Teaching Machines to Break Symmetric-Key Encryption

Mike Robeson BSc

Data Engineering MSc, University of Dundee
Supervised by Sasa Radomirovic & Jianguo Zhang

Executive Summary

Encryption protects sensitive information from malicious actors who might use it for personal gain. It is an important aspect of our daily lives, yet it tends to be unseen and unchallenged. If an actor were able to break encryption systems without warning, everyone's sensitive data would be at risk. The focus of modern encryption is to make the system as difficult to break as possible, whilst retaining usability. But could this compromise be exploited? There are existing attacks that have been tried for these systems, but these have all been explicitly programmed. So what if we could perform these attacks without explicitly programming how they should be done? Would this reveal a weakness that hadn't been seen before?

Neural networks have recently solved complex and varied problems that were previously thought impossible for computers to do. They do this by emulating the structure of the human brain - using neurons that fire when a specific input threshold has ben reached. The human brain is considered one of the most complex systems in the universe, so neural networks could hold extraordinary power.

This project aimed to test whether these powerful new models can break symmetric-key crypto-systems, succeeding where others have failed. It used an iterative analysis process, so lessons learned from analysing simple crypto-systems could be used for more complex crypto-systems. Data was generated for several crypto-systems from scratch to provide low level data that could provide meaningful results. Then Google's Tensorflow deep learning library was used to perform several classification and regression tests on this data. The project focussed on discovery, rather than fully complete solutions, so that it could be flexible in approaching this hitherto incomplete problem.

The project was somewhat successful, as it showed that simple crypto-systems can be broken by neural networks reliably for a variety of tasks. It also showed that the designed tasks were possible to solve and could retrieve relatively sensitive information. In doing so, there are already implications for the use of neural networks in the area of cryptography. However, more complex crypto-systems proved difficult for the networks to break. There were also issues with how the analysis was executed. But, lessons learned from this project can be used as reference for future work to inform future design approaches, crypto-system implementations to test, other network types to test and other useful tasks that can be attempted.

Declaration

I declare that the special study described in this dissertation has been carried out and the dissertation composed by me, and that the dissertation has not been accepted in fulfilment of the requirements of any other degree or professional qualification.

Certificate

I certify that Mike Robeson has satisfied the conditions of the Ordinance and Regulations and is qualified to submit this dissertation in application for the degree of Master of Science.

Acknowledgements

I'd like to thank:

- All the lecturers who have taught me over the past year - the constant supply of knowledge has been challenging but enthralling
- My parents, who have been a constant supply of support and encouragement over the past year
- My Dundee friends: Scott, Philip and Craig, who have made the past year all the more enjoyable, and helped me endure the final few months of pain
- My London friends: Rikki for the constant supply of DJ mixes, new albums and forward notice of nights out; Alex for the phone chats about life and Adam for the absolutely ridiculous messages we send to each other
- My grandfather, for being the reason I'm even able to do this degree
- Everyone on the computing masters courses - good luck to you all wherever you go on to

Special mentions go to Andy Cobley for getting me to come up to Dundee in the first place, project supervisors Sasa Radomirovic amd Jianguo Zhang for meetings & support throughout the project and Keith Edwards for co-ordinating the whole year.

Finally, I'd like to thank all of the following musicians for the music which has gotten me through the past few months:

Richard D. James (Aphex Twin, Polygon Window, The Universal Indicator), Autechre, Artefakt, Boards of Canada, To Rococo Rot, Lee Gamble, Oneohetrix Point Never, Cristoph De Babylon, Rythym & Sound, Deadbeat, Call Super, Objekt, Burial, Brian Eno, Boy Robot, Dictaphone, Donato Wharton, Jon Hopkins, The Chemical Brothers, The Remote Viewer, Clark, Donato Dozzy (Voices from the Lake), Neel (Voices from the Lake), DJ BONE, DJ Stingray, Slowdive, Bochum Welt, James Holden, Biosphere, Bill Converse, Daed, Phalaeh, Anthony Childs (Surgeon), Bandshell, Basic Channel, Wen, Answer Code Request, Anthony Naples, Alessandro Cortini, Actress, Abfahrt Hinwii, Daft Punk, Fennesz, deekie, Deru, Dopplereffekt, El Fog, eleven tigers, Four Tet, Gerry Read, Lukaz Wigflex (Spam Chop), Harmonia, Helios, Jefre Cantu-Ledesma, JoeFarr, Jon Brooks, Moritz Von Oswald, Juan Atkins, Kassem Mosee, Koen Holtkamp (Mountains), Kyle Bobby Dunn, Laraaji, Lukid, Nathan Fake, Luke Abbot, Maurizo, Morphosis, Moderat, Oren Ambarchi, The Other People Place, Drexciya, Pessimist, Plaid, Pole, Porn Sword Tobacco, Radiohead, Rumpistol, S Olbricht, Shinichi Atobe, SND, Soulwax, Stars of the Lid, System, Terekke, The Units, Untold, Vessel, A Winged Victory For The Sullen, Nine Inch Nails, worriedaboutsatan.... and many many many more.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

Encryption is an important issue in the modern technological era. It protects personal data on our devices, secures millions of bank transfers per day and protects our messages from interception for malicious use. As computation power increases over time, malicious actors can wield that power to break the lifeline of our privacy and security. The classic example of the Enigma code, broken by Alan Turing et al., demonstrates the devastating effect this can have. The current industry standard symmetric-key encryption is the Advanced Encryption Standard (AES).

A notable development in recent years of computational analysis / data processing is deep learning. Utilising Deep Neural Networks, Google's DeepMind built Alpha Go. Typical Go games consist of 150 - 450 moves, resulting in 10^{360} possible moves (permutations) to be considered. Alpha Go competed against human Go experts and won 5 out of 5 rounds - a challenge that many thought would be impossible due to the sheer complexity of the problem.

Whilst Alpha Go did not consider every possible move within the 10^{360} permutations, instead looking x moves ahead and judging the best quality move to take, this approach is interesting considering there are fewer total permutations for 256-bit AES encryption (10^{77}). A crypto system is considered semantically secure if there is no feasible method that can guess the values of unencrypted data with better than 50 percent probability. Considering the circa 10^{300} order of magnitude difference of permutations between Go and AES 256-bit, we can see the potential risk of an efficient adversary being developed.

Testing symmetric-key crypto-systems to ensure semantic secure is paramount to keeping sensitive information safe. The Data Encryption Standard (DES) was broken whilst it was still in widespread use several decades ago. But it took several years for a new crypto-system (AES) to be developed and accepted as the industry standard. 15 years ago, this may have been acceptable. But in a rapidly evolving world where more and more sensitive information is transferred every day, the effects of this reoccuring could be devastating.

1.2 Project Objectives

So it is vitally important to investigate whether widely adopted crypto-systems might still be semantically secure and, if not, the degree to which they are insecure.

This project will focus on five key objectives to test the security of these systems:

1. Show that simple crypto-systems that are known to be insecure could be broken by neural networks
2. Break simple crypto-systems that are known to be insecure
3. Show that semantically secure crypto-systems could be broken by neural networks.

4. Break semantically secure crypto-systems

5. Generate some approximations of keys or full crypto-systems

The project will use an iterative analysis process to investigate a range of crypto-systems, with each analysis iteration increasing in complexity. For each crypto-system, several classification and regression tasks will be attempted. If neural networks can perform any of these tasks to a sufficient degree of accuracy (i.e. better than a 50% guess), a further task for each objective will be to investigate their performance in contrast to existing attack methods. This will be done by searching through parameter spaces and interpreting the results.

2 Background

2.1 Encryption

Encryption is a method of encoding used to obfuscate information from adversaries (someone who shouldn't see the information) (*Applied Cryptography* 2016). There are many commercial symmetric key encryption implementations available (OpenSSL, 2016a), performing complex transformations on input data (plain texts), using a key to produce obfuscated data (cipher texts) (*Applied Cryptography* 2016). Encryption's roots stem from simpler systems called Ciphers (*Applied Cryptography* 2016).

2.2 Ciphers

Ciphers have been used for thousands of years (Cryptography, 2012). Early examples include the Scytale cipher, used by the Spartans during battle, and the Caesar cipher, used by Julius Caesar for private messages (Cryptography, 2012), or ROT13 (an extension of the Caesar cipher), which is still in use today. Their purpose is the same as modern encryption systems, i.e. to obfuscate data. But they are much simpler in design and insecure (*Applied Cryptography* 2016).

2.2.1 Caesar

All three of the above example ciphers perform a shift transform (Cryptography, 2012). Plain text content is typically natural language characters. To encrypt, plain text values are shifted by a pre-specified amount across the language's alphabet e.g. a shift value of 4. This shift value is the key of the cipher (Cryptography, 2012). Examples can be seen in table 1. So the Caesar cipher is easy to use and implement, but is insecure. Plain texts can be exposed through both direct or indirect attacks.

Plain-Text	Shift Value	Cipher-Text
A	1	B
B	5	G
ABC	1	BCD
XYZ	-3	UVW

Table 1: Example Caesar Cipher usage

2.2.2 Simple Substitution

Another simple cipher is the substitution cipher. Instead of a shift value, the key is a lookup table of characters that are substituted. The key, again, must be pre-agreed by both sender and recipient. The simple substitution cipher is easy to use and implement but suffers from the same weak security issues for direct and indirect attacks. Examples of the mapping (key) and usage can be seen in tables 2 and 3 respectively.

Plaintext Value	Cipher Text Value
A	B
B	Q
C	A
X	P
Y	T
Z	D

Table 2: Example Simple Substitution Cipher Mapping

Plain-Text Message	Cipher-Text Message
ABC	BQA
XYZ	PTD
ABX	BQP
XYC	PTA

Table 3: Example Simple Substitution Cipher usage using table 2's mapping

2.2.3 One Time Pad

The one time pad (OTP) was originally conceived in the late 1800s (Rijmenants, 2017). Instead of natural language, a plain text is encoded into a set of integers. The key must be of equal length to the plain text (*Applied Cryptography* 2016). To encrypt, a plain text is XORed with the key. This produces the cipher text. To decrypt, the XOR operation is reversed, using the cipher text and the key (*Applied Cryptography* 2016). Example encryption usage can be seen in table 5.

As a new key must be used for each message, OTPs are generally difficult to implement and have not seen widespread usage (Rijmenants, 2017). However, government agencies such as the NSA and KGB have historically used OTPs because it has perfect security (Rijmenants, 2017).

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4: XOR truth table

2.3 Perfect Security

A system is perfectly secure if, and only if, there is not an attack method which, given a cipher text, can return any value of the plain text with greater than 50%

Plain Text Bits	Key Bits	Cipher Text Bits
00	10	10
01	11	10
10	01	11
01	00	01

Table 5: Example OTP usage for very small key size

probability. If an adversary is able to perform an attack that retrieves even a single digit of a plain text with anything better than 50% probability, then the system in question is no longer perfectly secure (*Applied Cryptography* 2016).

Perfect security could be considered the ultimate goal of any crypto system. But implementation issues seriously restrict usability (*Applied Cryptography* 2016). For the example of OTP: key size must equal message length; secure exchange of keys is difficult; single use of keys must be adhered to and true randomness is difficult to obtain (Rijmenants, 2017). One example of how a perfectly secure cipher can be implemented incorrectly is the Multi-Time Pad (*Crypto 101* 2017).

2.3.1 Multi Time Pad

A Multi Time Pad (hereafter, MTP) is created in exactly the same way as an OTP, however, the key is static and does not change per message. Given 2 cipher texts, one can XOR these results to see the combined plain texts (*Crypto 101* 2017). This can be seen in figure 1 Mixedmath, 2011 .

Figure 1: 2 x Plain text images and randomised MTP key

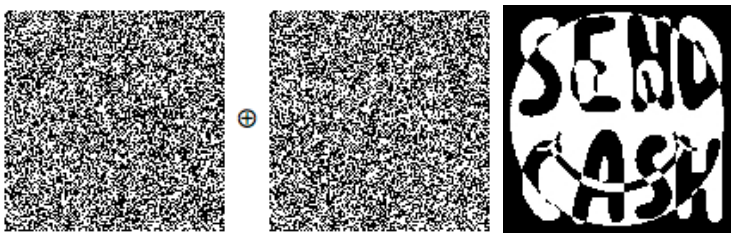


Figure 2: 2 x cipher text images being XORed together to expose the plain texts from figure 1

Aside from a XOR operation of cipher texts, as the number of messages encrypted using the same key increases, and given known plain and cipher texts, the probability of correctly guessing key bits increases (*Applied Cryptography* 2016). If we know that a 1 value occurs in a specific position in several plain texts, but the respective cipher texts all have a value of 0 for the same bit position, we can then deduce that the key bit must be a 1 value.

2.4 Symmetric Key Crypto Systems

2.4.1 Semantic Security

The definition of semantic security is similar to that of perfect security, with one notable difference. Instead of specifying there be no single direct attack method available, semantic security asks that there be no feasible attack method available (*Applied Cryptography* 2016). Rather than specify a crypto system must never be able to be broken, semantic security states that attacks should take far too long and use far too much computing power to be successful and reveal no partial information (*Applied Cryptography* 2016). Here, attacks refer to the ability to reveal any partial information about an unknown plain text from a known cipher text (*Applied Cryptography* 2016). The definition of semantic security is a compromise to increase usability of encryption systems, but maintain high levels of security (*Applied Cryptography* 2016).

The simple ciphers we have seen (excluding OTP) are all considered semantically weak . A small amount of computing power can break systems in a short time. In contrast, the AES-256 system requires more time for a brute force attack than the universe has currently been existence (Arora, 2012). Even this would require a massive number of the world's most powerful supercomputers to complete. So, whilst not perfectly secure, AES-256 is fine for modern use.

2.4.2 Block Ciphers

A type of modern crypto-system is the block cipher symmetric key algorithm (*Crypto 101* 2017). Blocks of plain text data are fed into the crypto-system, which outputs a block of the same size as the inputs. The size of inputs and outputs is known as the block size (*Applied Cryptography* 2016). If a system has a block size of 8 bytes, and we have a 16 byte message to obfuscate, the input data is split into 2 blocks of 8 bytes each. The output data, 2 blocks of 8 bytes each, corresponds directly to the relevant input block (Roeder, 2010). Block cipher inputs must have a length that is a multiple of the block size, else padding (i.e. extra miscellaneous characters) must be included (*Applied Cryptography* 2016).

There are several modes of operation for Block Ciphers(FIPS, 1980). Electronic Code Book mode (ECB) is the simplest. Blocks are processed by the crypto-system's base implementation with no additional transformations (Roeder, 2010). To encrypt, plain texts are passed through the system as input, alongside the key as per figure 3 (WhiteTimeWolf, 2013). To decrypt, the same process occurs in reverse, with the cipher text passed as input. Duplicate input blocks in ECB mode always produce the

same output blocks(FIPS, 1980). Figure 4 & table 6 demonstrate this with an image and text input respectively (Ewing, 1996).

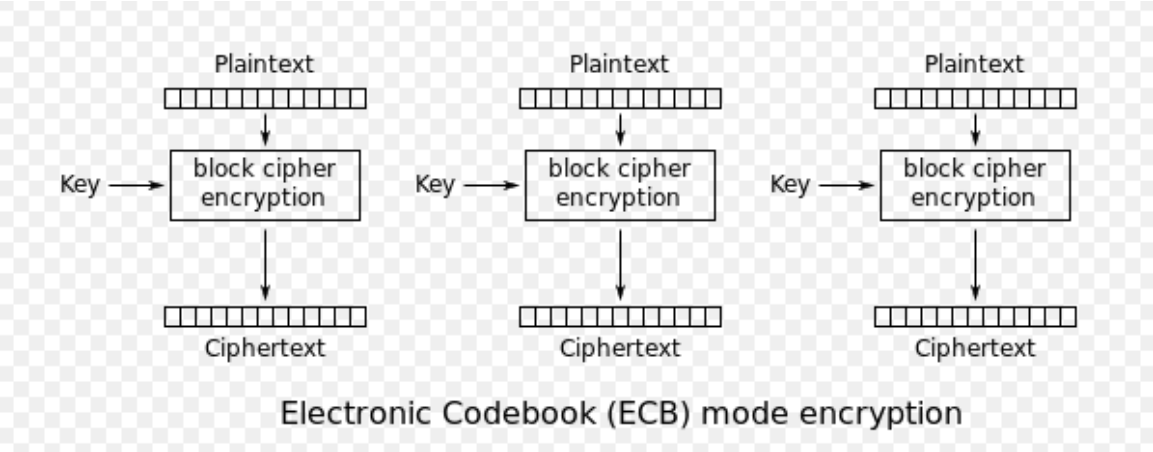


Figure 3: ECB mode encryption for block ciphers

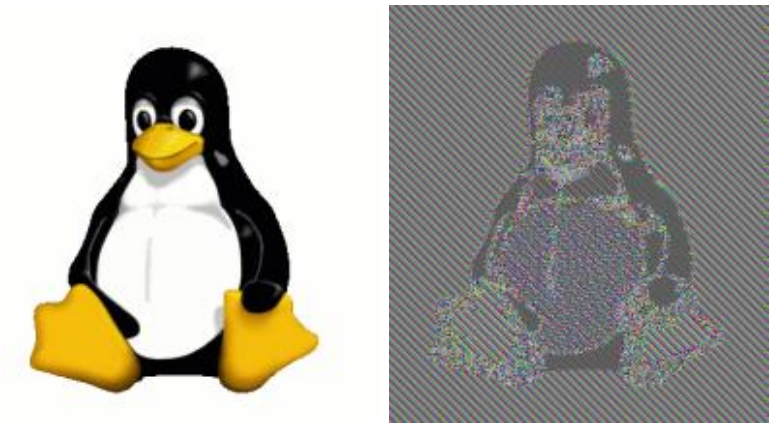


Figure 4: An example plain text image, and the same image encrypted in ECB mode

Block N^o.	Text Block	Hex Input Block	Hex Output Block
1	Now␣is␣t	4e6f772069732074	3fa40e8a984d4815
2	Now␣is␣t	4e6f772069732074	3fa40e8a984d4815
3	he␣time␣	68652074696d6520	6a271787ab8883f9
4	for␣all␣.	666f7220616c6c20	893d51ec4b563b53

Table 6: Example DES ECB mode outputs (FIPS, 1980)

ECB mode is not considered semantically secure (*Applied Cryptography* 2016). Other modes of block cipher operation, such as CBC (Cipher Block Chaining) or

PCBC (Propagating Cipher Block Chaining), add XOR operations between subsequent message blocks (the initial input block requires an randomised initialisation vector) (*Applied Cryptography* 2016). Using these methods, two duplicate input blocks, within the same message, will not produce duplicate output blocks (Roeder, 2010). We can see an example outputs for these modes in figure 5 (Lunkwill, 2006). Whilst the cipher text image looks random, some insecure crypto systems will simply produce a random looking output. Further to CBC and PCBC, there are modes which will transform a basic block cipher into a stream cipher, such as CFB (Cipher Feedback), OFB (Output Feedback) and Counter (CTR) (*Applied Cryptography* 2016).

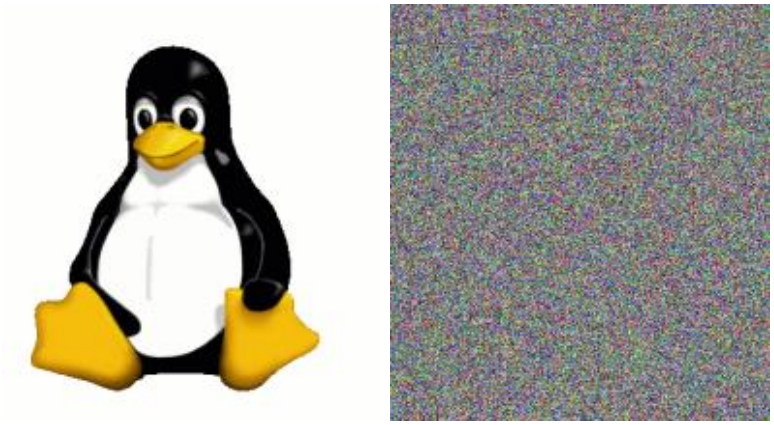


Figure 5: An example plain text image, and the securely encrypted image

2.4.3 Attacks on Symmetric Key Crypto Systems

There are three main attacks used to break symmetric key crypto systems, each using known plain and cipher text pairs to reveal the key.

The first, and simplest, is a brute force attack. A brute force program will randomly generate keys, decrypt the known cipher text with the random key and compare the decrypted data to the known plain text. If the decrypted data does not exactly match the plain text data, then another key is tried. Due to the random key selection, this is not a sophisticated attack and can be inefficient to perform. However, it can perform attacks on both stream and block ciphers.

Linear cryptanalysis is an attack specifically developed for block ciphers (Langford and Hellman, 1994). It constructs affine linear approximations of the cipher (Langford and Hellman, 1994). These affine transformations roughly represent the relationship between key bits and the plain / cipher text bits. An algorithm (Matsui's Algorithm 1 or 2) is then applied to probabilistically determine the likely values of key bits from the final round (Langford and Hellman, 1994). This is a more complicated attack than brute forcing, but can produce results with fewer known plain/cipher text pairs.

Finally, differential cryptanalysis is an attack used for both block and stream ciphers (and cryptographic hash functions) (Langford and Hellman, 1994). With block ciphers, the resulting cipher text differences from using various plain text inputs

can be analysed in an attempt to reveal the transformations involved in a cryptosystem (Langford and Hellman, 1994). This includes identifying patterns of non-random behaviour, such as repeated lookup tables or bit flips.

2.4.4 Block Cipher Operations

Generally speaking, most block ciphers use similar transformations to perform their encryption/decryption process. Some of these operations are detailed below, with example tables in Appendix 1 (Grabbe, 2006) (FIPS, 1977a).

Permutation

- Similar to the shifting of values using the Caesar cipher.
- Switches position of a block's bits according to a lookup table.

Block Division/Splitting

- Splits original input block into two (or more) subsequent blocks for further operations.

XOR

- As per OTP and MTP
- If both bit inputs are equal, return 0, else return 1

Shifting

- Essentially the same as the Caesar cipher
- Shift bit values left (or right) by a pre-defined number of positions.

Substitution (hereafter, S-Box)

- The cornerstone of the modern day crypto system
- Uses input bit values to refer to a lookup table
- Output bit values are the values stored in the table

2.5 Data Encryption Standard

2.5.1 Brief History and Overview

The Data Encryption Standard (hereafter DES) was developed in the early 1970s by IBM (Roeder, 2010). It was approved as the United States' federal standard in 1976 (for classified materials) and made publicly available in 1977 (for unclassified materials) (FIPS, 1977b). The Federal Information Processing Standards (the USA's official standards for data security methods) still included DES, in some form, up until 2005 when it was replaced by the Advanced Encryption Standard (AES) (FIPS, 1977b).

DES is a traditional block cipher with a block size of 64 bits and a key size of 56 bits (FIPS, 1977a). It was developed to be resistant to differential cryptanalysis, at the expense of resistance to a brute force attack (Langford and Hellman, 1994). DES combines a Fiestel structure (over 16 rounds) with key scheduling, permutation mapping, XOR operations and block divisions (FIPS, 1977a) (Grabbe, 2006).

2.5.2 Data Transforms/Operations

The following is derived from FIPS, 1977a and Grabbe, 2006.

Sub-Keys The first stage is to generate sub-keys which will be used in the Feistal function:

1. Permute the 64 bit key, K0, with the PC-1 table, resulting in a 56 bit permuted key, K1
2. Split K1 into two divisions, K1L and K1R
3. Perform left bit shifts according to shifts table on K1L and K1R, resulting in keys K2L and K2R
4. Repeat the operation a further 15 times, using K(N-1)L and K(N-1)R as input for K(N)L and K(N)R

Initial Plain Text Transforms The plain text data is permutated with the IP64 table, resulting in a permutated block of 64 bits. The block is then split into Left (L) and Right (R) blocks, of length 32 bits. The R block is then fed to the Feistal function.

Feistal Functions The Feistal function performs the following steps:

1. Expand the input block from 32 bits to 48 bits
2. XOR the expanded 48 bit block with the round's corresponding sub-key
3. Substitute values in the 48 bit block using the DES S-Box, resulting in an output of 32 bits

4. Permutate the 32 bits with the Perm32 table
5. XOR the result from step 4 with the L block from the initial DES plain text transformations and set as the next R value
6. Set the next L value as the result from step 4
7. Repeat for 16 rounds

Final Transforms Finally, the outputs of the 16th Feistal function (i.e. the final Left and Right values) are then combined together and permutated with the IP minus 1 table. This produces the final 64 bit cipher text.

2.5.3 Breaking DES

DES is now semantically insecure because of the small key-size and increases in computing power. The DESCHALL project successfully brute force attacked DES in 1997 (Curtin, 2007). Shortly after, the Electronic Frontier Foundation built a DES cracking machine, capable of a brute force attack within days (EEF, 1998). For a short while, DES was replaced by Triple DES, until the Advanced Encryption Standard was selected as it's next successor (FIPS, 1977b).

2.6 Machine Learning Background

Machine learning has gained traction in recent years for solving complex, real-world problems without being explicitly programmed. There are two learning types: supervised and unsupervised (Murphy, 2012).

Unsupervised learning favours simpler problems, for example, using logistic regression to predict future customer spending (MacKay, 2003). Models don't require training and return results quickly.

Supervised learning has been used extensively in the computer vision field. Models are trained on data to learn historical patterns and generalise a solution MacKay, 2003. For instance, recognising an apple in an image versus a banana. Supervised learning can use several different approaches (e.g. Support Vector Machines) but this project focusses specifically on Neural Networks (Murphy, 2012).

The simplest machine learning problems are classification or regression problems (Murphy, 2012). They either learn whether an input is type A or type B, or to predict an output from a given input (Ian Goodfellow and Courville, 2016). Machine learning follows the principle of Occam's razor in that the simplest solution is usually the best.

2.7 Neural Networks

2.7.1 Structure

Neural networks have existed for around 30 years, but recent increases in computing power and development of new models have dramatically increased their popularity

(McKenna, 2016). They are based on the structure of neurons in brains McKenna, 2016.

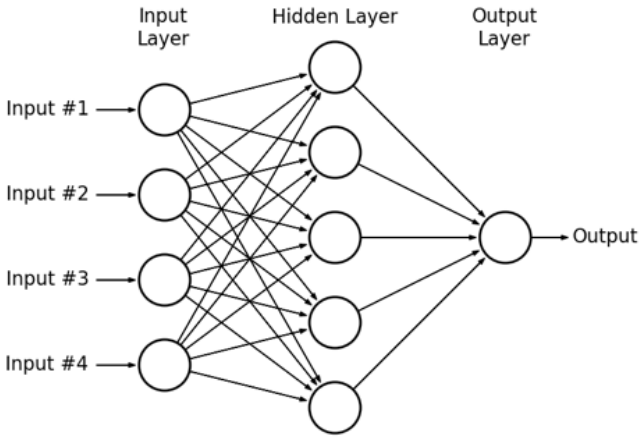


Figure 6: Diagram of a fully connected Multi Layer Perceptron (Neural Network) (Rajput, Chakravarti, and Kothari, 2015)

A typical fully-connected feedforward network comprises of several layers Murphy, 2012:

- 1 input layer, where data is fed into the network
- n hidden layers, where data is processed by the network
- 1 output layer, where the final prediction is made

Each layer consists of a number of neurons. Each neuron, usually, has a forward connection between itself and every neuron in the following layer.

$$y = \mathrm{P}(t=1 \mid \mathbf{x}, \mathbf{w}) = \frac{1}{1+e^{\mathbf{w}^T\mathbf{x}+w_0}}$$

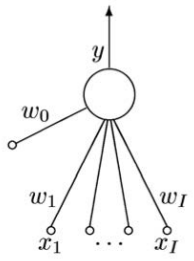


Figure 7: Weights (w_1,...,w_i), biases (w_0), inputs (x_1,...,x_i) and output (y) for a single neuron of a neural network (McKenna, 2017)

These forward connections are weighted sums in the form $y = wx + b$ (Murphy, 2012). Each connection has an associated weight value (w) which is multiplied by the input to the neuron (x). A bias value (b) is then added to the weighted sum, creating

one input for the next neuron ($y = mx + b$). All the connections to a neuron are added together to complete the weighted sum for a single neuron (Murphy, 2012). This can be seen in figure 7.

Any neuron not in an input layer also has an associated activation function, taking the combination of all weighted sums as it's input (Murphy, 2012). A typical activation function is Sigmoid, which can be seen in figure 8 below and has the form (Murphy, 2012):

$$y = \frac{1}{1 + e^{wx+w_0}}$$

For example, when:

$$x_1 = 1, x_2 = 2, w_1 = 4, w_2 = 3, w_0 = 5$$

The output y is calculated as:

$$\begin{aligned} y &= \frac{1}{1 + e^{(w_1x_1)+(w_2x_2)+w_0}} \\ &= \frac{1}{1 + e^{(1\times4)+(2\times3)+5}} \\ &= \frac{1}{1 + e^{15}} = 0.00000031 \end{aligned}$$

The output value (y) becomes the input value (x) for any connections to subsequent neurons.

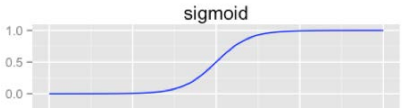


Figure 8: Sigmoid Activation Function

2.7.2 Training a Network

To train a network, the weights and biases are first initialised by assigning random values selected from a suitable random distribution. Then, one line of input data is fed to the input layer. This data feeds forward through the network's layers until it reaches the output layer, producing a real number prediction in the range{0,1}. The error between the correct output and the network's prediction is calculated. A neural network's prediction is dependent on it's weight and bias values. So all these weights and biases contribute to the prediction error (McKenna, 2016). To minimise the output error, weight and bias values are changed to values determined by the network's optimiser (Murphy, 2012).

This is backpropogation, the process through which Neural Networks "learn". Typically, backpropogation is done after a batch of data has passed through the

network (Murphy, 2012). The error calculation and the adjustments to weights and biases are performed for each batch (Murphy, 2012). Over a number of data batches, the neural network should minimise the error over time and "learn" a generalised solution (McKenna, 2016).

2.7.3 Minimising Error with Loss, Minima and Stochastic Gradient Descent

But to minimise error, how error is calculated must be defined. This is done using loss functions. Different loss functions are fundamentally different calculation methods for error (Murphy, 2012). Some simple loss functions are Cross Entropy, Log and Mean Square Error (Tensorflow, 2017b).

Next, the network's optimiser must be considered. The optimiser determines what new weight and bias values to choose, and is integral to training. There are several optimisers available, such as: Adam; ADAdelta; ADAgrad, PAD and Momentum (Dali, 2017) (Ian Goodfellow and Courville, 2016). The simplest optimiser model is Stochastic Gradient Descent (hereafter, SGD) (McKenna, 2016). SGD attempts different "jumps" in weight and bias values in order to minimise the chosen loss function (total error) (Murphy, 2012). This is demonstrated in figure 9 (McKenna, 2016).

SGD is parameterized according to a set learning rate which determines how much of a change to make to weight values per training step (Murphy, 2012). A higher learning rate produces smaller jumps, whilst a smaller value produces larger jumps (Ian Goodfellow and Courville, 2016). If the learning rate is set too high, SGD may get stuck in a local minima, without being able to find the global minima (on account of it not being able to jump out of the local minima) (Ian Goodfellow and Courville, 2016). Conversely, if the learning rate is set to low, then the neural network may take too long to discover the global minima, as it will continuously jump around different local minima (Ian Goodfellow and Courville, 2016).

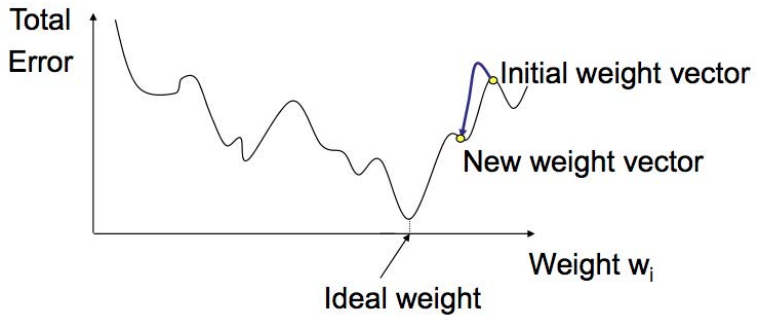


Figure 9: Diagram of the how gradient descent tries to find the global minimum

2.7.4 Testing a Network, Avoiding Overfitting and Reliability

Once trained, the performance of the network needs to be established. Various accuracy metrics exist including AUC and ROC curve tests (Zhang, 2016). The simplest

metric available is accuracy, the proportion of correct prediction values over total tests (Murphy, 2012).

Overfitting occurs when a network maps directly onto training data (Ian Goodfellow and Courville, 2016). It will generate great results when fed that data set, but will give poor results with unseen data (Zhang, 2016). To avoid this problem, data sets must be balanced and split into Training, Validation and Testing sets (Ian Goodfellow and Courville, 2016). An unbalanced data set is one where certain data points are more likely to appear in one data set than another. For example, if data is ordered sequentially then the earliest sequences may exist mostly in the Training set (Ian Goodfellow and Courville, 2016). Splitting the data into three sets means accuracy metrics can be run on three completely different sets of data, checking the solution for general applicability - as the network is only learning on the Training data (Zhang, 2016). Essentially this performs a double check on network performance. If a network is overfitting, Training accuracy will increase at a higher rate than Validation and Testing accuracies.

To check that the performance of the network is not random (i.e. a fluke), networks can be trained and tested several times, with the average of results as the final accuracy metric. This essentially performs a similar function to cross-validation (Ian Goodfellow and Courville, 2016).

2.7.5 Parameter Optimisation

There are a large number of possible parameter choices for neural networks. These choices include, but are not limited to: the hidden layer width; the number of hidden layers; the learning rate; the initial values of weights and biases; the optimiser and the loss function (Ian Goodfellow and Courville, 2016).

One could manually try many different parameters. This could work with background knowledge of the problem. However, in more complex cases, this may require an inordinate number of tests (Zhang, 2016). One way to reduce the workload for this is to automate the process through parameter optimisation (Zhang, 2016).

A grid-search parameter optmisation script will iterate through all possible values for all available parameters (Zhang, 2016). A neural network is trained and tested for each combination. Combining the results, one can see the effects on performance and select the best parameters (Ian Goodfellow and Courville, 2016). In most cases, the "best" set of parameters are those that provide higher accuracy and lower training time (Deeb, 2015).

While its useful for exploring the relationship between parameters and network performance, grid search takes a long time to produce results as its exhaustive. Another search method is random search, which chooses random parameter values from a range (Deeb, 2015). Thus rather than testing linearly, good results can be discovered earlier than with grid search. For example, a "good" learning rate parameter of 0.8 would need to iterate through 8 values (0, 0.1 0.2 etc.) until it finds an acceptable value. Random search, on the other hand, could find 0.8 on it's first test (Deeb, 2015).

3 Requirements Specification

This project's main focus is to determine if it is possible to break symmetric key encryption using machine learning. There is some previous work in breaking SDES (Alallayah and Alhamami, 2010) and DES SBoxes (Dourlens, 1996), but no published work has been found for breaking complete crypto-systems. Previous sections have shown that breaking symmetric key encryption is no small feat. It requires an inordinate amount of time and computational power for current semantically secure crypto-systems.

Initially, the project should start with the simplest crypto-systems and tasks. As more information is gathered from these simple problems, it can be used to attempt solutions at more complex crypto-systems and machine learning tasks. The project is open-ended and discovery focussed, rather than targeting fully complete solutions. It is intended as a first-stage discovery analysis for breaking full symmetric key systems with networks. So the project plan must be flexible & open to change, but with the need to aim for strict targets.

3.1 Primary Objectives

There are five primary objectives for this project:

1. Show that simple crypto-systems that are known to be insecure could be broken by neural networks
2. Break simple crypto-systems that are known to be insecure
3. Investigate whether semantically secure crypto-systems could be broken by neural networks
4. Attempt to break semantically secure crypto-systems
5. Generate some approximations of either keys or a full crypto-system

For objective 1, if a network cannot break the simplest crypto-systems, then later objectives will not be met. So objective 1 must be approached first. Objective 1 does not require a crypto-system to be completely broken, but only to show some results that suggest it can be. It is expected that this objective will be met relatively easily. To succeed at this objective, the networks must achieve significantly better than 50% accuracy in at least one test. For our purposes, an accuracy of 55% should be sufficient. Objective 2 naturally follows on from objective 1, with the added condition that the crypto-systems must be broken. To succeed at this objective, the networks must achieve over 90% accuracy. It's expected that this objective will easily be met.

Objective 3 is similar in nature to objective 1. If it is possible to show that simple crypto-systems could be broken, then it is likely that we show the same for semantically secure crypto-systems. To succeed at this objective, the networks must achieve significantly better than 50% accuracy in at least one test. For our purposes,

an accuracy of 55% should be sufficient. Objective 4, as per objective 2, follows from objective 3 and requires the networks to achieve over 90% accuracy.

Objective 5 is somewhat distinct from other objectives, but shares similarities. If it is possible to approximate a crypto-system output, for instance the key, then this reduces the time required for attack methods like brute forcing. To succeed at this objective, the networks must achieve over 55% accuracy in approximating either a key or cipher text.

Ideally, several crypto-systems should be tested for all primary objectives. But this depends on results from the project. As the project will use a limited data set (rather than exhaustive), there will be a degree of variance in accuracy results. So these arbitrary accuracy targets have been chosen to ensure that if a network's performance is 50.5%, it's not considered a successful run.

3.2 Secondary Objectives

There are also several secondary objectives that will be considered alongside the main objectives:

1. Search through network parameters and measuring their impact on accuracy
2. Demonstrate relationships, if any, between network performance and chosen parameters
3. Determine how reliably networks perform
4. Determine if networks are more efficient than other known attack methods

Secondary objectives can be considered as sub-objectives to the primaries and are less relevant to the success of the project. They are not required for the project to be determined as a success, but may provide more detailed understanding which will be useful in later analysis or future work.

3.3 Work Schedule & Project Plan

The original work schedule (as per table 7) focussed on an iterative analysis process, where the problems attempted become more complex over time. The idea is to learn from the behaviour of networks at simpler systems, using this knowledge to inform decisions for the later, more complex systems. For example, if DES can be broken with certain parameters, but they are vastly different to the parameters used for simple ciphers, then the parameters used for attacking DES should be prioritised for the 3DES problem.

Stage	Area	Task	Deadline
1	Crypto	Research	15-May
2	Crypto	Data Generation	21-May
3	Networks	Research	24-May
4	Networks	Builds	31-May
5	Simple Ciphers	Tests	01-Jun
6	Simple Ciphers	Analysis	06-Jun
7	DES	Tests	12-Jun
8	DES	Analysis	15-Jun
9	3DES	Tests	21-Jun
10	3DES	Analysis	26-Jun
11	AES	Tests	03-Jul
12	AES	Analysis	07-Jul
13	CNN Key Approximation	Tests	21-Jul
14	CNN Key Approximation	Analysis	27-Jul
15	CNN Full System Approximation	Tests	10-Aug
16	CNN Full System Approximation	Analysis	16-Aug
17	Finalisation	Compile Analysis	25-Aug
18	Finalisation	Report Write up	11-Sep

Table 7: Work Schedule for the project, with each crypto-system analysed in turn

4 Design

The design process focused on several key principles:

1. Tests should be focussed on discovering properties, rather than attempting full solutions
2. Tests should be performed iteratively, using previous results to inform future tests
3. Tests should increase in complexity, with higher complexity exposing more information
4. Networks must reliably handle large data sets and large and complex network structures
5. Code should be dynamic, flexible, efficient and as Pythonic as possible

4.1 Testing Tasks

The objective of the project is to discover if symmetric key encryption can be broken using machine learning techniques. To do this, there were four machine learning tasks designed to tell us something different about a crypto-system's behaviour. Each one increases in theoretical complexity with more useful information being exposed to an adversary.

4.1.1 Classification Task 1 (C1)

C1 aims to test whether it is possible to classify between a cipher text and a random string for a specific crypto-system. The definition of semantic security requires that crypto-systems should not produce cipher text bits that have a probability of being guessed greater 50%. Cipher texts generated from a semantically secure crypto-system should essentially seem like random data.

So if machine learning has any hope of breaking symmetric key encryption, it must be able to detect some non-random behaviour of the crypto-systems involved. It must be able to differentiate between a cipher text and a pseudorandom generated text. This is the simplest task possible, as it is fundamentally testing the semantic security of a crypto-system.

For C1, the networks will be provided with either a plain and cipher text pair (from a single key) or a plain and random text pair. The networks will then attempt to classify these inputs as either True or False (1/0) values respectively.

4.1.2 Classification Task 2 (C2)

Following on from C2, the requirement of 50% random guess probability not only applies to cipher texts generated from a single key against random data, but it also applies to cipher texts generated with different keys. If it is possible to determine

whether a specific cipher text, from a set of cipher texts, has been generated from a specific key and given plain text, then the system is no longer semantically secure. This task is more complex than C1 as the machine learning algorithm will need to learn any non-random behaviour of a crypto-system and be able to interpret it's effect on different outputs.

So C2 determines whether we can differentiate between different keys for the same plain text. The networks will be given a plain text and one of four possible corresponding cipher texts. It will then attempt to classify these inputs as either True or False (1/0) values. A True value will refer to cipher texts generated with key 1, and cipher texts generated from other keys as False.

4.1.3 Regression Task 1 (P1)

If a correct cipher text can be generated from a given plain text, then the it should be possible to produce more cipher texts for different plain text values. This will involve the networks reproducing the steps of a crypto-system, but will not expose any information about the crypto-system itself. An adversary could only use this as an attack vector for a single key, for a single crypto-system. This is an extension of the C2 task, but adds more complexity by requiring the network model the crypto-system, rather than detecting some non-random behaviour.

So P1 attempts to reproduce the entire crypto-system within a neural network. The networks will be given plain texts as the only inputs. It will then be trained to return the values of the cipher text.

4.1.4 Regression Task 2 (P2)

Finally, P2 is the most complex task attempted. To fully break a crypto-system, the key used for encryption needs to be retrieved. Doing this for any set of pairs of plain and cipher text data would make a crypto-system semantically insecure. It would enable an adversary to read all messages generated for a key. This is considered the most complicated task, as it must model the crypto-system and then output the key.

So P2 attempts to reproduce the entire crypto-system within a neural network, to output the value of the key used. The networks will be given plain texts and cipher texts for a single key. It will then be trained to return the values of the key.

4.2 Analysis Process

This problem needs to be approached from first principles, as it has not been completely solved with neural networks before. So by starting with simple tests and increasing their complexity, more is learnt about the nature of chosen crypto-systems as progress is made. Each time complexity is increased, the analysis should reveal more about the behaviour of crypto-systems.

The design of the analysis considered three main areas - choices of crypto systems to test, the types of networks to use and the parameters to use with those networks.

4.2.1 Crypto-System Choices

As per primary objectives one and two, the project needed to start with simple crypto-systems, subsequently moving onto more complex systems. To ensure the project also has reliable results, a range of crypto-systems need to be tested.

To satisfy these targets, the project first looked at the example simple ciphers as detailed in the background section (MTP, Caesar, Substitution Ciphers). By performing tests on three different ciphers the project should demonstrate that neural networks can be generally applied to cipher problems, rather than succeeding at a single cipher system. The three simple ciphers also perform similar operations to typical components in symmetric-crypto-systems. Being analogous to these components, it was assumed that simple cipher results will be analogous to results for symmetric-key components. So, understanding from their analysis will be used to try to break symmetric-key crypto-systems. It should be noted that the P2 task will not be run for the Simple Substitution cipher, as the key mapping is too complex to model without designing extra neural networks.

DES is a semantically insecure symmetric-key crypto-system that has been broken through several different attack vectors (Roeder, 2010). So, it makes sense to move to DES next. Following from DES, the security can be increased slightly by attempting 3DES, then moving on to semantically secure crypto-systems such as AES and Blowfish. Both AES and Blowfish are able to use different key sizes (with increased key sizes increasing the security) (*Applied Cryptography* 2016). The project focussed on starting with the smaller key sizes and move to the larger key sizes. Again, increasing complexity over the course of analysis.

4.2.2 Neural Networks Choices

Several types of neural networks were researched for this project. These are listed below, with the simplest first:

- Simple Neural Networks - A traditional feedforward multi-layer perceptron (as discussed in section **??**) with one hidden layer
- Deep Learning Networks - A traditional feedforward multi-layer perceptron (as discussed in section **??**) with n hidden layers for deep feature extraction
- Convolutional Neural Networks (CCNs) - Processing input values through convolution and pooling for approximations (McKenna, 2016)
- Long-Short-Term-Memory Networks (LSTMs) - Uses memory units to process data that is sequential, where a sequence value s_t (where $t = time$) will be defined by the value of s_{t-1} (Colah, 2015)
- Neural Turing Machine Networks (NTMs) - Learning an algorithmic computation like copying or sorting (Alex Graves and Danihelka, 2014)

Traditional machine learning analysis usually assumes that the simplest general solution is the best. Bearing this in mind, the project aimed to use the first three

network types (Simple, Deep and Convolutional) throughout. As before, the project started with the simplest networks, progressing towards more complexity as more information was gathered.

However, the CNNs were a special case. Rather than providing exact results, CNNs specialise in approximating results (McKenna, 2016). This is why they have seen excellent use in the field of computer vision. They use convolution layers and pooling layers alternatively to extract useful features from input data (McKenna, 2016). These useful features are then processed by a fully connected layer (as per the simple neural network), producing a prediction (McKenna, 2016). In layman's terms, one picture of a banana will not be exactly the same as another picture. But we can assume that the banana will look similar in both photos. So, an approximation of a banana can be recognised, the banana can probably be recognised.

With this in mind, tasks P1 and P2 should be attempted with a CNN so either cipher texts or keys might be approximated.

4.2.3 Parameter Choices

With neural networks, there are thousands of possible parameter choices that could lead to a solution. So it is imperative that limited set of parameters are chosen for testing. This limited set needs to provide plenty of possible combinations to test, whilst trying to retain simplicity in the networks.

Activation Functions We know that the networks will generally have to perform a lot of bitwise operations to solve the crypto problems (*Applied Cryptography* 2016). So in the case of output activation functions, we can assume that a network will perform better with functions that operate between a 0 or 1 value. So the Softsign and Sigmoid activation functions were chosen to be tested in output layers (Tensorflow, 2017b). However, in the interest of confirming this, the Softmax activation function was used too. Softmax should not work in these problems and should return poor results. Sigmoid was used as the default setting for initial tests. For hidden activations, there is a need to aim again for bitwise favouring functions, but also to consider the vanishing gradients problem. With many hidden layers, functions like Sigmoid can eventually tend to 0. Functions like Elu and Relu are specifically designed to combat the vanishing gradients problem, so these were chosen to be tested in hidden layers (Tensorflow, 2017b). However the Sigmoid function is the simplest available, so this was included in testing as well and used as the default for initial tests.

Optimisers Optimisers are harder to choose than activation functions, but we made some early decisions. The simplest optimiser is Stochastic Gradient Descent (SGD), so this was used for initial tests. ADAgrad, ADAdelta and ProximalAdagrad (PAD) were chosen for testing as they specialise in optimisation on sparse data(Dali, 2017). Whilst the networks' input data isn't sparse (50% probability of a 0 value leads to 0.5 sparsity), it will contain many 0 values. So, these optimisers may return good results in testing.

Loss Models For loss models, the simplest choice is Cross Entropy so this was used as the default for initial tests. Tensorflow has 17 choices of loss model at the time of writing (Google, 2015). Mean-Square-Error and Log losses were chosen as they're relatively simple traditional models, but may give good results. Hinge loss was also chosen as it typically used for maximum-margin classification (Rennie and Srebro, 2015).

Triplet loss was also considered, but never implemented.

4.3 Languages & API Choices

Several technologies will be used for different aspects of the project.

Python - Simple Ciphers Simple ciphers are easy to implement (they can be done by hand) and basic functions can be written in any programming language. This project used Python (version 3.6) to generate simple cipher data files. Python is a cross-platform "interpreted, interactive, object-oriented programming language". The main focus of Python has always been clear and readable code (Python, 2017a) so it is easy to use and to debug. So Python was chosen as the language for the simple cipher data generation. Python also provides dynamic behaviour (e.g. data types) (Python, 2017a), so it is possible to generate multiple outputs of different data types. This allows for testing data sets to ensure correctness.

PyCryptoDome - Symmetric-Key Crypto-Systems There are well known software libraries such as OpenSSL that perform a encryption for a multitude of crypto-systems (OpenSSL, 2016a). OpenSSL does allow for use with programming languages (OpenSSL, 2016b), but there are also external Python packages (Python, 2017b) that can perform symmetric key encryption such as PyCryptoDome (PyCryptoDome, 2017) in a much simler method. Using Python to generate crypto-system data meant that the project had a single language base and retained simplicity, as Python was already being used for simple cipher data. However, PyCryptoDome requires Python 3.4, so a different base environment needs to be used. Initially, Docker containers were used, but the added complexity and implementation issues (as detailed in section **??**) meant virtual environments were used later on. Again, using Python meant multiple outputs of different data types can be generated. This allows for testing data sets to ensure correctness.

Hardware Generating data files for both the simple cipher and full crypto-system was designed to be a relatively fast operation, so the programs were designed to be run on typical desktop hardware. But, because the project attempts to solve a complex and unproven problem, it was imperative to use hardware that can perform machine learning quickly and with large data sets. The project used the Department of Computing's three GPU servers for running large machine learning problems, the specifications of which can be seen in table 8.

Server Name	Number of GPUs	GPU Type	Total GPU Memory (MiB)
Rhea	1	Tesla K40c	12,204
Tesla	1	TITAN X (Pascal)	12,189
Titan	2	TITAN X (Pascal)	24,378

Table 8: UoD - Department of Computing GPU Server Hardware

Tensorflow API There are several popular open-source libraries for building neural networks (Nachum, 2016). One of the more recent is Tensorflow - an API released for Python by Google's DeepMind project (Google, 2015). It features multi-GPU support and focusses on distributed architecture to speed up computations and reduce learning time (Tensorflow, 2017c). As it is developed in Python, code in Tensorflow can written to take advantage of dynamic types and iterables. There are also other benefits, as discussed in the Rationale for Design & Implementation Decisions section.

Results Graphs Rstudio was used with the ggplot package to generate the heat maps and point graphs.

5 Implementation and Testing

5.1 Data generation

It should be noted that any discussion of "random" numbers actually refers to pseudo-random number generation. Most data was generated almost exclusively with Python 3.x, so the Mersenne Twister was the core random number generator. The data generation programs exclusively used the random uniform distribution (Python, 2017c).

5.1.1 CipherGen Program

As discussed previously, plain, cipher and random text data sets needed to be generated in such a way that:

- They have no inherent bias that would affect network results
- They give the networks some advantage
- Entries are not repeated

The plain, random and cipher texts for the Cipher encodings are all generated from *CipherGen.py*. The argparse module was used to provide a functional command line interface, requiring the user to define:

- Which Cipher to generate data for
- Where the data should be written to
- How large the data set should be
- What block sizes should the data be generated for

The user can optionally define the number of keys to generate cipher text data with as well. If this option is not chosen, then it is defaulted to 4.

The Caesar cipher data sets are generated by selecting random numbers within a set range and converting them to binary representations. Outputs are appended to data files to allow an increase to the size of the data set being worked with, if required.
The pseudocode for the process (given block size b, data set size n, keys k, maximum integer value m) is:

1. Randomly select k integers between 0 and 25 as keys
2. For each k:
    2.1. Append k to the key .txt file

3. Calculate the max and min values for all keys and calculate a max and min value for text generation from m

4. For each b:

    4.1. Until n data points are generated for plain and random texts:

        4.1.1. Randomly choose two integers (one for plain, the other for random texts) between min and max

        4.1.2. Convert the integers to binary representations

        4.1.3. Append the binary representations to respective .txt files

    4.2. For each k:

        4.2.1. For each plain text:

            4.2.1.1. Convert the binary representation back to integer

            4.2.1.2. Add the key value to the integer

            4.2.1.3. Convert back to binary representation

            4.2.1.4. Append binary representation to the cipher text .txt file for the key k

The Simple Substitution cipher data sets are generated by randomly selecting from a pre-defined set of 36 ASCII characters (c) and converting them to binary representations. Outputs are appended to data files to allow an increase to the size of the data set being worked with, if required. It should be noted that keys are not written for the Simple Substitution cipher, as the mappings are considered too complex for the networks to be used.

The pseudocode for the process (given block size b, data set size n, keys k) is:

1. For each k:

    1.1. Randomly generate a Substitution dictionary mapping

2. For each b:

    2.1. Repeat n times:

        2.1.1. Randomly select b characters from c for plain texts

        2.1.2. Append the result to a .txt plain text file

        2.1.3. Randomly select b characters from c for random texts

        2.1.4. Append the result to a .txt random text file

        2.1.5. For each k:

            2.1.5.1. Create cipher texts by swapping the plain text characters using the mapping in k

        2.1.6. Append each cipher text to it's own .txt file

        2.1.7. Append each k to the keys .txt file

The MTP data sets can be generated in two ways. The first is for small pad values. As the length of the pad decreases, the total possible combinations of data points decreases too. This leads to duplication in extreme circumstances. For small pad sizes, (< 23 bits), plain texts and random texts are generated using all possible combinations of bit values. Any other pad sizes (> 22 bits) are generated by randomly selecting bit values for the block length. The plain texts are then XORed with some randomly generated pads to create cipher texts. Outputs are appended to data files as binary representations. This allows us to increase the size of the data set we are working with, if required.

The pseudocode for the process with pads of block size < 23 (given block size b, data set size n, keys k) is:

1. For each b:
    1.1. Randomly generate k pads of length b containing $0, 1$ values
    1.2. Generate all possible combinations of $0, 1$ values for length b, producing an array A
    1.3. Randomly sample the A twice (one sample is plain texts (P), the other is random texts (R))
    1.4. Write the P and R to respective .txt files
    1.5. For each k:
        1.5.1. Append k to the keys .txt file
        1.5.2. For each p in P:
            1.5.2.1. XOR k with p
            1.5.2.2. Append the XOR output to the .txt cipher text file for k

The pseudocode for the process with pads of block size > 23 (given block size b, data set size n, keys k) is:

1. For each b:
2. Randomly generate k pads of length b containing $0, 1$ values
    2.1. Repeat n times:
        2.1.1. Randomly select an integer between 0 and 2^b and convert to binary representation for plain text
        2.1.2. Append the plain text entry to the plain text .txt file
        2.1.3. Randomly select an integer between 0 and 2^b and convert to binary representation for random text
        2.1.4. Append the random text entry to the random text .txt file
    2.2. For each k:
        2.2.1. Append k to the keys .txt file

2.2.2. For each p in P:
2.2.2.1. XOR k with p
2.2.2.2. Append the XOR output to the .txt cipher text file for k

5.1.2 CryptoPlainGen

As with the simple Ciphers, plain, cipher and random text data sets needed to be generated in such a way that:

- They have no inherent bias that would affect network results
- They give the networks some advantage
- Entries are not repeated

The plain, random and cipher texts for the crypto plain texts are all generated from *CryptoPlainGen.py*. The argparse module was used to provide a functional command line interface, requiring the user to define:

- What type of text to generate (plain or random)
- Where the data should be written to
- How large the data set should be
- What block sizes should the data be generated for

The plain text data set is generated by randomly selecting from a pre-defined set of 36 ASCII characters (c) and converting them to binary representations. Outputs are appended to data files to allow an increase to the size of the data set being worked with, if required.

$$c = \{ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789\}$$

The pseudocode for the random texts process is:

1. For each block size b:
    1.1. Until n data points are generated:
        1.1.1. Randomly choose b times $0, 1$ values
        1.1.2. Append the entry to the random texts .txt file

The pseudocode for the plain texts process is:

1. For each block size b:
    1.1. Until n data points are generated:
        1.1.1. Randomly choose b times characters from c as plain text entry
        1.1.2. Convert the plain text entry to a binary representation
        1.1.3. Append the entry to the plain texts .txt file

5.1.3 PyCryptoDome

Originally, the PyCrpytoDome module was used to generate the crypto-system cipher texts. PyCryptoDome required Python 3.4 so a virtual environment was installed and used for the encryption process (PyCryptoDome, 2017).

For each crypto-system, an individual script was created. The typical process for encrypting data was:

1. Generate k random keys using the $get\_random\_bytes()$ function in PyCryptoDome
2. For each k:
    2.1. Convert k to a string binary representation
    2.2. Append k to the keys.txt file
    2.3. For each row of data in a given plain text file:
        2.3.1. Encrypt the plain text data using k
        2.3.2. Convert the cipher text data from base64 (Python bytes format) into ASCII
        2.3.3. Write cipher text data to the following k cipher text .txt files
            - ASCII characters
            - Binary representation as string
            - Integer values of bytes
            - Hex values of bytes

This was tested first by using the $mini-tests.py$ script. This was a simple testing script which took plain or cipher text inputs and respectively encrypted or decrypted them using a given key. The script then printed out the resulting cipher or plain text which could be compared to the data in the file. The multiple representations of data were also used to double check values across files.

However, the manual nature of these checks led to an issue further down the line. PyCryptoDome outputs the cipher texts in Base64 format (PyCryptoDome, 2017). If there was a space character present when the PyCryptoDome scripts converted the cipher texts from Base64 into ASCII, the script would write the right hand side (from the space character onwards) of the Base64 data to the binary string representation cipher text file.

This issue was checked further by checking the length of all lines of the cipher text data files for each crypto-system. There was only one case of this issue present and it was easily fixed.

5.1.4 DES Python Implemntation

Because of the poor results from network tests with full DES from PyCryptoDome, a completely new program was written to provide:

- Outputs from each DES component (Feistel or not)

- Outputs from each round of DES
- The sub-keys used
- The final cipher text outputs

The new DES cipher texts were generated from *DES.py*. This program was based on a guide which detailed the operations performed at each stage of DES, but did not provide any code (Grabbe, 2006). However, there were examples of data inputs and outputs for each step of DES.

DES.py can be used to generate cipher texts on a per component basis, or for an entire run of the DES crypto system, writing intermediate outputs to .txt files after processing. Rather than using Base64, hex or any other representation of the data, *DES.py* only allows strings to be used. This simplified the process as plain text data is written as strings in the form 0101010.

The argparse module was used to provide a functional command line interface, requiring the user to define:

- The location of the plain texts .txt file
- The directory for output .txt files
- The type of run to perform (individual components or a full run)
- The number of plain text lines to process

If performing a full run of DES, the number of rounds to process and a file with keys are required. If performing a single component run, the component run is also required. Each section of DES is broken down into individual Python functions. Each time a function is used in a full run, the program writes the result to a specific .txt file. If running for a single component, the program calls only the required function and writes the output to a single .txt file.

5.1.5 Single Component Rounds

To test how much of an impact the rounds of DES were affecting results, a simplified rounds program was written. Again, argparse was used to provide command line functionality and required input file (plain text) and output file (cipher text) locations. The program itself calls the f_XOR, $Xpansion$ and $init\_split$ functions from the DES program. It uses these to perform multiple rounds of the $Xpansion$ function on data. Limited testing was performed, as the program uses functions from the DES program, which was known to work. But outputs were compared to inputs manually to confirm whether the program was behaving correctly.

5.1.6 Testing

The programs were generally tested by manually checking the data that was created and comparing against required output values. The python *print*() function was used extensively to check intermediate outputs in the programs when there was an observable issue with final outputs. The $dist\_freq\_checks\_des - bash.py$ file was also used to check the frequency distribution of bit values across the data sets. The random and cipher text data sets should always have returned a random uniform distribution. Depending on the crypto-system, the plain text data either returned non-randomly distributed values, or a random uniform distribution.

The original cipher texts from the PyCryptoDome implementation were used to check the correctness of outputs for $DES.py$. Each time a full DES run was performed, the outputs were compared. If there were any lines that did not match, then the example inputs and per component outputs were checked against the DES guide's provided examples. If a component was processing incorrectly, then this would quickly become apparent.

5.2 Neural Networks

The neural network programs were written in Python3.6 and used the Tensorflow v1.1 API. There were 3 Python files created:

- run_models.py - The main py file for running networks, and setting up network runs
- models.py - Definitions of networks and the steps for running
- funx.py - Data preparation and transformation

5.2.1 argparse

The argparse module was used to provide arguments from the command line to the $Run\_Models.py$ program. This was done to ensure network parameters did not need to be pre-defined within files and to be able to provide ranges for values to the networks. argparse is also recommended in the Tensorflow Documentation.

The following arguments are required by argparse:

- Plain Text file location
- Random Text / Key file location
- Cipher Text file locations
- Results Name
- Problem (C1, C2, P1, P2)

Optional arguments were also allowed. If they were not provided then default values were set for each. The following optional arguments allowed for multiple inputs for use in parameter optimisation:

- Number of Hidden Layers
- Number of Neurons per hidden layer
- Learning Rates
- Hidden Layer Activation Functions
- Output Layer Activation Functions
- Optimisers
- Loss Functions
- Initialisation Functions
- Initialisation Variables Values

The following optional arguments only allowed for single values:

- Number of Epochs
- Batch Size
- Data set size
- Run Type
- Number of Reliability Runs
- Number of Random Search Parameter Optimisation Tests

For each argument, values were checked to ensure they were of the correct type and (if a number) the correct range. The arguments were then passed to the main function to be used with the neural networks.

5.2.2 Run Types

Once argparse has passed the arguments to the *main* function in $Run\_Models.py$, the program must select the type of run the network needs to perform. There are 5 choices available:

- Basic Test - A single run is performed for the arguments provided
- Grid Search Parameter Testing - Multiple runs are performed for the range of arguments provided

- Random Search Parameter Testing - Multiple runs are performed for the range of arguments provided
- Data Size Testing - Multiple runs are performed for various data set sizes
- Time Taken Testing - Multiple runs are performed for various data set sizes and batch sizes

All the run types generally follow the same execution pattern:

1. Load the data sets
    1.1. Generate the training input and output data sets
    1.2. Generate the validation input and output data sets
    1.3. Generate the testing input and output data sets
2. Generate a dictionary of parameter values
3. Run the network for the specific parameters, returning metrics reults
4. Store network results data in specified results files

There are some small details that vary between the run types. The basic run type also output produces Tensorboard data files, which will be discussed later. The grid search parameter testing run type generates iterates through the range of arguments passed to it. For example, if learning rates of 0.2 and 0.8 are given, then networks are be run for each single decimal value in ascending order (i.e. $0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8$). Both the data set size and time taken run types execute in the same way as the grid search run type, except they iterate over specific variables only (data set size; data set size and batch size). The random search parameter testing run type executes similarly to the grid search type, except that values are randomly generated using the random uniform distribution for each run. The total number of random search runs is defined by the 'Number of Random Search Parameter Optimisation Tests' argument instead of an iterable.

Data set size and time taken testing types were not used during the analysis and will not be discussed in any detail.

5.2.3 Data Preparation and Conversion

Data must be transformed from binary strings in multiple files and processed so that it can be passed networks. This task is only performed when the network is initially run. The type of transformations performed depends on the machine learning task being performed (C1, C2, P1, P2). The *Run_models.py* file calls the *get_and_prepare_data*() function. *Run_models.py* then performs the following steps, with d defined as the required data set size.
For the C1 task:

1. Load d plain text data from plain text file

2. Load d cipher text data from a single cipher text file
3. Load d cipher text data from the random text file
4. For d entries:
    4.1. Randomly choose whether to use a random text or a cipher text entry
    4.2. Concatenate either the random or cipher text from the previous step together with the plain text entry (Network input)
    4.3. Create labels of 0 or 1 values depending on whether a random of cipher text was chosen
5. Generate training, validation and testing data sets from the input and labels lists

For the C2 task:

1. Load d plain text data from plain text file
2. Load d cipher text data from multiple cipher text files
3. For d entries:
    3.1. Randomly choose which cipher text entry to use
    3.2. Concatenate the cipher text from the previous step together with the plain text entry (Network input)
    3.3. Create labels of 0 or 1 values depending on which cipher text was chosen (1 for the first, 0 for all others)
4. Generate training, validation and testing data sets from the input and labels lists

For the P1 task:

1. Load d plain text data from plain text file
2. Load d cipher text data from a single cipher text file
3. Use the cipher text entries as the labels, plain texts as inputs
4. Generate training, validation and testing data sets from the input and labels lists

For the P2 task:

1. Load d plain text data from plain text file
2. Load k keys data from key file
3. Load d cipher text data from multiple cipher text files

4. For d entries:

    4.1. Randomly choose which cipher text entry to use

    4.2. Concatenate the cipher text from the previous step together with the plain text entry (Network input)

    4.3. Use keys as labels values depending on which cipher text was chosen

5. Generate training, validation and testing data sets from the input and labels lists

5.2.4 Network Model Definition

Network models are defined per reliability run and are determined by the parameter dictionary passed to the run_tf_nnet function from the $Run\_models.py$ file. In Tensorflow, a single network is contained in a session. Variables in a Tensorflow network do not actually hold any data (Google, 2015). Instead, data passes through them like "buckets". For each reliability run (r) a new session is created.

In general, for each r, the run_tf_nnet function does the following:

1. Initialise results storage variables
2. Calculate the widths of input and label data for the network, w_i and w_l
3. Calculate the the number of batches, b, required
4. Initialise the Tensorflow session, setting memory usage to increase as required
5. Define the input and prediction placeholders (x, y)
6. Define the network layers, selecting the appropriate parameters for each layer
7. Define how predictions (p) are made, and what counts as a correct prediction
8. Define how accuracy is calculated
9. Define the loss function to use
10. Define the optimiser to use, and how the network should use this for training
11. Initialise all the layer values
12. For each epoch e:

    12.1. For each b:

        12.1.1. Create the x and y batches

        12.1.2. Train the network on x and y

        12.1.3. Calculate b training, validation and test accuracies

12.2. Calculate e validation and test accuracies

12.3. Check for overfitting

13. Calculate the AUC for the e test data

5.2.5 Metrics

The networks uses accuracy, area under curve (AUC) and accuracy per epoch as the metrics for success. Accuracy, a, is defined as:

$a = (n\_c/n\_p) \times 100$

Where:

n_c = number of correct predictions
($n\_c = 1$ if label = rounded value of prediction, else $n\_c = 0$)
n_p = number of predictions

The AUC metric is defined in the Tensorflow API and uses the $tf.metrics.auc()$ function. The AUC metric was implemented as a double check on the accuracy metric. A large difference in values between accuracies and AUC can be used to check for overfitting issues.

5.2.6 Tensorboard Outputs

The Tensorflow API can also write out results files which can be used in the Tensorboard program (Tensorflow, 2017a). Doing this requires declaring which Tensorflow variables are to be measured in the network definitions, using the following pattern:

1. with $tf.name\_scope(variable\_name)$:
2. define the variable
3. define what to record

The files are then written to disk and can be viewed using the Tensorboard program. This is only implemented for the Basic run type.

Neural Network Code Testing Generally, all the code for the Neural Networks was tested by eye, using the Python $print()$ function or the Tensorflow $tf.Print(tf.eval(x))$ pattern where appropriate. Outputs were compared to expected values and any issues were noted and fixed.

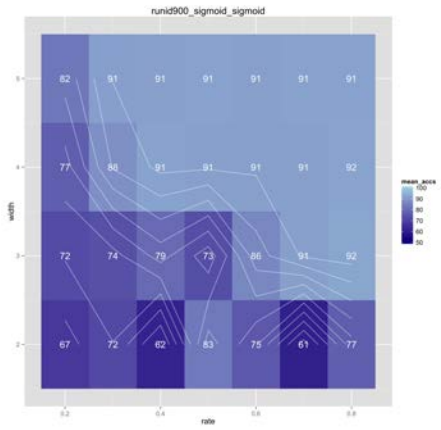
5.3 Graphics and Analysis

Graphs were generated in Rstudio using the ggplot package. Originally, graphs were implemented using the matplotlib python module, but ggplot allows for more complex visualisations. The heatmap graphs for grid search results (as per figure 10) were generated using the ggplot contour and raster geoms. The random search graphs (as per figure 20) used point geoms. The graphs were checked by comparing values in the results files versus the values on graphs for specific points.

6 Evaluation - Ciphers

6.1 Multi-Time Pad - 20 bit key

Figure 10: Average accuracy for basic grid search parameter optimisation for the MTP C1 task



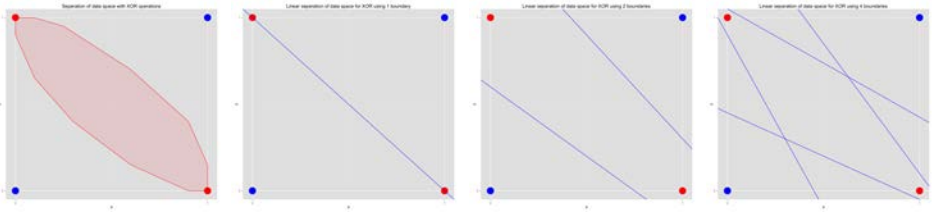
C1 - Initial Test Run The MTP cipher is easily classifiable by a network for the C1 problem, with results as high as 90%, as can be seen in figure 10. Increasing the number of hidden layer neurons reduces any dependancy on learning rate, and increases accuracy. Table 9 shows that increasing the number of hidden layer neurons also reduces the epochs required for the network to converge towards a solution.

Table 9: Average Epochs required for convergence per hidden layer width

Width	Epochs
2	187
3	180
4	163
5	149

To solve the C1 for a XOR operation, a neural network needs a minimum of 2 hidden layer neurons. This is demonstrated in figure 11, which shows estimated decision boundaries for classifying XOR operations. A single linear boundary cannot split the points correctly.

Figure 11: Decision Boundary when classifying a XOR operation



However, two linear boundaries can create a useful decision boundary. Furthermore, increasing the number of linear boundaries (i.e. the number of neurons) will result in a more accurate representation of the actual decision boundary.

Figure 12 shows the Tensorboard visualisations for a network with 5 nodes in the hidden layer and a learning rate of 0.8. Graphs 4 & 5 in figure 12 demonstrate how the biases and weights perform a Logical computation as a bitwise neural network.

Figure 12: Trained values per epoch of an example network's parameters

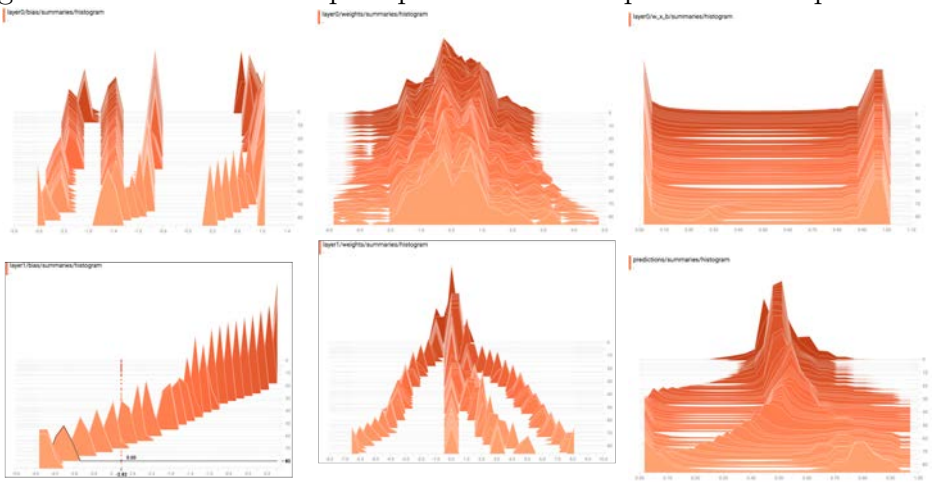
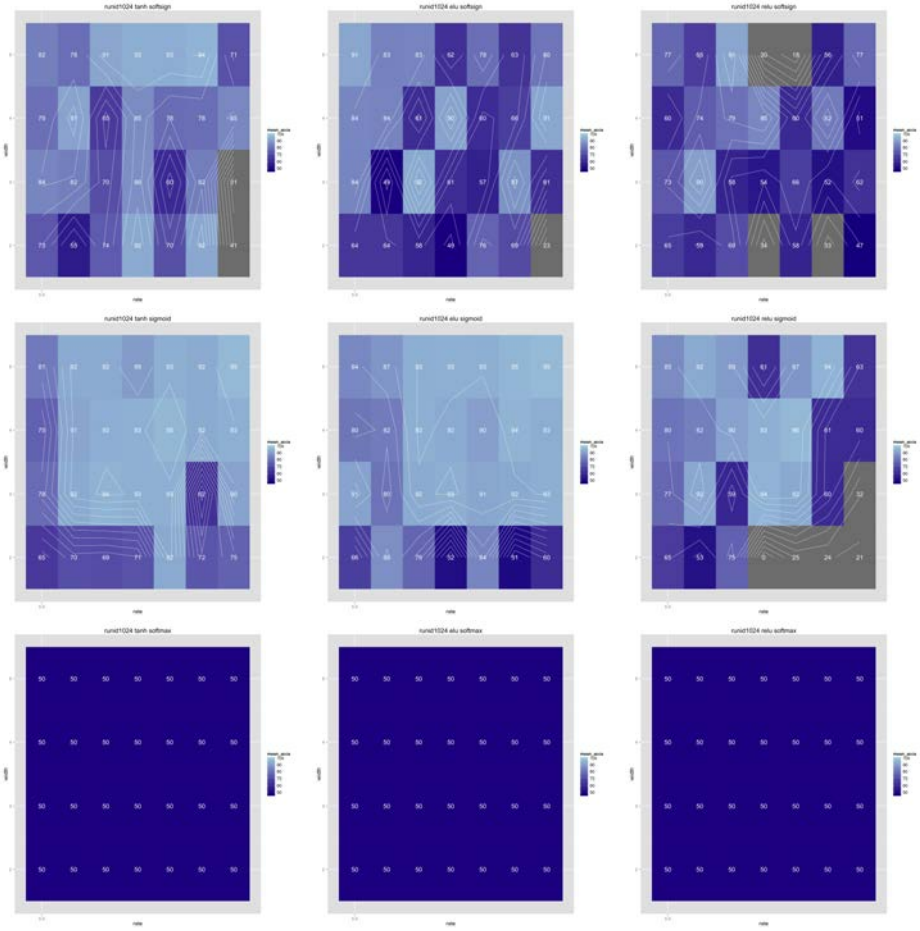


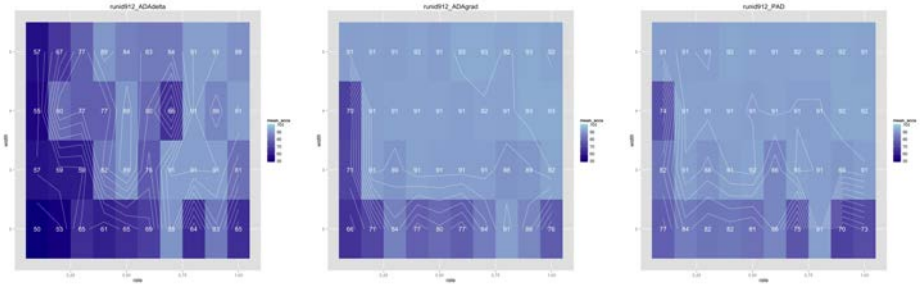
Figure 13: Average accuracy for grid search parameter optimisation of activation functions for the MTP C1 task



C1 - Activation Function Test Runs Figure 17 shows that Sigmoid is the best output activation function for C1. Somewhat surprisingly, Softsign, as an output activation function, provides some good results. But there is no clear relationship between excellent or good results. There seems to be some randomness in its performance. Softmax does not provide any results better than 50% probability.

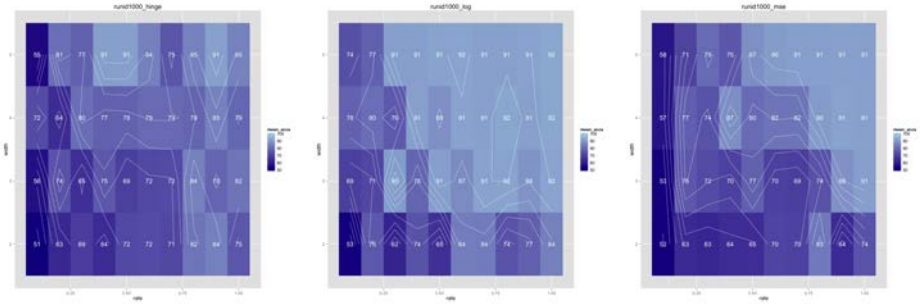
For hidden activation functions, Tanh; Elu and Sigmoid produce excellent results with a Sigmoid output activation function. Tanh and Elu perform slightly better than Sigmoid, with Tanh providing the highest results across the parameter space. Relu produces some good results as a hidden activation function, but is somewhat unreliable

Figure 14: Average accuracy for grid search parameter optimisation of optimisers for the MTP C1 task



C1 - Optimiser Test Runs Figure 14 shows, surprisingly, Gradient Descent (figure 16) performs better than ADAdelta. However, the ADAgrad and PAD optimisers perform better than Gradient Descent (figure 16). ADAgrad and PAD both specialise in training on sparse data sets (Dali, 2017). So it is interesting that ADAgrad and PAD perform well.

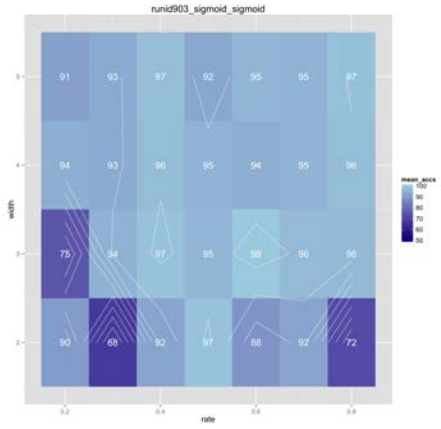
Figure 15: Average accuracy for grid search parameter optimisation of losses for the MTP C1 task



C1 - Loss Test Runs Figure 15 shows that Cross Entropy performs well for this problem, even though it is reliant on higher learning rates and wider hidden layers. Log loss provides some better results for thinner hidden layers, requiring smaller learning rates, but there are some areas within the space providing higher results that seem to be less reliable, e.g. a learning rate of 0.4 and a hidden layer width of 3.

MSE provides generally good results, except for very small learning rates. However, to obtain excellent accuracies, it requires wider hidden layers and higher learning rates than both Cross Entropy and Log losses. Hinge loss performs well, but does not provide accuracies as high as the other loss models.

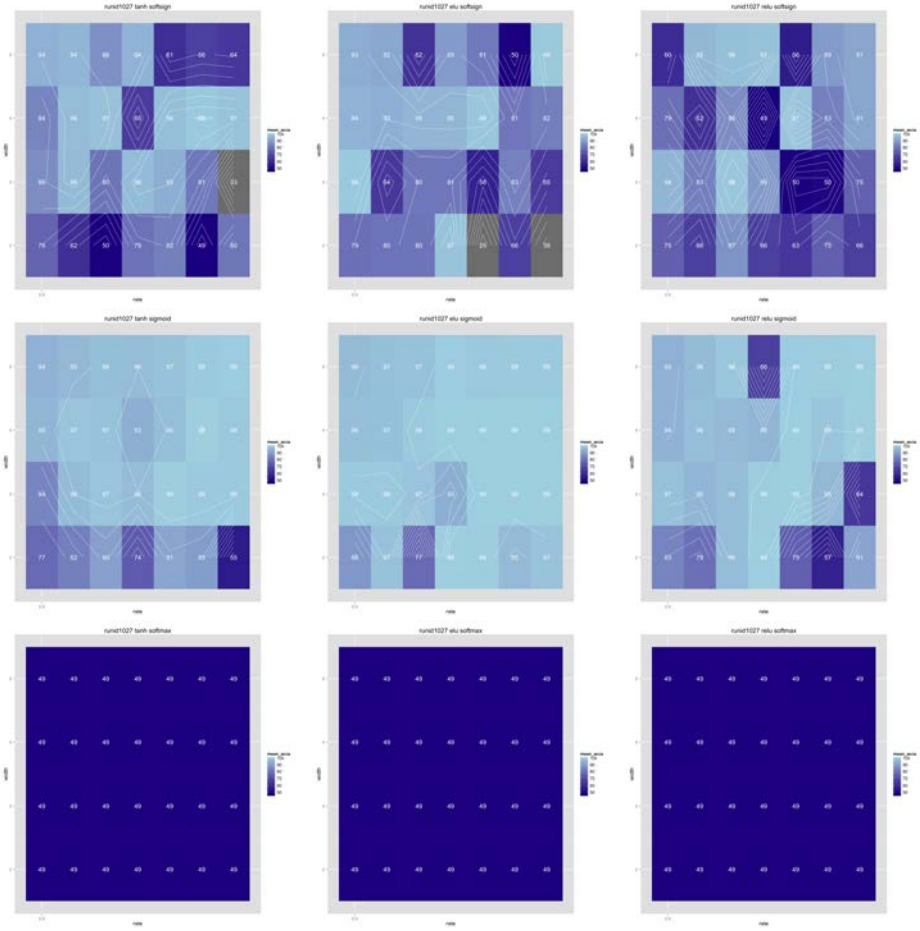
Figure 16: Average accuracy for basic grid search parameter optimisation for the MTP C2 task



C2 - Initial Test Run The C2 task was designed as a more complex problem for the networks to attempt to solve. However, figure 16 shows that, actually, the networks can accurately perform this classification for MTP20 with simple parameters. There are some areas (e.g. learning rate of 0.4) that achieve close to 100 percent accuracy over 3 runs.

This is likely due to the nature of the XOR operation. Each randomly generated key is likely to have a different bit value at one or two of its positions. So, the network only needs to check one or two input bit pairs to discern between two different keys.

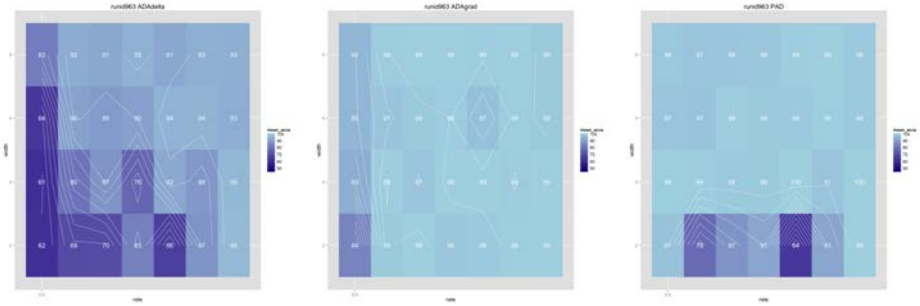
Figure 17: Average accuracy for grid search parameter optimisation of activation functions for the MTP C2 task



C2 - Activation Function Test Runs As with the C1 task, figure 17 shows that the sigmoid output activation function performs best. Softmax, again, provides no useful results, and Softsign, provides some good results but they do not seem reliable.

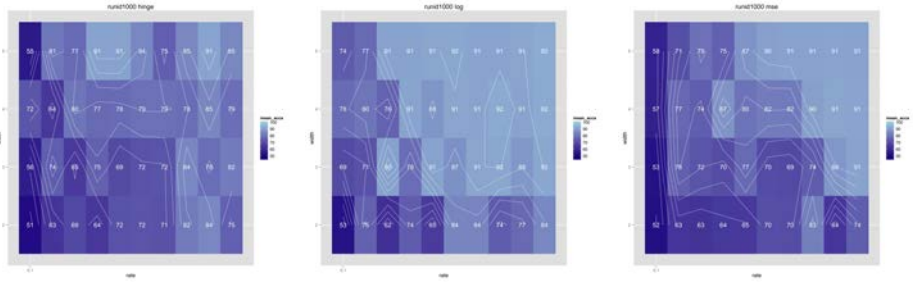
Interestingly, Relu provides comparatively useful results to Tanh and Elu, unlike the C1task. However, the Elu hidden activation function provides better results.

Figure 18: Average accuracy for grid search parameter optimisation of optimisers for the MTP C2 task



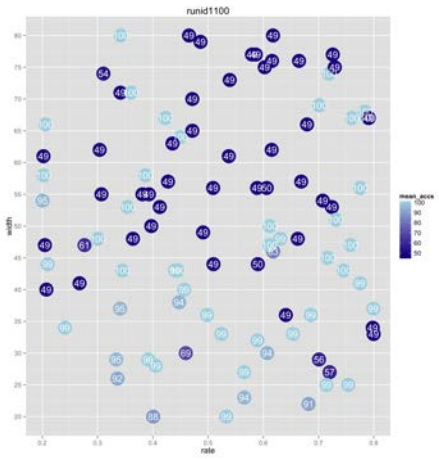
C2 - Optimiser Test Runs The optimiser C2 results are very similar to the C1 test results. As with the C1 test runs, figure 18 shows that ADAgrad and PAD both perform excellently. ADAgrad performs slightly better than PAD at a low number of neurons in the hidden layer. ADAdelta again provides some good results. Gradient Descent (figure 16) also performs better than ADAdelta.

Figure 19: Average accuracy for grid search parameter optimisation of loss functions for the MTP C2 task



C2 - Loss Test Runs Comparing the results of figure 16 to figure 19 shows that Cross Entropy is the optimal parameter choice for this problem. There are clearly more excellent and reliable results across the parameter space.

Figure 20: Average accuracy for random search parameter optimisation for the MTP P1 task

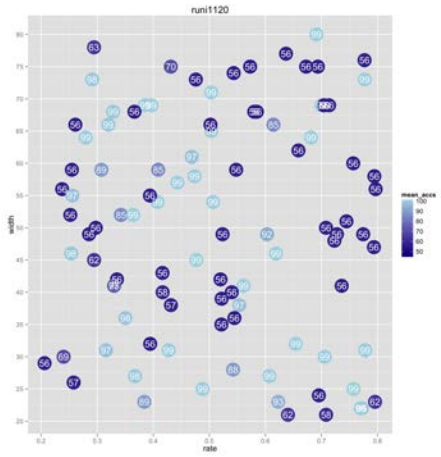


P1 - Initial Test Run P1 test results (figure 20) show networks can produce excellent results. But results become poorer as the number of neurons in the hidden layer increases. There seems to be an upper limit to the number of neurons which can produce reliable results. From 40 neurons and above, there seems to be around a 50 % chance of a useful result or not.

To perform the P1 task, the neural network essentially acts as the encryption system, turning plain texts into cipher texts. In doing so, it will replicate the operations

between the plain text and the key. The network is only given two input values per key bit, so working out whether a neuron needs to fire is a binary choice. If the key bit value is a 1, a plain text bit value of 1 needs to produce a 0. Conversely, a plain text value of 0 needs to produce a 1. This is simple subtraction operation, and the network only really needs to learn the bias (with weights set to 1).

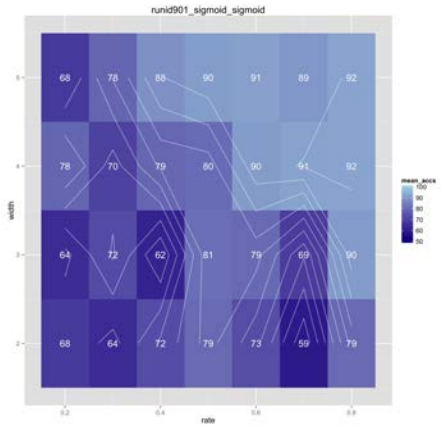
Figure 21: Average accuracy for random search parameter optimisation for the MTP P2 task



P2 - Initial Test Run Figure 20 shows us that networks can perform excellently at the P2 task. But the distribution of accuracies across the parameter space shows no linear relationships. Instead, good results are mixed in with poor results. Discovering why this happens will require further investigation.

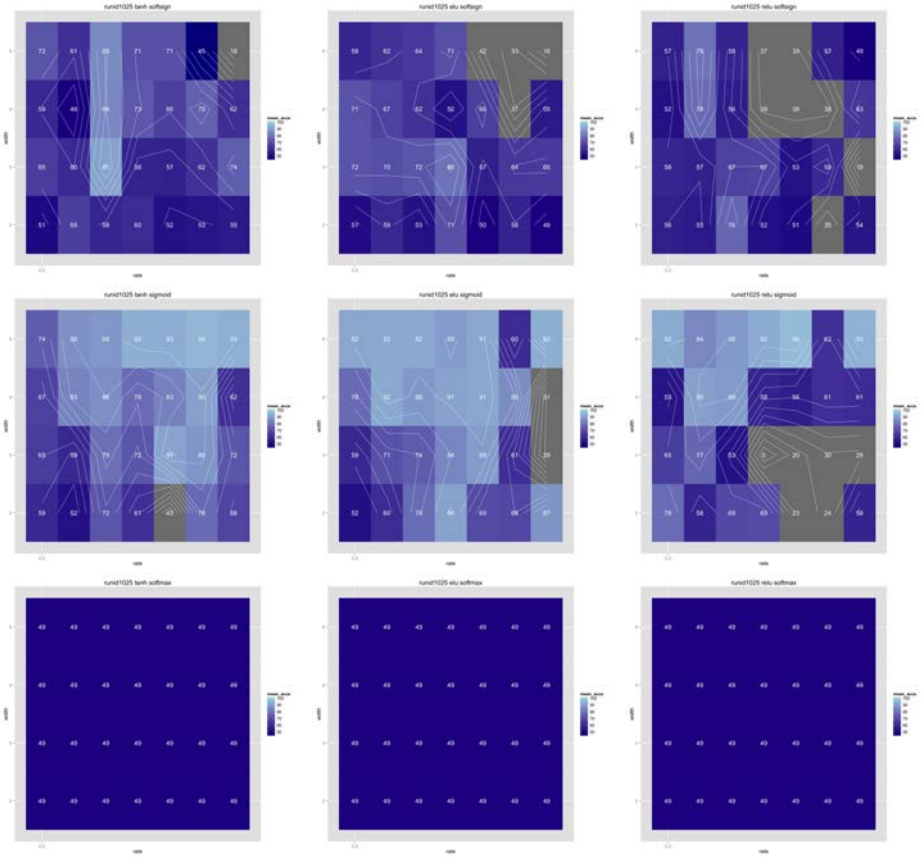
6.2 Caesar Cipher Runs

Figure 22: Average accuracy for basic grid search parameter optimisation for the Caesar C1 task



C1 - Initial Test Run For the C1 Caesar task, the network needs to be able recognise integer shifts in binary terms. For example, 0001 to 0010 for a shift of 1. The initial test run for Caesar C1, figure 22, looks very similar to the MTP20 C1 results set. Both prefer a combination of higher learning rates and wider hidden layers, returning excellent results. Across the parameter space there are generally good results. So the network is easily able to classify integer shifts even when represented in binary strings.

Figure 23: Average accuracy for grid search parameter optimisation of activation functions for the Caesar C1 task



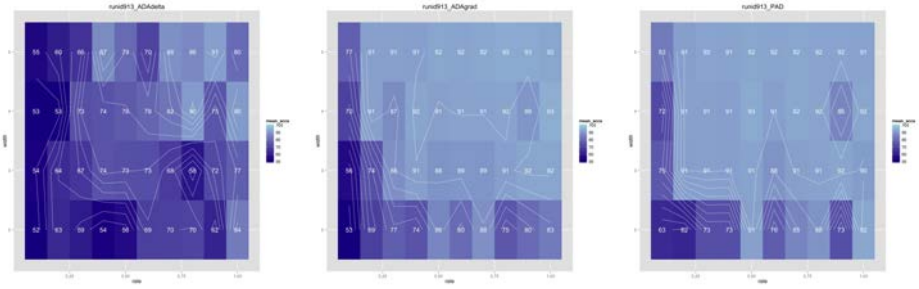
C1 - Activation Function Test Runs Figure 23 readily identifies that the sigmoid output activation functions provides the best performance. Softsign provides some good results but there is no clear relationship between performance and parameters, which means these results may be less reliable. Softmax, as expected, provides poor results.

In terms of hidden activation functions, Elu provides excellent results with a clear relationship between parameters and performance. Tanh also performs well, with Relu providing some good results, but nothing that can be deemed useful.

Interestingly, the Caesar C1 task seems to have a risk of overfitting, with 46 runs being cancelled. This is shown by the grey squares in figure 23. The cause of this

is unknown. It could be because the Caesar problem is too simple, but this requires further investigation.

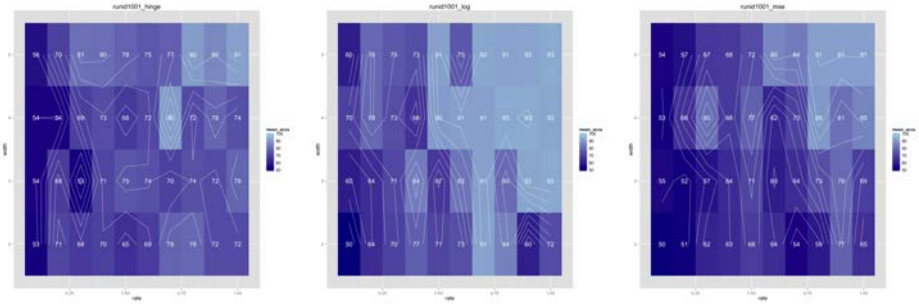
Figure 24: Average accuracy for basic grid search parameter optimisation for optimisers for the Caesar C1 task



C1 - Optimiser Test Runs Figure 24 shows the different results from the 3 chosen optimiser models. Both PAD and ADAgrad provide excellent results across the parameter space. PAD outperforms ADAgrad slightly as it provides a larger area of excellent results. ADAdelta surprisingly performs well, but is totally outclassed by ADAgrad and PAD.

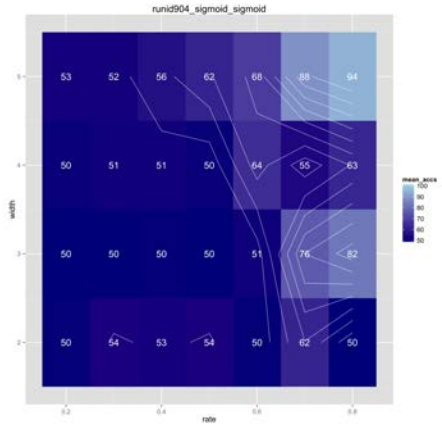
As mentioned previously, the PAD and ADAgrad optimisers are tuned towards handling sparse data so it is interesting that they perform well here (Dali, 2017).

Figure 25: Average accuracy for basic grid search parameter optimisation for loss functions for the Caesar C1 task



C1 - Loss Test Runs From figure 25, it can be seen that Log loss is far better than MSE and Hinge, providing a clear area of excellent results when combined with wider hidden layers and higher learning rates. However, when compared to figure 22 which uses Cross Entropy, both MSE and Hinge perform better. Log does provides some outliers with excellent results in spaces that one wouldn't expect. This is a concern, as randomness like this affects reliability of a network's performance.

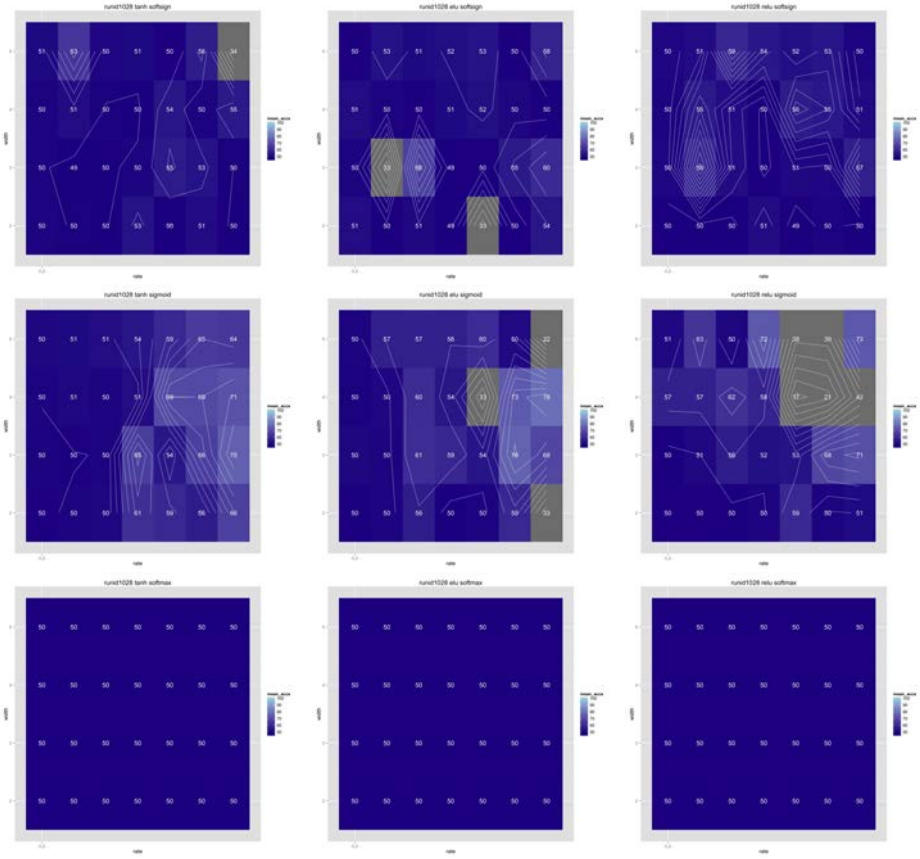
Figure 26: Average accuracy for basic grid search parameter optimisation for the Caesar C2 task



C2 - Initial Test Run Perhaps unsurprisingly, with the chosen parameter space, the Caesar C2 task is much harder to perform well on as can be seen in figure 26. Nevertheless, a combination of higher learning rates and wider hidden layers seems to provide possibly better results but the fact that this is not linearly correlated raises some concerns. The issue with this task could be that the key sizes for the data set have been set two small. As in, a key shift of 5 versus 20 is not a large enough value.

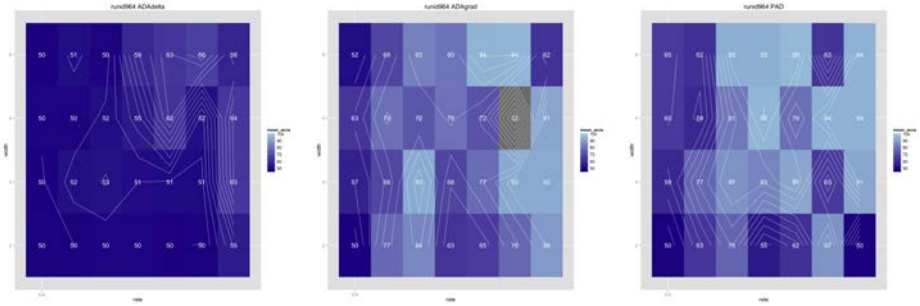
Also, the C1 problem requires that the network only classify between random and non-random data. C2, however, requires that it classifies data that is inherently similar. Having said this, around 50 percent of the parameter space provides good results. So this could be considered a good result.

Figure 27: Average accuracy for grid search parameter optimisation of activation functions for the Caesar C2 task



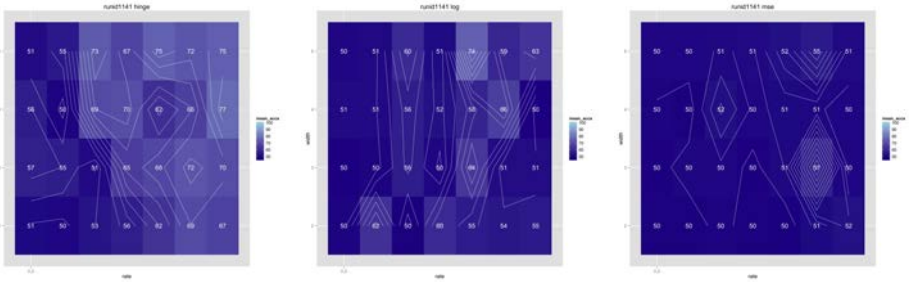
C2 - Activation Function Test Runs In figure 27 it can be sen thatthe choice of activation function does not seem to have much positive impact on test results. All the parameters tested performed worse than the base run. The Sigmoid output activation function provides some generally acceptable results in the region of 70%. But these are nowhere close to the 90% base run result in figure 27.

Figure 28: Average accuracy for grid search parameter optimisation of optmisers for the Caesar C2 task



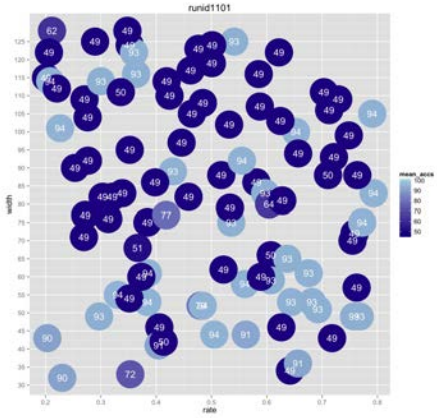
C2 - Optimiser Test Runs As seen with the MTP C2 problem in figures 16 and 18 for MTP C2, figure 26 shows that ADAdelta performs worse than Gradient Descent for this task. Again, ADAgrad and PAD both provide some excellent results over the parameter space. However, ADAgrad seems to perform slightly worse than PAD in this task. Not least because of a sub 50 % result, which is due to an overfitting run. PAD shows somewhat of a linear relationship between layer width and learning rate, with higher value for both producing better results. However, the parameter space shows that this is not completely reliable.

Figure 29: Average accuracy for basic grid search parameter optimisation for loss functions for the Caesar C2 task



C2 - Loss Test Runs From figure 29, it can be seen that Hinge loss provides somewhat better results than the base run, as per figure 26 which provide some higher results across a larger area of the parameter space. This can be considered a good result, as it seems to generalise the Caesar C2 problem more than the base run. However, the fact that Hinge loss doesn't reach any 90% accuracy rates is a concern. Both MSE and Log loss functions perform worse than the base run.

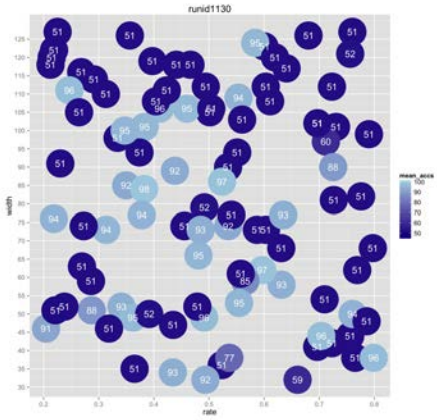
Figure 30: Average accuracy for basic grid search parameter optimisation for the Caesar P1 task



P1 - Initial Test Run Figure 30 shows that it is possible for the P1 problem to be completed for the Caesar cipher. It seems that networks favour fewer hidden

layer neurons, suggesting extra layers may confuse the network. However, there are poor results across the parameter space and there is no linear relationship between accuracies. This requires further investigation.

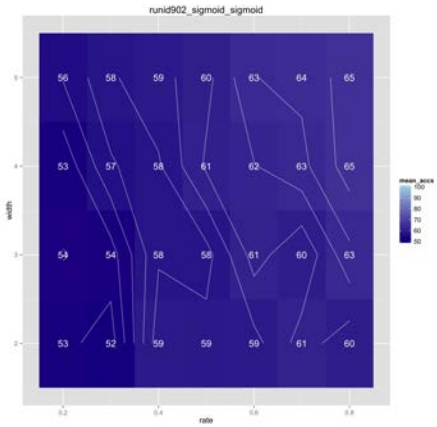
Figure 31: Average accuracy for random search parameter optimisation for the Caesar P2 task



P2 - Initial Test Run Figure 31 shows it's possible for the P1 problem to be completed for the Caesar cipher. However, good results seem to be surrounded by poor results, so it's not clear if there is a relationship between parameters and accuracy. But when a network does converge, it does very well. This requires further investigation.

6.3 Simple Substitution Cipher Tests

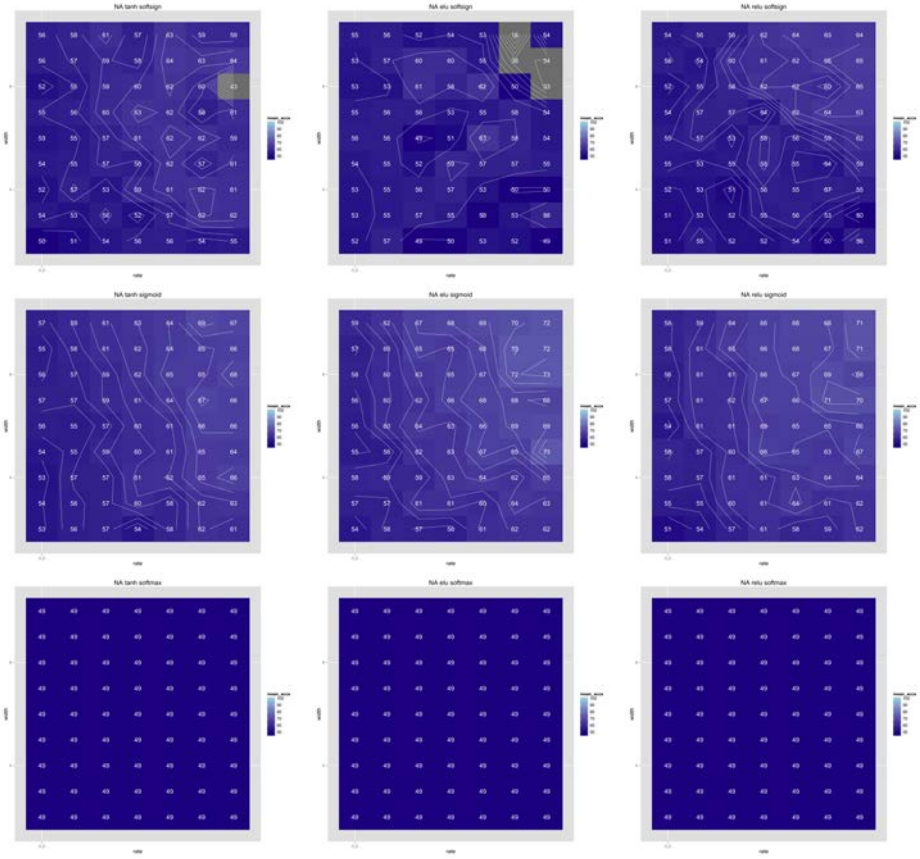
Figure 32: Average accuracy for basic grid search parameter optimisation for the Simple Substitution C1 task



C1 - Initial Test Run Figure 32 shows that layer width between 2 and 5 do not seem to be sufficient to provide excellent results. However, good results can be seen

with an increased learning rate and the accuracies follow a similar trend to that of MTP20 and Caesar in that an increase in hidden layer width seems to provide better results.

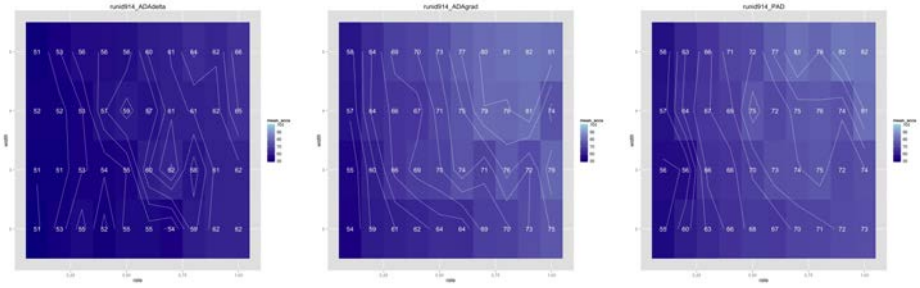
Figure 33: Average accuracy for grid search parameter optimisation of activation functions for the Simple Substitution C1 task



C1 - Activation Function Test Runs Figure 33 shows that the optimal activation function choices are an Elu hidden and Sigmoid output. This combination performs better at 5 layers than the base run (69% vs the base 65%) and out performs all other choices.

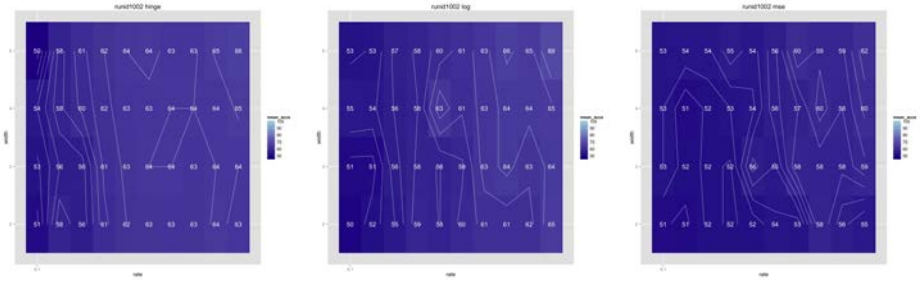
This test run was also run for a larger network sizes, and demonstrates that as network hidden layer width is increased, a correlating increase in performance occurs.

Figure 34: Average accuracy for basic grid search parameter optimisation for optimisers for the Simple Substitution C1 task



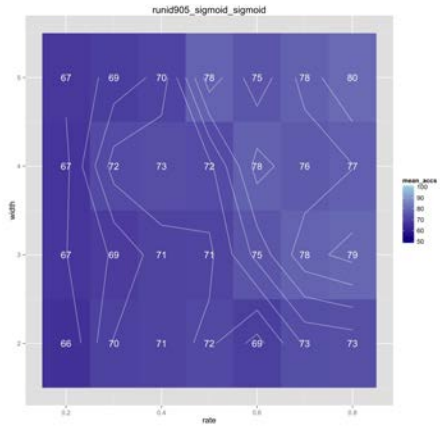
C1 - Optimiser Test Runs Figure 34 shows that ADAdelta performs comparably to SGD for this task, and once again is outperformed by ADAgrad and PAD. PAD again provides an excellent distribution of results, being more general across the parameter space than ADAgrad.

Figure 35: Average accuracy for basic grid search parameter optimisation for loss functions for the Simple Substitution C1 task



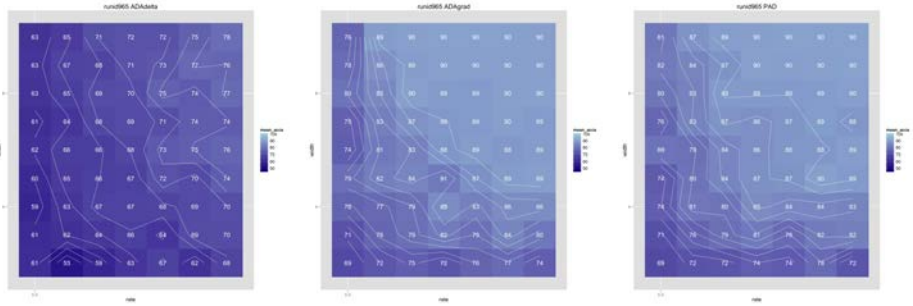
C1 - Loss Test Runs Figure 35 shows that Log and Hinge loss models both perform comparably to Cross Entropy, providing no results over 70 percent, and generally sitting around 55 to 65 percent. MSE Loss performs slightly worse, not reaching many 65 percent results, but it still performs reasonably well.

Figure 36: Average accuracy for basic grid search parameter optimisation for the Simple Substitution C2 task



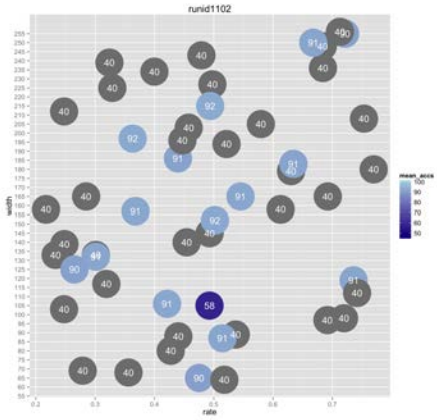
C2 - Initial Test Run The substitution C2 task seems to have been completed well when considering the C1 substitution result, as per figure 36. Having randomly selected "keys" for the substitution data sets will probably have helped here. It is interesting that the C2 task performs about 10 percent better than the C1 task. Similar to the MTP20 C2 task, this could be because the network only needs to check a few bit values to actually get a correct result.

Figure 37: Average accuracy for grid search parameter optimisation of optimisers for the Simple Substitution C2 task



C2 - Optimiser Test Runs Similarly to the C2 activation test runs, the Optimiser C2 tests were also performed for a wider set of layers than the base runs. In doing so, figure 37 shows a clear relationship between the width of layers and and increased learning rate, especially for the ADAgrad and PAD optimisers. Even at the 5 width layer level (the maximum number of neurons used for the original C2 base run), there are reliable results with a 10% increase on the results for Gradient Descent as per figure 36.

Figure 38: Average accuracy for random search parameter optimisation for the Simple Substitution P1 task



P1 - Initial Test Runs Figure 38 shows that it is possible for networks to complete the P1 task for the Substitution cipher. But, like the MTP and Caesar ciphers, there is no clear relationship between parameters and network performance. However, the presence of so many 40% results is strange as it is expected that a poor run should return 50% results - i.e. a random guess. Runs with an average accuracy of 40% suggest that the network is learning in the opposite direction to what is required - i.e. it is learning to provide the opposites of cipher texts. There were no runs recorded as having overfitted to the training data. This requires further investigation.

7 Evaluation - DES

7.1 DES - Single Component, Single Round

Figure 39: Average accuracy for random search parameter optimisation for DES IP64 table for the C1 task

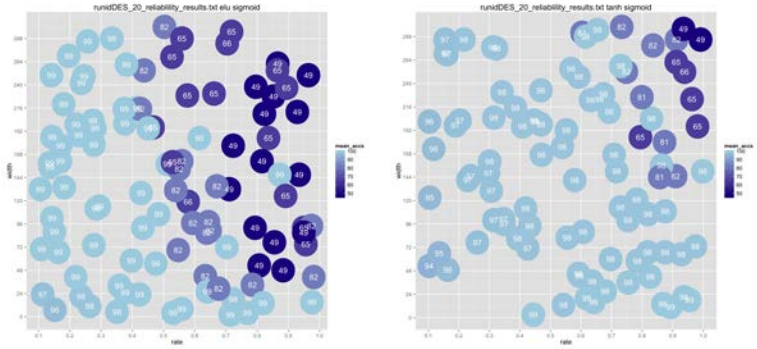


Figure 40: Average accuracy for random search parameter optimisation for DES expansion table for the C1 task

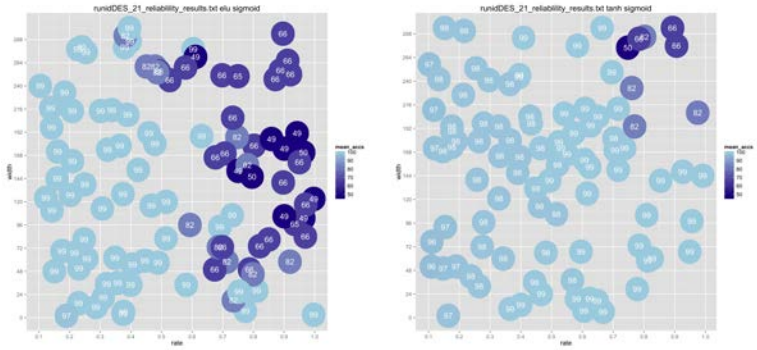


Figure 41: Average accuracy for random search parameter optimisation for DES IP-1 permutation table for the C1 task

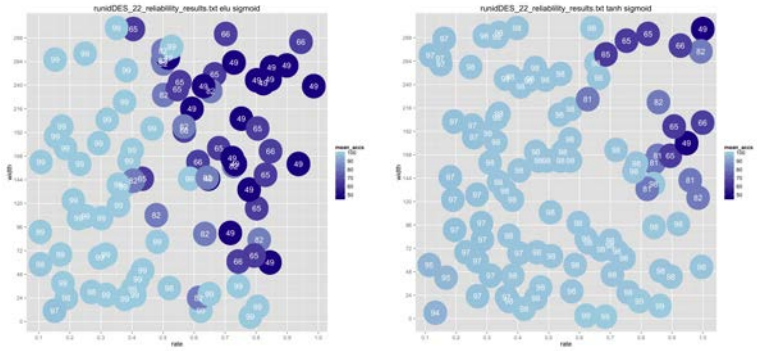


Figure 42: Average accuracy for random search parameter optimisation for DES substitution box for the C1 task

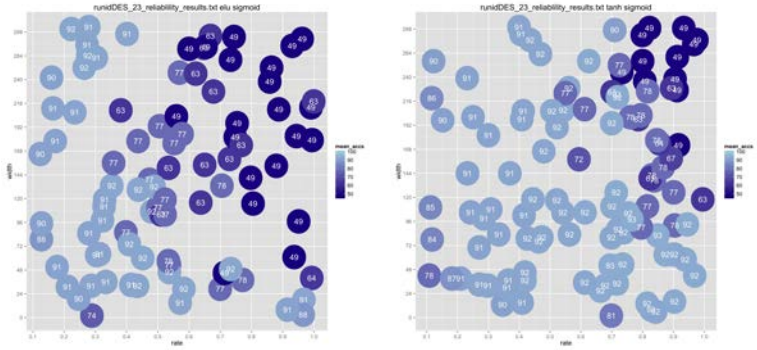
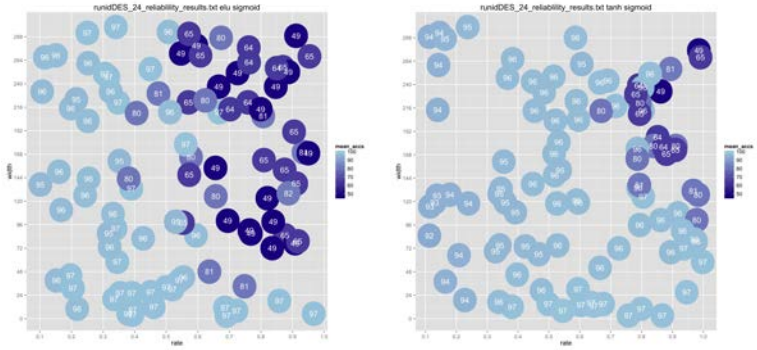


Figure 43: Average accuracy for random search parameter optimisation for DES 32 bit permutation table for the C1 task



C1 Simple neural networks can perform the C1 task against individual components of DES extremely well. Figures 39, 40, 41, 42, 43 clearly show Tanh hidden activation functions perform better than the Elu hidden activation functions. Interestingly, it can be seen that DES components can be classified with relatively small hidden layer widths. The networks also seem to prefer small learning rates across all these problems.

This could be due to the increasing complexity when using larger widths. The network may be finding it harder to find useful weight and bias values with higher widths. A high learning rate may also mean that the network gets "lost" in a local minima and cannot escape. This would explain the top right pockets of all these graphs.

It is not surprising that simple neural networks can handle this task so well. Individual DES components are essentially just logical transformations - either lookup tables or permutations. The ciphers we analysed already showed that this is actually quite an easy problem to solve.

Figure 44: Average accuracy for random search parameter optimisation for DES IP64 table for the P1 task

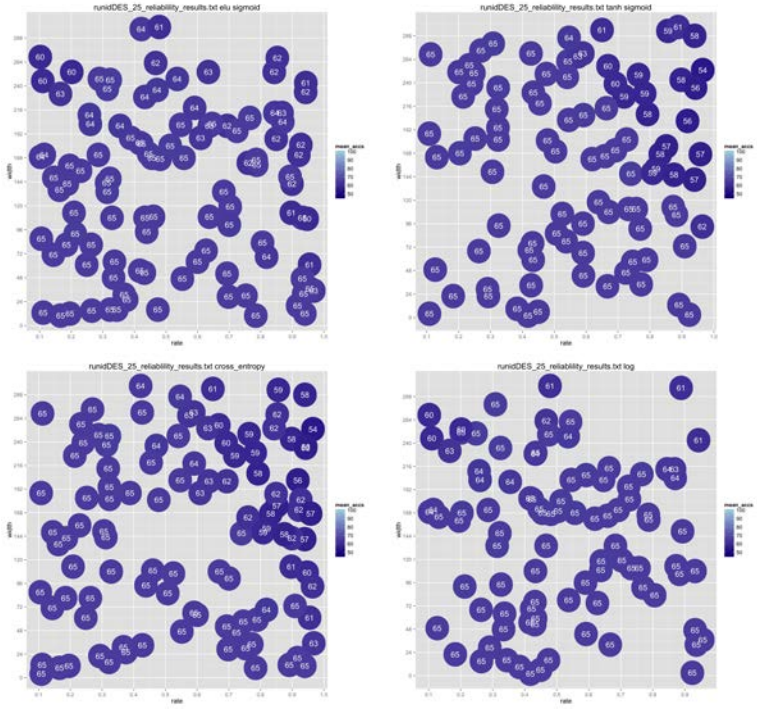


Figure 45: Average accuracy for random search parameter optimisation for DES expansion table for the P1 task

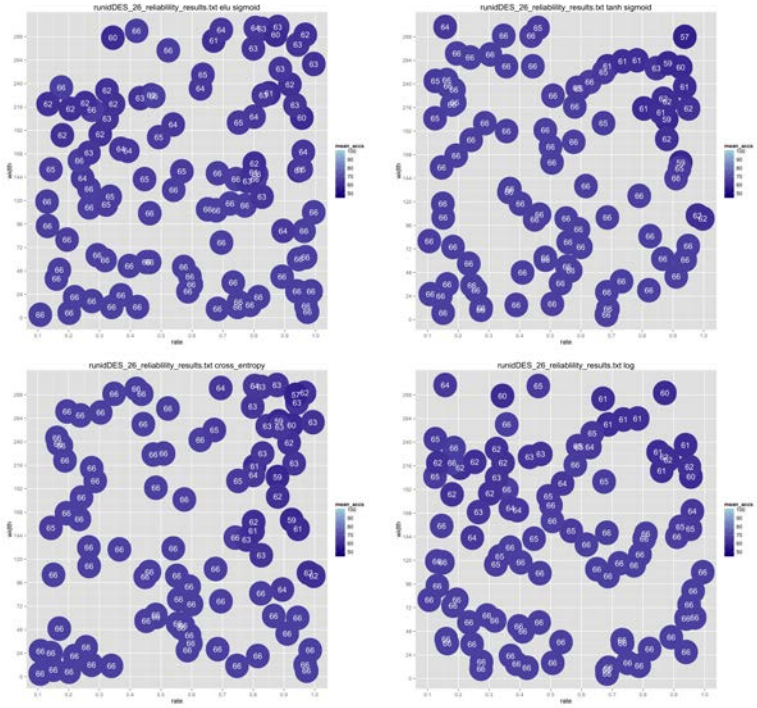


Figure 46: Average accuracy for random search parameter optimisation for DES IP-1 permutation table for the P1 task

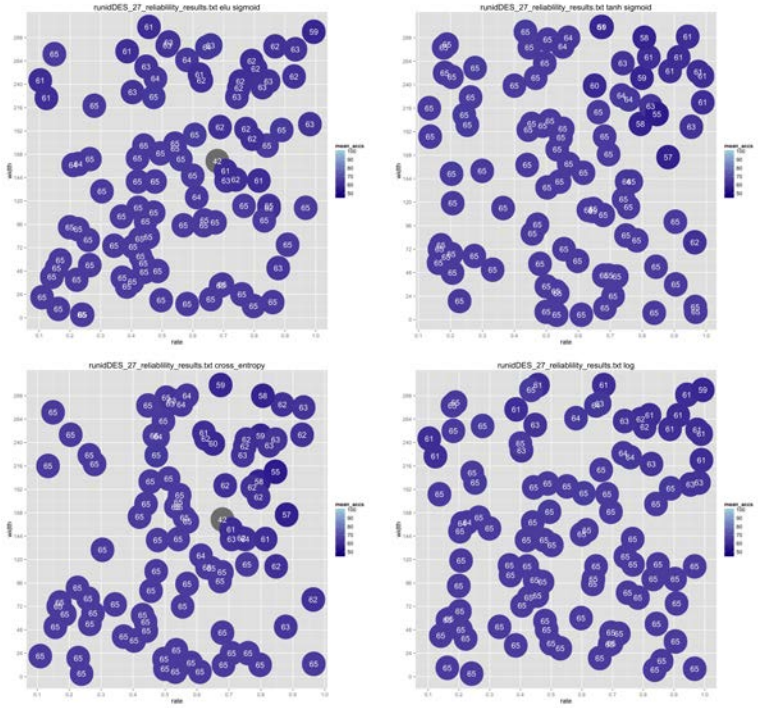


Figure 47: Average accuracy for random search parameter optimisation for DES substitution box for the P1 task

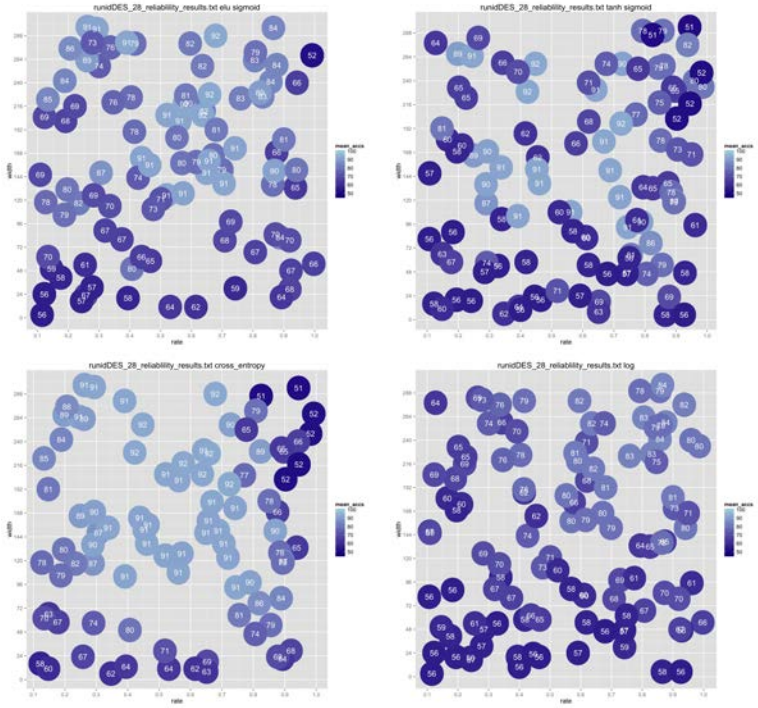
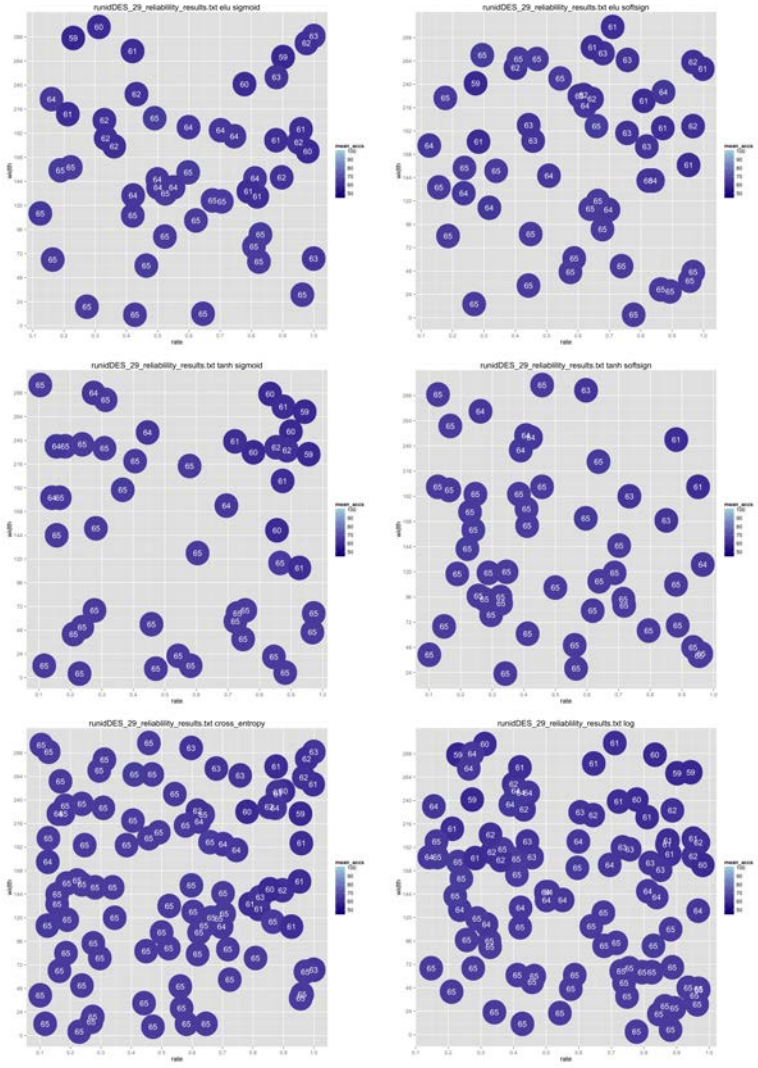


Figure 48: Average accuracy for random search parameter optimisation for DES 32 bit permutation table for the P1 task



P1 However, moving to the P1 task, the networks perform worse than with the C1 task. Interestingly, figures 44, 46, 48 and 45 show that the best results sit within a parameter space that closely resembles those for the C1 problem. For example, graph 2 (Tanh, Sigmoid) in figures 39 and 44 both show us that the networks do not perform well with high learning rates and high hidden layer widths.

Nevertheless, results of over 60 percent across the parameter space for all DES components are good results in themselves. It is possible that, when combined together, the network is able to propagate through these components providing good results even with added complexity. They also show that it is probably not the individual components that make DES secure, rather it is likely to be the way they are combined together.

Figure 47 shows excellent results across some parts of the parameter space. Graph 1 (Elu, Sigmoid) shows particular promise, as does graph 3 (Cross Entropy). This

is likely to be due to the fact that the total possible outputs of the neural network are much smaller than the varied possible inputs (as per the design of the SBox). Rather than interpreting the binary data, the network just needs to recognise the patterns between the output and input. As the Sbox is essentially a lookup table, this is apparently easy for the network to learn.

7.2 DES - Concurrent Components, Single Round

Figure 49: Average accuracy for random search parameter optimisation for the Initial DES Splits for the C1 task

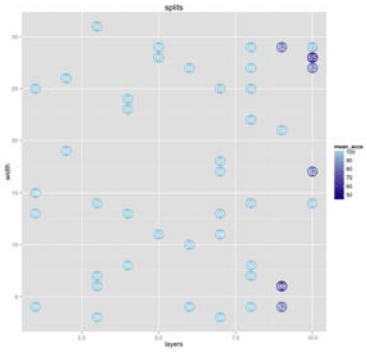


Figure 50: Average accuracy for random search parameter optimisation for the Expansion transform for the C1 task

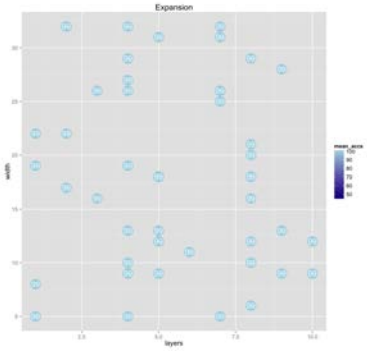


Figure 51: Average accuracy for random search parameter optimisation for the SBox transform for the C1 task

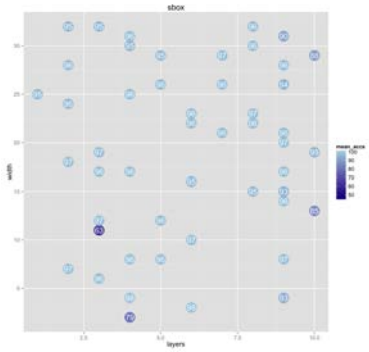


Figure 52: Average accuracy for random search parameter optimisation for the Perm32 transform for the C1 task

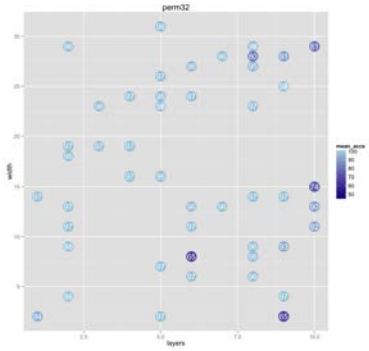
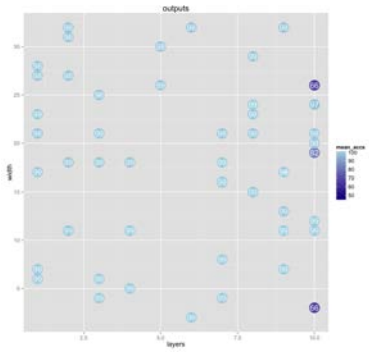


Figure 53: Average accuracy for random search parameter optimisation for the Feistel output for the C1 task



C1 Figures 49, 50, 51, 52 and 53 show the results from testing a single round of DES. Each figure shows the intermediate output from the component in question. It's clear that networks perform excellently over the entire Feistel function. There are some poor accuracy results, but not many, which there is confidence that networks can perform the C1 task for the Feistal function with ease.

7.3 DES - Two Round

Figure 54: Average accuracy for random search parameter optimisation for a single DES round for the C1 task

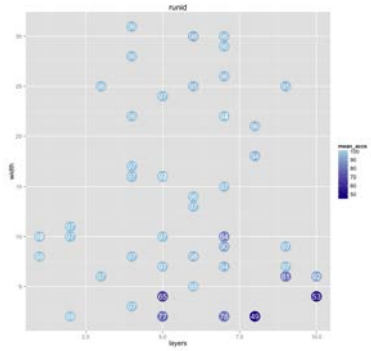
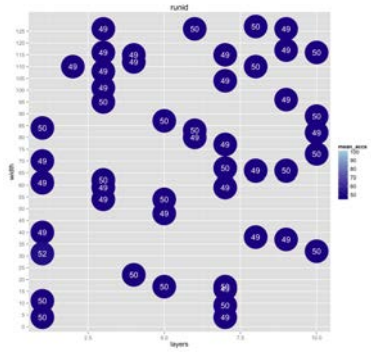
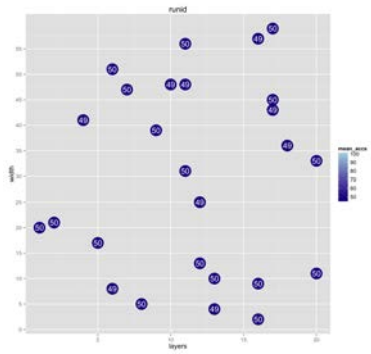


Figure 55: Average accuracy for random search parameter optimisation for a second DES round for the C1 task



C1 However, figure 55 shows that after a second round is included the performance of networks drops significantly with no results over 50%. So the network can't distinguish between cipher or random texts after only two DES rounds. As the inputs of round n_{-1} are used for round n, it might be that the network can not "remember" the outputs from the previous (n_{-1}) round.

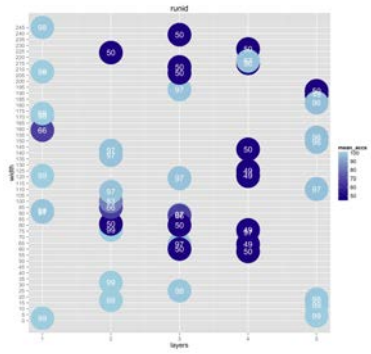
Figure 56: Average accuracy for random search parameter optimisation for a second DES round for the C2 task



C2 Figure 56 shows that the second round also returns poor results for the C2 problem, likely for the same reasons as the C1 task.

7.4 DES - Single Component, Multiple Rounds

Figure 57: Average accuracy for random search parameter optimisation for a simplified (Expansion box only) second DES round for the C1 task



In contrast with the problems with the C1 and C2 tasks, identified immediately above, when the Feistel function is removed and replaced with just an Expansion box figure 57 shows that the network performs the C1 task excellently. For this test, the network must have been able to remember the previous round's (n_{-1}) inputs for the current round (n). This confirms that it is the combination of the Feistel function and the multiple rounds implementation that gives DES it's strength against differential cryptanalysis.

8 Summary and Conclusions

8.1 Simple Ciphers

C1 tests Overall, the networks handled the simple ciphers well. All the C1 tasks returned greater than 50% results, meaning the networks could determine which text pair (plain & cipher or plain & random) were passed to it with some degree of accuracy. The MTP and Caesar ciphers produced over 90% results with relatively small networks. But the Substitution cipher saw poorer results, never exceeding 65% accuracy on basic texts. With the 36 character key size, it's probable that the substitution cipher requires somewhere between 30 and 40 neurons in the hidden layer to return over 90% results. However, the fact that a network with much fewer neurons than the characters in the key can return results over 50% at all is still useful. Generally, there seemed to be a relationship between the combination of the number of hidden layers neurons, learning rate and the resulting accuracy of tests.

From the parameter searching tests it can be seen that the Sigmoid function was the best output layer activation function for all tests. Softsign provided some useful results but nowhere near as good as Sigmoid. And Softmax, as expected, returned no useful results. Elu, Sigmoid and Tanh all performed well as hidden activation functions. Out of all three, Elu provided the best overall results across the ciphers. Relu interestingly did not have a clear relationship between accuracy and other parameters, but still provided some good results. The ADAgrad and PAD optimisers performed best, but even SGD provided some good results. This is interesting as ADAgrad and PAD specialise in optimising sparse data Dali, 2017. ADAdelta provided some useful results, but did not perform as well as the others. Finally, the Log and Cross Entropy loss models faired very well across all three ciphers. MSE and Hinge losses provided some good results, but only on a single test.

C2 tests The C2 tasks showed similar results to C1, but the Caesar cipher was especially tricky. This was likely due to the small variance in key values (5, 10, 15, 20). If the variances of key values for Caesar were increased, it is likely the results would have been higher. As before, the MTP cipher achieved excellent results. The Substitution cipher results again suffered due to the size of the key, but gave higher accuracies than the C1 results. Generally, there seemed to be a relationship between the combination of the number of hidden layers neurons, learning rate and the resulting accuracy of tests.

In terms of parameter searching, Sigmoid is again the optimal output activation function. Elu performed the best on the MTP data. But there were no useful results from the Caesar cipher results. Similarly to the C1 task, the ADAgrad and PAD optimisers performed best which is interesting as they specialise in optimising sparse data Dali, 2017. There was an instance of overfitting on the Caesar C2 test with PAD. Finally, the optimal loss choices across the ciphers varied, with Caesar preferring Hinge and MTP preferring Cross Entropy. Unfortunately, the Substitution tests were never completed for C2 activation functions or loss models because the parameter searching scripts took an inordinate amount of time to run.

P1 tests The P1 tests were limited to the basic runs because of time constraints. Unfortunately P1 tests for the Substitution cipher did not complete in time. However, there were still some excellent results. The MTP cipher provided excellent accuracies with thinner hidden layers, showing a clear relationship between the width of the hidden layer and network performance. But the networks suffered when too many neurons were present. The Caesar cipher performed well, returning some very high accuracies. However the reliability of the networks comes into question as there are 50% results mixed in with 90% results.

P2 tests The P2 tests were limited to the basic runs because of time constraints. Unfortunately P2 tests for the Substitution cipher did not complete in time. Whilst there were some excellent results, both the Caesar and MTP ciphers did not show any clear relationships between parameters and network performance. Positive results are intersected across the parameter space with poor results and this requires investigation. There are question to the reliability of these results, as it seems that only 50% of tests return a positive result. But running tests with more epochs or parameter searching may reveal more information about this.

8.2 DES

The results showed that when the individual DES components and Feistel function are tested on their own, they were breakable by neural networks for the C1 and, to a lesser extent, P1 tasks. The networks were able to do this with relatively thin layers and smaller learning rates. The networks were also able to break a basic implementation of the rounds structure for two rounds. However, when multiple rounds with the full Feistel function were tested, the networks struggled.

The combination of multiple rounds and the Feistel function would seem to give DES it's strength against differential cryptanalysis. It is conceivable this is because the network fails to remember the outputs from round n_{-1} for round n. However, the Expansion and multiple rounds test suggest that the network is able to remember a previous round's inputs. So the project has confirmed that DES is resilient to a simple differential cryptanalysis, as per it's original design. This in itself is a good result, as it confirms that neural networks can perform a type of cryptanalysis against a simplified version of DES. This results is also backed up by previous work in the subject (Alallayah and Alhamami, 2010). Considering the test data sets only provided 10^6 inputs, and linear cryptanalysis of DES has been shown to require 2^{40} plain and cipher text inputs, the data sets used in future must be much larger to seriously attempt the DES problem.

9 Critical Project Appraisal

9.1 Rationale for Design & Implementation Decisions

Data Set Choices and Generation The project used simple insecure ciphers so to guarantee some minimum useful results. Further to this, simple ciphers were chosen to give the project a solid analysis base to move forward from. The project used understanding about how a network behaved when approaching these simple problems .

Dynamic and "Lazy" Code Design Code was written to be as simple to operate as possible. Instead of specifying parameters or network definitions per run, code was always designed to do the hard work. For example, the network code determines the size of input and output layers from the size of input and output data. In doing so, time is saved and unnecessary complication when setting up runs is avoided.

Tensorflow Tenosrflow was originally designed for "large-scale distributed training and inference" (Tensorflow, 2017c). Tensorflow was chosen for this project out of a pool of several deep-learning neural network libraries (Nachum, 2016). Typically, variables in programming code are explicitly defined at one point or another. An explicitly defined list variable will contain the list items until the state of the variable is changed.

Instead of this, Tensorflow variables are actually nodes in a data flow graph (Google, 2015). Data sets flows through these nodes like buckets, rather than being explicitly declared. This makes Tensorflow excellent for distributed deep-learning, as it focusses on network definitions and the route which data takes (Google, 2015).

Furthermore, nodes can be declared via an iterable, so networks don't require parameters to be pre-defined. A single node can feed data through multiple routes in a network and nodes can process data from multiple inputs at once (Google, 2015).

Tensorflow is easy to implement, handles large volumes of data well and supports a range of other useful features, including:

- Tensorboard summary visualisations
- Model checkpointing
- Embeddings

However, having recently been released, there are some issues. For example, the AUC metric needs some obtuse code to implement. It requires the AUC Tensorflow node to be defined after the network has run. All other Tensorflow nodes are defined prior to the network running (Tensorflow, 2017b). There are also fewer parameter choices than other libraries like Theano, which has existed for longer. Finally, the Tensorflow documentation is sometimes incomplete, even though it is constantly being updated (Tensorflow, 2017b).

Overall, Tensorflow performed it's task well. It allowed for dynamic and "lazy" code practices, handled large volumes of data with ease and provided results that were possible to confirm with other metrics.

9.2 Project Evaluation

To decide whether this project has been successful, it is necessary to revisit the primary and secondary objectives as per the requirements section.

9.2.1 Objectives

It should be noted that the requirements specification was always intended to be flexible and open to change throughout the course of the project. There was little to no reliable published work on this subject so the project was always focussed on discovery, rather than hard and fast implementation deadlines.

Objective 1 - Show that simple crypto-systems that are known to be insecure could be broken by neural networks Test results showed that simple and insecure ciphers can return over 50% accuracy across many of the tests. So this objective was completed for all the simple insecure ciphers that were tested. In terms of the secondary objectives, parameter searching was performed so relationships could be extrapolated from the results data and networks were found to be reliable in tests. But some tasks were not parameter searched for all the simple ciphers, with the Substitution cipher particularly affected. Also, no known attack methods were compared to neural network performance to investigate efficiency. Finally, too much time was required to complete this objective, with severe delays due to test durations as well as issues with code.

Objective 2 - Break simple crypto-systems that are known to be insecure Test results showed that simple and insecure ciphers can be fully broken for many of the tests. So this objective was completed for all the simple insecure ciphers that were tested. In terms of the secondary objectives, parameter searching was performed so relationships could be extrapolated from the results data and networks were found to be reliable in tests. But some tasks were not parameter searched for all the simple ciphers, with the Substitution cipher particularly affected. However, no known attack methods were compared to neural network performance to investigate efficiency. Also, too much time was required to complete this objective, with severe delays due to test durations as well as issues with code.

Objective 3 - Show that semantically secure crypto-systems could be broken by neural networks. Test results showed that DES components and simplified implementations (e.g. a single DES round) can be broken reliably by neural networks. Results showed that neural networks struggled with the multiple rounds combined with the Feistel function, but a simplified Feistel function with multiple

rounds can be broken. So the project showed that neural networks can perform differential cryptanalysis on a simplified DES system. As DES can be broken with 2^{47} chosen plain texts with differential cryptanalysis, it can be said that that DES could possibly be broken by neural networks. So this objective was completed only for the DES crypto-system. In terms of the secondary objectives, parameter searching was not performed so relationships could not be extrapolated from the results data. Also, no known attack methods were compared to neural network performance to investigate efficiency. However, networks were found to be reliable in tests.

Objective 4 - Break semantically secure crypto-systems Some initial tests were run for the DES crypto-system to attempt this problem. However, none returned any results over 50%. As such, the project failed to meet this objective.

Objective 5 - Generate some approximations of keys or full crypto-systems No tests were run or code implemented for this objective. So the project failed to meet this objective.

9.2.2 Review of Project Plan & Work Schedule

Initially the original plan and work schedule were followed. However, as testing took longer than expected and there were initial teething problems, the first set of stages were pushed back by several weeks. Following on from that, the lack of any good results from the initial DES testing meant that the original plan had to be abandoned. Rather than focussing on the remaining crypto-systems, the analysis focussed primarily on DES to investigate why networks were struggling to return any good results.

The project plan and work schedule, as detailed in the requirements specification, were always intended to be flexible and open to change. As such, deviations from it were to be expected and are not a cause for significant alarm.

9.3 Lessons Learned

Even though the project was not completely successful, there are certain lessons that can be learned regrading it's approach. These lessons could be used alongside the recommendations for future work to produce a subsequent, in-depth analysis. Some involve implementation issues, whilst others cover the approach to the analysis and research issues.

9.3.1 Project Design and Implementation

Parameter Space Searching The project focussed heavily on exploring parameter spaces, a secondary objective. They provided some interesting results, but they were not worth the extra time that was needed to run. The secondary objectives for exploring the parameter space and understanding relationships should not have been considered at all. If good reliable test runs can be performed (i.e. an average of 90%

accuracy over 3 test runs) then the network has solved the problem and the analysis should have moved on. If any parameter objectives should have been considered, the project should have performed one for parameter optimisation. By doing this, the project would be searching for the simplest parameters that provide an excellent result rather than continually searching through the parameter space. Incidentally, this would have improved outputs giving only the best results for a combination of parameters, rather than for individual parameters.

Simple Ciphers vs. SDES Rather than using SDES (Simplified DES), which performs one DES round with a simplified Feistel function (Lamagna, 2013), the project used a range of simple ciphers. This meant the project spent far too long on the early stages of analysis, rather than moving forward through crypto-systems quickly. Furthermore, SDES has been classified by neural networks in the past (Alallayah and Alhamami, 2010). So using simple ciphers was not explicitly required, although they provided useful background knowledge. More in depth research was required, rather than taking some well published answers on the subject for granted (Holzner, 2011).

Class Based Network Code Originally the project used a class based neural network code base. This led to unwieldy and confusing code that was difficult to test and was not flexible when it came to test runs. This was good enough for simple cipher test runs (using a single layer). But when the analysis moved onto more complex problems with larger layers, this made parameter space searching much harder. The intention was always to move to a more dynamic design, which was implemented later on.

Output Activation Code Issue There was a major issue discovered one month prior to the deadline for the project. A variable was not properly passed to the network at run-time, which meant that all network test runs used the Sigmoid activation function in the output layer. All tests that had been completed up to that point had to be rerun. This was a major delay to the project and seriously impacted the results. As such, double checking the values of variables should have been considered during the testing phase. A separate testing function should have been implemented. This function would take arguments provided by the argparse module and check the actual variable values passed to network later on.

Input File, Results File and Test Run Management Instead of using a manual process for run management, a complete management system should have been implemented. There are some systems available for Tensorflow (chovanecm, 2017), but not for the specific environments this project used. As such, one should have been designed and built to solve the following issues. This could have been done relatively simply using Jenkins continuous integration, Docker images and Python (or bash) scripts.

Input Files Because various crypto-systems were selected for this project, there were many large data files. Managing these data files became increasingly difficult over time. Tensorflow placeholders, the variables that are fed input data, can handle data sets of varying size. As such, it would have been possible to generate data and run the neural networks concurrently. As batches of data are generated, they could have been temporarily stored, and fed to the network once it had finished it's previous batch. Data files also wouldn't have to be transferred onto GPU servers.

Results Files As per the input data files, the results files eventually became confusing to manage. These were worse than the data files as there were 16 results, each with 4 output files, per input data file. As such, these became incredibly complex to manage over time. Throughout the project, a Microsoft Excel spreadsheet was maintained to keep track of run progress, the progress of results files (had graphs been generated for them, for example) and the location of the files. But this manual process became increasingly complex and time intensive.

Test Runs Test runs were executed using bash scripts on the GPU servers. Different runs were executed on the server using the $screen - S$ command, allowing for different names to be given to different runs. However, this was not an automatic process, bash scripts tended to contain multiple tests to run and there was no output of the number of tests that had been performed. This meant that it was difficult to view the progress of and manage the time being taken by runs.

9.3.2 Personal Approach

Work Schedule Management Unfortunately, because the project was open-ended and discovery based, it became difficult to manage workloads and work schedules. This became especially apparent when the project struggled with full DES encryption. Instead of revisiting the project plan and work schedule, the project moved forward in an less organised manner. Rather than attempting what seemed like a good idea at the time, the work schedule and plan should have been consistently updated and reviewed. Furthermore, weekly targets (e.g. breaking Caesar, performing parameter searches for MTP etc.) were made, but should have been monitored and adjusted as required.

"Breaking The Wall" For the last month or so of the project, the number of hours spent doing highly efficient work reduced dramatically. Prior to this, I had not taken a break from the project for over two months, and had been working long, unhealthy hours. Approaching problems with a clear, open mind and being able to move forward became increasingly hard. To combat this, time should have been allotted in the work schedule and the project plan for a period of rest. Furthermore, the regular periods of rest that were budgeted in (two per week) should have been adhered to more strictly.

10 Recommendations for Future Work

10.1 Testing

Classification Task 3 (C3) Classifying different crypto-systems or ciphers against one another for determining:

- Which crypto-system has been used for cipher texts
- if a crypto-system is less semantically secure than others

For example, classifying OTP vs. MTP should lead to a network learning to detect when plain text and cipher texts belong to MTP, as MTP is less secure. It is then expected that DES vs. OTP vs. MTP (0, 0, 1 classes respectively) would lead to accurate classifications. If so, it would be sensible to assume that AES-128 vs. AES-192 vs. AES-256 (1, 0, 0 classes respectively) would lead to a correct classification, if networks were able to successfully break AES-128.

DES Multiple Rounds In this project, it was discovered that the multiple rounds of DES, combined with the multiple operations in the Feistel function, means it is secure against an attack from neural networks. Future work should be done to determine how many elements of the Feistel function and rounds of DES are needed to reduce network performance.

10.2 Code

Random Search Parameter Optimisation The project focussed on exploring the parameter space of tests, rather than parameter optimisation. However, the code can be modified to search for the minimum parameters that provide the highest accuracy.

The random search code in *Run_models.py* can be used. If parameter settings have given us good results (e.g. over 90%), the next set of parameters could be set simpler ones only, e.g. smaller networks, learning rates closer to 0.5, sigmoid activation functions instead of Relu etc. If the network doesn't achieve over 90% for a defined number of iterations (the argparse argument *popt_iter* could be used for this), the network should stop.

Numpy *.npy* files The data preparation functions for the networks are quite overwhelming and have many complicated functions that can be difficult to modify. Tensorflow uses Numpy arrays for data inputs. So, data could be pre-prepared for each task and stored in Numpy *.npy* files. They could then be loaded directly into the networks. However, this requires many small files per experiment which could lead to unforeseen complications.

GPU Servers Run Monitoring As mentioned in the evaluation of the project, there were significant issues relating to managing runs, input files and output files. Currently, the available GPU servers do not have a run management system in place. However, this could be implemented as an internal web service. The GPU servers could be listed, with their current run-time stats (% memory used etc.). A variety of machine learning libraries could be included in this, rather than being limited to Tensorflow. Then a user only needs to upload the definition of the model and the input & class label data. Results can be presented in a standard format. Extra outputs (if required) can still be written to disk. Instead of having to monitor and control runs through shell commands, users would be able to schedule and prioritise their runs across all the GPU servers.

10.3 Network Types

Convolutional Neural Networks Using a Convolutional Neural Network (hereafter, CNN) to approximate cipher texts or keys was discussed early on in the project, but never implemented. This would require the network learning approximations of functions in crypto-systems and providing approximate outputs. The cipher texts or keys could then be used for further security work by an adversary. In the case of a key, an approximation that can reduce the time required for a brute force attack would be an excellent result. A good test case for this is the MTP cipher. It already gave good results using simple networks, so testing this could be quite simple.

Generative Adversarial Networks Previous work shows it is possible to use a Generative Adversarial Network (GAN) to create new encodings to use for security purposes (Abadi, 2016). GANs work by using two networks in an adversarial fashion - a generator and a discriminator. The discriminator is trained first, learning to recognise random inputs vs. some real data. The generator then uses a random seed as its input to create some output. This output is then classified against more real data by the discriminator. The generator network is then trained on the error between it's output and the real data. The objective is for the generator to be able to learn to 'trick' the discriminator into believing it's data is the real one, and to do it reliably.

For future work of this project, GANs could be used for P1 or P2 problems. GANs could be especially useful for Stream Cipher crypto-systems. Stream ciphers typically use a random seed derived from the key, which is then XORd with plain texts multiple times to create a cipher text.

LSTMs and RNNs Recurrent Neural Networks (hereafter, RRNs) add in a memory unit to networks. They use this to remember sequences. Information is stored in the memory unit for each step of training. New inputs are stored on top of previous inputs as a mathematical function. So they store every step of a sequence in some mathematical form within the memory unit.

Long-Short-Term-Memory (LSTM) networks add in gates to the memory unit.

These allows the network to learn whether to keep, update or forget the data in the memory unit.

The multiple rounds combined with multiple components in DES is where the project saw dramatic decreases in network performance because the network needed the previous inputs of a round and calculate the outputs of each component. Essentially, the network needs to remember parts of a sequence. LSTM memory units could therefore be used to perform this task, remembering previous inputs for future layers.

DES was used with ECB mode in this project but other block cipher modes for DES could also be looked at with an LSTM, as they use operations on different outputs of sequential DES data blocks to obfuscate information further. So block one will influence the output of block two, with block one influenced by an initialisation vector. The modes CFB (Cipher Feedback), OFB (Output Feedback) and Counter (CTR) all turn a block cipher crypto-system into a pseudo stream cipher crypto-system. It would be interesting to investigate whether LSTMs can also be used on these modes.

Neural Turing Machines Neural Turing Machines (NTMs) were researched in some detail but did not show too much promise at the start of the project. NTMs typically learn computational operations like sorting or moving blocks of data. As block cipher crypto-systems use computational operations throughout, NTMs warrant further investigation and testing.

10.4 Crypto Tests

AES Because it was not possible to break the DES block cipher in ECB mode, AES was never attempted. However, there is still work that can be done. Component testing can be performed on AES, to check if DES components are not the only ones that can be broken by networks. Test can be run as per this project, testing more and more combinations of components and numbers of rounds. This would confirm whether its the combination of rounds and transformations that obfuscate information in block ciphers. Also, the AES SBox can be modified by a user (*Applied Cryptography* 2016). So different versions can easily be tested as a part of the AES single component testing. Finally, some tasks could be performed on the Gallois field, teaching networks to recreate it's outputs (performing part of the Rijndael SBox generation process) (*Applied Cryptography* 2016).

Key size variance Another test than can be performed is the effect of various key sizes on the ability of a neural network to reach a solution. A larger key size should increase the complexity of the problem. So, it follows that it should be harder for a neural network to solve.

Some work was done in the course of this project on the MTP cipher, but revealed no useful results. But the key sizes used were very small compared to each other.

It should be noted that the Public Key project demonstrated a relationship between Public Key key size and time required for networks to converge to a solution

These tests can easily be performed using different data sets on the simple cipher data. The only thing required is new data sets. With some modification to the *DES.py* program, it could be possible to test DES with larger and smaller keys sizes (e.g. 8, 16, 32, 64 bits keys)

References

Abadi, Andersen (2016). *Learning to Protect Communications with Adversarial Neural Cryptography*. Paper. URL: https://arxiv.org/abs/1610.06918.

Alallayah El-Wahed, Amin and Alhamami (2010). *Attack of Against Simplified Data Encryption Standard Cipher System Using Neural Networks*. Journal Article.

Alex Graves, Greg Wayne and Ivo Danihelka (2014). *Neural Turing Machines*. Paper. URL: https://arxiv.org/pdf/1410.5401v2.pdf.

Applied Cryptography (2016). Book. URL: https://crypto.stanford.edu/~dabo/cryptobook/draft_0_3.pdf.

Arora (2012). [Online]. URL: http://www.eetimes.com/document.asp?doc_id=1279619.

chovanecm (2017). *Sacredboard*. [Online]. URL: https://github.com/chovanecm/sacredboard/.

Colah (2015). [Online]. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Crypto 101 (2017). Book. URL: https://www.crypto101.io/Crypto101.pdf.

Cryptography, Practical (2012). *Practical Cryptography - Caesar Cipher*. [Online]. URL: http://practicalcryptography.com/ciphers/caesar-cipher/.

Curtin, C Matthew (2007). [Online]. URL: http://www.interhack.net/projects/deschall/.

Dali, Salvador (2017). [Online]. URL: https://stackoverflow.com/questions/36162180/gradient-descent-vs-adagrad-vs-momentum-in-tensorflow.

Deeb, Ahmed El (2015). [Online]. URL: https://medium.com/rants-on-machine-learning/smarter-parameter-sweeps-or-why-grid-search-is-plain-stupid-c17d97a0e881.

Dourlens, Sbastien (1996). *Applied Neuro-cryptography*. Thesis. URL: http://s.dourlens.free.fr/AppliedNeuroCryptography.pdf.

EEF (1998). [Online]. URL: https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html.

Ewing, Larry (1996). Image. URL: https://commons.wikimedia.org/w/index.php?curid=828161.

FIPS (1977a). Paper. URL: http://csrc.nist.gov/publications/fips/archive/fips46-3/fips46-3.pdf.

— (1977b). [Online]. URL: http://csrc.nist.gov/publications/PubsFIPSArch.html.

— (1980). [Online]. URL: http://csrc.nist.gov/publications/fips/fips81/fips81.htm.

Google (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. Paper. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

Grabbe (2006). [Online]. URL: http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm.

Holzner, Andre (2011). *Can neural networks decrypt files?* [Online]. URL: https://stackoverflow.com/questions/5989096/neural-network-for-file-decryption-possible.

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016). "Deep Learning". Book in preparation for MIT Press. URL: http://www.deeplearningbook.org.

Lamagna, Professor Edmund (2013). *DES Simplified.* [Online]. URL: https://www.cs.uri.edu/cryptography/dessimplified.htm.

Langford, Susan K. and Martin E. Hellman (1994). *Differential-Linear Crypt analysis.* Paper.

Lunkwill (2006). Image. URL: https://commons.wikimedia.org/w/index.php?curid=797926.

MacKay, D.J.C. (2003). *Information Theory, Inference and Learning Algorithms.* Cambridge University Press. ISBN: 9780521642989. URL: https://books.google.co.uk/books?id=AKuMj4PN_EMC.

McKenna, Stephen (2016). *Neural Networks.* Lecture Materials.

— (2017). *A single neuron.* Image.

Mixedmath (2011). Image. URL: https://crypto.stackexchange.com/questions/59/taking-advantage-of-one-time-pad-key-reuse.

Murphy, K.P. (2012). *Machine Learning: A Probabilistic Perspective.* Adaptive computation and machine learning. MIT Press. ISBN: 9780262018029. URL: https://books.google.co.uk/books?id=NZP6AQAAQBAJ.

Nachum, Ofir (2016). [Online]. URL: https://www.quora.com/What-is-the-best-open-source-neural-network-library.

OpenSSL (2016a). *OpenSSL Crypto Libraries.* [Online]. URL: https://www.openssl.org/docs/manmaster/man3/.

— (2016b). *OpenSSL FAQs.* [Online]. URL: https://www.openssl.org/docs/faq.html.

PyCryptoDome (2017). *PyCryptoDome reference documentation.* [Online]. URL: http://pycryptodome.readthedocs.io/en/latest/.

Python (2017a). *Python General FAQs.* [Online]. URL: https://docs.python.org/3/faq/general.html.

— (2017b). *Python Pypi Packages.* [Online]. URL: https://pypi.python.org/pypi.

— (2017c). *random - Generate pseudo-random numbers.* [Online]. URL: https://docs.python.org/2/library/random.html.

Rajput, Chakravarti, and Kothari (2015). *A Comprehensive Study On The Applications of Machine Learning For Diagnosis Of Cancer.* Image.

Rennie and Srebro (2015). Paper. URL: http://ttic.uchicago.edu/~nati/Publications/RennieSrebroIJCAI05.pdf.

Rijmenants, Dirk (2017). [Online]. URL: http://users.telenet.be/d.rijmenants/en/onetimepad.htm.

Roeder, Tom (2010). [Online]. URL: http://www.cs.cornell.edu/courses/cs5430/2010sp/TL03.symmetric.html.

Tensorflow (2017a). URL: https://www.tensorflow.org/get_started/summaries_and_tensorboard.

Tensorflow (2017b). *API reference*. [Online]. URL: https://www.tensorflow.org/api_docs/.
— (2017c). *Architecture Overview*. [Online]. URL: https://www.tensorflow.org/serving/architecture_overview.
WhiteTimeWolf (2013). Image. URL: https://commons.wikimedia.org/w/index.php?curid=26434116.
Zhang, Jianguo (2016). *Neural Networks*. Lecture Materials.

11 Appendix 1 - Background Information

Permutation Position Map	3	2	4	1	5
Original Bits	1	0	1	0	1
Permutated Bits	1	0	0	1	1

Table 10: Example of a Permutation Transformation

Original Bits	011100
Left Division	011
Right Division	100

Table 11: Example of a Block Division operation

Original Bits (X)	10001
Shift X right by 1 (Y)	11000
Shift Y Left by 2	00011

Table 12: Example of a Shift transformation

x	0	1
00	1010	1111
01	1011	1000
10	0001	1100
11	0000	0010

Table 13: Example of a single SBox

Original Bits	101
LR lookups	11
Middle lookup	0
Output Bits	0000

Table 14: Example of an SBox lookup transformation using table 13

12 Appendix 2 - Network Test Run Parameters

12.1 Ciphers

12.1.1 C1 & C2 tasks

Initial Test Runs

- Layers - 1
- Width Range - 2 to 5
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

MTP & Caesar Activation Functions Parameter Searches

- Layers - 1
- Width Range - 2 to 5
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Relu, Elu, Tanh
- Output activation functions - Softmax, Softsign, Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

Substitution Activation Functions Parameter Searches

- Layers - 1
- Width Range - 2 to 10
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Relu, Elu, Tanh
- Output activation functions - Softmax, Softsign, Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

MTP & Caesar Optimiser Parameter Searches

- Layers - 1
- Width Range - 2 to 5
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - ADAgrad, ADAdelta, PAD
- Loss models - Cross Entropy

Substitution Optimiser Parameter Searches

- Layers - 1
- Width Range - 2 to 10
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Log, MSE, Hinge

MTP & Caesar Loss Parameter Searches

- Layers - 1
- Width Range - 2 to 5
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - ADAgrad, ADAdelta, PAD
- Loss models - Cross Entropy

Substitution Loss Parameter Searches

- Layers - 1
- Width Range - 2 to 10
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Log, MSE, Hinge

12.1.2 P1 & P2 tasks

MTP Initial Test Run

- Layers - 1
- Width Range - 20 to 80
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

Caesar Initial Test Run

- Layers - 1
- Width Range - 32 to 128
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

Substitution Initial Test Run

- Layers - 1
- Width Range - 64 to 256
- Learning rate range - 0.2 to 0.8
- Epochs - 200
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Sigmoid
- Output activation functions - Sigmoid
- Optimisers - SGD
- Loss models - Cross Entropy

12.2 DES

12.2.1 DES components

- Layers - 1
- Width Range - 2 to 300
- Learning rate range - 0.1 to 1
- Epochs - 100
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Elu, Tanh
- Output activation functions - Softmax, Sigmoid
- Optimisers - ADAgrad
- Loss models - Cross Entropy

12.2.2 DES Concurrent Components

- Layers - 1 to 10
- Width Range - 2 to 32
- Learning rate range - 0.5
- Epochs - 100
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Tanh
- Output activation functions - Sigmoid
- Optimisers - ADAgrad
- Loss models - Cross Entropy

12.2.3 DES Per Round Outputs

- Layers - 1 to 10
- Width Range - 2 to 256
- Learning rate range - 0 to 1
- Epochs - 100
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Tanh, Elu
- Output activation functions - Sigmoid, Softsign
- Optimisers - ADAgrad, PAD
- Loss models - Cross Entropy, Log

12.2.4 DES Basic Round Implementation (Expansion Box only)

- Layers - 1 to 5
- Width Range - 2 to 256
- Learning rate range - 0 to 1
- Epochs - 100
- Reliability Runs - 3
- Data set size 1000000
- Hidden activation functions - Tanh, Elu
- Output activation functions - Sigmoid, Softsign
- Optimisers - ADAgrad, PAD
- Loss models - Cross Entropy, Log