AlgoNet: C^∞ Smooth Algorithmic Neural Networks for Solving Inverse Problems

Felix Petersen
University of Konstanz
felix.petersen@uni.kn

Christian Borgelt
University of Salzburg
christian@borgelt.net

Oliver Deussen
University of Konstanz
oliver.deussen@uni.kn

Abstract

Artificial neural networks revolutionized many areas of computer science in recent years since they provide solutions to a number of previously unsolved problems. On the other hand, for many problems, classic algorithms exist, which typically exceed the accuracy and stability of neural networks. To combine these two concepts, we present a new kind of neural networks—algorithmic neural networks (AlgoNets). These networks integrate smooth versions of classic algorithms into the topology of neural networks. Our novel reconstructive adversarial network (RAN) enables solving inverse problems without or with only weak supervision.

1 Introduction

Artificial Neural Networks are employed to solve numerous problems, not only in computer science but also in all other natural sciences. Yet, the reasoning for the topologies of neural networks seldom reaches beyond empirically-based decisions.

In this work, we present a novel approach to designing neural networks—algorithmic neural networks (short: AlgoNet). Such networks integrate algorithms as algorithmic layers into the topology of neural networks. However, propagating gradients through such algorithms is problematic, because crisp decisions (conditions, maximum, etc.) introduce discontinuities into the loss function. If one passes from one side of a crisp decision to the other, the loss function may change in a non-smooth fashion—it may "jump." That is, the loss function suddenly improves (or worsens, depending on the direction) without these changes being locally noticeable anywhere but exactly at these "jumps." Hence, a gradient descent based training, regardless of the concrete optimizer, cannot approach these "jumps" in a systematic fashion, since neither the loss function nor the gradient provides any information about these "jumps" in any place other than exactly the location at which they occur. Therefore, a smoothing is necessary, such that information about the direction of improvement becomes exploitable by gradient descent also in the area surrounding the "jump." That is, by smoothing, e.g., an `if`, one can smoothly, by gradient descent, undergo a transition between the two crisp cases using only local gradient information.

Generally, for end-to-end trainable neural network systems, all components should at least be C^0 smooth, i.e., continuous, to avoid "jumps." However, having C^k smooth, i.e., k times differentiable and then still continuous components with $k \geq 1$ is favorable. This property of higher smoothness allows for higher-order derivatives and thus prevents unexpected behavior of the gradients. Hence, we designed smooth approximations to basic algorithms where the functions representing the algorithms are ideally C^∞ smooth. That is, we designed pre-programmed neural networks (restricted to smooth components) with the structure of given algorithms.

Such algorithmic layers (Sec. 3) can assist in finding an appropriate solution for (ill-posed) inverse problems. For that, we introduce the Reconstructive Adversarial Network (RAN) in (Sec. 3).

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

2 Related Work

Related work [1]–[3] in neural networks focused on dealing with crisp decisions by passing through gradients for the alternatives of the decisions. There is no smooth transition between the alternatives, which introduces discontinuities in the loss function that hinder learning, which of the alternatives should be chosen. TensorFlow contains a sorting layer (`tf.sort`) as well as a while loop construct (`tf.while_loop`). Since the sorting layer only performs a crisp relocation of the gradients and the while loop has a crisp exit condition, there is no gradient with respect to the conditions in these layers. Concurrently, we developed a smooth sorting layer and a smooth while loop.

Theoretical work by DeMillo *et al.* [4] proved that any program could be modeled by a smooth function. Consecutive works [5]–[7] provided approaches for smoothing programs using, i.a., Gaussian smoothing [6], [7].

3 Algorithmic Layers

Algorithmic layers, i.e., smooth approximations, exist for any Turing computable algorithm [8]. To design a smooth algorithmic layer, all discrete cases (e.g., conditions of `if` statements or loops) have to be replaced by continuous or smooth functions. The essential property is that the implementation is differentiable with respect to all internal choices and does not—as in previous work—only carry the gradients through the algorithm [1]. For example, an `if` statement can be replaced by a sigmoid-weighted sum of both cases. By using a smooth sigmoid function, the statement is smoothly interpreted. Hence, the gradient descent method can influence the condition to hold if the content of the `then` case reduces the loss and influence the condition to fail if the loss is lower when the `else` case is executed. Thus, the partial derivative with respect to a neuron is computed because the neuron is used in the `if` statement. In contrast, when propagating back the gradient of the `then` or the `else` case depending on the value of the condition, there is a discontinuity at the points where the value of the condition changes and the partial derivative of the neuron in the condition equals zero.

The logistic sigmoid function (Eq. 1) is a C^∞ smooth replacement for the Heaviside sigmoid function (Eq. 2), which is equivalent to the `if` statement. Alternatively, one could use other sigmoid functions, e.g., the C^1 smooth step function $x^2 - 2 \cdot x^3$ for $x \in [0, 1]$, and 0 and 1 for all values before and after the given range, respectively.

$$s_1(x, s) = \frac{1}{1 + e^{-x \cdot s}} \qquad \text{with } s = 1 \quad (1)$$

$$s_2(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{else} \end{cases} \quad (2)$$

After designing an algorithmic layer, we can use it to solve for its inverse by using the Reconstructive Adversarial Neural Network (RAN).

4 Reconstructive Adversarial Networks (RAN)

Reconstructive Adversarial Networks (RAN) use an algorithm that solves for the inverse of a given problem. For example, they use a smooth renderer for 3D-reconstruction, a smooth iterated function system (IFS) for solving the inverse-problem of IFS, and a smooth text-to-speech synthesizer for speech recognition. While RANs could be used in supervised settings, they are designed for unsupervised or weakly supervised solving of inverse-problems. Their concept is the following:

Input ($\in A$) $\rightarrow$ `Reconstructor` $\rightarrow$ Goal $\rightarrow$ `smooth inverse` $\rightarrow$ Smooth version of input ($\in B$)

This structure is similar to auto-encoders and the encoder-renderer architecture presented by Che *et al.* [2]. Such an architecture, however, cannot directly be trained since there is a domain shift between the input domain A and the smooth output domain B. Thus, we introduce domain translators ($a2b$ and $b2a$) to translate between these two domains. Since training is extremely hard with three consecutive components, of which the middle one is highly restrictive, we use the RAN as a novel training schema for these components. For that, we also include a discriminator to allow for adversarial training of the components $a2b$ and $b2a$. Of our five components four are trainable (the reconstructor `R`, the domain translators $a2b$ and $b2a$, and the discriminator *D*), and one is non-trainable (the smooth inverse *Inv*).

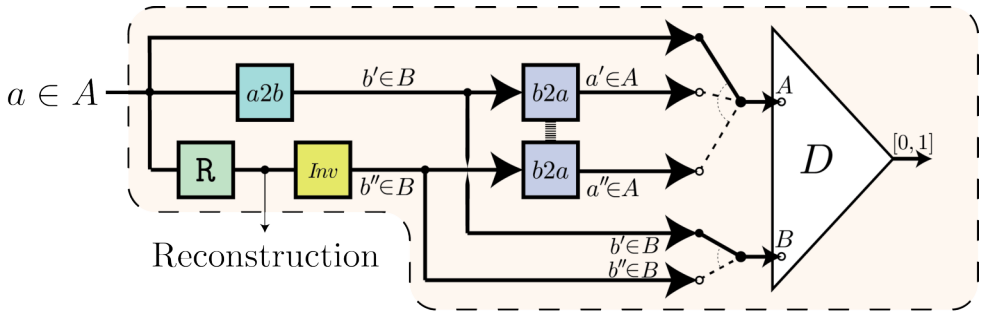

Figure 1: RAN System overview. The reconstructor receives an object from the input domain A and predicts the corresponding reconstruction. The reconstruction, then, is validated through our smooth inverse. The latter produces objects in a different domain, B, which are translated back to the input domain A for training purposes ($b2a$). Unlike in traditional GAN systems, the purpose of our discriminator D is mainly to indicate whether the two inputs match in content, not in style. Our novel training scheme trains the whole network via five different data paths, including two which require another domain translator, $a2b$.

Since, initially, neither the reconstructor nor the domain translators are trained, we are confronted with a causality dilemma. A typical approach for solving such causality dilemmas is to solve the two components coevolutionarily by iteratively applying various influences towards a common solution. Fig. 1 depicts the structure of the RAN, which allows for such a coevolutionary training scheme.

The discriminator receives two inputs, one from space A and one from space B. One of these inputs (either A or B) receives two values, a real and a fake value; the task of the discriminator is to distinguish between these two, given the other input. For training, the discriminator is trained to distinguish between the different path combinations for the generation of inputs. Consecutively, the generator modules are trained to fool the discriminator. This adversarial game allows training the RAN.

In the following, we will present this process, as well as its involved losses, in detail. Our optimization of $\mathtt{R}$, $a2b$, $b2a$, and D involves adversarial losses, cycle-consistency losses, and regularization losses. Specifically, we solve the following optimization:

$$\min_{\mathtt{R}} \min_{a2b} \min_{b2a} \max_{D} \mathcal{L} \qquad \text{or in greater detail} \qquad \min_{\mathtt{R}} \min_{a2b} \min_{b2a} \max_{D} \sum_{i=1}^{5} (\alpha_i \cdot \mathcal{L}_i) + \mathcal{L}_{\text{reg}}.$$

where α_i is a weight in $[0, 1]$ and $\mathcal{L}$, and $\mathcal{L}_i$ shall be defined below. $\mathcal{L}_{\text{reg}}$ denotes the (optional) regularization losses imposed on the reconstruction output.

We define $b', b'' \in B$ and $a', a'' \in A$ in dependency of $a \in A$ according to Fig. 1 as

$$b' = a2b(a) \qquad b'' = \mathit{Inv} \circ \mathtt{R}(a) \qquad a' = b2a(b') \qquad a'' = b2a(b'').$$

With that, our losses are (without hyper-parameter weights)

$$\begin{aligned}
\mathcal{L}_1 &= \mathbb{E}_{a\sim A}[\log D(a, b'')] + \mathbb{E}_{a\sim A}[\log(1 - D(a, b'))] + \mathbb{E}_{a\sim A}[\|b'' - b'\|_1] \\
\mathcal{L}_2 &= \mathbb{E}_{a\sim A}[\log D(a, b'')] + \mathbb{E}_{a\sim A}[\log(1 - D(a'', b''))] + \mathbb{E}_{a\sim A}[\|a'' - a\|_1] \\
\mathcal{L}_3 &= \mathbb{E}_{a\sim A}[\log D(a, b')] + \mathbb{E}_{a\sim A}[\log(1 - D(a'', b'))] + \mathbb{E}_{a\sim A}[\|a' - a\|_1] + \mathbb{E}_{a\sim A}[\|b'' - b'\|_1] \\
\mathcal{L}_4 &= \mathbb{E}_{a\sim A}[\log D(a, b'')] + \mathbb{E}_{a\sim A}[\log(1 - D(a', b''))] + \mathbb{E}_{a\sim A}[\|a' - a\|_1] + \mathbb{E}_{a\sim A}[\|b'' - b'\|_1] \\
\mathcal{L}_5 &= \mathbb{E}_{a\sim A}[\log D(a, b')] + \mathbb{E}_{a\sim A}[\log(1 - D(a', b'))] + \mathbb{E}_{a\sim A}[\|a' - a\|_1].
\end{aligned}$$

We alternately train the different sections of our network in the following order:

1. The discriminator D
2. The translation from B to A ($b2a$)
3. The components that perform a translation from A to B ($\mathtt{R}$+*Inv*, $a2b$)

For each of these sections, we separately train the five losses $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4$, and $\mathcal{L}_5$. In our experiments, we used one Adam optimizer [9] for each trainable component ($\mathtt{R}$, $a2b$, $b2a$, and D).

5 Experiments

For our experiments we developed an unsupervised 3D reconstruction method using a C^∞ smooth 3D mesh renderer [11]. Our reconstructor is a network mapping from one or multiple images to a set of 3D coordinates, and our smooth inverse is a smooth 3D mesh renderer. As domain translators, we used the `pix2pix` network as well as a convolutional and deconvolutional ResNet [12].

Describing the smooth renderer in great detail would exceed the scope of this paper. The main differences to common ray tracers are that our renderer performs a smooth rasterization and a smooth occlusion handling / z-buffer.

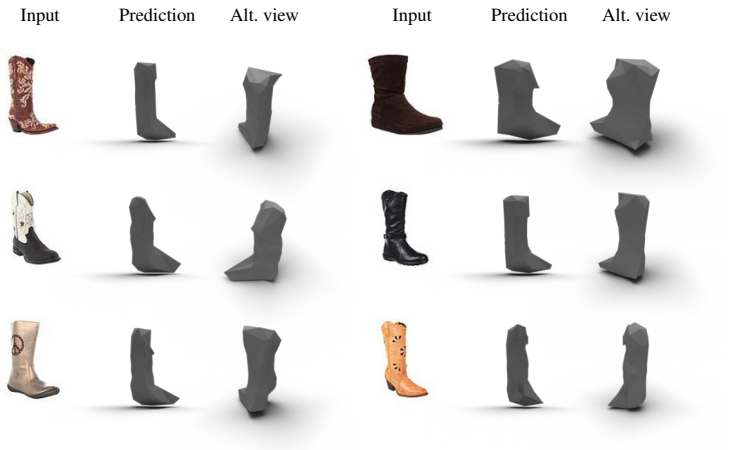

Figure 2: Single-view reconstruction results from the UT Zappos50K dataset [10] (camera-captured images). With an increased supervision, our method achieves significantly better results.

Training the RAN with the scheme described in Sec. 4, we achieved first results on unsupervised 3D reconstruction trained only on camera-captured images. Some qualitative results for that are presented in Fig. 2. Other inverse problems that we experimented on are speech recognition as well as the inverse problem of iterated function systems.

6 Discussion and Conclusion

We presented AlgoNets as a new kind of layers for neural networks and RANs as a novel technique for solving ill-posed inverse problems. Concurrent with their benefits, AlgoNets, such as the aforementioned rendering layer, can get computationally very expensive. On the other hand, the rendering layer is very powerful since it allows training a 3D reconstruction without 3D supervision using the RAN. Since the RAN is a very complex architecture that requires a very specific training paradigm, it can also take relatively long to train it. To accommodate this issue, we found that by increasing some loss weights and introducing a probability of whether the computation is executed, the training time can be reduced by a factor of two or more.

The AlgoNet can also be used in such a way that algorithmic layers solve sub-problems of a given problem to assist a neural network in solving a larger problem. This principle could also be used in the realm of explainable artificial intelligence [13] by adding residual algorithmic layers into neural networks and then analyzing the neurons of the trained AlgoNet. For that, network activation and/or network sensitivity can indicate the relevance of the residual algorithmic layer. To compute the network sensitivity of an algorithmic layer, the gradient with respect to additional weights (constant equal to one) in the algorithmic layer could be computed. By that, similarities between classic algorithms and the behavior of neural networks could be inferred. An alternative approach would be to gradually replace parts of trained neural networks with algorithmic layers and analyzing the effect on the new model accuracy.

In the future, we will develop a high-level smooth programming language to improve smooth representations of higher-level programming concepts. Adding trainable weights to the algorithmic layers to improve the accuracy of smooth algorithms and/or allow the rest of the network to influence the behavior of the algorithmic layer is subject to future research. Another future objective is the exploration of neural networks not with a fixed but instead a smooth topology.

References

[1] Mart'in Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. [Online]. Available: `https://www.tensorflow.org/`; `https://www.tensorflow.org/api_docs/python/tf/sort`; `https://www.tensorflow.org/api_docs/python/tf/while_loop`;;

[2] C. Che, F. Luan, S. Zhao, K. Bala, and I. Gkioulekas, "Inverse Transport Networks," Sep. 2018. [Online]. Available: `http://arxiv.org/abs/1809.10820`.

[3] P. Henderson and V. Ferrari, "Learning to Generate and Reconstruct 3D Meshes with only 2D Supervision," Jul. 2018. [Online]. Available: `https://arxiv.org/abs/1807.09259`.

[4] R. A. DeMillo and R. J. Lipton, "Comments on "Defining Software by Continuous Smooth Functions," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 307–309, 1993, ISSN: 00985589. DOI: `10.1109/32.221140`.

[5] Y. Nesterov, "Smooth minimization of non-smooth functions," *Mathematical Programming*, vol. 103, no. 1, pp. 127–152, 2005, ISSN: 00255610. DOI: `10.1007/s10107-004-0552-5`.

[6] S. Chaudhuri and A. Solar-Lezama, "Smoothing a Program Soundly and Robustly," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 277–292, ISBN: 978-3-642-22110-1.

[7] Y. Yang and C. Barnes, "Approximate Program Smoothing Using Mean-Variance Statistics, with Application to Procedural Shader Bandlimiting," Jun. 2017. [Online]. Available: `https://arxiv.org/abs/1706.01208`.

[8] F. Petersen, C. Borgelt, and O. Deussen, "AlgoNet: C^∞ Smooth Algorithmic Neural Networks," May 2019. [Online]. Available: `http://arxiv.org/abs/1905.06886`.

[9] D. P. Kingma and J. Ba, "Adam: {A} Method for Stochastic Optimization," *CoRR*, vol. abs/1412.6, 2014. [Online]. Available: `http://arxiv.org/abs/1412.6980`.

[10] A. Yu and K. Grauman, "Fine-Grained Visual Comparisons with Local Learning," in *Computer Vision and Pattern Recognition (CVPR)*, Jun. 2014.

[11] F. Petersen, A. H. Bermano, O. Deussen, and D. Cohen-Or, "Pix2Vex: Image-to-Geometry Reconstruction using a Smooth Differentiable Renderer," Mar. 2019. [Online]. Available: `http://arxiv.org/abs/1903.11149`.

[12] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," Nov. 2016. [Online]. Available: `https://arxiv.org/abs/1611.07004`.

[13] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining Explanations: An Overview of Interpretability of Machine Learning," May 2018. [Online]. Available: `https://arxiv.org/abs/1806.00069`.