BioTechMed GRAZ

Learning a Poisson Equation

Verena Christina Horak

Institute for Mathematics
and Scientific Computing
University of Graz, Austria

1 Formulation of the problem

In physics, as in other natural sciences, it is of utmost importance to solve partial differential equations (PDE). Due to the rise of machine learning techniques it seems natural to learn PDEs with neural networks. The focus of this report lies on the following Poisson equation:

$$\begin{aligned} -\operatorname{div}(q\nabla u) &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned}$$

For reasons of simplicity, it is further assumed, that $f = 1$ everywhere and q is a function that takes on values in $[1, 2]$ on Ω. Ω is represented as a 16×16 grid. As it is common in natural sciences as well as in the medical domain (i.e., MRI scans) that the actual u can be measured at least partially, it is assumed that u is known on Ω and the Poisson equation should be solved for q, which might be, for example, a parameter related to a specific type of tissue as seen in an MRI scan.

1.1 Generation of training data

One of the first things that has to be considered is the generation of training data. For this purpose, a large number of (u, q)-pairs were generated by a MATLAB code of the structure of Algorithm 1. This code generates the training data **F3** consisting of qs and u pairs where the qs have the form of the beginning of a cosine based series development with a total of 3×3 random variables as coefficients. To ensure that every q only takes on values within the closed interval $[1, 2]$, a final scaling is done. For every such q the Poisson equation is solved for u with a 5-point-stencil. These us constitute the input of the neuronal nets presented in the next section, whereas the qs are the labels (the output) that should be learned. In the same style training data **F8** was generated with 8×8 random variables of coefficients which results in a more complex structure.

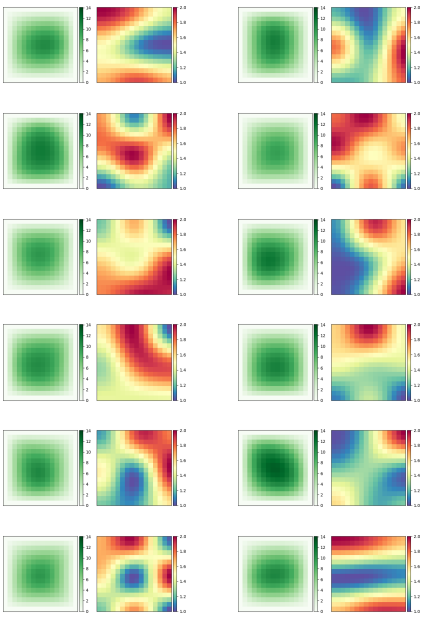


Figure 1: Examples of (u, q)-pairs generated by Algorithm 1.

Furthermore, two different piecewise constant training data were investigated: For **RandSQ2**, always 2×2 pixels get the same random value, whereas **VOR** consists of Voronoi diagrams. Last but not least, **Rand** is the data set where q is a random field.

To give an impression of the difficulty of learning this Poisson equation by only having (u, q)-pairs, Figure 1 should demonstrate how homogeneous the input data and how different the output data (labels) – the corresponding q to a given u – are.

Figure 2 gives on one hand an impression of all investigated types of qs as well as, on the other hand, another impression of how hard this training is with respect to the similarities of the corresponding us.

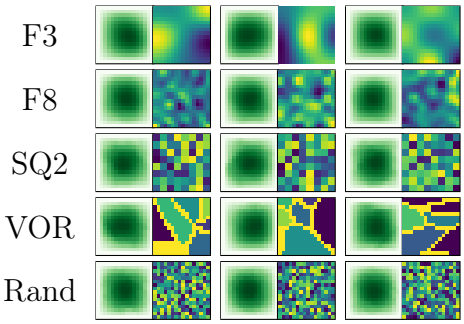


Figure 2: (u, q)-pairs for different types of qs.

Algorithm 1: Generation of a (u,q)-pair for F3

A = 0 and f = 1 everywhere (16×16);

[X,Y] = meshgrid((0:15)/16*pi);

for *k=0:2* **do**
  for *l=0:2* **do**
    A = A + randn(1)*cos(k*X).*cos(l*Y);

q = 1 + (A - min(A(:)))/(max(A(:)) - min(A(:)));

u ← solve Poisson equation for this q;

2 The model evolution

The models described in this chapter were all realized in Keras (version: 2.1.4) [2] on top of Tensorflow (version: 1.13.1) [1] and the results were computed on graphical processing units (GPUs) with the following specifications:

NVIDIA Product	CUDA Cores	Memory in GB
GeForce GTX 750 Ti	640	2
4x Tesla K20m (Mephisto, 2 weeks)	2496	5
GeForce GTX 1050 Ti (1 month)	768	4
Tesla K40c (one run)	2880	12

If not mentioned otherwise, Adam [7] with its suggested standard learning rate of 0.001 was chosen as optimizer, ReLU (Rectified Linear Unit) as (first) activation, glorot_uniform [3] as initializer, and mean squared error (mse) as loss function.

2.1 Dense layers...

To get a first impression of how a sequential neuronal network should be designed in order to be able to learn the Poisson equation as described above, it was very useful to play around with just a few dense layers of different sizes. After some experiments, it seemed to be a good choice using four times as many neurons for these dense layers as the number of input values and only two (for **Dense 1**) or three (for **Dense 2**) such layers together with a dropout layer and the final output layer of same size as the output. These two nets are depicted in Figure 3. In this figure as well as in all other figures describing neuronal networks, the layers are displayed from top to bottom. Furthermore, dense layers are colored in blue and the number of neurons of each layer is printed inside the bar, whereas dropout layers are indicated by green bars as well as the abbreviation "DO" followed by the dropout rate. Dropout layers in general deactivate a certain percentage of the layers neurons and therefore constitute a regularization tool that helps to avoid over-fitting.

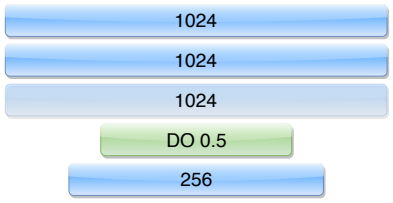


Figure 3: **Dense 1** and **Dense 2**

The next real improvement was reached by introducing a so-called encoder-decoder architecture, i.e., a net for which the dense layers resemble the shape of an hourglass. Whereas the upper triangular seems to guarantee a faster convergence behavior and a lower training and validation loss, the lower triangle helps to propagate the necessary information to all neurons in the output layer. This narrowing of the whole network can be interpreted as a possibility to "focus" on the most important data on which the output is based. The configuration of layers and their associated neurons for **Dense 5**was determined empirically. Using less than 32 neurons in the narrowest layer in the middle of the hourglass resulted in over-fitting. All other changes of the number of layers or number of neurons that were tested just led to worse results than those stated in Figure 4.

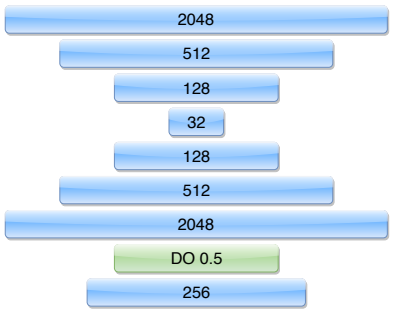


Figure 4: **Dense 5**

2.2 ... and convolutional layers...

Although the results gained from Dense 5 are already remarkable, it might seem strange that the originally 2-dimensional data (a 16×16 grid) is fed in as a 1-dimensional input vector of length 256, although the generation of the data ensures that there is some neighborhood connection.

To benefit from this a priori knowledge, the upper triangle of Dense 5 was replaced by a combination of layers that is well-established in the context of image processing and helps to gather information from locally close neighbors. Typically, two blocks of 32 and 64 convolutional filters of a certain smaller size, i.e., 2×2, are generated, their results are gathered by a so-called max-polling which takes the maximum value of a specified rectangle.

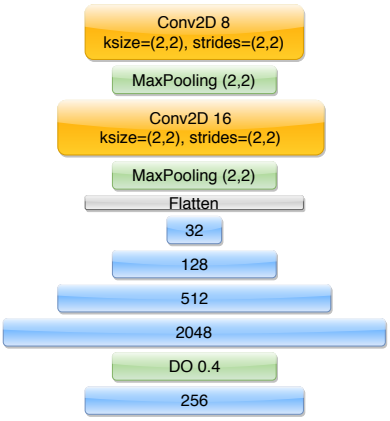


Figure 5: **Conv 8**

Within the context of this report, it was determined empirically that a total of 8 and 16 of such convolutional filters, with a kernel size of 2×2 and a striding of 2 in both directions, gives the best results. Here, a striding of 2 means that not the next neighbors are taken into account but the one after the next. To be able to continue with the lower triangle of Dense 5, the 2-dimensional result gained from the second of these max-pooling layers (green bars) needs to be flattened, which is indicated by a gray bar. The structure of this network **Conv 8** is depicted in Figure 5, which displays the convolutional layers as orange bars. At this point, only the combination of the convolutional block and the lower dense triangle resulted in remarkable improvements compared to Dense 5.

After playing around with this type of network, the impression was gained that Conv 8 is already the best network without changing too much of its structure, since for some weeks no other network could beat its performance. Yet a simple copy&paste action leads to the more complex network **Conv 9**, which is presented in Figure 6. Although the first run of this network was really promising, the loss and validation loss of the second run were worse than those of Dense 5. Although all of the tested networks have slightly different loss functions in two of the runs, the difference between the two runs of Dense 9 indicates that there seems to be some other explanation than just some stochastic effects. And indeed it turned out that the standard learning rate of Adam, namely 0.001, was not appropriate and a change to 0.0005 lead to the expected behavior.

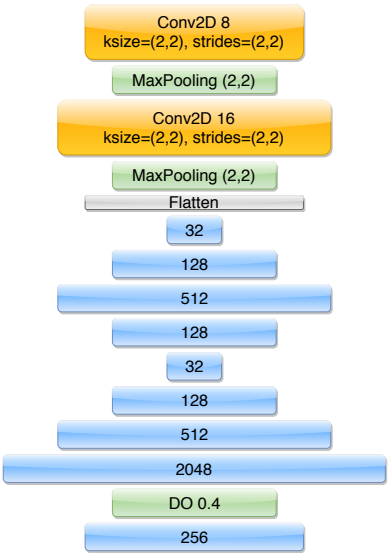


Figure 6: **Conv 9**

In this setting, it is also possible to narrow the second dense layer with 32 neurons to a much smaller number without getting into the situation of over-fitting. An attempt to

see whether this type of net can extract the hidden information that the training data was generated with, 9 random values, resulted in Conv 11, which is more or less as accurate as as Conv 8 but much slower. Conv 10 has the same structure as Conv 9 except that the second 32-layer is narrowed to 9.

2.3 . . . and only convolutional layers. . .

Since the latest developments in the field of neural networks tend to focus on architectures where only convolutional layers are used, it was interesting to investigate the effects of such nets for the above introduced test data. The first idea was to replace the dense layers in the lower part of the hour glass of Conv 8 by so-called transposed convolutional layers, which are sometimes also called deconvolution.

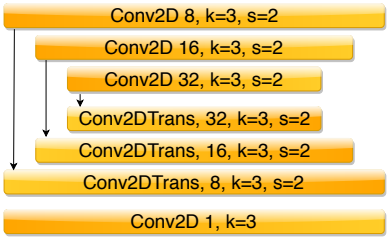


Figure 7: **Unet**-like [9] network: skip-connected autoencoder

Although the latter name might not be the best choice, it perfectly describes the way it is mainly used: "undoing" a convolution with respect to some striding. The next remarks were highly motivated by some recent blog entries:

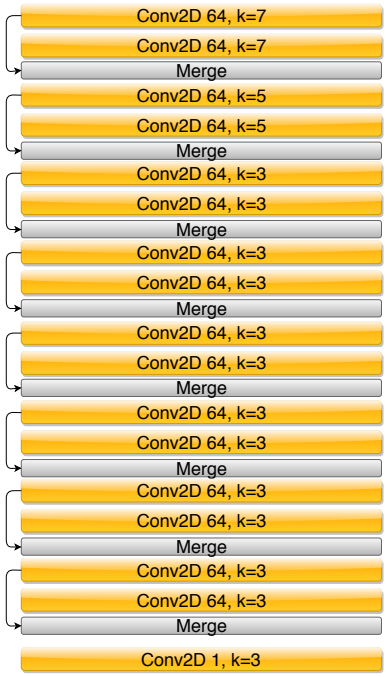


Figure 8: **ResNet**-like network [5]: skip connections every second layer.

To reduce artifacts, it is advised to replace MaxPooling layers by striding of a convolutional layer. Since there are no dense layers, dropout for overfitting avoidance does not make sense any more. In the case of overfitting, one should use normalization and/or regularization instead. Both batch and instance normalization were tested but did not yield better results, which might be a consequence of the fact that the height of the investigated us plays an important role for learning the qs and therefore should not be changed – which would happen if they were normalized. Also weight regularization as well as kernel regularization did not lead to better results. Any combination of L1 and/or L2 regularization showed one of two results: either the learning process was hardened so that the network was not able to even come close to something like a solution, or it was still overfitting. But to be realistic, simple convolution-only networks do not overfit so easily as corresponding ones with dense layers and they generally have nicer loss curves. Nevertheless, if such convolutional networks are too deep

or "strong" in any other sense, how to deal with is yet an open question.

For all of the introduced test data except Rand, reasonable good results were obtained by at least one of the architectures sketched in Figure 7 and 8. For those two types of architectures, the best way of "merging" two layers for the skip connection turned out to be a concatenation. Although other possibilities as adding or averaging can be interpreted as mathematically equivalent, in practice they show different behavior. Another advantage of merging by concatenation is the greater flexibility by means of different numbers of filters as well as the number of different layers to be merged.

To summarize the advantages of convolution-only architectures, one should clearly mention the faster and nicer convergence behavior compared to the networks described above. Also the tendency of overfitting during the training is not as high as for comparable solutions with dense layers, and hence the need of regularization measures does not play such an important role. Skipping is a useful tool for avoiding vanishing gradients and in some sense it is a relatively cheap way of storing and reusing already calculated results for other parts of the network. Obviously, the drawback for this highly complex architecture is that there are more parameters, i.e., there is much more work to do and also more memory used so that it might be hard to train it on a single GPU. And even if this is not a problem it might occur that the teams block each other resulting in slower or not so good results compared to simple GANs.

2.4 ... and finally GANs and cGANs

Another interesting network architecture to investigate, is a so called ***G**enerative **A**dversarial **N**etwork* which was introduced in [4]. Here, the whole network is divided into two subnets which try to be "better" than the other part which is displayed in Figure 9. The typical example which can be found in this context is that somebody (the *generator*) wants to forge something (e.g. money or art) and another person (the *discriminator*) in a shop needs to find out if a presented thing is forged or not. As both have their own interests (a very good forgery vs. the ability to decide if something is forged), they will get better and better and hence after a (actually longer) while the generator should have learned the task to predicting a suitable q for a given u in a hopefully better way than presented before. Unfortunately, it is quite hard to train such GANS which Jonathan

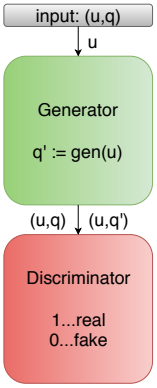


Figure 9: **GAN** structure

Hui[1] summarizes as following:

- Non-convergence: the model parameters oscillate, destabilize and never converge
- Mode collapse: the generator collapses which produces limited varieties of samples
- Diminished gradient: the discriminator gets too successful that the generator gradient vanishes and learns nothing
- Unbalance between the generator and discriminator causing overfitting
- Highly sensitive to the hyperparameter selections.

After carefully testing GANs regarding these points, it was very easy to confirm all of them. Indeed it is very hard to find a constellation for which the basic idea works – imagine exactly one already perfect actor: if the generator is flawless, the discriminator will not have any chance to distinguish fake goods from real ones; if the discriminator is excellent, the forger will not get any idea of what to improve for the next attempt.

Nevertheless, there is an interesting approach to double this constellation and couple it in such a way that it stabilizes itself – at least a bit more than just presented. The basic idea of a so-called *cycle (consistent) generative adversarial network* (cf. [10]) is to train two GANs such that one half tries to master the u to q part, the other half the opposite direction starting with q and predicting u which is displayed in Figure 10. So far so good, the interesting step is now the possibility to cross over to the other half of the network, i.e., starting with u a fake image q can be generated, and by passing this fake q to the other generator we get a (double) faked u which should be identical to the input u. Obviously, to get proper (u, q) and (q, u) pairs, one has to include this idea – the so-called *cycle consistency* – to the overall loss function. If all works well, this complex architecture not even provides the inversion of the original problem, it is often able to balance better than a simple GAN due to the thought that instead of a one-vs-one relation now the four players build teams of two.

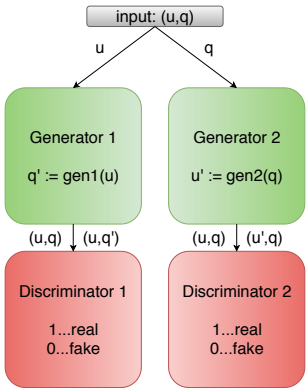


Figure 10: cycle-consistent generative adversarial network (**cycGAN**)

[1] `https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b`

3 Discussion

The results presented are gained from a total of 100000 training examples and 10000 test and 10000 validation examples packed in batches of size 2048 – only adversarial networks need much smaller batch sizes like 8 up to 32. As all the other structures and hyperparameters, the batch size was determined to be the best based on empirical results.

Even though (cyc)GANs are hard to train and require a long training phase, it was worth focusing on them since the other architectures had other drawbacks, e.g. the choice between not learning the idea versus overfitting. Due to the complex structure of GANs, they can modify the outcome of their generators in a more sophisticated way without struggling overfitting.

To get an idea of this different behavior, Figure 11 demonstrates that even single pixels can be changed individually during the learning process. Here, a single pixel in the middle of the image in the first row within the constant greenish area is predicted yellowish for about five epochs when the GAN realizes that this is no good. In the same manner as appearing from "nowhere", this special pixel behavior disappears to "nowhere".

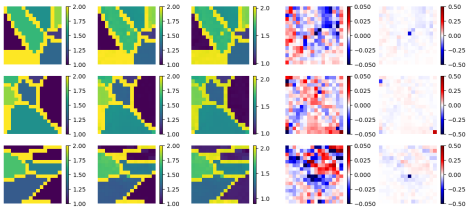


Figure 11: A single pixel changes.

Another important point yielding these good results is the usage of a suitable activation function. As (leaky) ReLU failed for the complex GAN structure, two other functions were investigated: generalized **Swish** [8] and Kristian Bredies' answer to Swish – which I named **Brish**. Whereas the Swish function

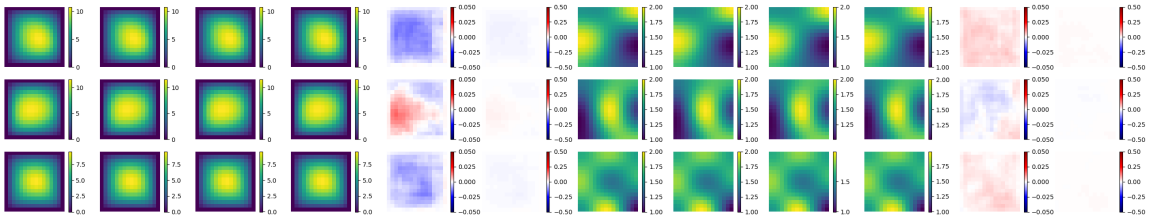
$$f(x) = x \cdot \mathrm{sig}(\beta \cdot x)$$

has only one parameter called β to vary, for Brish

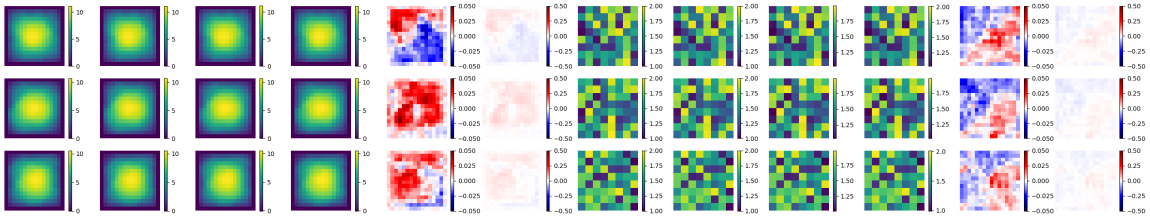
$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \int_{\mathbb{R}} e^{-\frac{y^2}{2\sigma^2}} \cdot (\text{leaky})\mathrm{ReLU}(x-y)\,\mathrm{d}y$$

there are three of them: the Gauß'ian σ and the here hidden slopes for the negative and positive ReLU parts. Both variants result in much better learning behavior of the GANS and right now it is difficult to say which is actually better suited for these problems.

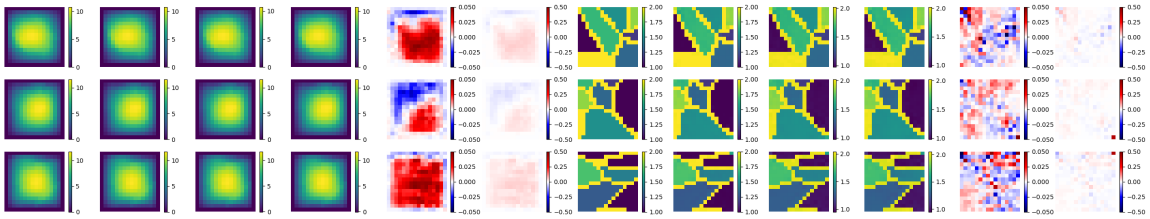
Last but not least, it should be mentioned that for a useful prediction of **Rand** also the boundary of q has to be passed to the generator. Even though this is not really surprising for experts of the theory of inverse PDEs (cf. [6, Ch. 1.2]), it would have been really nice if a neural network would have solved this without knowledge of the boundaries. Nevertheless, the results for the different test sets are displayed in Figure 12.



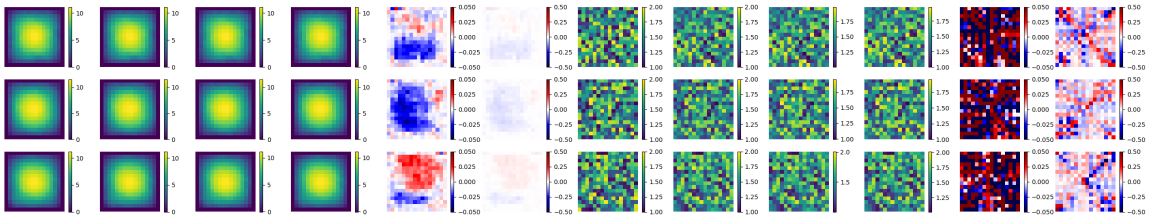
(a) **F3**: loss of 3.7414e-6



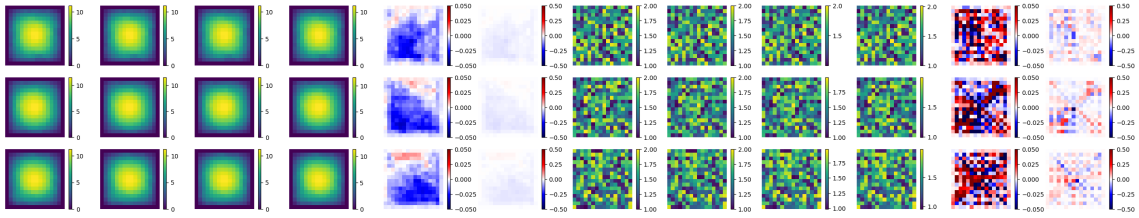
(b) **SQ2**: loss of 1.7942e-5



(c) **VOR**: loss of 3.024e-4



(d) **Rand** without boundary information: loss of 0.01593



(e) **Rand** with boundary information: loss of 1.7711e-3

Figure 12: Loss of the described datasets gained by cycGANs. The first six columns correspond to the us, the other six columns to the qs. Within both halves, the first column displays the original image, the second and third ones the generated version clipped to the according range and unclipped, respectively, the forth to the double faked image (cf. cycle consistency). The last two images display the difference between the first and the third columns in two different scales.

References

[1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. `https://www.tensorflow.org/`, 2015.

[2] F. Chollet et al. Keras. `https://keras.io/`, 2015.

[3] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics. Society for Artificial Intelligence and Statistics*, 2010.

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[6] V. Isakov. *Inverse Problems for Partial Differential Equations*. Springer New York, New York, NY, 1998.

[7] D. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. 2014.

[8] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for Activation Functions. *CoRR*, abs/1710.05941, 2017.

[9] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *CoRR*, abs/1505.04597, 2015.

[10] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.