&\begin{bmatrix} \tilde{y}_1^{w_{11}}\tilde{y}_2^{w_{12}} \dots \tilde{y}_n^{w_{1n}} \\ \tilde{y}_1^{w_{21}}\tilde{y}_2^{w_{22}} \dots \tilde{y}_n^{w_{2n}} \\ \vdots \\ \tilde{y}_1^{w_{m1}}\tilde{y}_2^{w_{m2}} \dots \tilde{y}_n^{w_{mn}} \end{bmatrix} \end{aligned} \tag{14}$$

Notice that each element in this vector is a monomial function of $\tilde{\mathbf{y}}$. This allows us to rewrite each of the equalities in (13) as

$$z_i^{(k+1)} = \log(1 + d_i^{(k)} c_i^{(k)} (\tilde{y}_1^{w_{i1}^{y(k)}} \tilde{y}_2^{w_{i2}^{y(k)}} \dots \tilde{y}_n^{w_{in}^{y(k)}})) \quad \forall\, i = 0, \dots, m-1 \tag{15}$$

Taking $\tilde{z}_i^{(k+1)} = e^{z_i^{(k+1)}}$ analogously to how we constructed $\tilde{\mathbf{y}}$ allows us to get a bit closer to a posynomial:

$$\tilde{z}_i^{(k+1)} = 1 + d_i^{(k)} c_i^{(k)} (\tilde{y}_1^{w_{i1}^{y(k)}} \tilde{y}_2^{w_{i2}^{y(k)}} \dots \tilde{y}_n^{w_{in}^{y(k)}}) \quad \forall\, i = 0, \dots, m-1 \tag{16}$$

Because the last term of this expression is a monomial function of $\tilde{\mathbf{y}}$ and $c_i^{(k)}$ is a positive constant, if $d_i^{(k)}$ is a generalized posynomial function of $\tilde{\mathbf{y}}$ for all i and k, then $\tilde{z}_i^{(k)}$ is a generalized posynomial function of $\tilde{\mathbf{y}}$ for all i and k.

Proposition 1. *$d_i^{(k)}$ is a generalized posynomial of $\tilde{\mathbf{y}}$ for all i and k*

Proof.

$$\begin{aligned} \mathbf{d}^{(k)} &= e^{\mathbf{W}^{z(k)}\mathbf{z}^{(k)}} \\ &= \begin{bmatrix} \tilde{z}_1^{(k)\,w_{11}^{z(k)}} \tilde{z}_2^{(k)\,w_{12}^{z(k)}} \dots \tilde{z}_m^{(k)\,w_{1m}^{z(k)}} \\ \tilde{z}_1^{(k)\,w_{21}^{z(k)}} \tilde{z}_2^{(k)\,w_{22}^{z(k)}} \dots \tilde{z}_m^{(k)\,w_{2m}^{z(k)}} \\ \vdots \\ \tilde{z}_1^{(k)\,w_{m1}^{z(k)}} \tilde{z}_2^{(k)\,w_{m2}^{z(k)}} \dots \tilde{z}_m^{(k)\,w_{mm}^{z(k)}} \end{bmatrix} \end{aligned} \tag{17}$$

Via the same algebra as we did in (14). $\mathbf{z}^{(1)} = \log(1 + e^{\mathbf{W}^{y(0)}\mathbf{y}})$, thus $\tilde{\mathbf{z}}^{(1)} = 1 + e^{\mathbf{W}^{y(0)}\mathbf{y}}$ which means that each element $\tilde{z}_i^{(1)}$ is a posynomial of $\tilde{\mathbf{y}}$ due to (14). Looking at $d_i^{(1)}$, we make the following observations,

1. $d_i^{(1)}$ takes each of the posynomials $\tilde{z}_i^{(1)}$, raises it to a positive power, and multiplies them together. The positivity of the power is due to the fact the matrices $\mathbf{W}^{z(k)}$ are strictly positive by construction, which would not be true of traditional neural networks. This means that $d_i^{(1)}$ is a *generalized* posynomial function of $\tilde{\mathbf{y}}$.
2. Equation (16) tells us that $\tilde{z}_i^{(2)}$ is 1 plus a generalized posynomial ($d_i^{(1)}$), times a positive constant ($c_i^{(k)}$), times a monomial ($\tilde{y}_1^{w_{i1}^{y(k)}} \tilde{y}_2^{w_{i2}^{y(k)}} \dots \tilde{y}_n^{w_{in}^{y(k)}}$), which means that $\tilde{z}_i^{(2)}$ is a generalized posynomial.

Observation 1) and 2) together imply via induction that each of the $d_i^{(k)}$ is a generalized posynomial for $k = 1 \dots K-1$. □

Thus, we have shown that by making two small changes to the Input Convex Neural Network architecture, it is possible to reformulate the function instantiated by the network as a generalized posynomial. This means that the prediction step in ICNNs, which involves optimizing this function, can be solved with efficient geometric programming algorithms. We leave it to future work to explore the practical performance on real datasets of such an approach compared to the bundle entropy method employed in (Amos et al., 2016).

4.3. A Neural Network Approach to Learning Posynomials

Having shown the link between prediction in ICNNs and Geometric Programming, we consider the task of fitting posynomials to data. We first noticed that a one layer ICNN, barring some post-processing, encoded a monomial of its inputs in each neuron. This led us to believe that we could leverage the wealth of computational work in the field of deep learning to fit posynomial models to data. An added benefit of this was the fact that the learned posynomial model could be directly read out from the weights of the network. This addresses one of the main criticisms of deep learning, namely the fact that neural networks are often used as black boxes for function approximation with little to no intuition on what each neuron contributes to the output. We were able to benchmark this approach against the work described in (Calafiore et al., 2014), which approached fitting posynomials to data from a convex optimization perspective using square-root LASSO on a large basis set of monomials.

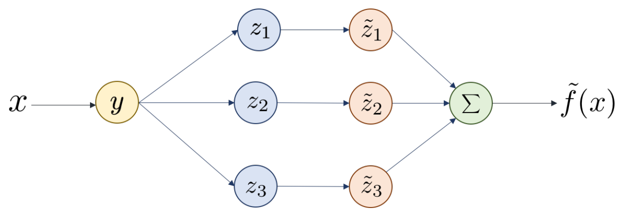

Figure 14. Neural network architecture for fitting a posynomial made of of three monomial terms

Our problem is as follows: Given data $\{(\mathbf{x}, f(\mathbf{x}))\}$ we would like to approximate $f(\mathbf{x})$ by a posynomial function $\tilde{f}(\mathbf{x})$ by training a one hidden layer neural network. Our motivating idea was the fact that barring post-processing, a softplus activation function encodes a posynomial of the log of its inputs. We first show that a soft-plus neural network can fit a posynomial. We then discuss training the neural network, and how using a regularizing term in our objective can be used to discover the number of monomials in the posynomial. We then show some preliminary numerical results that suggest that this method for fitting posynomials is fast and accurate.

4.3.1. Posynomial encoding through softplus activation functions

To show that a neural network with one softplus hidden layer and some pre- and post-processing can encode a posynomial function of our inputs $\mathbf{x}$, we proceed as follows:

1. Pre-process input $\mathbf{x} \in \mathbb{R}^n$ into neural network input $\mathbf{y} \in \mathbb{R}^n$

$$\mathbf{y} = log(\mathbf{x})$$

   Where $log(\mathbf{x})$ is the element-wise log of $\mathbf{x}$.

2. Pass $\mathbf{y}$ through the softplus hidden layer of width M. Output of each neuron i is denoted as z_i for $i = 1, ..., M$. Each neuron has a set of associated weights $\mathbf{w}_i$ and a bias term b_i.

$$z_i = log(1 + e^{w_i^\top y + b_i})$$

3. Exponentiate the output of each neuron in the hidden layer, the output of this layer is denoted $\tilde{z}_i$ for $i = 1, ...M$.

$$\tilde{z}_i = e^{z_i}$$

   We can now see that $\tilde{z}_i$ encodes a monomial of $\mathbf{x}$ with an affine term. We let $\tilde{b}_i := e^{b_i}$ be the coefficient of the monomial.

$$\begin{aligned}\tilde{z}_i &= 1 + e^{w_i^\top y + b_i} \\ &= 1 + \tilde{b}_i \prod_j^n x_j^{w_{ji}}\end{aligned}$$

4. Pass all $\tilde{z}_i$ through a linear activation function with a bias of $-M$. This output is the posynomial approximation to $f(x)$, denoted $\tilde{f}(x)$.

$$\tilde{f}(x) = \sum_i^M \tilde{z}_i - M \quad (18)$$

$$= \sum_i^M \tilde{b}_i \prod_j^n x_j^{w_{ji}} \quad (19)$$

   We note that the weights of the hidden layer are the powers of each monomial, and that the bias of each layer encodes the log of coefficient of the monomial. Further, since the bias term encodes the log of the coefficient, the coefficient is constrained to be strictly positive, as desired from our definition of posynomials.

The architecture of this neural network is illustrated in Figure 14. We note that we could use a hidden layer with an exponential activation function instead of a softplus, removing the need for the $\tilde{z}$ layer and retaining the ease of interpretability. However, we found that that exponential activation function performs worse numerically because it suffers from vanishing and exploding gradients, making it difficult to train.

4.3.2. Fitting Posynomials

Now that we have shown that a neural network with one softplus hidden layer can encode a posynomial, we now discuss the actual fitting of a posynomial to data. In general, what we would like to do when training is to solve the optimization problem:

$$\min_{\mathbf{w}_1, \ldots \mathbf{w}_M, b_1, \ldots b_M} L(\tilde{f}(x), f(x))$$

Subject to the neural network architecture constraints, where $L(\cdot, \cdot)$ is a loss function of our choice. However, because of the neural network architecture we can actually over-paramatrize our desired posynomial by making M (the number of monomials in our posynomial) very large, and then introduce a sparsity-promoting term in our objective to only fit monomials that have large contributions the the final output. Thus our optimization becomes:

$$\min_{\mathbf{w}_1, \ldots \mathbf{w}_M, b_1, \ldots b_M} L(\tilde{f}(x), f(x)) + \lambda_1 \|W\|_1 + \lambda_2 \|\tilde{b}\|_1$$

Where W is the $n \times M$ matrix of weight vectors and $\tilde{b}$ is a vector of the biases in the hidden layer. Though this objective is non-convex in W and b, the neural network form allows us to use the tensorflow library and gradient based optimizers that have been tailored for training deep neural networks to perform a fast and efficient gradient descent on this objective. This results in a neural network that represents a posynomial approximation to our data. Further, the LASSO approach helps us identify the number of monomials in our posynomial.

4.3.3. Numerical Results

To analyze the accuracy and efficiency of fitting a posynomial with a neural network, we benchmarked our method on Example 1 from (Calafiore et al., 2014). We recreated the exact experiment described. The task was to fit the posynomial:

$$f(x) = x_2^{1.5}x_3^3 + 2x_1^2x_3^{-1} + 3x_2^{3.2} + 4x_1^{0.5}x_2^{-2}x_3$$

Given 600 data samples, $\{x, g(x)\}_{i=1}^{600}$ where $g(x_i) = f(x_i) + \epsilon_i$. ϵ_i is 0 mean noise with a noise-to-signal standard deviation ratio of 1%, we fit our data with a mean square log loss function and the regularization parameters $\lambda_1 = \lambda_2 = 0.001$. We analyze the results

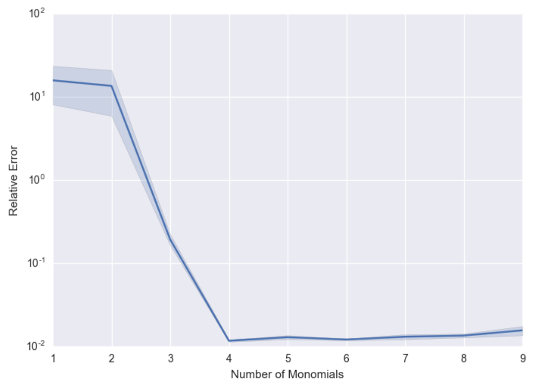

Figure 15. Average Relative Error after 1400 epochs of training for neural networks encoding posynomials with different numbers of monomials

of our neural networks through the relative error metric, calculated here by comparing the vector of predicted values Y_{pred} from 600 new data points to their actual values, again corrupted with noise as was done during training, and denoted Y_{noised}. This matches the analysis done in (Calafiore et al., 2014). The relative error is calculated as:

$$\text{relative error} = \frac{\|Y_{pred} - Y_{noised}\|_2}{\|Y_{noised}\|_2}$$

Figure 15 shows the relative error after 1400 epochs of training for training a network with widths in the range $[1, 9]$, corresponding to 1 monomial to a posynomial made up of 9 monomials. We can clearly see that the regularization terms in the training objective help us pick out the correct number of monomials to fit. We can see that the relative error achieves its minimum value with 4 monomials. As we add more monomials after this point, the error increases slightly. This occurs because the coefficient of an individual monomial cannot be set to 0, because it is encoded by an exponential function which can only take values in $\mathbb{R}_{++}$. However, be regularizing the weights and biases a monomial can be brought arbitrarily close to 0 as training progresses, giving us the desired posynomial function in as few monomials as possible.

Having identified the correct number of monomials, we can now analyze the learned posynomial. We fit a neural network with 4 monomial units to 600 noised data points. After 17.5 seconds, the relative error had converged to 0.010 and the learned posynomial was:

$$\tilde{f}(x) = 1.03x_2^{1.5}x_3^3 + 2.07x_1^2x_3^{-1} + 2.98x_2^{3.2} + 3.94x_1^{0.5}x_2^{-2}x_3$$

As we can see, the neural network learned the exponents correctly with slight errors in the coefficients. The qualitative results are shown in Figure 16, where we have plotted the true posynomial function with noise (blue dots) and our neural network approximation (red lines) for the two regions displayed in (Calafiore et al., 2014).

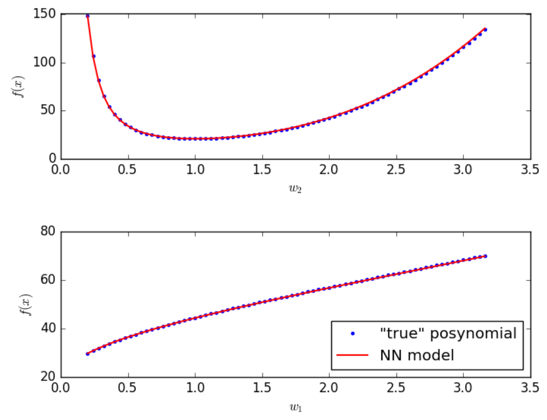

Figure 16. Qualitative results of $\tilde{f}(x)$ vs $f(x)$. Relative Error of 0.010. Top: The true posynomial vs the fitted posynomial for $x_1 = 2.3$, $x_3 = 0.9$ and $x_2 \in [0.2, 3.2]$. Bottom: The true posynomial vs the fitted posynomial for $x_1 \in [0.2, 3.2]$,$x_2 = 0.7$, $x_3 = 3.1$

.

These results are comparable to the results from (Calafiore et al., 2014), where they were able to find the posynomial up to a relative error of 0.011. However, the time taken in (Calafiore et al., 2014) was 91 seconds, as compared with the 17.5 seconds for the neural network method.

Along with the small numerical error of not being able to set monomials to 0 if they are not needed, fitting the posynomial with a neural network still has some limitations. First, we are only guaranteed to find a local minimizer of our objective using gradient descent since our objective is non-convex. Further, it can be more sensitive to the values of the hyper-parameters like the learning rate, regularization parameters, and loss functions. However, from this initial result, fitting posynomials with neural networks seems like a promising approach to the problem of finding expressive yet relatively easy to optimize models for complex systems.

5. Conclusion

In summary, studying the connections between posynomials and neural networks has yielded insights regarding both models, though much remains to be explored. ICNNs are a promising step in the direction of developing models that retain the expressive power of neural networks but also leverage the advantages of convex optimization. Our reformulation of the network function as a posynomial may yield a better inference algorithm based on geometric programming. Further, our proposed maximum entropy method may yield a more effective max-margin training algorithm than the proposed gradient descent based algorithm. In practice, we found ICNNs difficult to train and highly sensitive to hyper-parameters. Finally, we've shown that deep learning machinery is applicable to learning posynomial models, with preliminary results showing fast and accurate fitting of posynomial functions in the presence of noise.

Moving forwards, we believe that ICNNs can be useful in training robots with unknown dynamics to follow trajectories. We also believe that it may be possible to train deep neural networks to learn *generalized* posynomial models from data, and yet retain the ease of interpretability in the same way we were able to learn posynomial models using neural networks.

References

Abadi, Martın, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Amos, Brandon, Xu, Lei, and Kolter, J. Zico. Input convex neural networks. *arXiv preprint arXiv:1609.07152*, 2016.

Bansal, Somil, Akametalu, Anayo K, Jiang, Frank J, Laine, Forrest, and Tomlin, Claire J. Learning quadrotor dynamics using neural network for flight control. *arXiv preprint arXiv:1610.05863*, 2016.

Boyd, Stephen and Vandenberghe, Lieven. *Convex Optimization*, volume 25. 2010. ISBN 9780521833783. doi: 10.1080/10556781003625177. URL `https://web.stanford.edu/{~}boyd/cvxbook/bv{_}cvxbook.pdf`.

Calafiore, Giuseppe C, Ghaoui, Laurent El, and Novara, Carlo. Sparse Identification of Posynomial Models. 2014.

Collobert, Ronan and Weston, Jason. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 160–167. ACM, 2008.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E, Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

Hornik, Kurt, Stinchcombe, Maxwell, and White, Halbert. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint*, arXiv:1412, 2014.

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

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Ratliff, Nathan D., Bagnell, J. Andrew, and Zinkevich, Martin A. (Online) Subgradient Methods for Structured Prediction. *Journal of Machine Learning Research*, 2007.

Taskar, Ben. *Learning Structured Prediction Models: A Largin Approach*. PhD thesis, Stanford University, 2004.

Zarrouk, David, Haldane, Duncan W, and Fearing, Ronald S. Dynamic legged locomotion for palm-size robots. In *SPIE Defense+ Security*, pp. 94671S–94671S. International Society for Optics and Photonics, 2015.