Using Machine Learning Models for Hashing

Anh Tran
University of Utah
anhb.tran@utah.edu

Samuel Gera
University of Utah
Sam.gera25@gmail.com

1 INTRODUCTION

In cryptography, hash functions are utilized for a lot of tasks such as integrity protection, signing, or storing password. Thus, being able to design a good hash function that is not only secure but also fast is crucial. As of 2020, the National Institute of Standards and Technology approved 17 hash functions [3]. While they are highly secure, one of the main downsides of these function is that the hashing process is public knowledge. Thus, with the ever increasing computing power and technology, it is only a matter of time before these secure hash functions are broken. Moreover, the architectures of these hash families are usually immutable, which reduce the robustness of these algorithms. As a result, it is crucial for researchers and industry leaders to come up with new hash functions that can easily be mass produced and customized according to the need of users while being secure and hard to break.

Machine learning (ML), especially Deep Neural Network (DNN), has been widely applied to multiple domains such as image recognition, natural language processing, autonomous driving, or financial data processing. Recently, researchers have started using ML algorithms for hashing. However, these hash functions are mainly used for dimensionality reduction and not for security purposes. However, one of the core features of an ML model is that it is a black box. Meaning, even with the knowledge of its architecture, it is extremely difficult to explain the behavior of the models, especially DNN. We could leverage this attribute and derive a hash function for security purposes.

There are several advantages of using a DNN model for hashing purposes. Firstly, as mentioned above, DNN models are a black box and their behavior are very unpredictable and hard to replicate and reverse. Secondly, DNN models can be constructed to be deterministic, which is crucial for a hash function to have the same output for the same input. Thirdly, DNN models can consist of a large number of parameters that will be infeasible to guess the set of weights by brute force method. Moreover, since each model is initialized differently and uses a different set of training data, even if an adversary were to acquire the model architecture, it would still be impossible for them to perfectly replicate a model. Finally, it is easy to customize and produce different models to fit different purposes.

In this project, we propose a novel approach to create a family of hash functions using a DNN. Our contribution is as follows:

- Successfully create a model that can hash any sized input to a predetermined output.
- Evaluate this model's hash capability based on collision rate, uniqueness and randomness. Our results indicate that using DNN models for secure hashing is possible.

2 RELATED WORK

Machine learning approaches for hashing techniques have received great attention from researchers. These DNN models are used to map higher dimension data to lower dimension data. Previous works [4, 12] used DNNs to converts high features data into its binary presentation. [2, 5, 8, 11] more specifically focused on using DNNs to hash an image while preserving its optimal features for large scale image searches. While works such as [1, 10] expanded this idea to a more general multimedia large scale database searches.

However, while these works are solid with great theoretical justification, they focus more on preserving the optimal features of the original input for searching purposes rather than for security reasons. In this project, we are more interested in the security aspect of hashing using DNN models and how we can use this in an adversary model.

3 ADVERSARY MODEL

3.1 Neural Net Hash Utility

Our proposed application of Neural Net Hashing, is simply that of an alternative to commonly used, but known, hash algorithms such as secure hashing algorithm (SHA) and message digest method 5 (MD5) (Still used, even through it's existent vulnerabilities). Due to SHA's great success, it is a commonly known and widely used algorithm. The attractiveness of Deep Neural Net Hashing is it's inherent dynamic aspect due to it's ability to change output dimensions, and unique hash output, depending on the model used to compute a hash, for any one piece of content.
Our proposed DNN hashes are a function of both the content to be digested, and the model used to compute the hash. Thus, we are able to generate a unique hash family based on a set of models, where each model offers the advantage of a new, secure, and unknown hash function, with the output of each model also introducing it's own deterministic unique output for any one piece of content (Assuming the unlikely event that the output collides, does not occur).

3.2 Neural Net Hash Security

With a good enough model and hashing pipeline, DNN Hashes offer the same security and utility of traditional hash functions used for integrity protection and message authentication:

(1) Given any length input, the hash function will always produce a fixed-length output.
(2) It is computationally infeasible to find a message given the message digest.
(3) It is computationally infeasible to find two message with the same message digest.
(4) The output should reflect changes in the input.
(5) A hash value's subset can also be used as a hash.

However, Neural Net hashing has the added benefit of being able to dynamically switch between hash functions h_i within a hash family H.

Now when considering an adversary who monitors traffic between two network nodes, it may be the case that they exchange similar/specific hashed messages during authentication or data transfer. Typically these exchanges introduce entropy using some random nonce R_i, s.t. these hashed messages could not be replayed or learned of. DNN hashing offers that same benefit, due to the ability of dynamically switching between hash functions within a hash family. Each message should have a unique hash considering it's computation using a different model. This is without the vulnerability of pseudo-random number generation if a sub-optimal random number generator is used, and it's seed discovered.

Furthermore, attempting to reconstruct the model in order to imitate produced hashes requires either knowledge of the hyper-parameters and data set corresponding to a particular model currently being used, or o.w. attempting to brute-force/reconstruct the weights of the given neural net. A quick calculation will show that ResNet50, the model we are planning to use, has over 23 millions trainable parameters. It is important to point out that these parameters will not be in a binary form, but in a floating point form. In other words, there are an infinite combinations of parameters. Thus, trying to guess the set of weights is impossible. In the case an attacker does somehow gain access to a DNN being used to compute hashes, our hash algorithm using the DNN still offers collision protection, and one-way properties of any typical hash function (they can not do much anyways). However, for this project, we did not implement such large model due to time constraint. We used a much simpler model but still maintain this security aspect.

3.3 Use in Traditional Client-Server setting

In order for two nodes to effectively use neural net hashing in order to dynamically switch between hash functions on a per-connection basis (or anything smaller scope), it requires that they have a set of pre-computed models $M_1, M_2, ..., M_n$, s.t. they generate a unique hash family H. This could be established through having a set of standards shared among a set of network nodes, or having some initiation message encrypted using RSA public key cryptography, where a set of data sets $D_1, D_2, ..., D_n$ and a set of hyper parameters $\lambda_1, \lambda_2, ..., \lambda_n$ are exchanged in order to establish a set of models $M_1, M_2, ..., M_n$ to serve as the unique hash family H.

After this initialization, whenever two nodes initialize ciphers for an encrypted connection, the extra step of choosing a common model could be used as well. This allows versatility in the size of the hash output (as different models can have different output spaces), as well as the hash timing (larger/smaller models may take varying time), and the benefit of different hash functions on the basis of per connection. Consider the following initialization for an SSL based protocol (Handshake phase) utilizing Neural Net Hashing as an example in figure 1 Note that in the example a MAC can still be generated by just repeatedly using the model to hash the output of a preliminary hash and prepend it with the content. This is because our hash input dimension is unbounded, despite the neural network input being bounded.

4 METHODOLOGY

4.1 Pipeline

The basic structure of the hash is as follow:

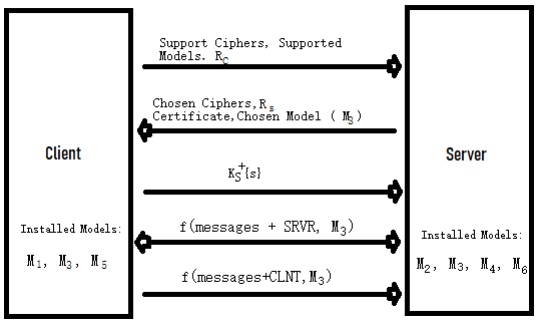

Figure 1: Where f is a function that produces a hash output given the content to be hashed, and model to use

(1) Pre-process: The document is vectorized into an array and is padded or broken into chunks depending on the feature size.
(2) Hash: The feature vector will be fed into the model and output a hash or multiple chunks of a hash.
(3) Post-process: If there are multiple chunks of a hash, they will be concatenated and reduced into the desired number of bits.

4.2 Pre-Processing

For pre-processing, we simply convert the input document into an array of 8 bit chunks. If the length of the array is less than the model's input dimension, then we pad it. If not, we will pad the array so that the length is divisible by the input dimension and then we break the array into chunks.

4.3 Model

Here, we describe the model architecture and training procedure.

4.3.1 Model architecture. Our model is a simple feed forward DNN with 4 hidden layers each with 512, 264, 128, 32 neurons respectively and a ReLU activation function [9] after every hidden layer. The ReLU activation function is computed as follows:

$$ReLU(x) = (x)^+ = max(0, x) \tag{1}$$

When hashing, given an input x, the set of weights from the model $\theta = [w_1, w_2, w_3, w_4, w_5]$, the hashing is computed as:

$$Hash(x) = w_5 \cdot ReLU(w_4 \cdot ReLU(w_3 \cdot ReLU(w_2 \cdot ReLU(w_1 \cdot x)))) \tag{2}$$

Since each weight w_i is a fixed real number matrix, the above equation is guaranteed to be deterministic.

Even though this model is simple, there are still roughly 700,000 parameters $\in \mathbb{R}$. It is infeasible for an adversary to guess the model's set of weights by any brute force method. Even if an adversary knows the model's architecture, it is still impossible to replicate the exact model without knowing the initial weights as well as the training procedure which includes data set randomization, training epochs randomization and training batch sampling.

4.3.2 Model training. There are two approaches to train this model. For both approaches, we generate the training input the same as we would do for testing. Explicitly, we randomized 50,000 sentences, pre-processed them and fed them into the DNN model. The only difference is that in the first approach, we randomize the targets or the labels for each sentences. In the second approach, we hashed each sentence using SHA-256 and used that as a label. The second approach performed better than the first one. We hypothesized that the random algorithm that we used is a pseudo random algorithm and thus introduced a lot of collision into the training set. While using SHA-256 guaranteed us an acceptable amount of randomness. After creating the training set, we simply trained the model for 50,000 epochs. During each training epoch, we sampled 256 data points from the training set to train the model.

4.4 Post-Processing

Deep neural nets having a fixed and specifiable output space is extremely useful for imitating that particular characteristic of a hash function. However, having a fixed input size for a deep neural net is a problem that immediately arises. If we have any document, message, or other content, bigger than our input space for our neural net, we have no way to singularly feed it into the model. As a result, the output will have shape chunk_size x output_dim.

To remedy this, we utilized Principal Component Analysis [7]. By stacking the numerous vectors containing real numbers' output from the model (vectors having as many rows as the output space) into a single matrix. We then extracted the first principal component of the matrix, and projected our data down to one single vector as the size of the output space, while retaining as much information as possible. This process is deterministic, so we will still retain the correlation between unique input corresponding with unique output.

To convert the array to the final hash, depending on our desired output shape, we combined the bytes of each element in the output array. For example, in this report, since we want a 32 bits output, we take the last 8 bits of each element from our 4 dimension output array and concatenate them together to get 32 bits.

5 EXPERIMENT

To test for collision rate and randomness, we ran one thousand trials. For each trial, we continuously generated random sentences and fed it into the model to collect the number of inputs it took to generate a collision in each trial to measure the bit entropy. To simplify the testing procedure, we chose the output dimension as 4. To produce the hash, after getting the output from the model, we took the last 8 bits from each element in the output array and concatenated them together to get a 32 bit hash. Thus, for each trial, we were only testing for 2^{16} inputs. The result for the collision test is reported in subsection 6.1 and for randomness is reported in subsection 6.2.

To test for uniqueness, we generated a few sentences that are very similar to each other and evaluated if the hash output from the model is distinguishable enough for each input. The result is reported in Figure 3.

6 RESULT

6.1 Collisions

Our results reported an average of 76209.2 trials before a signal collision was found. This was on an experiment run 1,000 times. Which equates to roughly 76 million hashes. This gave us a collision rate of $1.31218 * 10^{-5}$. A detailed result can be found in Figure 2

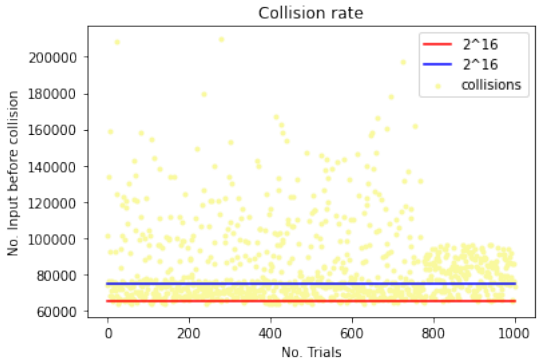

Figure 2: The collision rates of 1000 trials

6.2 Bit Entropy

In order to calculate the bit entropy of our output space corresponding to the neural net hash algorithm for the model we implemented in testing, we can use the following formula[6]

$$n \approx \frac{1-k}{ln(1-c/k)}$$

In the above, c correlates with the number of collisions found over k trials. Using our results from our collision testing, we get $n \approx 5.80738 * 10^9$. Which, when take the log_2 of, reports 32.43 bits of entropy. Which, considering we have an output space of 32 bits, is a rough overestimate. But, this does confirm the robustness of our DNN hash.

6.3 Uniqueness

We did not do extensive testing of uniqueness. We programmed a function to generate extremely similar input, and hashed over 2^16 trials for collision. We did not get any, we also recorded the hash for multiple different inputs, to observe the sensitivity of the hash and got promising results

Input Sentence	Output Hash
The quick brown fox jump over the fence	\x18\xe4 \xd2 \xdc
The quick brown fox jump over the fence!	\x9e \xdf0b
The quick brown fox jump over over the fence	\xc4d \xf9q
The quick brown fox jump over the fence??	\xf8* \xfb \x1e

Figure 3: Similar sentences, w/ corresponding hash codes

6.4 Discussion

Due to the time constraint, we were not able to perform more ablation tests. We are interested in whether or not an increase in

hidden layers or number of neurons per layer would improve the performance of the model or not. While having more layers might not greatly increase in security, it will increase in model's size and execution speed.

One thing that is certain while working on this project is the robustness of DNN for hashing purposes. Changing input or output dimensions is quite effortless. In addition, we could quickly produce multiple different models. Just by adding extra hidden layers or changing the number of neurons per layer will result in a new model.

One of our main concerns with using DNN is with the security aspect that we were not able to confirm due to time issue. Specifically, SHA 256 hash algorithm is a series of bits rotation, shift and XOR. With DNNs, it is strictly a mathematical operation. We are curious if given enough inputs and outputs, any other DNN model can approximate our model and break the algorithm. While this is very difficult because the final hash output is not the model output, it would still be an interesting experiment to perform.

7 CONCLUSIONS

In this report, we introduced a novel hash approach for security using DNN models. We were able to evaluate the model based on collision rate, randomness, and uniqueness. Our results demonstrate that DNNs can serve as a robust alternative to the current SHA family. However, more research will be needed before this method could be implemented in an industrial environment.

Furthermore, the implementation of DNN based hashing requires more resources and initialization than what it takes to use typical hashing methods. Thus, exploring research related to ways in which we can circumvent this requirement, or leverage cloud computing to address it, is paramount in it's application.

8 GITHUB

here

REFERENCES

[1] Zhangjie Cao, Mingsheng Long, Jianmin Wang, and Philip S. Yu. 2017. HashNet: Deep Learning to Hash by Continuation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[2] Miguel A Carreira-Perpinán and Ramin Raziperchikolaei. 2015. Hashing with binary autoencoders. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 557–566.

[3] Quynh H Dang et al. 2015. Secure hash standard. (2015).

[4] Thanh-Toan Do, Anh-Dzung Doan, and Ngai-Man Cheung. 2016. Learning to hash with binary deep neural network. In *European Conference on Computer Vision*. Springer, 219–234.

[5] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. 2015. Deep hashing for compact binary codes learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2475–2483.

[6] Aubrey Jaffer. 2013. Hash Entropy. (2013). https://people.csail.mit.edu/jaffer/III/Hash-Entropy

[7] Ian T Jolliffe and Jorge Cadima. 2016. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374, 2065 (2016), 20150202.

[8] Hanjiang Lai, Yan Pan, Ye Liu, and Shuicheng Yan. 2015. Simultaneous Feature Learning and Hash Coding With Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Jürgen Schmidhuber. 2014. Deep Learning in Neural Networks: An Overview. *CoRR* abs/1404.7828 (2014). arXiv:1404.7828 http://arxiv.org/abs/1404.7828

[10] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2012. Semi-supervised hashing for large-scale search. *IEEE transactions on pattern analysis and machine intelligence* 34, 12 (2012), 2393–2406.

[11] Rongkai Xia, Yan Pan, Hanjiang Lai, Cong Liu, and Shuicheng Yan. 2014. Supervised hashing for image retrieval via image representation learning. In *Twenty-eighth AAAI conference on artificial intelligence*.

[12] Ziming Zhang, Yuting Chen, and Venkatesh Saligrama. 2016. Efficient training of very deep neural networks for supervised hashing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1487–1495.