University College London

Dept. Computer Science

Automatic Feature Selection for Website Fingerprinting

Axel Goetz

BSc. Computer Science

Supervisors:

Dr. George Danezis

Jamie Hayes

April 16, 2017

Abstract

Anonymity networks such as Tor enable their users to anonymously browse the Internet. But there are several attacks that allow adversaries to identify users, one of these is called *website fingerprinting*, which relies a local, passive eavesdropper who performs pattern analysis on the encrypted data to classify web pages. In most prior works, adversaries extract *fingerprints* by relying on a time-consuming, laborious feature extraction process. Here, we present several deep-learning techniques (*stacked autoencoder* and a *sequence-to-sequence model*) that can be used to automate this process, propose a technique to evaluate their performance and use this technique to compare our models to current state-of-the-art attacks.

We find that the sequence-to-sequence models seems to get the highest performance with a 93% maximum for a binary classification task and a 39% accuracy in the multiclass classification both within an open-world scenario. On top of this, we also show that both deep learning models are robust and can be used to extract fingerprints from data that was recorded under different circumstances.

Although our approach currently does not outperform the state-of-the-art attacks, it shows us that it is in fact possible to automate the feature extraction process without the need of any domain-specific knowledge.

Contents

Chapter 1

Introduction

1.1 The Problem

The internet has become an essential tool for communication in the lives of many. But privacy has always remained a major concern. This is the reason why nowadays most web content providers are slowly moving away from HTTP to HTTPS. For instance, at the time of writing, around 86% of Google's traffic is encrypted, which is a significant improvement compared to 2014 when only around 50% of the traffic was sent over HTTPS [13]. However, this encryption only obscures the content of the web page and does not hide what websites a user might be visiting or in general who they might be communicating with.

Consequently, Internet Service Providers (ISPs) can easily obtain a lot of information about a person. This can be a 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 users 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* to anonymize people[44].

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)* [44]. However, in this work 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 satisfy these conditions are known as *website fingerprinting* (WF). This attack relies on the fact that Tor does not significantly alter the shape of the network traffic [16]. Hence, the adversary can infer information about the content by analysing the raw traffic. For instance by examining 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%, which caused Tor to take various countermeasures [31, 34].

However, the main problem with majority of the WF attacks proposed in the research literature is that they rely on a laborious, time-consuming manual feature engineering process, which often requires expert knowledge of the underlying TCP/IP protocols. The reason behind this is that most primitive machine learning (ML) techniques only take fixed-length vectors as their input while a traffic data is represented by a variable-length amount of packets. The features that are often proposed are based on intuition and trial and error arguments as to why they identify specific web pages. But there is no guarantee that they are in fact the most appropriate ones.

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. Next, we aim to use these features in existing attacks to see if our model is successful in the aforementioned task.

1.2 Aims and Goals

We can subdivide the project up into several 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 real WF attack.
   - Analyse the impact of the rapidly changing nature of some web pages.
   - Review if previous works make any assumptions that could impact the effectiveness of a real 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 autoencoders, sequence-to-sequence models and bidirectional encoder-decoder models.
   - Identify a dataset that is large enough to train our unsupervised deep-learning models.
   - 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 features..
   Goals:
   - Identify several different models that have successful been applied to a website fingerprinting attacks and implement those models.
   - Extract the same hand-picked features as have previously been used with the respective attacks.
   - Investigate an appropriate technique for evaluating the results of different models.
   - Compare the hand-picked features to the automatically generated ones for different classifiers. 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 project can be split up into three different aims, which is why we also approach it in three different stages:

- First, we examine different existing website fingerprinting classifiers to gain a deeper understanding of the concept.
- Next, we perform more research into different automatic feature selection models and implement the most appropriate ones.
- Finally, we evaluate the performance of the feature extraction models using different methods such as training existing WF classifiers with the generated features.

1.4 Report Structure

The general report has a very simple structure. In the following section we further explore similar works and several concepts that are necessary to understand the basics of website fingerprinting and our specific attack. Next, we identify the threat model and design an attack that uses our automatic feature generation model. Finally, we explore several methods of evaluating the performance of different feature generation models and use these methods to perform a thorough analysis to see how our attack compares to existing ones.

Chapter 2

Background Information and Related Work

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 will form 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 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 [12]. 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* [12]. We show how the protocol works by providing 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.
   - From these, Alice picks three random Tor nodes and labels them *one, two* and *three*.
   - Alice communicates with the first node and shares a symmetric key.
   - From now on, all the messages that Alice will send will be encrypted and sent to the first node and forwarded from there onwards. Including the messages to share a symmetric key with the second node, such that the second node does not know Alice's true identity.
   - Again, now all messages are first sent to the first node and forwarded to the second node, which it in turn forwards to the third and final node. This way, Alice shares the final symmetric key with the third node.

   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 true identity of Alice.

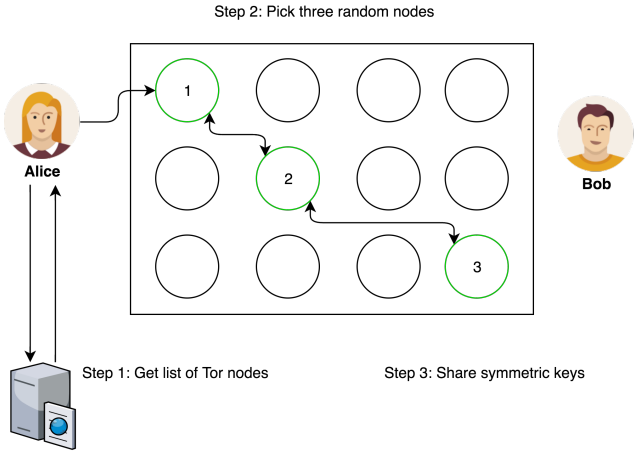


Figure 2.1.1: An example of a connection setup for the onion routing protocol.

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 *(adressed to Bob)* using the shared key from the third node and addresses that encrypted data to the third node. Next, it encrypts that data again using the key from the second node and addresses it to the second node. Finally, as expected, it encrypts the data a final time 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 can it see Bob's address, since that information is still encrypted.
   - Next, the first node forwards the data, it just decrypted, 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 decryption, this final node can see the data and where it is going but it does not know where it originally came from. So it forwards the data to Bob and none of the nodes know all of the following: the data, the final destination or where it originally came from.

   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 [35]. The key is that none of the nodes know the complete path that has been taken.

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 [12].

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 [44, 35].

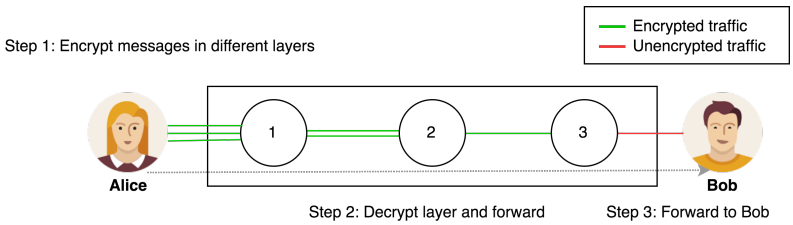


Figure 2.1.2: Sending a message with the onion routing protocol.

2.2 Related Work

2.2.1 Website Fingerprinting

Website fingerprinting (WF) is the process of a *local* adversary attempting to identify which web pages a specific client visits by analysing their traffic traces. By this we mean that the attacker can eavesdrop on the traffic between the client and the first Tor node, as shown in figure 2.2.1. 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.

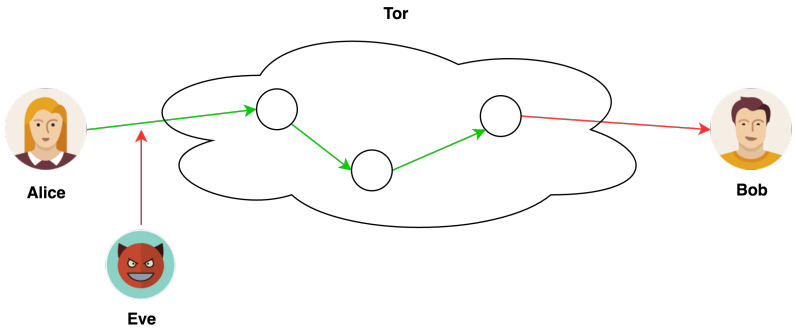


Figure 2.2.1: Threat model for a simple website fingerprinting attack.

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 previously trained 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 term "web pages" rather than "websites". The reasoning behind this is that a website often consists of multiple web pages, 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 [18]. Although the attack was very simple, it and other earlier works show the possibility that encrypted channels might still leak information about the URL [18, 51]. 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 such as Tor, the model only achieved a 2.96%, which they claim was caused by a suboptimal configuration [17].

Around the same period Y. Shi et al. performed an attack that specifically focused on anonymity systems such as Tor [40]. Using a cosine similarity, they achieved around 50% accuracy on an even smaller dataset of only 20 web pages [40]. 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 [17, 31].

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 [31]. 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*, however, 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 [31]. 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 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 [53].

Next, using a completely different approach, Hayes et al. extract fingerprints using a *random forest* [16]. 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 and as an unmonitored page otherwise. The interesting aspect is that changing the value of k allows them to vary the number of *true positives* (TP) and *false positives* (FP) [16].

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. in their k-NN attack, where they got around 92% accuracy. In an open-world scenario, on the other hand, they claim to have gotten 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 [37]. But rather than padding or 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 into fixed-sized cells of 512 bytes. Next, in response to the first attack by Panchenko et al., Tor also supported randomized ordering of HTTP pipelines [31, 16, 34]. 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 and *SENDME's*, which ensure flow control [30]. Although Wang et al. proposed a probabilistic method to remove these [52], 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 [34]. 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, 53].

BuFLO, on the other hand, is a *simulatable* defense [53], designed by Dyer et al. that performs packet padding and sending data at a constant rate in both directions [11]. 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 [25].

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* [55, 31]. Decoy pages, or sometimes called *camouflage*, is a very simple technique that involves loading another web page whenever a page is requested. This process provides more background noise, which makes fingerprinting more difficult [31]. Traffic morphing, on the other hand, 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 [55].

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 analyze if they are reasonable.

The first assumption we examine are 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 considerably higher in a real attck [33].

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%, with 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% [33].

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% [21]. 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 [21]. 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 [15], most assume that the adversary knows 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* and inally, 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 essentially 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 either *supervised* or *unsupervised learning*. Supervised learning means that we know the expected output and minimize the error between the actual output and the expected output. Whilst unsupervised learning tries to find some patterns in the data without knowing the labels.

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*. They essentially weight different pieces of evidence by assigning a different weight to every input. Next, the output of the neuron can be calculated as follows:

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

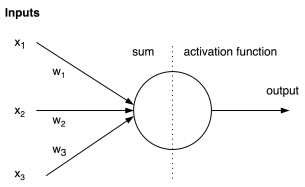
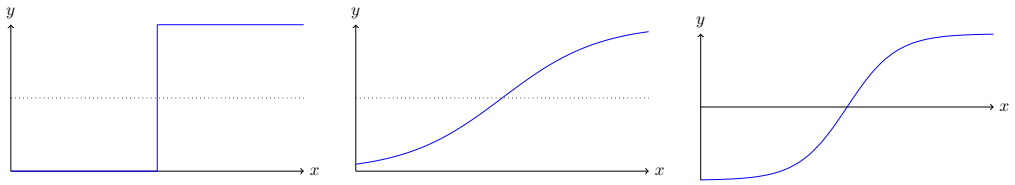


Figure 2.3.1: Model of a perceptron with three inputs.

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 2.3.2.



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

Figure 2.3.2: Examples of different activation functions.

Now to learn a function, the neuron adapts the weights 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* ($\text{MSE} = (x - x')^2$) and *absolute loss* ($\text{AL} = |x - x'|$) [24]. 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 2.3.3, 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 [50].

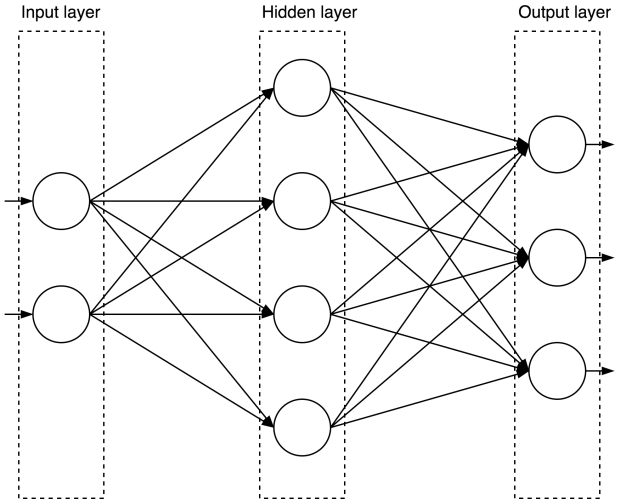


Figure 2.3.3: Example of an feedforward neural network with one hidden layer.

Now that we know how to construct these networks, we still need to be able to assign the appropriate weights to every connection such that the loss function is minimized. These 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 [24]:

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*.

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 the model calculates the propagations of a fixed amount of input vectors and only once this has finished, the weighs are updates such that they minimize the outputs for the entire batch. The size of these batches is a hyperparameter of the model.

2.3.2 Stacked autoencoder

These feedforward neural nets can be used to perform feature selection by using a specific 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].

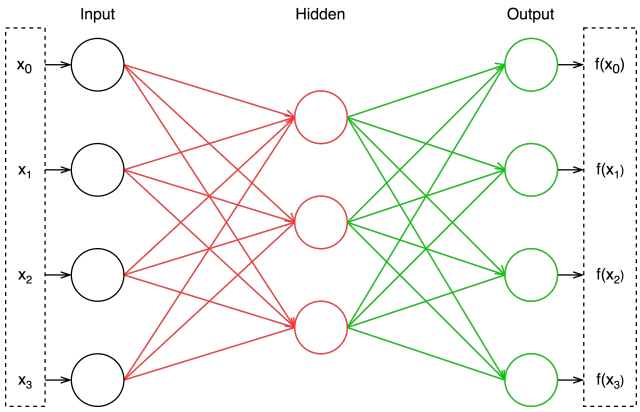


Figure 2.3.4: Structure of a simple autoencoder.

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 to learn the appropriate weights [24]. 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.

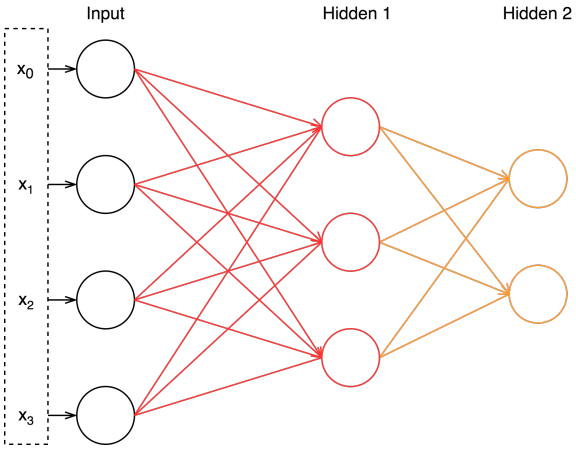


Figure 2.3.5: Stacked autoencoder.

2.3.3 Recurrent Neural Networks

Although stacked autoencoders have successfully been used in several cases [37, 54], 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 of a feedforward neural net. More specifically, 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 2.3.6 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 at every time step [6]. The outputs can be calculated as follows:

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

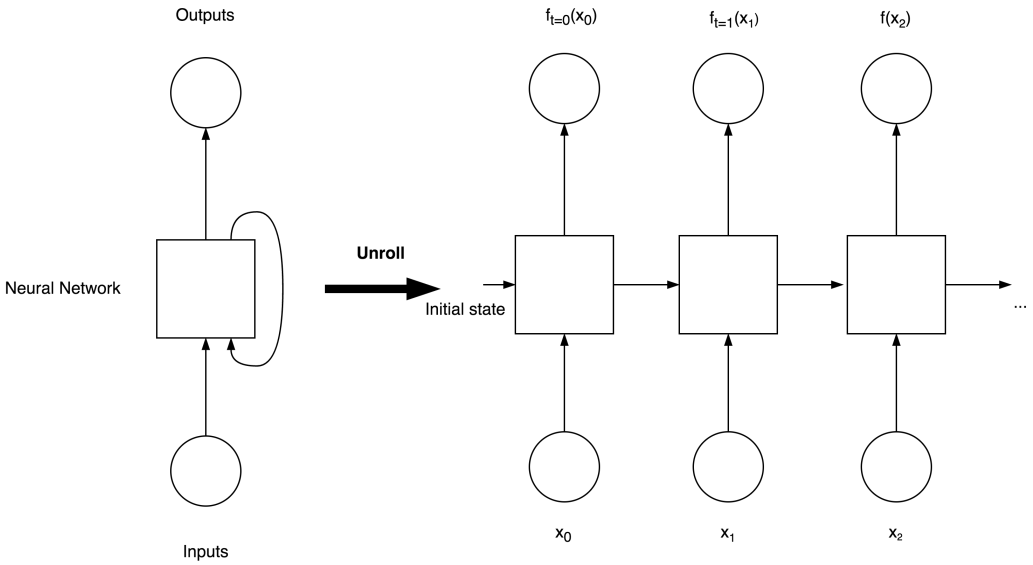


Figure 2.3.6: Structure of a RNN and an example of the unrolling process [6].

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 [19, 28, 9]. Rather than just having a single neural network layer, LSTMs consist of four different layers, that all interact in a novel way [28]. These interactions are outlined in figure 2.3.7. Although a full description is outside the scope of this paper, the general idea is based on the fact that the top state remains relatively unchanged and can therefore store long-term dependancies. Whilst the bottom state contains more short term information [28, 19].

The *gated recurring unit* (GRU) is a slightly more simple version of a 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 pace. Whilst LSTM cells have a greater expressive power [19, 28, 9].

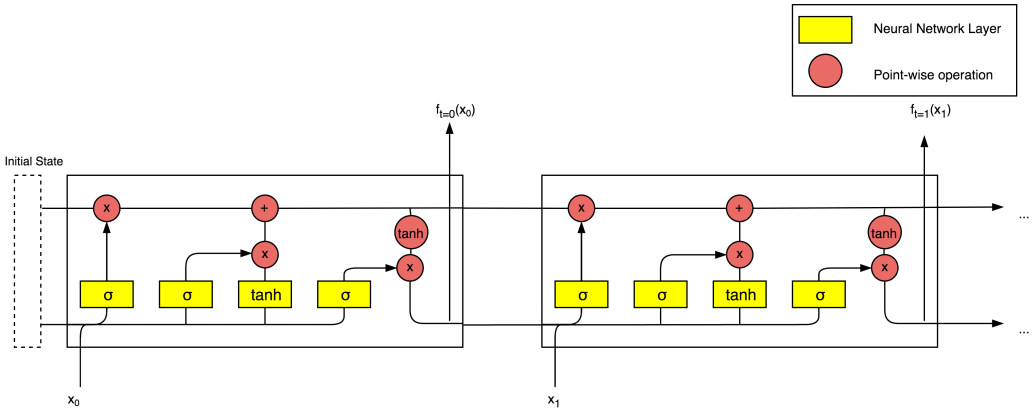


Figure 2.3.7: Interactions within a LSTM cell [28].

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 W. Instead, we use a different optimization method, called *backpropagation through time* (BPTT). BPTT is very similar to standard backpropagation with the key difference being that the gradients for the weights are summed 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*, which was 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, 42, 39]. However, since the encoder produces 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 the *thought vector* as the extracted features.

The model works as follows, each box in figure 2.3.8 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. After that, 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 [39].

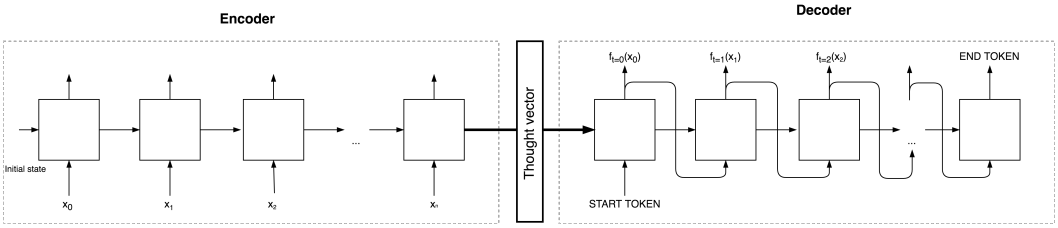


Figure 2.3.8: Structure of a sequence-to-sequence model [5].

If we train this model on a copy task, where it tries to reproduce the input sequence from the 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 [42]. 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 [42].

In addition to reversing the traces, Oshri et al. introduced the idea of using a *bidirectional RNN encoder* [29]. 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.

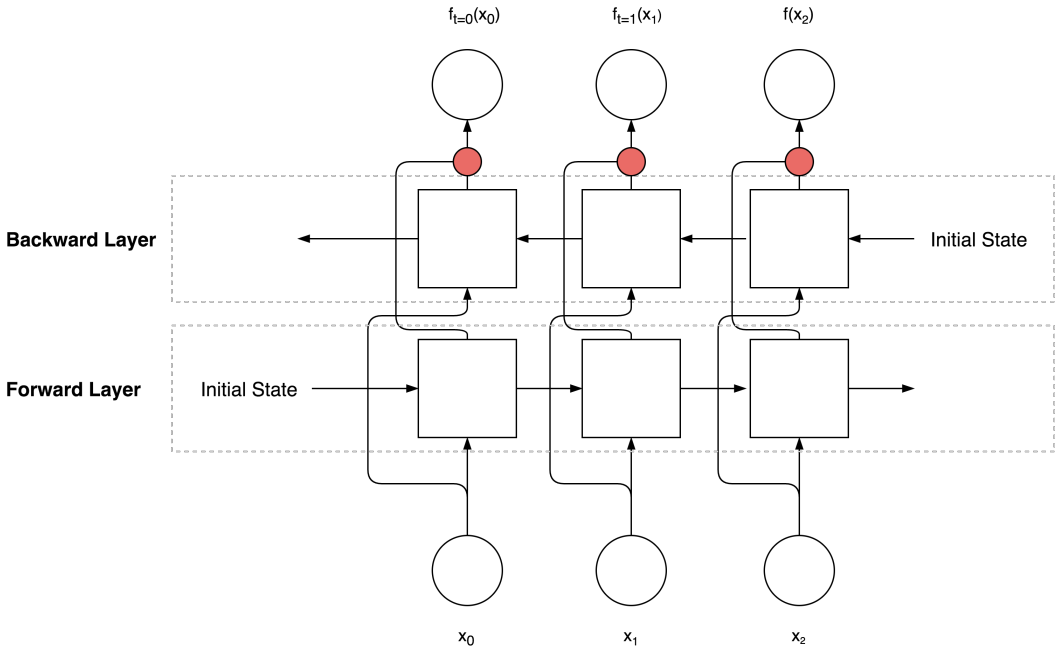


Figure 2.3.9: Structure of a bidirectional RNN [6].

This kind of model sounds more appropriate than one with a simple encoder. However, Oshri et al. argue that the performance gain is limited, whilst introducing double the amount of parameters [29].

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 task. where they are most often used as a classification task where the model classifies which word is likely to be next [39, 9, 29, 42].

2.3.5 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 2.3.10, the model generalizes 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 [24].

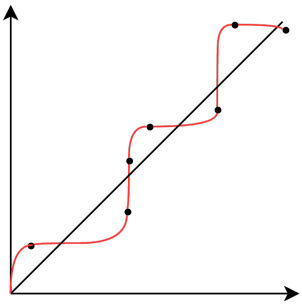


Figure 2.3.10: Example of overfitting.

Overfitting can be reduced without the need to decrease the size of the network by performing *regularization*. The most popular techniques are *L1*, *L2*, *dropout layers* and *batch normalization* [24]. 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 [24]. Whilst L2 has a different *regularization parameter*, which generally performs better than L1 in practice [24].

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 algorithm. 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 networks will overfit in a different way so the average should reduce the total amount of overfitting [24].

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 [20]. 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 [20].

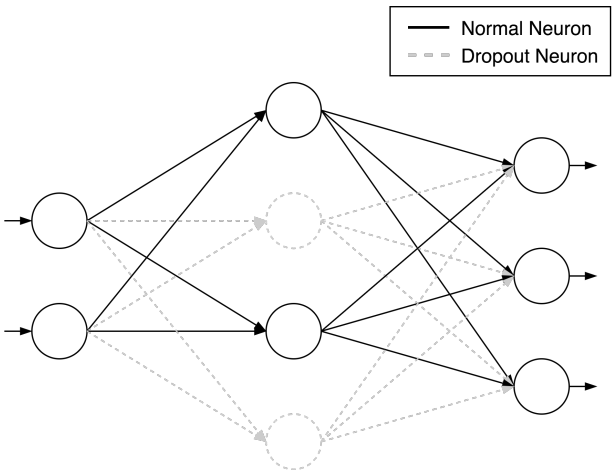


Figure 2.3.11: Feedforward network with dropout.

2.4 Software Libraries

There are various different numerical computation libraries to efficiently implement deep learning algorithms, the most common ones being *Tensorflow*, *Theano*, *Keras* and *Torch* [43, 45, 22, 47]. The first three all provide their API in Python, whilst torch's API is in lua but 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 its low-level API is provides more than enough flexibility to implement the required models. It also provides 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 [43]. Theano is very similar to Tensorflow but it does not provide all of the high-level tools [45].

Keras, on the other hand, runs on top of either Tensorflow or Theano. 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 [22]. Finally, Pytorch is also a very attractive option but the current release is still a beta version [47].

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, we will use Tensorflow to implement all of our deep learning models. 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. Both of which were chosen since they are in python, therefore allowing easy code-reuse for different modules.

2.5 Data Sets

In order to collect the necessary data, a web crawler needs to visit a large set of web pages over Tor where the traffic is recorded for every page visited. This web crawler can emulate user browsing or simply visit web pages in *Alexa's 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) [44] hence these cells are a simple representation of the traffic. Next, probabilistic techniques are used to remove SENDMEs [52]. After this processing, a cell looks like a list of tuples, each containing a time value and a direction, which represent whether it was an incoming or outgoing packet.

Timestamp	**Direction**
0.0	OUT
0.0630009174347	OUT
0.575006008148	IN
0.575006008148	IN
0.691473960876	IN
0.719605922699	OUT

Table 2.1: Extract of a cell trace [14]

All of there individual Tor cells are then stored within different files. The names of these files will have the following format for monitored pages `<page_id>-<instance_id>.cell` and `<page_id>.cell` for unmonitored pages.

There are several data sets readily available that have already been preprocessed. Some of the largest ones are the ones collected by Wang et al. [53], Greschbach et al. [14]. 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 samples of each website in Alexa's top $9,000$ and one sample for $909,000$ unmonitored sites [14]. Both of these datasets are definitely large enough to train our deep learning models.

Chapter 3

Threat Model and Problem Statement

In the following section, we further describe the threat model, from which we deduce a more detailed problem statement.

3.1 Threat Model

We consider an adversary, who wants to perform a website fingerprinting attack against Tor. Tor, specifically, since it has become one of the most widely used internet privacy tools. As specified in figure 2.2.1, the adversary is a *local eavesdropper*. Hence, the attacker passively collects encrypted web traffic between the client and the first Tor node, called the *entry guard*. This is achieved by either monitoring the link itself or relying on a compromised entry guard. Next, the adversary performs an analysis on that data to classify which specific web pages the client is visiting.

This analysis can be performed with several different goals in mind. The first one is to identify whether or not a user visits a web page from a set of monitored web pages. Thus the attack is essentially a *binary classification problem*, where you label a web page as *monitored* or *unmonitored*. Or the adversary might want to know which specific web pages a user visits within a set of monitored pages, which is a *multiclass classification problem*.

Within this adversary model we do make various assumptions. First of all, the adversary is not interested in blocking Tor traffic nor in modifying any traffic. Next, the adversary is able to replicate the conditions under which the user browses the internet such as download speeds, OS and TBB. On top of this, the adversary can also determine the beginning and the end of a user session on a web page and that the attacker has enough resources to collect traffic and train a deep learning model. Finally, we also make the adversary is unable to decrypt the traffic and thus only has access to metadata such as traffic direction, length and timing of packets.

3.2 Problem Statement

At the time of writing, most WF attacks use primitive machine learning techniques that require a fixed-length input. Even though a traffic trace consists of a variable-length sequence of packets. Therefore, these works often rely on a laborious, time-consuming process to extract fixed-length representations, or *fingerprints*. But there is no guarantee that these fingerprints are the most appropriate ones. On top of that, the previously mentioned process often requires domain-specific knowledge, making the attack even more difficult. Thus, here we investigate the use of automatic feature generation techniques to extract features automatically, without the need for any domain-specific knowledge.

Hence, our main contribution is the creation of a new deep-learning models, capable of learning fixed-length fingerprints from variable-length traces. This means that we will not be focusing on creating a new attack, but rather re-using existing attacks with these generated features. Next, we will contrast the performance of these different models and note which ones seem to be the most appropriate for the threat model described above.

Chapter 4

Attack Design

In this section, we describe the design of our automatic feature generation model, outline the attack strategy and explain certain design decisions. Most of this section will be split into two different sections. First we describe the feature generation process and then the overall attack.

4.1 Stacked Autoencoder

A stacked autoencoder takes a fixed-length input vector and tries to learn a function that compresses that vector. Thus if we were to use it for our fingerprinting extraction model, we will need to preprocess the variable-length traces into a fixed-length input. There are various different manners of doing this. One of the most naive ones is to find the longest length trace and pad all of the other traces up to that length. However, in Greschbach et al.'s dataset, this length is around $250,000$ [14], which means that our network will need to be incredibly deep to extract a short trace.

Instead, we pick the average length of the traces, which is around $3,000$ and cut or pad traces which are longer or shorter. On top of that, we deal with the fact that each packet is represented by a tuple, by just multiplying the time by the direction. Therefore, all of the outgoing packets are positive whilst the incoming ones will be negative.

Finally, since the output of the activation functions used is either between 0 and 1 or -1 and 1, we will scale the batches accordingly.

4.2 Sequence-to-Sequence Model

As described in section 2.3.4, a sequence-to-sequence model is able to learn how to construct a fixed-length representation from a variable-length sequence, which means that we will not have to perform as much preprocessing as for our autoencoder.

Although we already know the overall structure, there are still a couple of design decisions that had to be made specifically for our attack. These decisions are outlined in the following section.

One of the parameters which we will examine first is the *amount of hidden states* in the RNN cells. This number affects the amount of neurons for each layer with the cells. Hence, if for example the amount of hidden neurons is set to 100, the state with an LSTM cell is represented by vector of length 200 (since the state is represented by two vectors of length 100). The higher this number, the easier it should be to learn a representation, as the compression factor is lower. But we also need to consider the fact that the higher the amount of hidden neurons, the more variables the model needs to learn and thus the more complex the computations are.

Each value in a trace can be represented by a vector of length two (timestamp and direction), as seen in table 2.1. Therefore, if the amount of hidden cells is not equal to two, we will also need to *project* the input and output to the necessary dimensions.

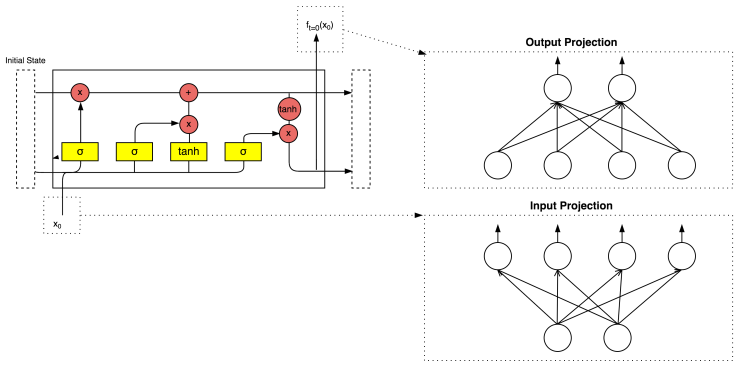


Figure 4.2.1: Example of projection within a LSTM cell with 4 hidden states.

Some of the traces can be particularly long and therefore the network needs to be unrolled to extreme lengths [14]. In fact, given memory constraints, this becomes a major problem. But it can be solved by cutting the traces after a couple seconds since it has been shown that the first part of a trace carries more information than the latter.

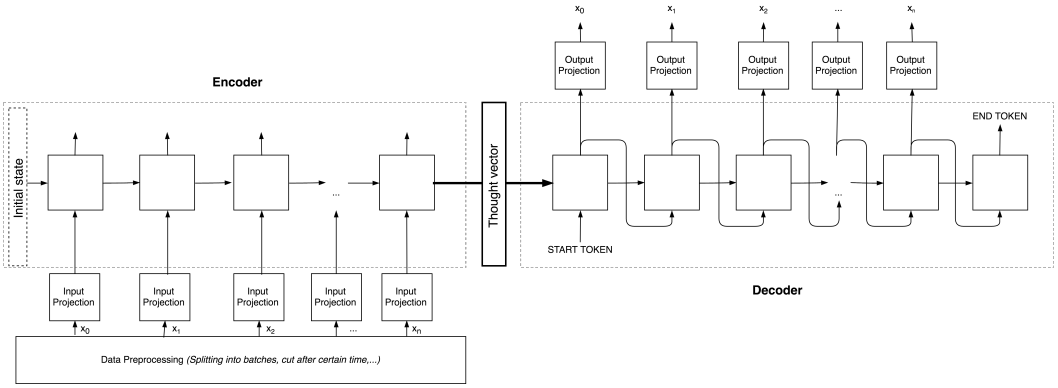


Figure 4.2.2: Overal structure of our sequence-to-sequence model.

Based on the above, we can construct our *computational graph* in Tensorflow as outlined in figure 4.2.2.

4.3 Attack Strategy

Here we consider an adversary that relies on deep learning to extract fingerprints for a website fingerprinting attack. This adversary can have two different goals in mind, as previously stated in section 3.1. The full attack, however, can be split up into four different stages, namely *data collection*, *fingerprint extraction training*, *classifier training* and *the attack*.

Data Collection

1. Choose web pages that the attacker wishes to monitor.
2. Collect traffic for a set of monitored and unmonitored sites.
3. Convert the raw TCP data into Tor cells.
4. Remove SENDMEs and other noise.

Fingerprint Extraction Training

5. Further process the data into batches and perform any other preprocessing required by the model such as cutting or padding the traces.
6. Prepare the fingerprint extraction model.
7. Train the fingerprint extraction model on a copy task for monitored and some unmonitored web pages.
8. Extract fingerprints from data by using the trained model.

Classifier Training

9. Given a classifier, train it using the extracted fingerprints.
10. Measure performance of classifier.

The Attack

11. Passively capture traffic from Tor users.
12. Pre-process the collected data.
13. Extract fingerprints using the trained fingerprint extraction model.
14. Classification via the trained classifier.

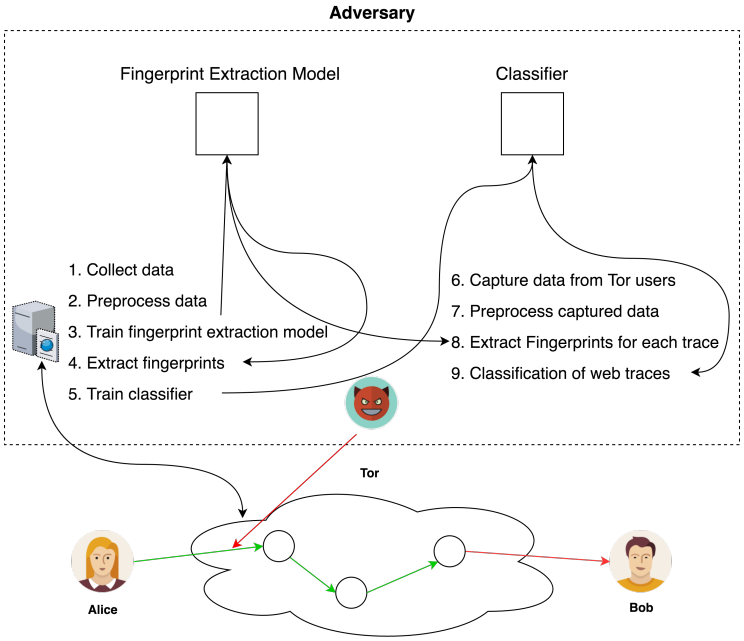


Figure 4.3.1: Attack strategy.

4.3.1 Data Collection

As previously mentioned, the data collection process first requires the adversary to choose a set of n websites to monitor. Next, the adversary crawls these pages a total of i iterations to ensure that a classifier has enough data to generalize. As suggested by Wang et al. when performing page loads, browser caching should be disabled since Tor does not allow caching to disk and therefore the browser cache should be cleared every time a web page is loaded[52]. After the collection, the TCP data is converted into Tor cells and probabilistic algorithms are used by Wang et al. [52] to remove SENDMEs. This data can be processed further, however we chose not to since the model should be able to learn how to perform this processing.

Most of our analysis will be done using the dataset provided by Greschbach et al. [14], which can be used for open-world analysis since it provides us with 100 samples for Alexa's top 9000 websites and one sample of one sample for $909,000$ unmonitored sites. For the rest of this paper, we will refer to this dataset as `GRESCHBACH`. Next, to see how our model performs on a dataset, whose data is recorded at a different time and under different circumstances, we will also be using the dataset provided by Wang et al. [53], which we will call `WANG14` [30]. This set is slightly smaller with 100 monitored websites with 90 instances each and 8400 unmonitored sites.

4.3.2 Fingerprint Extraction Training

In order to truly evaluate the model, we need to split the data up into a training and validation set. We do not train the model on any data in the validation set but instead use it to see how well the model performs on unseen data. For this split, we use a *stratified shuffle split*, meaning that we shuffle the data and then perform the split, whilst preserving the class distributions. On top of training the feature extractor with monitored pages, we also train it on unmonitored pages, as it needs to be able to extract features effectively from both sets.

During training and extracting the fingerprints, *mini-batch processing* will always be used. This will allow us to gain a performance boost and perhaps even have a faster convergence. When dividing the data up into these batches, we also need to determine how big they will be. The bigger they are, the larger the performance gain will be but the lower the accuracy might be. Additionally, the size of the batches also depend on the amount of available memory since we cannot have the VM run out of memory whilst training.

On top of determining the batch size, the individual models require different preprocessing steps and tuning of different parameters.

Stacked Autoencoder

As previously mentioned, after dividing the data up into batches, we either need to cut or pad the traces such that they all are of a fixed-length. Next, we know that the first layer needs to have the same amount of neurons than the length of the input. But after that, there are a variety of different architectures that need to be chosen. Some of which are outlined below:

- How many hidden layers we want. The more there are, the more complex the functions are that the model can learn but the harder it becomes to train the model.
- The amount of hidden neurons in each layer. This number should gradually decrease to the number of features we would like to extract.
- The activation function of the neurons. The most popular ones being *sigmoid*, *ReLu* and an *atan*.
- Whether or not to include batch normalization at every step.

Now that the model has been constructed, there are still several learning parameters that need to be tuned:

- The optimizer to use (*adam*, *gradient descent* or *RMSProp*) [43].
- Learning rate (γ) for the previously chosen optimizer.
- Amount of traces within a single mini-batch (b).
- Cost, or loss function (f) to minimize (*mean squared error* (MSE), *absolute loss* (AL) or *cross-entropy*) [43].

Sequence-to-Sequence Model

Each batch can either be presented in *batch-major* or *time-major* form. Although time-major is slightly more efficient [43], we opt for a batch-major form, since it makes the fingerprint extraction process easier. Next, after the data has been divided into mini-batches, we perform some further processing such as cutting the traces after several seconds. Finally, since all of the traces within a batch need to be of the same length, padding is performed as a final preprocessing step.

After collecting and fully preprocessing the data, the adversary can start to construct the sequence-to-sequence model. However, in order to do so, there are a variety of different architectures that need to be considered, some of which are outlined below:

- Which sort or RNN cells to use. This can either be a GRU or an LSTM cell. We could also potentially investigate the usefulness of multilayered RNN cells but we expect the performance gain to be very limited.
- Using a bidirectional encoder to ensure that the output at time t is not only affected by past information but also on future information.
- The amount of hidden states within a RNN cell, which affects the size of the fingerprints.

Now that the model has been constructed, the adversary still has to chose various learning parameters such as:

- The optimizer to use (*adam*, *gradient descent* or *RMSProp*) [43].
- Learning rate (γ) for the previously chosen optimizer.
- Amount of traces within a single mini-batch (b).
- Cost, or loss function (f) to minimize (*mean squared error* (MSE), *absolute loss* (AL) or *cross-entropy*) [43].
- After how much time the traces are cut.

After these parameters have been tuned, the computational graph can be constructed in Tensorflow and the model can be trained. When this training has been completed, fingerprints can finally be extracted for all the traces in the test set.

4.3.3 Classifier Training

When the adversary has extracted the fingerprints for websites within the test set, they need to train a classifier. Most works so far rely on some sort of *supervised machine learning* techniques such as *support vector classifiers* (SVC), *k-nearest neighbours* (kNN), *random forests* (RF) or *naive bayes* (NB) [31, 30, 53, 16, 15]. All of these algorithms rely on different techniques but an explanation of their inner workings is outside the scope of this paper. Instead, we will consider them as *black box models*. This means that all we know is that we can apply a `fit` function to the models, which causes them to learn how to classify the fingerprints and a `predict` function, which predicts the classes of given inputs.

To measure the performance of our black-box models, we use a similar technique as we did in the previous section. We split our test set up into two more sets, a *classifier training set* and a *classifier test set*. But since training a classifier, requires less time, we can use another technique, called *stratified k-fold validation*. Here we split our original test set up into k, mutually exclusive, folds. Next, one of the folds is chosen to be the classifier test set and all of the other folds form the classifier training set. This process is repeated for k iterations, where for each iteration, a different fold is chosen to be a test set. But again, we preserve the class distributions within all of the folds.

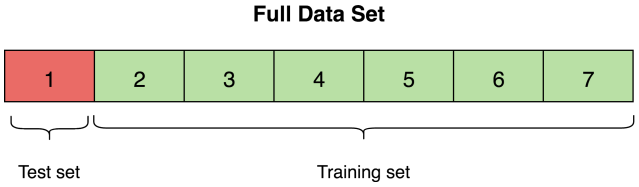


Figure 4.3.2: Example of one iterations in a k-fold validation ($k = 7$).

For each iteration of the validation process, several statistics can be recorded and then averaged over all iterations. The k-fold validation process ensures that every data point will be in the test set at least once and therefore giving us an accurate measure of these statistics.

Some of the performance measures that we will use in the evaluation stage are outlined below within the context of a WF attacks:

- **True Positive Rate (TPR)** is the probability that a monitored page is classified as the correct monitored page [16].
- **False Positive Rate (FPR)** is the probability that an unmonitored page is incorrectly classified as a monitored page [16]

- **Bayesian Detection Rate (BDR)** is the probability that a page corresponds to the correct monitored page, given that the classifier recognized it as that monitored page [16]. This can be calculated as follows:

$$\text{BDR} = \frac{TPR \times \Pr(M)}{TPR \times \Pr(M) + FPR \times \Pr(U)}$$

where

$$\Pr(M) = \frac{|\text{Monitored}|}{|\text{Total Pages}|}, \quad \Pr(U) = 1 - \Pr(M)$$

This measure essentially indicates the practical feasibility of the attack, as the adversary is mainly concerned with this specific measure [16].

- **Accuracy (A)** is the percentage of correctly classified instances. Although it can be used as a rough indicator, it will not be used in the final conclusions because of the *accuracy paradox*, which arises due to class imbalance.

- **F1-Score (F1)** measures the harmonic mean between precision and recall [38].

$$\begin{aligned} \text{F1} &= 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \\ &= \frac{2 \times TP}{2 \times TP + FP + FN} \end{aligned}$$

where

$$\text{recall} = \frac{TP}{TP + FN}, \quad \text{precision} = \frac{TP}{TP + FP}$$

This measure is particularly useful since it is not affected by class imbalance.

Rather than using another classifier, we could potentially add a *softmax layer* on top of our encoder, like V. Rimmer uses on top of her stacked autoencoder [37]. We could use this idea for both our autoencoder and the sequence-to-sequence model.

This process would involve first training the sequence-to-sequence model, then stacking the softmax layer on top of each unrolled cell in the encoder and using it for classification. What would be interesting about this approach is that the adversary can analyse how certain the classifier is, as it analyses packets more and more packets from the trace. However, this is outside the scope of this paper, as we are focusing on the feature extraction process.

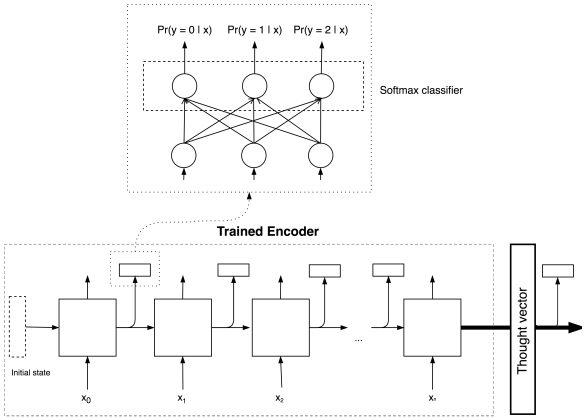


Figure 4.3.3: Example of encoder with softmax layer for 3 web pages.

4.3.4 The Attack

Finally, after the adversary has trained the required models, the real WF attack can start. First, the adversary starts capturing web traffic data between the user and the entry guard, as shown in figure 2.2.1. Next, the data is processed for the fingerprint extraction process, as described in section 4.3.2. After all the processing has finished, fingerprints are extracted using the previously trained model. Finally, those fingerprints are used as features for a classifier, which classifies the traffic into web pages.

The time between data collection for training and performing the WF has to be kept as small as possible since Juarez et al.'s experiments show that website's content changes greatly over time, therefore affecting the accuracy of the attack [21].

4.4 Code Structure

In the work, we will not be conducting the final stage of a WF attack. Instead, we will be reporting the results on the test sets. All of this is reflected in the overall structure of the code, which consists of four main components, as can be seen in figure 4.4.1.

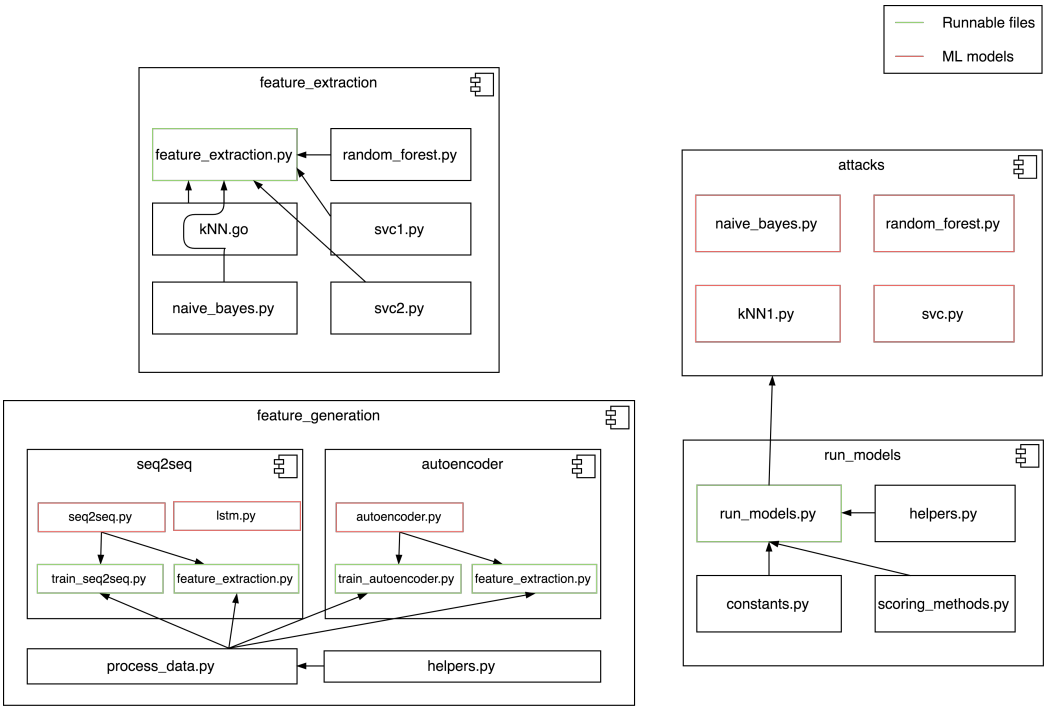


Figure 4.4.1: Diagram of how different components are related.

All of the data that is used within this work has already been preprocessed into Tor cells with SENDMEs removed, as described in section 4.3.1. The `feature_generation` module contains all of the code to further preprocess the data, train the fingerprint extraction models and to extract the fingerprints using the respective model.

The classifiers that will be used for the attacks are defined in the `attacks` module. We tried to pick a variety of existing models to measure how our extracted fingerprints work on different classifiers. The logic to actually test the attacks is defined in the `run_models` module, which also does some data preprocessing, defines the logic for the stratified k-fold validation and the different scoring methods.

Finally, there is also the `feature_extraction` module, whose use will be explained later.

As can be seen, all of the code is written in Python, due to the wide availability of machine learning tools. Except for the `kNN.go` file, in the `feature_extraction` module, which is written in *Golang* to gain a performance boost [36].

Chapter 5

Evaluation and Testing

In the following section, we outline how we will be evaluating and testing the suggested models. Next, we will perform that evaluation and present the final results.

5.1 Experimental Setup

Since our deep model requires a large amount of computation, we like to make use of parallelization. Hence, all of our experiments that involve deep learning will be run on an *Amazon EC2 p2.xlarge* instance. This VM has a *NVIDIA K80 GPU* with 12 GiB of GPU memory. All of the instances used were setup with both *CUDA 8* and *cuDNN v5.1* [43, 27].

The rest of the experiments are run on a 2016 MacBook Pro, with a 2.9GHz Intel Core i5 and 8GB of RAM, running MacOS 10.12. To make sure that the same Python environment is used on both these machines, we consistently use *Python 3.6* and a *virtual environment* for the python dependencies.

As previously mentioned, the main dataset used is `GRESCHBACH` but we will also be using some of the data in the `WANG14` dataset to see how the model performs on data that was recorded under different circumstances. For both these datasets, we will only be using the preprocessed Tor cells and not the raw TCP traffic data.

Finally, in all of the experiments that are be conducted below, we only consider an *open-world scenario*. This means that the test set will contain both monitored and unmonitored pages that the fingerprint extraction models and the classifier have never seen before. For this to work, we train the models on a large set of monitored web pages but also on a small percentage of unmonitored web pages such the classifiers can distinguish between both.

5.2 Evaluation Techniques

There are several different manners in which we can evaluate the feature selection process. First of all, we could analyse how the model performs on unseen traces, as it learns. If the difference between both the training error and the error on an unseen instance increases, the model will clearly be overfitting.

However, this data only show us how well the model is at reproducing the trace from a fingerprint but not how well the fingerprints perform in a WF attack. For this we need to train a classifier and see how well it performs by using the metrics described in section 4.3.3.

To be able to compare these fingerprints with hand-picked ones, we could train the classifiers with the hand-picked features and with the automatically generated ones. These hand-picked features are often chosen by experts and after a careful analysis. Hence, if the classifier with our fingerprints were to get similar results or even outperform the classifiers with the hand-picked features, we know that the fingerprint extraction model has been successful. For these results to be accurate, we do not change any (hyper)parameters within the classifiers. Thus everything, except for the features, remains the same.

For the classifiers, we pick a small set of four existing models. We aim to pick models

that have had an influence on the WF field whilst also having a variety of different classifiers. This set includes the two *support vector classifiers* (SVCs) used by Panchenko et al. [31, 30], the k-fingerprinting attack, which relies on a *random forest* (RF) used by Hayes et al. [16] and finally the *k-nearest neighbours* (kNN) classifier used by Wang et al. [53].

For all of these models, we extract the exact same features as outlined in the respective papers. The code for this feature extraction process can be found in the `feature_extraction` module.

We also aim to use the exact same hyperparameters described in the respective papers. More specifically:

- **SVC** [31] - a *radial basis function* (RBF) kernel with $C = 2^{17}$ and $\gamma = 2^{-19}$.
- **SVC** [30] - uses the same hyperparameters as in the previous SVC but with different features.
- **RF** [16] - a good accuracy/time tradeoff when $k = 3$ and $num\_trees = 20$.
- **kNN** [53] - also has a good accuracy/time tradoff when $k = 2$ and $k_{reco} = 5$.

We do need to note that these parameters have been specifically tuned for the hand-picked features and not for our fingerprints.

5.3 Evaluation

As mentioned in section 4.3.2, for both deep learning models. we need to make a couple design decisions regarding different architectures and learning parameters. We perform several experiments here to see which ones are the most appropriate.

5.3.1 Stacked Autoencoder

Learning Parameter Tuning

First, we start by varying the mini-batch sizes from 20 to 600 in steps of 20 for a simple model with an input layer of 3000 cells, and two hidden layers with 1600 and 200 neurons respectively. The higher the batch size, the longer it takes before making a weight update and the lower the value, the more noise in the training data. We notice that a total batch size of 400 seems to be a good tradeoff.

Next, a *mean squared error* (MSE) loss function with an *RMSProp* optimizer and a 0.01 learning rate and seem to yield the most appropriate results. Finally, we also use batch normalization for all experiments since it allows the model for faster convergence.

Architecture Tuning

Our experiments show that a *sigmoid* activation function continuously results in better learning with a variety of different hidden layers with different sizes.

The amount of hidden layers is a slightly more difficult decision. Since we want the simplest network possible that is able to learn a representation Hence, we experiment with networks with a total of 1 up to 3 hidden layers. For each of these, the input layer will consist of 3000 nodes and we will attempt to extract 200 features, which means that the sizes of the hidden layers will gradually decrease to 200 neurons.

Figure 5.3.1 already shows us that a network with two hidden layers provides a good complexity/training error tradeoff. Now that we know the depth of the network, we also need to consider changing the size of the final hidden layer since it represents the amount

Figure 5.3.1: Learning curves for changing the depth of the stacked autoencoder

of features that will be extracted. The more features we introduce, the more time and data we require to learn the classification task. Whilst if the amount of features is too low, the classifiers might not be able to learn how to effectively classify any of the web pages. Hence, we base the size of the final state on the amount of features used in previous WF attacks.

Model	**Features**
SVC [31]	305
SVC [30]	104
RF [16]	150
kNN [53]	3737

Table 5.1: Amount of features for existing attacks.

Based on table 5.1, we vary the amount of features between 100 and 300 in steps of 50. From figure 5.3.2, we determine that around 200 nodes provides us with the best tradeoff. Therefore, throughout the rest of the report when we refer to a stacked autoencoder, its architecture consists of 3 layers with 3000, 1600 and 200 nodes each.

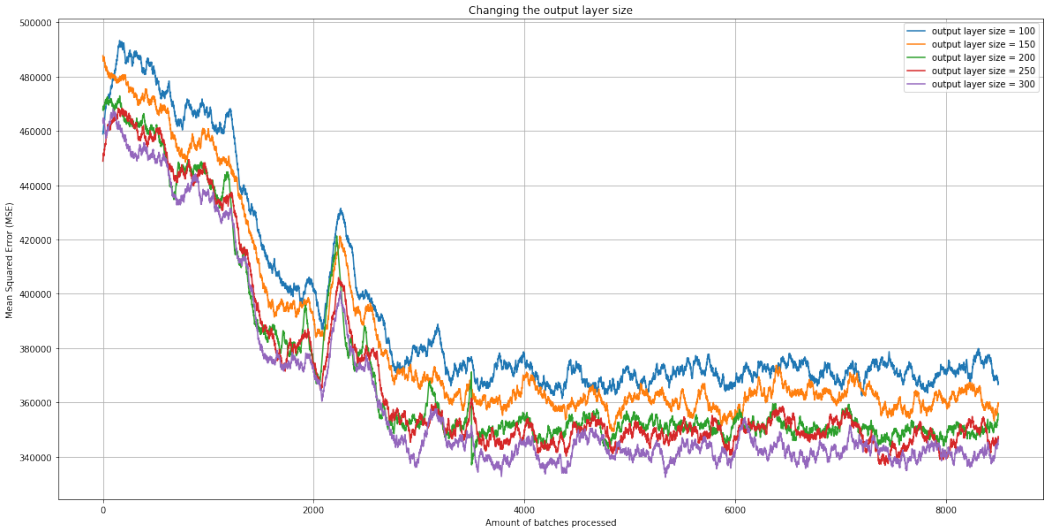


Figure 5.3.2: Learning curves for changing the size of the middle layer in the stacked autoencoder

5.3.2 Sequence-to-Sequence Model

Learning Parameter Tuning

We try to aim to get the appropriate values for the learning parameters within a simple encoder and decoder with LSTM cells and 120 hidden states.

After experimentation, the maximum batch size that our EC2 instance could handle memory-wise is around 400. Thus through the rest of the report we will use a mini-batch size of 400.

Next, we vary the learning rate γ from 0.01 to 0.000001 with various optimizers (*adam*, *gradient descent* or *RMSProp*) and loss functions (*mean squared error (MSE)* or *absolute loss* (AL)). After trying a wide variety of different permutations, an *adam optimizer* continuously demonstrated better results. We already expected this since adam optimizers are computationally efficient, require relatively little amount of memory and tend to perform very well with problems that have a large amount of parameters [23], which is ideal since our network can be unrolled to a very large lengths.

Next, we also note that the best quality of data compression was achieved with a *MSE loss function* and a learning rate of 0.000002. Hence, we set $\lambda = 0.000002$, $b = 400$ and use an adam optimizer with a MSE loss function for the rest of our experiments.

Since some of the traces are relatively long, it might be worth cutting the them after a certain amount of time. However, to compare after which time to cut the trace, we cannot simply base our analysis on the learning curve because the shorter the trace, the smaller the error is likely to be. Therefore, we will cut the traces after 2, 6 and 10 seconds, use these values to train a sequence-to-sequence model and train binary classifiers on the extracted fingerprints. Next, we can compare the performance of these classifiers to analyse how much information each part of the trace carries.

Figure 5.3.3 shows us that majority of the information is in fact carried in the first couple of seconds of the trace. Hence, for the rest of our experiments we will be cutting the traces after 10 seconds.

Finally, we also use batch normalization for all experiments since it allows the model for faster convergence.

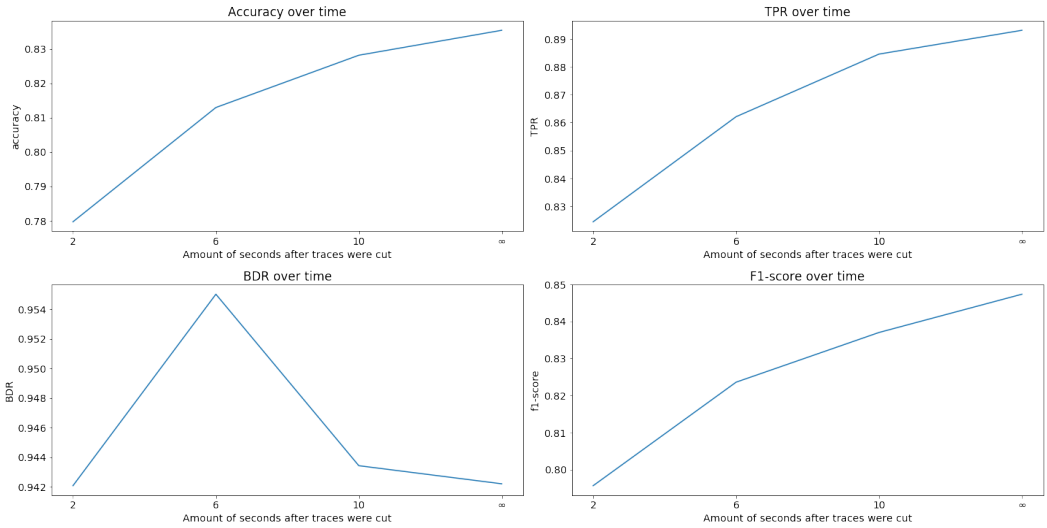


Figure 5.3.3: Average performance measures of all classifiers after cutting traces.

Architecture Tuning

Now that we have made a decision on which learning parameters to use, we can start changing the architecture of the sequence-to-sequence model to see which ones yield the best results.

Hidden States

We first start by examining the amount of hidden states in the network. These directly affect the size of the fingerprints that will be extracted. In fact, the amount of features extracted is exactly double the amount of hidden states. Thus, based on table 5.1, we vary the amount of hidden states between 60 to 140 in steps of 20 to see which ones yield the most appropriate results.

For these experiments we train a sequence-to-sequence model with a unidirectional encoder, LSTM cells and without cutting or reversing the traces. The training data consists $120,000$ monitored and unmonitored web pages, which are shuffled to avoid overfitting on any specific web page. We only train the model for one epoch, as we seem to have enough data for the model to converge within that epoch. Hence, every sample that the model sees in the figure below is one that it has never seen before. So we can easily determine that the model is not overfitting.

Figure 5.3.4 clearly shows us that the smaller the amount of hidden states, the faster the network seems to learn the reconstruction task. On the other hand, the higher the amount of states, the lower the final error seems to be. Since we aim to compromise between computational complexity and the time it takes to train the model, around 100 hidden states seems to be the most appropriate.

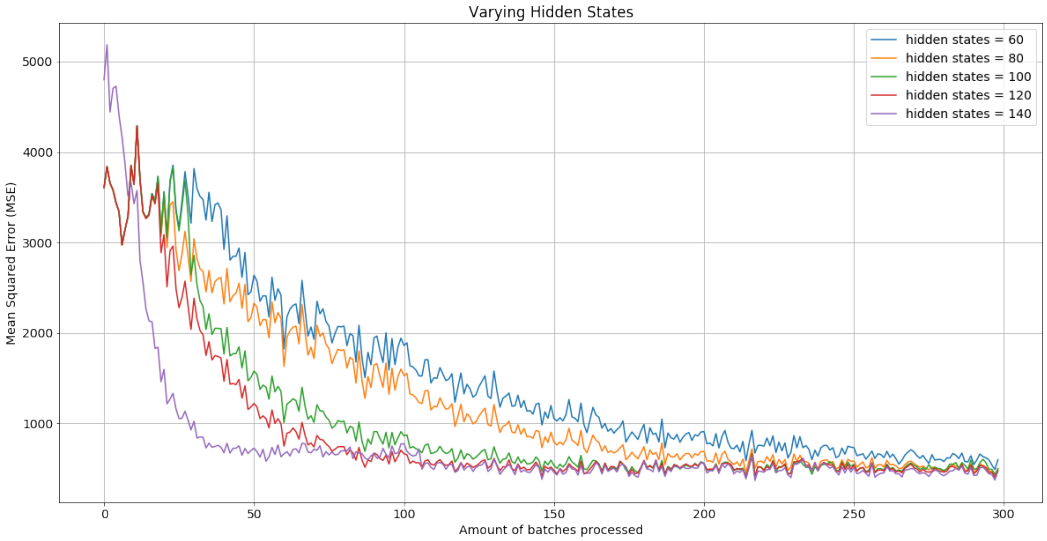


Figure 5.3.4: MSE over the amount of traces processed for varying hidden states.

Bidirectional

For these experiments, we consider a smaller range of hidden state values from 80 to 120 in steps of 20. Again, for all of these we will be using LSTM cells without cutting or reversing the traces, with all of the learning parameters described above and the exact same training set used in the previous experiment.

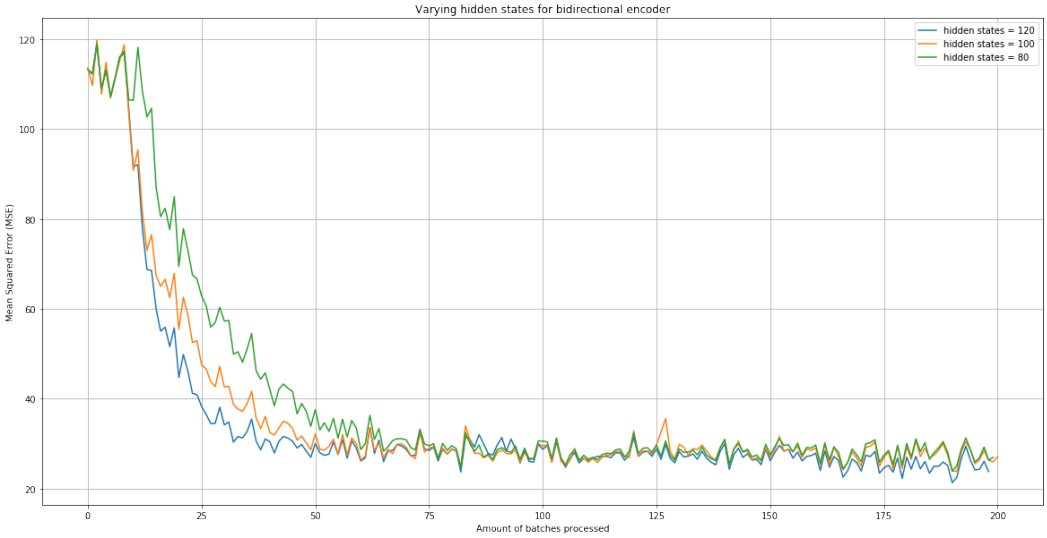


Figure 5.3.5: MSE over the amount of traces processed for varying hidden states for a bidirectional encoder.

As can be seen in figure 5.3.5, around 80 hidden states seems to provide the best complexity/error tradeoff.

LSTM or GRU Cells

Here, we train a sequence-to-sequence model with both a unidirectional and bidirectional encoder. These will both have GRU cells with 100 and 80 hidden states respectively. Furthermore, we recreate the exact same training conditions as in the previous experiments.

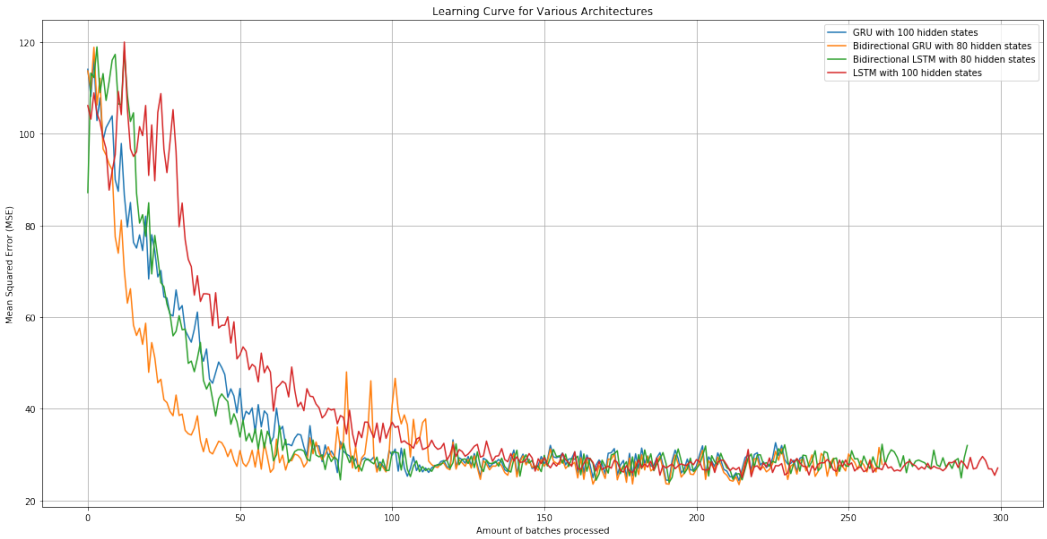


Figure 5.3.6: Learning curves for different cell types.

As can be seen in figure 5.3.6, all of the different architectures converge to a similar value. Although GRU cells seem to converge faster, they are also slightly more unstable especially around batch 80 to 100. The most stable model seems to be the bidirectional encoder with LSTM cells. Although it has more parameters than the unidirectional encoder models, it takes a very similar amount of time to train it. Hence, throughout the rest of the report when we refer to a sequence-to-sequence model, its architecture consists of a bidirectional encoder with 80 hidden states.

5.3.3 Classifier Performance

We have previously analysed the models' performance based on how well it copies the original input from a fingerprint. But to examine how well our model performs during a real WF attack, we compare its performance on different existing classifiers with hand-picked features. This means that we choose a set of existing WF attacks and recreate them. Next we run the exact same attack but with both the hand-picked and automatically generated features.

Note that our results might be slightly lower than in their respective papers since we do not aim to recreate the full attack. Rather than optimizing different hyperparameters, we aim to use these classifiers and the hand-picked features as an indicator as to how well the fingerprint extraction models perform.

We expect that the automatically generated features will perform worse than the hand-picked ones due to the complexity of the task. However, we still hope to show that it is in fact possible to automate this feature selection process till a certain extent.

As mentioned in section 3.1, there are two main threat models that we need to consider. The first one is a binary classification task, where the adversary wants to see whether or not a user is visiting any webpages within a given set. Whilst the other threat model involves the adversary having a set of monitored pages, and wants to know which specific pages the user is visiting in that set. Hence, it is a multiclass classification problem.

Although there are different techniques for evaluating binary and multiclass classification models, we will only use the scoring statistics outlined in section 4.3.3. This allows us for easy comparisons between the different threat models. We do expect that the binary classification models will perform better than the multiclass ones due to the smaller amount of options available.

Aforementioned, we have already selected a total of four different existing attacks. We will refer to the first SVC attack by Panchenko et al. [31] as `svc1` and the second one [30] as `svc2`. Whilst we refer the k-fingerprinting attack by Hayes et al. [16] as `RF` and finally the attack by Wang et al. [53] as `kNN`.

Binary Classification

We first start by analysing the simplest threat model, namely binary classification. For all of the models below, we aim to extract the exact same hand-picked features as were described in the respective papers to the best of our knowledge.

For training these models, we use an extract from the `GRESCHBACH` dataset with a total of 100 monitored web pages with 70 instances each and 5000 unmonitored web pages. We then split this set into a training and validation set using a stratified split. The training set will contain 90% of the monitored web pages whilst we vary the amount of unmonitored pages to see how the models perform.

After the set is split up into a training and validation set, we perform a *stratified k-fold validation* with $k = 3$ on the training set Then finally we train the classifiers on all of the training data and evaluate them on the test set.

The results for the k-fold validation on the training set for the hand-picked features are outlined in table 5.2. Here, we used a total of 10% of the unmonitored data for training. As expected, the results with a small amount of unmonitored data is relatively high.

Model	Accuracy	BDR	TPR	FPR	F1
`svc1`	0.91 ± 0.003	0.99 ± 0.001	0.97 ± 0.001	0.07 ± 0.002	0.90 ± 0.005
`svc2`	0.91 ± 0.008	0.99 ± 0.001	0.95 ± 0.003	0.06 ± 0.004	0.90 ± 0.008
`RF`	0.93 ± 0.003	0.99 ± 0.001	0.97 ± 0.006	0.05 ± 0.003	0.92 ± 0.005
`kNN`	0.88 ± 0.007	0.99 ± 0.003	0.97 ± 0.004	0.10 ± 0.002	0.94 ± 0.004

Table 5.2: Performance statistics hand-picked features on a binary classification task with k-fold validation whilst training on 10% of the unmonitored pages.

Next, we will be analyzing the performance of these classifiers with the automatically generated features. We do note that from here on we refer to `svc1` and `svc2` as `svc` since they both have the same hyperparameters but were trained on different hand-picked features.

Model	Accuracy	BDR	TPR	FPR	F1
`svc`	0.92 ± 0.001	0.99 ± 0.001	0.98 ± 0.001	0.07 ± 0.002	0.89 ± 0.003
`RF`	0.77 ± 0.012	0.76 ± 0.004	0.87 ± 0.009	0.15 ± 0.004	0.86 ± 0.007
`kNN`	0.74 ± 0.010	0.73 ± 0.009	0.85 ± 0.004	0.18 ± 0.007	0.84 ± 0.009

Table 5.3: Performance statistics autoencoder features on a binary classification task with k-fold validation whilst training on 10% of the unmonitored pages.

Both table 5.3 and 5.4 show that the performance on a small amount of unmonitored pages is very similar with the `svc` model but slightly lower for both the `RF` and `kNN` attacks.

Model	Accuracy	BDR	TPR	FPR	F1
svc	0.93 ± 0.001	0.99 ± 0.001	0.99 ± 0.001	0.06 ± 0.003	0.90 ± 0.002
RF	0.86 ± 0.004	0.99 ± 0.001	0.83 ± 0.003	0.07 ± 0.005	0.88 ± 0.003
kNN	0.81 ± 0.008	0.95 ± 0.007	0.97 ± 0.007	0.14 ± 0.012	0.89 ± 0.009

Table 5.4: Performance statistics sequence-to-sequence features on a binary classification task with k-fold validation whilst training on 10% of the unmonitored pages.

Now we will the performance when training the classifiers on the full training set and evaluating them on the validation set, whilst changing the amount of unmonitored pages we train the model on. Clearly, figure 5.3.7 shows us that the models suffer if we introduce a large amount of unmonitored pages in the test set. But the more unmonitored instances we train on, the better the classifiers seem to perform.

Additionally, figure 5.3.7 also shows that the RF classifier seems to perform best whilst training in on both the hand-picked and automatically generated features. Next, we also note that the hand-picked features currently still get the best overall performance, followed by the sequence-to-sequence features and the autoencoder features.

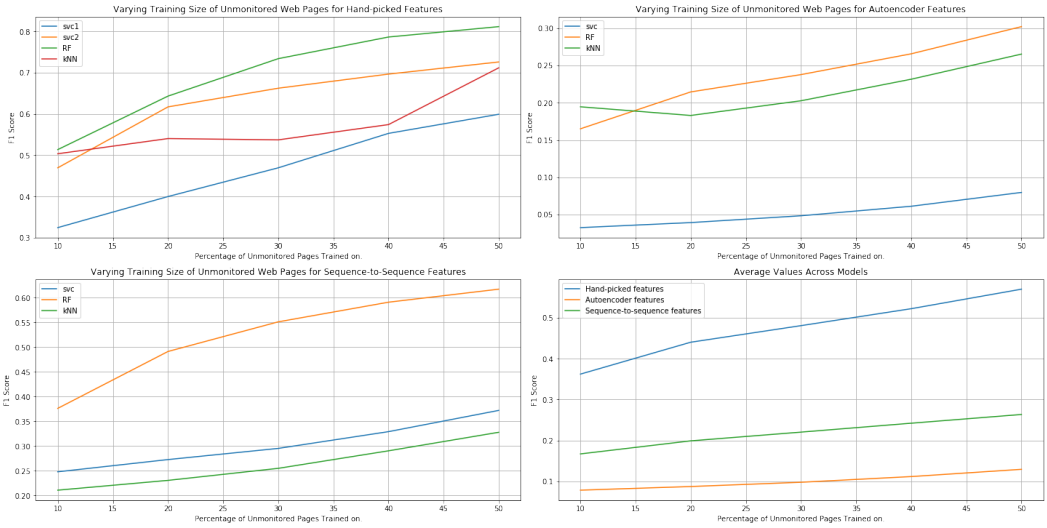


Figure 5.3.7: Varying the amount of unmonitored pages trained on for different features.

Multiclass Classification

The multiclass classification scenario is slightly more complex due to the larger array of options. Hence, we also expect considerably lower results, especially on the test set with a large amount of unmonitored pages.

Model	Accuracy	BDR	TPR	FPR	F1
svc1	0.57 ± 0.013	0.99 ± 0.001	0.59 ± 0.014	0.08 ± 0.004	0.70 ± 0.012
svc2	0.59 ± 0.007	0.99 ± 0.001	0.61 ± 0.007	0.07 ± 0.007	0.72 ± 0.009
RF	0.59 ± 0.011	0.99 ± 0.001	0.58 ± 0.011	0.02 ± 0.004	0.72 ± 0.012
kNN	0.55 ± 0.015	0.92 ± 0.006	0.55 ± 0.008	0.09 ± 0.005	0.69 ± 0.013

Table 5.5: Performance statistics hand-picked features on a multiclass classification task with k-fold validation whilst training on 10% of the unmonitored pages.

Table 5.5 shows that the performance does indeed drop on the multiclass classification task. On the other hand, both table 5.6 and 5.7 show that the performance for automatically generated features is even lower than the hand-picked ones.

Model	Accuracy	BDR	TPR	FPR	F1
svc	0.22 ± 0.003	0.54 ± 0.002	0.17 ± 0.002	0.16 ± 0.004	0.29 ± 0.004
RF	0.25 ± 0.009	0.62 ± 0.003	0.18 ± 0.008	0.13 ± 0.007	0.30 ± 0.009
kNN	0.20 ± 0.015	0.48 ± 0.006	0.17 ± 0.006	0.20 ± 0.007	0.28 ± 0.011

Table 5.6: Performance statistics autoencoder features on a multiclass classification task with k-fold validation whilst training on 10% of the unmonitored pages.

Model	Accuracy	BDR	TPR	FPR	F1
svc	0.35 ± 0.004	0.69 ± 0.008	0.24 ± 0.003	0.13 ± 0.014	0.37 ± 0.002
RF	0.39 ± 0.005	0.83 ± 0.004	0.27 ± 0.008	0.07 ± 0.014	0.42 ± 0.006
kNN	0.31 ± 0.011	0.56 ± 0.004	0.22 ± 0.09	0.20 ± 0.003	0.33 ± 0.009

Table 5.7: Performance statistics sequence-to-sequence features on a multiclass classification task with k-fold validation whilst training on 10% of the unmonitored pages.

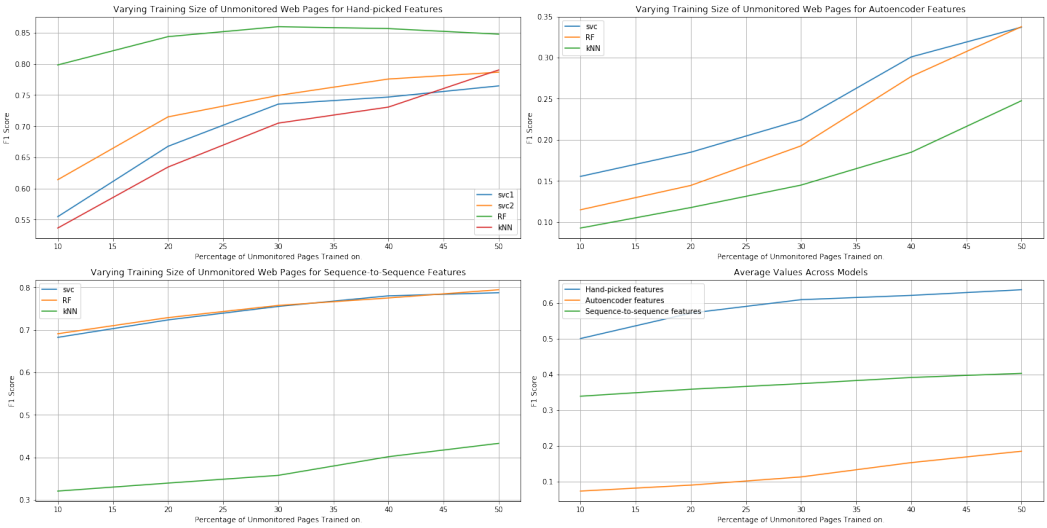


Figure 5.3.8: Varying the amount of unmonitored pages trained on for different features.

But the behavior of the classifiers gets even more interesting when training on a larger amount of unmonitored pages. Figure 5.3.8 shows that the stacked autoencoder has a very similar performance as on the binary classification task. Whilst the sequence-to-sequence model performs almost as well as the hand-picked features. In fact, the sequence-to-sequence features even perform better on the svc.

Different Circumstances

Beside analysing how the fingerprint extraction models perform on data within the same dataset, it would be interesting to examine how it performs on data recorded under different circumstances. It has already been shown that the performance of the classifiers is greatly impacted by the network, time and the TBB version. But that doesn't necessarily mean that our fingerprint extraction model is impacted similarly.

If the deep learning models are not impacted by these flaws, an adversary would only need to train the fingerprint extraction model once and then it could continue to use it and only retrain the classifiers, like some sort of *transfer learning*

To test this premise, we use the models that we previously trained on the same 120,000 web pages within the `GRESCHBACH` dataset. More specifically, a LSTM bidirectional encoder with 80 hidden states and a stacked autoencoder with sizes of the hidden layers being 3000, 1600 and 200. Next, we extract the fingerprints from the Tor cells within the `WANG14` dataset using this model, train a set of classifiers on these fingerprints using k-fold validation and note down their performance.

For the following experiments, we train the classifiers using $k = 3$ and for both datasets, we pick a total 100 monitored web pages with 70 instances each and 5000 unmonitored web pages.

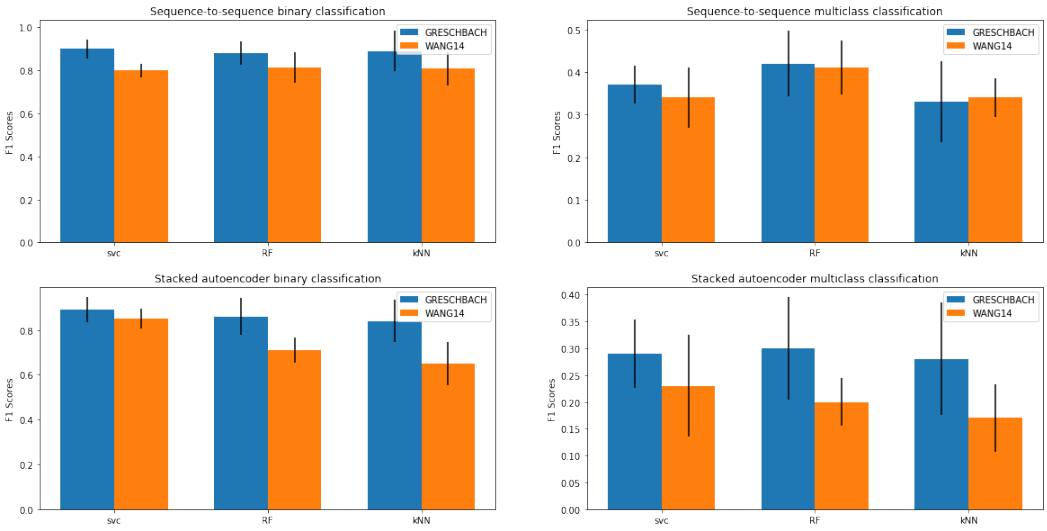


Figure 5.3.9: Classifier performance on the WANG14 dataset with automatically extracted features.

Figure 5.3.9 shows us that the performance drops slightly when extracting features on the `WANG14` dataset. Especially with the stacked autoencoder. However, the sequence-to-sequence model seems to achieve similar F1 scores on both the `GRESCHBACH` and `WANG14` dataset.

5.4 Unit Tests

On top of evaluating the results, we also needed to ensure that the code behaves as we expect it to. For this we use unit tests. Some bits of the code, such as the Tensorflow models are difficult to test but we can still test all of the preprocessing to see the correct values are produced. For this we use Python's standard `unittest` module [49]. The reason for this choice is that it is flexible and the standard Python unit testing framework, which means it is commonly used.

On top of unit tests, *Travis* was also used [48]. Travis is a popular tool, that has an easy integration with Github, for continuous integration. Therefore, every time a commit is pushed to the remote repository, Travis runs all of the tests automatically. If one of the tests fails, Travis then automatically notifies all the contributors.

Finally, to check if our tests cover our entire codebase, *codecov* is used [10]. This tool automatically checks how much of the codebase all of the unit tests cover. At the time

of writing, the coverage is 93%. The bits that aren't covered by unit tests, such as the Tensorflow implementation of the deep learning models, have been carefully examined to see if they behave as expected by using the Tensorflow debugger [43].

Chapter 6

Conclusion

The aim of our research was to apply deep learning techniques to a WF environment to automatically extract fingerprints from a variable-length trace. We do in fact show that this is possible by introducing a novel approach with two deep learning models, namely a *stacked autoencoder* and a *sequence-to-sequence model*.

The attack works in three main stages. First, all of the traffic is collected and preprocessed into Tor cells. Next, any of the previously mentioned deep learning models attempts to learn underlying patterns in the traffic traces and uses these to extract fingerprints through an unsupervised process. Finally, the extracted fingerprints are used to classify the trace as certain web pages.

Similarly, we also introduce a novel technique for evaluating the performance of such fingerprint generation models, which involves comparing them to a human benchmark. We achieve this by training existing classifiers on both hand-picked features and the automatically generated features and compare their performance. This allows us to see how well the deep learning models perform, compared to experts who have done thorough feature analysis.

During all of the performed experiments, we focused on an open-world setting with a local passive adversary against Tor. We show that for our best setup, we manage to achieve a 93% accuracy in a binary classification task and a 39% accuracy in the multiclass classification task. This is comparable to hand-picked features since they attain a maximum of 93% and 59% respectively. In fact, we even observe that our generated features seem to perform better than certain hand-picked features, given that the classifier is trained on a large amount of unmonitored pages.

We also discovered that a sequence-to-sequence model continuously seemed to perform better than a stacked autoencoder within all the threat models that we examined. This is most likely due to the fact that the autoencoder assumes that all of its inputs are independent of each other, which is not the case in our specific scenario. However, the problem still remains that some traces can be extremely long, which results in a slow training of our model. In fact, it took an average of 8 hours to train the sequence-to-sequence model, which is considerably slower than current state-of-the-art attacks.

However, we have showed that once the deep learning model is trained, it can be used to extract fingerprints from traces that were recorded under different conditions. Hence, the model would only need to be trained once on a large variety of data and it can then be used for a long period of time, without the need of retraining the model, unlike existing classifiers

On top of evaluating the results, we also made various observations about the traffic data. For instance, we note that majority of the information is carried within the first couple seconds of the trace and that most traces can be represented using vectors of size 200.

Furthermore we do note that our attack has been based on several assumptions. For instance, we assume that the adversary knows where the trace of a single page starts and ends, that the adversary can recreate the exact same conditions such as internet speed and TBB that the user uses to browse the web and finally that the content of web pages does not change. Although the exact same assumptions have been made in previous WF works, we do note that some of these are not realistic and therefore might have a large impact on the scoring statistics if the attack were to be used in a real-life scenario. Equally important is the impact that false negatives can have on the attack, as outlined by M. Perry [33].

In conclusion, our research does not improve the results of existing works, but it does expose the possibility to automate the fingerprint extraction procedure. Until now, almost all attacks have relied on a manual feature extraction process that require expertise in the WF domain. However, we show that this time-consuming process can be automated. Although currently the performance of our automatically generated features is not as high as the hand-picked ones, given enough data the correct deep learning model, an adversary could potentially perform a WF without the need for any domain-specific expertise.

6.1 Future Work

This work shows that the WF attacks currently still seem to perform better with hand-picked features rather than automatically generated ones but there is still much room for future improvements. Here we consider several different manners how we could improve or extend this work. Although we definitely will not cover all the different possible extensions, we try to list the most interesting ones.

As previously mentioned in section 4.3.3, we could add a *softmax layer* on top of the encoder in a trained sequence-to-sequence model. Not only would this allow us to perform the classification with the sequence-to-sequence model, but it would also allow us to analyse how different evidence affects the classification. You would technically only need one softmax layer, after the fingerprint has been extracted. But having one after every cell, allows us to see how different packets change the prediction of our model. This could then be used as a tool for analysis which features the model actually extracts.

There have also been a variety of different defenses, some of which have been outlined in section 2.2.3. Some works have examined the the effectiveness of their attack, when these defenses were used [16, 53]. It would be interesting to see if the deep learning models might still be able to effectively extract fingerprints, even with these defenses deployed. This could include both training the model on data where the defense was deployed or training it on normal data and analysing whether it can still extract the fingerprints if the defense is deployed during the attack stage.

Juarez et al. have already shown that WF attacks suffer from the rapid changing nature of the content of web pages [21]. Thus on top of analysing how defenses impact the attack, we could also potentially analyse how the performance of the fingerprint extraction process is affected over time. We have already shown that the models are still successful when extracting fingerprints from other datasets. However, this is not fully show that the models are not affected by content changes within web pages. This could be fully examined by collecting our own data over a period of time and see how the performance of a trained model changes. If the performance is not affected, we could save a large amount of time retraining the fingerprint extractor.

We could also potentially research the possibility that training our model with data collected over time and under different circumstances would also make the model more robust. Since technically, the more different training instances it sees, the better it should get at identifying features. Additionally, we could also investigate how well the models perform when collecting features when given more realistic user behavior. Hence, rather than visiting one page and waiting a certain amount of time before loading the next one, the data can be more realistic such as where the user has multiple tabs open at the same time.

On top of training the model with more realistic browsing data, we could also evaluate its performance for *Tor hidden services*. This is a protocol for Tor users to hide their location while offering a service such as hosting a website [46]. There is already evidence that these services can be classifier using a WF attack [16] but it would be interesting to see how our models would perform on this data.

Rather than extending this work by using more of different kinds of data, we could also improve the deep learning models. Currently, one of the main weaknesses is that the traces can be very long, which in turn makes our model very deep. We solved this issue here by cutting the traces after a certain amount of time since most the first part of the trace carries more information than the latter. However, this might not be the ideal solution. There might be another solution or perhaps even another model that does not have this weakness but still manages to map variable-length sequences into a fixed-length representation.

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] *Code Coverage*. URL: https://codecov.io/.

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

[23] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

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

[25] 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.

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

[27] *NVIDIA cuDNN*. Mar. 2017. URL: https://developer.nvidia.com/cudnn.

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

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

[30] 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.

[31] 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.

[32] *PEP 8 - Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/.

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

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

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

[36] Pylls. *Go kNN*. Jan. 2017. URL: https://github.com/pylls/go-knn.

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

[38] *Scikit-learn: Documentation.* URL: http://scikit-learn.org/stable/.

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

[40] 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.

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

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

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

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

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

[46] *Tor Hidden Services.* URL: https://www.torproject.org/docs/hidden-services.html.en.

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

[48] *Travis - Continuous Integration.* URL: https://travis-ci.com/.

[49] *unittest - unit testing framework.* URL: https://docs.python.org/3/library/unittest.html.

[50] Gregory Valiant. “Learning polynomials with neural networks”. In: (2014).

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

Appendix A

System Manual

Here, we outline the technical details of the project such that the development of the system could be continued by a third party.

First, we will look at the tools required to run the system then we will look at the overall system and how specific modules connect together. Finally, we explain the specific components and outline how you can contribute to this project.

All of the code is hosted in a private repository on Github. If given access, it can be cloned from the following url (https://github.com/AxelGoetz/website-fingerprinting.git).

1 Tools Required and Installation Instructions

Most of the project has been written in Python, more specifically, everything has been designed and tested with version 3.6. Our Travis build has also been configured to run all of the tests with Python 3.5, which means that we still get notified if anything breaks under a specific Python version. Some of the code has also been written in Golang, those parts have been adapted from an implementation of Wang et al.'s kNN attack for performance reasons [53, 36].

Next, after the correct version of Python and Golang are installed, we also require a *virtual environment* to manage all of our python dependencies. If you have `pip` on your system, this can easily installed as follows:

```
pip install virtualenv
```

After the virtual environment is installed, go to the project directory and run the following commands to first create the environment and then activate it.

```
virtualenv venv
source venv/bin/activate
```

At any time, you can go out of the virtual environment by running `deactivate`. After you are in the environment, we need to install a list of dependencies, which can be found in the `requirements.txt` file. This can be done by running:

```
pip install -r requirements.txt
```

This will install a list of dependencies but the main ones are *Tensorflow v1.1*, *numpy*, *sklearn* and *scipy*. There are a couple others but they are not used for major components.

If you plan to use the GPU support on Tensorflow, there are a couple more steps that need to be taken such as installing *CUDA 8* and *cuDNN v5.1*. These instructions can be found on the Tensorflow site[1]. Our project relied on an Amazon EC2 p2.xlarge with one NVIDIA K80 GPU but the code can be run on any GPU card with CUDA compute capability 3.0 or higher [43].

[1] https://www.tensorflow.org/install

In order to run the experiments, the datasets will have to be downloaded next. The main dataset used is `GRESCHBACH`[2] but we also use `WANG14`[3]. These should be put in a data folder in the main directory where the directory containing the `GRESCHBACH` data, should be renamed to `cells` and the `WANG14` data to `WANG14_cells`.

2 System Overview

Figure 4.4.1 already shows the overall structure of the and how different components interact. Next, in section 4.4 we also explain what the individual sections do. Hence, here we describe the same using a *directory tree*.

```
/
├── attacks - Contains the code for all of the classifiers
├── data
│   └── cells - All of the individual cell files from the GRESCHBACH dataset
├── feature_extraction - All of the code to extract the hand-picked features
│   from different models
├── feature_generation - Perform automatic feature generation
│   ├── autoencoder - Implementation of the autoencoder, how to run it and how
│   │   to extract features
│   │   ├── autoencoder - Autoencoder class
│   │   ├── feature_extraction - Used to extract features after model has been
│   │   │   trained
│   │   └── train_autoencoder - Runnable file for training
│   └── seq2seq - Implementation of the sequence-to-sequence model, how to run
│       it and how to extract features
│       ├── seq2seq - Sequence-to-sequence class
│       ├── feature_extraction - Used to extract features after model has been
│       │   trained
│       └── train_seq2seq - Runnable file for training
├── report - The latex and pdf files for this report
├── run_models - Provides the infrastructure to run a specific attack in the
│   attacks directory
├── tests - The unit tests to test everything
├── gitignore
├── travis
├── LICENSE
├── README - More information on the project and how to install/run everything
└── requirements - The python packages required
```

[2]https://nymity.ch/tor-dns/#data
[3]https://cs.uwaterloo.ca/ t55wang/wf.html

3 Components

Here how the individual components work and how to potentially extend them.

Attacks

This components implements all of the classifiers that are used to compare the hand-picked features with the automatically generated ones. Each of these classes needs to implement the interface, which has a `fit` and `predict` method. This is similar to the machine learning models provided in *sklearn* [38]. The fit method takes an X and y input and changes its internal structure such that it minimizes the error. The predict method on the other hand just takes X as its input and returns a vector y of what it predicts the input represents.

The models should also have a `is_multiclass` flag, which represents the fact whether the classifier performs a binary or multiclass classification task.

Feature Extraction

All of the logic to extract the hand-picked features for different attacks is in this module. If a new attack is added, a new file should be created in this directory, with the code to extract all the required features from a given trace. Next, this function needs to be added to the list in the `feature_extraction.py` file. This file essentially goes over all the traces within the `data/cells` directory and stores the extracted features within the `data` directory.

Feature Generation

This is one of the main components of this work, containing the code to automatically generate features. Currently, we have two models defined, namely a sequence-to-sequence model, which can be found in `seq2seq` and an autoencoder in `autoencoder`. In these directories, we first have a file, containing a class that defines a model and then two more files, which are executables, used to train and extract the features (`train_<model_name>.py` and `feature_extraction.py`).

When running the training files, after every epoch, the infrastructure will save the computational graph such that training can be continued even if it is interrupted. The name of these files depend on the Tensorflow version and the model you are training but generally look like `<model_name>_model.meta`.

Finally, all of the logic to actually extract the features is in the `feature_extraction.py` files, which again stores the fingerprints in the `data` directory.

If you wish to implement a new model, add a new directory within the `feature_generation` folder with the model name. Then create three new files, which contain the model definition, training and extracting code. The model just needs to implement two main functions such that it can be used by the rest of the infrastructure. These are `train_on_copy_task` and `get_vector_representations`, which both take some data as their input and either train the model or extract features.

Run Models

All of the infrastructure to actually run the classifiers is defined within this module. First, it defines the logic to preprocess all of the necessary data and perform the *k-fold validation*. Next, it also defines all of the scoring methods and how to actually run the models. The main logic is actually in the `run_models.py` file, which allows the user to tune certain parameters such that they can run different models.

Whenever a new classifier is added to the `attacks` directory, it should also be added within the `run_models.py` file such that a user knows that it can be run.

Unit Tests

For unit testing, we use the standard `unittest` module, which is very simple to use. To create more tests, create a new file in the `tests` directory, which starts with test and then the name of what you will be testing. Next, create a class within that file, which extends the `unittest.TestCase` class and add the methods of what you would like to test.

Then to run all of the unit tests within the `tests` directory, simply run:

```
python -m unittest discover
```

4 Contributing

There are a variety of different extensions possible, some of which are outlined in section 6.1. If you decide to implement one of these, we use a very standard git workflow so if you would like to contribute, follow these steps:

1. Fork the project repo.
2. Create a branch with the name of the particular improvement/extension that you will be working on.
3. After you are done, make sure that you run all of the tests and check if everything still works.
4. Submit a pull request from your branch to the master branch and make sure all of the tests pass on Travis.

If you discover an issue or want to work on an extension, please create an issue on our Github issues page to let people know that you are working on this particular extension.

Style Guide

When contributing, please do note that we try to adhere to the *PEP 8*[4] style guide for our python code [32]. This is done for consistency reasons and readability of the code.

Although it is not necessary to adhere to all of the guidelines, we strongly suggest for you to adhere to the basics. If the code within a pull request is unreadable or does not adhere to any of the standards, there is a very strong change that it will not be merged.

[4] https://www.python.org/dev/peps/pep-0008/

Appendix B

User Manual

In the following section, we explain how the provided code can be used to run the experiments.

Firstly, in order to run the experiments, we require data. Any dataset you want can be used, as long as each cell is represented as a Tor cell, where the SENDMEs have been removed (the same format as in table 2.1). However, for this specific project we will be using two datasets, namely `GRESCHBACH`[1] and `WANG14`[2]. Both of these need to be placed in the `data` directory.

After this, you need to be install and initialize a virtual environment and install the necessary dependencies as outlined in appendix A.

Next, there are four main scripts that need to executed to perform the WF attack:

1. `train_<model_name>.py`
2. `feature_generation.py`
3. `feature_extraction.py`
4. `run_models.py`

All of these scripts take several command-line parameters, which are outlined below. These could also be displayed if you run the scripts as follows:

```
python <script_name>.py --help
```

Train Model

As previously mentioned, we first need to train the fingerprint extraction model. This can be done by executing either of the commands below in the virtual environment, depending on which model you would like to train:

```
python feature_generation/seq2seq/train_seq2seq.py
python feature_generation/autoencoder/train_autoencoder.py
```

Both of these take several command line parameters in order to change the behavior of the models. All of these are outlined below:

[1] https://nymity.ch/tor-dns/#data
[2] https://cs.uwaterloo.ca/ t55wang/wf.html

Parameter	Type	Default	Description
batch_size	Integer	100	The size of each mini-batch.
bidirectional	Boolean	False	If true, the model will use a bidirectional encoder and a normal one otherwise.
encoder_hidden_states	Integer	120	The amount of hidden states in each RNN cell. The size of the fingerprints depends on this value. $len(\textit{fingerprint}) = 2 \times \textit{encoder\_hidden\_states}$
cell_type	String	LSTM	Which specific type of cell to use. Currently only support LSTM and GRU.
reverse_traces	Boolean	False	If true, reverses the traces and leaves them untouched otherwise. This should not be used when `bidirectional` is true.
max_time_diff	Float	Infinite	The maximum time difference *(in seconds)* after which you start cutting the traces. For instance, if set to 1, all of the traces will be cut after one second.
extension	String	.cell	The extension of the Tor cell files. We expect that they are in the following format `<webpage_id>-<instance>.<extension>`.
learning_rate	Float	0.000002	The learning rate used whilst training.
batch_norm	Boolean	False	If true, will use batch normalization within the RNN cells and otherwise the normal Tensorflow cells.

Table B.1: Parameters for the `train_seq2seq.py` file.

Parameter	Type	Default	Description
batch_size	Integer	100	The size of each mini-batch.
extension	String	.cell	The extension of the Tor cell files. We expect that they are in the following format `<webpage_id>-<instance>.<extension>`.
learning_rate	Float	0.0001	The learning rate used whilst training.
activation_func	String	sigmoid	The activation function used for the neurons.
layers	List	`[1500, 500, 100]`	The sizes of the respective layers in the encoder and decoder.
batch_norm	Boolean	False	If true, will use batch normalization for the individual layers and does not perform any normalization otherwise.

Table B.2: Parameters for the `train_autoencoder.py` file.

For example to run a simple encoder-decoder with GRU cells, a batch size of 200 and 200 hidden states, run the following command:

```
python feature_generation/seq2seq/train_seq2seq.py --batch_size
    ↪ 200 --encoder_hidden_states 200 --cell_type "GRU"
```

After running this, a couple files should be created in the main directory. First of all `loss_track.pkl`, which is a pickled file, containing the object that represents the loss over time. Next, there should also be a couple `<model_name>_model` files with different extensions, which contain the saved computational graph. Finally, it also creates a `X_test` and `y_test` file in the `data` directiory. These contains the paths and the labels to the files, which were not used for training.

Feature Generation

After the fingerprint extraction has been trained, the features need to be extracted, which can be achieved with the `feature_generation.py` module. Since we do not want to perform any testing on the same data as we trained the model on, it only extracts fingerprints from the traces in the `X_test` file.

Next, these features are stored in either the `data/seq2seq_cells` or `data/ae_cells` directory with a `.cellf` extension.

Most of the flags here are the same as in the previous section. Hence, we will only list the new arguments.

Parameter	Type	Default	Description
graph_file	String	<model_name>_model	The name of where you saved the graph. You should not need to change this, except if you change the graph name in the code.

Table B.3: Extra parameters for the `feature_generation.py` file.

For instance, to extract features from the model that we previously trained, we can run:

```
python feature_generation/seq2seq/train_seq2seq.py --batch_size
    ↪ 200 --encoder_hidden_states 200 --cell_type "GRU"
```

Feature Extraction

We then want to compare these automatically generated features with the hand-picked ones. These can be extracted using the `feature_extraction.py` script, which again has several parameters. After this script is done running, the features should be stored in the appropriate folders within the `data` directory. Again, all of the files with the extracted features will have the `cellf` extension.

Parameter	Type	Default	Description
all_files	Boolean	False	If true, it generates features for all cells and otherwise just the ones in the `X_test` file.
extension	String	.cell	Represents the extension of the cell files.

Table B.4: Parameters for the `feature_extraction.py` file.

For example:

```
python feature_extraction.py --extension ".cells"
```

Run Models

Finally, we can run the classifiers on all the extracted features using the `run_model.py` script. After finishing the k fold validation, the model then prints out the different scoring statistics.

Parameter	Type	Default	Description
model	String	kNN	Which model to run, the options are *kNN*, *random_forest*, *svc1* and *svc2*.
dir_name	String	Name of handpicked directory	If specified, it will use the features in the given directory, otherwise the standard ones for a specific model.
is_multiclass	Boolean	True	If true, trains the classifier on a multiclass task and binary otherwise.
extension	String	.cell	Represents the extension of the cell files.

Table B.5: Parameters for the `run_model.py` file.

For example, to run a random forest model with automatically generated fingerprints from the sequence-to-sequence model on a multiclass problem:

```
python run_model.py --model "random_forest" --dir_name "
    ↪ seq2seq_cells"
```

Appendix C

Project Plan

1 Problem Statement

Anonymity networks like Tor use what is called onion routing where each layer in the onion represent a new layer of encryption. This allows Tor users to freely browse the web without an ISP, government, or anyone else that might be able to sniff the traffic before the first Tor node to see which websites or services the user is accessing. However even with various layers of encryption, an attacker might still be able to infer which web page a client is browsing by performing a *website fingerprinting attack*. The attack often uses machine learning to identify several trends in the network traffic such as the number of packets per second, their size, etc. But most of these attacks rely on a trail and error process of picking the features. Hence, there is no guarantee that the features used are the most appropriate ones or even any good at all. Therefore this project will analyse the use deep learning techniques such as stacked autoencoders to automatically identify features and test their effectiveness compared to the hand-picked ones in various different models.

2 Aims and Objectives

The aims and objectives for this project are as follows:

1. **Aim:** Critically review the effectiveness of current website fingerprinting attacks.
   Objectives: 1. Analyse various models that are currently used in fingerprinting attacks. 2. See if there are any flaws in the reasoning or the experimentation. 3. Examine how would a small percentage of false positives impacts a potential attack. 4. Analyse how the rapid changing nature of some web pages would impact the attack.

2. **Aim:** Generate features automatically using deep learning techniques for a website fingerprinting attack.
   Objectives: 1. Examine different deep learning feature selection methods such as stack autoencoders. 2. Pick the most appropriate method for a website fingerprinting attack. 3. Collect the necessary data to train the feature selection method. This includes a dataset that is collected over a short period of time (days) and another one that would be collected over an extended period of time (weeks). 4. Extract a set of features using this data. 5. Compare these features to existing hand-picked ones.

3. **Aim:** Train existing models with the automatically generated features and test their effectiveness compared to hand-picked ones.
   Objectives: 1. Identify various models that could be used to test the new features with. 2. Implement the models. 3. Identify various sets of hand-picked features to compare the automatically generated features with. 4. Train those models using both hand-crafted features and the generated features. 5. Compare the effectiveness of the generated features to hand-picked ones in those models in a closed-world environment. 6. Compare their effectiveness in an open-world scenario. 7. Analyse if the feature selection technique can find persistent features that are spread across a period of time and study if this helps with the classification over time. 8. Compare the effectiness of the automatically generated features when a user uses various common defeneses against website fingerprinting like camouflage. 9. Test the attack on tor hidden services as opposed to websites. 10. Investigate an appropriate technique for evaluating the result. 11. Analyse which features tend to be the most informative (highest entropy).

3 Deliverables

The project aims to produce the following deliverables:

1. Summary of website fingerprinting attacks. This includes any related work and an analysis how effective a fingerprinting attack could be (see Aim 1).
2. An analysis of the most appropriate automatic feature generation model.
3. A dataset to train both the feature generation model and the models used for the website fingerprinting attack.
4. Fully documented source code for generating the features and the models used for the attack.
5. A strategy for testing the models.
6. An analysis of using the generated features compared to hand-picked one using diferent models. This includes how the feature generation process might be able to identify persistent features that are spread across a period of time.

4 Work Plan

Project start to end October (4 Weeks)

Research current website fingerprinting attacks.
Research various method for automatic feature selection.

Mid-October to mid-November (4 Weeks)

Refine aims and objectives.
Further research into using automatic feature selection for website fingerprinting attacks.

November (4 weeks)

Collect necessary data.
Initial experimentation with feature selection.

End November to mid-January (8 weeks)

Implement feature selection.
Implementation of various models used for attacks.
Research on how to evaluate the effectiveness of a model.
Work on the Interim Report.

Mid-January to mid-February (4 weeks)

Perform tests and evaluate the performance.

Mid-February to end of March (6 weeks)

Work on Final Report.

Appendix D

Interim Report

1 Current Progress

We have updated some requirements that were mentioned in the project plan. Rather than first collecting a large dataset of website traces over a long period of time, we decided to start with existing datasets to perform some experiments. Then later, if we get the opportunity to do so, we can still collect our own data.

Given the nature of the challenge, a majority of the time will be spend on researching models that perform automatic feature selection. After careful examination, there are only two couple deep learning methods that seem to be appropriate, a *RNN Encoder Decoder* or a *Bidirectional RNN Encoder*. We also considered using a *Stacked Autoencoder* but it did not seem to be appropriate as it requires a fixed-length vector as an input. Hence, if we were to use it, we either have to pad or compress the traces, which are both not elegant solutions. In addition to simply training these models on the data, we might perform *denoising* on the them, which essentially means learning the models to distinguish uncorrupted data from corrupted data. This final step allows us to fine tune the encoders to get a consistent performance even if some of the data is noisy.

Although we have started experimenting with some of these models, they have not been fully implemented. However we are still on schedule for doing so.

Finally, we have also selected a set of previously successful website fingerprinting attacks. An environment has been set up, some of these models have been implemented and the infrastructure is in place to extract a set of hand-picked features from the raw data. This will allow us to quickly perform a performance comparison with the manually engineered features and the automatically selected ones.

2 Remaining Work

The first priority is to fully implement a *RNN Encoder Decoder* or a *Bidirectional RNN Encoder* and fine tune it. Next, we have to perform the analysis on how appropriate our automatically engineered features are compared to hand-picked ones. So this includes finishing the implementation of existing models, and the infrastructure to extract the features.

Next, we need to pick a set of criteria that we will use to compare the predictive power of several models. Using these criteria we will then have to perform a thorough analysis of how the automatically engineered features compared to the hand-picked ones.

Finally, after all of the above has been completed, we will focus on finished writing the final report. If there is still time remaining, we might still try to collect our own data over an extended period of time

Appendix E

Code Listing

In the following section we include the code that implements the unsupervised deep learning models used.

1 Autoencoder

```
"""
Implements a simple stacked autoencoder.

Hyperparameters to tune:
------------------------
- Learning rate
- Activation function (sigmoid, ReLU, atan)
- Amount of neurons in each layer
- Learning function (GradientDescentOptimizer,
    RMSProp, AdamOptimizer)
- Batch size
"""
"""
Implements a simple stacked autoencoder.

Hyperparameters to tune:
------------------------
- Learning rate
- Activation function (sigmoid, ReLU, atan)
- Amount of neurons in each layer
- Learning function (GradientDescentOptimizer,
    RMSProp, AdamOptimizer)
- Batch size
"""
import numpy as np
import tensorflow as tf

from sys import stdout, path
from os import path as ospath

from sklearn.preprocessing import MinMaxScaler

path.append(ospath.dirname(ospath.dirname(ospath.
    abspath(__file__))))
import helpers

class AutoEncoder():
    """
    Implements an autoencoder that tries to learn a
        representation for web page traces.

    Atrributes:
        - activation_func is a tensorflow function,
            representing the activation function used
            .
             *(Often found in `tf.nn`)*
        - encoder is a computation, representing the
            encoder layers
        - decoder is another computtational graph,
            representing the decoder layers
```

```
    - loss is the operation for the mean squared
        error (MSE)
    - train_op is the train operation (`RMSProp
        `)
    - layers is a list of integerrs, determining
        the amount of layers and their size
    - is_training is a boolean representing
        whether you are training the autoencoder
        or not *(used in the batch_norm layer)*.
    - batch_size
    - learning_rate
"""

def __init__(self, layers, batch_size,
    activation_func=tf.nn.sigmoid, saved_graph=
    None, sess=None, learning_rate=0.0001,
    batch_norm=False):
    """
    @param layers is a list of integers,
        determining the amount of layers and
        their size
        starting with the input size
    """
    if len(layers) < 2:
        print("Amount of layers must be greater
            than 1")
        exit(0)

    self.batch_size = batch_size
    self.learning_rate = learning_rate
    self.activation_func = activation_func
    self.batch_norm = batch_norm
    self.is_training = True

    # Use this in data preprocessing
    self.layers = layers

    self._make_graph(layers)

    if saved_graph is not None and sess is not
        None:
        self.import_from_file(sess, saved_graph)

def _make_graph(self, layers):
    """
    Constructs the computational graph

    @param layers is a list of integers,
        determining the size of the layers
    """
    self._init_placeholders(layers[0])

    self.encoder = self._init_encoder(layers)
    self.decoder = self._init_decoder(layers)

    self._init_train()

def _init_placeholders(self, first_layer):
    """
    The main placeholders for input and output
        data
    """
    self.encoder_inputs = tf.placeholder(tf.
        float32, [self.batch_size, first_layer])
```

```
        # We could technically use the same value as
            encoder_inputs but we do not
        # for future possible extensions
        self.decoder_targets = tf.placeholder(tf.
            float32, [self.batch_size, first_layer])


    def _get_layer(self, layer_input,
        size_last_layer, size_current_layer):
        """
        Returns a layer with a batch normalized
            input, depending on the `batch_norm flag`

        @param layer_input is the value used as an
            input to the layer.
        @param size_last_layer is the size of the
            last layer (used in weight) or the size
            of the input
        @param size_current_layer is the size of the
            current layer (used in weight and bias)
        """
        weight = tf.Variable(tf.random_normal([
            size_last_layer, size_current_layer]))
        bias = tf.Variable(tf.random_normal([
            size_current_layer]))

        if not self.batch_norm:
            return self.activation_func(tf.add(tf.
                matmul(layer_input, weight), bias))


        layer_input = tf.contrib.layers.batch_norm(
            layer_input,
                                    center=True, scale=True,
                                    is_training=self.
                                        is_training,
                                    scope='bn{}-{}'.format(
                                        size_last_layer,
                                        size_current_layer))

        return self.activation_func(tf.add(tf.matmul
            (layer_input, weight), bias))

    def _init_encoder(self, layers):
        """
        Creates the layers of the decoder and
            returns the last layer.
        """
        previous_layer = None

        # We don't want to enumerate over the last
            one
        for i in range(len(layers) - 1):
            current_layer = None
            if previous_layer is None:
                current_layer = self._get_layer(self
                    .encoder_inputs, layers[i],
                    layers[i + 1])
            else:
                current_layer = self._get_layer(
                    previous_layer, layers[i], layers
                    [i + 1])

            previous_layer = current_layer

        # Will be the last layer
```

```
        return previous_layer

    def _init_decoder(self, layers):
        """
        Creates the decoder graph and returns the
            last layer
        """
        previous_layer = None

        # We don't want to enumerate over the last
            one
        for i in range(len(layers) - 1, 0, -1):
            current_layer = None
            if previous_layer is None:
                current_layer = self._get_layer(self
                    .encoder, layers[i], layers[i -
                    1])
            else:
                current_layer = self._get_layer(
                    previous_layer, layers[i], layers
                    [i - 1])

            previous_layer = current_layer

        # Will be the last layer
        return previous_layer

    def _init_train(self):
        """
        Create the train operation
        """
        self.loss = tf.reduce_sum(tf.square(self.
            decoder_targets - self.decoder))
        # Which optimizer to use? `
            GradientDescentOptimizer`, `AdamOptimizer
            ` or `RMSProp`?
        self.train_op = tf.train.AdamOptimizer(self.
            learning_rate).minimize(self.loss)

    def _init_batch_norm(self):
        """
        Adds a batch normalization layer.
        """
        self.decoder = tf.contrib.layers.batch_norm(
            self.decoder,
                        center=True, scale=True,
                        is_training=self.is_training
                            ,
                        scope='bn')

    def set_is_training(is_training):
        """
        Sets the `is_training` class variable, used
            in th batch normalization layer.
        If `batch_norm == False`, this does not make
             a difference but if it is true, this
            variable should be set to false after
            training.
        """
        self.is_training = is_training

    def _process_trace(self, trace, n):
        """
        Cuts the traces after `n` steps or pads them
             such that they are of length `n`.
```

```
        """
        features = []

        for packet in trace:
            # Either positive or negative depending
                on whether its incoming or outgoing.
            features.append(packet[0] * packet[1])

            if len(features) == n:
                break

        for i in range(len(features), n):
            features.append(0)

        return features


    def next_batch(self, batches, in_memory):
        """
        Returns the next batch in some fixed-length
            representation.
        Currently we use Panchenko et al.'s
            cumulative traces

        @param batches an iterator with all of the
            batches (
             if in_memory == True:
                 in batch-major form without padding
             else:
                 A list of paths to the files
        )
        @param in_memory is a boolean value
        @return if in_memory is False, returns a
            tuple of (dict, [paths]) where paths is a
             list of paths for each batch
             else it returns a dict for training
        """
        batch = next(batches)
        data_batch = batch

        if not in_memory:
            data_batch = [helpers.read_cell_file(
                path) for path in batch]

        data_batch = [self._process_trace(trace,
            self.layers[0]) for trace in data_batch]

        min_max_scaler = MinMaxScaler()
        data_batch = min_max_scaler.fit_transform(
            data_batch)

        encoder_inputs_ = data_batch
        decoder_targets_ = data_batch

        train_dict = {
            self.encoder_inputs: encoder_inputs_,
            self.decoder_targets: decoder_targets_,
        }

        if not in_memory:
            return (train_dict, batch)
        return train_dict

    def save(self, sess, file_name):
        """
```

```
        Save the model in a file

        @param sess is the session
        @param file_name is the file name without
            the extension
        """
        saver = tf.train.Saver()
        saver.save(sess, file_name)

    def import_from_file(self, sess, file_name):
        """
        Imports the graph from a file

        @param sess is the session
        @param file_name is a string that represents
             the file name
             without the extension
        """

        # Get the graph
        saver = tf.train.Saver()

        # Restore the variables
        saver.restore(sess, file_name)



def train_on_copy_task(sess, model, data,
                       batch_size=100,
                       max_batches=None,
                       batches_in_epoch=1000,
                       verbose=False):
    """
    Train the `AutoEncoder` on a copy task

    @param sess is a tensorflow session
    @param model is the autoencoder model
    @param data is the data (in batch-major form and
         not padded or a list of files (depending on
        `in_memory`))
    """
    batches = helpers.get_batches(data, batch_size=
        batch_size)

    loss_track = []

    batches_in_data = len(data) // batch_size
    if max_batches is None or batches_in_data <
        max_batches:
        max_batches = batches_in_data - 1

    try:
        for batch in range(max_batches):
            print("Batch {}/{}".format(batch,
                max_batches))
            fd, _ = model.next_batch(batches, False)
            _, l = sess.run([model.train_op, model.
                loss], fd)

            loss_track.append(l)

            if batch == 0 or batch %
                batches_in_epoch == 0:
                model.save(sess, 'autoencoder_model'
                    )
```

```
                helpers.save_object(loss_track, '
                    loss_track.pkl')

                if verbose:
                    stdout.write('  minibatch loss: 
                        {}\n'.format(sess.run(model.
                        loss, fd)))
                    predict_ = sess.run(model.
                        decoder_outputs, fd)
                    for i, (inp, pred) in enumerate(
                        zip(fd[model.encoder_inputs].
                        swapaxes(0, 1), predict_.
                        swapaxes(0, 1))):
                        stdout.write('  sample {}:\n
                            '.format(i + 1))
                        stdout.write('    input     
                            > {}\n'.format(inp))
                        stdout.write('    predicted 
                            > {}\n'.format(pred))
                        if i >= 0:
                            break
                    stdout.write('\n')

        except KeyboardInterrupt:
            stdout.write('training interrupted')
            model.save(sess, 'autoencoder_model')
            exit(0)

        model.save(sess, 'autoencoder_model')
        helpers.save_object(loss_track, 'loss_track.pkl'
            )

        return loss_track


    def get_vector_representations(sess, model, data,
        save_dir,
                                   batch_size=100,
                                   max_batches=None,
                                   batches_in_epoch=1000,
                                   extension=".cell"):
        """
        Given a trained model, gets a vector
            representation for the traces in batch

        @param sess is a tensorflow session
        @param model is the autoencoder model
        @param data is the data (in batch-major form and
             not padded or a list of files (depending on
            `in_memory`))
        """
        batches = helpers.get_batches(data, batch_size=
            batch_size)

        batches_in_data = len(data) // batch_size
        if max_batches is None or batches_in_data <
            max_batches:
            max_batches = batches_in_data - 1

        try:
            for batch in range(max_batches):
                print("Batch {}/{}".format(batch,
                    max_batches))
                fd, paths = model.next_batch(batches,
                    False)
                l = sess.run(model.encoder, fd)
```

```
            file_names = [helpers.
                extract_filename_from_path(path,
                extension) for path in paths]

            for file_name, features in zip(
                file_names, list(l)):
                 helpers.write_to_file(features,
                     save_dir, file_name,
                     new_extension=".cellf")

    except KeyboardInterrupt:
        stdout.write('Interrupted')
        exit(0)

    return results
```


2 Sequence-to-sequence model

```
"""
This file implements a RNN encoder-decoder model (
    also known as sequence-to-sequence models).

We made the choice not to implement an attention
    mechanism (which means that the decoder is
    allowed to have a 'peak' at the input).
The reason why is because we are not trying to
    maximize the output of the decoder but instead
    the feature selection process.
(http://suriyadeepan.github.io/2016-06-28-easy-
    seq2seq/)

We will use batch-major rather than time-major even
    though time-major is slightly more efficient
    since it makes the feature extraction process a lot
    easier.

We will not be using bucketing because traces of the
     same webpage will have the same length.
Therefore every batch, we will most likely be
    training the seq2seq model on one webpage

! Does encoder share weights with decoder or not (
    Less computation vs natural (https://arxiv.org/
    pdf/1409.3215.pdf))
! Reverse traces (https://arxiv.org/pdf/1409.3215.
    pdf)

Hyperparameters to tune:
------------------------

- Learning rate
- Which cell to use (GRU vs LSTM) or a deep RNN
    architecture using `MultiRNNCell`
- Reversing traces
- Bidirectional encoder
- Other objective functions (such as MSE,...)
- Amount of encoder and decoder hidden states
"""
import numpy as np
import tensorflow as tf

from sys import stdout, path
from os import path as ospath

from tensorflow.contrib.rnn import LSTMStateTuple

path.append(ospath.dirname(ospath.dirname(ospath.
```

```
    abspath(__file__))))
import helpers


class Seq2SeqModel():
    """
    Implements a sequence to sequence model for real
        values

    Attributes:
        - encoder_cell is the cell that will be used
            for encoding
            (Should be part of `tf.nn.rnn_cell`)
        - decoder cell is the cell used for decoding
            (Should be part of `tf.nn.rnn_cell`)

        - seq_width shows how many features each
            input in the sequence has
            (For website fingerprinting this is only
                2 (packet_size, incoming))
        - batch_size

        - bidirectional is a boolean value that
            determines whether the encoder is
            bidirectional or not
        - reverse is also a boolean value that when
            if true, reversed the traces for training
    """

    def __init__(self, encoder_cell, decoder_cell,
        seq_width, batch_size=100, bidirectional=
        False, reverse=False, saved_graph=None, sess=
        None, learning_rate=0.0006):
        """
        @param saved_graph is a string, representing
            the path to the saved graph
        """
        # Constants
        self.PAD = 0
        self.EOS = -1

        self.reverse = reverse
        self.seq_width = seq_width
        self.batch_size = batch_size
        self.learning_rate = learning_rate

        self.bidirectional = bidirectional

        self.encoder_cell = encoder_cell
        self.decoder_cell = decoder_cell

        self._make_graph()

        if saved_graph is not None and sess is not
            None:
            self.import_from_file(sess, saved_graph)

    def _make_graph(self):
        """
        Construct the graph
        """

        self._init_placeholders()

        self._init_encoder()
        self._init_decoder()
```

```
        self._init_train()

    def _init_placeholders(self):
        """
        The main placeholders used for the input
            data, and output
        """
        # The usual format is: `[self.batch_size,
            max_sequence_length, self.seq_width]`
        # But we define `max_sequence_length` as
            None to make it dynamic so we only need
            to pad
        # each batch to the maximum sequence length
        self.encoder_inputs = tf.placeholder(tf.
            float32,
             [self.batch_size, None, self.seq_width])

        self.encoder_inputs_length = tf.placeholder(
            tf.int32, [self.batch_size])

        self.decoder_targets = tf.placeholder(tf.
            float32,
             [self.batch_size, None, self.seq_width])

    def _init_encoder(self):
        """
        Creates the encoder attributes

        Attributes:
            - encoder_outputs is shaped [
                max_sequence_length, batch_size,
                seq_width]
                    (since time-major == True)
            - encoder_final_state is shaped [
                batch_size, encoder_cell.state_size]
        """
        if not self.bidirectional:
            with tf.variable_scope('Encoder') as
                scope:
                self.encoder_outputs, self.
                    encoder_final_state = tf.nn.
                    dynamic_rnn(
                     cell=self.encoder_cell,
                     dtype=tf.float32,
                     sequence_length=self.
                         encoder_inputs_length,
                     inputs=self.encoder_inputs,
                     time_major=False)
        else:
            ((encoder_fw_outputs,
              encoder_bw_outputs),
             (encoder_fw_final_state,
              encoder_bw_final_state)) = (
                tf.nn.bidirectional_dynamic_rnn(
                    cell_fw=self.encoder_cell,
                     cell_bw=self.encoder_cell,
                     inputs=self.encoder_inputs,
                     sequence_length=self.
                         encoder_inputs_length,
                     dtype=tf.float32, time_major=
                         False)
                )

            self.encoder_outputs = tf.concat((
                encoder_fw_outputs,
```

```
                encoder_bw_outputs), 2)

            if isinstance(encoder_fw_final_state,
                LSTMStateTuple):
                encoder_final_state_c = tf.concat(
                    (encoder_fw_final_state.c,
                        encoder_bw_final_state.c), 1)

                encoder_final_state_h = tf.concat(
                    (encoder_fw_final_state.h,
                        encoder_bw_final_state.h), 1)

                self.encoder_final_state =
                    LSTMStateTuple(
                    c=encoder_final_state_c,
                    h=encoder_final_state_h
                )

            else:
                self.encoder_final_state = tf.concat
                    (
                    (encoder_fw_final_state,
                        encoder_bw_final_state), 1)

    def _init_decoder(self):
        """
        Creates decoder attributes.
        We cannot simply use a dynamic_rnn since we
            are feeding the outputs of the
        decoder back into the inputs.
        Therefore we use a raw_rnn and emulate a
            dynamic_rnn with this behavior.
        (https://github.com/tensorflow/tensorflow/
            blob/master/tensorflow/python/ops/rnn.py)
        """
        # EOS token added
        self.decoder_inputs_length = self.
            encoder_inputs_length + 1

        def loop_fn_initial(time, cell_output,
            cell_state, loop_state):
            elements_finished = (time >= self.
                decoder_inputs_length)

            # EOS token (0 + self.EOS)
            initial_input = tf.zeros([self.
                batch_size, self.decoder_cell.
                output_size], dtype=tf.float32) +
                self.EOS
            initial_cell_state = self.
                encoder_final_state
            initial_loop_state = None  # we don't
                need to pass any additional
                information

            return (elements_finished,
                    initial_input,
                    initial_cell_state,
                    None,  # cell output is dummy
                        here
                    initial_loop_state)

        def loop_fn(time, cell_output, cell_state,
            loop_state):
            if cell_output is None:  # time == 0
                return loop_fn_initial(time,
```

```
            cell_output, cell_state,
            loop_state)

    cell_output.set_shape([self.batch_size,
        self.decoder_cell.output_size])

    emit_output = cell_output

    next_cell_state = cell_state

    elements_finished = (time >= self.
        decoder_inputs_length)
    finished = tf.reduce_all(
        elements_finished)

    next_input = tf.cond(
        finished,
        lambda: tf.zeros([self.batch_size,
            self.decoder_cell.output_size],
            dtype=tf.float32), # self.PAD
        lambda: cell_output # Use the input
            from the previous cell
    )

    next_loop_state = None

    return (
        elements_finished,
        next_input,
        next_cell_state,
        emit_output,
        next_loop_state
    )

    decoder_outputs_ta, decoder_final_state, _ =
        tf.nn.raw_rnn(self.decoder_cell, loop_fn
        )
    self.decoder_outputs = decoder_outputs_ta.
        stack()
    self.decoder_outputs = tf.transpose(self.
        decoder_outputs, [1, 0, 2])

    with tf.variable_scope('
        DecoderOutputProjection') as scope:
        self.decoder_outputs = self.projection(
            self.decoder_outputs, self.seq_width,
            scope)

def _init_train(self):
    self.loss = tf.reduce_sum(tf.square(self.
        decoder_targets - self.decoder_outputs))

    # Which optimizer to use? `
        GradientDescentOptimizer`, `AdamOptimizer
        ` or `RMSProp`?
    self.train_op = tf.train.AdamOptimizer(self.
        learning_rate).minimize(self.loss)

def projection(self, inputs, projection_size,
    scope):
    """
    Projects the input with a known amount of
        features to a `projection_size amount of
        features`

    @param inputs is shaped like [time, batch,
```

```
    input_size] or [batch, input_size]
@param projection_size int32
@param scope outer variable scope
"""
input_size = inputs.get_shape()[-1].value

with tf.variable_scope(scope) as scope:
    W = tf.get_variable(name='W', shape=[
        input_size, projection_size],
                        dtype=tf.float32)

    b = tf.get_variable(name='b', shape=[
        projection_size],
                        dtype=tf.float32,
                        initializer=tf.
                            constant_initializer
                            (0, dtype=tf.
                            float32))

input_shape = tf.unstack(tf.shape(inputs))

if len(input_shape) == 3:
    time, batch, _ = input_shape  # dynamic
        parts of shape
    inputs = tf.reshape(inputs, [-1,
        input_size])

elif len(input_shape) == 2:
    batch, _depth = input_shape

else:
    raise ValueError("Weird input shape: {}"
        .format(inputs))

    linear = tf.add(tf.matmul(inputs, W), b)

    if len(input_shape) == 3:
        linear = tf.reshape(linear, [time, batch
            , projection_size])

    return linear

def next_batch(self, batches, in_memory,
    max_time_diff=float("inf")):
    """
    Returns the next batch.

    @param batches an iterator with all of the
        batches (
        if in_memory == True:
            in batch-major form without padding
        else:
            A list of paths to the files
    )
    @param in_memory is a boolean value
    @param max_time_diff **(should only be
        defined if `in_memory == False`)**
        specifies what the maximum time
            different between the first packet in
            the trace and the last one should be

    @return if in_memory is False, returns a
        tuple of (dict, [paths], max_length)
        where paths is a list of paths for each
        batch
        else it returns a dict for training
```

```
        """
        batch = next(batches)
        data_batch = batch

        if not in_memory:
            data_batch = [helpers.read_cell_file(
                path, max_time_diff=max_time_diff)
                for path in batch]
            for i, cell in enumerate(data_batch):
                data_batch[i] = [packet[0] * packet
                    [1] for packet in cell]

        data_batch, encoder_input_lengths_ = helpers
            .pad_traces(data_batch, reverse=self.
            reverse, seq_width=self.seq_width)
        encoder_inputs_ = data_batch

        decoder_targets_ = helpers.add_EOS(
            data_batch, encoder_input_lengths_)

        train_dict = {
            self.encoder_inputs: encoder_inputs_,
            self.encoder_inputs_length:
                encoder_input_lengths_,
            self.decoder_targets: decoder_targets_,
        }

        if not in_memory:
            return (train_dict, batch, max(
                encoder_input_lengths_))
        return train_dict

    def save(self, sess, file_name):
        """
        Save the model in a file

        @param sess is the session
        @param file_name is the file name without
            the extension
        """
        saver = tf.train.Saver()
        saver.save(sess, file_name)
        # saver.export_meta_graph(filename=file_name
            + '.meta')

    def import_from_file(self, sess, file_name):
        """
        Imports the graph from a file

        @param sess is the session
        @param file_name is a string that represents
            the file name
            without the extension
        """

        # Get the graph
        saver = tf.train.Saver()

        # Restore the variables
        saver.restore(sess, file_name)


def train_on_copy_task(sess, model, data,
                       batch_size=100,
                       max_batches=None,
                       batches_in_epoch=1000,
```

```
                    max_time_diff=float("inf"),
                    verbose=False):
    """
    Train the `Seq2SeqModel` on a copy task

    @param sess is a tensorflow session
    @param model is the seq2seq model
    @param data is the data (in batch-major form and
         not padded or a list of files (depending on
        `in_memory`))
    """
    batches = helpers.get_batches(data, batch_size=
        batch_size)

    loss_track = []

    batches_in_data = len(data) // batch_size
    if max_batches is None or batches_in_data <
        max_batches:
        max_batches = batches_in_data - 1

    try:
        for batch in range(max_batches):
            print("Batch {}/{}".format(batch,
                max_batches))
            fd, _, length = model.next_batch(batches
                , False, max_time_diff)
            _, l = sess.run([model.train_op, model.
                loss], fd)
            loss_track.append(l / length)

            if batch == 0 or batch %
                batches_in_epoch == 0:
                model.save(sess, 'seq2seq_model')
                helpers.save_object(loss_track, '
                    loss_track.pkl')

                if verbose:
                    stdout.write('  minibatch loss:
                        {}\n'.format(sess.run(model.
                        loss, fd)))
                    predict_ = sess.run(model.
                        decoder_outputs, fd)
                    for i, (inp, pred) in enumerate(
                        zip(fd[model.encoder_inputs].
                        swapaxes(0, 1), predict_.
                        swapaxes(0, 1))):
                        stdout.write('  sample {}:\n
                            '.format(i + 1))
                        stdout.write('    input
                            > {}\n'.format(inp))
                        stdout.write('    predicted
                            > {}\n'.format(pred))
                        if i >= 0:
                            break
                    stdout.write('\n')

    except KeyboardInterrupt:
        stdout.write('training interrupted')
        model.save(sess, 'seq2seq_model')
        exit(0)

    model.save(sess, 'seq2seq_model')
    helpers.save_object(loss_track, 'loss_track.pkl'
        )
```

```
    return loss_track

def get_vector_representations(sess, model, data,
    save_dir,
                           batch_size=100,
                           max_batches=None,
                           batches_in_epoch=1000,
                           max_time_diff=float("inf"),
                           extension=".cell"):
    """
    Given a trained model, gets a vector
        representation for the traces in batch

    @param sess is a tensorflow session
    @param model is the seq2seq model
    @param data is the data (in batch-major form and
         not padded or a list of files (depending on
        `in_memory`))
    """
    batches = helpers.get_batches(data, batch_size=
        batch_size)

    batches_in_data = len(data) // batch_size
    if max_batches is None or batches_in_data <
        max_batches:
        max_batches = batches_in_data - 1

    try:
        for batch in range(max_batches):
            print("Batch {}/{}".format(batch,
                max_batches))
            fd, paths, _ = model.next_batch(batches,
                False, max_time_diff)
            l = sess.run(model.encoder_final_state,
                fd)

            # Returns a tuple, so we concatenate
            if isinstance(l, LSTMStateTuple):
                l = np.concatenate((l.c, l.h), axis
                    =1)

            file_names = [helpers.
                extract_filename_from_path(path,
                extension) for path in paths]

            for file_name, features in zip(
                file_names, list(l)):
                helpers.write_to_file(features,
                    save_dir, file_name,
                    new_extension=".cellf")

    except KeyboardInterrupt:
        stdout.write('Interrupted')
        exit(0)
```

3 Batch-normalized LSTM cell

```
"""
Implements batch normalized LSTM cells described in
    https://arxiv.org/pdf/1510.01378.pdf.
"""

from tensorflow.contrib.rnn import LSTMCell,
    LSTMStateTuple

from tensorflow.python.framework import ops
```

```
from tensorflow.python.framework import tensor_shape
from tensorflow.python.framework import tensor_util
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import clip_ops
from tensorflow.python.ops import embedding_ops
from tensorflow.python.ops import init_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import nn_ops
from tensorflow.python.ops import
    partitioned_variables
from tensorflow.python.ops import random_ops
from tensorflow.python.ops import variable_scope as
    vs

from tensorflow.python.ops.math_ops import sigmoid
from tensorflow.python.ops.math_ops import tanh
from tensorflow.python.ops.rnn_cell_impl import
    _RNNCell as RNNCell

from tensorflow.python.platform import tf_logging as
     logging
from tensorflow.python.util import nest

from tensorflow.contrib.layers import batch_norm

import contextlib

_BIAS_VARIABLE_NAME = "biases"
_WEIGHTS_VARIABLE_NAME = "weights"

@contextlib.contextmanager
def _checked_scope(cell, scope, reuse=None, **kwargs
    ):
if reuse is not None:
  kwargs["reuse"] = reuse
with vs.variable_scope(scope, **kwargs) as
    checking_scope:
  scope_name = checking_scope.name
  if hasattr(cell, "_scope"):
    cell_scope = cell._scope  # pylint: disable=
        protected-access
    if cell_scope.name != checking_scope.name:
      raise ValueError(
          "Attempt␣to␣reuse␣RNNCell␣%s␣with␣a␣
              different␣variable␣scope␣than␣"
          "its␣first␣use.␣␣First␣use␣of␣cell␣was␣
              with␣scope␣'%s',␣this␣"
          "attempt␣is␣with␣scope␣'%s'.␣␣Please␣
              create␣a␣new␣instance␣of␣the␣"
          "cell␣if␣you␣would␣like␣it␣to␣use␣a␣
              different␣set␣of␣weights.␣␣"
          "If␣before␣you␣were␣using:␣MultiRNNCell
              ([%s(...)]␣*␣num_layers),␣"
          "change␣to:␣MultiRNNCell([%s(...)␣for␣_␣
              in␣range(num_layers)]).␣␣"
          "If␣before␣you␣were␣using␣the␣same␣cell␣
              instance␣as␣both␣the␣"
          "forward␣and␣reverse␣cell␣of␣a␣
              bidirectional␣RNN,␣simply␣create␣"
          "two␣instances␣(one␣for␣forward,␣one␣for
              ␣reverse).␣␣"
          "In␣May␣2017,␣we␣will␣start␣
              transitioning␣this␣cell's␣behavior␣"
          "to␣use␣existing␣stored␣weights,␣if␣any,
              ␣when␣it␣is␣called␣"
          "with␣scope=None␣(which␣can␣lead␣to␣
```

```
            silent␣model␣degradation,␣so␣"
          "this␣error␣will␣remain␣until␣then.)"
          % (cell, cell_scope.name, scope_name,
              type(cell).__name__,
              type(cell).__name__))
else:
  weights_found = False
  try:
    with vs.variable_scope(checking_scope, reuse
        =True):
      vs.get_variable(_WEIGHTS_VARIABLE_NAME)
    weights_found = True
  except ValueError:
    pass
  if weights_found and reuse is None:
    raise ValueError(
        "Attempt␣to␣have␣a␣second␣RNNCell␣use␣
            the␣weights␣of␣a␣variable␣"
        "scope␣that␣already␣has␣weights:␣'%s';␣
            and␣the␣cell␣was␣not␣"
        "constructed␣as␣%s(...,␣reuse=True).␣␣"
        "To␣share␣the␣weights␣of␣an␣RNNCell,␣
            simply␣"
        "reuse␣it␣in␣your␣second␣calculation,␣or
            ␣create␣a␣new␣one␣with␣"
        "the␣argument␣reuse=True." % (scope_name
            , type(cell).__name__))

# Everything is OK.  Update the cell's scope and
    yield it.
cell._scope = checking_scope  # pylint: disable=
    protected-access
yield checking_scope
```

```
class BNLSTMCell(LSTMCell):
  """Long short-term memory unit (LSTM) recurrent
      network cell.
  The default non-peephole implementation is based
      on:
    http://deeplearning.cs.cmu.edu/pdfs/
        Hochreiter97_lstm.pdf
  S. Hochreiter and J. Schmidhuber.
  "Long Short-Term Memory". Neural Computation, 9(8)
      :1735-1780, 1997.
  The peephole implementation is based on:
    https://research.google.com/pubs/archive/43905.
        pdf
  Hasim Sak, Andrew Senior, and Francoise Beaufays.
  "Long short-term memory recurrent neural network
      architectures for
   large scale acoustic modeling." INTERSPEECH,
       2014.
  The class uses optional peep-hole connections,
      optional cell clipping, and
  an optional projection layer.
  """

  def __init__(self, num_units, input_size=None,
               use_peepholes=False, cell_clip=None,
               initializer=None, num_proj=None,
                   proj_clip=None,
               num_unit_shards=None, num_proj_shards
                   =None,
               forget_bias=1.0, state_is_tuple=True,
               activation=tanh, is_training=True,
                   batch_norm=True):
```

```
    """Initialize the parameters for an LSTM cell.
    Args:
      num_units: int, The number of units in the
        LSTM cell
      input_size: Deprecated and unused.
      use_peepholes: bool, set True to enable
        diagonal/peephole connections.
      cell_clip: (optional) A float value, if
        provided the cell state is clipped
        by this value prior to the cell output
        activation.
      initializer: (optional) The initializer to use
        for the weight and
        projection matrices.
      num_proj: (optional) int, The output
        dimensionality for the projection
        matrices.  If None, no projection is
        performed.
      proj_clip: (optional) A float value.  If `
        num_proj > 0` and `proj_clip` is
        provided, then the projected values are
        clipped elementwise to within
        `[-proj_clip, proj_clip]`.
      num_unit_shards: Deprecated, will be removed
        by Jan. 2017.
        Use a variable_scope partitioner instead.
      num_proj_shards: Deprecated, will be removed
        by Jan. 2017.
        Use a variable_scope partitioner instead.
      forget_bias: Biases of the forget gate are
        initialized by default to 1
        in order to reduce the scale of forgetting
        at the beginning of
        the training.
      state_is_tuple: If True, accepted and returned
        states are 2-tuples of
        the `c_state` and `m_state`.  If False, they
        are concatenated
        along the column axis.  This latter behavior
        will soon be deprecated.
      activation: Activation function of the inner
        states.
      is_training: Python boolean describing whether
        or not you are currently
        training. Should only be changed if `
        batch_norm == True`
      batch_norm: Python boolean that indicated
        whether or not the cell is
        batch normalized
    """
    self._is_training = is_training
    self._batch_norm = batch_norm

    super().__init__(num_units, input_size=
        input_size,
                     use_peepholes=use_peepholes,
                         cell_clip=cell_clip,
                     initializer=initializer, num_proj=
                         num_proj, proj_clip=proj_clip,
                     num_unit_shards=num_unit_shards,
                         num_proj_shards=num_proj_shards,
                     forget_bias=forget_bias,
                         state_is_tuple=state_is_tuple,
                     activation=activation)
```

```
def __call__(self, inputs, state, scope=None):
  """Run one step of LSTM.
  Args:
    inputs: input Tensor, 2D, batch x num_units.
    state: if `state_is_tuple` is False, this must
        be a state Tensor,
      `2-D, batch x state_size`.  If `
          state_is_tuple` is True, this must be a
      tuple of state Tensors, both `2-D`, with
          column sizes `c_state` and
      `m_state`.
    scope: VariableScope for the created subgraph;
        defaults to "lstm_cell".
  Returns:
    A tuple containing:
    - A `2-D, [batch x output_dim]`, Tensor
        representing the output of the
      LSTM after reading `inputs` when previous
          state was `state`.
      Here output_dim is:
         num_proj if num_proj was set,
         num_units otherwise.
    - Tensor(s) representing the new state of LSTM
        after reading `inputs` when
      the previous state was `state`.  Same type
          and shape(s) as `state`.
  Raises:
    ValueError: If input size cannot be inferred
        from inputs via
      static shape inference.
  """
  num_proj = self._num_units if self._num_proj is
      None else self._num_proj

  if self._state_is_tuple:
    (c_prev, m_prev) = state
  else:
    c_prev = array_ops.slice(state, [0, 0], [-1,
        self._num_units])
    m_prev = array_ops.slice(state, [0, self.
        _num_units], [-1, num_proj])

  dtype = inputs.dtype
  input_size = inputs.get_shape().with_rank(2)[1]
  if input_size.value is None:
    raise ValueError("Could not infer input size
        from inputs.get_shape()[-1]")
  with _checked_scope(self, scope or "lstm_cell",
                      initializer=self.
                          _initializer) as
                      unit_scope:
    if self._num_unit_shards is not None:
      unit_scope.set_partitioner(
          partitioned_variables.
              fixed_size_partitioner(
              self._num_unit_shards))
    # i = input_gate, j = new_input, f =
        forget_gate, o = output_gate
    lstm_matrix = _linear([inputs, m_prev], 4 *
        self._num_units, bias=True)
    i, j, f, o = array_ops.split(
        value=lstm_matrix, num_or_size_splits=4,
            axis=1)
    # Diagonal connections
    if self._use_peepholes:
      with vs.variable_scope(unit_scope) as
```

```
      projection_scope:
    if self._num_unit_shards is not None:
      projection_scope.set_partitioner(None)
    w_f_diag = vs.get_variable(
        "w_f_diag", shape=[self._num_units],
            dtype=dtype)
    w_i_diag = vs.get_variable(
        "w_i_diag", shape=[self._num_units],
            dtype=dtype)
    w_o_diag = vs.get_variable(
        "w_o_diag", shape=[self._num_units],
            dtype=dtype)

if self._use_peepholes:
  res = (sigmoid(f + self._forget_bias +
     w_f_diag * c_prev) * c_prev +
      sigmoid(i + w_i_diag * c_prev) * self.
          _activation(j))
  if self._batch_norm:
    c = batch_norm(res,
                   center=True, scale=True,
                   is_training=self.
                       _is_training,
                   scope='bn')
  else:
    c = res
else:
  res = (sigmoid(f + self._forget_bias) *
     c_prev + sigmoid(i) *
      self._activation(j))
  if self._batch_norm:
    c = batch_norm(res,
                   center=True, scale=True,
                   is_training=self.
                       _is_training,
                   scope='bn')
  else:
    c = res

if self._cell_clip is not None:
  # pylint: disable=invalid-unary-operand-type
  c = clip_ops.clip_by_value(c, -self.
      _cell_clip, self._cell_clip)
  # pylint: enable=invalid-unary-operand-type
if self._use_peepholes:
  m = sigmoid(o + w_o_diag * c) * self.
      _activation(c)
else:
  m = sigmoid(o) * self._activation(c)

if self._num_proj is not None:
  with vs.variable_scope("projection") as
      proj_scope:
    if self._num_proj_shards is not None:
      proj_scope.set_partitioner(
          partitioned_variables.
              fixed_size_partitioner(
              self._num_proj_shards))
    m = _linear(m, self._num_proj, bias=False)

  if self._proj_clip is not None:
    # pylint: disable=invalid-unary-operand-
        type
    m = clip_ops.clip_by_value(m, -self.
        _proj_clip, self._proj_clip)
    # pylint: enable=invalid-unary-operand-
```

```
            type

    new_state = (LSTMStateTuple(c, m) if self.
        _state_is_tuple else
                 array_ops.concat([c, m], 1))
    return m, new_state

def _linear(args, output_size, bias, bias_start=0.0)
    :
  """Linear map: sum_i(args[i] * W[i]), where W[i]
     is a variable.
  Args:
    args: a 2D Tensor or a list of 2D, batch x n,
       Tensors.
    output_size: int, second dimension of W[i].
    bias: boolean, whether to add a bias term or not
       .
    bias_start: starting value to initialize the
       bias; 0 by default.
  Returns:
    A 2D Tensor with shape [batch x output_size]
       equal to
    sum_i(args[i] * W[i]), where W[i]s are newly
       created matrices.
  Raises:
    ValueError: if some of the arguments has
       unspecified or wrong shape.
  """
  if args is None or (nest.is_sequence(args) and not
      args):
    raise ValueError("`args` must be specified")
  if not nest.is_sequence(args):
    args = [args]

  # Calculate the total size of arguments on
     dimension 1.
  total_arg_size = 0
  shapes = [a.get_shape() for a in args]
  for shape in shapes:
    if shape.ndims != 2:
      raise ValueError("linear is expecting 2D
         arguments: %s" % shapes)
    if shape[1].value is None:
      raise ValueError("linear expects shape[1] to
         be provided for shape %s, "
                       "but saw %s" % (shape, shape
                          [1]))
    else:
      total_arg_size += shape[1].value

  dtype = [a.dtype for a in args][0]

  # Now the computation.
  scope = vs.get_variable_scope()
  with vs.variable_scope(scope) as outer_scope:
    weights = vs.get_variable(
        _WEIGHTS_VARIABLE_NAME, [total_arg_size,
           output_size], dtype=dtype)
    if len(args) == 1:
      res = math_ops.matmul(args[0], weights)
    else:
      res = math_ops.matmul(array_ops.concat(args,
         1), weights)
    if not bias:
      return res
    with vs.variable_scope(outer_scope) as
```

```
        inner_scope:
      inner_scope.set_partitioner(None)
      biases = vs.get_variable(
          _BIAS_VARIABLE_NAME, [output_size],
          dtype=dtype,
          initializer=init_ops.constant_initializer(
              bias_start, dtype=dtype))
    return nn_ops.bias_add(res, biases)
```

4 Batch-normalized GRU cell

```
"""
Implements a batch normalized GRU cell.
Unfortunately, there aren't any research papers that
    examine how to exactly implement this.

But most of it is based on https://arxiv.org/pdf
    /1510.01378.pdf
"""

from tensorflow.contrib.rnn import GRUCell

from tensorflow.python.framework import ops
from tensorflow.python.framework import tensor_shape
from tensorflow.python.framework import tensor_util
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import clip_ops
from tensorflow.python.ops import embedding_ops
from tensorflow.python.ops import init_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import nn_ops
from tensorflow.python.ops import
    partitioned_variables
from tensorflow.python.ops import random_ops
from tensorflow.python.ops import variable_scope as
    vs

from tensorflow.python.ops.math_ops import sigmoid
from tensorflow.python.ops.math_ops import tanh
from tensorflow.python.ops.rnn_cell_impl import
    _RNNCell as RNNCell

from tensorflow.python.platform import tf_logging as
    logging
from tensorflow.python.util import nest

from tensorflow.contrib.layers import batch_norm

import contextlib

_BIAS_VARIABLE_NAME = "biases"
_WEIGHTS_VARIABLE_NAME = "weights"

@contextlib.contextmanager
def _checked_scope(cell, scope, reuse=None, **kwargs
    ):
  if reuse is not None:
    kwargs["reuse"] = reuse
  with vs.variable_scope(scope, **kwargs) as
      checking_scope:
    scope_name = checking_scope.name
    if hasattr(cell, "_scope"):
      cell_scope = cell._scope  # pylint: disable=
          protected-access
      if cell_scope.name != checking_scope.name:
        raise ValueError(
```

```
            "Attempt to reuse RNNCell %s with a "
                "different variable scope than "
            "its first use.  First use of cell was "
                "with scope '%s', this "
            "attempt is with scope '%s'.  Please "
                "create a new instance of the "
            "cell if you would like it to use a "
                "different set of weights.  "
            "If before you were using: MultiRNNCell
                ([%s(...)] * num_layers), "
            "change to: MultiRNNCell([%s(...) for _
                in range(num_layers)]).  "
            "If before you were using the same cell "
                "instance as both the "
            "forward and reverse cell of a "
                "bidirectional RNN, simply create "
            "two instances (one for forward, one for
                reverse).  "
            "In May 2017, we will start "
                "transitioning this cell's behavior "
            "to use existing stored weights, if any,
                when it is called "
            "with scope=None (which can lead to "
                "silent model degradation, so "
            "this error will remain until then.)"
            % (cell, cell_scope.name, scope_name,
                type(cell).__name__,
                type(cell).__name__))
    else:
      weights_found = False
      try:
        with vs.variable_scope(checking_scope, reuse
            =True):
          vs.get_variable(_WEIGHTS_VARIABLE_NAME)
        weights_found = True
      except ValueError:
        pass
      if weights_found and reuse is None:
        raise ValueError(
            "Attempt to have a second RNNCell use "
                "the weights of a variable "
            "scope that already has weights: '%s'; "
                "and the cell was not "
            "constructed as %s(..., reuse=True).  "
            "To share the weights of an RNNCell, "
                "simply "
            "reuse it in your second calculation, or
                create a new one with "
            "the argument reuse=True." % (scope_name
                , type(cell).__name__))

      # Everything is OK.  Update the cell's scope and
          yield it.
      cell._scope = checking_scope  # pylint: disable=
          protected-access
      yield checking_scope


class BNGRUCell(GRUCell):
  """Gated Recurrent Unit cell (cf. http://arxiv.org
      /abs/1406.1078)."""

  def __init__(self, num_units, input_size=None,
      activation=tanh, is_training=True, batch_norm=
      True):
    self._is_training = is_training
```

```
    self._batch_norm = batch_norm

    super().__init__(num_units, input_size,
        activation)

  def __call__(self, inputs, state, scope=None):
    """Gated recurrent unit (GRU) with nunits cells.
       """
    with _checked_scope(self, scope or "gru_cell"):
      with vs.variable_scope("gates"):  # Reset gate
          and update gate.
        # We start with bias of 1.0 to not reset and
            not update.
        value = sigmoid(_linear(
          [inputs, state], 2 * self._num_units, True
              , 1.0))
        r, u = array_ops.split(
            value=value,
            num_or_size_splits=2,
            axis=1)
      with vs.variable_scope("candidate"):
        res = self._activation(_linear([inputs, r *
            state],
                                       self._num_units
                                           , True))

        if self._batch_norm:
          c = batch_norm(res,
                         center=True, scale=True,
                         is_training=self.
                             _is_training,
                         scope='bn1')
        else:
          c = res

        new_h = u * state + (1 - u) * c
      return new_h, new_h

def _linear(args, output_size, bias, bias_start=0.0)
    :
  """Linear map: sum_i(args[i] * W[i]), where W[i]
      is a variable.
  Args:
    args: a 2D Tensor or a list of 2D, batch x n,
        Tensors.
    output_size: int, second dimension of W[i].
    bias: boolean, whether to add a bias term or not
        .
    bias_start: starting value to initialize the
        bias; 0 by default.
  Returns:
    A 2D Tensor with shape [batch x output_size]
        equal to
    sum_i(args[i] * W[i]), where W[i]s are newly
        created matrices.
  Raises:
    ValueError: if some of the arguments has
        unspecified or wrong shape.
  """
  if args is None or (nest.is_sequence(args) and not
      args):
    raise ValueError("`args` must be specified")
  if not nest.is_sequence(args):
    args = [args]

  # Calculate the total size of arguments on
```

```
    dimension 1.
total_arg_size = 0
shapes = [a.get_shape() for a in args]
for shape in shapes:
  if shape.ndims != 2:
    raise ValueError("linear is expecting 2D
        arguments: %s" % shapes)
  if shape[1].value is None:
    raise ValueError("linear expects shape[1] to
        be provided for shape %s, "
                     "but saw %s" % (shape, shape
                         [1]))
  else:
    total_arg_size += shape[1].value

dtype = [a.dtype for a in args][0]

# Now the computation.
scope = vs.get_variable_scope()
with vs.variable_scope(scope) as outer_scope:
weights = vs.get_variable(
    _WEIGHTS_VARIABLE_NAME, [total_arg_size,
        output_size], dtype=dtype)
if len(args) == 1:
  res = math_ops.matmul(args[0], weights)
else:
  res = math_ops.matmul(array_ops.concat(args,
      1), weights)
if not bias:
  return res
with vs.variable_scope(outer_scope) as
    inner_scope:
  inner_scope.set_partitioner(None)
  biases = vs.get_variable(
      _BIAS_VARIABLE_NAME, [output_size],
      dtype=dtype,
      initializer=init_ops.constant_initializer(
          bias_start, dtype=dtype))
return nn_ops.bias_add(res, biases)
```