Can an Interpretable Network be Undetectably Backdoored?

Matjaž Leonardis

Department of Computer Science
University of Texas at Austin
matjaz@cs.utexas.edu

Abstract

Recently, many schemes have been proposed for "backdooring" neural network models. Apart from their relevance to computer security and AI safety they are also related to questions about the limits of interpretability of machine learning models. Intuitively, interpretability of machine learning models and detectability of backdoors should go hand in hand. In this work, we present a very simple network that can perfectly perform a classification task on a given dataset and analyze whether it can be undetectably backdoored. We show the network achieves its classification effectiveness by 'memorizing' the dataset, despite the fact the dataset contains of $O(nd)$ parameters (n data points, each a d-dimensional vector) and the network can be described by only $O(n + d)$ parameters. Moreover, despite being interpretable we argue the network can still be undetectably backdoored, unless one has full knowledge of the dataset. Even in cases where the backdoor can be detected not much can be learned about the inputs the attacker can use to trigger it.

1 Introduction

In recent years, many schemes have been proposed for "backdooring" neural networks. Some of these schemes have focused on "poisoning" the dataset and examining whether subsequent safety training can remove the backdoors[6][3]. Others have focused on modifying the weights in the first and final layers of the models to achieve surprising effects[10]. Still others have raised questions about whether backdoors can be inserted in a way where their detection is as hard as breaking known cryptographic protocols[5]. Proposals for how they could be used in the watermarking of model outputs have also been made[1].

A mathematical definition that would unify all these schemes and attack models is yet to emerge, however what they all share in common is that at the end of the backdooring process the client is in possession of model that performs as expected and looks as though it was constructed in the way it was supposed to while the attacker retains the ability to prompt the model on inputs where the network behaves in undesirable ways.[4]

Apart from being relevant to the security of machine learning models and AI safety, backdoors also share an interesting connection with interpretability of machine learning models. Intuitively, if we had full mechanistic ability to understand the models, this ability should also enable us to detect the backdoors contained in them. Conversely, if backdoors can be inserted in a way where they cannot be detected, that suggests a limit to our ability to interpret machine learning models.

It is this relationship that we aim to further explore in this paper. We present a very simple neural network and examine the ways in which it can be backdoored. The network is perhaps the simplest kind of network that can achieve perfect results (arbitrarily small error) on any training set. It achieves this by "memorizing" the dataset in a way that uses fewer parameters than are present in the dataset itself. Because of its simplicity, the network is interpretable and its outputs on every input can be attributed to data points in the training dataset. (There are strong reasons to believe that this might be possible even for the most advanced models.[9]) Surprisingly, the network also generalizes well on classification tasks where nearest-neighbor approach achieves good generalization. In spite of this, the network can be backdoored in a way that is undetectable unless the client is in possession of the entire dataset. Even when the backdoor can be detected, nothing can be learned about the inputs the attacker might use to activate it (except for a range of values the "hashes" of the attacker's inputs would have to take). This, counterintuitively, suggests that models easiest to undetectably backdoor are those that resemble a look-up table rather than those that embody a more complex logic when processing their inputs.

2 The Model

Suppose we have a training dataset for some classification machine learning task. It consists of n data points $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ each a d dimensional vector that is associated with a classification $c_1, c_2, ..., c_n$. The classifications take values from 1 to c, the total number of classes. For such a dataset, we construct a neural network model that performs with arbitrarily small loss when tested on inputs from this dataset.

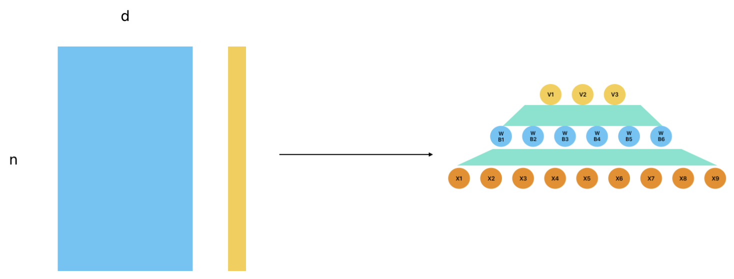

Figure 1: From dataset to model

The model itself is a two-layer network. The network works like a typical neural network with the added restriction that all of the neurons in the first layer apply an identical set of weights $w = (w_1, w_2, ..., w_d)$ to the input (they differ only in their biases). The exact number of neurons in the first layer m is variable and depends on the dataset we're creating the network for. However as we shall see $m \leq n$. For the sake of simplicity we assume $b_1 \leq b_2 \leq ... \leq b_m$. Each neuron in the last layer is associated with a set of m weights $v_i = (v_1^i, v_2^i, ..., v_m^i)$. It merely takes the scalar product of its weights with the outputs of the preceding layer and doesn't add any biases nor apply any non-linearities. The remaining details of the network are unimportant but for the sake of definiteness we assume that the non-linearity used in the first layer is the sigmoid function and that the outputs of the final layer are combined using softmax to obtain probabilities for each classification.

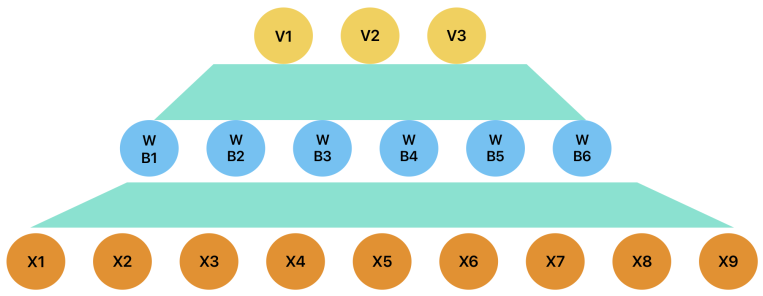

Figure 2: Basic two-layer network model achieving dataset memorization. The values in this example are $d = 9, c = 3, m = 6$. The values in the first layer represent the components of a single data point, not multiple data points

Formally, the outputs of the neurons in the first layer $l_1^1, l_2^1, ..., l_m^1$ are given by:

$$l_i^1 = \sigma(w \cdot x - b_i)$$

We subtract the bias b_i rather than add it for clarity of presentation. It allows us to interpret b_i as the threshold the product $w \cdot x$ needs to exceed to activate the neuron.

Denoting by $l^1 = (l_1^1, l_2^1, ..., l_m^1)$ the combined outputs of the first layer, the outputs of the second layer are given by:

$$l_i^2 = v_i \cdot l^1$$

Finally, the probability that the given input has classification i is denoted by p_i and is given by:

$$p_i = \frac{e^{l_i^2}}{\sum_{j=1}^{c} e^{l_j^2}}$$

We shall show how to construct such a network for a given dataset in Section 4. However, there are already reasons to find it surprising that this type of network can achieve a perfect score on any given dataset. Intuitively, one might think that a network that is capable of scoring perfectly on the dataset would have to "memorize" it. Yet, the dataset consists of $O(nd)$ parameters, yet this network can be effectively described by merely $O(n + d)$ parameters. Although strictly speaking the network contains $m(d + c + 1)$ parameters it can be effectively described using only $d + 2m$ parameters. We need to remember the d parameters that represent w, a further m parameters for the biases of the neurons in the first layer and as we shall see a further m classification values to construct the weights in the final layer. This is so despite the fact that no structural assumptions on the nature of the dataset have been made.

How can "memorization" with so few parameters be achieved? As we shall see, the way this is done is key to allowing us to backdoor the network. If the network was just a straightforward memorization of the dataset one would expect to be able to recover the entire dataset from the network and thereby also detect any backdoors that might have been added. Instead, as we shall see, there is no way to recover the data points $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ that were used to construct the model, although the model still correctly classifies them when presented with one.

To explain the model we first explain how the relevant representation of the dataset can be obtained outside the neural network context and then explain how it is embodied in this specific neural network. We then explain why this network can also generalize well and finally analyze how to backdoor it.

3 How Does the Network "Memorize" the Dataset?

Suppose we have a dictionary: a large set of key-value pairs. It is useful to imagine the key sizes as large, while the set of possible values that can be associated with them as being small. This is also a feature of many machine learning classification tasks, where the key might be an image consisting of millions of pixels that is then classified in one of a few thousand categories.

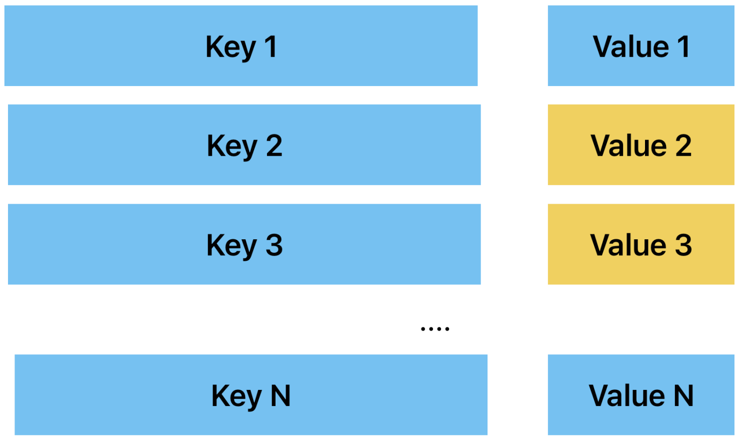

Figure 3: A dictionary

One could store this dictionary in memory by storing both keys and values, perhaps with additional tree-like indices. However for many applications we might never need to check which keys are present in the dictionary, merely returning the right value when presented with a key. This is the case with machine learning classifiers. One needn't be able to reproduce the images the classifier was trained on, merely to correctly classify them if presented with them again. So instead of storing the keys we can apply some hash function to them and only store the hashes of the keys together with the values associated with them.

To find the value associated with a candidate key, we compute its hash and check if it is present in our list to find the relevant value.

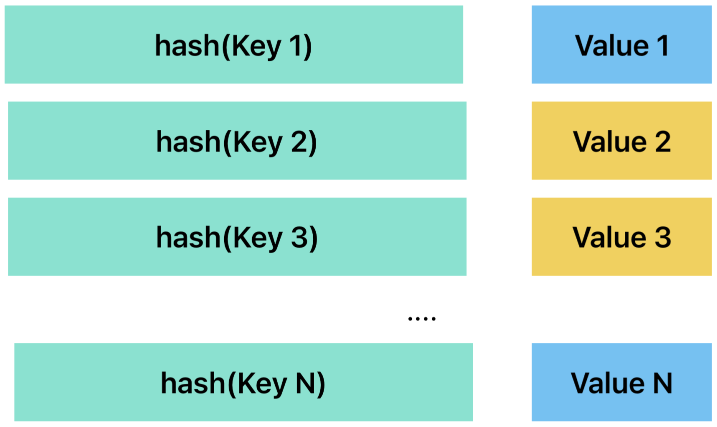

Figure 4: A dictionary where we store the hashes of the keys

But we can go a step further - because the number of possible values that can be associated with each key is small, we don't even need to remember all the hashes. Rather, we can sort the hashes to obtain a sorted list of hashes $H_1, H_2, ..., H_N$. Suppose now that hashes H_i and H_{i+1} are the first pair of consecutive hashes associated with different values. We remember some value $H_i < H_1^* < H_{i+1}$ that is to be taken as a signal that hash values higher than H_1^* are no longer to be associated with the value associated with H_i but with the value of H_{i+1} instead. We compute all such "division points" for every pair of consecutive hashes associated with different values. In doing so we obtain a sequence of values $H_1^* < H_2^* < ... < H_m^*$.

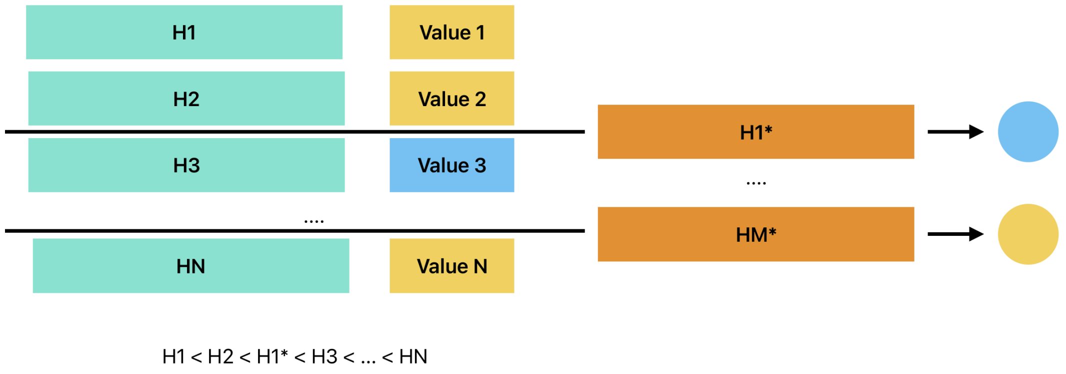

Figure 5: Simplified representation of range-based classification

To find the value associated with a given key we compute its hash value H and then find the largest number in our sequence H_i^* such that $H_i^* < H$. The value associated with that transition is the value associated with our key.

Put another way, rather than storing the hashes we can store a number of buckets where all of the hash values in a given bucket are classified in the same way.

Note also that we have in effect "generalized" our dictionary. If presented with a key that was not originally present in the dictionary this procedure will return a value for it. The value returned will match the value of one of the two keys in the original dataset that neighbor the given one in hash values.

This is how the dataset ends up being represented in the neural network. The network will then contain the representation of the hash function and the number of division points will correspond to the number of neurons in the first layer. We turn to exact construction next.

4 Constructing the Network: Slicing the Dataset

One further geometric analogy might be useful to understand how this model is obtained. The data set can be imagined as a cloud of points somewhere in a high-dimensional space.

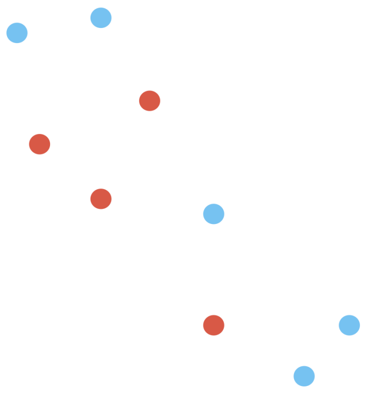

Figure 6: Data Cloud

We start with a hyperplane, with all of the points on one side of it at the beginning. We will be slicing the dataset by making cuts parallel to this plane. We can always pick a plane such that no two points belonging to two different classes are ever on the same plane parallel to the initially chosen one. Moreover, we can always cut the cloud of points ensuring that only points belonging to the same class are ever in the same slice. The following diagram illustrates this idea:

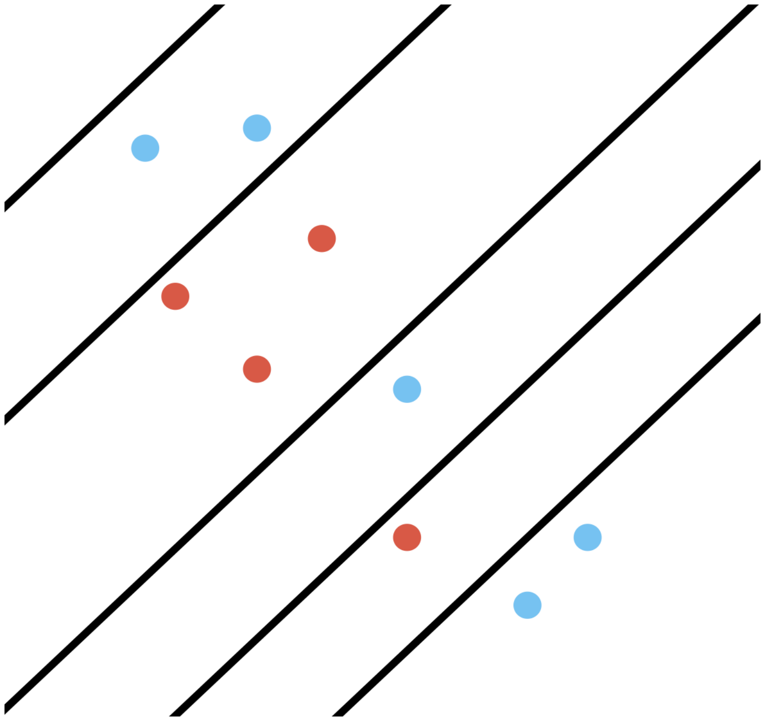

Figure 7: Geometric slicing analogy: hyperplanes intersecting data cloud

The way we move from this representation to the neural network model is as follows. The perpendicular direction to the hyperplane corresponds to w, the weights of the neurons in the first layer. The biases $b_1, b_2, ..., b_m$ of those neurons correspond to the division points where the cuts we have made intersect with the perpendicular line. The projections of the points on the same line

$\mathbf{x}_1 \cdot w, \mathbf{x}_2 \cdot w, ..., \mathbf{x}_n \cdot w$ are the "hashes" of individual data points in the spirit of the discussion from the previous section. This diagram illustrates this idea:

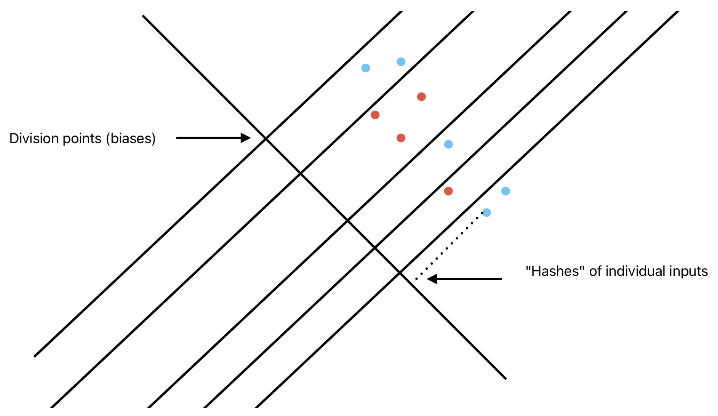

Figure 8: Projection of data points onto a shared direction vector

Here is how we compute all of the weights and biases in the first layer. Starting with the dataset $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ we pick any vector of weights w, ensuring only that $\mathbf{x}_i \cdot w \neq \mathbf{x}_j \cdot w$ for all i, j where the classifications associated with $\mathbf{x}_i$ and $\mathbf{x}_j$ are different. An infinite number of vectors w meets this condition and can even be found with random sampling.

Next, we compute the values $\mathbf{x}_1 \cdot w, \mathbf{x}_2 \cdot w, ..., \mathbf{x}_n \cdot w$ and sort them, obtaining a list $h_1 < h_2 < ... < h_n$. For all i where the classifications associated with h_i and h_{i+1} are different we add a neuron to the first layer with weights w and a bias b such that $h_i < b < h_{i+1}$. Any bias satisfying this constraint works, although $b = \frac{h_i + h_{i+1}}{2}$ is perhaps the natural choice. We also add a neuron with a bias $b < h_0$ at the beginning of the sequence to ensure that at least one neuron in the first layer is activated for all points in the dataset. The significance of this will be shown when we construct the final layer. This completes the construction of the first layer.

If we compare this construction with the more general description from the previous section it is clear that the data points play the role of keys, their classifications the role of values, the scalar product with the weights $\cdot w$ the role of the hash function and the biases of the neurons in the first layer the role of division points.

To sum up the comparison of the two pictures in a table:

Table 1: Comparison of the two pictures

General picture	Neural network picture
Keys	Data points (x_i)
Values	Classifications (c_i)
Hash function	Scalar product with weights of neurons in the first layer (w)
Hashes of keys	Products $\mathbf{x}_i \cdot w$
Transition hash values	Biases of the neurons in the first layer (b_i)

Next, we determine the weights in the final layer. For every data point, the activations of neurons in the first layer take approximately the following form 111111...1110000...000. In words, it is some sequence of activations very close to 1 followed by a sequence of activations very close to 0. They can be made to come arbitrarily close to this sequence by scaling the weights and biases of the first layer by a factor with no effect on the geometry of the overall construction. If a point is located in a particular "slice" then the neurons associated with this and all of the preceding slices will activate while the neurons associated with the upcoming slices will not activate. For each neuron in the last layer there are some slices that ought to be classified as belonging to it and the remaining ones which are not. Accordingly we want the output of the neuron to be high when the point is located in the former and we want the output value of the neuron to be very low (highly negative) for the latter. The final weights associated with this particular neuron can be determined by solving a system of equations where L is a large positive value and s is a small (highly negative) value. Since this system is not singular such values can always be found.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \cdot v_i = \begin{bmatrix} L \\ s \\ s \\ \vdots \\ L \end{bmatrix}$$

Solving this system for each output node ensures correct classifications on the given dataset. The output of the neuron associated with the right classification will be very high while the outputs of the remaining ones will be very low. It can be immediately verified this model can achieve arbitrarily small error on the dataset by appropriately scaling the weights of the model and the values L and s.

5 Can this Type of Network also Generalize well?

Despite the fact that the network is only designed to perform well on the given dataset, experiments show that for some machine learning tasks constructing such a network with respect to the training set results in one that also performs well on the test set. That is, a network prepared in this way performs well even on instances it hasn't seen. One such task is the image classification task on the standard MNIST dataset. (The details of the experiment are described below).

This might intuitively seem surprising. Continuing with the hashing analogy, if the hash function used was a cryptographic one would not expect that the hash of a data point from a particular class would have a bias to be close to other values of the same class. But the hash function $\cdot w$ is what is in the literature[2][7][8] referred to as a *locality-sensitive hash function.* This means that when two points are close, their hash values will also tend to be close and if the two points are far apart, their hash values will only be close with a low probability. In fact the difference in the hash values of two data points is bounded by their distance in the following sense. Suppose we have two points $\mathbf{x}_1$ and $\mathbf{x}_2$. Then the hash values associated with each are $H_1 = w \cdot \mathbf{x}_1$ and $H_2 = w \cdot \mathbf{x}_2$. Then by Cauchy-Schwartz inequality we have:

$$|H_1 - H_2| = |w \cdot (\mathbf{x}_1 - \mathbf{x}_2)| \leq ||w|| \cdot ||\mathbf{x}_1 - \mathbf{x}_2||$$

This means that we would expect such networks to perform better than random on tasks where the nearest-neighbor approach also performs well. As the following image illustrates, only points located in the strip can result in a point being classified differently than its nearest neighbor.

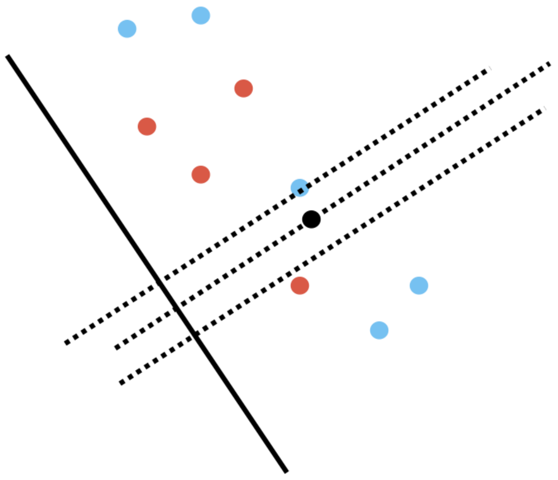

Figure 9: Only a point between the doted lines could cause a different answer than nearest neighbor

We demonstrate this in the case of the MNIST dataset by comparing three different approaches - nearest-neighbor, hash-bucket model described in this paper and what we'd expect if the hash function was cryptographic. We describe the first two approaches.

MNIST dataset consists of $60,000$ training examples and of $10,000$ test images. Each image is made out of 28 by 28 single byte pixels (a total of 784). We compare the approaches on the full set of 10 different classes as well as all 45 pairs of binary classification problems that can be constructed from it.

The nearest-neighbor approach performs no training. Rather it takes the training set as a given and then for each instance in the test set computes the distance relative to all points in the training set. The distance measure used is the standard Euclidean distance. That is, if $a_1, ..., a_{784}$ and $b_1, b_2, ..., b_{784}$ represent the pixel values of the two images, then the Euclidean distance is given by $\sum_{i=1}^{784}(a_i - b_i)^2$. This approach attributes to the test image the same classification as the image in the training set closest to it.

For the hash-bucket test we pick 1000 values of w at random, construct the model we described in this paper for each (we pick the division points in the middle of two hashes with different classifications) and check how they perform on the test set. Each parameter of w is sampled form a normal distribution with mean 0 and variance 1. We record the best result obtained in this way.

The nearest-neighbor approach achieves a 97% accuracy on the full MNIST digit classification dataset and performs with even higher levels of accuracy on the binary problems. The hash-bucket approach achieves 18% accuracy on the full dataset which is much worse but still significantly better than random. On binary problems this approach yields accuracies form $98.1\% - 69.2\%$, in all cases better than random. In fact, the theory of locality-sensitve hash functions was developed because of the challenge of scaling nearest-neighbor to large datasets which normally involved computing the level of similarity with every item in the dataset whenever we made a query[2][7]. Locality-sensitive hash functions are useful for narrowing the search. In effect, what the network described in this paper does is classify the inputs based on similarity defined by a single locality-sensitive hash function which explains the results.

The results are described in more detail in the following table:

Table 2: Summary of experimental results

	Nearest-neighbor	Hash-buckets neural network random w	Random expected
Full MNIST dataset	96.9 %	18.3 %	10 %
0 vs 1	99.9 %	98.1 %	50 %
0 vs 2	99.5 %	85.9 %	50 %
0 vs 3	99.8 %	84.0 %	50 %
0 vs 4	99.9 %	87.5 %	50 %
0 vs 5	99.8 %	88.6 %	50 %
0 vs 6	99.4 %	86.2 %	50 %
0 vs 7	99.9 %	87.1 %	50 %
0 vs 8	99.4 %	85.2 %	50 %
0 vs 9	99.6 %	90.1 %	50 %
1 vs 2	99.6 %	87.5 %	50 %
1 vs 3	99.8 %	90.7 %	50 %
1 vs 4	99.6 %	91.6 %	50 %
1 vs 5	99.9 %	86.4 %	50 %
1 vs 6	99.8 %	92.9 %	50 %
1 vs 7	99.3 %	88.9 %	50 %
1 vs 8	99.9 %	86.4 %	50 %
1 vs 9	99.7 %	92.4 %	50 %
2 vs 3	99.7 %	79.5 %	50 %
2 vs 4	99.9 %	80.4 %	50 %
2 vs 5	100.0 %	76.3 %	50 %
2 vs 6	99.7 %	78.8 %	50 %
2 vs 7	98.7 %	78.0 %	50 %
2 vs 8	99.4 %	71.7 %	50 %
2 vs 9	99.8 %	85.1 %	50 %
3 vs 4	99.9 %	82.0 %	50 %
3 vs 5	98.0 %	69.6 %	50 %
3 vs 6	99.9 %	85.7 %	50 %
3 vs 7	99.4 %	86.6 %	50 %
3 vs 8	98.5 %	72.0 %	50 %
3 vs 9	99.1 %	82.8 %	50 %
4 vs 5	99.8 %	79.7 %	50 %
4 vs 6	99.7 %	82.8 %	50 %
4 vs 7	99.3 %	77.5 %	50 %
4 vs 8	99.6 %	79.4 %	50 %
4 vs 9	98.3 %	69.2 %	50 %
5 vs 6	99.4 %	78.9 %	50 %
5 vs 7	99.9 %	80.8 %	50 %
5 vs 8	98.6 %	69.9 %	50 %
5 vs 9	99.3 %	76.4 %	50 %
6 vs 7	99.9 %	88.0 %	50 %
6 vs 8	99.7 %	82.5 %	50 %
6 vs 9	99.9 %	86.1 %	50 %
7 vs 8	99.7 %	84.8 %	50 %
7 vs 9	98.8 %	70.4 %	50 %
8 vs 9	99.4 %	79.4 %	50 %

6 Backdooring the Network

The notion of a backdoor is somewhat nebulous and makes most sense in the context of a real system which has been modified to contain a mechanism that allows the attacker to influence its behavior while remaining imperceptible to the end user.

Since we don't make any structural assumptions on the nature of the dataset in this paper we take the meaning of backdoor to be that the attacker is able to modify the behavior of the model on one chosen input $\mathbf{x}^*$ and examine whether such a modification can be detected in black-box and white-box settings. In the white-box context we examine whether the end user can distinguish a non-backdoored model from a backdoored one.

Let's suppose that there is some target input $\mathbf{x}^*$, outside of the dataset that we wish to classify in an incorrect way. A useful way of imagining this is to think of $\mathbf{x}^*$ as an image of a digit for which many similar digits are present in the dataset that are correctly classified.

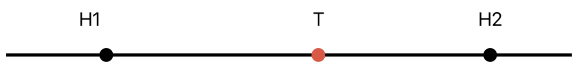

Figure 10: Target input

Let us refer to the scalar product $b^* = w \cdot \mathbf{x}^*$. The hash b^* will generally be located between two division points $b_i < b^* < b_{i+1}$ that correspond to a certain classification, assumed to be different than the one we want to assign to $\mathbf{x}^*$. We backdoor the model by partitioning this bucket by inserting two new division points $b_1^* = b^* - \epsilon$ and $b_2^* = b^* + \epsilon$ where we can make $\epsilon > 0$ as small as we wish. This will ensure that the backdoor works while the only inputs where the network behaves differently are those with hashes very close to those of b^* - the behavior of the network will change on a negligibly small fraction of the inputs. This requires us to insert two new nodes in the first layer and recompute the final layer.

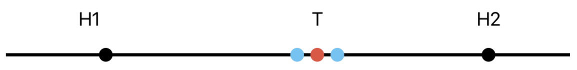

Figure 11: Backdoor

We now turn to the question of whether such a backdoor can be detected. Because the behavior of the network will change on only an arbitrarily small fraction of the inputs, trying to detect it in a black-box fashion will be infeasible. In the context of white-box testing the client is in effect dealing with a large number of meaningless division points and the hash function that is being used. The division points don't themselves correspond to meaningful inputs nor can meaningful inputs be constructed from them. What the client can do is try validating the various ranges. Of particular interest are those that are very short, but experiments show short ranges routinely arise even in non-backdoored models. Client testing can only vindicate a range but can't show that a given range corresponds to a backdoor.The client could remove unvalidated ranges but in doing so risks severely diminishing the performance of the model.

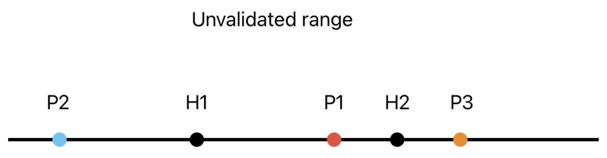

Figure 12: Hashes that could account of a range

Finally we consider if the client has access both to the full model as well as the dataset the model was supposed to be constructed from. If the model is assumed to have a single backdoor, then so long as the client is missing as few as three inputs it is conceivable that any range that hasn't been accounted for is the result of the missing data points. Depending on what restrictions have been made on the way division points have been inserted missing as few as a single input could make the backdoor remain plausibly to be a valid range.

If the client has access to the full dataset then he can detect the presence of the backdoor (ranges without any inputs from the training set). However he can't detect what the triggering input x^* is. He only learns the range $[b^* - \epsilon, b^* + \epsilon]$ from which x^* can't be directly reconstructed. Additional knowledge about the structure of the dataset could reveal more information about x^* but it depends on the specifics of the machine learning task.

7 Conclusion

Although the model is simple it highlights many of the important considerations for thinking about backdoors and AI safety in more complex models.

Firstly, many neural network models beyond this simple model can be thought of as being approximations of the Voroni diagram that nearest-neighbor classification would produce and can be analyzed in similar ways. For example, one might approach backdooring two-layer models where no restrictions exist on the weights of the neurons in the first layer in a similar way. A similar analysis could also be done on models that involve embedding spaces. It indicates that when exploring the backdooring of more complex models attention should turn to their special architectural features where it has presently not been.

Secondly, it shows that while many papers that consider detecting and defending against backdoors focus on black-box testing or alternatively white-box scrutinizing of the weights of the model the most useful tool in defense against backdoors is actually the knowledge of the training set.

That might be known to the user but in the case of most models in use today remains obscure (even the types of datasets that were used in supervised fine-tuning remain largely obscure, even for open-source models). A useful direction for exploration might be what is the least amount of information that needs to be provided about the dataset that could be useful in finding the backdoors as well as help in assessing the models' capabilities.

Finally, it shows that the presence of backdoors can be a useful guide to understanding the nature of knowledge embodied in the model. While it may at first seem counterintuitive, models that embody general principles should be harder to undetectably backdoor than those that resemble a hashed version of the training set. Newton's laws are harder to backdoor than a look-up table with thousands of entries. It shows that the most interesting questions surrounding backdoors and their detectability and removability only make sense in the context of models that embody some kind of deeper knowledge.

References

[1] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: watermarking deep neural networks by backdooring. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1615–1631, USA, 2018. USENIX Association.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[3] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning, 2017.

[4] Paul Christiano, Jacob Hilton, Victor Lecomte, and Mark Xu. Backdoor defense, learnability and obfuscation. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.

[5] Shafi Goldwasser, Michael P. Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models, 2024.

[6] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M. Ziegler, Tim Maxwell, Newton Cheng, Adam Jermyn, Amanda Askell, Ansh Radhakrishnan, Cem Anil, David Duvenaud, Deep Ganguli, Fazl Barez, Jack Clark, Kamal Ndousse, Kshitij Sachan, Michael Sellitto, Mrinank Sharma, Nova DasSarma, Roger Grosse, Shauna Kravec, Yuntao Bai, Zachary Witten, Marina Favaro, Jan Brauner, Holden Karnofsky, Paul Christiano, Samuel R. Bowman, Logan Graham, Jared Kaplan, Sören Mindermann, Ryan Greenblatt, Buck Shlegeris, Nicholas Schiefer, and Ethan Perez. Sleeper agents: Training deceptive llms that persist through safety training, 2024.

[7] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA, 1998. Association for Computing Machinery.

[8] Anand. Rajaraman and Jeffrey D. Ullman. *Mining of massive datasets.* Cambridge University Press, New York, N.Y., 2012.

[9] Sheng-Yu Wang, Aaron Hertzmann, Alexei A. Efros, Jun-Yan Zhu, and Richard Zhang. Data attribution for text-to-image models by unlearning synthesized images, 2025.

[10] Irad Zehavi, Roee Nitzan, and Adi Shamir. Facial misrecognition systems: Simple weight manipulations force dnns to err only on specific persons, 2024.