# An Introduction to an Application of the Implicit Function Theorem

Justin Chiu
Cornell Tech
jtc257@cornell.edu

July 28, 2021

**Abstract**

Gradient-based learning forms the foundation of modern machine learning, and automatic differentiation allows ML practitioners to easily compute gradients. While automatic differentiation only costs a constant multiple of the time and space required to evaluate a function, it has its limitations. In particular, when evaluating a function itself is expensive, the direct application of automatic differentiation is infeasible. In this report, we review the implicit function theorem (IFT) and its use in reducing the cost of computing gradients in scenarios where function evaluation is expensive, focusing on the application of implicit differentiation to variational inference.

## 1 Introduction

Gradient-based learning underpins many of the recent successes in machine learning, particularly advances involving neural networks. The key to the success of gradient-based methods is automatic differentiation (AD), which has greatly increased the development speed of machine learning research by allowing practitioners to circumvent the error-prone and time-consuming process of computing gradients manually. AD operates by reducing functions into compositions of atomic operations, for which we have a library of derivatives for, and composing those derivatives via the chain rule. (introduce the representation of functions as evaluation procedures / computational graphs, following Griewank and Walther [2008]) While efficient relative to the evaluation of the function in question, taking only a multiplicative constant longer than the evaluation itself, this may be prohibitively expensive if the original function evaluation itself is costly. An example of this is if the function takes the form of an unrolled loop, a common artifact of iterative methods. As naive AD requires storing all of the intermediate values at each point, storing the output of all computations at every iteration of a loop can quickly become infeasible due to memory limitations.

There are a variety of methods for overcoming the space limitations of AD, of which we only mention three: checkpointing, reversible computation, and implicit differentiation. A first method, checkpointing, improves space complexity at the cost of time. Rather than storing all intermediate computation, checkpointing instead recomputes values when needed. This can result in a large

slowdown, and also requires careful choosing of which computationals subgraphs to checkpoint. A second method is an improvement upon checkpointing, called reversible computation [Maclaurin et al., 2015, Gomez et al., 2017], which improves space complexity at the cost of expressivity, but not speed. Reversible computation ensures that the gradient with respect to (wrt) input depends only on the output, allowing the input to be discarded during function evaluation. This is typically accomplished by ensuring that the input is easily reconstructed from the output, restricting the expressivity of layers. A third method is implicit differentiation, which potentially improves space complexity at the cost of stronger assumptions. Implicit differentiation relies on the implicit function theorem (IFT), which gives conditions under which derivatives can be computed independent of intermediate computation. The primary condition is the characterization of the output as the solution to a series of equations.

In this report, we will cover the use of the implicit function theorem in OptNet [Amos and Kolter, 2017], which allows us to use the output of an optimization problem inside a neural network. Explain OptNet

**Bilevel Optimization** One application of implicit differentiation is bilevel optimization. Bilevel optimization problems are, as implied by the name, optimization problems with another nested inner optimization problem embedded within. Methods for solving bilevel optimization typically proceed iteratively. For every iteration when solving the outer optimization problem, we must additionally solve an inner optimization problem.

The application we focus on in this report is expressing individual layers of a neural network declaratively as the solution of an optimization problem [Amos and Kolter, 2017, Agrawal et al., 2019, Gould et al., 2019]. This allows models to learn, without heavy manual specification, the constraints of the problem in addition to the parameters of the objective. We will discuss this in further detail later on in the report, when we go over OptNet [Amos and Kolter, 2017].

Other applications that can be formulated as bilevel optimization problems are hyperparameter optimization, metalearning, and variational inference. Hyperparameter optimization formulates hyperparameter tuning, such as the shrinkage penalty in Lasso, as a bilevel optimization problem by computing gradients wrt the penalty through the entire learning procedure of the linear model [Lorraine et al., 2019b]. (Other works on hyperparam opt [Maclaurin et al., 2015, Bertrand et al., 2020]) Similarly, metalearning learns the parameters of a model such that the model is able to quickly be adapted to a new task via gradient descent [Finn et al., 2017, Rajeswaran et al., 2019]. This is accomplished by differentiating through the learning procedure of each new task. Finally, a variant of variational inference follows a very similar format: semi-amortized variational inference (SAVI) aims to learn a model that is able to provide a good initialization for variational parameters that are subsequently updated to maximize a lower bound objective [Kim et al., 2018]. This is also accomplished by differentiating through the iterative optimization procedure applied to the variational parameters during inference. (Other VI papers [Wainwright and Jordan, 2008, Johnson et al., 2017])

finish loop, how does IFT help with these applications


(a) (b)

Figure 1: Illustration (a) contains circle, defined by the relation $a^2 + b^2 - 1 = 0$. While we cannot differentiate the relation directly, we can compute derivatives at the point $a_1$ using the local parameterization $f_1(a) = \sqrt{1 - b^2}$ that holds in a neighbourhood around $a_1$. We cannot use the same parameterization at $a_2$ as the derivative is undefined. Separately, illustration (b) shows an example relationship between the parameters $\theta$, solution $x$, and implicit function $x^*(\theta)$ of the IFT. The rectangle depicts a space which, in this example, contains both $\theta$ and $x$. This is not necessary for the IFT, but simplifies illustration. The parameters $\theta$ provide an initial point, which is then iteratively refined into solution $x$, shown by the curved line. If $x$ satisfies the conditions of the IFT, then the IFT guarantees the existence of the implicit solution mapping $x^*(\theta)$, and tells us how to compute $\frac{dx^*(\theta)}{d\theta}$. This is useful if the iterative procedure (squiggly line), is too expensive to store in memory for use in automatic differentiation.

# 2 The Implicit Function Theorem

The implicit function theorem (IFT) has a long history, as well as many applications in a wide variety of fields such as economics and differential geometry. For an overview of the history of the IFT and some of its classical applications in mathematics and economics, see the book by Krantz and Parks [2003].

## 2.1 What is the IFT?

Consider the unit circle, governed by the relation $F(a, b) = a^2 + b^2 - 1 = 0$, which can be interpreted as a system of equations. As $F$ fails the vertical line test, we cannot write $b$ as a function of $a$ globally. This prevents us from taking derivatives, for example $\frac{db}{da}$. However, we can use local parameterizations: $f_1(a) = \sqrt{1 - b^2}$ if $b > 0$ or $f_2(a) = -\sqrt{1 - b^2}$ if $b < 0$. These local parameterizations then allow us to take derivatives $\frac{db}{da}$ at particular points $(a, b)$ using the corresponding parameterization. See Fig. 1 for an illustration. The IFT generalizes this example, and formalizes the conditions under which there exist smooth local parameterizations for a given relation or system of equations. Unlike in this example, the IFT does not give those local parameterizations. It only gives the derivatives, leaving the local functions implicit, hence the 'implicit' in IFT.

Formally, the IFT gives sufficient conditions under which the solution $x \in \mathbb{R}^m$ to a system of equations, $F(\theta, x) = 0$ with $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$, can locally be written as a function of just the parameters $\theta \in \mathbb{R}^n$, i.e. there exists a solution mapping $x^* : \mathbb{R}^n \to \mathbb{R}^m$ in the neighbourhood of the

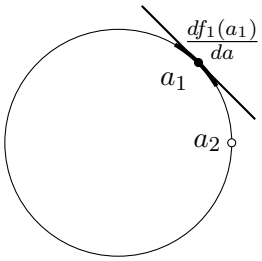


Figure 2: A circle, defined by the relation $a^2 + b^2 - 1 = 0$. While we cannot differentiate the relation directly, we can compute derivatives at the point $a_1$ using the local parameterization $f_1(a) = \sqrt{1 - b^2}$ that holds in a neighbourhood around $a_1$. We cannot use the same parameterization at $a_2$ as the derivative is undefined.

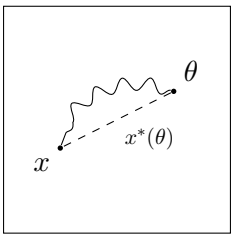


Figure 3: An example relationship between the parameters $\theta$, solution $x$, and implicit function $x^*(\theta)$ of the IFT. The rectangle depicts a space which, in this example, contains both $\theta$ and $x$. This is not necessary for the IFT, but simplifies illustration. The parameters $\theta$ provide an initial point, which is then iteratively refined into solution $x$, shown by the curved line. If $x$ satisfies the conditions of the IFT, then the IFT guarantees the existence of the implicit solution mapping $x^*(\theta)$, and tells us how to compute $\frac{dx^*(\theta)}{d\theta}$. This is useful if the iterative procedure (squiggly line), is too expensive to store in memory for use in automatic differentiation.

particular point $(\theta, x) \in \mathrm{dom}(F)$. These conditions are as follows:

1. We have a solution point $(\theta, x)$ that satisfies the system of equations $F(\theta, x) = 0$.
2. $F$ has at least continuous first derivatives: $F \in \mathcal{C}^k$.
3. The Jacobian of $F$ wrt $x$ evaluated at the solution point $(\theta, x)$ is nonsingular: $\det \frac{dF}{dx} \neq 0$.

Given these conditions hold for $F, \theta$, and $x$, we are then able to assert the existence of the implicit solution mapping $x^*(\theta)$, and determine its derivative $\frac{dx^*(\theta)}{d\theta} = -[\frac{dF(\theta,x)}{dx}]^{-1} \frac{dF(\theta,x)}{d\theta}$. We provide an illustration in Fig. 1.

## 2.2 Why is the IFT useful?

As mentioned in Sec. 1, the IFT is useful for saving memory when performing automatic differentiation. The main benefit of the IFT is that once we have a solution to a system of equations, we can forget about how we obtained that solution but still compute derivatives.

As an example, say we obtained a solution using gradient descent, which from an initial point $\theta = x_0$ produces $x_1, x_2, ..., x_{1000}$ if you run it for 1000 iterations. Without the IFT, you would have to store all 1000 $x$'s and the computation used to produce them. The IFT lets you throw away $x_1$ through $x_{999}$, and gives you the derivative $\frac{dx_{1000}}{dx_0}$.

In particular, we will show how to compute derivatives of the solution of an optimization problem wrt parameters of the problem without storing intermediate computations. We will use the optimality criteria of the optimization problem to define a system of equations, then apply the IFT to compute the Jacobian of the solution wrt the parameters. This methodology allows us to use the solution to an optimization problem as the output of a layer within a neural network, as done in OptNet [Amos and Kolter, 2017].

Afterwards, we will cover an application of the IFT to speeding up variational inference.

# 3 Embedding Optimization inside a Neural Network

As an introductory example, we will replace the softmax layer of a neural network with an equivalent function defined as the output of an optimization problem, then derive derivatives using the IFT. We will start by reviewing softmax and its expression as an optimization problem. After checking the conditions of the IFT hold, we can then compute derivatives. Since the Jacobian of softmax is known, we can directly verify that the IFT gives the correct answer.

## 3.1 The Chain Rule

Before we get into the details of softmax, we first setup the learning problem. We are interested in training a neural network model that consists of stacks of layers, all of which take the output of the previous layer as input. In order to train such a model we rely on backpropagation, also known as reverse mode AD. Reverse mode AD propagates error information backward through the network via the chain rule for computing derivatives of composite functions. Key to this process is the Jacobian matrix of each layer, the matrix of partial derivatives of the layer's outputs wrt its inputs.

As an example, let's say we would like to maximize $b = f_3(f_2(f_1(a)))$ with a gradient-based method, where $b \in \mathbb{R}$; $f_3 : \mathbb{R}^n \to \mathbb{R}$; $f_1, f_2 : \mathbb{R}^n \to \mathbb{R}^n$; and $a \in \mathbb{R}^n$. We can compute the derivatives of $b$ wrt each layer via the chain rule. Focusing on the first layer $f_1$, we have $\frac{db}{df_1} = \frac{db}{df_3}\frac{df_3}{df_2}\frac{df_2}{df_1}$, where each term is the Jacobian matrix for the respective layer in the denominator. These derivatives can then be used to update the parameters of each layer, completing one iteration of the learning process.

Given its key role in learning, we will proceed towards computing the Jacobian of softmax using the IFT by first reviewing softmax.

## 3.2 Softmax

Softmax is often used to parameterize categorical distributions within neural networks, such as in attention layers. It has its origins in statistical mechanics and decision theory, and functions as a differentiable surrogate for argmax.

Softmax assumes that we have $n$ items with independent utilities, $\theta \in \mathbb{R}^n$, which indicate preferences. Softmax then gives the following distribution over items: $x_i = \frac{\exp(\theta_i)}{\sum_j \exp(\theta_j)}$, with $x \in \mathbb{R}^n$. Interestingly, softmax arises as the solution of an optimization problem [Gao and Pavel, 2018].

The output of softmax is the solution of the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & x^\top \theta + H(x) \\ \text{subject to} \quad & x^\top \mathbf{1} = 1 \\ & x \succeq 0, \end{aligned} \tag{1}$$

where $H(x) = -\sum_i x_i \log x_i$ is the entropy. This corresponds to an entropy-regularized argmax optimization problem. We will refer to this as the softmax problem.

Our goal is to compute the Jacobian of softmax $\frac{dx}{d\theta} = \frac{d\text{softmax}(\theta)}{d\theta}$ using the IFT and the optimization problem above. This would allow us to compute gradients via autodiff inside a neural network via the chain rule, although we will stop at computing the Jacobian. While this is not of practical use (there is a closed-form equation for both softmax and its Jacobian), we use it as an introduction to the mechanism behind OptNet (and other differentiable optimization layers) [Amos and Kolter, 2017, Agrawal et al., 2019]. Applying the IFT to optimization problems consists of four steps:

1. Find a solution to the optimization problem.
2. Write down the system of equations.
3. Check that the conditions of the IFT hold.
4. Compute the derivative of the implicit solution mapping wrt the parameters.

We assume the first step has been done for us, and we have some solution $x$ to the softmax problem. We will then use the IFT to compute gradients of $x$ wrt the parameters $\theta$ by following the rest of the steps.

### 3.3 The KKT conditions determine $F$

Given an optimization problem, the KKT conditions determine a system of equations that the solution must satisfy [Karush, 1939, Kuhn and Tucker, 1951]. We will use the KKT conditions of the softmax problem in Eqn. 1 to determine the function $F$ in the IFT.

First, we introduce dual variables $u \in \mathbb{R}, v \in \mathbb{R}^n$ and write out the Lagrangian:

$$\mathcal{L}(\theta, x, u, v) = x^\top \theta + H(x) + u(x^\top \mathbf{1} - 1) + v^\top x.$$

We include $u, v$ as solution variables. We then have the following necessary conditions for a solution $(x, u, v)$, i.e. the KKT conditions:

$$\begin{aligned} \nabla_x \mathcal{L}(\theta, x, u, v) &= 0 && \text{(stationarity)} \\ u(x^\top \mathbf{1} - 1) &= 0 && \text{(primal feasibility)} \\ \text{diag}(v)x &= 0 && \text{(complementary slackness)} \\ v &\succeq 0 && \text{(dual feasibility)} \end{aligned} \tag{2}$$

As we are interested in the primal variable or solution $x$, we focus on the first three conditions.

In full, the system of equations $F(\theta, x, u, v) = 0$ is

$$\begin{aligned} \theta + -\log(x) - 1 + u\mathbf{1} + v &= 0 \\ u(x^\top \mathbf{1} - 1) &= 0 \\ \operatorname{diag}(v)x &= 0. \end{aligned} \tag{3}$$

Now we can check the conditions of the IFT. Any solution $(x, u, v)$ will satisfy $F(\theta, x, u, v) = 0$, and $F \in \mathcal{C}^1$. All that remains is to check that the Jacobian matrix of $F$ is non-singular.

The Jacobian $\frac{dF}{d(x,u,v)} \in \mathbb{R}^{n \times n+1+n}$ is given by

$$\frac{dF}{d(x,u,v)} = \begin{bmatrix} \operatorname{diag}(x)^{-1} & -\mathbf{1} & -I_n \\ u\mathbf{1}^\top & x^\top \mathbf{1} - 1 & 0 \\ \operatorname{diag}(v) & 0 & \operatorname{diag}(x) \end{bmatrix}. \tag{4}$$

Since a solution must be feasible, we know that $x^\top \mathbf{1} = 1$ and $u > 0$. With the additional information that the domain of $H(x)$ adds the implicit constraint that $\forall i, x_i > 0$, we can deduce that the Jacobian of $F$ is full rank and therefore has nonzero determinant. This shows that the conditions of the IFT hold.

### 3.4 The Jacobian of Softmax

Now that we have shown that the conditions of the IFT hold, we can proceed to apply the second part of the IFT in order to compute $\frac{dx}{d\theta}$. The second part of the IFT tells us that we can compute the Jacobian of the solution mapping $\frac{d(x,u,v)}{d\theta} = \frac{dx^*(\theta)}{d\theta} = \left[\frac{dF}{d(x,u,v)}\right]^{-1} \frac{dF}{d\theta}$, then pick out the relevant components.

The second term, $\frac{dF}{d\theta}$, is simple. Since $\theta$ only appears in the first vector-valued function of $F$ (see Eqn. 3), we have

$$\frac{dF}{d\theta} = \begin{bmatrix} I_n \\ 0 \end{bmatrix}. \tag{5}$$

The large amount of sparsity allows us to skip some computation further down.

Next, we have to invert the Jacobian from Eqn. 4:

$$\left[\frac{dF}{d(x,u,v)}\right]^{-1} = \begin{bmatrix} \operatorname{diag}(x)^{-1} & -\mathbf{1} & -I_n \\ u\mathbf{1}^\top & x^\top \mathbf{1} - 1 & 0 \\ \operatorname{diag}(v) & 0 & \operatorname{diag}(x) \end{bmatrix}^{-1}. \tag{6}$$

We use the block-wise inversion formula

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} (A - BD^{-1}C)^{-1} & 0 \\ 0 & (D - CA^{-1}B)^{-1} \end{bmatrix} \begin{bmatrix} I & -BD^{-1} \\ -CA^{-1} & I \end{bmatrix},$$

where

$$A = \begin{bmatrix} \text{diag}(x)^{-1} & -\mathbf{1} \\ u\mathbf{1}^\top & 0 \end{bmatrix} \qquad B = \begin{bmatrix} -I_n \\ 0 \end{bmatrix}$$
$$C = \begin{bmatrix} \text{diag}(v) & 0 \end{bmatrix} \qquad D = \text{diag}(x).$$

However, by complementary slackness, we have $v = 0$, reducing the above to

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & -BD^{-1} \\ 0 & I \end{bmatrix}.$$

As we are interested in computing $\frac{dx}{d\theta}$, rather than the full derivative $\frac{d(x,u,v)}{d\theta}$, in addition to the sparsity of $\frac{dF}{d\theta}$, we only have to solve for the upper-left $n \times n$ block of $A^{-1} \in \mathbb{R}^{n+1 \times n+1}$. To do so, we will repeat the same block-wise inverse computation. Let us denote $A = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$. First, we compute the Schur complement of $A$,

$$A/E = H - GE^{-1}F = 0 + u\mathbf{1}^\top \text{diag}(x)\mathbf{1} = ux^\top \mathbf{1}. \tag{7}$$

Since $x$ is feasible, we have $A/E = u$ due to the equality constraints ($x$ must sum to 1 as a probability mass function). Then, we have

$$A^{-1} = \begin{bmatrix} \text{diag}(x)^{-1} & -\mathbf{1} \\ u\mathbf{1}^\top & 0 \end{bmatrix}^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}^{-1} = \begin{bmatrix} E^{-1} + E^{-1}F(A/E)^{-1}GE^{-1} & -E^{-1}F(A/E)^{-1} \\ -(A/E)^{-1}GE^{-1} & (A/E)^{-1} \end{bmatrix}. \tag{8}$$

Plugging in, we have

$$\begin{aligned} A^{-1} &= \begin{bmatrix} \text{diag}(x) - \text{diag}(x)\mathbf{1}u^{-1}u\mathbf{1}^\top \text{diag}(x) & \text{diag}(x)\mathbf{1}u^{-1} \\ -u^{-1}u\mathbf{1}^\top \text{diag}(x) & u^{-1} \end{bmatrix} \\ &= \begin{bmatrix} \text{diag}(x) - xx^\top & u^{-1}x \\ -x^\top & u^{-1} \end{bmatrix}. \end{aligned} \tag{9}$$

Pulling out the top-left $n \times n$ block yields the Jacobian $\frac{dx}{d\theta} = \text{diag}(x) - xx^\top$, which agrees with directly differentiating softmax [Martins and Astudillo, 2016].

With this, we have shown that we can rewrite softmax as an optimization problem, and differentiate the solution of that problem wrt the parameters in a solver-agnostic manner.

### 3.5 Limitations

In order to compute the derivative $\frac{dx}{d\theta}$, we had to invert the Jacobian of $F$. However, The first part of $F$ (recall from Eqn. 3), the stationarity condition $\nabla_x \mathcal{L} = 0$, already involved the Jacobian of the Lagrangian $\mathcal{L}$. In general, this means that in order to apply the IFT to solutions of optimization problems, we must compute the inverse Hessian (or at least a Hessian-vector-product). The Hessian is a matrix of size $O(n^2)$, and inverting this would take $O(n^3)$ computation. Thankfully, there are relatively cheap ways of approximating this computation, such as with approximate (inverse) Hessian-vector-product techniques [Rajeswaran et al., 2019, Lorraine et al., 2019a].

### 3.6 Extensions

The methods we covered can be extended to variations of argmax problems other than the softmax problem. The softmax problem relaxed the argmax problem by introducing entropy regularization. Rather than regularizing with entropy, one could instead alter the objective to find the Euclidean projection of the parameters onto the probability simplex, resulting in SparseMax [Martins and Astudillo, 2016]. While the output of softmax variants often have a closed form expression, the IFT provides another way of deriving their Jacobians and could potentially pave the way for generalizations.

More generally, the IFT can be applied to cases where, unlike softmax, we do not have an explicit functional form (i.e., the unit circle), and outputs are governed only by a system of equations. This includes more general optimization problems, such as quadratic programs [Amos and Kolter, 2017] or other convex optimization problems [Agrawal et al., 2019], as well as the solutions of differential equations [Chen et al., 2018].

## 4 OptNet

OptNet generalizes the methodology applied above to the softmax problem by including parameterized constraints and computing the gradients of the solution wrt all parameters. This allows us to learn not only the objective, but also the constraints. Constraint learning allows models with optimization layers to perform well on tasks with hard constraints, such as learning to play Sudoku from only inputs and outputs [Amos and Kolter, 2017].

### 4.1 Quadratic Programs

OptNet applies the IFT to quadratic programs (QPs) in particular. As one of the first attempts to apply the IFT to differentiable optimization layers in neural networks, OptNet targeted QPs due to their status as the simplest nonlinear optimization problem (speculation). The methodology remains the same as the softmax problem: given a QP and a solution, use the KKT conditions to produce a system of equations then apply the IFT / implicit differentiation to compute the derivative of the solution wrt the parameters of the objective and constraints.

Quadratic programs take the following form:

$$\begin{array}{ll} \text{maximize} & \frac{1}{2}x^\top Qx + q^\top x \\ \text{subject to} & Ax = b \\ & Gx \leq h, \end{array} \tag{10}$$

where we optimize over $x$ and the parameters are $\theta = \{Q, q, A, b, G, h\}$. Compared to the softmax problem in Eqn. 1, we see that the main difference is just the learnable parameters in the constraints. As the application of the IFT is almost identical, we will not cover it in detail.

The second contribution of OptNet was the extension of a state-of-the-art interior point solver [Amos and Kolter, 2017], and its adaptation to parallel machines (GPUs) and batch processing. While outside the scope of this report, see the paper by Amos and Kolter [2017] for the details.

# 5 Semi-Amortized Variational Inference (POSTPONED)

We now apply the IFT to variational inference.

Variational inference has found success in recent applications to generative models, in particular by allowing practitioners to depart from conjugate models and extend emission models with expressive neural network components. The main insight that led to this development is that inference can be amortized through the use of an inference network. One approach to variational inference, stochastic variational inference (SVI), introduces local, independent variational parameters for every instance of hidden variable. While flexible, the storage of all variational parameters is expensive, and the optimization of each parameter independently slow []. Amortized variational inference (AVI) solves that by instead using a hierarchical process. Variational parameters are produced hierarchically via an inference network, which in turn generates the local variational parameters []. The resulting local parameters may or may not be subsequently optimized.

Failure to further optimize local variational parameters may result in an amortization gap []. Prior work has shown that this gap can be ameliorated by performing a few steps of optimization on the generated local parameters obtained from the inference network, and even by propagating gradients through the optimization process. Optimizing through the inner optimization problem results in semi-amortized variational inference (SAVI) [].

As our main motivating example, we will examine whether we can apply the IFT to SAVI. We will start by formalizing the problem of variational inference for a simple model.

We will start with a model defined by the following generative process, used by Dai et al. [2019] to analyze posterior collapse:

1. Choose a latent code from the prior distribution $z \sim p(z) = N(0, I)$.
2. Given the code, choose an observation from the emission distribution $x \mid z \sim p_\theta(x \mid z) = N(\mu_x(z, \theta), \gamma I)$,

where $\mu_x(z, \theta) \equiv \mathrm{MLP}(z, \theta)$ and $\gamma > 0$ is a hyperparameter. This yields the joint distribution $p(x, z) = p(x \mid z)p(z)$.

Since the latent code $z$ is unobserved, training this model would require optimizing the evidence $p(x) = \int p(x, z)$. However, due to the MLP parameterized $\mu_x$, the integral is intractable. Variational inference performs approximate inference by introducing variational distribution $q_\phi(z \mid x)$ and maximizing the following lower bound on $\log p(x)$:

$$\log p(x) - D_{\mathrm{KL}}\left[q(z \mid x) || p(z \mid x)\right] = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\right] = L(\theta, \phi). \tag{11}$$

(Write out objective in full.)

While SVI introduces local parameters for each instance of $z$, and AVI uses a single $q(z \mid x)$ for all instances, we will follow the approach of SAVI. We will perform inference as follows: For each instance $x$, produce local variational parameter $z^{(0)} = g(x; \phi)$. Obtain $z^*$ by solving $\mathcal{L}(\theta, z^{(0)}) = 2$, with (local) optima $\ell^*$. Take gradients through the whole procedure, i.e. compute $\frac{\partial \ell^*}{\partial \phi} = \frac{\partial \ell^*}{\partial z^*}\frac{\partial z^*}{\partial z^{(0)}}\frac{\partial z^{(0)}}{\partial \lambda}$. The main difficuty lies in computing $\frac{\partial z^*}{\partial z^{(0)}}$. (Highlight challenge)

In order to avoid the memory costs of storing all intermediate computation performed in a solver, we will instead apply the IFT. In order to apply the IFT, we must satisfy the three conditions. First, we must have a solution point to a system of equations, $F(x_0, z_0) = 0$. In this setting, we will use the KKT conditions of the optimization problem to define $F$.

# 6 Limitations

# References


A. Agrawal, B. Amos, S. T. Barratt, S. P. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. *CoRR*, abs/1910.12430, 2019. URL http://arxiv.org/abs/1910.12430.

B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. *CoRR*, abs/1703.00443, 2017. URL http://arxiv.org/abs/1703.00443.

Q. Bertrand, Q. Klopfenstein, M. Blondel, S. Vaiter, A. Gramfort, and J. Salmon. Implicit differentiation of lasso-type models for hyperparameter optimization, 2020.

T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. *CoRR*, abs/1806.07366, 2018. URL http://arxiv.org/abs/1806.07366.

B. Dai, Z. Wang, and D. P. Wipf. The usual suspects? reassessing blame for VAE posterior collapse. *CoRR*, abs/1912.10702, 2019. URL http://arxiv.org/abs/1912.10702.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL http://arxiv.org/abs/1703.03400.

B. Gao and L. Pavel. On the properties of the softmax function with application in game theory and reinforcement learning, 2018.

A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. *CoRR*, abs/1707.04585, 2017. URL http://arxiv.org/abs/1707.04585.

S. Gould, R. Hartley, and D. Campbell. Deep declarative networks: A new hope. *CoRR*, abs/1909.04866, 2019. URL http://arxiv.org/abs/1909.04866.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2nd edition, 2008.

M. J. Johnson, D. Duvenaud, A. B. Wiltschko, S. R. Datta, and R. P. Adams. Composing graphical models with neural networks for structured representations and fast inference, 2017.

W. Karush. *Minima of functions of several variables with inequalities as side conditions.* PhD thesis, Thesis (S.M.)–University of Chicago, Department of Mathematics, December 1939., 1939.

Y. Kim, S. Wiseman, A. C. Miller, D. Sontag, and A. M. Rush. Semi-amortized variational autoencoders, 2018.

S. Krantz and H. Parks. The implicit function theorem : History, theory, and applications / s.g. krantz, h.r. parks. 01 2003. doi: 10.1007/978-1-4612-0059-8.

H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950*, pages 481–492, Berkeley and Los Angeles, 1951. University of California Press.

J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. *CoRR*, abs/1911.02590, 2019a. URL http://arxiv.org/abs/1911.02590.

J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. *CoRR*, abs/1911.02590, 2019b. URL http://arxiv.org/abs/1911.02590.

D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning, 2015.

A. F. T. Martins and R. F. Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. *CoRR*, abs/1602.02068, 2016. URL http://arxiv.org/abs/1602.02068.

A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *CoRR*, abs/1909.04630, 2019. URL http://arxiv.org/abs/1909.04630.

M. Wainwright and M. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1:1–305, 01 2008. doi: 10.1561/2200000001.

# A Example Appendix

Neural ODEs use reversibility.