Architectures for Neuro-Cryptanalysis

COS485 Final Report

12 May 2020

Benjamin Kuykendall, Alberto Mizrahi, Abhishek Kumar Singh

Mentor: Jeffrey Cheng

Abstract: The known-plaintext attack of cryptanalysis can be phrased as a supervised machine learning problem. In particular, given many ciphertexts with known plaintexts, consider the task of predicting plaintext from unseen ciphertext. This is equivalent to learning the decryption function for a fixed key. In this setting, we pose the following question: does mimicking the structure of the known decryption function in a machine learning model aid in learning?

For concreteness and expediency, our target function is limited to a round-reduced DES encryption scheme. Further we limit the choice of models to three types of neural-network and neural-network inspired types. Our results support the hypothesis in a limited way. While completely mimicking the structure of the decryption function limits training, using an RNN to capture the repeated rounds of decryption is quite successful.

Introduction and previous works

Defining encryption and the neuro-cryptanalysis problem

An encryption algorithm is intended to hide messages (henceforth plaintexts) by encrypting them into some other representation (henceforth ciphertexts). For correctness, we must be able to decrypt, meaning to recover the plaintext from the ciphertext. All encryption algorithms have some secret value (called a key).

In particular, we are studying a *secret key block cipher*. Formally, this has the following guarantee: for any sequence of distinct plaintexts, and for a random choice of key, the ciphertexts should be indistinguishable from a truly random string of the appropriate length. Here, indistinguishable means that no efficient algorithm can detect, with probability better than ½, whether it is viewing the ciphertexts or the random string [BelRog05].

Of course, given enough time to crack it, no cryptosystem is secure in this sense. In particular we define the *known-plaintext cryptanalysis problem*. Fix a key k. Fix C* ciphertext corresponding to unknown plaintext P*. Let (P, C) be further known plaintext/ciphertext pairs. We say an adversary succeeds in cryptanalysis if it can recover, with good probability, P’ such that the normalized Hamming distance between P* and P’ is lower than ½.

Many cryptanalytic techniques recover the key k as an intermediate step. If this is successful, then P* can be recovered exactly by computing the value of Decrypt(k, C*). However, this is not strictly necessary: perhaps we can solve the more general problem directly. Further, this can be recast as a supervised machine learning problem: given samples C with labels P, cryptanalysis is exactly the problem of learning how to label fresh samples. In other words, we want to learn the function $x \mapsto \mathrm{Dec}(k, x)$.

The DES encryption algorithm

DES (or the Data Encryption Standard [NIST99]) is an encryption standard dating to the 1970s. It saw significant adoption at the time; however, it has since fallen out of use. Regardless, DES is emblematic of modern cipher design: features including Feistel rounds and substitution-permutation networks are still seen in contemporary ciphers including Blowfish and AES.

We describe the structure of DES at two levels: as a series of *Feistel rounds*, and as a *round function* that is used in each round. The algorithm takes as input a 56-bit key and a 64-bit message. The algorithm begins by applying a fixed *input permutation* of the bits. It proceeds by applying 16 Feistel rounds. It finishes by applying the inverse of the input permutation. The input and output permutations have no impact on the security of the scheme, but exist as artifacts of certain hardware implementations.

In each Feistel round, the following transformation is applied. The state is split in half. The *round function* is applied to the first half of the state. The output of the round function is XORed into the second half of the state. Finally, the two halves of the state are swapped. In the round function, the 32-bit input is expanded into 48 bits by duplicating certain bits. It is then XORed into a *round key*, derived from the overall key. This derivation is not discussed here. The result is then split into chunks of 6-bits and fed into 8 arbitrary functions each with 4 bits of output. These arbitrary functions are called *substitution boxes* and are implemented as fixed-function tables. Finally the 32 output bits of the substitution boxes are concatenated and then fed through another fixed *permutation*.

Due to the particular structure of a Feistel cipher, the decryption function is identical; however, the order of the round keys must be reversed in order to undo the rounds in the opposite order they were applied.

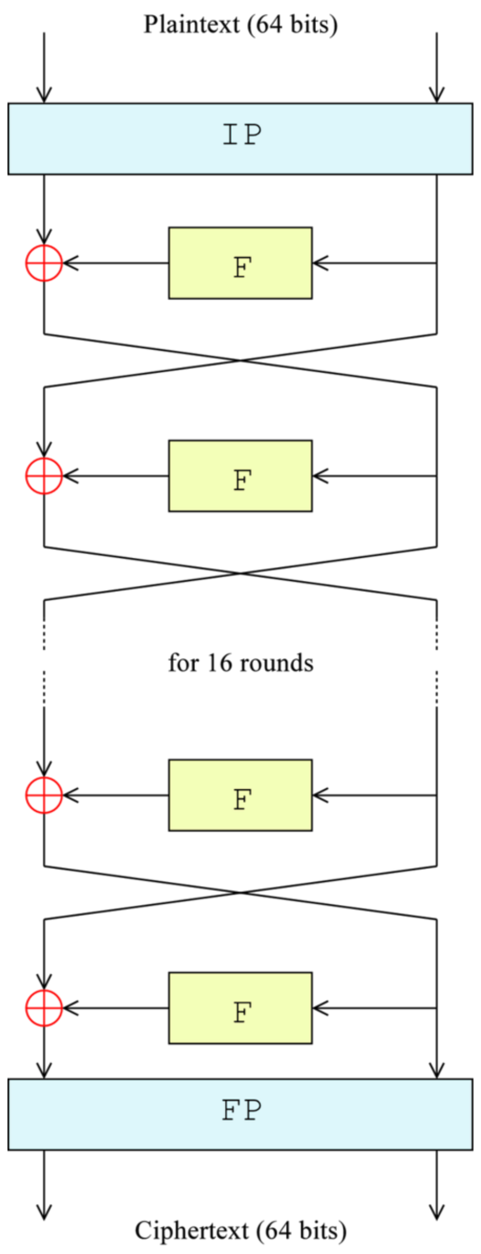

Figure 1: Diagram of Feistel Round structure of DES encryption. The state flows from top to bottom. F denotes the round function.

(source: https://commons.wikimedia.org/wiki/File:DES-main-network.png)

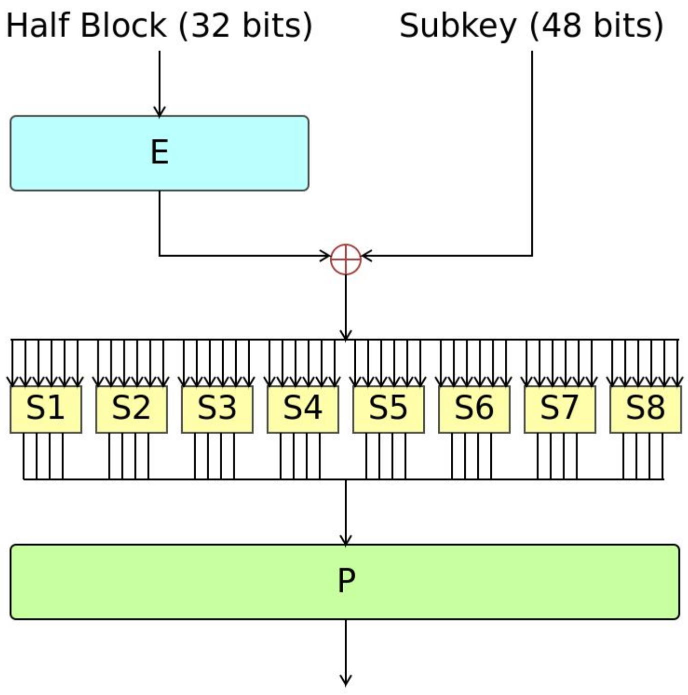

Figure 2: Diagram of the round function of the DES encryption.
E denotes the expansion transformation. S1 through S8 denote the independent substitution boxes. P denotes the round permutation.

(source: https://commons.wikimedia.org/wiki/File:Data_Encription_Standard_Flow_Diagram.svg)

Standard attacks on DES

All encryption algorithms are vulnerable to a brute force attack: simply try all the keys until you find one consistent with all the samples. This attack takes time exponential in the length of the key; in particular for DES it takes in expectation about 2^{56} evaluations. It only requires a handful of known-plaintext samples, however.

Other standard techniques require more samples but less time. We do not describe the details of the attacks, but simply list their results for comparison. For example, differential cryptanalysis takes on the order of 2^{47} samples [BihSha92], and linear cryptanalysis takes 2^{41} [BirCanQui04][KnuMat00]. State of the art attacks, specifically tailored to DES, can break the cipher with only 2^{39} samples [Junod01]. In practice, all of these attacks take weeks or months on supercomputers. Thus, while DES is insecure under high-resource attacks, it is still rather hard to break.

Round-reduction

This setting is too hard to break given our resources and the scope of this project. Thus we consider a weakened version of the cipher. As mentioned above, DES uses 16 Feistel rounds. By simply reducing the number of rounds to n, we obtain a weakened cipher DES-n. The cryptanalysis of DES-2 is well within our means, and thus we use it to study different approaches. Hopefully, what works in a few rounds works for many rounds. If given sufficient resources, our attacks on DES-2 could be scaled up, and the same behaviors should apply.

Hypothesis and goals

We study the effect of architecture choice on the success of neuro-cryptanalysis of 2-round DES. In particular, we try to take inspiration from the known structure of the decryption function. This is made concrete in our description of three architectures: a baseline model with no obvious relation to the structure of decryption, an RNN inspired by the repeated rounds present in the decryption function, and a custom model that models the decryption function exactly, save for the unknown round keys.

By incorporating our domain knowledge, we hope to find less powerful models (meaning models with fewer parameters) that are still capable of learning how to decrypt. This is a desirable property, as less powerful models can be faster to evaluate and train. Further, they seem to be more interpretable, capturing something about the decryption process.

Previous works

Several attempts at neuro-cryptanalysis using neural networks can be found in the literature. The technique can be very successful with primitive ciphers [Greydanus17]. But for modern block ciphers, successes have been more limited. In [Alani12], the authors use a cascaded feedforward net for neuro-cryptanalysis of DES. This model suggested by them serves as our baseline for comparison. In [HuZhao18], the authors have used the ideas presented in [Alani12] for performing neuro-cryptanalysis of the AES cipher. In [Jayachandiran18], the author presents an application of neural networks for performing neuro-cryptanalysis of the Simon cipher. Note that Alani's work targeted DES with full 16 rounds. On the other hand, Jayachandiran attacks a simplified Simon cipher for one and two rounds only. None of these works consider sufficiently diverse architectures to make the comparisons we are interested in.

These prior works are very shy in presenting the details of their implementation and hyper-parameters used in training which makes reproducing their work very difficult. In particular, despite considerable effort, we failed to reproduce the work of Alani We discuss this attempt in detail.

Reproducing results of Alani work

Reported results

In their paper [Alani12], Alani discusses the use of cascading neural networks to obtain very low training and validation bit error rates (BER) when learning to decrypt the full 16-round DES. In particular, they report that using cascading neural nets with layouts between 128-256-256-128 and 128-512-512-128, they can train their models to achieve average training errors below 5% BER and validation errors below 15% with high probabilities, while only using 4096 plaintext/ciphertext pairs for training.

Exact reproduction of techniques

Since the original implementation of Alani is done in Matlab, we attempted to reproduce their results in that environment (the code for this can be found in the Appendix). See the appendix for the full code. The datasets and keys were generated using a PRNG (pseudo-random number generator) as indicated in the paper. Furthermore, Matlab provides a convenient way of creating and training cascading neural nets using the *cascadeforwardnet()* and *train()* methods. Finally, we initialized the networks' weights using the *initlay()* as explained in the paper. However, after running multiple trails of training for various epochs and even larger training datasets, we were unable to reproduce the results in their paper. The model provided in the paper reported a BER of 50% on the training and validation datasets.

Further attempts in this regime

Given that the Matlab environment does not provide much flexibility to tune hyperparameters and try different architectural choices, we also attempted to implement Alani's paper in PyTorch with the hope that its greater flexibility would allow us to have more successful training sessions. Towards this end, we investigated multiple points and tried various combinations including

- Multiple activation functions: sigmoid, tanh, and ReLU

- Various optimizers: SGD, Adam, and AdaGrad
- Multiple initializations: standard, Xavier and Kaiming
- Additional hyperparameters: learning rates, L1 and L2 regularization

Even after all this, our only success was to achieve low training error rates - the validation BER steadfastly remained at 50%. This led us to conclude that the network was actually overfitting the data with no learning really occurring at all (this makes sense intuitively: our model has about 300K parameters while the training only has 4098 data points!). When we then attempted to use bigger datasets than the one in Alani, the training set blew up to 50% BER, indicating that it was not really learning anything specific to decrypting DES.

As a last attempt we emailed the authors of the paper with the hope that they could indicate some potential details that we had missed from our implementation. However, at the time of the submission of this paper, we have yet to receive a response. It seems likely that the paper was not solving the purported problem. Perhaps they were reporting training BER instead of validation BER. Or perhaps they were actually learning an easier function like 1-round DES. Without their source code or data set, we cannot determine what the authors actually did.

Design and implementation

Code

All of our project's code can be found in the following notebook:

https://colab.research.google.com/drive/1dUHjDu1d-KVcO2mUvMh6L0owQNnqotCP?usp=sharing

Not all the cells are run in the linked notebook. Instead, the resulting outputs are included in this document. Regardless the link includes all that is necessary to replicate our results.

Preliminary steps

Data synthesis

To avoid having to code our own implementation of DES, we utilize a standard and popular Python library [Wong19]. However, this library does not support the ability to encrypt with round reductions (i.e. use fewer than the 16 rounds that are standard for DES). To support this, we forked the library and made the necessary changes.

The next step is to then generate the plaintext/ciphertext dataset. Towards this end, a random key as well as 100k ($\approx 2^{17}$) random plaintexts, each one with 64 bits of length, are generated using a PRNG (a pseudorandom number generator) provided in the standard *secrets* module in Python 3. Each plaintext is then encrypted with the key utilizing our custom DES library. This generates a total of 100k ciphertexts which are then saved.

Notice that an alternative approach to generating the training dataset would be to simulate "real" plaintext. For example, we could have extracted (e.g. English) sentences from various texts and literatures and use that as the plaintexts. However, we decided that it was best to instead use randomly generated data because our goal was to train a model that could break the DES directly. We were concerned that utilizing human texts might lead to our model leveraging the contexts and similarities of these texts to learn successful predictions without truly understanding how to break the encryption scheme.

The datasets used for this project are available in the following zip directory

https://www.cs.princeton.edu/~brk/COS485/data.zip

Regression vs classification

Each ciphertext prediction is 64 bits. There are two main ways of predicting what each of these bits should be, corresponding to two different ways we can pose our goal. In the regression version, corresponding to every output bit, the model outputs a single float between 0 and 1. The prediction is then obtained by rounding it to 0 or 1. To measure the error, we can then use a loss function like MSE.

In the classification version, for every bit, two confidences are output which are then normalized using softmax. These two numbers then represent the guessed probabilities that the particular output bit is either a 0 or 1. The choice with the maximum probability is then chosen as the prediction for that bit. The error is then calculated using the cross entropy loss.

Both approaches make intuitive sense. However, in practice, we noticed that regression often converged badly (e.g. 0.34 BER) as compared to classification. We believe that this is due to the fact that the former has some bad local minima. In particular, using MSE, the model can substantially lower the error by outputting 0.5 for every bit thereby creating a local minimum that was hard to avoid even with good initialization. This observation thus led us to consider all our models in terms of classification.

Cascading and skip connections

Our first objective was to analyze how well a plain fully-connected feedforward network performed in this task. Taking care to create a model with roughly the same number of parameters as the more complex ones we were planning to implement, we attempted training on the architecture shown in Figure 3.

Unfortunately, no combination of hyperparameters and architectural choices were found to make it converge to any BER better than the trivial one of 0.5 for 2-round DES, even in the training dataset.

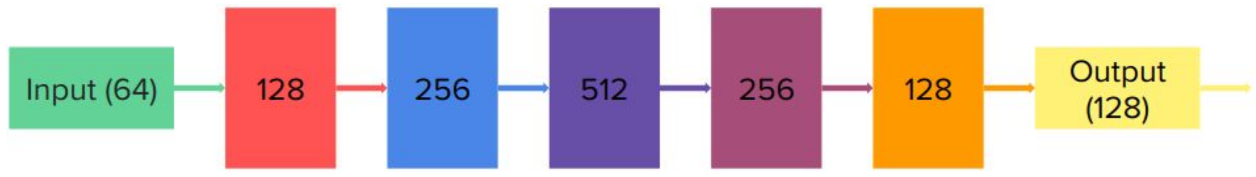

Figure 3: A fully connected feedforward neural net.
This is the first model that was trained on the dataset

The next option considered was the use of a cascading net, which is discussed in multiple places in the literature [Alani12] [HuZhao18]. In this architecture, each layer takes every previous layer as input. Thus, we can think of this as a fully connected net with every possible skip connection included.

The results obtained with the cascading architecture were positive and the net was indeed able to learn 2-round DES to a high degree. The intuition behind this might be due to the nature of the DES decryption. In particular, this algorithm itself has some “cascading” behavior in that each round takes the prior two outputs as input. Thus, we believe that this architecture is capturing this functionality which is then reflected in the low BER seen in training and testing.

Architectural choices

In many natural machine language problems, we know nothing or little about the nature or structure of the target function we are attempting to learn. But the situation in our project is just the opposite: we know the target function with complete certainty. Namely, the target function is the 2-round DES decryption algorithm for some choice of key. So one overriding question we investigated in each of the architectures we analyzed was: does incorporating the structure of the target function into our net help it learn in any way?

Architecture 1: the Baseline

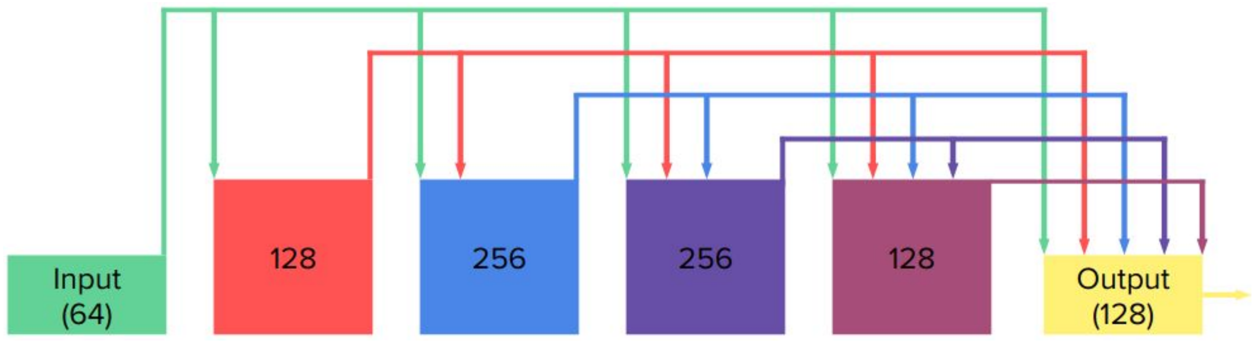

Figure 4: Architecture 1 - the (cascading) baseline neural network.

The first architecture for which we obtained significant positive results was for the cascading neural network shown in the figure above. We briefly motivated the need for cascading in the previous section. To choose the size and number of layers, we incrementally made the network larger until reasonable learning began to occur.

Architecture 2: the RNN

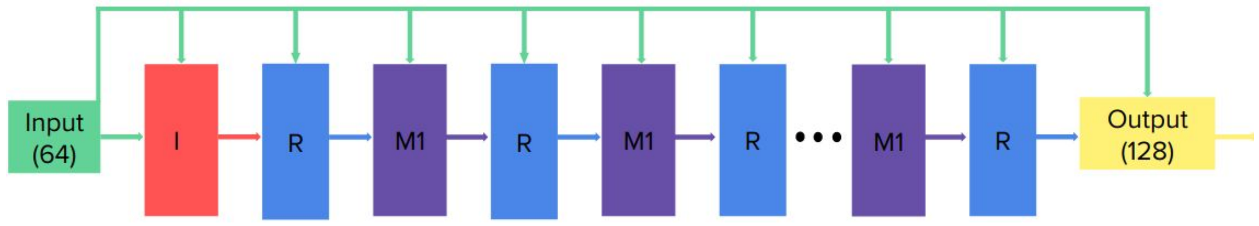

Figure 5: Architecture 2 - the RNN.

The RNN architecture is motivated by the structure of the DES encryption/decryption algorithm which has repetitions of two kinds of functional blocks. Hence we design an RNN with similar structure and hope that it can efficiently represent the DES decryption algorithm.

The idea behind each layer is as follows:

1. Layer I: Maps the input to a higher dimensional space.
2. Layer R: First kind of repetitive step in decryption.
3. Layer M1: Repeated intermediate processing between layer R's (Layer M1 can be a single layer or a group of multiple fully connected layers)

The output represented confidences for the classification problem described in the previous section.

Architecture 3: the Decryptor

For this model, we tried to use as much of the structure of the known decryption function as possible. In particular, we took each element of the decryption function and replaced it with a smooth continuation. The round keys were left as learnable parameters.

In particular, from our description of the decryption function above, we have three types of functions to consider.

1. Bit permutations (the initial and final permutations, as well as the round permutation and the “switching” between rounds are all bit permutations). These functions are already smooth.
2. XOR (both with the round key inside of each round, and between the first and second half of the state at the end of each round). We used the approximation ReLU((x-y)(2-x+y)). Because of the ReLU, the result is bounded by [0,1]. Then the quadratic ensures that it is large only when x and y are far apart.
3. S-boxes. We took a weighted sum over all possible outputs. The weights were proportional to the L1 distance between the actual input and the input corresponding to that output. This gives the correct behavior when the input is binary, and on non-binary inputs, it interpolates smoothly between outputs.

In particular for S-boxes, there are many different continuations to choose from. In fact there were lots of choices for when to clamp or normalize values that are [0,1] in the decryption function, but perhaps fall outside of this range in the continuation. Overall, we found learning was most successful with no intermediate clamping or normalization.

The output was a 64 bit vector x. To fit into our classification-based learning harness, we output confidences [x, 1-x].

Further model selection

Given our success with the RNN architecture, we explored it extensively for various sizes of the network and number of repetitions in the RNN. The choice of hyper-parameters in Table 1 was used in training all of the models discussed.

Our base RNN model 128(I)-256(R)-128(M1) with 2 repetitions achieved 29% validation BER. In order to improve the generalization results, we tried to make individual layers wider. We saw that the RNN model 256(I)-512(R)-256(M1) with 4 repetitions achieved a validation BER of 24%. When we made the layers even wider, e.g. the RNN model 5125(I)-1024(R)-512(M1), we saw the training error getting reduced; however the validation error was more than 40% which implies that the model was over-fitting.

Next, we tried to make Layer M1 deeper by making it comprise of 2 fully connected layers. The RNN model 256(I)-512(R)-256-256(M1) with 2 repetitions achieved a 24% validation BER and 21% training BER which was the best result for us.

Hence, the final RNN model selected was 256(I)-512(R)-256-256(M1) with 2 repetitions.

Optimizer:	Adam
Learning rate:	[0.0001 - 0.0006]
Error function:	cross entropy
Training set:	100k random samples
Weights init:	Xavier normal
Bias init:	all one
Activation function (hidden):	ReLU
Activation function (output):	linear
Loss function:	Cross-Entropy Loss

Table 1: RNN hyper-parameters and training settings.

Layer I	Layer R	Layer M1	Repetitions	Train BER	Val BER
128	256	128	2	0.28	0.29
128	256	128	4	0.25	0.27
256	512	256	2	0.23	0.26
256	512	256	4	0.22	0.24
512	1024	512	4	< 0.2	> 0.4*
384	640	384	2	0.17	0.27
256	512	2x256†	2	0.21	0.24

* training did not converge for this model
† in this model we replaced M1 with 2 independent layers

Table 2: RNN validation results by network parameter.

Results and discussion

Final results

In addition to reporting the raw performance of models, we provide two other important results. First we measure the power of each model, to give further context to their performance. Second we show their training curves to give an idea of how easy or hard they are to train in practice.

Model performance

We performed another round of training for the selected model of each kind and measure their performance on the validation set. The results can be summarized as follows. The cross-entropy loss function is used throughout.

	Training loss	Training BER	Validation loss	Validation BER
Baseline	0.4915	0.2140	0.5146	0.2526
RNN	0.4393	0.2177	0.4531	0.2473
Decryptor	0.7374	0.4735	0.7384	0.4734

Table 3: Training and validation results by architecture.

Finally we evaluated the best model, namely the RNN with parameters described in the model selection section, on a held-out test set. This resulted in a BER of 0.2507.

Comparing model power

We see that both the baseline and RNN networks can achieve 25-30% validation BER. The cascaded feedforward baseline has a lot of expressive power with 316224 trainable parameters.

The decryptor net had 96 parameters, one per bit of the round key. This is close to the theoretical minimum size of a model that can learn decryption. However, it was quite hard to train, turning out quite sensitive to the way we implemented the smooth approximations of relevant functions. Even in the best setting we found, error rates left much to be desired.

The RNN models on the other hand, seem to be a happy medium. They can achieve similar performance to the cascading model with many fewer parameters (139904 for the base RNN model). As a result we feel that stronger RNN models might be the feasible way of building models for decryption with more than two rounds.

The failure of the decryptor net is an instance of the *proper vs. improper learning* problem in theoretical machine learning [KearnsVazirani94]. Sometimes, regardless of plain model power, learning a model with the same form as the target function is harder than learning the same function with a more general model.

Training characteristics

The results are shown in Figures 6, 7, 8. The plots were taken from a separate run, so the numbers may vary slightly. Regardless, they demonstrate the general characteristics of the models.

Baseline model: The cascaded feedforward baseline model is very difficult to train. The training is very slow and requires fine-tuning various hyper-parameters. The slow training is also evident from the training curves.

RNN model: The RNN models achieve similar test and validation BER as the cascaded feedforward net and are much easier to train. A wide range of hyper-parameters can be used for successfully training the network.

Decryptor model: this model performed too poorly to make further qualitative comparison to the others meaningful. Our BER was strictly less than ½, which proves that some learning occurred. However, we ran out of time to further adjust the architecture and training. We hope that given further time a better BER could be achieved.

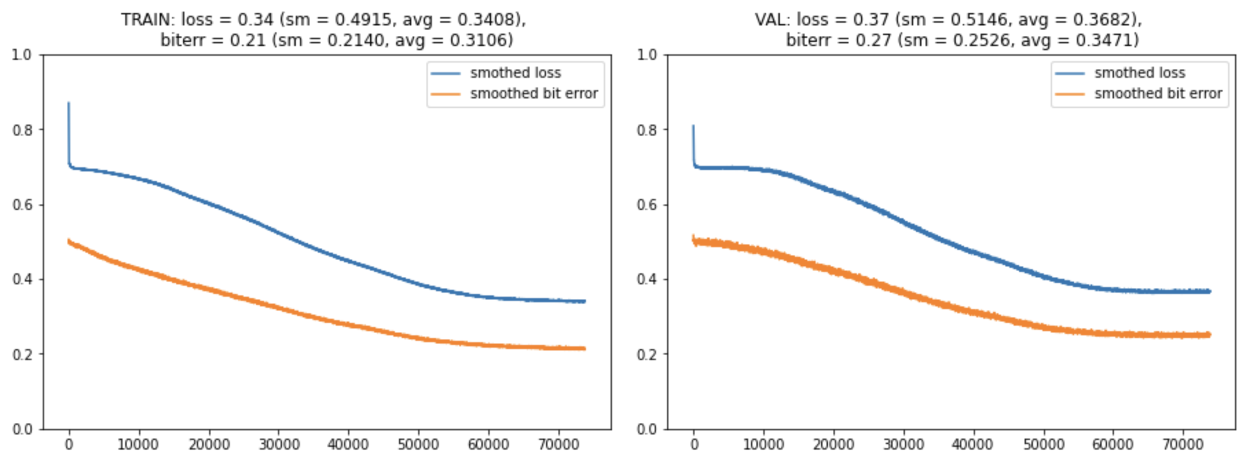

Figure 6: Training curve for baseline model.

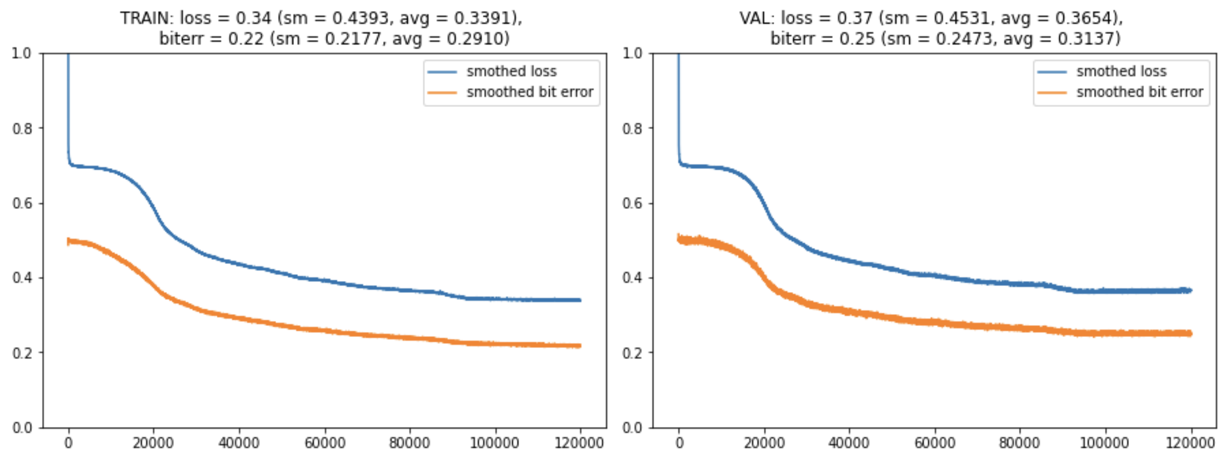

Figure 7: Training curve for RNN model 256(I)-512(R)-256-256(M1).

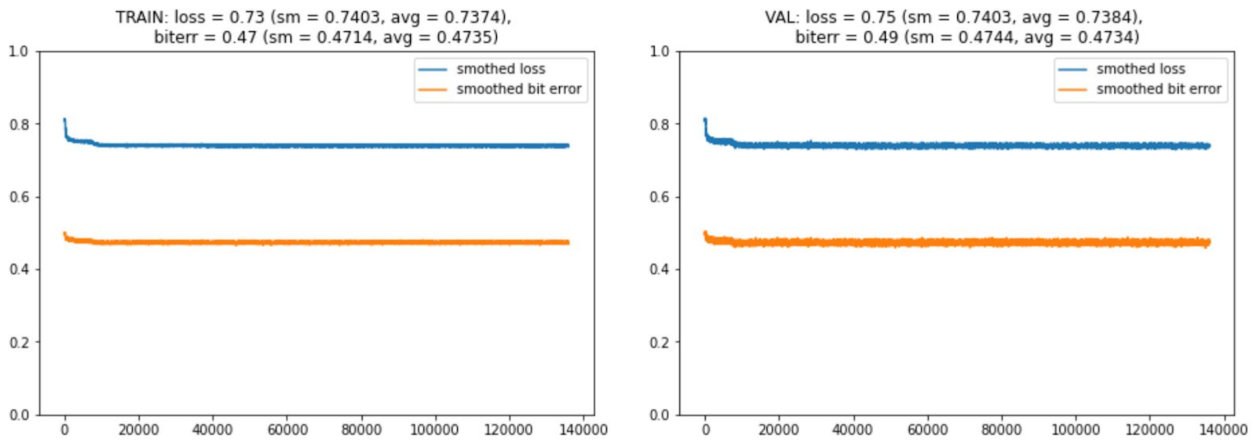

Figure 8: Training curve for decryptor model.

Strengths and weaknesses

Compared to prior work, the greatest strength of our project is the diversity of models: we try three distinct architectures on the same dataset and use their performance results to draw conclusions about cryptanalysis more generally. In addition to diverse architectures, we performed a rather thorough hyperparameter selection for the RNN model. We further provide public source code and data, which makes our work reproducible.

Our work is weak in supporting the extensibility of our techniques in ciphers with more rounds. If we had results for, say, 5 rounds, then we could plot our error as a function of the number of rounds and model size. With data like this, extrapolation to the full 16 rounds might be reasonable. But since we could only get training to occur on 2 rounds (1 round is trivial since half of the plaintext appears directly in the ciphertext) such extrapolations are impossible.

We also believe some of our preliminary steps were important. In particular, we were quite misled by the plurality of papers posing decryption as a regression problem. By making the switch to classification, we immediately began seeing better results. By documenting learning like this, we can hope to guide future attempts at neuro-cryptanalysis.

References

[Alani12] Alani M.M. (2012). “Neuro-Cryptanalysis of DES and Triple-DES”. *Neural Information Processing*. ICONIP 2012.

[BelRog05] Bellare, Mihir; Rogaway, Phillip (2005). *Introduction to Modern Cryptography*. “Chapter 5: Symmetric Encryption" p. 93.

[BihSha92] Biham, Eli; Shamir, Adi. (1992). “Differential Cryptanalysis of the Full 16-Round DES”. *Advances in Cryptology* (CRYPTO 1992).

[BirCanQui04] Biryukov, Alex; Cannière, Christophe De; Quisquater, Michaël (2004). “On Multiple Linear Approximations”. *Advances in Cryptology* (CRYPTO 2004).

[Greydanus17] Greydanus, Sam. “Learning the Enigma with Recurrent Neural Networks”.

[HuZhao18] Hu, Xinyi; Zhao, Yaqun. (2018). “Research on Plaintext Restoration of AES Based on Neural Network”. *Security and Communication Networks* 2018. 1-9.

[Jayachandiran18] Jayachandiran, Kowsic. “A Machine Learning Approach for Cryptanalysis”. (2018).

[Junod01] Junod, Pascal. (2001). “On the Complexity of Matsui's Attack”. *Selected Areas in Cryptography* (SAC 2001).

[KnuMat00] Knudsen, Lars; Mathiassen, John(2000). “A Chosen-Plaintext Linear Attack on DES”. *Fast Software Encryption* (FSE 2000).

[NIST99] National Institute of Standards (1999). “Data Encryption Standard”. *Federal Information Processing Standards* (FIPS 46-3).

[Wong19] Wong, Eric. (2019). “DES”. *Python Package Index*.

Appendix

Matlab code implementing Alani’s model [Alani12].

```
load('np_cipher.mat')
load('np_plain.mat')

input_data = cipher(1:90000,:)';
target_data = plain(1:90000,:)';

net = cascadeforwardnet([128,256,256,128],'trainscg');
net = train(net,input_data,target_data);
view(net)

test_data = cipher(90001:90500,:)';
test_target_data = plain(90001:90500,:)';

y = net(test_data);
y = round(y);

perf = norm(abs(y - test_target_data),'fro')^2/(500*64);
```