ECE 509 Final Project: Cryptography by Adversarial Training of Neural Networks

Cody Gunton and Joshua Mack

I. Introduction

In the recent paper [1], the authors Abadi and Andersen introduce a novel form of cryptography. They do not describe new cryptographic protocols as such, but rather introduce a method for producing a cryptosystem synthesized by adversarial training of neural networks.

A summary of their basic idea is as follows. Three neural networks are constructed, which we will call Alice, Bob and Eve. Alice and Bob have a shared goal of communicating secretly in the presence of Eve, who can see whatever information Alice and Bob can exchange. Each time they communicate, Alice takes some input plaintext and processes it through her internal layers to produce a ciphertext. Bob then processes that through his layers. Eve also processes the same ciphertext through her layers. At the outset, there is no reason that the output obtained by Bob or Eve should be close to the original plaintext. In order to create a functional cryptosystem, the neural networks are subjected to adversarial training, where Alice and Bob's layers are tuned to make Bob's output close to the plaintext and to make Eve's far from the plaintext, while Eve's layers are tuned to make her output as close to the plaintext. Hence, with each iteration, Alice and Bob improve at communicating securely, while Eve improves at breaking their encryption, and these two improvements drive each other in a sort of feedback loop.

The first goal of this project was to implement the procedure just described to produce cryptographically trained neural networks (henceforth, CTNN's). The reasons for doing this were three-fold. First, and most simply, there is value in verifying the work of others. Second, the two implementations of Abadi-Andersen-style neural cryptography that could be found on the Internet when we began our work did not produce what seemed like sensible results to us. The TensorFlow implementation [4] we found gave us issues in the convergence of error to a reasonable value and the Theano implementation [3] we found gave us issues with running out of memory while training. We should note that Abadi and Andersen have not yet published their code. The third reason is related to the second goal for our project; we wanted to asses the *strength* of the cryptography produced by the neural networks, so we needed an implementation we understood very well. It should be noted that Abadi and Andersen do not provide any measure of the strength of the cryptography, so we really did not know what to expect.

In the end, we were able to train neural networks in cryptography. With relatively little training (training completed in a few hours on an Nvidia GTX 1070), our Alice and Bob could communicate with virtually no errors — with an average bit error rate of roughly $0.5*10^{-8}$ — while Eve could could not successfully break their encryption thanks to a near 50% bit error rate. For analysis of the strength of their cryptography, we focused on two questions. First: is there any bias in the space of ciphertexts? It is a general principle of cryptography that asymmetries can be exploited, so we would like to understand the nature of the space of ciphertexts. For this we employed principal component analysis and found that there is in fact a noteworthy amount of asymmetry in the space of ciphertexts. Our second assessment was of the strength of the cryptography against differential attacks, i.e., attacks exploiting similarities in ciphertexts produces from similar plaintexts.

The report is structured as followed. We provide a brief section describing relevant concepts from machine learning in the particular context of our problem. We then describe our implementation, which is essentially the implementation of [1]. We then provide our analysis of the ciphertexts produced by our networks. We end with a discussion of issues deserving of further study.

II. Background on Neural Networks

A neural network is essentially a collection of operations performed in stages on an input, each stage being referred to as a "layer". The operations involve some parameters that are tuned, or "trained", as one commonly says in this context, by a gradient descent procedure, in an attempt to drive the network toward performing

whatever task is required. The most common use of neural networks is in classification problems, where the training involves telling the network whether it correctly classified some input or not, so that later on the network will be able to classify "unlabeled data".

Our networks closely follow the design suggested in [1]. Their design choices were driven by a mix of practicality and a sense of the basic structure the networks would need to encrypt information. Two fundamental properties of good cryptosystems are confusion and diffusion. These properties are achieved in many symmetric ciphers by combining operations that permute blocks of text and mix them with secret keys. With this in mind, Abadi and Andersen suggest networks with both fully-connected layers and convolutional layers.

In a fully-connected layer, every output is a function of all of the preceding inputs. In fact, a fully-connected layer is nothing more than a matrix multiplication followed by a nonlinear function applied to each output, the use of nonlinear functions being to allow for the modelling of nonlinear response variables. Such layers have many variables, and hence can be costly to train. One is included, however, so that Alice would have total freedom to permute the inputs. In fact, the fully-connected also creates a great deal of confusion, since its output is not even a string of bits, but rather a collection of floating point numbers that are linear combinations of the entries in the input string.

After the fully-connected layer, Abadi and Andersen suggest three convolutional. In a convolutional layer, a "filter", which really is a vector of floats and which, in practice, is small relative to the input size, is slid across the input. In each position, the filter and some portion of the input are combined by taking a dot product; see Figure 1 for an illustration of this concept. The "stride" with which the filter is applied is 1 if the filter lies over every position of the input (perhaps using some padding to make this possible). The stride is 2 if the filter visits every other possible position, and so on. Following Abadi and Andersen, after a fully-connected layer, we process the data through three convolutional layers with filters of widths 4, 2, 1, 1, with strides 1, 2, 1 and 1, respectively. Note that, in the stride-2 layer, an input of length 40 is compressed to a length of 20. We note also that each layer really can consist of multiple parallel filters, the number being the "depth" of the layer. Our depths were 2, 4, 4 and 1, respectively,

The convolutional layers after the fully-connected layer provide an additional degree of confusion and diffusion, but are less costly to train, given that they each

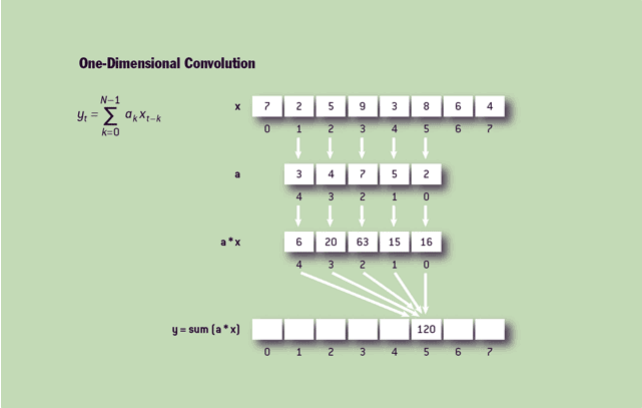


Fig. 1. Source: *Computing without Processors* by Satnam Singh, Computer Architecture, Volume 9, issue 6, June 27, 2011

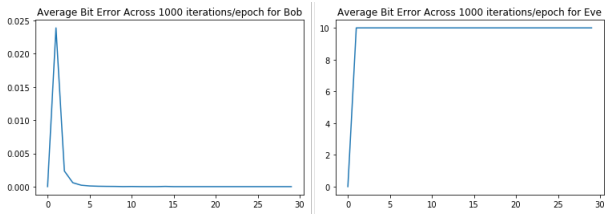


Fig. 2. Bob's and Eve's progress during training

contain far fewer weights than a fully-connected layer. Note also that the convolutional layers do not offer the possibility of a permutation, since the output of a single dot product only depends on a small number of the inputs to the convolutional layer.

III. Implementation Details

We built our neural networks using TensorFlow 1.1 and NumPy 1.12.1 in Python 3.6.1. A summary of the included code is provided as an appendix. The architecture itself is scalable to any number of bits, but we chose 20 bits to be the size of our plaintexts and keys as it provided a good balance of a large plaintext/ciphertext space to explore while still being small enough to train efficiently. We trained our network on an Nvida GTX 1070 for 30000 epochs with a batch size of 4096 per epoch. In each epoch, a set of 4096 random plaintexts were chosen along with 4096 random keys, and they were batched together through the network, with the goals of Eve and Bob to recover the plaintexts where Eve was not provided with the key while Bob was.

IV. Cryptanalysis

It is a general principle in cryptography that patterns often amount to vulnerabilities. The Vigenère cipher was finally broken using repeated strings in ciphertexts;

if a repeated string appears enough times, then by counting the gaps between repetitions, an attacker can infer possible keyword lengths, then break the cipher by trial-and-error using frequency analysis. The first Enigma machines were broken, in part, by exploiting a pattern of behavior of the German cryptographers. A hash function is considered unsafe (in fact, it is considered not to be a cryptographic hash function at all), if it is feasible for an attacker to find two inputs producing the same output. In all of these situations, we see that an attacker hopes to exploit asymmetries in the systems they're attacking.

Given that no analysis of CTNN cryptography (at least in the sense of Abadi-Andersen) was available to us, our analysis deals with some of the most basic yet fundamental questions. Rather than to study a particular type of attack, keeping in mind the principles described above, we look for general metrics that could inform an attacker as to what sort of attack might be viable by looking for asymmetries. We pause to note that the technique of frequency analysis, which is crucial in breaking monoalphabetic and Vigenère ciphers, does not seem to be available as a route to attacking CTNN's; there is substantial diffusion in the networks, and, moreover, the ciphertexts produced are not actually strings of bits but arrays of floating point numbers, which cannot unambiguously be decoded. To perform frequency analysis, the ciphertext must be able to be represented in the same alphabet as the plaintext, and while the plaintext in our case is bits, the ciphertext is a string of floating point numbers that can vary between -1 and 1.

We consider two questions relating to the structure of "cipherspace." By this, we mean the set of all possible 20-bit outputs produced by a single, trained network, using a fixed key. Of course, in practice one must consider the use of multiple keys, but leave such practical questions to the enterprising attacker. Concretely, we consider an input space consisting of a set B of all binary strings of length 20, represented as 2^{20} integer vectors in $\mathbb{R}^{20}$. A ciphertext is produced from a given $b \in B$ by first concatenating the fixed key (also from B), performing matrix multiplication, taking applying the sigmoidal function, taking a number of dot products again, applying the sigmoidal function again, taking a number of dot products again, then applying the $\tanh$ function. The result is a vector $T(b) \in \mathbb{R}^{20}$. We refer to the set $T(B) = \{T(b) : b \in B\}$ of 2^{20} vectors in $\mathbb{R}^{20}$ as "cipherspace".

Below, we provide two different analyses of the structure of cipherspace. Our use of Principal Component Analysis (PCA) helps us to understand the shape of cipherspace; for instance, is it spherical? Are there asymmetries that could be exploited by an attacker? Our statistical study of nearest neighbors asks about subsets of cipherspace of size 20, in anticipation of differential attacks; do similar plaintexts map to nearby points in cipherspace? Are certain plaintexts special in the sense of being especially close to their nearest neighbors?

A. PCA on Cipherspace

One of the first questions we had when beginning our analysis of this cryptosystem is "what does this space of encrypted texts *look* like?. Are the ciphertexts uniformly distributed? Is there structure inherent in the space that could lead to potential attack avenues in the future?" For example, if, say, one plaintext input is mapped to a unique point in this space completely isolated from other ciphertexts, that provides a lot of information to an enterprising attacker. As such, we looked for a way to potentially visualize this 20-dimensional space of ciphertexts.

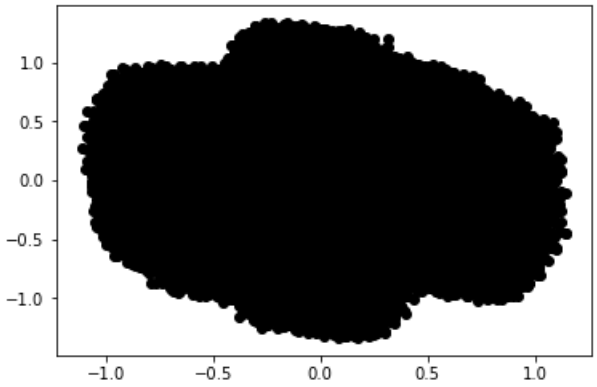


Fig. 3. 2D PCA Projection of Ciphertext Space

We settled on the Scikit-Learn [5] version of Principle Component Analysis to accomplish this task, with the implementation given in `pca_cipherspace.py`. Principle Component Analysis is a technique that decomposes data based on the eigenvalues of the covariance matrix of that data. In simpler terms, it finds the axes in which the data has the most variance — the first principle component is a vector that runs along the axis of most variation, the second is orthogonal to the first and runs along the axis of second most variation, etc. Thus, we reasoned that if the data were symmetrically distributed about this cipherspace, we should be able to find the 20 principle components of this data in $\mathbb{R}^{20}$ and be able to tell by noting that some principle components

	0
0	0.077
1	0.069
2	0.067
3	0.062
4	0.058
5	0.057
6	0.057
7	0.053
8	0.052
9	0.049
10	0.048
11	0.046
12	0.043
13	0.043
14	0.041
15	0.040
16	0.036
17	0.036
18	0.034
19	0.033

Fig. 4. Explained Variance Ratio for Each PCA Component

	0	1	2	3	4
0	0.133	0.478	0.157	-0.149	-0.209
1	0.065	-0.051	0.075	-0.261	-0.145
2	0.141	-0.057	-0.080	0.190	0.264
3	0.002	0.304	-0.460	-0.324	0.088
4	0.712	0.035	-0.356	0.240	-0.227
5	0.042	0.190	0.439	0.478	0.054
6	0.066	0.477	0.065	0.128	0.147
7	-0.157	-0.014	0.109	-0.164	-0.362
8	0.006	-0.454	0.043	0.058	0.096
9	0.225	-0.063	-0.145	-0.221	0.455
10	0.103	-0.102	0.275	0.068	0.234
11	0.008	-0.035	-0.278	0.116	-0.247
12	0.121	-0.023	0.235	-0.123	-0.291
13	0.136	-0.179	-0.001	0.125	-0.189
14	-0.066	-0.072	0.055	-0.159	-0.133
15	0.129	0.179	0.049	0.272	0.049
16	-0.009	-0.063	-0.056	0.002	0.227
17	0.513	-0.077	0.399	-0.432	-0.001
18	0.040	-0.242	-0.121	0.226	-0.325
19	0.201	-0.219	0.043	-0.041	0.146

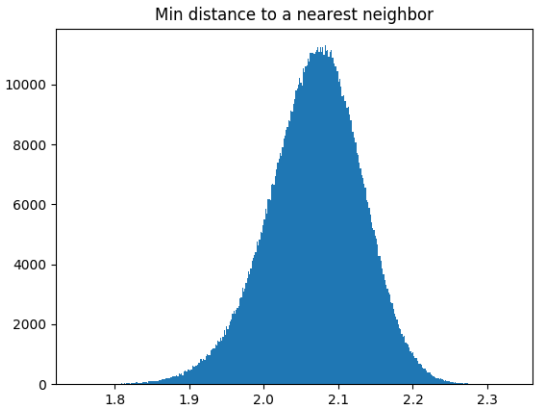
Fig. 5. First 5 Components of all 20 PCA Vectors

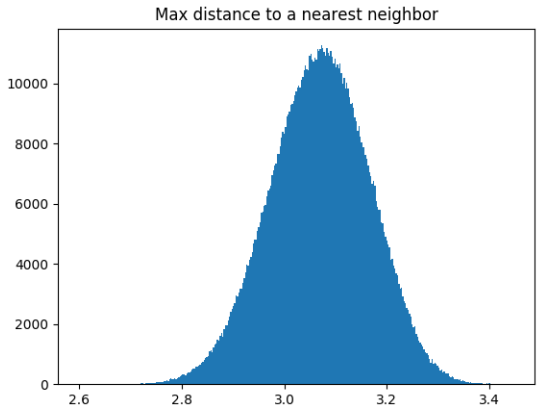
are "responsible" for different amounts of the overall variability than others.

One quick check to see if the data is symmetric in $\mathbb{R}^{20}$ is to see if, when projected down to $\mathbb{R}^2$, the data is still symmetric (just like how a sphere in 3D is a circle in 2D). Using just the first two principle components, we projected the data down to $\mathbb{R}^2$, and as can be seen in Figure 3, the result is clearly not perfectly symmetric. There is some symmetry there, to be sure, but it's not uniform.

With this in mind, we proceeded to analyze the results of the 20 dimensional PCA. Figure 4 shows the PCA Explained Variance Ratio for the 20-dimensional data. The explained variance ratio is, simply put, the percentage of the variance that is explained by each principle component. Principle components that lie along axes of higher variance "explain" a higher percentage of the variance in the data, and as the k^{th} principle component is defined to be the eigenvector corresponding to the k^{th} largest eigenvalue of the covariance matrix of the data, this list is in descending order. The main takeaway from this is that if the data were entirely symmetric or spherical, we should expect these values to all approximately be $1/20 = 0.05$ as no one axis of the data would contain more variation than any other. This reaffirms the conclusion that the space of ciphertexts certainly has structure that could potentially be exploited in a cryptosystem that is built on top of it. It is reasonable to ask what those structures may be, and for that, we need to look at the components of the principle components themselves.

Figure 5 lists the first 5 components of all 20 principle component vectors. Each row gives a single principle component vector $v \in \mathbb{R}^{20}$, with the entries of v lying along the columns, of which the first 5 components are shown. From this, it can be seen that the second entry of the first principle component, at position (0, 1), has a larger magnitude than surrounding components. As the first principle component also has the highest value of explained variance ratio, this implies that changes in the plaintext input cause a relatively larger amount of variation in the second bit of the output ciphertext. This kind of analysis can be performed with the rest of the principle components and their corresponding explained variance ratios, and it seems reasonable that this kind of analysis may open the door for differential cryptanalytic techniques to break a cryptosystem of this form.





B. *Statistics for Differential Attacks*

The analysis described in this section is conducted in the file `neighbors.py`. We generated the space B described above. For each $b \in B$ we generated the 20-element set N_b of nearest neighbors of b, i.e., the set of all elements of $n \in B$ differing from B in exactly one coordinate, i.e.,

$$N_b = \{n \in B : \ ||b - n||_{\ell_1} = 1\}.$$

Where

$$||v - w||_{\ell_1} = \sum_{i=1}^{20} \mid v_i - w_i \mid$$

We consider the subset $T(N_b)$ of cipherspace, and compute the set

$$D_b = \{||T(b) - T(n)||_{\ell_1} : \ n \in N_b\}.$$

From this we pull

$$m_b = \min D_b.$$

$$M_b = \max D_b.$$

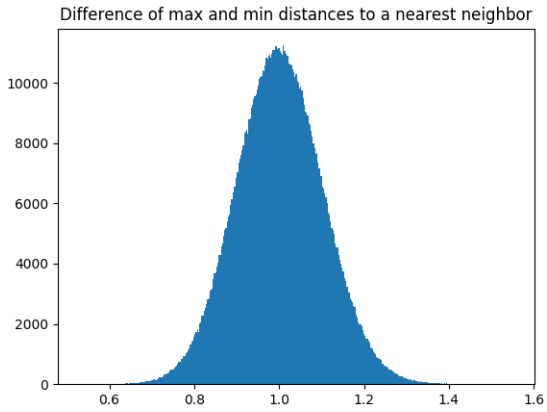
and

$$g_b = M_b - m_b.$$

We binned each quantity for varying b.

The first histogram shows the distribution of m_b's. We see an apparently normal distribution, though there may be a very small degree of left skewness.

$$\begin{aligned} \text{mean}(m_b) &= 2.06802 \\ \text{variance}(m_b) &= 0.00398 \\ \text{skewness}(m_b) &= -0.28974 \\ \text{kurtosis}(m_b) &= 0.27377 \end{aligned}$$



The max distances appear to be approximately normally distributed with a small degree of left skewness. We have

$$\begin{aligned} \text{mean}(M_b) &= 3.06943 \\ \text{variance}(M_b) &= 0.00940 \\ \text{skewness}(M_b) &= -0.09991 \\ \text{kurtosis}(M_b) &= -0.09021 \end{aligned}$$

The gaps g_b also appear to be normally distributed, with no apparent skew. We have

$$\begin{aligned} \text{mean}(g_b) &= 1.00140 \\ \text{variance}(g_b) &= 0.01051 \\ \text{skewness}(g_b) &= 0.06477 \\ \text{kurtosis}(g_b) &= 0.06045 \end{aligned}$$

We conclude that most plaintexts give ciphertexts which are at a distance very close to 2 from the ciphertexts of their nearest neighbors, but that there are certain plaintexts, those giving values in the tails of the first

histogram, which are special. We believe it is possible these special plaintext could be used to launch an attack against a CTNN cryptosystem, though we do not propose such an attack here. Regardless, it is clear from these results that the cryptosystem formed between Alice and Bob certainly doesn't display a level of diffusion that is to be expected of a modern cryptosystem like AES.

V. Further Directions and Conclusion

Our work has verified that it is possible to train CTNN's and, in fact, it is possible to do in a short period of time and without special hardware. Our analysis has found cause for concern about the strength of this cryptography, since we found asymmetries in the space of ciphertexts produced from a fixed input key as well as a low amount of of diffusion from the plaintext input to ciphertext output.

In terms of future work, first, we would also like to reiterate Abadi and Andersen's point that the prospect of training CTNN's to use asymmetric ciphers, and to use pre-built cryptography tools, is interesting and deserving of further exploration.

Next, there are many further questions to ask pertaining to applications of this cryptography. A primary question would be: is it efficient, compared to other forms of cryptography? Without looking at the inner workings of other cryptosystems such as AES, our sense is that CTNN's are likely inefficient. Also, what is the strength of the cryptography as the networks continue to train? In a similar vein, what potential is there such that when using CTNN's, the cyptography protocol *itself* evolves? We think there may be situations where this property is desirable, and the ability to build strong evolutionary cryptosystems would, if nothing else, certainly be interesting.

Finally, we also wonder about what effect the network design has on the cryptography. How would the cryptography change if we used a second fully-connected layer after the convolutional layers? Would this significantly impact training time? How would cipherspace change. Also, with or without design changes, can attacks against the cryptography be found?

VI. Appendix: Summary of included files

- **encoder.py** A class for encoding plaintext strings as binary strings. For simplicity, input strings are only allowed to consist of lower-case Roman alphabetic character plus six special characters "?; !.,". This class is not strictly necessary, but it is helpful to make comparison of input plaintext with nets' decipherings of ciphertexts more organic.
- **nets.py** This is where the neural networks are constructed. The file defines two classes. One class, `Net()`, provides a single neural net. Given an input size and parameters for the convolutional layers, `Net()` produces an object containing two sorts of TensorFlow variables: a collection `self.fc_weights()` of weights for the net's fully connected layer, and a collection `self.conv_weights()` of weights for the net's convolutional layers. Each instance has methods `self.fc_layer()` and `self.conf_layer()`, which take an input string of floating point numbers (in practice, numpy arrays) and perform the matrix operations specified by each layer type. Finally, each instance of `Net()` has a loss function that will be used to drive the training of the network. The file also provides a class `Trio()`. An instance of this class contains three instances of `Net()` named Alice, Bob and Eve. This class contains a method `self.train()` which trains the networks using tensorflow's implementation of AdamOptimizer [6] for the loss functions described in Abadi and Andersen [1]. It also contains methods for encrypting a plaintext (as Alice) or decrypting a plaintext as Bob or Eve. The encryption and decryption can be done in batches, so as to minimize the number of calls to run a TensorFlow session.
- **main.py** This file specifies the particular parameters we use in our study of CTNN's. In the `main()` function, a TensorFlow session is created. A `Trio()` object is created and trained. After the training, a saver is built and used to export the weights of the three networks. Statistical measure of the training efficacy are also produced.
- **loader.py** This file is used to load a trained model. The file looks much like main.py, except that after the session, trio and saver are created, the TensorFlow variables are initialized and then saved weights are restored. This file allows us to use single trained model during our analysis of the cryptography.
- **neighbors.py** This file provides methods for studying the space of ciphertexts of a trained model. It contains a simple function that produces all nearest neighbor plaintexts of a given plaintext. It defines helper functions for the primary function `get_all_distances()`, which produces possi-

ble 20-bit plaintexts and, for each such c_0, finds the largest and smallest distance of c_0 to one of its 20 nearest neighbors.

- **map_cipherspace.py** This file contains code that enumerates the list of 20-bit plaintexts and encrypts them all using a trained model. After this, it writes them to a file for use in the PCA analysis of those ciphertexts.
- **pca_cipherspace.py** This file contains the code to load a file of saved ciphertexts from `map_cipherspace.py` and performs Principle Component Analysis on them, yielding 2D, 3D, and 20D decompositions of the data.

REFERENCES

[1] Abadi, Martín and Andersen, David G., *Learning to Protect Communications with Adversarial Neural Cryptography*, http://arxiv.org/abs/1610.06918, 2016.

[2] Goodfellow, Ian and Bengio, Yoshua and Courville, Aaron, *Deep Learning*, MIT Press, http://www.deeplearningbook.org, 2016.

[3] Schoneveld, Liam, *Adversarial Neural Cryptography in Theano*, https://nlml.github.io/neural-networks/adversarial-neural-cryptography/, 2016

[4] Anand, Ankesh, *Neural Cryptography Tensorflow*, https://github.com/ankeshanand/neural-cryptography-tensorflow, 2016

[5] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. et al, *Scikit-learn: Machine Learning in Python*, http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html, 2011

[6] Kingma, Dierdrik and Ba, Jimmy, *Adam: A Method for Stochastic Optimization*, https://arxiv.org/abs/1412.6980, 2014