JacNet: Learning Functions with Structured Jacobians

Jonathan Lorraine [*] [1] [2] **Safwan Hossain** [*] [1] [2]

Abstract

Neural networks are trained to learn an approximate mapping from an input domain to a target domain. Often, incorporating prior knowledge about the true mapping is critical to learning a useful approximation. With current architectures, it is difficult to enforce structure on the derivatives of the input-output mapping. We propose to directly learn the Jacobian of the input-output function with a neural network, which allows easy control of derivative. We focus on structuring the derivative to allow invertibility, and also demonstrate other useful priors can be enforced, such as k-Lipschitz. Using this approach, we are able to learn approximations to simple functions which are guaranteed to be invertible, and easily compute the inverse. We also show a similar results for 1-Lipschitz functions.

1. Introduction

Neural networks (NNs) are the main workhorses of modern machine learning, finding use in approximating functions for a wide range of domains. Two traits that drive NNs success are (1) they are sufficiently flexible to approximate arbitrary functions, and (2) we can easily structure the output to incorporate certain prior knowledge about the range (e.g., softmax output activation for classification).

NN flexibility is formalized by showing they are *universal function approximators*. This means that given a continuous function y on a bounded interval I, a NN approximation $\hat{\mathrm{y}}_{\boldsymbol{\theta}}$ can satisfy: $\forall \mathrm{x} \in I, |\mathrm{y}(\mathrm{x}) - \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x})| < \epsilon$. Hornik et al. (1989) show that NNs with a single hidden layer and non-constant, bounded activation functions are universal approximators. While NNs can achieve point-wise approximations with arbitrary precision, little can be said about derivatives of the approximation w.r.t. the input. For example, NNs with step-function activations are flat almost everywhere. Therefore, they can not approximate arbitrary input derivatives, yet are still universal approximators.

The need to use derivatives of approximated functions arise in a number of scenarios. For example, in generative adversarial networks (Goodfellow et al., 2014), the generator differentiates through the discriminator to ensure the learned distribution is closer to the true distribution. In some multi-agent learning algorithms, an agent differentiates through how another agent responds (Foerster et al., 2018). Alternatively, hyperparameters can differentiate through a NN to see how to move to minimize validation loss (MacKay et al., 2019; Lorraine & Duvenaud, 2018).

We begin by giving background into the problem in § 2 and discuss related work in § 3. Then, we introduce relevant theory for our algorithm in § 4 followed lastly by our experimental results in § 5.

1.1. Contributions

- We propose a method for learning functions by learning their Jacobian, and provide empirical results.
- We show how to to make our learned function satisfy regularity conditions - invertible, or Lipschitz - by making the Jacobian satisfy regularity conditions
- We show how the learned Jacobian can satisfy regularity conditions by an appropriate output activation choice.

2. Background

In this section we set up the standard notation used in § 4. Our goal is to learn a C^1 function $\mathrm{y}(\mathrm{x}) : \mathcal{X} \to \mathcal{Y}$. Denote $\mathrm{d}_{\mathrm{x}}, \mathrm{d}_{\mathrm{y}}$ as the dimension of $\mathcal{X}, \mathcal{Y}$ respectively. Here, we assume that $\mathrm{x} \sim p(\mathrm{x})$ and y is deterministic. We will learn the function through a NN - $\hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x})$ - which is parameterized by weights $\boldsymbol{\theta} \in \Theta$. Also, assume we have a bounded loss function $\mathcal{L}(\mathrm{y}(\mathrm{x}), \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x}))$, which attains its minimum when $\mathrm{y}(\mathrm{x}) = \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x})$. Our population risk is $R(\boldsymbol{\theta}) = \mathbb{E}_{p(\mathrm{x})}[\mathcal{L}(\mathrm{y}(\mathrm{x}), \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x}))]$, and we wish to find $\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} R(\boldsymbol{\theta})$. In practice, we have a finite number of samples from the input distribution $\mathcal{D} = \{(\mathrm{x}_i, \mathrm{y}_i) | i = 1 \dots n\}$, and we minimize the empirical risk:

$$\hat{\boldsymbol{\theta}}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \hat{R}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, {}^1\!/\!_n \sum_{\mathcal{D}} \mathcal{L}(\mathrm{y}_i, \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x}_i, \boldsymbol{\theta}))$$

It is common to have prior knowledge about the structure of $\mathrm{y}(\mathrm{x})$ which we want to bake into the learned function. If

[*] Equal contribution [1] University of Toronto [2] Vector Institute. Correspondence to: Jonathan Lorraine <lorraine@cs.toronto.edu>.

First workshop on *Invertible Neural Networks and Normalizing Flows* (ICML 2019), Long Beach, CA, USA

we know about the bounds of the output domain, properly structuring the predictions through output activation is an easy way to enforce this. Examples include using softmax for classification, or ReLU for a non-negative output.

In the more general case, we may want to ensure our learned function satisfies certain derivative conditions, as many function classes can be expressed in such a way. For example, a function is locally invertible if its Jacobian has a non-zero determinant in that neighbourhood. Similarly, a function is k-Lipschitz if its derivative norm lies inside $[-k, k]$.

We propose to explicitly learn this Jacobian through a NN $J_{\boldsymbol{\theta}}(\mathrm{x})$ parameterized by θ and use a numerical integrator to evaluate $\hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x})$. We show that with a suitable choice of output activation for $J_{\boldsymbol{\theta}}(\mathrm{x})$, we can guarantee our function is globally invertible, or k-Lipschitz.

3. Related Work

Enforcing derivative conditions: There is existing work on strictly enforcing derivative conditions on learned functions. For example, if we know the function to be k-Lipschitz, one method is weight clipping (Arjovsky et al., 2017). Anil et al. (2018) recently proposed an architecture which learns functions that are guaranteed to be 1-Lipschitz and theoretically capable of learning all such functions. Amos et al. (2017) explore learning scalar functions that are guaranteed to be convex (i.e., the Hessian is positive semi-definite) in their inputs. While these methods guarantee derivative conditions, they can be non-trivial to generalize to new conditions, limit expressiveness, or involve expensive projections. Czarnecki et al. (2017) propose a training regime which involves penalizing the function when it violates higher order constraints. This does not guarantee the regularity conditions and requires knowing the exact derivative at each sample - however, it is easy to use.

Differentiating through integration: Training our proposed model requires back-propagating through a numerical integration procedure. We leverage differentiable numerical integrators provided by Chen et al. (2018) who use it to model the dynamics of different layers of a NN as an ordinary differential equations. FFOJRD (Grathwohl et al., 2018) use this for training as it model layers of a reversible network as an ODE. Our approach differs in that we explicitly learn the Jacobian of our input-output mapping, and integrate along arbitrary paths in the input or output domain.

Invertible Networks: In Behrmann et al. (2018), an invertible residual network is learned with contractive layers, and a numerical fixed point procedure used to evaluate the inverse. In contrast, we need a non-zero Jacobian determinant and use numerical integration to evaluate the inverse. Other reversible architectures include NICE (Dinh et al., 2014), Real-NVP (Dinh et al., 2016), RevNet (Jacobsen et al., 2018), and Glow (Kingma & Dhariwal, 2018).

4. Theory

We are motivated by the idea that a function can be learned by combining initial conditions with an approximate Jacobian. Consider a deterministic C^1 function $\mathrm{y} : \mathcal{X} \to \mathcal{Y}$ that we wish to learn. Let $J_{\mathrm{x}}^{\mathrm{y}} : \mathcal{X} \to \mathbb{R}^{\mathrm{d_x} \times \mathrm{d_y}}$ be the Jacobian of y w.r.t. x. We can evaluate the target function by evaluating the following line integral with some initial condition $(\mathrm{x}_o, \mathrm{y}_o = \mathrm{y}(\mathrm{x}_o))$:

$$\mathrm{y}(\mathrm{x}) = \mathrm{y}_o + \int_{\boldsymbol{c}(\mathrm{x}_o, \mathrm{x})} J_{\mathrm{x}}^{\mathrm{y}}(\mathrm{x}) ds$$

In practice, this integral is evaluated by parameterizing a path between x_o and x and numerically approximating integral. Note that the choice of path and initial condition do not affect the result by the fundamental theorem of line integrals. We can write this as an explicit line integral for some path $\boldsymbol{c}(t, \mathrm{x}_o, \mathrm{x})$ from x_o to x parameterized by $t \in [0, 1]$ satisfying $\boldsymbol{c}(0, \mathrm{x}_o, \mathrm{x}) = \mathrm{x}_o$ and $\boldsymbol{c}(1, \mathrm{x}_o, \mathrm{x}) = \mathrm{x}$ with ${}^{d}\!/_{dt}(\boldsymbol{c}(t, \mathrm{x}_o, \mathrm{x})) = \boldsymbol{c}'(t, \mathrm{x}_o, \mathrm{x})$:

$$\mathrm{y}(\mathrm{x}) = \mathrm{y}_o + \int_{t=0}^{t=1} J_{\mathrm{x}}^{\mathrm{y}}(\boldsymbol{c}(t, \mathrm{x}_o, \mathrm{x}))\boldsymbol{c}'(t, \mathrm{x}_o, \mathrm{x}) dt$$

A simple choice of path is $\boldsymbol{c}(t, \mathrm{x}_o, \mathrm{x}) = (1 - t)\mathrm{x}_o + t\mathrm{x}$, which has $\boldsymbol{c}'(t, \mathrm{x}_o, \mathrm{x}) = \mathrm{x} - \mathrm{x}_o$. Thus to approximate $\mathrm{y}(\mathrm{x})$, we can combine initial conditions with an approximate $J_{\mathrm{x}}^{\mathrm{y}}$. We propose to learn an approximate, Jacobian $J_{\boldsymbol{\theta}}(\mathrm{x}) : \mathcal{X} \to \mathbb{R}^{\mathrm{d_x} \times \mathrm{d_y}}$, with a NN parameterized by $\boldsymbol{\theta} \in \Theta$. For training, consider the following prediction function:

$$\hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x}) = \mathrm{y}_o + \int_{t=0}^{t=1} J_{\boldsymbol{\theta}}(\boldsymbol{c}(t, \mathrm{x}_o, \mathrm{x}))\boldsymbol{c}'(t, \mathrm{x}_o, \mathrm{x}) dt$$

We can compute the empirical risk, $\hat{R}$, with this prediction function by choosing some initial conditions $(\mathrm{x}_o, \mathrm{y}_o) \in \mathcal{D}$, a path $\boldsymbol{c}$, and using numerical integration. To backpropagate errors to update our network parameters $\boldsymbol{\theta}$, we must backpropagate through the numerical integration.

4.1. Derivative Conditions for Invertibility

The inverse function theorem (Spivak, 1965) states that a function is locally invertible if the Jacobian at that point is invertible. Additionally, we can compute the Jacobian of f^{-1} by computing inverse of the Jacobian of f. Many non-invertible functions are locally invertible almost everywhere (e.g., $y = x^2$).

The Hadamard global inverse function theorem (Hadamard, 1906), is an example of a global invertibility criterion. It states that a function $f : \mathcal{X} \to \mathcal{Y}$ is globally invertible if the Jacobian determinant is everywhere non-zero and f is *proper*. A function is *proper* if whenever $\mathcal{Y}$ is compact, $f^{-1}(\mathcal{Y})$ is compact. This provides an approach to guarantee global invertibility of a learned function.

4.2. Structuring the Jacobian

We could guarantee that our Jacobian for an $\mathbb{R}^n \to \mathbb{R}^n$ mapping is invertible, by guaranteeing it has non-zero eigenvalues. For example, with a small positive ϵ we could use an output activation of:

$$J'_{\boldsymbol{\theta}}(\mathrm{x}) = J_{\boldsymbol{\theta}}(\mathrm{x})J^T_{\boldsymbol{\theta}}(\mathrm{x}) + \epsilon I$$

Here, $J_{\boldsymbol{\theta}}(\mathrm{x})J^T_{\boldsymbol{\theta}}(\mathrm{x})$ is a flexible PSD matrix, while adding ϵI makes it positive definite. A positive definite matrix has strictly positive eigenvalues, which implies invertibility. However, this output activation restricts the set of invertible functions we can learn as the Jacobian can only have positive eigenvalues. In future work, we wish to explore activations that are less restrictive while still guaranteeing invertibility.

Many other regularity conditions on a function can be specified in terms of the derivatives. For example, a function is Lipschitz if the derivatives of the function are bounded, which can be done with a k-scaled $\tanh$ activation function as our output activation. Alternatively we could learn a complex differentiable function by satisfying $\partial u/\partial a = \partial v/\partial b, \partial u/\partial b = -\partial v/\partial a$, where u, v are output components and a, b are input components. We focus on invertibility and Lipschitz because they are common in machine learning.

4.3. Computing the Inverse

Once the Jacobian is learned, it allows easy computation of f^{-1} by integrating the inverse Jacobian along a path $\boldsymbol{c}(t, \mathrm{y}_o, \mathrm{y})$ in the output space, given some initial conditions $(\mathrm{x}_o, \mathrm{y}_o = \hat{\mathrm{y}}_{\boldsymbol{\theta}}(\mathrm{x}_o))$:

$$\mathrm{x}(\mathrm{y}, \boldsymbol{\theta}) = \mathrm{x}_o + \int_{t=0}^{t=1} \left(J_{\boldsymbol{\theta}}(\boldsymbol{c}(t, \mathrm{y}_o, \mathrm{y}))\right)^{-1} \boldsymbol{c}'(t, \mathrm{y}_o, \mathrm{y}) dt$$

If the computational bottleneck is inverting $J_{\boldsymbol{\theta}}$, we propose to learn a matrix which is easily invertible (e.g., Kronecker factors of $J_{\boldsymbol{\theta}}$).

4.4. Backpropagation

Training the model requires back-propagating through numerical integration. To address this, we consider the recent work of Chen et al. (2018) which provides tools to efficiently back-propagate across numerical integrators. We combine their differentiable integrators on intervals with autograd for our rectification term $\boldsymbol{c}'$. This provides a differentiable numerical line integrator, which only requires a user to specify a differentiable path and the Jacobian.

The integrators allow a user to specify solution tolerance. We propose annealing the tolerance to be tighter whenever the evaluated loss is lower than our tolerance. In practice, this provides significant computational savings. Additionally, if the computational bottleneck is numerical integration, we may be able to adaptively select initial conditions $(\mathrm{x}_o, \mathrm{y}_o)$ that are near our target, which will reduce the number of evaluations steps in our integrator.

4.5. Conservation

We run into complexities when our input domain dimensionality $\mathrm{d_x} > 1$. For simplicity, assume $\mathrm{d_y} = 1$ and we are attempting to learn a vector field that is the gradient. Vector fields that are the gradient of a function are known as conservative. Our learned function $J_{\boldsymbol{\theta}}$ is a vector field but not necessarily conservative. As such, $J_{\boldsymbol{\theta}}$ may not be the gradient of any scalar potential function and the value of our line integral depends on the path choice. Investigating potential problems and solutions to these problems is relegated to future work.

5. Experiments

In our experiments, we explore learning invertible and Lipschitz functions with the following setup: Our input and output domains are $\mathcal{X} = \mathcal{Y} = \mathbb{R}$. We select $\mathcal{L}(\mathrm{y}_1, \mathrm{y}_2) = \|\mathrm{y}_1 - \mathrm{y}_2\|$. Our training set consists of 5 points sampled uniformly from $[-1, 1]$, while our test set has 100 points sampled uniformly from $[-2, 2]$. The NN architecture is fully-connected with a single layer with 64 hidden units and output activation on our network depends on the gradient regularity condition we want. We used Adam (Kingma & Ba, 2014) to optimize our network with a learning rate of 0.01 and all other parameters at defaults. We use full batch gradient estimates for 50 iterations.

To evaluate the function at a point, we use an initial value and parameterize a linear path between the initial and terminal point and use the numerical integrator from Chen et al. (2018). The path is trivially the interval between x_o and x because $\mathrm{d_x} = 1$, and our choice of the initial condition is $\mathrm{x}_o = 0$. We adaptively alter the tolerances of the integrator, starting with loose tolerances and decreasing them by half when the training loss is less than the tolerance, which provides significant gains in training speed.

For invertible function experiment, we learn the exponential function $\mathrm{y}(\mathrm{x}) = \exp(\mathrm{x})$. We use an output activation of $J'_{\boldsymbol{\theta}}(\mathrm{x}) = J_{\boldsymbol{\theta}}(\mathrm{x})J_{\boldsymbol{\theta}}(\mathrm{x})^T + \epsilon I$ for $\epsilon = 0.0001$, which guarantees a non-zero Jacobian determinant. Once trained, we take the learned derivative $J'_{\boldsymbol{\theta}}(\mathrm{x})$ and compute it's inverse, which by the inverse function theorem gives the derivative of the inverse. We compute the inverse of the prediction function, by integrating the inverse of the learned derivative. Figure 1 qualitatively explores the learned function, and the top of Figure 3 quantitatively explores the training procedure.

For the Lipschitz experiment, we learn the absolute value function $\mathrm{y}(\mathrm{x}) = |\mathrm{x}|$, which is a canonical 1-Lipschitz example in Anil et al. (2018). We use an output activation on our NN of $J'_{\boldsymbol{\theta}}(\mathrm{x}) = \tanh(J_{\boldsymbol{\theta}}(\mathrm{x})) \in [-1, 1]$, which guarantees

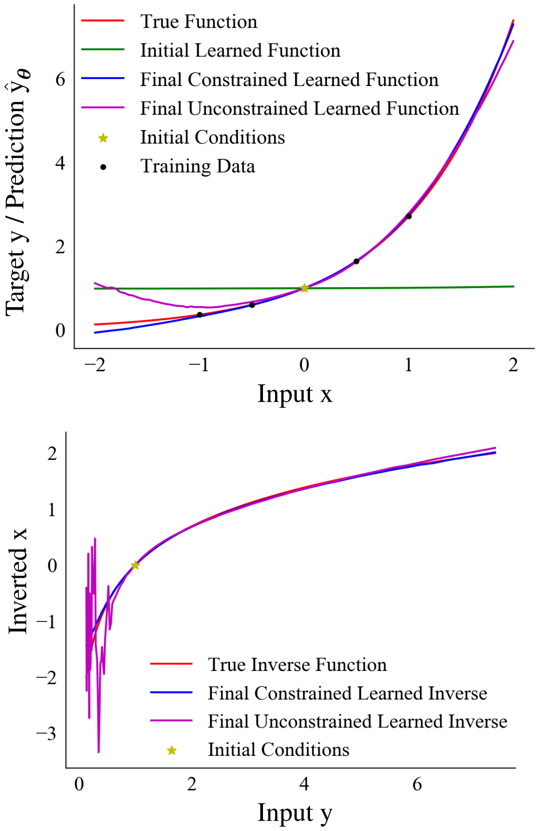


Figure 1. A graph of the *invertible* target function $\exp(x)$, the prediction function at the trained network weights, and the prediction function at the initial network weights. The initial prediction function is inaccurate, while the final prediction function matches the target function closely. Additionally, the learned inverse of the prediction function closely matches the true inverse function. We include graphs of an unconstrained learned function whose Jacobian can be zero. Note how the unconstrained function is not invertible everywhere.

our prediction function is 1-Lipschitz. Figure 2 qualitatively explores the learned function, and the bottom of Figure 3 quantitatively explores the training procedure.

6. Conclusion

We present a technique for approximating a function by learning its Jacobian and integrating it. This method may prove useful when attempting to guarantee properties of the function's Jacobian. A few examples of this include learning invertible, Lipschitz, or complex differentiable functions. Promising small scale experiments are presented, motivating further exploration in scaling up the results. In the future, we hope that this work may facilitate domain experts to easily incorporate a large variety of Jacobian regularity conditions into their models.

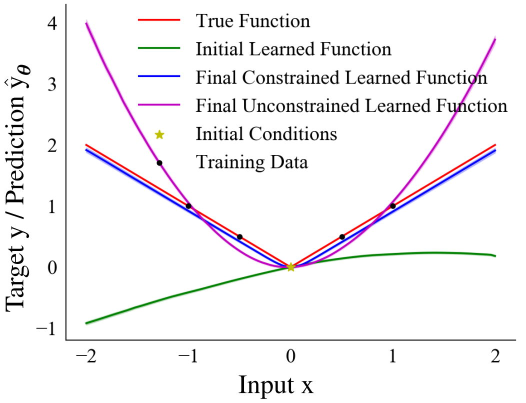
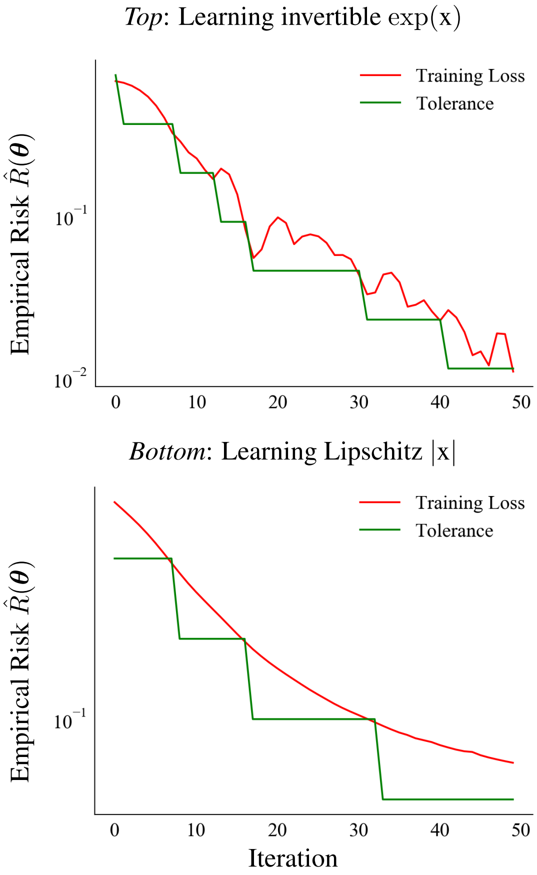


Figure 2. A graph of the *1-Lipschitz* target function $|x|$, the prediction function at the trained network weights, and the prediction function at the initial network weights. Note how the initial prediction function is inaccurate, while the final prediction function matches the target function closely. Additionally, note how the learned prediction function is 1-Lipschitz at initialization and after training. We include graphs of an unconstrained learned function whose derivative is not bounded by [-1, 1]. This does not generalize well to unseen data.


Top: Learning invertible $\exp(x)$

Bottom: Learning Lipschitz $|x|$

Figure 3. A graph of the empirical risk or training loss versus training iteration. As training progresses we tighten the tolerance of the numerical integrator to continue decreasing the loss at the cost of more computationally expensive iterations. *Top*: The training dynamics for learning the invertible target function. *Bottom*: The training dynamics for learning the Lipschitz target function.

References

Amos, B., Xu, L., and Kolter, J. Z. Input convex neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 146–155. JMLR. org, 2017.

Anil, C., Lucas, J., and Grosse, R. Sorting out lipschitz function approximation. *arXiv preprint arXiv:1811.05381*, 2018.

Arjovsky, M., Chintala, S., and Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

Behrmann, J., Duvenaud, D., and Jacobsen, J.-H. Invertible residual networks. *arXiv preprint arXiv:1811.00995*, 2018.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pp. 6571–6583, 2018.

Czarnecki, W. M., Osindero, S., Jaderberg, M., Swirszcz, G., and Pascanu, R. Sobolev training for neural networks. In *Advances in Neural Information Processing Systems*, pp. 4278–4287, 2017.

Dinh, L., Krueger, D., and Bengio, Y. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Foerster, J., Chen, R. Y., Al-Shedivat, M., Whiteson, S., Abbeel, P., and Mordatch, I. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 122–130. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Grathwohl, W., Chen, R. T., Betterncourt, J., Sutskever, I., and Duvenaud, D. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

Hadamard, J. Sur les transformations ponctuelles. *Bull. Soc. Math. France*, 34:71–84, 1906.

Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Jacobsen, J.-H., Smeulders, A., and Oyallon, E. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kingma, D. P. and Dhariwal, P. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, pp. 10215–10224, 2018.

Lorraine, J. and Duvenaud, D. Stochastic hyperparameter optimization through hypernetworks. *arXiv preprint arXiv:1802.09419*, 2018.

MacKay, M., Vicol, P., Lorraine, J., Duvenaud, D., and Grosse, R. Self-tuning networks: Bilevel optimization of hyperparameters using structured best-response functions. In *International Conference on Learning Representations (ICLR)*, 2019.

Spivak, M. *Calculus on manifolds: a modern approach to classical theorems of advanced calculus*. Addison-Wesley Publishing Company, 1965.