Deep Learning in Scientific Computing
Neural Ordinary Differential Equations

Joshua Aurand **Niall Siegenheim** **Luca Wolfart** **Dana Zimmermann**

1 Introduction

Deep neural networks (DNNs) have become the go-to architecture for many supervised and unsupervised learning tasks in the broad field of computer science.

Research has shown that using so-called skip connections in DNNs improves the performance of image classification tasks. [4] These have been branded as residual networks, or short, ResNets.

Neural Ordinary Differential Equations (Neural ODEs) [1] bridge the fields of DNNs and numerics by interpreting residual networks as discretizations of ordinary differential equations. Therefore, it is possible to use conventional ODE solvers, as studied in the field of numerics, to find the output of a residual network.

The goal of our project is to investigate the performance of Neural ODEs and reproduce some of the results from [1]. In particular, we want to verify the possibility of using Neural ODEs as a drop-in replacement for ResNets.

2 Methods

As introduced in [4], a residual network consisting of m residual blocks can be written as

$$\mathbf{x}_{j+1} = \mathbf{x}_j + F_j(\mathbf{x}_j),\ j = 0, \dots, m \tag{1}$$

where each F_j is a block of multiple neural network layers. The skip connection can be identified as adding the output of the previous block $\mathbf{x}_j$ to the output of the current block $F_j(\mathbf{x}_j)$.

Observing the ordinary differential equation

$$\frac{d\mathbf{x}(t)}{dt} = F(\mathbf{x}(t)) \tag{2}$$

notice that Equation 1 is equivalent to the Explicit Euler discretization of Equation 2, as known from numerics, with timestep size $h = 1$, initial condition $\mathbf{x}_0$, and integration time $t \in [0, n + 1]$.

This observation makes it possible to interpret each layer in the residual network as a timestep when solving an initial value problem.

The authors of [1] argue that subsequently, we may use any common ODE solver to find the output of the residual network. That way, it is possible to exploit the benefits of modern ODE solvers, including adaptive time stepping and automatic switching between different integration methods depending on the stiffness of the problem. A common choice is ODEPACK [6].

Adjoint Method When using self-written implicit or commercial solvers, we cannot calculate the gradients of their outputs with respect to the parameters using standard backpropagation due to the solvers' black-box nature. As these gradients are required to train the model using (stochastic) gradient descent, the authors of [1] suggest using the adjoint sensitivity method. It allows calculating the gradients of outputs of black-box functions by solving an adjoint ODE backwards in time.

Given the parameters θ, the output $\mathbf{x}(t_1)$ of the Neural ODE, and the gradient of the loss with respect to the the output $\frac{\partial L}{\partial \mathbf{x}(t_1)}$, we want to find the gradient of the loss with respect to the input of the Neural ODE $\frac{\partial L}{\partial \mathbf{x}(t_0)}$ (for further backpropagation) and with respect to the parameters of the Neural ODE $\frac{\partial L}{\partial \theta}$ (for optimization).

To that end, we solve the adjoint ODE

$$\begin{aligned}&\frac{d}{dt}\left[\mathbf{x}(t), \frac{\partial L}{\partial \mathbf{x}(t)}, \frac{\partial L}{\partial \theta}\right] \\ &= \left[F(\mathbf{x}(t), t), -\left(\frac{\partial L}{\partial \mathbf{x}(t)}\right)^T \frac{\partial F}{\partial \mathbf{x}(t)}, -\left(\frac{\partial L}{\partial \mathbf{x}(t)}\right)^T \frac{\partial F}{\partial \theta}\right]\end{aligned} \tag{3}$$

with initial condition $\left[\mathbf{x}(t_1), \frac{\partial L}{\partial \mathbf{x}(t_1)}, \mathbf{0}_{|\theta|}\right]$ by integrating backwards in time, from t_1 to t_0. As the solution we find $\left[\mathbf{x}(t_0) \frac{\partial L}{\partial \mathbf{x}(t_0)}, \frac{\partial L}{\partial \theta}\right]$, containing our desired quantities. For a more complete derivation, we refer to [1].

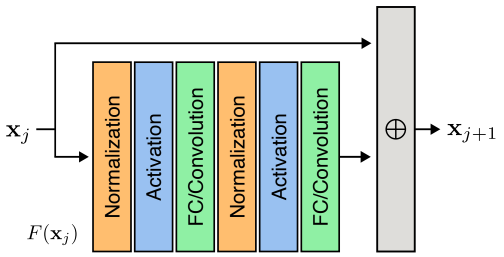

Figure 1: Data flow through a residual block

Residual Blocks For all experiments we use standard residual blocks as proposed in [5]. In particular one such block consists of a sequential application of the following functions: normalization layer, activation function, fully connected (FC) layer, normalization layer, activation function, FC layer. The output of these operations is added to the input to create the residual connection. For image processing tasks the fully connected layers are typically replaced by convolutional layers. Please also refer to Figure 1 for a schematic overview of a residual block.

Neural ODE Blocks A Neural ODE Block has exactly the same architecture as a single residual block with the only difference being that there is no residual connection. In order to create an explicit time dependence the current time is added to the input state. For the convolutional version this is achieved by adding a single channel with constant value equal to the current time.

Adaptive ODE Solvers ODE solvers are often implemented in an adaptive way as an attempt of minimizing the error. For Runge-Kutta methods this can be done by solving the ODE with two methods of different order and using their difference as an approximate error. If the approximate error is too large, Δt is decreased, the result is rejected and the current step is repeated. To speed up computation, Δt is slightly increased when a step is accepted. For more details we refer to [7]. By using an adaptive solver a Neural ODE is, in theory, able to adjust the complexity of the computation to the complexity of the problem since the number of timesteps corresponds to the number of residual blocks in a classic architecture.

3 Results

3.1 Image Classification

MNIST Classification Image classification, a supervised learning task, involves training a machine learning model to assign predefined labels or classes to images using labeled training data. As done in [1], we showcase the performance of a trained ResNet model on the MNIST dataset, which consists of handwritten digits, comparing it to a Neural ODE model trained with a Runge-Kutta solver. All models first use strided convolutions to downsample the input images by a factor of 4. We train ResNets with 1 and 6 residual blocks. For the Runge Kutta network we use a non-adaptive Explicit Euler solver with 6 timesteps. Apart from the difference in the architecture all models are trained with exactly the same training scheme using the Adam optimizer [10] and no data augmentations. The results of this can be found in Table 1. The test errors are slightly worse than what is reported in [1] and in our results the Neural ODE actually performed better than the ResNet with 6 blocks. These discrepancies might be due to differences in the training schemes and different ODE solvers used. Since the exact training information given in [1] is not detailed, we cannot say what exactly those differences are. However, the results do indeed verify that a Neural ODE achieves similar performance to a ResNet.

Model	Test Error	# Params
ResNet-1	0.51%	0.15M
ResNet-6	0.58%	0.51M
RK-Net	0.52%	0.15M

Table 1: MNIST comparison of using a ResNet and Neural ODEs

Different ODE Solvers We also compare various ODE solvers for the MNIST Training. Each model is trained for 100 epochs on the MNIST dataset. We also report the VRAM consumption to compare the memory costs of the different ODE solvers. The results can be found in Table 2. Surprisingly, the lowest order method achieves the highest accuracy, though only by a small margin. Higher order methods only lead to significantly longer training times and higher memory consumption without any benefits. Even the adaptive method is not able to beat the Explicit Euler integrator. Both the implicit methods we test also perform worse than the explicit methods. They have a significantly worse test error with a longer training time. Memory consumption is slightly lower since the standard backpropagation for gradient computation is replaced by the adjoint method. However, the memory savings are rather small since we have to solve an implicit equation for each integration step, which is costly in itself (the implicit equations were solved using the L-BFGS optimizer).

We also tried using implicit solvers implemented in the Scipy package [17]. None of them were usable in practice due to low speed, only running on the CPU, and prohibitive memory costs. With 32 GB RAM we are only able to run the MNIST training with a batchsize of 1 and a training iteration for a single sample already takes several minutes. This makes training practically impossible, hence we cannot include results for those integrators.

CIFAR-10 Classification Additionally, we repeat the comparison on the CIFAR-10 dataset, consisting of 60000 color images belonging to 10 different classes. The results are shown in Table 3. The same models are used as described in Section 3.1 and, again, the same training scheme using the Adam optimizer is used for all models. However, this time the data used for training is augmented by randomly flipping the images horizontally as well as rotating them. Similar to the MNIST comparison, the Neural ODE performs slightly better than the ResNet-6 but slightly worse than the ResNet-1. It can be seen that using the CIFAR-10 dataset leads to a higher test error for both models, which is to be expected since the classification in this case is considerably harder.

Model	Test Error	# Params
ResNet-1	19.21%	0.15M
ResNet-6	20.10%	0.51M
RK-Net	19.59%	0.15M

Table 3: CIFAR-10 comparison of using a ResNet and Neural ODEs

3.2 Image Segmentation

Image segmentation is the task of assigning every pixel of an image to a class, out of a set of classes. In others words, the image is subdivided in the various objects which compose it.

A common approach to implement segmentation is by using a UNet [13]. This neural network architecture consists of a contracting path, which applies several convolutions one after the other, to extract contextual information, and of an expanding path, which uses the information from the contracting path and localizes it.

As a common modification to this architecture in computer vision tasks, we replace the contracting path by a ResNet, combining the advantages of UNets and ResNets. Then, to solve the same task

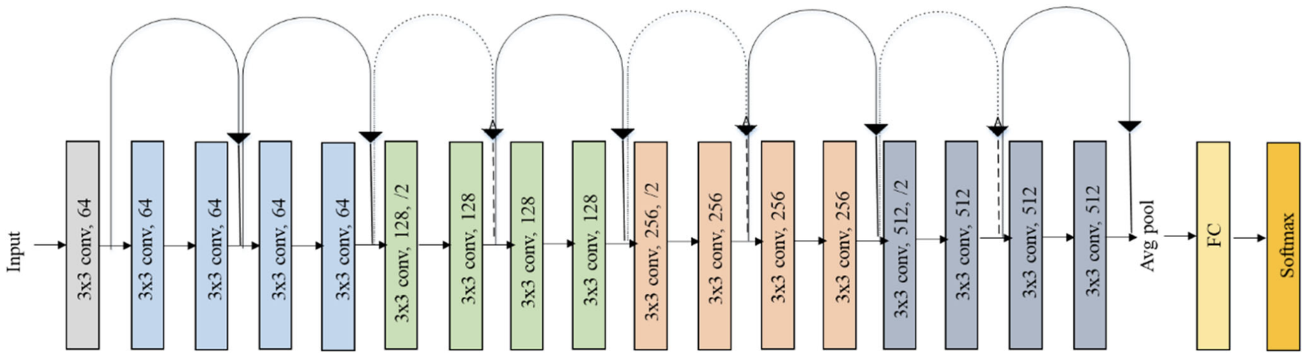

Figure 2: ResNet-18 architecture [11]

Table 2: Comparing Different Runge-Kutta Solvers for MNIST Training. The training was done on an NVIDIA RTX 2080 TI.

Solver	Direct/Adjoint	Adaptive	Order	Test Error	Training Time	VRAM Consumption
Explicit Euler	Direct	✗	1	0.52%	9m	5.0 Gb
Explicit Midpoint	Direct	✗	2	0.54%	14m	5.1 Gb
Fehlberg 4	Direct	✗	4	0.54%	42m	5.2 Gb
Fehlberg 5	Direct	✗	5	0.56%	40m	5.2 Gb
Fehlberg 4-5	Direct	✓	5	0.54%	48m	6.0 Gb
Implicit Euler	Adjoint	✗	1	1.55%	21m	4.8 Gb
Implicit Midpoint	Adjoint	✗	2	1.42%	20m	4.9 Gb

using Neural ODEs, we replace the ResNet contracting path of the UNet with Neural ODEs, as illustrated in section 2.

The specific ResNet used is the ResNet-18 [4], illustrated in Fig. 2, showing the network's layers with the respective number of channels. One can see that the number of channels doubles in every block, which is distinguished by a different color. The arrows indicate skip connections, every block has one skip connection inside of it, and there is one skip connection between blocks.

To use a Neural ODE, all the four blocks were substituted with one Convolutional ODE layer each. For these layers, Explicit Euler was chosen as the integrator. To achieve consistency with the ResNet-18 version, we chose $\Delta t = 0.5$.

The two architectures, ResNet-UNet and NeuralODE-UNet, were benchmarked on a version of the Cityscapes dataset [2] subsampled for the paper [9]. It consists of driving scenes, for which each image has a semantic segmentation. It has 2975 training images and 500 validation images.

Both the architectures were trained for 14 epochs.

Table 4: Results for Image Segmentation

Model	Dice	IoU	# Params
ResNet-UNet	0.851	0.742	18M
NeuralODE-UNet	0.836	0.719	24M

Table 4 shows the results of the benchmark on the Cityscapes dataset. The two chosen metrics were the Dice coefficient, which is defined as $2 * |A \cup B|/(|A| + |B|)$, where the two sets A and B correspond to the set of pixels assigned to one class by the model (e.g. A) or the pixels assigned to one class by the label (e.g. B). The average for all the classes is taken. The other metric is the IoU (Intersection over Union), which is defined as $|A \cap B|/|A \cup B|$. For both metrics a higher value is better since it corresponds to a higher overlap between prediction and label. The ResNet-UNet outperforms the NeuralODE-UNet, but the results are still comparable.

3.3 Normalizing Flows

Normalizing Flows (NF) are a type of generative model used for distribution estimation that rely on a latent distribution $p_z(z)$ which is transformed to the unknown data distribution p_{data} by means of an invertible transformation f. In particular, $f(z) = x \sim p_x(x)$, where $z \sim p_z(z)$ and $p_x(x)$ aims to match the data distribution. The latent distribution is chosen as a simple distribution that allows direct sampling, for instance a Uniform or a Normal distribution. By sampling according to p_z and applying f we can sample p_{data}. In addition, a key advantage of NFs is that they allow for explicit density models due to the change of variables formula:

$$\log p_x(x) = \log p_z(z) - \log \left| \det \frac{\partial f}{\partial z} \right| \tag{4}$$

The explicit density model makes training a NF model straightforward since the data likelihood is tractable and can be optimized directly. When designing a NF model special care has to be taken to guarantee the invertibility of f and allow an efficient computation of the Jacobian determinant. As it turns out, we can use a Neural ODE to circumvent these issues. Chen et al. prove a continuous version of the change of variables formula [1]. If z is distributed as $p(z(t))$ and follows $\frac{dz}{dt} = f(z(t), t)$ then the following holds:

$$\frac{\partial \log p(z(t))}{\partial t} = -\text{trace}\left(\frac{df}{dz(t)}\right) \tag{5}$$

Hence we only need to compute a trace instead of a determinant, and due to the ODE dynamics, uniqueness of the ODE solution

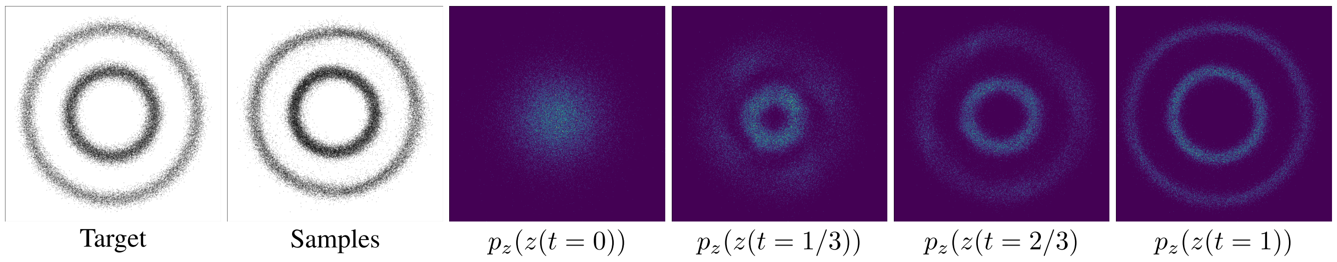

Figure 3: Results for training a CNF on a synthetic dataset consisting of two noisy, concentric circles.

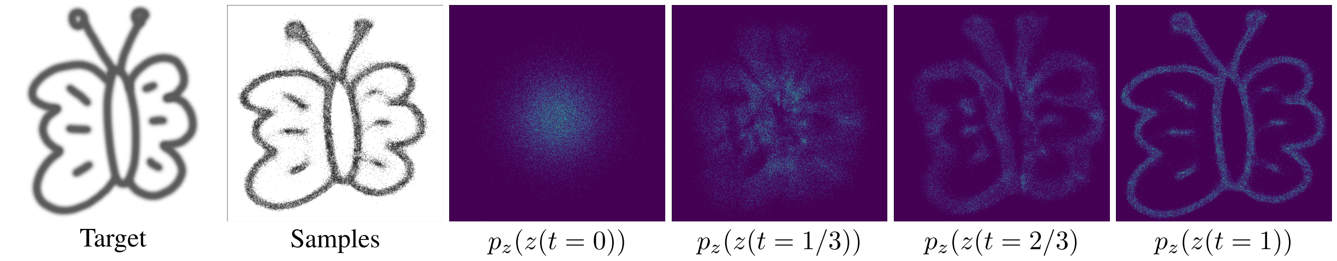

Figure 4: Results for training a CNF on a dataset constructed from a grayscale image.

implies invertibility of the entire transformation. Continuous Normalizing Flows (CNF) rely on Equation 5 to transform the latent distribution to the approximate data distribution. The parametrization of f itself is fairly involved. Since the trace is a linear operation f can be efficiently parametrized as a sum of many different functions $f = \sum_{i=1}^{M} f_i$, which essentially correspond to hidden units in a discrete NF. For these the number of hidden units is typically 1 as the computation of the Jacobian determinant would become prohibitively expensive otherwise due to the $\mathcal{O}(M^3)$ complexity. Hence a CNF model allows for more expressive models with cheaper cost. The functions f_i are parametrized by small neural networks. To introduce a time-dependence, a Hypernetwork-like architecture is used [3]. The Hypernetwork takes the time t as input and predicts the parameters $\theta_i(t)$ of the different networks f_i. The Hypernetwork also predicts gating values $\sigma_i(t) \in (0, 1)$ to control which functions are applied at which time. Note that the $\tanh$ activation function is used, as higher-order derivatives are needed. In total the CNF model can then be described by

$$f(t, z(t)) = \sum_{i=1}^{M} \sigma_i(t) f_i(z(t); \theta_i(t)) \tag{6}$$

During training, the ODE is solved backwards in time to go from the data distribution at $t = 1$ to the latent distribution at $t = 0$. To then sample the CNF after training, the ODE is solved forwards in time from $t = 0$ to $t = 1$.

Figure (3) shows the result of training a CNF on a synthetic dataset consisting of two noisy, concentric circles. The CNF is able to match the bimodal distribution well. It is also possible to visualize how the CNF transforms the latent distribution over time by inspecting the state z at intermediate times. In Figure (4) we also show results of training a CNF on image data. To transform the grayscale image into a valid format, we threshold pixel values and store the (normalized) pixel coordinates of the remaining pixels as a dataset consisting of 2d points. This example shows that a CNF is even able to capture complex distributions with flimsy details. Both models used the adaptive Fehlberg 4-5 method as the ODE solver.

4 Conclusion

In this report, we reproduce the results of Neural ODEs, as presented in [1], and show that this architecture is also applicable to other tasks, such as image segmentation. In particular, our results prove that it is possible to train a network with fewer parameters using Neural ODEs and achieve similar test errors. Furthermore, the architecture of Neural ODEs simplifies the implementation of Normalizing Flows tremendously.

However, it is unclear how this architecture is beneficial in most settings. The original authors argue that this architecture makes it possible to leverage more refined ODE solvers for training a neural network to get better results. It is unclear, however, how this holds in practice, as refined ODE solvers require more computational resources, and we were not able to observe any improvements in test errors by using more sophisticated ODE solvers compared to the simple Explicit Euler solver.

Examining the most-cited publications citing this work [8, 15, 16, 14, 12], excluding review papers, reveals that most authors relate their work to Neural ODEs, but do not use the architecture of incorporating ODE solvers into neural networks as proposed here. This underlines our assumption that this architecture has seen little use. We see the contribution of this paper as of theoretical nature, showing the connection between certain neural network architectures and ordinary differential equations.

5 Individual Contributions

Joshua Aurand was responsible for implementing the base architecture for Neural ODEs and vanilla ResNets and performed the experiments for MNIST and normalizing flows. Niall Siegenheim implemented the adjoint method and experimented with different implicit and black-box solvers. Luca Wolfart conducted the experiments for image segmentation. Dana Zimmermann was responsible for extending the code to perform image classification also on the CIFAR dataset. All group members contributed to this report.

References

[1] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[2] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, Los Alamitos, CA, USA, jun 2016. IEEE Computer Society.

[3] David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks, 2016.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

[6] Alan C. Hindmarsh. ODEPACK: Ordinary differential equation solver library. Astrophysics Source Code Library, record ascl:1905.021, May 2019.

[7] Ralf Hiptmair. Numerical methods for computational science and engineering, chapter 11: Numerical integration. `https://people.math.ethz.ch/~grsam/NumMeth/NumCSE_Lecture_Document.pdf`, 2022.

[8] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[9] P. Isola, J. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5967–5976, Los Alamitos, CA, USA, jul 2017. IEEE Computer Society.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[11] Andrea Loddo, Sara Buttau, and Cecilia Di Ruberto. Deep learning based pipelines for alzheimer's disease diagnosis: A comparative study and a novel deep-ensemble method. *Computers in Biology and Medicine*, 141:105032, 2022.

[12] Vishal Monga, Yuelong Li, and Yonina C Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Processing Magazine*, 38(2):18–44, 2021.

[13] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.

[14] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in neural information processing systems*, 33:7462–7473, 2020.

[15] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020.

[16] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[17] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.