COMP480 Term Project Final Report

Edward Feng (jf44), Xinhao Liu (xl59)

Problem Description

As previously described in our checkpoint one and checkpoint two reports, the problem we want to solve is to use machine learning to create adversarial traffic such that we can cause a denial-of-service attack for a server that uses a hash table or bloom filter to check for malicious traffics. The basic background is that the server has a hash table or bloom filter that is keeping track of all of the known malicious traffic. Whenever the traffic sent to the server causes a cache hit, it will take the server a longer period of time than normal to respond because it needs to check whether the traffic is really malicious as hash table and bloom filter both have a non-zero false positive rates. Thus, to explore this problem, we trained a couple of neural network classifiers that can tell whether a certain traffic is likely to cause a cache hit or not in our target server's cache. We fed randomly generated traffics into the classifiers to collect those traffics that are labeled by the model as to be likely to cause cache hits, send them all to the server to see if they can cause more collision than completely random traffic. We started our experiment with a simple setup that the server's cache system is a hash table with two-universal hash function, which is also a bloom filter with only one hash function.

More specifically, to answer the question of whether we can train classifiers to generate filtered traffics that are better than random traffics at causing collisions, there are two fundamental questions that need to be answered first:

- Can machine learning models learn the pattern of a hash function? (Can a neural network predict the hash value h(x) given x?)
- Can machine learning models learn the pattern and distribution of data stored in a hash table? (Can it learn whether h(x) in the target hash table is a zero or a one?)

Thus, our specific goal here is to train a neural network that incorporates the above two steps into its model architecture. Our detailed experiments will be described later in this report. This is a brand-new topic that has rarely been touched in any previous research. To test whether we are capable of doing it, we found a couple of publications that somewhat relate to our goal and they will be described in the next section.

Literature survey

Hash functions are usually thought of as black boxes by application programmers since they seem to be random and safe. However, as pointed out in [1], the number and power of attacks on standard hash functions have been increasing. All attacking methods developed so far are collision-finding attacks. That is, trying to construct two messages that result in the same hash value. Since it is an inherently hard problem, other than trying to find exact collisions, researchers have also proposed concepts of pseudo-collision, free-start collision, and near-collision to relax the requirement of hashing collisions and develop attacking algorithms for these definitions of collisions. We now survey two of these methods in the literature.

Wang et al. [2] presented a differential attack using modular integer subtraction as the measure of difference to attack MD5, one of the most widely used cryptographic hash functions. Their method also applies to other simpler hash functions such as MD4, RIPEMD, and HAVAL. Specifically, they define a very complicated system of differentials between hash inputs and outputs of each iteration of the compression algorithm, and use bit-manipulation based message (hash input) modification techniques to search the input space twice for a pair of collision messages. The probability of finding the first message in collision is 2^{-37}, and the probability of finding the second message in collision is 2^{-30}. The complexity of their algorithm is 2^{39} MD5 operations.

Kelsey et al. [3] developed an attack on Damgard-Merkle construction, which is under the hood of many modern hash functions. Instead of only being able to find two messages that collide, their algorithm is able to generate a large number of messages that result in collisions. They call it *herding attack*. In the first phase of a herding attack, the attacker repeatedly applies regular collision-finding attacks against a hash function to build a diamond structure, which is essentially a Merkle tree of hashing inputs. In the second phase of the attack, the attacker exhaustively searches for a string which collides with one of the diamond structure's intermediate states, and builds a suffix in a complicated way from there.

However, all these methods are cryptographic algorithms and exploit the inherent mathematical weakness of hash functions. None of them is machine learning based. What we can find in machine learning literature are papers like [4]. Instead of breaking a hash function, the authors are trying to learn data-dependent hash functions that perform well in fast nearest neighbor searching. Similar works are [5], [6]. In general, the machine learning community is more concerned about constructing an efficient hash function for big data processing, rather than breaking one.

References

[1] M. Ilya, *Hash functions: Theory, Attacks, and Applications*. Microsoft Research, Silicon Valley Campus, Nov. 14, 2005.

[2] Wang, Xiaoyun, and Hongbo Yu, How to break MD5 and other hash functions. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg. 2005.

[3] Kelsey, John, and Tadayoshi Kohno. Herding hash functions and the Nostradamus attack. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg. 2006.

[4] X. Li, G. Lin, C. Shen, A. van den Hengel, and A. Dick. Learning hash functions using column generation. *Proc. ICML*. 2013.

[5] Kumar, Shaishav, and Raghavendra Udupa. Learning hash functions for cross-view similarity search. *22nd International Joint Conference on Artificial Intelligence*. 2011.

[6] K. Lin, H. F. Yang, J. H. Hsiao, and C. S. Chen. Deep learning of binary hash codes for fast image retrieval. *Computer Vision and Pattern Recognition Workshops (CVPRW).* 2015.

Hypothesis

We hypothesize that by using supervised learning with labeled data points, a neural network can be trained to learn the pattern of data in the corresponding hash table.

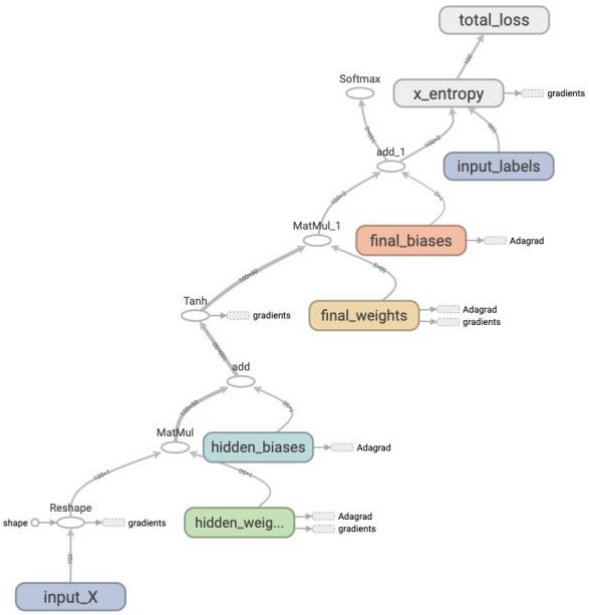
Experimental Settings

Background and setup:

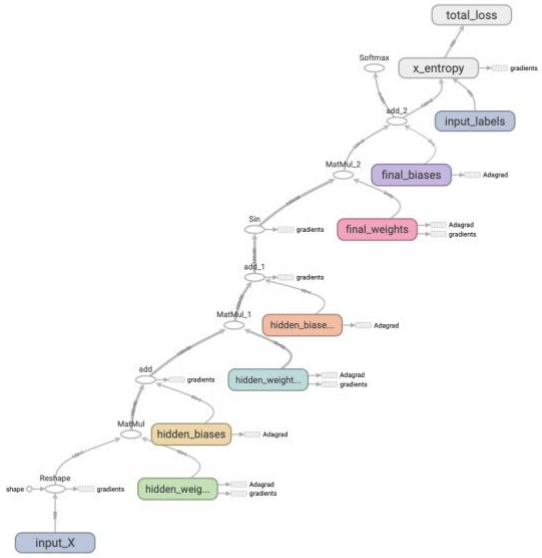
- To simplify the problem for experiment purpose, we used integers to simulate the network traffic being sent through the network. Each individual data packet is represented by one integer.
- For this experiment, we used a simple hash function that is $h(x) = (ax + b) \% TABLE\_SIZE$, where for all of the experiments I conducted, a = 37, b = 47, TABLE_SIZE = 10000.
- To load the table with caches, I randomly sampled 0.65 * TABLE_SIZE integers from 0 to 5 * TABLE_SIZE and inserted them into the hash table. Due to hash collision, there are approximately 0.5 * TABLE_SIZE buckets that are marked as 1 after the insertion.
- We used a similar technique to generate training and testing data. Specifically, we randomly sampled TABLE_SIZE * 2 integers from 0 to 3 * TABLE_SIZE and labeled them 1 or 0 based on whether they cause a cache hit or cache miss in the table we just loaded. After removing duplicates, there are approximately the same amount of cache hit data and cache miss data. Then I split the data in a 4:1 ratio for training and testing purpose.
- Our goal is to let the model correctly tell us whether a given integer should be labeled as 0 or 1.
- All of the parameters above (table size, hash function, table load, amount of training and testing data) can be specified by manually changing the parameters in the source code and the detailed instruction will be provided at the end of this document.

Machine Learning Models:

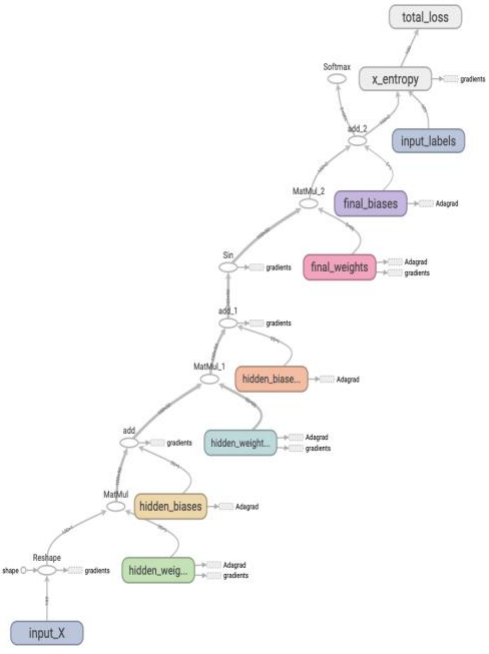
- The model architectures we used are simple feed forward neural networks with one input node, various number of hidden layers of 50 nodes and 2 output nodes.
- For the hidden layers, we varied the choice of activation functions including tanh, relu, sine (the reason why we used sine will be explained later), and plain feedforward without activation.
- We calculated the loss by calculating the cross entropy and we measure the prediction with a softmax layer at the end.
- Then we used the adagrad optimizer provided by TensorFlow which automatically adapts the learning rate to the parameters, we set the initial learning rate to 0.01
- We ran 15000 training iterations with 100 data as a batch for each iteration, each batch is randomly drawn from the whole training dataset.
- Below are the detailed description and visualization of the different model architectures we tried (visualization from TensorBoard):



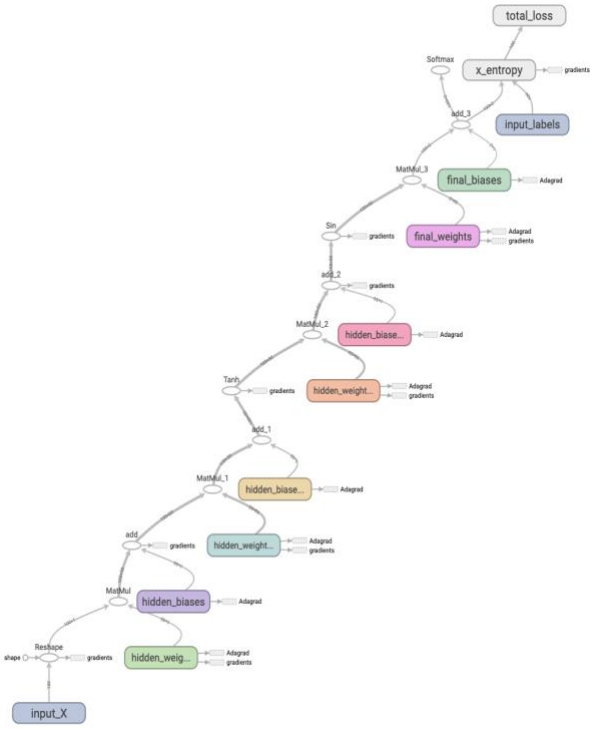
1 hidden layer with tanh activation



2 hidden layers with tanh+sine activations



2 hidden layers with plain+sine activations



3 hidden layers with plain+tanh+sine activations

As shown above in the visualization, the models basically follow the same pattern besides that the number of hidden layers may vary and the activation functions after the matmul and add operation are different. We've also tried other architectures such as three hidden layers with tanh+sine+relu activations, tanh+tanh+sine activations, etc.

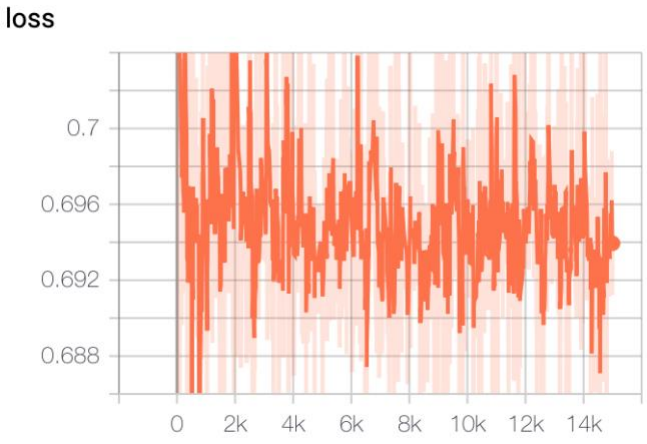
It is worth noticing that we used sine as the activation function for one of the hidden layers for almost every model, which is rarely seen in other machine learning problems since sine is a periodic function unlike commonly used activation functions such as tanh, relu, sigmoid. The reason we used sine is that in a two universal hash function, there's a mod operation at the end. As stated in the problem description, one of our fundamental goal is to learn the pattern of a hash function, and to come up with a model that can imitate the effect of a mod operation, we need to have something that is periodic. As shown below, this is the Fourier transformation of t mod theta, which also involves a sine function to imitate the mod operation.

$$f(t) = \frac{\theta}{2} - \frac{\theta}{\pi} \sum_{n=1}^{N} \frac{1}{n} \sin \frac{2\pi n t}{\theta}$$

Instead of varying the model architectures, such as the number of hidden layers and activation functions, I also tuned different parameters such as initial learning rate, number of iterations, and size of training data set.

Experimental Results

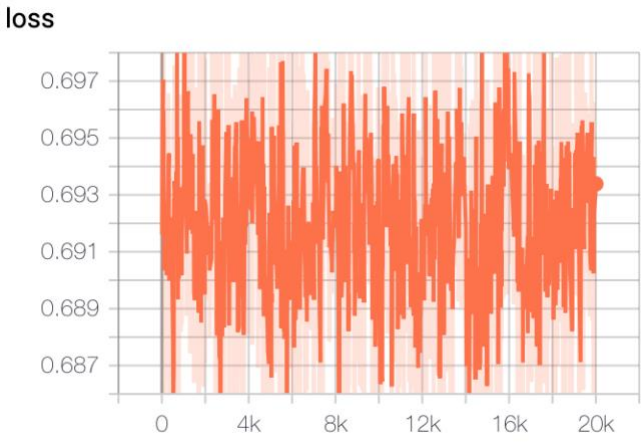
- For each model architecture we tried, as we've tried running 10000, 15000, 20000 iterations, the total loss in each iteration doesn't vary much and there's no obvious tendency of loss getting lower along number of iterations.
- For all different models the loss is very stable and fluctuates at around 0.695.
- The number of correct labels for each batch is fluctuating at around 50-55 out of 100.
- For different models with different parameters, the rate of correct labeling on the test set is roughly 50% across each run, which means the model couldn't really learn the features of the hash table and the effect is basically the same as random guessing. The specific analysis on the results will be provided at the end of this section.
- Below are graph visualizations of loss versus number of iterations from TensorBoard:



2 hidden layers, tanh + sine activations
15,000 iterations
Initial learning rate = 0.1
Batch size = 100

Number of training data: 11,463 (roughly 1:1 split on positive and negative)

Testing: 1512 out of 2911 correct (roughly 50%)



3 hidden layers, tanh + sine + relu
20,000 iterations
Initial learning rate = 0.05
Batch size = 100

Number of training data: 22,880 (roughly 1:1 split on positive and negative)

Testing: 2933 out of 5515 correct (roughly 50%)

Basically, all the other combinations of parameters give us similar results: all of the other loss versus iteration graphs show that loss is not being able to get lower across iterations, and correctness ratio on testing set is about 50%. Below is a table of the testing correctness ratio of some representive runs with different models and different parameters.

Model	Training size	Iterations	Batch size	Correctness ratio
plain + sine	11,877	10,000	100	51.2%
tanh + sine	34,427	15,000	200	53.3%
plain + sine + relu	10,976	20,000	100	50.6%
tanh + sine + relu	22,880	15,000	200	49.5%
plain + sine	22,182	15,000	200	51.0%
tanh + sine	33,215	20,000	100	49.9%
plain + sine + relu	22,124	15,000	200	50.3 %
tanh + sine + relu	22,756	10,000	200	53.1%
plain + sine	22,338	20,000	200	52.2%
tanh + sine	10,979	20,000	100	52.6%
plain + sine + relu	21,981	15,000	200	53.0%
tanh + sine + relu	22,883	15,000	100	51.0%

(Training size is not always exactly the same for different runs because when we're randomly generating numbers there are some duplicates and they're removed)

It can be observed that different combinations always give us a correctness ratio of around 50%. Thus, even if we use the classifier to collect and send the traffics it thinks are more likely to cause cache hits on the server, the result is still the same as sending purely random traffic.

Summary and Conclusion:

Based on the experiment conducted, it can be concluded that machine learning models didn't really learn the pattern of our hash table. And unfortunately, we have to reject our hypothesis mentioned earlier. To our current understanding, we address this failure by bringing it back to the two fundamental questions we mentioned at the beginning of this report. The first question is can machine learning learn the pattern of a hash function (can it predict h(x) given x); the second question is can machine learning learn the data distribution in a hash table (given h(x), can it predict whether h(x) is a 1 or 0). The first one is very challenging as it involves the mod operation, where we imitated its behavior in our model by using the sine activation function. However, the key to training machine learning models is gradient descent, where each weight is updated iteratively according to its gradient, but since we cannot take the gradient of a mod operation, the model can be hard to learn and that's part of the reason why in the TensorBoard graph we never see the loss actually getting lower. The second question is also difficult to solve because the distribution of data in the server's hash table can never be known unless we know the hash function it is using. And the distribution of data in the hash table is basically random, which doesn't have any mathematical pattern. However, the nature of machine learning is to come up with some mathematical equation that can generalize the distribution of the data, and since the data here doesn't follow any mathematical pattern, the distribution of data in the hash table can be extremely difficult to model using machine learning. Overall, our experiments showed that these two fundamental challenges are very difficult to solve and we have to reject our hypothesis based on our experiments.

Code Documentations

In the submitted zip file, there are python files of the major model architectures we have experimented with and the model architectures are consistent with their file names. To run the python code, your computer needs to have tensorflow and numpy installed. To tune the parameter, simply change the global variables on top of the python file to desired values. These parameters include batch size, number of hidden units, number of training iterations, number of training data to generate, and table size. If you want to experiment with different hash functions, simply go change the function h(x) on line 48.

To visualize graphs, use command "tensorboard –logdir=/Path/To/TermProjectCode/logs/model". For example "tensorboard –logdir=/User/Desktop/TermProjectCode/logs/tanh"
Then use your browser to open the specified page to visualize the graph of the model and the loss vs. iterations plot.