Learning and Memorization

Satrajit Chatterjee [1]

Abstract

In the machine learning research community, it is generally believed that there is a tension between memorization and generalization. In this work, we examine to what extent this tension exists, by exploring if it is possible to generalize by memorizing alone. Although direct memorization with a lookup table obviously does not generalize, we find that introducing depth in the form of a network of support-limited lookup tables leads to generalization that is significantly above chance and closer to those obtained by standard learning algorithms on several tasks derived from MNIST and CIFAR-10. Furthermore, we demonstrate through a series of empirical results that our approach allows for a smooth tradeoff between memorization and generalization and exhibits some of the most salient characteristics of neural networks: depth improves performance; random data can be memorized and yet there is generalization on real data; and memorizing random data is harder in a certain sense than memorizing real data. The extreme simplicity of the algorithm and potential connections with generalization theory point to several interesting directions for future research.

1. Introduction

Neural networks trained through stochastic gradient descent (SGD) can memorize their training data. Although practitioners have long been aware of this phenomenon, Zhang et al. (2017) recently brought attention to it by showing that standard SGD-based training on AlexNet gets close to zero training error on a modification of the ImageNet dataset even when the labels are randomly permuted. This leads to an interesting question: *If neural nets have sufficient capacity to memorize random training sets why do they generalize on real data?* A natural hypothesis is that nets behave differently on real data than on random data. Arpit et al. (2017) study this question experimentally and show that there are apparent differences in behavior. They conclude that generalization and memorization depend not just on the network architecture and optimization procedure but on the dataset itself.

But what if networks fundamentally do *not* behave differently on real data than on random data, and, in both cases, are simply memorizing? This is a difficult question to explore for two reasons. First, it is hard to provide a direct answer. Whereas it is easy to tell when a net is memorizing random data (the training error goes to zero!), there is no easy way to tell when a network is memorizing real data as opposed to "learning". Second, and perhaps more importantly, it contradicts the intuitive notion—inherent in the preceding discussion—that memorization and generalization are at odds. This work attempts to shed light on this second difficulty by investigating the following: *How much can you learn if memorization is all you can do? Is generalization even possible in this setting?*

At first, generalization in such a setting of pure memorization may seem hopeless: the simplest way to memorize would be to build a lookup table from the training data. Although this approach works for special cases where the input population is finite and small, it fails in general since the examples seen during training are unlikely to match test examples. One way to get around this limitation is to use k-Nearest Neighbors (k-NN) or any of its variants at test time. While k-NNs work well on many problems, they fail on problems where it is not easy to construct a semantically meaningful distance function on the input space. In such cases, the obvious syntactic distance functions (e.g., say Euclidean distance between images viewed as vectors in $\mathbb{R}^d$) do not work well. Indeed some of the most interesting results from deep learning have been the discovery—through learning—of semantically meaningful distance functions (via embeddings).

Therefore, in this work we do not allow ourselves a distance function. Instead, we get around the problem by applying the notion of depth, which has been wildly successful in improving the performance of neural networks, to direct memorization. We build a network of lookup tables (also

[1]Two Sigma, New York, NY, USA. Correspondence to: Satrajit Chatterjee <satrajit.chatterjee@twosigma.com>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

called "luts") where the luts are arranged in successive layers much like a neural network. However, unlike a neural network, training happens through memorization and does not involve backpropagation, gradient descent, or any explicit search. Now, since in contrast to a neuron, the function implemented by a lut can be arbitrarily complex, without some means to control the complexity, the notion of depth is vacuous. We control the complexity of a function learned by a lut in the simplest possible way: we limit the support (and thereby the size) of the lut. Each lut in a layer receives inputs from only a few luts in the previous layer, which are picked at random when the network is constructed. This kind of restriction on local function complexity is similar to what is found to work well in deep neural networks. For example, a convolutional filter is obviously support-limited, and a fully connected layer although not support-limited is nevertheless limited in expressivity. Furthermore, the learned weight matrices in neural networks are often sparse or can be made so with no loss in accuracy (Han et al., 2015).

We need two restrictions before we can proceed to an algorithm. First, for simplicity, we focus our attention on binary classification problems. Second, because lookup tables work naturally with discrete inputs, in this work we limit ourselves to discrete signals. In fact, the inputs and all intermediate signals in the network of lookup tables are binary. The restriction is not as extreme as it may appear. There are a number of results in quantized and binary neural networks showing that limited precision is often sufficient (e.g. Rastegari et al., 2016). Furthermore, even in real-valued neural networks, we need mechanisms such as convolution and pooling to ensure that certain types of small changes in the inputs (e.g., a small displacement) do not lead to large changes in output. In principle, similar mechanisms could be used in a fully discrete setting to handle real-valued quantities.

With these restrictions in place, we are now ready to proceed.

2. A Single Lookup Table

Let $\mathbb{B} = \{0, 1\}$ and consider the problem of learning a function $f : \mathbb{B}^k \to \mathbb{B}$ from a list of training examples where each example is an (x, y) pair. Since we want to learn by memorizing, we construct a lookup table with 2^k rows (one for each possible bit pattern $p \in \mathbb{B}^k$ that can appear at the input) and two columns y^0 and y^1. The y^0 entry for the row corresponding to pattern p (denoted by c_{p0}) counts how many times p is associated with output 0 in the training set, i.e., the number of occurrences of $(p, 0)$ in the training set. Similarly, the y^1 entry for row p (denoted by c_{p1}) counts how many times the pattern p is associated with the output 1 in the training set, i.e., the number of occurrences of $(p, 1)$ in the training set. Note that for a pattern p it is possible for both c_{p0} and c_{p1} to be greater than zero since due to Bayes error both $(p, 0)$ and $(p, 1)$ may be present in the training examples. It is also possible for both c_{p0} and c_{p1} to be zero if the input p never appears in the training examples. We call such a lookup table a k-input lookup table or a k-lut since the inputs are bit vectors of length k.[1]

Next, we associate a boolean function $\hat{f} : \mathbb{B}^k \to \mathbb{B}$ with the lookup table in the following manner:

$$\hat{f}(p) = \begin{cases} 1 & \text{if } c_{p1} > c_{p0}, \\ 0 & \text{if } c_{p1} < c_{p0}, \\ b & \text{if } c_{p1} = c_{p0} \end{cases}$$

where $b \in \mathbb{B}$ is picked uniformly at random when fixing $\hat{f}$ in order to break ties. In other words, $\hat{f}$ maps an input p to the output that is most often associated with it in the training set (breaking ties randomly). We say that $\hat{f}$ is the function learned by the lookup table.

Example 1. Let $k = 3$ and consider learning a function $f : \mathbb{B}^3 \to \mathbb{B}$ from 7 examples shown on the left below. The lookup table that we learn is shown in the middle, and the truth table of the learned function $\hat{f}$ is shown on the right. The entries in the truth table which have been picked randomly to break ties are indicated by an asterisk.

x	y
$x_0x_1x_2$	
000	0
000	1
000	1
001	1
100	0
110	0
110	1

p	y^0	y^1
$x_0x_1x_2$		
000	1	2
001	0	1
010	0	0
011	0	0
100	1	0
101	0	0
110	1	1
111	0	0

p	$\hat{f}$
000	1
001	1
010	0^*
011	1^*
100	0
101	1^*
110	1^*
111	0^*

Note that $\hat{f}$ gets all training examples correct except for the first and sixth. This is the best we can do on this set of training examples because the Bayes error rate is non-zero. □

If we measure training error as the average 0–1 loss on the training set, this procedure to learn $\hat{f}$ has the following properties:

1. **Optimality.** The learned function $\hat{f}$ is Bayes-optimal on the training set, i.e., there is no function $g : \mathbb{B}^k \to \mathbb{B}$ with training error strictly less than that of $\hat{f}$. In particular, the training error is zero iff the training set has zero Bayes error.
2. **Monotonicity.** If we have more information for each x in the training set, i.e., we augment each training

[1] Typically k is small (less than 10) and so the the table can be stored explicitly. The input bit vector (viewed as an integer) can be used to directly index into the table.

example with m extra bits of information (keeping the labels fixed) and use the above procedure to now learn a new function $\hat{g} : \mathbb{B}^{k+m} \to \mathbb{B}$, then the training error of $\hat{g}$ is no more than that of $\hat{f}$.

Proof Sketch. Optimality is easy to see since the total training error is the sum of the training error for each possible pattern p which is minimized by choosing the majority class for each p. Monotonicity holds since if not, then we can compose the obvious projection $\mathbb{B}^{k+m} \to \mathbb{B}^k$ with $\hat{f}$ to get a contradiction with the optimality of $\hat{g}$. □

Note that monotonicity implies in particular that the training accuracy at the output of a lut is no worse than that at any of its inputs. Furthermore, as we make the luts larger, the training error cannot increase but only decrease. This is interesting since there are no restrictions on the m extra bits: they could be completely non-informative. These properties will prove useful in the next section as we consider networks of luts.

To summarize, the procedure described to learn a single lookup table in this section is essentially memorization in the presence of Bayes error, where the idea is to simply remember the output that is most commonly associated with an input in the training set.

3. A Network of Lookup Tables

Now consider a binary classification task on MNIST (LeCun & Cortes, 2010) of separating the digits '0' through '4' (we map these to the 0 class) from the digits '5' through '9' (the 1 class) where the pixels are 1-bit quantized. Thus the task is to learn a function $f : \mathbb{B}^{28\times 28} \to \mathbb{B}$. We call this the *Binary*-MNIST task (overloading binary here to mean both binary classification and binary inputs).

In principle, we could use the procedure in Section 2 to learn this function. However, since we have only 60,000 training examples in MNIST, most of the $2^{28\times 28}$ rows in the lookup table would have 0 entries in both columns, and hence the function learned would be mostly random and have very poor generalization to inputs outside the training set.

As discussed in the introduction, we get around this problem by introducing depth. Instead of learning a giant lookup table with $2^{28\times 28}$ entries, we learn a network of (much) smaller lookup tables. The network consists of d layers with each layer l ($1 \le l \le d$) having n_l k-input lookup tables. Each lut in first layer ($l = 1$) receives its inputs from a k-random subset of the network inputs. A lut in a layer $l > 1$ receives inputs from a k-random subset of the luts in layer $l - 1$. The connectivity is fixed at network creation time and does not change during training or inference. The final layer of the network has a single lookup table (i.e., $n_d = 1$) which is the output of the network. By analogy with neural networks, we call the final layer the *output layer* and the other layers *hidden layers*.

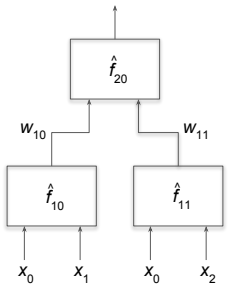

Figure 1. The network from Example 2.

We train the lookup tables layer by layer, where the target of *each* lookup table is the final output. We start from the first layer and work our way to the output. Once a layer has been learned, we use the functions associated with its luts (the $\hat{f}$s of Section 2) to map its inputs to outputs. These outputs serve as the inputs for the next layer, which is learned next. Continuing our analogy with neural networks, we call the output values of a layer *activations*.

Inference is similar to training: We start from the inputs and evaluate each layer in order using the functions learned at each lut to map inputs to outputs.

Example 2. We modify Example 1. Instead of learning a single lut with $k = 3$ inputs, we learn a network of $k = 2$ luts. The network shown in Figure 1 has $d = 2$ layers. The first layer has 2 luts (i.e., $n_1 = 2$) which are connected to inputs x_0 and x_1 of the network. The second layer (which is also the output layer) has 1 lut (i.e., $n_2 = 1$) which is connected to the outputs of the two luts in the first layer. (The connections were made randomly when the network was created.) Using the procedure in Section 2, the two lookup tables learned in the first layer (using y as the target) along with their corresponding functions $\hat{f}_{10}$ and $\hat{f}_{11}$ are:

$\frac{p}{x_0x_1}$	y^0	y^1	$\hat{f}_{10}$
00	1	3	1
01	0	0	1*
10	1	0	0
11	1	1	1*

$\frac{p}{x_0x_2}$	y^0	y^1	$\hat{f}_{11}$
00	1	2	1
01	0	1	1
10	2	1	0
11	0	0	1*

Let the output of the luts in the first layer be w_{10} and w_{11}, i.e., $w_{10} = \hat{f}_{10}(x_0x_1)$ and $w_{11} = \hat{f}_{11}(x_0x_2)$. The learning problem for the lut in the second layer is shown in the tables below. For convenience, on the left we show the primary inputs x_0, x_1 and x_2, the first layer activations w_{10} and w_{11} (which are the inputs of the lut), and the target output y. On the right we show the table and the learned function $\hat{f}_{20}$:

x / $x_0x_1x_2$	$w_{10}w_{11}$	y
000	11	0
000	11	1
000	11	1
001	11	1
100	00	0
110	10	0
110	10	1

p / $w_{10}w_{11}$	y^0	y^1	$\hat{f}_{20}$
00	1	0	0
01	0	0	1^*
10	1	1	0^*
11	1	3	1

In this case the function implemented by the network of 2-luts has the same performance on the training set as the function learned by the 3-lut in Example 1. Since there are fewer possible patterns in the case of smaller luts, we expect better pattern coverage during training and hence better generalization. □

Implementation. The memorization procedure described here is linear in the size of the training data, requiring two passes over the training set. It is computationally efficient since it only involves counting and dense table lookups and does not require floating point. It is also easy to parallelize since each lut in a given layer is independent, and the counts can be computed on disjoint subsets of the training data and then combined (using, for example, a reduction tree). Note that using this property it is possible to execute the algorithm on extremely large datasets where all the training examples may not fit on a single machine with only the summary statistics of the data (the counts in the lookup tables) being exchanged across machines.

4. Experiments

Experiment 1. In the first experiment, we apply the above procedure to the Binary-MNIST task (as defined in Section 3) to see if this approach to memorization can generalize. For this experiment, we construct a network with 5 hidden layers of 1024 luts and 1 lut in the output layer. We set $k = 8$, i.e., each lut in the network takes 8 inputs.

The network achieves a training accuracy of 0.89 on this task, which is perhaps not so surprising since we are memorizing the training data after all. But what is surprising is that the network achieves an accuracy of 0.87 on a held-out set (the 10,000 test images in MNIST) which indicates generalization.

This result is not state-of-the-art on this variant of MNIST (see Experiment 4), but that is not the point. It is significantly above the 0.5 accuracy that would be expected by chance, and this is achieved by an algorithm that only memorizes and performs no explicit search.

The training and test accuracies are stable: there is very little variation from run to run. In other words, very little depends on the actual random choices made when deciding the topology of the network. To understand why this is the case, we look at training accuracies of the luts in the network. Since the target for each lut in the network is the final classification target, we can examine the accuracy of a lut as a function of its layer.

Table 1. Layer by layer training accuracy of network of 8-input lookup tables on Binary-MNIST. Note that the statistics for layer 0 do not correspond to luts but to the 28×28 inputs.

LAYER	LUT COUNT	TRAINING ACCURACY MEAN	STD	MIN	MAX
0	784	0.5072	0.0340	0.4042	0.6572
1	1024	0.6055	0.0403	0.5120	0.7299
2	1024	0.7431	0.0191	0.6721	0.7877
3	1024	0.8297	0.0068	0.8038	0.8526
4	1024	0.8655	0.0033	0.8562	0.8751
5	1024	0.8808	0.0015	0.8759	0.8853
6	1	0.8898	0.0000	0.8898	0.8898

Table 1 shows the summary statistics for the accuracies of luts in each layer. We observe that as depth increases the average accuracy of the luts in a layer goes up. In other words, depth helps. Some intuition for this is provided by the monotonicity property of the luts: the output of a lut cannot have lower accuracy than any of its inputs (Section 2).

Furthermore, we observe in Table 1 the dispersion in accuracy across the luts (measured either by standard deviation (std) or the difference between max and min) goes down. Therefore, as depth increases the specifics of the connectivity matters less and the network automatically becomes more stable with respect to the random choices made during construction. Indeed we can say something stronger: we have seen in our experiments (not shown in Table 1) that as depth increases the activations of the luts in a layer become more correlated with each other, and hence become more interchangeable. While this correlation is good for stability with respect to connectivity, it causes diminishing returns with additional depth.

Remark. The perceptive reader looking at Table 1 will also notice that we are wasting computation: the single output lut in layer 6 receives input from only 8 of the 1024 luts in layer 5 and these in turn can at most receive inputs from 64 luts from layer 4. Although a different topology would be more computationally efficient, this specific choice allows us to compare the different layers more easily. We have not optimized this aspect since it typically takes less than 30 seconds using a single threaded unoptimized implementation (Python with NumPy) to run an experiment.

Experiment 2. As discussed in the introduction and in Section 3, we do not expect unbridled memorization in the form of a large lookup table (say $k = 28 \times 28$ in the case of Binary-MNIST) to generalize at all. This motivated our

Table 2. Effect of varying lookup table size on Binary-MNIST.

k	ACCURACY			
	ON REAL DATA		ON RANDOM DATA	
	TRAINING	TEST	TRAINING	TEST
2	0.66	0.66	0.51	0.53
4	0.81	0.81	0.53	0.54
6	0.85	0.86	0.55	0.52
8	0.89	0.87	0.60	0.51
10	0.94	0.89	0.69	0.50
12	0.99	0.90	0.82	0.51
14	1.00	0.83	0.92	0.51
16	1.00	0.66	0.98	0.52

Table 3. Performance of some standard methods along with memorization and random guessing on the Binary-MNIST task. Note that unlike the others CONV NET is not permutation invariant.

METHOD	ACCURACY	
	TRAINING	TEST
CONV NET	0.98	0.98
5-NEAREST NEIGHBORS	0.99	0.97
1-NEAREST NEIGHBOR	1.00	0.97
RANDOM FOREST (10 TREES)	1.00	0.96
MEMORIZATION	**0.99**	**0.90**
LOGISTIC REGRESSION	0.87	0.87
NAÏVE BAYES	0.76	0.77
RANDOM GUESS	0.50	0.50

exploration of a network of smaller lookup tables parameterized by k (the number of inputs of each lut). We now vary k to see if we can control the amount of memorization and to see the effect it has on generalization. To avoid changing too much at once, we keep the number of layers and the number of luts per layer the same as in Experiment 1.

The results are shown in the first 3 columns of Table 2. With small values of k, the network finds it difficult to memorize the training data. As intuitively expected (see also the monotonicity property in Section 2), as k increases the training accuracy goes up with perfect memorization at $k = 14$, i.e., long before 28×28. However, larger luts generalize less well, and the best test accuracy of 0.90 is achieved at $k = 12$ though with substantially good memorization of the training data (0.99). Interestingly, there is a clear monotonic increase in the generalization gap measured as the difference between training and test accuracy with increasing k.

Experiment 3. In this experiment—along the lines of those performed in Zhang et al. (2017)—we randomly permute the labels in the training set and repeat Experiment 2 on this "random" dataset. The results are shown in columns 4 and 5 of Table 2. As expected, with increasing k the network gets better at memorizing the training data, and the test accuracy hovers around chance (0.5) though with significant variation ($\pm$ 0.05). This may be viewed as empirical evidence that the Rademacher complexity goes up with k.

However, and this may be surprising for a pure memorization algorithm, memorizing random data turns out to be *harder* than memorizing real data (columns 2 and 3 of Table 2) in the sense that a larger k is required to get the same accuracy with random data than with real data. For example, it takes until $k = 12$ to get comparable training accuracy on random data as $k = 4$ gets on real data. This result corroborates the findings in Arpit et al. (2017, §3 and §4) that real data is easier to fit than random data. *But it also means that we cannot conclude that any such difference observed in neural networks is because they do not use brute force memorization on real data.* As this experiment shows, such differences can appear even with brute force memorization.

Finally, at $k = 12$ we have a network that is able to memorize random data (random training accuracy of 0.82) and yet generalizes to test data when trained on real data (real test accuracy of 0.90). This is very similar to findings of Zhang et al. (2017) in the context of neural networks. Kawaguchi et al. (2017, §3) argue that this phenomenon is universal and our result may be viewed as further empirical evidence for their claim showing that this phenomenon can happen even in the simplified setting of just memorization.

Experiment 4. For completeness, we compare memorization with several standard methods and the results are shown in Table 3. We have not specifically tuned the other methods since our goal is not to beat the state-of-the-art but to get a sense of how memorization alone does when compared to the standard methods. The best performance is obtained by a LENET-style convolutional network with 2 convolutions (64 and 32 filters respectively) each followed by a corresponding max pool layer, and 3 fully connected layers (256, 128 and 2 units respectively) with softmax output. The net is trained for 6 epochs with stochastic gradient descent and dropout.

Once again, compared to random guessing which has 0.50 test accuracy, memorization does quite well with a test accuracy of 0.90 (using the $k = 12$ configuration from Experiment 2) and beats logistic regression and naïve Bayes. Interestingly, 1- and 5-Nearest Neighbors do well too (test accuracy of 0.97) though recall that they are provided with a distance function which memorization does not have access to and must in a sense discover.

Experiment 5. We now consider the task of separating the i-th digit in MNIST from the j-th digit, which gives us $\binom{10}{2} = 45$ binary classification tasks, which we collectively call Pairwise-MNIST. The images are binarized as before.

Figure 2 shows the training accuracy and the test accuracy for each of those 45 experiments for 8 different values of k.

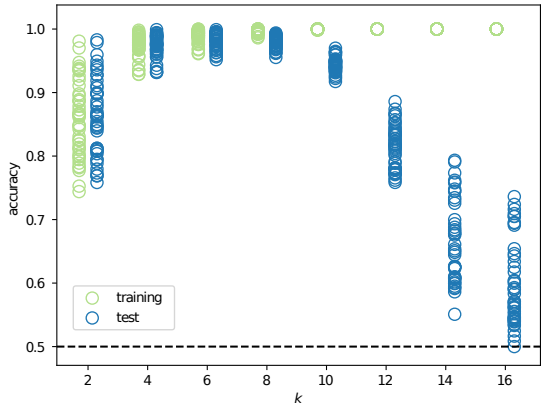

Figure 2. 45×8 pairs of training and test accuracies as a function of the size of the lookup table (k) for Pairwise-MNIST.

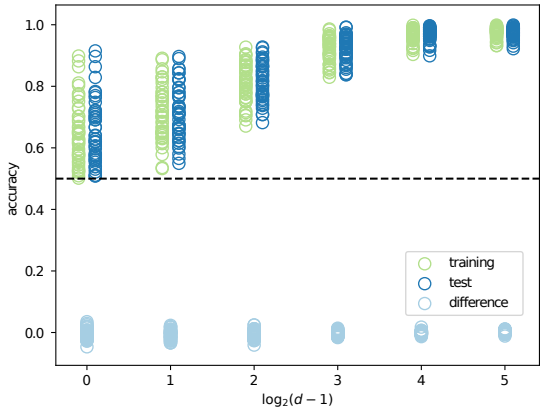

Figure 3. The effect of depth (d) on networks with small lookup tables ($k = 2$) for Pairwise-MNIST.

As in Experiment 2, we find that as k increases, the training accuracy increases (reaching 1.0), but the test accuracy falls off. If we look at the best test accuracies for a given task (across k), on 31 out of the 45 tasks, we do better than 0.98. The worst of these is 0.95 which is the best memorization can do for separating '4' and '9'. This is still significantly better than the 0.5 we would expect by chance. Typically the best test accuracies are achieved at $k = 6$ and $k = 8$.

Experiment 6. In Experiment 5 we notice that the variation is quite high for $k = 2$. This indicates that the depth of the network is insufficient for proper mixing. To investigate this further, we keep $k = 2$ and vary the number of hidden layers from 2^0 to 2^5. Each hidden layer still has 1024 luts. Figure 3 shows how the training and test accuracies vary with the depth of the network. It is interesting to note that the test accuracy continues to improve even for relatively deep networks (16 or 32 hidden layers), and we get very high test accuracies even with such small lookup tables. Furthermore, we note that the variation in the generalization error (difference between training and test accuracies) decreases with increasing depth.

Experiment 7. Next we look at memorization on CIFAR-10 which is a collection of 32 pixel by 32 pixel color images belonging to 10 classes. As with Binary-MNIST, we quantize each color channel to 1 bit and try to separate the classes 0 through 4 from classes 5 through 9. This gives us the Binary-CIFAR-10 task where we have to learn a function $f : B^{3\times32\times32} \to B$ from 50,000 images. Incidentally, the quantization of each color channel to 1-bit significantly degrades the signal making it a difficult task for humans.

For this task, we construct a network with 5 hidden layers each with 1024 luts and one output layer with 1 output. We set $k = 10$ for the luts. This network is able to achieve a training accuracy of 0.79 and a test accuracy of 0.63. Although not as impressive in absolute terms as the memorization result on Binary-MNIST, it is still significantly above chance (0.50). Furthermore, as before, the result is very stable and does not depend on a specific random topology chosen when the network is constructed.

We compare memorization with several standard methods in Table 4. By comparing Table 4 with Table 3 it is clear that Binary-CIFAR-10 is a harder task than Binary-MNIST since all the methods perform significantly worse on it. The best test accuracy of 0.71 is again from a LENET-style network similar to the one used in Experiment 4, but with 40 epochs of training. We believe a ResNet-style architecture (He et al., 2016) may potentially do better here but since our goal is not to achieve state-of-the-art but see how memorization does, we leave this to future work. For the same reason we don't explore data augmentation here which is a standard technique for CIFAR-10.

Once again, memorization compares favorably on test accuracy with the other methods, and compared to Binary-MNIST it does relatively better here since it ties with the nearest neighbor searches.

Table 4. Performance of some standard methods along with memorization and random guessing on the Binary-CIFAR-10 task. Note that unlike the others CONV NET is not permutation invariant.

METHOD	ACCURACY	
	TRAINING	TEST
CONV NET	0.93	0.71
RANDOM FOREST (300 TREES)	1.00	0.66
5-NEAREST NEIGHBORS	0.75	0.63
1-NEAREST NEIGHBOR	1.00	0.63
MEMORIZATION	**0.79**	**0.63**
LOGISTIC REGRESSION	0.64	0.56
NAÏVE BAYES	0.55	0.56
RANDOM GUESS	0.50	0.50

Table 5. Training accuracy (below diagonal) and test accuracy (above diagonal) on Pairwise-CIFAR-10.

	PLANE	AUTO	BIRD	CAT	DEER	DOG	FROG	HORSE	SHIP	TRUCK
PLANE		0.77	0.77	0.80	0.82	0.81	0.84	0.81	0.71	0.79
AUTO	0.96		0.79	0.77	0.79	0.80	0.80	0.78	0.76	0.67
BIRD	0.95	0.98		0.68	0.63	0.69	0.66	0.72	0.83	0.81
CAT	0.96	0.98	0.96		0.70	0.61	0.68	0.71	0.81	0.76
DEER	0.96	0.98	0.95	0.96		0.73	0.69	0.71	0.83	0.81
DOG	0.98	0.99	0.97	0.96	0.97		0.72	0.70	0.82	0.79
FROG	0.97	0.98	0.95	0.95	0.97	0.96		0.75	0.85	0.80
HORSE	0.98	0.99	0.96	0.97	0.96	0.98	0.97		0.81	0.75
SHIP	0.93	0.96	0.98	0.98	0.98	0.98	0.99	0.98		0.77
TRUCK	0.97	0.95	0.98	0.97	0.97	0.98	0.98	0.97	0.98	

Experiment 8. In this experiment, we consider the Pairwise-CIFAR-10 tasks which are defined analogously to Pairwise-MNIST. We use the same network architecture as in Experiment 7 instead of optimizing specifically for these tasks. Training accuracies are generally 0.95 and above whereas the test accuracies range from 0.61 (CAT v/s DOG) to 0.85 (FROG v/s SHIP) with an average test accuracy of 0.76 which is significantly above chance.

Experiment 9. To get qualitative insight into the decision boundaries learned with different levels of memorization, we classify points in the region $[-2, 2] \times [-2, 2] \in \mathbb{R}^2$ as being inside or outside the circle $x^2 + y^2 \leq 1.6^2$. Our dataset consists of points on a 100×100 grid in this region which has been partitioned into equal test and training sets (Figure 4, leftmost column). To make this a hard problem we encode each point as pair of 10-bit fixed-point numbers. We learn this function $f : \mathbb{B}^{20} \to \mathbb{B}$ using networks with 32 layers each with 2048 luts and vary k. With $k = 10$ (rightmost column), the training set is memorized perfectly but (as seen on test) the concept is not learned. However, memorizing with $k = 2$, we learn a simpler concept that is not faithful around the "corners" (as can be seen by zooming in) but one that generalizes almost perfectly to test. Finally, $k = 6$ provides a satisfactory compromise between the two extremes. Thus, once again, we see that memorization if done carefully can lead to good generalization.

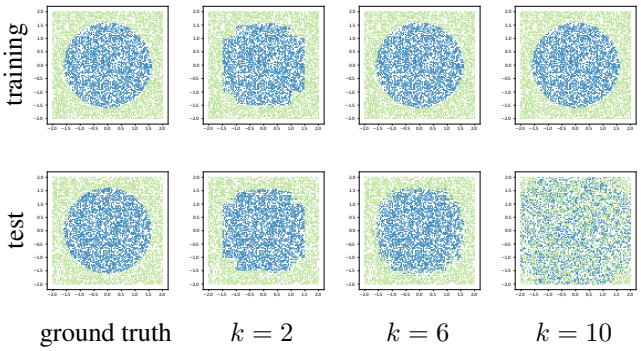

Figure 4. The decision boundaries learned in Experiment 9.

5. Comparison with Other Methods

It is instructive to compare our memorization procedure with a few commonly used procedures for learning.

k-Nearest Neighbors. The key difference, as noted in the introduction, is that k-NNs require a user-specified distance function which is often syntactic notion of distance such that induced by treating an image as a vector in $\mathbb{R}^d$. These syntactic notions of distance do not work well on more challenging tasks and one may view such a learning problem as essentially that of discovering a semantically meaningful distance function. We see this in our experiments: the distance function helps more with Binary-MNIST (Experiment 4) than it does with Binary-CIFAR-10 (Experiment 7). Furthermore, in a separate experiment we found that augmenting the table lookup with 1-NN search at test time did not significantly improve test accuracy for Binary-CIFAR-10 where memorization was already tied with k-NNs.

Additionally, k-NN requires storing the entire training set and is typically computationally more expensive at test time. For example, on Binary-MNIST the standard k-NN implementation in scikit-learn (Pedregosa et al., 2011) took more than an hour to evaluate performance on the training and test sets (as opposed to seconds with memorization). There has been work on speeding up nearest neighbor search by using locality sensitive hashing (Indyk & Motwani, 1998) and, more recently, with random projections (Li & Malik, 2016). In that context, one may view each lookup table as implementing a trivial locality sensitive hash function where the distance metric arises from exact equality, and the network as an ensemble through cascading of such nearest neighbors classifiers.

Neural Networks. The initial motivation for this work was to understand neural networks better; particularly to explore with a model the idea that perhaps SGD is a sophisticated way to memorize training data in a manner that generalizes and that perhaps there are simpler ways to memorize data

as well that may yet generalize. However, a key difference is that gradient descent-based training can learn useful intermediate representations or targets for hidden layers. In this work we have side stepped that question, by simply setting the intermediate target to be the final output. It is an interesting line of research to see if we can find a way to learn useful intermediate signals in this setting perhaps by purely combinatorial methods. Practically, that would give us a method to learn purely binary neural networks without using floating point at all, which is useful in resource constrained environments.

Random Forests. Trees in a random forest are constructed over a subset of the data by iteratively evaluating different input variables to optimize purity after splitting on the variable (Breiman, 2001). In contrast, memorization uses the whole dataset and does not solve any optimization problem (which makes it more computationally efficient). Furthermore, random forests combine the tree predictions using voting whereas memorization uses cascading.

Cascading and Stacked Generalization. A recent extension of random forests are Deep Forests (Zhi-Hua Zhou, 2017) where multiple random forests are constructed at each level and then cascaded using the idea of stacked generalization (Wolpert, 1992) which is a generalization of cross-validation. In contrast, layers of luts are far simpler, and memorization propagates outputs based on what has been memorized over the entire training data. Due to the manner in which we construct the lookup tables and the corresponding functions (using the counts of the patterns) it is not clear to us that stacked generalization will help.

Spectral Methods. There is a rich literature on the theory of learning boolean functions ($f : \mathbb{B}^k \to \mathbb{B}$ in our notation) (Mansour, 1994) which looks at theoretical learning guarantees under assumptions on the input distribution (typically uniform) and on the spectrum of the function (e.g. f can be approximated by a sparse and low degree polynomial in the boolean fourier basis). Recently, Hazan et al. (2017) have used these techniques in hyperparameter optimization where they find them to be practically useful (the distributional assumption is not fatal for this application). This line of work does not deal with depth, but only linear combinations of the basis functions. However, there is similarity in having a low degree in the fourier basis and our notion of support-limited memorization. These are similar structural priors and our results and those of Hazan et al. may be viewed as evidence that real world functions satisfy these priors.

Learning Boolean Circuits. There is relatively little prior work in directly learning boolean circuits (Oliveira & Sangiovanni-Vincentelli, 1994; Tapp, 2014). However, it is interesting to note that the memorization algorithm in Section 3 although developed independently and from different considerations is similar to the greedy algorithm described by Tapp.[2] An important difference is that instead of learning a single tree, we learn a network which makes learning more stable (as seen in Experiment 1).

6. Conclusion

The experiments of Zhang et al. (2017) and Arpit et al. (2017) on training with random data lead naturally to the question that if neural networks can memorize random data and yet generalize on real data, are they perhaps doing something different in the two cases. This work started with the opposite thought: What if in both cases they are simply memorizing? This, in turn, leads to the question of whether it is even possible to generalize from pure memorization. Naïve memorization with a lookup table is too simplistic a model but, as we saw, a slightly more complex model in the form of a network of support-limited lookup tables does significantly better than chance and is closer to the standard algorithms on a number of binary classification problems from MNIST and CIFAR-10. (To investigate if this result holds on other datasets is an important area of future work.)

Furthermore, this model replicates some of the key observations with neural networks: the performance of a network improves with depth; it memorizes random data and yet generalizes on real data; and memorizing random data is harder than real data. In particular, the last observation implies that we cannot rule out memorization based on differences in the hardness of learning between real and random data.

For future work, we would like to understand why memorization generalizes. Now, since the size of the hypothesis space is bounded by 2^{n2^k} (where n is the number of k-luts in the network), we can use results from PAC-learning to bound the generalization gap, but these bounds are typically weak or vacuous.[3] Rademacher complexity may be useful for small k (say 2), but for moderate k—where the Rademacher complexity is high yet there is generalization—we would need a different approach, perhaps one based on stability (Bousquet & Elisseeff, 2002). In this connection, we expect the results in Devroye & Wagner (1979) to apply to a single lut, but extensions are needed to handle networks of luts, i.e., depth. Furthermore, these would have to incorporate details of the construction since not every network of luts generalizes (even for $k = 2$).

Finally, given the computational efficiency of memorization, we would like to extend it to a practically useful algorithm for learning, but that would likely involve introducing some form of explicit optimization or search.

[2] We thank David Krueger for noticing the connection.

[3] For example, using Theorem 2.2 in Mohri et al. (2012) for the experiments in Table 2 (with $\delta = 0.01$ for concreteness) bounds the gap to 0.34 for $k = 2$. The bound doubles as k increases by 2.

Acknowledgments

I thank Ben Rossi, Vinod Valsalam, Rhys Ulerich, and Eric Allen for many useful discussions and Larry Rudolph and Steve Heller for their feedback on the paper.

References

Arpit, D., Jastrzebski, S. K., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S., Maharaj, T., Fischer, A., Courville, A. C., Bengio, Y., and Lacoste-Julien, S. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 233–242, 2017. URL http://proceedings.mlr.press/v70/arpit17a.html.

Bousquet, O. and Elisseeff, A. Stability and generalization. *J. Mach. Learn. Res.*, 2:499–526, March 2002. ISSN 1532-4435. doi: 10.1162/153244302760200704. URL https://doi.org/10.1162/153244302760200704.

Breiman, L. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

Devroye, L. and Wagner, T. Distribution-free performance bounds for potential function rules. *IEEE Transactions on Information Theory*, 25(5):601–604, September 1979. ISSN 0018-9448. doi: 10.1109/TIT.1979.1056087.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 1135–1143, Cambridge, MA, USA, 2015. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969239.2969366.

Hazan, E., Klivans, A. R., and Yuan, Y. Hyperparameter optimization: A spectral approach. *CoRR*, abs/1706.00764, 2017. URL http://arxiv.org/abs/1706.00764.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Indyk, P. and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pp. 604–613, New York, NY, USA, 1998. ACM. ISBN 0-89791-962-9. doi: 10.1145/276698.276876. URL http://doi.acm.org/10.1145/276698.276876.

Kawaguchi, K., Pack Kaelbling, L., and Bengio, Y. Generalization in Deep Learning. *ArXiv e-prints*, December 2017. URL https://arxiv.org/abs/1710.05468v2.

LeCun, Y. and Cortes, C. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Li, K. and Malik, J. Fast k-nearest neighbour search via dynamic continuous indexing. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 671–679. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045462.

Mansour, Y. *Learning Boolean Functions via the Fourier Transform*, pp. 391–424. Springer US, Boston, MA, 1994. ISBN 978-1-4615-2696-4. doi: 10.1007/978-1-4615-2696-4_11. URL https://doi.org/10.1007/978-1-4615-2696-4_11.

Mohri, M., Rostamizadeh, A., and Talwalkar, A. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN 026201825X, 9780262018258.

Oliveira, A. L. and Sangiovanni-Vincentelli, A. Learning complex boolean functions: Algorithms and applications. In Cowan, J. D., Tesauro, G., and Alspector, J. (eds.), *Advances in Neural Information Processing Systems 6*, pp. 911–918. Morgan-Kaufmann, 1994.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M. (eds.), *Computer Vision – ECCV 2016*, pp. 525–542, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

Tapp, A. A new approach in machine learning. *ArXiv e-prints*, September 2014.

Wolpert, D. H. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning requires rethinking generalization. In *Proceedings of the International Conference on Learning Representations ICLR*, 2017.

Zhi-Hua Zhou, J. F. Deep forest: Towards an alternative to deep neural networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 3553–3559, 2017. doi: 10.24963/ijcai.2017/497. URL https://doi.org/10.24963/ijcai.2017/497.