AlgoNet: $C^\infty$ Smooth Algorithmic Neural Networks

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 and data structures into the topology of neural networks. A forward AlgoNet includes algorithmic layers into existing architectures while a backward AlgoNet can solve inverse problems without or with only weak supervision. In addition, we present the algonet package, a PyTorch based library that includes, inter alia, a smooth evaluated programming language, a smooth 3D mesh renderer, and smooth sorting algorithms.

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 and data structures into the topology of neural networks. For that, approximations to basic algorithms have to be designed in a smooth fashion: i.e., the functions representing the algorithms are ideally $C^\infty$ smooth. We do that by designing a pre-programmed neural network that has the structure of a given algorithm. Generally, for end-to-end trainable neural network systems, all components should at least be $C^0$ smooth, i.e., continuous. However, having $C^k$ smooth, i.e., $k$ times differentiable and then still continuous components with $k > 1$ is favorable. This property of higher smoothness allows for more differentiations and thus avoids unexpected behavior of the gradients. Thus, the optimal smoothness would be $C^\infty$ smoothness, and we strive for this property.

We present two versions of AlgoNets: forward and backward AlgoNets. Forward AlgoNets are composed of algorithmic layers that are inserted between layers of a neural network. Specifically, such layers are located either between two successive layers or as an additional residual connection between any two layers. Backward AlgoNets use a simplified smooth inverse of a given problem and are used to provide supervision by enabling round trips of reconstruction and application of an algorithm leading back to an approximation of the input.

In this work, we describe the basic concept of AlgoNets and present some applications. In Section 3.1.1 we start by showing that any algorithm, which can be emulated by a Turing machine, can be approximated by a $C^\infty$ smooth function. In Section 4 we present some algorithmic layers that we designed to solve basic problems. All described algorithmic layers are provided in the algonet package, a PyTorch based library for AlgoNets.

Preprint. Under review.
2 Related Work

In previous work [2], we presented the Reconstructive Adversarial Network (RAN) for unsupervised 3D-reconstruction from images. Some previous works [3]–[5] already used algorithms in neural network architectures. However, these algorithms are only locally differentiable versions of the discrete algorithms. For example, the Sorting-Layer of TensorFlow (\texttt{tf.sort}) sorts its input and is considered differentiable because it also relocates the gradients with respect to the sorting. Thus, in every case where the value of ArgSort changes, there is a discontinuity in the sorting layer. Moreover, the gradient with respect to the order of the sorting is constant zero. Another example is the while loop in TensorFlow (\texttt{tf.while_loop}). This construction executes its content while the discrete loop condition holds. Again, there is no derivative with respect to the condition. Thus, without gradients that are dependent on the algorithm, a neural network has no means of using and controlling the algorithm. Our algorithmic layers provide gradients with respect to the ordering of elements or the condition of a while loop.

Generally, as shown by Friedrichs [6], functions in a finite-dimensional Banach space can be approximated by an $C^\infty$ smooth function. DeMillo \textit{et al.} [7] proved that any program can be modeled by a smooth function. Consecutive works [8]–[10] provided approaches for smoothing programs using, inter alia, Gaussian smoothing [9], [10].

The finite differences method, which was introduced by Lewy \textit{et al.} [11], is an essential tool for finding numerical solutions of partial differential equations. Analog to that, we use finite differences to compute the spatial derivative using the finite differences layer in Sec. 4.3.

3 AlgoNet

In this Section, we present the method of algorithmic neural networks in greater detail. For that, first, we describe how to obtain smooth approximations of algorithms and then show that such an approximation can be found for any Turing computable algorithm. Consecutively, we explain the forward and backward AlgoNet.

3.1 Smooth algorithms

To design a smooth algorithm, 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. For example, an if-statement can be replaced by a sigmoid-weighted sum of both cases.

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

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

Here, the sigmoid function (Eq. 1) is a $C^\infty$ smooth replacement for the heaviside step function (Eq. 2), which is equivalent to the if-statement. Alternatively, one could use the $C^1$ smooth step function

$$s_3(x) = \begin{cases} 0 & \text{if } x < 0 \\ 3 \cdot x^2 - 2 \cdot x^3 & \text{if } 0 \leq x \leq 1 \\ 1 & \text{else} \end{cases}$$

Another example is the max-operator, which can be replaced by the SoftMax operator.

$$\text{SoftMax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

As we will explain in Section 4.4, we propose the weighted SoftMax with the name $m$SoftMax to allow a list that is fed to the SoftMax operator to have smooth weights indicating which elements are
\[ \text{wSoftMax}_i(x, w) = \frac{\exp(x_i) \cdot w_i}{\sum_{i=0}^{\|w\|-1} \exp(x_i) \cdot w_i} \quad (5) \]

After designing an algorithmic layer, we can use it to build a neural network as shall be described in Sections 3.2 and 3.3.

### 3.1.1 Smooth WHILE-Programs

In this Section, we present a smooth approximation to a very simple programming language based on the WHILE-language by Uwe Schöning [12]. The WHILE-language is Turing-complete, and for variables (var) its grammar can be defined as follows:

\[
\text{prog} = \text{WHILE var }\not= 0 \text{ DO } \text{prog} \text{ END} \\
| \text{prog prog} \\
| \text{var} := \text{var} \quad // \text{ left and right var unequal} \\
| \text{var} := \text{var} + 1 \quad // \text{ left and right var equal} \\
| \text{var} := \text{var} - 1 \quad // \text{ left and right var equal}
\]

Here, \( \text{var} \in \{ x_n \mid n \in \mathbb{N}_0 \} \) and while \( x_0 \) is the output, \( x_1 \ldots x_n \) where \( n \in \mathbb{N}_0 \) are the inputs. Provided the Church’s thesis holds, this language covers all effectively calculable functions. We generate the approximation to this language by executing all statement only to the extent of its probability. I.e., we keep track of a probability \( p \), indicating whether the current statement is still executed. Initially, \( p = 1 \). In the body of a while loop, the probability for an execution is

\[
p_{\text{exec}}(x) = p_{\text{prev}} \cdot \frac{(e^{sx} - 1)^2}{e^{2sx} + 1} = p_{\text{prev}} \cdot (1 - \text{sech}(sx)) \quad (6)
\]

Here, \( x \) is the value of the current variable, \( s \) is the steepness, and \( p_{\text{prev}} \) is the probability of the previous execution of the body or the probability of the execution of the loop itself, respectively. To apply the probabilities on the allocation, increment and decrement operators, we redefine them as follows:

\[
\begin{align*}
\text{x0} & := \text{x1} & \rightarrow & \text{x0} := p \cdot \text{x1} + (1 - p) \cdot \text{x0} \quad (7) \\
\text{x0} & := \text{x0} + 1 & \rightarrow & \text{x0} := \text{x0} + p \quad (8) \\
\text{x0} & := \text{x0} - 1 & \rightarrow & \text{x0} := \text{x0} - p \quad (9)
\end{align*}
\]

Based on the smoothness of the hyperbolic secant, our version of the WHILE-language is \( C^\infty \) smooth.

As an experiment, we implemented the multiplication on positive integers, as shown on the left:

\[
\begin{align*}
\text{WHILE x2 }\not= 0 \text{ DO } & | \text{WITH p1 := 1; p1'} := p1 \cdot (1 - \text{sech}(x2)) \text{ DO} \\
\text{x3} & := \text{x1} & | \text{x3} := p1 \cdot \text{x1} + (1 - p1) \cdot \text{x3} \\
\text{WHILE x3 }\not= 0 \text{ DO } & | \text{WITH p2 := p1; p2'} := p2 \cdot (1 - \text{sech}(x3)) \text{ DO} \\
\text{x0} & := \text{x0} + 1 & | \text{x0} := \text{x0} + p2 \\
\text{x3} & := \text{x3} - 1 & | \text{x3} := \text{x3} - p2 \\
\text{END} & | \text{END} \\
\text{x2} & := \text{x2} - 1 & | \text{x2} := \text{x2} - p1 \\
\text{END} & | \text{END}
\end{align*}
\]

Contrary to the discrete implementation, the smooth interpretation (as on the right) can interpolate the result for arbitrary values \( x_1, x_2 \in \mathbb{R}_+ \). It is clear that this example itself has no practical application, but it illustrates how a discrete algorithm can be converted to a smooth approximation. For very high-level algorithms, a manual smooth approximation may vastly outperform this automatic approximation; nevertheless, it can be applied to any algorithm. While the naïve approach assumes a constant maximum number of loop iterations, the while loop can also be stopped if the probability reaches numerically negligible regions.
Figure 1: Multiplication of $x_0$ and $x_1$ implemented as a general multiplication, smooth WHILE-program and WHILE program of $\lfloor x_0 \rfloor$ and $\lfloor x_1 \rfloor$ (without casting to integers, the while program would fail). The height indicates the result of the function. Contrary to the general notion, the color indicates not the values but the analytic gradient of the function.

3.2 Forward AlgoNet

The AlgoNet can be classified into two flavors, the forward and the backward AlgoNet. To create a forward AlgoNet, we use algorithmic layers and insert them into a neural network. By doing so, the neural network may or may not find a better local minimum by additionally employing the given algorithm. We do so by using one of the following options for each algorithmic layer:

- Insert between two consecutive layers (Fig. 2a).
- Insert between two consecutive layers and also skip the algorithmic layer (Fig. 2b).
- Add a residual connection between any two layers and apply the algorithmic layer on the residual part (Fig. 2c).

Generally, algorithmic layers have no trainable weights. Regarding the accuracy, the output of smooth WHILE-programs differs from the discrete WHILE-programs by a small factor and offset. While one could counter this issue by designing more precisely approximating smooth functions, we think that it is not only easier but may also yield better results to include trainable weights and biases to the algorithmic layer. For that, one should regularize these weights and biases should to be close to one and zero, respectively. These parameters could be trained on a dataset of integer input/output pairs of the respective discrete WHILE-program to fit the algorithmic layer. Moreover, the algorithmic layer could also be trained to fit the surrounding layers better.

3.3 Backward AlgoNet—RAN

While forward AlgoNets can use arbitrary smooth algorithms—of course a to the problem-related algorithm might perform better—backward AlgoNets use an algorithm that solves the inverse of the given problem. I.e., a smooth renderer for 3D-reconstruction, a smooth iterated function system (IFS) for solving the inverse-problem of IFSs, and a smooth text-to-speech synthesizer for speech recognition. While backward AlgoNets could be used in supervised settings, they are designed for unsupervised or weakly supervised solving of inverse-problems. Their general structure is the following:

$$\text{Input } (\in A) \rightarrow \text{Reconstructor} \rightarrow \text{Goal} \rightarrow \text{smooth inverse} \rightarrow \text{Smooth v. of input } (\in B)$$
Figure 3: 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, $a2b$.

This structure is similar to auto-encoders and the encoder-renderer architecture presented by Che et al. [4]. Such an architecture, however, cannot directly be trained since there is a domain shift between the input domain $A$ and the smooth output domain $B$. While one could use domain translators ($a2b$ and $b2a$) to translate between these two domains, a training with three consecutive components, of which the middle one is highly restrictive, is barely possible, we introduce a novel training schema for these components: the Reconstructive Adversarial Network (RAN). To summarize, we have four components, three of which are trainable: a reconstructor, a smooth inverse, and the domain translators $a2b$ and $b2a$. In addition to these four components, we introduce a discriminator and use an adversarial training scheme related to the training of GANs.

Since we initially neither have a trained reconstructor nor a trained domain translator, this problem constitutes 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. 3 depicts the structure of the RAN, which allows for such a coevolutionary training scheme.

3.3.1 Training

In the following, we will explain our novel 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. We then alternately train this scheme with the possible paths to obtain values of spaces $A$ and $B$, respectively. The applied training scheme for the RAN depicted in greater detail in our previous work, where we successfully used the RAN for 3D unsupervised 3D reconstruction [2]. In the following, we will present our losses in detail.

Our optimization of $R$, $a2b$, $b2a$, and $D$ involves adversarial losses, cycle-consistency losses, and regularization losses. Specifically, we solve the following optimization:

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

where $\alpha_i$ is a weight in $[0, 1]$ and $\mathcal{L}$, $\mathcal{L}_i$, and $\mathcal{L}_{\text{reg}}$ shall be defined below.

We define $b', b'' \in B$ and $a', a'' \in A$ in dependency of $a \in A$ according to Fig. 3 as

$$b' = a2b(a) \quad a' = b2a(b')$$
$$b'' = \text{Inv} \circ R(a) \quad a'' = b2a(b'')$$
With that, our losses are

\[ \mathcal{L}_1 = \mathbb{E}_{a \sim A} [\log D(a, b')] + \mathbb{E}_{a \sim A} [\log(1 - D(a, b'))] + \mathbb{E}_{a \sim A} [\|b'' - b'\|_1] \]

\[ \mathcal{L}_2 = \mathbb{E}_{a \sim A} [\log D(a, b'')] + \mathbb{E}_{a \sim A} [\log(1 - D(a'', b''))] + \mathbb{E}_{a \sim A} [\|a'' - a\|_1] \]

\[ \mathcal{L}_3 = \mathbb{E}_{a \sim A} [\log D(a, b')] + \mathbb{E}_{a \sim A} [\log(1 - D(a', b'))] + \mathbb{E}_{a \sim A} [\|a' - a\|_1] + \mathbb{E}_{a \sim A} [\|b'' - b'\|_1] \]

\[ \mathcal{L}_4 = \mathbb{E}_{a \sim A} [\log D(a, b'')] + \mathbb{E}_{a \sim A} [\log(1 - D(a'', b''))] + \mathbb{E}_{a \sim A} [\|a' - a\|_1] + \mathbb{E}_{a \sim A} [\|b'' - b'\|_1] \]

\[ \mathcal{L}_5 = \mathbb{E}_{a \sim A} [\log D(a, b')] + \mathbb{E}_{a \sim A} [\log(1 - D(a', b'))] + \mathbb{E}_{a \sim A} [\|a' - a\|_1] \]

This results in our combined loss of (without weights)

\[ \mathcal{L} = \mathbb{E}_{a \sim A} [\log D(a, b')] + \mathbb{E}_{a \sim A} [\log(1 - D(a, b'))] + \mathbb{E}_{a \sim A} [\|b'' - b'\|_1] + \mathcal{L}_{\text{reg}} \]

\[ + \mathbb{E}_{a \sim A} [\log D(a, b'')] + \mathbb{E}_{a \sim A} [\log(1 - D(a', b'))] + \mathbb{E}_{a \sim A} [\|a' - a\|_1] \]

\[ + \mathbb{E}_{a \sim A} [\log(1 - D(a'', b''))] + \mathbb{E}_{a \sim A} [\|a'' - a\|_1] \]

\[ + \mathbb{E}_{a \sim A} [\log(1 - D(a'', b''))] \]

\[ + \mathbb{E}_{a \sim A} [\log(1 - D(a'', b''))] \]

where \( \mathcal{L}_{\text{reg}} \) are the regularization losses on the reconstruction.

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 \( B \) (\( R + Inv \), \( a2b \))

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 \([13]\) for each of the trainable components \( R \), \( a2b \), \( b2a \), and \( D \).

## 4 Applications

In this section, we present specific AlgoNet-layers. Specifically, we present a smooth sorting algorithm, a smooth median, a finite differences layer, a weighted SoftMax, smooth iterated function systems, and a smooth 3D mesh renderer.

### 4.1 SoftSort

The SoftSort-layer smoothly sorts a tensor along an array of scalars by repeatedly exchanging adjacent elements if necessary. Fig. 1 shows the structure of the SoftSort algorithm. Contrasting our approach, the sorting layer in TensorFlow \([3]\) is not smooth and does not consider gradients with respect to the ordering induced by the sorting.

### 4.2 SoftMedian

The mean is a commonly used measure for reducing tensors, e.g., for normalizing a tensor. While the median is robust against outliers, the mean is sensitive to all data points. This has two effects: firstly, the mean is not the most representative value because it is influenced by outliers; secondly, the derivative of a normalization substantially depends on the positions of outliers. I.e., outliers, which might have accelerated gradients in the first place, can influence all values during a normalization like \( x' := x - \bar{x} \). While one would generally avoid these potentially malicious gradients by cutting the gradients of \( \bar{x} \), this is not adequate changes in \( \bar{x} \) are expected. To reduce the consideration of outliers in a smooth way, we propose the SoftMedian, which comes in two styles, a precise and slower as well as a significantly faster version that only discards only a fixed number of outliers.

The precise version sorts the tensor with SoftSort and takes the middle value(s). For that, it is not necessary to carry out the entire SoftSort, but only a relevant subset of computations has to be performed to compute the middle value.

The much faster variant is to take a SoftMin of the sum of SoftMax and SoftMin, and, if desired, weight the SoftExtrema recursively with the SoftMedian. By that, the influence of the highest and lowest values is reduced.
Figure 4: The structure of SoftSort. Here, the exchanges of adjacent elements are represented by matrices $M_i$. By multiplying these matrices with tensor $a$, we obtain $b$, the sorted version of $a$. By instead multiplying with tensor $t$, we obtain $t'$: $t$ sorted with respect to $a$. For sorting $n$ values, we need $n - 1$ steps for an even $n$ and $n$ steps for an odd $n$; to get a probabilistic coarse sorting, even fewer steps may suffice. $s$ denotes the steepness of the sorting such that for $s \rightarrow \infty$ we obtain a non-smooth sorting and for $s \rightarrow 0$ all resulting values equal the mean of the input tensor.

4.3 Finite differences

The finite differences layer computes the finite differences for one or multiple given dimensions of a tensor. For that, we subtract the tensor from itself shifted by one in the given dimension. Optionally, we normalize the result by shifting the mean to zero and/or add padding to output an equal sized tensor. With that, it is possible to integrate a spatially- or temporal-derivating layer into neural networks.

4.4 WeightedSoftMax

We define the weighted SoftMax as:

$$m\text{SoftMax}_i(x, w) := \exp(x_i) \cdot w_i \over \sum_{i=0}^{\|w\|-1} \exp(x_i) \cdot w_i = \exp(x_i + \log w_i) \over \sum_{i=0}^{\|w\|-1} \exp(x_i + \log w_i)$$ (12)

Accordingly, we define the weighted SoftMin (analogue to SoftMin/SoftMax) as $w\text{SoftMin}(x, w) := m\text{SoftMin}(x, w)$. By that, we enable a smooth selection to apply (in this case) the SoftMax function only to relevant values.

4.5 Smooth Iterated Function Systems

Iterated function systems (IFS) allow the construction of various fractals using only a set of parameters. For example, pictures of plants like the Barnsley’s fern can be generated using only $4 \times 6 = 24$ parameters. Numerous different plants and objects can be represented using IFS. Since they are parametric representations, they can be stored on very small space and easily be adjusted, e.g., for
changing the plants to increase the variety of multiple plants. Finally, there are very fast algorithms to generate an image from IFS.

While IFS provide many advantages, solving the inverse problem of IFS, i.e., finding an IFS representation for any given image, is very hard and still unsolved. Towards solving this inverse problem, we developed a \( C^\infty \) smooth approximation to IFS.

Given a two-dimensional IFS with \( n \) bi-linear functions \( (f_i)_{i \in \{1..n\}} \) \( f_i(x, y) := (x + a_1 + a_2x + a_3y, y + a_4 + a_5x + a_6y) \), we repeatedly randomly select one function \( f \in (f_i)_{i \in \{1..n\}} \) and apply it to an initial position or the proceeding result. We do that process arbitrarily often and plot every intermediate step.

The problematic step in this process is the plotting. We treat this problem using differentiable rasterization, i.e., by computing for each pixel the intensity of a Gaussian at the predicted point. By setting the standard deviation, different levels of details can be optimized. To provide higher consistency, our approach, optionally, performs the random choices in advance using a seed.

While the parameters of IFSs can be optimized to fit an image, a predicting approach is favorable since it is much faster.

4.6 Smooth Renderer

Lastly, we include a \( C^\infty \) smooth 3D mesh renderer to the AlgoNet library. Compared to previous differentiable renderers, this renderer is fully and not just locally differentiable. Moreover, the continuity of the gradient allows for seamless integration into neural networks by avoiding unexpected behaviour altogether. By taking the decision which triangles cover a pixel and which of these triangles is the closest to the camera smoothly, our renderer is \( C^\infty \) smooth. By employing a smooth depth buffer, we allow color handling and shading. We describe our smooth renderer for optimization of meshes and for mesh prediction using the RAN in greater detail in our previous work [2].

5 Discussion and Conclusion

Concluding, in this work, we presented AlgoNets as a new kind of layers for neural networks and developed a \( C^\infty \) Turing complete interpreter. We have implemented the presented layers on top of PyTorch and will publish our AlgoNet library upon publication of this work.

Concurrent to its benefits, some AlgoNets can be computationally very expensive. For example, the rendering layer requires a huge amount of computation, while the run time of the finite-differences layer is almost negligible. On the other hand, the rendering layer can be very powerful if used correctly.

The AlgoNet could also be used in the realm of explainable artificial intelligence [14] by adding residual algorithmic layers into neural networks and then analyzing the neurons of the trained AlgoNet. That is, 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 future work, we could develop a higher level smooth programming language to improve the smooth representations of higher level programming concepts. The similarities of our smooth WHILE-programs to analog as well as quantum computing shall be explored in future work. Another future objective is the exploration of neural networks not with a fixed but instead with a smooth topology.
References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,” in NIPS-W, 2017.

[2] 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.

[3] Martín 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/api_docs/python/tf/while_loop; https://www.tensorflow.org/api_docs/python/tf/sort.

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

[5] 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.

[6] K. O. Friedrichs, “The identity of weak and strong extensions of differential operators,” Transactions of the American Mathematical Society, vol. 55, no. 1, pp. 132–151, 1944.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] F. K. C. R. Lewy H., “ÜBER DIE PARTIELLEN DIFFERENZENGLEICHUNGEN DER MATHEMATISCHEN PHYSIK,” Mathematische Annalen, vol. 100, pp. 32–74, 1928. [Online]. Available: http://eudml.org/doc/159283.

[12] U. Schöning, Theoretische Informatik - kurz gefasst, Spektrum Akademischer Verlag, 2008, ISBN: 9783827448241. [Online]. Available: https://books.google.de/books?id=eFqeJAAAQBAJ.

[13] D. P. Kingma and J. Ba, “Adam: (A) Method for Stochastic Optimization,” CoRR, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980.

[14] 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.