An LSTM You Can Count On: Learning Randomized Counting With LSTMs

Avi Swartz
avi_swartz@college.harvard.edu

Dimitar Karev
d_karev@college.harvard.edu

Mark Kong
mark_h_kong@college.harvard.edu

Abstract

Artificial neural networks are currently used with great success for approximating functions for which there are no clear rules on how outputs are generated. However, neural networks have not generally been applied to approximating functions that have very clearly defined algorithms, especially functions with randomized algorithms. In this paper, we show that several naïve implementations of LSTMs cannot learn the function that counts the number of occurrences of events in a list of time slots, but that LSTMs can be trained to use a differentiable data structure to do so. We also show that several naïve implementations of LSTMs are unable to learn Morris' algorithm, a simple randomized counting algorithm, unless it is given help in the form of substantial data preprocessing. These results provide an sense of whether current common practices in machine learning can learn algorithms that are easily implemented by hand, especially randomized ones.

1 Introduction

In recent years, artificial neural networks have been successfully used to solve a wide variety of tasks, especially ones for which there is no clear algorithm capable of solving the task. Additionally, artificial neural networks have been shown to be capable of imitating the behavior of important data structures and algorithms [1]. However, it is still unclear whether artificial neural networks can learn to imitate the behavior of randomized algorithms. In this paper, we attempt to develop architectures based off the Long Short-Term Memory (LSTM) network that are capable of learning *Morris' algorithm*, a simple randomized algorithm for counting, as a case study.

A simple way to count events is to have a counter that is incremented every time the event happens. If there are a total of m events, this would take $O(\log m)$ space to store the counter. However, if memory is a concern (e.g. if the number of events is too large), this might be too much. Morris' algorithm [2] allows us to approximate the count with less space. Instead of incrementing the counter y every time an event happens, increment y with probability $1/2^y$ each time an event happens. This makes y a random variable with expected value $E[2^y] = m + 1$, which in turn means that $y \approx \log m$ [3]. Then y only takes $O(\log \log m)$ space to store and still can be used to approximate m.

Previously, Hitron and Parter [4] showed that it is possible to construct stochastic spiking neural networks (SNNs) that implement a modified version of Morris' algorithm to construct an approximate counter. This allowed them to use asymptotically fewer neurons than a deterministic network would need to count the number of events, although the counter only worked with a time window of fixed width t. Furthermore, the parameters for the neurons were preset and not learned, and the size of the network grows with t. So although in this particular case the SNNs were able to utilize randomness

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

to greatly improve the efficiency of a neural network, this setup is not quite suitable for running randomized algorithms in general use.

We will study whether it is possible for an LSTM to be given a source of random noise and learn to use this randomness correctly. We will attempt to train this network to implement Morris' algorithm for any string length with any count. If successful, this would mean that classic artificial neural networks are capable of learning randomized algorithms.

2 Nonrandomized counting with a simple LSTM

We first tried to see how well a simple LSTM could count the number of 1s in a binary sequence.

The general equations for an LSTM as implemented in PyTorch [5] are as follows, quoted directly from the PyTorch documentation [6]:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
\tag{1}
$$

> where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

Because the dimension of the output h_t at time t is $h \times 1$ (where $h = |h_t|$ is the number of hidden units in the LSTM), we added a "combine" matrix W_d with dimensions $1 \times h$ to condense the output into a single scalar. This gives one more equation

$$y_t = W_d h_t \tag{2}$$

where y_t is the scalar output of the LSTM network at time t.

2.1 Training

The inputs to the network were randomly generated strings $x \in \{0,1\}^k$ of some length k, which means that the input x_i in equation (1) was the ith element of string x. The final output of the network was taken as y_{-1}, the condensed output of the LSTM at the final time step. Since the goal is to count the number of ones in the string, which in this case is the same as the sum of the string, we used the loss function

$$(y_{-1} - \sum_i x_i)^2$$

for a single string and the loss over a batch of strings was the average of the losses over each string. Note that the average was not weighted based on the lengths of the strings, even though longer strings have more possible counts and therefore likely greater prediction variance.

We did not do any minibatching, and a gradient step was taken on the loss averaged over every string in the training data set. The loss on the validation set was calculated every 100 gradient steps. The training was terminated once 10 validations had gone by without the validation loss decreasing by more than 0.001. The optimizer used was the Adam optimizer with default PyTorch settings [7].

After training was finished, the network was evaluated on the test set, with the mean square error reported just as for the train set.

2.2 Single length prediction

We first tested whether the neural network was capable of counting the number of 1s in a binary string of fixed length. In order to determine this, we fixed a number k and generated n unique binary strings of length k. The n strings were then partitioned into disjoint train, validate, and test sets and the network was trained as in section 2.1

Upon training, we found that with 3 or more hidden units, the network achieved an average test loss of no more than 0.01 on all string lengths k tested (multiples of 16 up to $k = 128$). This means that the average prediction was no more than $\sqrt{0.01} = 0.1$ away from the true value, which implies that the network would make perfect predictions on a large number of the strings after rounding to the nearest integer.

2.3 Variable length prediction

We next tested whether we could train the neural network to count on strings of variable length. For this case, we fixed a number k and then generated n unique binary strings of length $\leq k$. The n strings were then partitioned into a train, validate, and test set, taking care to ensure that every string length was represented in every data partition, and the network was trained as in section 2.1.

After training, we found that a network with 5 hidden units performed the best with a maximum test error of 0.075 for $k \leq 128$. This corresponds to an average error of less than $\sqrt{0.075} \approx 0.274$, which means that a large number of the predictions would be correct after rounding. Furthermore, since there was a lot of variability in training from different initializations, it seems reasonable that even better performance could be achieved with hyperparameter optimization and trying multiple initializations.

2.4 Separate train and test lengths

From the previous two sections, it seems that the neural network is capable of counting strings it has not seen before as long as it has previously seen strings of the same length. Naturally, the next thing to test is whether it can learn to count in strings longer than any it has trained on. To do this, we fixed a number k and generated n_1 unique binary strings of length $\leq k$ which were split into train and validate. To form the test set, we generated n_2 unique binary strings of length in the range $[k + 1, 2k]$ to guarantee that each test string was longer than all the possible train and validation strings.

Six different versions of the network with different numbers of hidden units were trained as in section 2.1 on strings of length ≤ 64, and the results of evaluating the network on test strings of varying length are shown in the left plot of figure 1. The string in the train set with the largest true count had a count of 46, which is indicated by the red dashed line. It seems that the network learned to predict any value up to the largest value that it had encountered in the train set, but is unable to make predictions that go much higher, even with as many as 100 hidden units.

Mathematically, this observation makes sense. From equation 2, the final output of the network is $y_{-1} = W_d h_{-1}$, and from the Cauchy-Schwarz inequality, we know that

$$|y_{-1}| = |W_d \cdot h_{-1}| \leq ||W_d|| \cdot ||h_{-1}||$$

From equation 1, we know that h_t is the elementwise product of a sigmoid and a hyperbolic tangent, which means that each entry in h_t is in the range $[-1, 1]$ and that $||h_{-1}|| \leq \sqrt{h}$. This gives that

$$|y_{-1}| \leq \sqrt{h}||W_d||$$

and since W_d is fixed during evaluation, there is an upper bound to how big y_{-1} can be on any evaluation string. Since the network would increase $||W_d||$ during training to be large enough to output any sum that it comes across but not have any need to increase it further, it makes sense that the output reaches a maximum close to the maximum value encountered during training.

Because the network has an effective cap on its output, we generated new data to train on and ensured that all the train, validate, and test strings had a count of no more than 32 ones. It seems that even when all the test strings are bounded in the total count of 1s and the network is mathematically capable of predicting the true value, the neural network still does not understand how to deal with strings that are longer than the ones it trained on (see the right plot of figure 1).

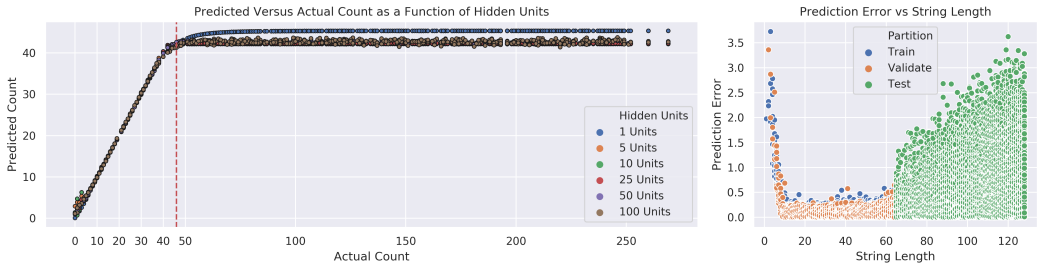

Figure 1: Results from evaluating the network from section 2.4. Left: training the network with 6 different hidden sizes and evaluating on the same large test data. The red dashed line indicates the largest string count present in the train data. Right: Training and evaluating the network with a hidden size of 10 while ensuring that no train or test string had a total count of more than 32. The inaccuracies for string length < 10 are due to there being so few possible strings of such a short length that they did not impact the average loss and were not noticed by the optimizer.

3 Nonrandomized counting with a differentiable counter

Since a vanilla LSTM was unable to learn to count in a generalizable manner, we created a differentiable counter that the LSTM could learn to operate, similar to the differentiable data structures in [1]. The way we did this was by transforming equation (2) by adding another variable $r_t = \sigma(y_t) \in [0, 1]$. We can view r_t as the probability of incrementing a counter r at time t, with the hope that the neural network will learn to set $r_t = 0$ when $x_t = 0$ and $r_t = 1$ when $x_t = 1$.

We define the final value of the counter to be $r = \sum_t r_t$. Because we allow partial increments of the counter (e.g. incrementing the counter by 0.3 when $r_t = 0.3$), r is differentiable and the neural network can learn to optimize the loss

$$(r - \sum_t x_t)^2$$

These networks were also trained as in section 2.1.

3.1 Counting on a binary string

The network was trained on binary strings of length ≤ 64 and tested on binary strings of length in the range $[65, 128]$ with an average test loss of 0.0009. Since this corresponds to an average prediction error of less than $\sqrt{0.0009} = 0.03$, the network is clearly able to count the number of 1s in a binary string.

3.2 Counting on an integer string

However, because when counting the number of 1s in a binary string the end goal is to have $r_t = x_t$, there is a chance that the task is too simple and the network might not be able to manage in cases when x_t is more expansive than just $x_t \in \{0, 1\}$. To make sure that the network is actually able to count occurrences beyond simple 0 and 1 cases, x was changed to be a string of integers in the range $\{-10, \cdots, 10\}$, and the network's goal was to count the number of integers ≥ 0.

The network was trained on such strings of length ≤ 64 and tested on integer strings of length in the range $[65, 128]$ with an average test loss of 0.00142. This corresponds to an average prediction error of less than $\sqrt{0.00142} \approx 0.0376$, making most predictions round to the correct integer.

We evaluated this network on relatively large test strings (length 100 to 10,000) and the network performed perfectly (see figure 2). According to the left plot, the network appears to predict sums accurately for all tested string lengths, and has no indication of losing accuracy towards the end. According to the middle plot, the network seems to have an constant error rate of 1-3 off the true value for every 10,000 digits counted. According to the right plot, the network is incrementing the counter with either probability 0 or 1, and has learned to use the counter deterministically, as desired.

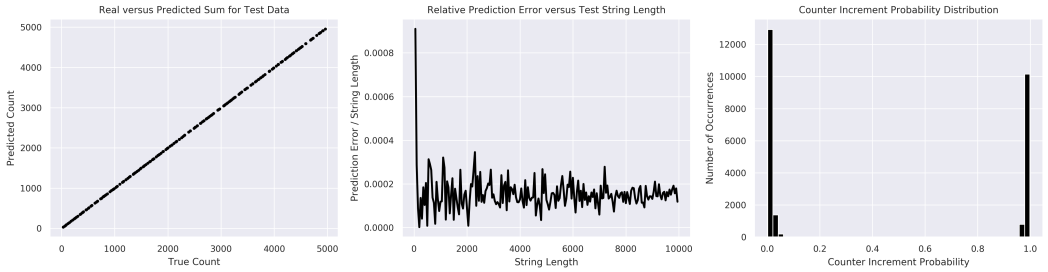

Figure 2: Results from evaluating the network from section 3.2 on large test strings. Left: a comparison between the predicted and true count for each test string. Middle: a plot of the relative error $\frac{|\text{predicted}-\text{true}|}{\text{length}}$ as a function of string length. Right: a distribution of the counter increment probabilities r_t after training.

Taken together, these suggest that the network counts occurrences of events at a consistent accuracy regardless of the string length.

4 Randomized counting

Since neural networks are capable of learning to use a differentiable counter, we next checked whether a neural network could implement Morris' algorithm. We operationally defined this as being able to take a string together with a sequence of random numbers and learn to have the same behavior that a manual implementation of Morris' algorithm would have using the same random numbers.

We updated equations 1 and 2 by changing the input x_i from being the scalar value of the ith entry in the string to being a 2×1 vector created by concatenating the scalar value in the string along with a corresponding random number. All the parameters in the LSTM were then updated to have dimensions compatible with this new vector, and the network was trained to incorporate the random numbers.

4.1 Preprocessed random information

The first step in seeing if the neural network would be capable of learning Morris' algorithm was to give the network heavily preprocessed random information to make it very easy for the network to learn.

At each position in each string in the datasets, a random integer from the interval $[0, 2^y)$ was generated, where y was the current value of Morris' algorithm run on the previous entries in the string and the previously generated random integers.

The result was that after these preprocessings, Morris' algorithm could be effectively run just by counting the number of times the string had a value of 1 and the random integer had a value of 0, since a value of 0 occurs with probability $\frac{1}{2^y}$, with y being the current state of the counter. After training as in section 2.1, the neural network was able to learn how to do this with an average test loss of 0.00149 and an average prediction error of less than $\sqrt{0.00149} \approx 0.04$

4.2 Less processed random information

In the previous preprocessing step, the network was able to learn to generalize. However, that amount of preprocessing essentially reduced the problem to checking whether the random number is 0.

To reduce the amount of preprocessing, we concatenated a single random float in the range $[0, 1)$ with each entry of each string. This still keeps the total difficulty for the network somewhat low by ensuring it never needs to consider more than one random number at a time, but now the network has to learn to increment when the float has a value $\leq \frac{1}{2^y}$ instead of just a constant 0.

Upon implementation, the train and validation losses of the network got stuck at around 0.44 even after the stopping patience was upped to 25 epochs without improvement. This suggests that this task

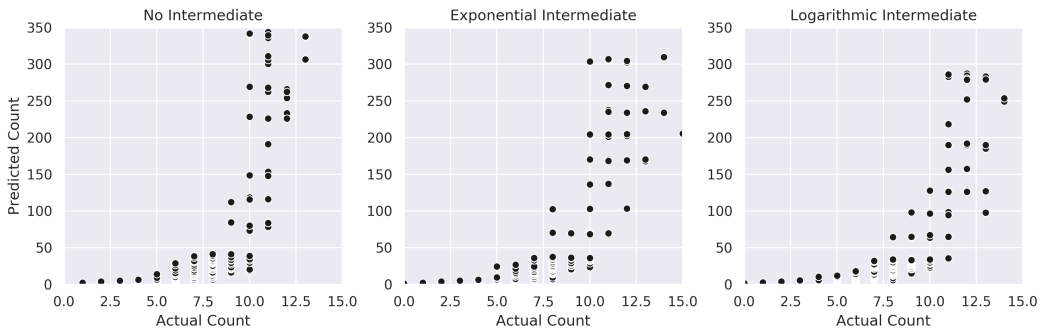

Figure 3: The network in section 4.2 was trained once with random binary floats concatenated directly onto the string (left plot), once with the floats passed through an intermediate LSTM with a exponential activation (middle), and once with the floats passed through an intermediate LSTM with a logarithmic activation (right). After training with a patience of 25 epochs, all three networks were evaluated on test strings of length ≤ 10000 and their predicted versus actual Morris' algorithm counts are plotted here.

was difficult for the network to learn, since in the previous task the network reached a validation loss of 0.001. Nonetheless, since the starting validation loss was 1, the train and validation loss were never more than 0.1 apart, and the network reduced both losses down to 0.44, it seems that the network still managed to learn something.

Even though the network was able to learn something on the similar size validation data, the network performed miserably on testing data (see the left plot of figure 3). Since the network needs to be able to determine when a random float has a value $b \leq \frac{1}{2^y}$ (or equivalently when $y \geq -\log_2 b$), the failure could be because the network is finding it difficult to approximate the exponential or logarithmic functions.

In order to help out the network by providing it with the exponential and logarithmic functions, the network was given an additional LSTM with a logarithmic activation. The additional LSTM mapped the sequence of random floats b_i to an output sequence of values $\log(b_i'^2)$ (the square is to avoid taking the logarithm of a negative number), which was concatenated to the string x_i. The hope was that the additional LSTM would learn to do something with the sequence of random floats (perhaps nothing, if nothing is needed), and then having the logarithm provided would remove the need for the network to learn how to take the logarithm function. Another version of the network was trained with an exponential activation on the LSTM, just in case having an exponential function would prove to be useful.

As seen in figure 3, even with the help of the intermediates to provide difficult to approximate functions, the networks still failed to process large test strings correctly. This means that even with the major assistance of having a logarithm provided to the network, the network was still not able to learn that it should compare y to $\log b$.

4.3 Unprocessed random information

In the most general attempt at learning, we wanted to give the neural network an excess amount of random bits from $\{0, 1\}$ to accompany each string. The expectation was that the neural network would learn to read as many random bits as necessary to sample a $\frac{1}{2^y}$ probability for running Morris' algorithm.

The manual implementation of Morris' algorithm at this point would be as follows: to simulate a draw of probability $\frac{1}{2^y}$, we sample y random $\{0, 1\}$ bits and see if they are all 0. If they are, we would increment the counter and do nothing otherwise.

This requires no preprocessing of the random bits at all, and the question becomes whether a neural network could learn to do this. While this may seem more conceptually difficult than the previous case, it is technically simpler as the network only has to sum y bits instead of taking a log or exponent, and it is feasible that the network could succeed here where it failed in the previous case.

Figure 4: Results from evaluating the network in section 4.3. The left figure shows the prediction accuracy of the train, validate, and test sets as a function of string count. The clouds are kernel density estimators created using the Seaborn `kdeplot` command on default parameters [8]. Perfect accuracy is the black line. The right figure shows the prediction accuracy (lower loss is higher accuracy) as a function of string length. The colors have the same meaning across both plots.

The network is constructed as follows: there are three LSTMs: `count`, `bits`, and `switch`, each of which has a certain number of hidden units and outputs to a single scalar per time point as in equations 1 and 2.

The `count` LSTM is designed just as in the one in section 4.1: It takes each entry in the string, concatenates it with a processed random number, and outputs a probability of incrementing the counter. However, instead of having us do the processing, the processing is done by the `bits` LSTM.

The `bits` LSTM is designed just as the one in section 3.1. It takes a set of random bits and outputs a scalar aggregate of those bits.

The `switch` LSTM is also designed as the one in section 3.1. This network will pass through the next unused random bits one by one until its scalar aggregate becomes larger than 1. At that point, it will pass along all the bits it looked at up to the `bits` LSTM.

When behaving correctly, the LSTMs would behave as follows: When the current Morris' algorithm count is y, the `switch` LSTM will just predict a constant value of $\frac{1}{y}$. This means that it would take the next y random bits and give them to `bits`. The `bits` LSTM would then count the number of 1s in the bits given to it, and pass the count up to `count`. The `count` LSTM would then increment the counter if it is given a 0 from `bits`.

When correctly trained, the network would reduce to the network in section 4.1 and would perform Morris' algorithm perfectly. Furthermore, each individual LSTM is learning a task that we have shown learnable on its own. Therefore, it seems reasonable that the network should be able to learn how to perform Morris' algorithm without any data preprocessing necessary.

During training, however, it was observed that the network was able to learn a little but had trouble learning well. The initial validation loss of the network was around 400 and the final loss was around 1, indicating that the network did learn something. However, the network spent a lot of time fluctuating around a validation loss of 1, and even with an increased patience of 25 epochs, the network never made a substantial and permanent improvement beyond that.

In the left plot in figure 4, the train and validation sets are close to the perfect prediction diagonal line, whereas the test set is well above the line. Since the network did well on the train and validation set, this demonstrates that network was able to learn something about the data. However, the network did fail to predict the test set, indicating that it did not learn to generalize to unseen data.

In the right plot in figure 4, the prediction loss steadily increases as the string length increases. This indicates that even though the parts of the network are theoretically suitable for learning Morris' algorithm on unprocessed data, the network was not able to actually learn anything that was suitably general.

5 Conclusion

In conclusion, it seems that both the task of counting events and the task of incorporating random information to simulate randomized algorithms are difficult tasks for neural networks to learn.

It turns out that an LSTM on its own is unable to learn how to count even the number of ones in a binary string, even though it just amounts to taking the sum over the entire string. Instead, we made the network predict a flag at each time point indicating whether the entry at that point should be included in the count and then manually summed over all the flags.

Because the LSTM is now predicting a flag for each entry in the input, it opens the door to the possibility of having more complicated rules governing the flags. In particular, it might be possible for the LSTM to learn Morris' algorithm, a randomized algorithm where a flag is generated with probability inversely proportional to the exponent of the number of already generated flags.

However, the LSTM proved unable to learn the desired behavior, except in the case where the random information provided to the network was preprocessed to make it a rather simple decision: include the element in the count if the random bit equals 0 and don't otherwise. As soon as the network was expected to learn a decision rule about the random bits that changed depending on context, the network failed to learn.

The techniques used in this paper are generally reflective of common machine learning practices. Since our neural networks were unable to simulate randomized algorithms without a large amount of help, this suggests that more research in this topic is needed to push machine learning to be able to handle randomized algorithms.

Code

The notebooks containing our code are located at
`https://github.com/azswartz/Randomized-LSTM-Counting`

References

[1] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to Transduce with Unbounded Memory. *arXiv:1506.02516 [cs]*, November 2015. URL `http://arxiv.org/abs/1506.02516`. arXiv: 1506.02516.

[2] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[3] Piotr Indyk and Jelani Nelson. Sketching algorithms for big data: Lecture 1 Notes, August 2017. URL `https://www.sketchingbigdata.org/fall17/lec/lec1.pdf`.

[4] Yael Hitron and Merav Parter. Counting to ten with two fingers: Compressed counting with spiking neurons. *arXiv preprint arXiv:1902.10369*, 2019.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[6] Torch Contributors. Source code for torch.nn.modules.rnn. URL `https://pytorch.org/docs/stable/_modules/torch/nn/modules/rnn.html#LSTM`.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[8] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesbeck, Antony Lee, and Adel Qalieh. mwaskom/seaborn: v0.8.1 (september 2017), September 2017. URL `https://doi.org/10.5281/zenodo.883859`.