University College London

Dept. Computer Science

Automatic Feature Selection for Website Fingerprinting

Axel Goetz

BSc. Computer Science

Supervisors:

Dr. George Danezis

Jamie Hayes

March 21, 2017

Abstract

Contents

1 Introduction

1.1 The Problem

The internet has become an essential tool for communication for a majority of the population. But privacy has always remained a major concern, which is why nowadays most web content providers are slowly moving away from HTTP to HTTPS. For instance, at the time of writing, around 86% of Googles traffic is encrypted. This is a significant improvement compared to 2014 where only 50% was sent over HTTPS [12]. However, this encryption only obscures the content of the web page and does not hide what websites you are visiting or in general who you might be communicating with.

Hence, an Internet Service Provider (ISP) can easily obtain a lot of information about a person. This is an especially large concern for people living in oppressive regimes since it allows a government to easily spy on its people and censor whatever websites they would like. To circumvent these issues, several anonymization techniques have been developed. These systems obscure both the content and meta-data, allowing a user to anonymously browse the web. One of the most popular low-latency anonymization networks is called Tor, which relies on a concept called Onion Routing [38].

The list of known attacks against Tor is at the time of writing very limited and most of them rely on very unlikely scenarios such as having access to specific parts of the network *(both entry and exit nodes)* [38]. However, for this report we will make a more reasonable assumption that an attacker is a *local eavesdropper*. By this we mean that the entity only has access to the traffic between the sender and the first anonymization node, like ISPs.

One of the most successful attacks that satisfies these conditions is known as *website fingerprinting* (WFP). It relies on the fact that Tor does not significantly alter the shape of the network traffic [15]. Hence, the attack infers information about the content by analysing the raw traffic. For instance by analysing the packet sizes, the amount of packets and the direction of the traffic, we might be able to deduce which web pages certain users are visiting. Initially, Tor was considered to be secure against this threat but around 2011, some techniques such as the *support vector classifier* (SVC) used by Panchenko et al. started to get recognition rates higher than 50% [28].

However, one of the main problems with majority of the attacks proposed in the research literature, is that they rely on a laborious, time-consuming manual feature engineering process. The reason why is because most primitive machine learning techniques are only able to process fixed-length vectors as its input. Therefore features need to be picked based on intuition and trial and error processes that reveal the supposedly most informative features.

Thus the goal of this paper is to investigate the use of novel deep-learning techniques to automatically extract a fixed-length vector representation from a traffic trace that represents loading a web page. Next, we aim to use these features in existing attacks to see if our model successfully identified the appropriate features.

1.2 Aims and Goals

We can subdivide the project up into different aims, each with their own goals:

1. **Aim:** Critically review the effectiveness of current website fingerprinting attacks.
   Goals:
   - Analyse various models that are currently used in fingerprinting attacks.
   - Examine how would a small percentage of false positives impacts a potential attack.
   - Analyse how the rapid changing nature of some web pages would impact the attack.
   - Review if there are any assumptions that are being made that could impact the effectiveness of an attack.
2. **Aim:** Automatically generate features from a trace that represents loading a webpage.
   Goals:
   - Critically compare various different feature generation techniques such as stacked authoencoders, RNN sequence-to-sequence and bidirectional RNN encoder decoder models.
   - Identify a dataset that is large enough to train a deep-learning model.
   - Compare several software libraries to perform fast numerical computation such as Tensorflow, Torch, Theano and Keras.
   - Implement the most appropriate feature generation model in one of the previously mentioned software libraries.
3. **Aim:** Train existing models with the automatically generated features and test their performance compared to hand-picked ones.
   Goals:
   - Identify five different models that have previously been successful in various website fingerprinting attacks and implement those models.
   - Extract the same hand-picked features from our dataset as mentioned in the respective papers.
   - Investigate an appropriate technique for evaluating the results of different models.
   - Compare the hand-picked features compared to the automatically generated ones for different models. In addition, we also want to investigate their effectiveness in different threat models. For instance if an adversary wants to identify which specific web pages a user is visiting *(multi-class classification)* or if the adversary just wants to know whether the user visits a web page from a monitored set of web pages *(binary classification)*.

1.3 Project Overview

As previously mentioned, the general project can be split up into three different aims. Hence, we also approached it in three stages:

- First, we examine different existing website fingerprinting models to gain a deeper understanding of the concept.
- Next, we perform more research to contrast different automatic feature selection models and implement the most appropriate model.
- Finally, we compare the effectiveness of hand-picked features with the automatically generated ones by training them on different existing website fingerprinting models.

1.4 Report Structure

The general report has a very simple infrastructure. In the following section we further explore several concepts that are necessary to understand the basics of website fingerprinting and the specific attack that we will be performing in our experiment. Next, we identify the threat model and design an attack for that threat model that automatically generates features. Finally, we explore several methods of evaluating the performance of different models with different features and contrast the hand-picked features with the automatically generated ones. ../report.bib

2 Background Information, Related Work and Research

In the following chapter, we further explore the motivation for undertaking the project, analyse the current state of the project domain and outline the research that forms the basis for the rest of the report.

2.1 The Problem

As previously mentioned, the goal of this project is to automate the feature selection process for a website fingerprinting attack. By this we mean that given a specific trace, our model should be able to produce a fixed-length vector that is a close representation of the respective trace. However, before we delve into the details of the attack, we first need to gain a greater understanding of some concepts such as onion routing, website fingerprinting in general and deep learning.

2.1.1 Onion Routing

To preserve privacy, we do not only need to obscure the content of a webpage but also hide who is taking to whom [11]. Tor achieves both of these by making use of a technique, called *onion routing*, which is a very simple protocol that can be divided up into three phases: connection setup, data movement and connection tear-down [11]. We show how it works by taking a simple example of Alice trying to communicate with Bob.

1. **Connection Setup:**
    - Alice's Tor client obtains a list of Tor nodes from a directory server.
    - Then Alice picks three random Tor nodes and labels them *one, two* and *three*.
    - Alice communicates with the first node and shares a symmetric key.

- Next, Alice sends messages to the first node, which are then forwarded to the second node to share another symmetric key to the second node.
- Finally, Alice continues sending messages to the first node, which are forwarded to the second and finally to the third node to share the final symmetric key. What is important here is that we use a secure key-sharing algorithm such that only Alice and the respective node know the keys. Additionally, since all of the traffic is forwarded from the first node, the second and the third nodes do not know the identity of Alice.

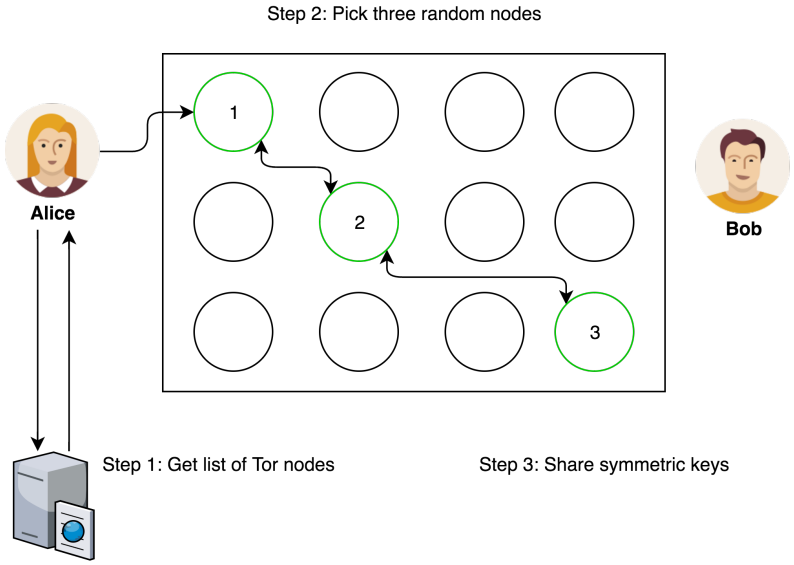


Figure 2.1: An example of a connection setup for the onion routing protocol.

2. **Data Movement:**

- Before Alice can send any data, it first needs to encrypt it in different layers. By this we mean that first it encrypts the data *(with Bob's address)* using the shared key from the third node. Next, it encrypts that data again *(with the address of the third node)* using the key from the second node. Finally, as expected, it encrypts the data a final time *(with the address of the second node)* using the shared key from the first node.
- Now Alice is ready to send the data to the first node.
- Once the first node received the data, it decrypts it using the shared key. This reveals the address to the second node. The key is here that the first node cannot see the data nor where it is going, since that is still encrypted.
- Next, the first node forwards the data that it just unencrypted to the second node. Again, this node decrypts the data, revealing the address to the third node but now it doesn't know what the data is, where the final destination is or where it originally came from.
- Lastly, the second node forwards the data to the third node. After encryption, this final node can see the data and where it is going but it does not know where it came from. So it forwards the data to Bob and not a single party should be able to know the data, the final destination and where it originally came from except for Alice and Bob.

Now we know why the protocol is called onion routing because it encrypts the data in multiple layers and at every node, one of the layers of the onion is peeled off [31]. The key is that none of the nodes know the complete path that has been taken.

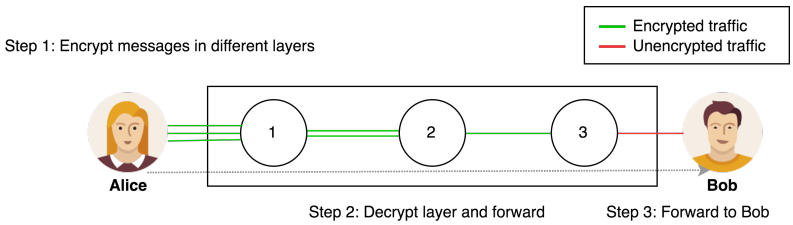


Figure 2.2: Sending a message with the onion routing protocol.

3. **Connection Tear-down:** This can be initiated by any of the nodes or the client and the process is very straightforward. Either the client sends a request for a tear-down to the first node to remove any data on the connection (including the shared key), which is then forwarded to the other nodes. Or one of the nodes sends a tear-down message to both the previous node and the next node, which are then forwarded in both directions [11].

Tor generally uses the same circuit for connections that happen within around 10 minutes after creating a circuit. Later requests, however, are given a new circuit [38, 31].

2.2 Related Work

2.2.1 Website Fingerprinting

Website fingerprinting (WF) is the process of attempting to identify which web pages a specific client visits by analysing their traffic traces. Hence, an attacker is considered to be *local* . By this we mean that they can eavesdrop on the traffic between the client and the first Tor node, as shown in figure 2.3. So it can be anyone from a person on the same network to an ISP. The reason as to why the attacker has to be local is because in onion routing systems, it is the only place in the network where you still know the true identity of the client.

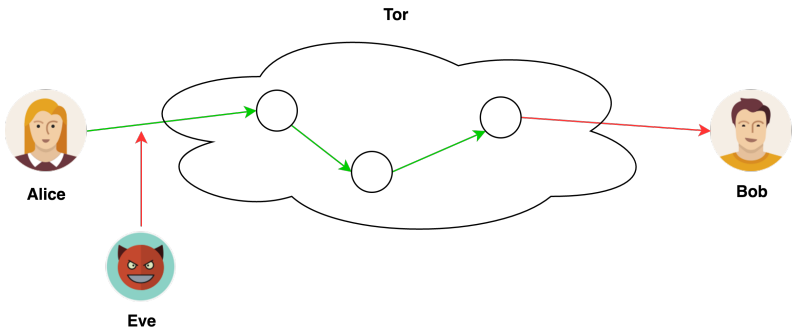


Figure 2.3: Threat model for a simple website fingerprinting attack.

In general, the attack is as follows. The attacker first collects a dataset that contains packet traces for loading several web pages. In practice, these web pages are the ones that an attacker is interested in monitoring. From those traces, the attacker extracts a fixed number of features, the so-called *fingerprint*. Next, it uses those fingerprints to train some

sort of model *(often a machine learning model)* to find patterns in the data. The attacker observes packet traces, generated by the client, and uses the model to classify which web pages the user is visiting. Therefore, an attack is essentially a classification problem, where you try to map packet traces into web pages.

We should denote that throughout this report, we will be using the word "web pages" rather than "websites". The reasoning behind this is that a website often consists of multiple HTML documents, which can result in significantly different traffic traces, depending on which document you load.

The term *"website fingerprinting"* was first mentioned by A. Hintz who performed a simple traffic analysis on Safeweb, an encrypting web proxy that tried to protect user's privacy [17]. Although the attack was very simple, it and other earlier works show the possibility that encrypted channels might still leak information about the URL [17, 42]. Later, in 2009, the first fingerprinting attack against Tor was executed by Herrmann et al. using a *Multinomial Naive Bayes* model. They managed to get a relatively high accuracy for several protocols, such as *OpenSSL* and *OpenVPN*, on a small set of 775 web pages. However on more specialised anonymization networks, the model only achieved a 2.96% for Tor, which they said was caused by a suboptimal configuration [16].

Around the same period Y. Shi et al. performed an attack that specifically focused on anonymity systems such as Tor [34]. Using a cosine similarity, they achieved around 50% accuracy on an even smaller dataset of only 20 web pages [34]. The first real threat however, was by A. Panchenko et al. who used a *Support Vector Classifier* (SVC) on the same dataset of 775 web pages as Herrmann et al. and got a 54.61% success rate [16, 28].

All of the previously mentioned research, except the one done by Panchenko et al. have only considered a *closed-world scenario*. A closed-world setting means that all of the web pages are known in advance [28]. For instance, when Herrmann et al. trained their model on a dataset of 775 web pages, they made the assumption that a client could only visit one of those web pages and none other. In an *open-world scenario*, the attacker does not know in advance which URLs the victim might visit. The most prominent example of this is when the authorities want to monitor which people try to access a set of censored sites [28]. In order to achieve this, the models need to be trained on both *monitored* and *unmonited web pages*.

Wang et al. later conducted an attack on Tor using a large open-world dataset [44]. Using a novel *k-Nearest Neighbor* (k-NN) classifier with *weight adjustment* on a very large feature set *(3736 features)*. In addition to getting around 90% accuracy, they also significantly reduced the time needed for training [44].

Using a completely different approach, Hayes et al. extract fingerprints using a *random forests* [15]. This novel technique involved storing a *fingerprint* as a set of leaves within that forest. Next, they simply use the *hamming distance* as a distance metric to find the k-nearest neighbors. If all the labels within those k instances are the same, the model classifies the new instance as the previously mentioned label. The interesting aspect is that changing the value of k allows them to vary the number of *true positives* (TP) and *false positives* (FP) [15].

Finally, Panchenko et al. improved upon their previous attack to create one current state-of-the-art methods. They tested their approach on several datasets, including the one used by Wang et al. on their k-NN attack, where they got around 92% accuracy. In an open-world scenario, on the other hand, they got up to 96% accuracy.

2.2.2 Automatic Feature Selection

There are not many works that have examine the use of automatic feature selection techniques in the context of a website fingerprinting attacks. First, Abe et al. study the use of

a *stacked autoencoder* with a *softmax classifier* [1]. However, since a stacked autoencoder still requires a fixed-length input, they pad and truncate cells, whose length is shorter or longer than 5000. With it, they manage to achieve a 88% accuracy [1].

V. Rimmer takes a very similar approach, as she also uses a stacked autoencoder with a softmax classifier [32]. But rather than padding and truncating the cells, she transforms the traces into a fixed-length histogram or *wavelength coefficients*. With this, she manages to achieve a 71% accuracy.

2.2.3 Defenses

Not only has there been research regarding different attacks but there are also various works that describe possible defenses. First of all, Tor already implements *padding*, which means that all packets are padded to a fixed-sized cell of 512 bytes. Next, in response to the first attack by Panchenko et al., Tor also supported randomized ordering of HTTP pipelines [28, 15, 30]. Finally, on top of these defenses, fingerprinting on Tor is made more difficult by all of the background noise present. This is due to the fact that Tor also sends packets for circuit construction or just *SENDME's*, which ensure flow control [27]. Although Wang et al. proposed a probabilistic method to remove these [43], they might still make the classification process slightly more difficult.

Lua et al. designed an application-level defense, that was able to successfully defends against a number of classifiers by modifying packet size, timing of packets, web object size, and flow size [30]. This is achieved by splitting individual HTTP requests into multiple partial requests, using extra HTTP traffic as a cover and making use of HTTP pipelining [7]. Although this has been a relatively effective technique to obfuscate traffic, several attacks have proven that this defense only still does not suffice [7, 44].

BuFLO, on the other hand, is a *simulatable* defense [44], designed by Dyer et al. that performed packet padding and sending data at a constant rate in both directions [10]. The disadvantage of this method is the high overhead required in order to keep the constant data rate. Some extensions have been described that try to minimize this overhead such as Nithyanand's work that uses existing knowledge of a website traces to maintain a high level of security [23].

More recently, Cherubin et al. developed the first website fingerprinting defense on the server side [8]. This can be particularly interesting for *Tor hidden services* that want to provide, all of their users, the privacy that they require. The attack uses a technique, called *ALPaCA*, which pads the content of a web page and creates new content to conceal specific features on a network level [8].

Finally, there are also some other techniques such as *decoy pages* and *traffic morphing* [46, 28]. Decoy pages or *camouflage* is a very simple technique that involves loading another web page whenever a web page is requested. This process provides more background noise that makes fingerprinting more difficult [28]. Whilst traffic morphing is a slightly more complex technique that changes certain features in the traffic in order to make it appear as if another page is loaded [46].

2.2.4 Critical Analysis

Most attacks, that have been described above, are based on a set of different assumptions. Here we list these assumptions and see if they are reasonable.

The first one we examine is the open and closed-world scenarios. One of the main problems with website fingerprinting is the amount of web pages readily available on the web. An open-world scenario tries to solve this issue by only classifying a small amount of web pages and by labelling the other ones as unmonitored. However, machine learning theory states that the bigger the world size, the more data is required. So the small size of

the *hypothesis space* compared to our world size, could have a direct impact on the amount of false positives since the more web pages there are, the higher the probability that one of the traces will be very similar to one in the monitored set. Therefore, the false positive rates, described in the previously mentioned papers, might be higher in real life [29].

Nonetheless, even if those false positive rates are accurate, a very small amount of false negatives could have a large impact on the classification. M Perry shows that if the FP rate is as low as 0.2% and just a 100 page loads, around 18% of the user base would be falsely accused of visiting at least one monitored website. Or after 1000 page loads, this percentage increases to around 86% [29].

There are also a variety of different factors that are often not considered such as the rapidly changing nature of some web pages. Juarez et al. show that it takes around 9 days for the accuracy to drop from 80% to under 50% [20]. Additionally, the content of some web pages is dynamic and some of the traces will vary, depending on who visits the website, making the classification for a large set of people difficult. Not only is dynamic websites an issue but different users will also be using different versions of the *Tor Browser Bundle* and might load the web page from different locations. This can decrease the accuracy with 70% and 50% respectively [20]. On top of this, we also need to consider multi-tab browsing, where a client might be loading multiple web pages at the same time. Although some papers consider this [14], most assume that the attacker know where the trace of a single page starts and ends.

2.3 Deep Learning and Automatic Feature Selection

In the following section, we will give a very short introduction to deep learning and describe some of the deep learning solutions that allow us to perform feature extraction. We start by introducing deep learning by exploring *artificial neural networks* and *stacked autoencoders*. Then we move on to *recurrent neural networks* and *sequence-to-sequence models*. Finally, we describe some of the issues with deep learning, such as the *vanishing gradient problem*. All of these explanations assume some familiarity with neural networks and only aim to give a high-level overview of the most important concepts.

Machine learning models basically take some value as its input and output a value, whilst trying to minimize some sort of error. All of the learning models that we explore below are forms of *supervised learning*. This means that we know the expected output and minimize the error between the actual output and the expected output.

2.3.1 Artificial Neural Networks

Artificial neural networks consist of a network of nodes, called *neurons*. These neurons are named and modeled after their biological counterparts. One of the simplest ones, is called a *perceptron*, which consists of a set of *binary inputs*, *weights* and an *activation function*. Hence, essentially it weights different evidence by assigning a different weight to every input. Next, the output of the neuron can be calculated as follows:

$$\text{output} = f(\sum_i w_i x_i)$$

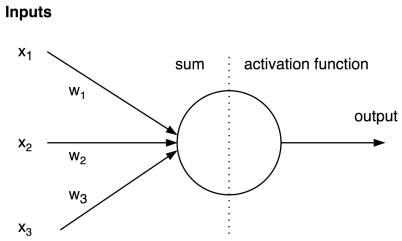
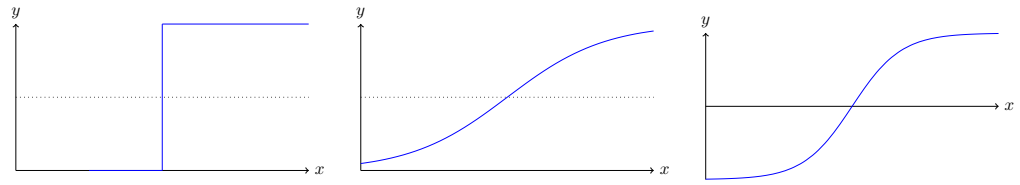


Figure 2.4: Model of a perceptron with three inputs.

This function f represents the *activation function*. Essentially, the activation function expresses the idea that a neuron can 'fire' after the sum of the inputs exceeds a certain threshold. There are certain different functions such as the *step function*, *sigmoid function* and the *tanh function*, which are all outlined in figure 2.5.



(a) The step function. (b) The sigmoid function. (c) The tanh function.

Figure 2.5: Examples of different activation functions

Now to learn a function, it adapts the weight w_i such that it minimizes the error between the predicted and the expected output. In order to achieve this, we need some manner of quantifying the error, called a *loss function*. The most commonly used ones are the *mean squared error* (MSE $= (x - x')^2$) and *absolute loss* (AL $= |x - x'|$) [22]. There are may other cost functions such as *cross-entropy*, but they are not often used for the models that will be described below.

To build a neural network, several of these perceptrons are connected together. The most standard network is a *multilayer perceptron*, also called a *feedforward neural net*. This specific network, as can be seen in figure 2.6, consists of an *input layer*, one or more *hidden layers* and an *output layer*. All of these layers have an arbitrary number of neurons and the connections between these neurons can only go from left to right and can never form a loop. It is known that these kinds of networks can learn to approximate any function [41].

Now that we know how to construct these networks, we still need to manage to assign the appropriate weights to every connection such that the loss function is minimized. There weights can be learnt by using an algorithm, called *backpropagation*. The optimization process usually starts with initializing all of the weights with a random value and then running backpropagation, which is structured as follows [22]:

1. Compute the outputs, given a certain input *(feedforward pass)*.
2. Calculate the error vector.
3. Backpropagate the error by computing the differences between the expected output and the actual output, starting at the output layer and working towards the input layer.
4. Compute the partial derivatives for the weights.
5. Adjust the weights by subtracting these derivatives, multiplied by the *learning rate*.

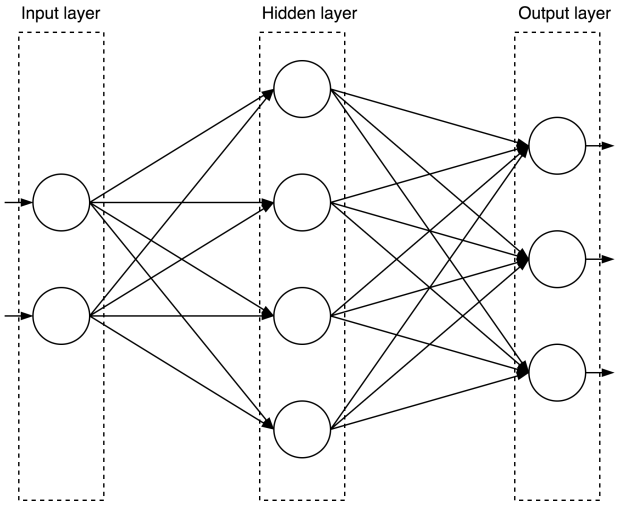


Figure 2.6: Example of an feedforward neural network with one hidden layer

The learning rate is essentially a hyperparameter to the model that defines how fast the network learns. If its value is high, the model learns quickly and if the value is low, the model learns more slowly but the learning process will be more accurate. The propagation process is also often done in *batches*, which means that you calculate the propagations of a fixed amount of input vectors and only once this has finished, the weighs are changed. The size of these batches is a hyperparameter of the model.

2.3.2 Stacked autoencoder

These feedforward neural net can be used to perform feature selection, by using a network called an *autoencoder*. This network tries to learn the *identity function* $f(x) \approx x$ when the number of neurons in the hidden layer is smaller than the ones in the input and output layers. Hence, essentially the network is trying to learn to learn how to compress the initial feature vector [2].

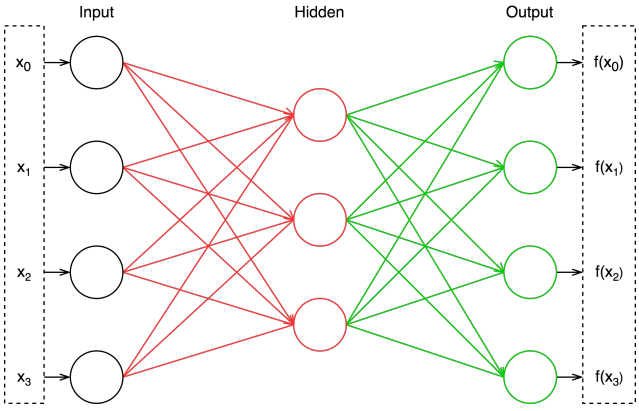


Figure 2.7: Example of a simple autoencoder

In order to learn an even more compressed representation of the input, multiple layers can be introduced, where each hidden layer contains even less nodes than the previous one. However, the problem with this approach is that the deeper the network, the more difficult it can be learn the appropriate weights [22]. A solution to this problem is a greedy approach

where each layer is trained in turn and then stacked on top of each other. By this we mean that we first train the first layer, just as before. Next, the weights of the first layer are used to transform the raw input in a compressed representation. This representation is then used to train the second layer and so on.

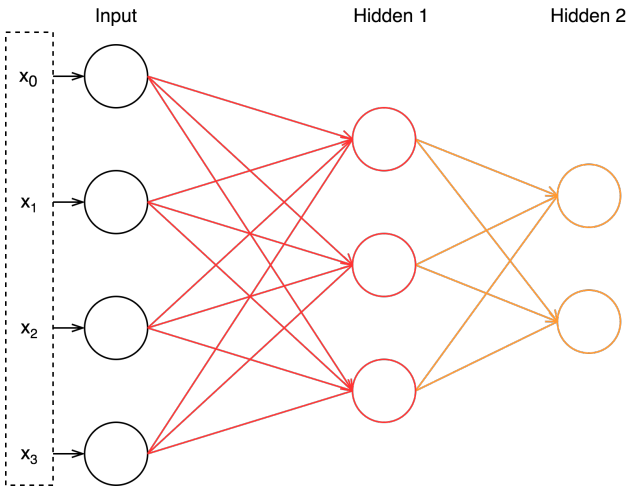


Figure 2.8: Stacked autoencoder

2.3.3 Recurrent Neural Networks

Although stacked autoencoders have successfully been used in several cases [32, 45], they still have a couple of drawbacks. First of all, they require a fixed length input, which means that sequential information will need to be preprocessed. Next, they also assume that all of the inputs and outputs are independent of each other [6] whilst this might not be the case for sequential data such as the cell traces. Hence, we will look at *recurrent neural networks* (RNNs), which relaxes some of the restrictions with a feedforward neural net. They allow connections to form loops.

These loops basically represent that the network can be *unrolled*. This we mean that if we have the same network as in figure 2.9 and you have a sequence that has a length of n, you unroll the network for n steps. Hence, essentially an RNN has 'memory' that affects the outcome of the computation [6]. At every step, you can calculate the output by performing the calculation:

$$\text{result}, h_t = f(h_{t-1}, x_t)$$

Unlike traditional deep networks, every layer in the unrolled network shares the same weights W. This greatly reduces the amount of parameters the network needs to learn [6].

One of the problems with RNNs are that they struggle to learn long-term dependencies [4]. But some cells, more specifically *long short term memory* (LSTM) and *gated recurring unit* (GRU) cells do not have this problem [18, 25, 9]. Rather than just having a single neural network layer, LSTMs consist of four different layers, that all interact in a novel way [25]. These interactions are outlined in figure 2.10. Although a full description is outside the scope of this paper, essentially the top line remains relatively unchanged and can therefore store long-term dependancies. Whilst the bottom line contains more short term information [25, 18].

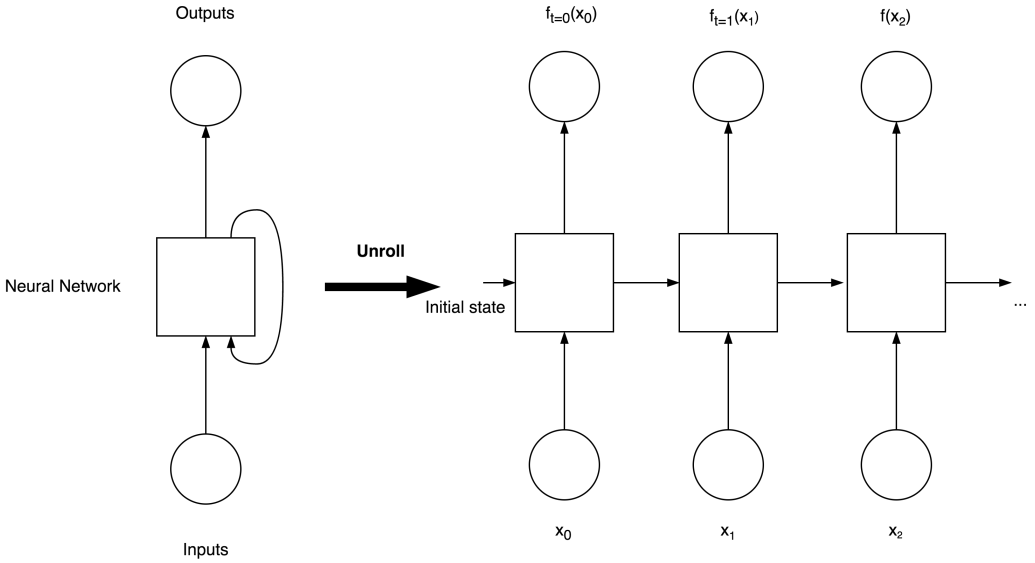


Figure 2.9: Structure of a RNN and an example of the unrolling process [6].

The *gated recurring unit* (GRU) is a slightly more simple model than an LSTM. The cell combines several of the gates and states to minimize the amount of parameters the model needs to learn. Therefore, GRU allows a model to learn at a faster space. Whilst LSTM cells have a greater expressive power [18, 25, 9].

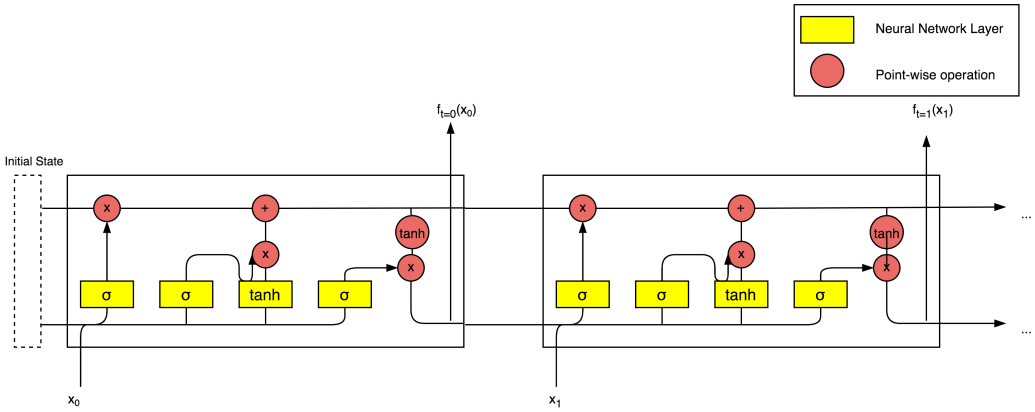


Figure 2.10: Interactions within a LSTM cell [25].

Recurrent neural nets cannot make use of the same backpropagation algorithm used by feedforward neural nets since the unrolled layers share the same set of weights. Instead, we use a different optimization method, called *backpropagation through time* (BPTT). BPTT is very similar to standard backpropagation with the key difference of summing up the gradients for the weights at every time step [6].

2.3.4 Sequence-to-Sequence Model

One specific type of RNN, that could be used for feature generation, is a *sequence-to-sequence model*, introduced by Cho et al. [9]. Sometimes called a *encoder-decoder model*, it consists of two RNNs where one RNN encodes a sequence into a fixed-length representation and the other decodes those representations into another variable-length sequence [9]. The

models have been proven to be successful in several *natural language processing* (NLP) tasks such as translation tasks [9, 36, 33]. However, since the model extracts a fixed-length representation, we might be able to train a sequence-to-sequence model on a copy task, just like an autoencoder, and use that *thought vector* as the extracted features.

The model works as follows, each box in figure 2.11 represents an RNN cell. Next, the encoder RNN is unrolled, depending on the length of the input vector. After the encoder is finished, the final state vector is stored in a *thought vector* variable. Next, the decoder is unrolled and run, with the though vector as its initial state and a start token as the first input. Finally, the output of the previous cell is then used as the input to the next cell until the network produces the end of sequence token [9]. The encoder and the decoder can share the same weights or, as is more common, use a different set of parameters [33].

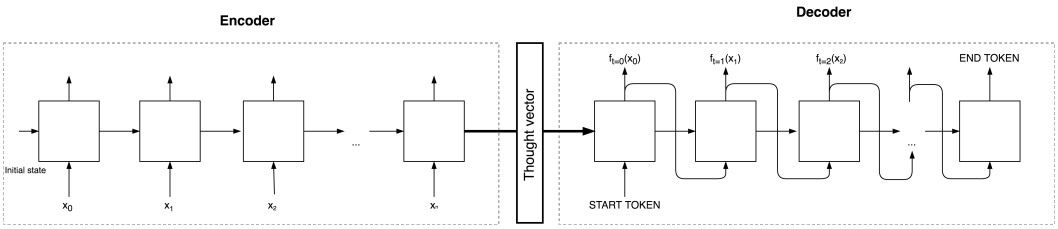


Figure 2.11: Structure of a sequence-to-sequence model [5].

If this model is now trained on a copy task, where it tries to reproduce the input sequence from a thought vector, it means that just like an autoencoder, the network has learned to compress any variable-length sequence into a fixed-length representation. The size of this though vector, however, is another hyperparameter to the model.

To allow a sequence-to-sequence to learn more complex representations, Sutskever et al. experimented with multilayered LSTM cells [36]. They showed that this model was very successful in a English to French translation task and managed to cope with long term dependencies. Additionally, they also showed that reversing the sentences made the learning process easier. Sutskever et al. do not have a complete explanation for this phenomenon but they believe that it is due to the many short-term dependencies in the dataset [36].

In addition to reversing the traces, Oshri et al. introduced the idea of using a *bidirectional RNN encoder* [26]. The idea behind this kind of encoder is based on the fact that the output at time t might not only depend on past information but also on future information. To combat this issue, the encoder consists of two layers, the forward and the backward layer, stacked on top of each other. Next, the output is computed, based on the hidden state of both layers [6]. Finally, to extract the thought vector, the last state of both layers are extracted and averaged.

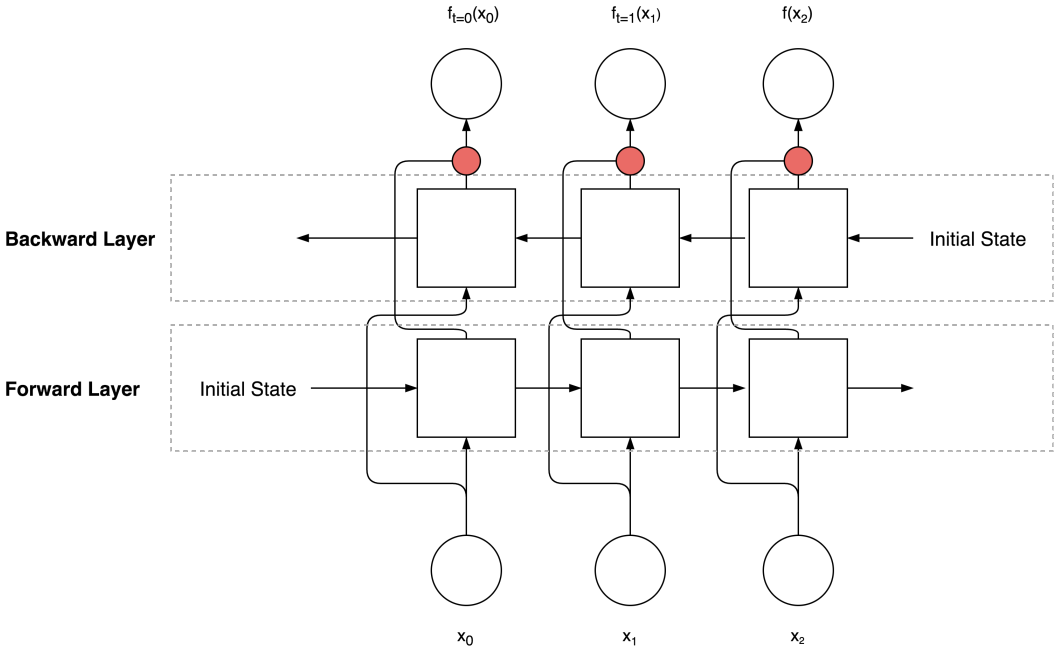


Figure 2.12: Structure of a bidirectional RNN. [6]

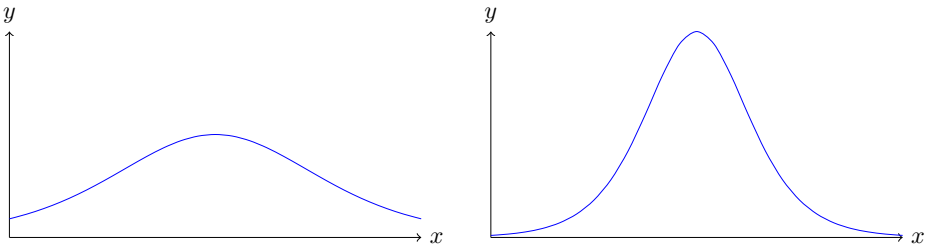
Some research by Bahdanau et al. has been to improve the performance of encoder-decoder models by introducing *attention mechanisms* [3]. This mechanism allows the decoder to have a more direct access to the input, thereby relieving the encoder by having to embed all the information in a fixed-length vector. Although this technique might work well for translation tasks, our work focuses on the feature extraction process. Therefore, we will not perform any analysis on sequence-to-sequence models with an attention mechanism.

At the time of writing, there hasn't been much research regarding using sequence-to-sequence models for feature extraction. In fact, currently they are most often used for translation and other NLP tasks. Hence, they are most often used as a classification task where the model classifies which word is likely to be next [33, 9, 26, 36].

2.3.5 The Vanishing Gradient Problem

Deep learning suffers from an problem, called the *vanishing gradient problem* [22]. This problem represents the fact that neurons in early layers tend to learn a lot slower than those in the last layers. The vanishing gradient problem is also the reason why some RNNs are not able to learn long-term dependencies. This is due to the fact that the derivatives of both tanh and sigmoid activation functions approach zero near both ends. Thus when one of the neurons is saturated, which means that the value of the gradient gets close to zero, it drives the gradients of previous neurons to zero as well [6]. Hence, since those gradients are close to zero, they vanish after a couple of time steps. As RNNs tend to be a lot deeper than traditional feed-forward networks, this problems tends to be a lot more common [6].

One solution to this problem is to use a ReLU activation function, as the derivative is either 0 or 1 [6]. But both LSTM and GRU cells were especially designed to solve this issue in RNNs and therefore we will be using those throughout the rest of this paper [25, 18, 9].



(a) Derivative of the sigmoid function. (b) Derivative of the tanh function.

Figure 2.13: Derivatives of activation functions

2.3.6 Regularization

Neural networks can easily *overfit*. By this we mean that the model learns noise in the training data, rather than learning to generalize. For instance, as can be seen in figure 2.15, the model is generalize when it fits a straight line but overfits otherwise. The reason why neural nets often overfit is because of the large amount of free variables [22].

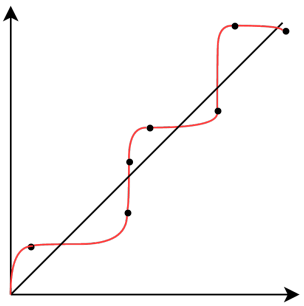


Figure 2.14: Example of overfitting.

Overfitting can be reduced by performing *regularization* without the need to decrease the size of the network. The most popular techniques are *L1*, *L2*, *dropout layers* and *batch normalization* [22]. Both L1 and L2 regularization work by penalizing large weights and thus making the functions less complex. They do this in a different fashion, for instance L1 adds the sum of the absolute values of the weights to the cost function [22]. Whilst L2 has a different *regularization parameter*, which generally performs better in practice [22].

Dropout, on the other hand, doesn't modify the cost function but it changes the network. Essentially, it ignores several neurons while performing an iteration of the back-propagation process. After that iteration, it picks a set of different neurons to ignore and repeats the process. The effect of this is that the network basically consists of an average of various different networks. Each of those neurons will overfit in a different way so the average should reduce the total amount of overfitting [22].

Finally, batch normalization is a relatively new technique that also changes the network slightly by making the normalization process part of the network and performing it for every minibatch [19]. Usually this wouldn't be considered a regularization technique. However, it has been proven that it acts as a regularizer, and sometimes eliminates the need for dropout layers [19].

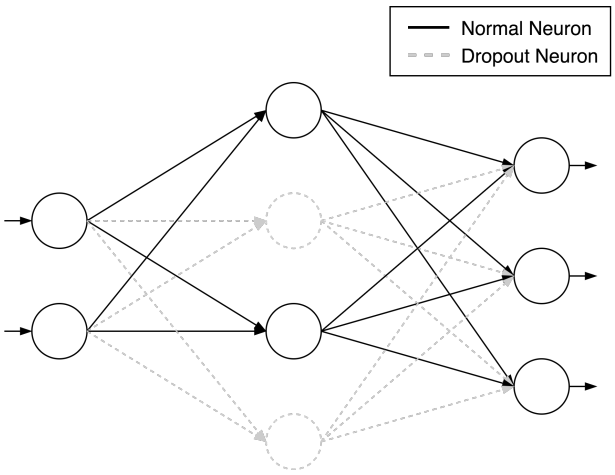


Figure 2.15: Feedforward network with dropout.

2.4 Software Libraries

There are various different numerical computation libraries to efficiently implement deep learning algorithms. However, the most common ones are *Tensorflow*, *Theano*, *Keras* and *Torch* [37, 39, 21, 40]. The first three all provide their API in Python, whilst torch only be used in Lua. Although a python API, called *Pytorch* has been open-sourced very recently. All of the above allow you to use *cuDNN* to perform GPU-accelerated computations.

Tensorflow has a C++ backend and allows GPU computations to speed up computations. The low-level API is provides more than enough flexibility to implement a sequence-to-sequence model. They do also provide a high-level API that include different RNN cells and even a sequence-to-sequence model. However, this model has been specifically designed for NLP tasks and therefore does not support the computations that we are trying to achieve [37]. Theano is very similar to Tensorflow but it does not provide all of the high-level tools [39].

Keras, on the other hand, runs on top of either Tensorflow or Theano as a backend. Hence, the library provides high-level functions to create different models, rather than the low-level APIs provided by both Tensorflow and Theano. Although this library would allow us to quickly prototype a sequence-to-sequence model, it does not provide the low-level access that we might need in order to be able to tune certain hyperparameters [21]. Finally, Pytorch is also a very attractive option but the current release is still a beta version [40].

After a careful analysis, Tensorflow provides us with the highest flexibility whilst still providing tools to easily perform certain calculations with their high-level API. Hence, the decision to use Tensorflow for the implementation of our main model. On top of Tensorflow, we will also be using sklearn, which provides us with a large amount of machine learning models to perform some of our testing, and numpy for fast data preprocessing.

2.5 Data Sets

In order to collect data, a web crawler needs to visit a large set of website over Tor where the traffic is recorded for every page visited. This web crawler can emulate user browsing or just visit websites such as the one constructed by Panchenko et al. [27] or simply visit web pages in *Alexas Top 10,000*, which contains the most commonly visited pages. Or real data can be collected from users but due to privacy concerns, this data is often hard to get

by.

After a web-crawler has been set up, data is collected through a TCP dump. This data is then often preprocessed and converted into *Tor cells*. The reason for this is because Tor pads packets to a fixed-length (512 bytes) [38] hence these cells are a simple representation of the traffic. Next, probabilistic techniques are used to remove SENDMEs [43]. After this processing, a cell looks like a list of tuples, each containing a time value and a class label, which represent whether it was an incoming or outgoing packet.

There are several data sets readily available that have already been preprocessing. Some of the largest ones are the one used by Wang et al. [44], Greschbach et al. [13]. Wang et al's dataset contain traces for a 100 monitored websites with 90 instances each and 8400 unmonitored sites, all from Alexa's top 10,000. Greschbach et al. collected an even larger dataset with 100 sampled of Alexa's top $9,000$ websites and one sample for $909,000$ unmonitored sites [13].

3 Attack Design and Implementation

3.1 Threat Model

3.2 Design Overview

3.3 Sequence-to-Sequence Model

3.4 Website Fingerprinting Models

3.5 Hand-picked Features

4 Evaluation and Testing

4.1 Evaluation Techniques

4.2 Evaluation

4.2.1 Data Collection

4.2.2 Sequence-to-Sequence Model

4.3 Testing

5 Conclusion

5.1 Future Works

- TODO: How it performs against defenses - TODO: Regularisation - TODO: Tor hidden services - TODO: Collecting data over time - TODO: More realistic user behavior

Bibliography

[1] Kota Abe and Shigeki Goto. "Fingerprinting Attack on Tor Anonymity using Deep Learning". In: *Proceedings of the APAN Research Workshop 2016* ().

[2] *Autoencoders*. URL: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/.

[3] Dzmitry Bahdanau and KyungHyun Cho. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ICLR* (2015). URL: https://arxiv.org/pdf/1409.0473.pdf.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[5] Denny Britz. *Deep Learning for Chatbots*. May 2016. URL: http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/.

[6] Denny Britz. *Introduction to RNNs*. July 2016. URL: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[7] Xiang Cai et al. "Touching from a Distance: Website Fingerprinting Attacks and Defenses". In: *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12* (2012). DOI: 10.1145/2382196.2382260. URL: http://pub.cs.sunysb.edu/~rob/papers/fp.pdf.

[8] Giovanni Cherubin, Jamie Hayes, and Marc Juarez. "Website Fingerprinting Defenses at the Application Layer". In: *Proceedings on Privacy Enhancing Technologies* 2 (2017), pp. 165–182.

[9] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014). DOI: 10.3115/v1/d14-1179. URL: https://arxiv.org/pdf/1406.1078v3.pdf.

[10] Kevin P Dyer et al. "Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 332–346.

[11] David Goldschlag, Michael Reed, and Paul Syverson. "Onion routing". In: *Communications of the ACM* 42.2 (1999), pp. 39–41.

[12] Google. "Google Transparency Report". In: (Mar. 2017).

[13] Benjamin Greschbach et al. "The Effect of DNS on Tor's Anonymity". In: *arXiv preprint arXiv:1609.08187* (2016).

[14] Xiaodan Gu, Ming Yang, and Junzhou Luo. "A novel Website Fingerprinting attack against multi-tab browsing behavior". In: *2015 IEEE 19th International Conference on Computer Supported Cooperative Work in Design (CSCWD)* (2015). DOI: 10.1109/cscwd.2015.7230964.

[15] Jamie Hayes and George Danezis. *k-fingerprinting: A Robust Scalable Website Fingerprinting Technique*. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes.

[16] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive bayes classifier". In: *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM. 2009, pp. 31–42.

[17] Andrew Hintz. "Fingerprinting websites using traffic analysis". In: *International Workshop on Privacy Enhancing Technologies*. Springer. 2002, pp. 171–178.

[18] Sepp Hochreiter and Jurgen Schmidhuber. "Long short term memory". In: *Neural computation* 9.8 (1997), pp. 1735, 1780.

[19] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[20] Marc Juarez et al. "A Critical Evaluation of Website Fingerprinting Attacks". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14* (2014). DOI: 10.1145/2660267.2660368. URL: http://www1.icsi.berkeley.edu/~sadia/papers/ccs-webfp-final.pdf.

[21] *Keras: Deep Learning library for Theano and TensorFlow*. URL: https://keras.io/.

[22] Michael Nielsen. *Neural Networks and Deep Learning*. 2017.

[23] Rishab Nithyanand, Xiang Cai, and Rob Johnson. "Glove: A bespoke website fingerprinting defense". In: *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. ACM. 2014, pp. 131–134.

[24] *NumPy Documentation*. URL: http://www.numpy.org/.

[25] Christopher Olah. *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[26] Barak Oshri and Nishith Khandwala. "There and Back Again: Autoencoders for Textual Reconstruction". In: (). URL: http://cs224d.stanford.edu/reports/OshriBarak.pdf.

[27] Andriy Panchenko et al. "Website Fingerprinting at Internet Scale". In: (). URL: https://www.comsys.rwth-aachen.de/fileadmin/papers/2016/2016-panchenko-ndss-fingerprinting.pdf.

[28] Andriy Panchenko et al. *Website fingerprinting in onion routing based anonymization networks*. 2011. DOI: 10.1145/2046556.2046570. URL: https://www.freehaven.net/anonbib/cache/wpes11-panchenko.pdf.

[29] Mike Perry. *A Critique of Website Traffic Fingerprinting Attacks*. URL: https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks.

[30] Mike Perry. "Experimental defense for website traffic fingerprinting". In: *Tor project Blog* (2011). URL: https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting.

[31] The Tor Project. "Tor Overview". In: (). URL: https://www.torproject.org/about/overview.html.en.

[32] Vera Rimmer. "Deep Learning Website Fingerprinting Features". In: (2016). URL: https://www.esat.kuleuven.be/cosic/publications/thesis-280.pdf.

[33] *Sequence-to-Sequence Models*. URL: https://www.tensorflow.org/tutorials/seq2seq.

[34] Yi Shi and Kanta Matsuura. "Fingerprinting attack on the tor anonymity system". In: *International Conference on Information and Communications Security*. Springer. 2009, pp. 425–438.

[35] *Stacked Autoencoders*. URL: http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders.

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: (). URL: https://arxiv.org/abs/1409.3215.

[37] *Tensorflow Documentation*. URL: https://www.tensorflow.org/.

[38] *The Tor Project*. URL: https://www.torproject.org/docs/faq.

[39] *Theano Documentation*. URL: http://deeplearning.net/software/theano/.

[40] *Torch Documentation*. URL: http://torch.ch/.

[41] Gregory Valiant. "Learning polynomials with neural networks". In: (2014).

[42] David Wagner, Bruce Schneier, et al. "Analysis of the SSL 3.0 protocol". In: *The Second USENIX Workshop on Electronic Commerce Proceedings*. Vol. 1. 1. 1996, pp. 29–40.

[43] Tao Wang and Ian Goldberg. "Improved website fingerprinting on Tor". In: *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society - WPES '13* (2013). DOI: 10.1145/2517840.2517851. URL: https://www.freehaven.net/anonbib/cache/wpes13-fingerprinting.pdf.

[44] Tao Wang et al. "Effective Attacks and Provable Defenses for Website Fingerprinting". In: *USENIX Security Symposium* 23 (Aug. 2014). URL: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-wang-tao.pdf.

[45] Zhanyi Wang. "The Applications of Deep Learning on Traffic Identification". In: (). URL: https://www.blackhat.com/docs/us-15/materials/us-15-Wang-The-Applications-Of-Deep-Learning-On-Traffic-Identification-wp.pdf.

[46] Charles V Wright, Scott E Coull, and Fabian Monrose. "Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis." In: *NDSS*. Vol. 9. 2009.