Circuit-Based Intrinsic Methods to Detect Overfitting

Sat Chatterjee
Google AI
Mountain View, CA
schatter@google.com

Alan Mishchenko
Department of EECS
University of California
Berkeley, CA
alanmi@berkeley.edu

Abstract

The focus of this paper is on intrinsic methods to detect overfitting. These rely only on the model and the training data, as opposed to traditional extrinsic methods that rely on performance on a test set or on bounds from model complexity. We propose a family of intrinsic methods called Counterfactual Simulation (CFS) which analyze the flow of training examples through the model by identifying and perturbing rare patterns. By applying CFS to logic circuits we get a method that has no hyper-parameters and works uniformly across different types of models such as neural networks, random forests and lookup tables. Experimentally, CFS can separate models with different levels of overfit using only their logic circuit representations without any access to the high level structure. By comparing lookup tables, neural networks, and random forests using CFS, we get insight into why neural networks generalize. In particular, we find that stochastic gradient descent in neural nets does not lead to "brute force" memorization, but finds common patterns (whether we train with actual or randomized labels), and neural networks are not unlike forests in this regard. Finally, we identify a limitation with our proposal that makes it unsuitable in an adversarial setting, but points the way to future work on robust intrinsic methods.

1 Introduction

This paper considers methods to detect overfitting of a model based only on the model and the training data. We call such methods *intrinsic* in contrast to *extrinsic* methods relying on additional knowledge, such as, the performance of the model on examples held out from the training process, details of the process used to find the model (e.g., multiple hypothesis testing with registration), or limitations of the function family to which the model belongs (e.g., VC dimension, Rademacher complexity) or of the size of the parameter space of the model (e.g. Akaike Information Criterion).

Intrinsic methods are of practical interest since with modern deep models, we find that extrinsic estimates based on model complexity are typically vacuous since these models are powerful enough to fit arbitrary data [Zhang et al., 2017]. Consequently, practitioners resort to studying performance on a held out dataset (or cross validation), but this is unsatisfactory for a couple of reasons. First, this means that in a low data setting, we cannot use all the data for training, but have to keep significant portions aside as held-out (e.g. see discussion in Dietterich [1998]) Second, it may be difficult to ensure a pristine hold out that is not touched during the research process particularly if the project is long running. (Even with a few queries to the hold out during the research process, it is possible to start fitting to the hold out [Dwork et al., 2015].)

Intrinsic methods are also interesting from a theoretical perspective. Imagine that we have sufficient computing power to, say, enumerate all neural networks (and their weights) up to a certain size. Among all the networks that fit the data well, intrinsic methods could distinguish between those networks that generalize well from those that do not, and we could view the model as a *certificate*

Preprint. Under review.

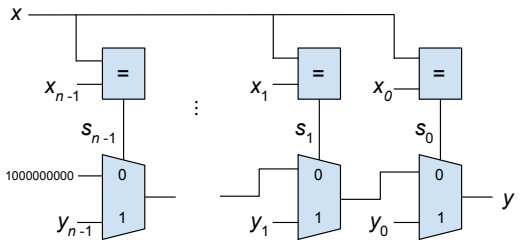

Figure 1: Sketch of a circuit implementing a lookup table.

of generalization.[1] In addition, if the intrinsic method was efficient, it would mean that supervised learning (and not just fitting) is in NP. Intrinsic methods can also help shed light on why neural networks trained with stochastic gradient descent generalize in spite of their large capacity. Some recent analyses based on normalized margin, curvature, etc. [Bartlett et al., 2017, Rangamani et al., 2019, Arora et al., 2018, Neyshabur et al., 2018] may be seen as intrinsic estimates for generalization albeit specialized to neural networks.

It is useful to consider intrinsic methods in the context of a protocol involving two agents. Let $\mathcal{S}$ be a public dataset drawn from a distribution $\mathcal{D}$ that generates samples infrequently (e.g. quarterly financial statements and returns of public companies, or public health data on treatments and patient outcomes). Suppose Arthur wants to build a model from $\mathcal{S}$ but instead of doing so himself, he outsources it to Merlin, an untrusted adversary. Merlin comes back with a model $\mathcal{M}$ but does not disclose any details of his modeling process. How can Arthur convince himself that $\mathcal{M}$ is not horribly overfit? For example, $\mathcal{M}$ could simply be a lookup table built from $\mathcal{S}$. Normally Arthur would evaluate $\mathcal{M}$ on new samples from $\mathcal{D}$, but in our setup, Arthur does not have any samples other than those in $\mathcal{S}$ since all the existing data is public. Now, if Arthur only has access to $\mathcal{M}$ as a black box and he can only evaluate $\mathcal{M}$ on elements of $\mathcal{S}$, it appears there is little he can do to distinguish a good model from a lookup table. But, and this is the central question of this paper, *can Arthur do better if he has access to the internal signals in the implementation of $\mathcal{M}$?*

We take a first step towards answering this question by studying a naturally-motivated family of intrinsic methods called *Counterfactual Simulation* (CFS) and evaluating their efficacy experimentally on a benchmark problem. The main idea behind CFS is to analyze the flow of the training examples in $\mathcal{S}$ through the structure of $\mathcal{M}$. This is only a first step since although CFS shows promise in practice, it has significant limitations. In particular, our experiments show that even if we could prove bounds based on CFS, they would not be tight enough in an adversarial setting. However, we hope that this paper encourages research to overcome these limitations or to show that no such method can exist, especially for learning tasks of practical interest.

2 Counterfactual Simulation (CFS)

The structure of$\mathcal{M}$ can be described at different levels of abstraction. For instance, if $\mathcal{M}$ is a fully connected feed forward neural network, we can describe it as a sequence of layers. Going one level lower, we can describe it as a directed acyclic graph (DAG) of (fixed or floating point) adders, multipliers, and pointwise non-linearities. Finally, at the lowest level of abstraction, we can describe the structure of $\mathcal{M}$ as a DAG of primitive logic gates such as 2-input And gates and inverters, i.e., as a combinational logic circuit. In our setup, it is natural to work at this lowest level, i.e., logic gates since it allows different kinds of models such as lookup tables, random forests, and neural networks all to be mapped into the same format. Thus, Merlin need not disclose even what type of model he has built, but simply provides Arthur with a combinational logic circuit for the model.

To make this concrete, consider the MNIST image classification problem which we will use as a running example. $\mathcal{D}$ is the distribution of handwritten digits and their classes. $\mathcal{S}$ is a sample from $\mathcal{D}$ of 60,000 images x_i and their corresponding labels y_i (thus, $0 \leq i < 60000$) i.e., the MNIST training set. Each x_i is 6,272 bits wide (corresponding to 28 $\times$ 28 pixels $\times$ 8 bits per pixel), and each y_i is 10

[1] Keeping a hold out set would not help us here—what would that even mean?

bits wide (for a 1-hot representation of the 10 possible classes). Therefore, a classifier to solve this problem is a circuit with 6,272 Boolean inputs and 10 Boolean outputs.

Suppose Merlin's model for the MNIST classifier is a simple lookup table. How would the circuit for it look? Figure 1 sketches one possibility. The 6,272-bit input x is compared with each of the examples x_i in turn and if there is a match, the corresponding 10-bit output y_i is selected. If no example matches, then the model (arbitrarily) returns the 1-hot vector representing class '0'. Now, if we simulate this circuit on examples in $\mathcal{S}$, we notice that there are internal signals in the circuit that are capable of identifying specific training examples. For example, the signal s_0 (the output of the $x = x_0$ block) is 1 (true) for the training example x_0 and 0 (false) for all others. In this case, we say 1 is a *rare pattern* for s_0 since s_0 rarely takes on the value 1 on the training set. Formally, if a signal s in $\mathcal{M}$ takes on the value v at most l times on the training set $\mathcal{S}$, we call v a l-rare pattern for s.

This observation leads to the first of the two main ideas behind CFS: *The presence of l-rare patterns suggests overfitting*, and, therefore, poor generalization since they open up the possibility that $\mathcal{M}$ has special logic to detect and handle specific examples. A count of rare patterns, however, does not directly translate into a metric for generalization (without building a predictive model of *that*!). Furthermore, although a pattern may be rare it may also be an *observability don't care* (ODC), i.e., it may have no influence on the output of the circuit. For example, if the signal with the rare pattern only feeds into an And gate whose other input is 0 when the rare pattern appears, then the value of the rare pattern does not matter in deciding the output of the circuit.

We address both problems with the second main idea behind CFS: *perturbed simulation of a training example* where we simulate an example through $\mathcal{M}$ as usual, but when we encounter a l-rare pattern, instead of propagating it to the fanouts (i.e., gates that depend on this signal), we perturb the pattern and simulate the fanouts with the perturbed pattern. A natural perturbation is to propagate the opposite value instead of the rare pattern. In our running example, this corresponds to propagating a 0 instead of 1 for signal s_0 to the mux when simulating the training image x_0. In this manner, we prevent the model from identifying x_0 and we see that the output for x_0 is no longer necessarily y_0.

We perform perturbed simulation for each training example in turn and measure the resulting average accuracy over the training set. We call this quantity the training accuracy obtained through l-CFS. In our running example of the lookup table, it is easy to see that the training accuracy obtained through 1-CFS is no better than random chance (since each training example is mapped to class '0' under 1-CFS). Now, since random chance is what one would expect to be the generalization of the lookup table (i.e. its accuracy on $\mathcal{D}$), it is tempting to conjecture that 1-CFS training accuracy is a good estimate of accuracy on $\mathcal{D}$. Although that is not the case as we shall see empirically in Section 3, we find that the difference in training accuracy between normal simulation and l-CFS is a good measure of the degree of overfit of $\mathcal{M}$.

Other Types of CFS. There are other variants of the procedure described above (which we call *Simple CFS* or just CFS). Of particular interest is *Composite CFS* which is useful for circuits with gates that have many inputs or are at higher levels of abstraction. In Composite CFS, we look at rare patterns in combinations of signals feeding a gate and perturb when a rare combination is seen. Another possibility is to randomize the perturbation. However, since in our experiments these variants produced results very similar to Simple CFS, they are not considered further in this paper.

3 Experimental Results

CFS Implementation. Our implementation of l-CFS works on a directed acyclic graph $\mathcal{G}$ representing a combinational logic circuit where each node is either the constant 0, a primary input, a 2-input And gate, or a 2-input Xor gate. An edge is either a direct connection or an inverter and represents a Boolean function in terms of the primary inputs. This is a variant of an And-Inverter Graph, a standard data structure in modern logic synthesis used to handle circuits with hundreds of millions of nodes where we propagate constants but do not extract common subexpressions.

We make two passes through the nodes of $\mathcal{G}$ in topological order starting from primary inputs. In the first pass, we simulate the training set through $\mathcal{G}$ to obtain the counts of different patterns in the circuit. In the second pass, we use the counts from the first pass to perturb the l-rare patterns. In Simple CFS, this boils down to replacing signals that take on a value of 0 on most examples with the constant 0 signal and likewise for 1. CFS thus runs in linear time in the size of the graph and

the training data. For performance, the simulations are done in a bit parallel manner for all training examples at the same time. To avoid running out of memory, we use reference counting to recycle storage for intermediate simulated values when they are no longer needed. A typical run of l-CFS in our experiments takes less than 10 minutes on a 3.7GHz Xeon CPU and less than 2GB of RAM.

Benchmark Problem. While the discussion from the previous section shows how CFS can discover overfit when the model is a simple lookup table, it is not clear if CFS would be effective on neural networks trained with stochastic gradient descent (SGD). To answer this question, we trained 3 neural networks for MNIST in TF-Keras and compiled them down into combinational logic circuits. All 3 networks have the same architecture: an input layer of size 784 (i.e., 28×28), 3 fully connected ReLU layers with 256 nodes each, and a final softmax layer with 10 outputs. (Thus, the total number of trainable parameters is 335,114.) We also performed some experiments with Fashion MNIST and the results are similar.

The first two networks (`nn-real-2` and `nn-real-100`), were trained on the MNIST training set for 2 epochs and 100 epochs respectively. They get to training (top-1) accuracies of 97% and 99.90% respectively. The third network (`nn-random`) was trained on a variant of MNIST where the output labels in the training set are permuted pseudo-randomly and trained for 300 epochs to get to a training accuracy of 91.27%.[2] (Through out we used ADAM with default parameters and batch size of 64.) As expected, `nn-real-2` is the least overfit and gets to a validation set accuracy of 97% (i.e., has a negligible generalization gap), `nn-real-100` is more overfit getting to a validation set accuracy of 98.24% (a gap of 1.66%), and finally, the validation accuracy of `nn-random` is 9.73% (i.e., close to chance) confirming that it is horribly overfit.

Conversion to Logic Circuits. This is done by generating logic subcircuits composed of 2-input And/Xor gates and inverters for each of the operations in the neural network. Weights and activations are represented by signed 8-bit and 16-bit fixed point numbers respectively with 6 bits reserved for the fractional part. (Weights from training are clamped to $[-2.0, 2.0)$ before conversion to fixed point.) Each multiply-accumulate unit multiplies an 8-bit constant (the weight) with a 16-bit input (the activation) and accumulates in 24 bits with saturation. The constant multiplications are done by finding a minimal combination of bit-shifts (multiplications by powers of 2) and additions or subtractions. For example, $5 \times u$ is implemented as $4 \times u + u$ and $11 \times u$ as $16 \times u - 4 \times u - u$. ReLUs are implemented with a comparator and a multiplexer. The outputs of each network are the 10 signed 16-bit activations before the softmax. When evaluating accuracy (with CFS or without) we pick the class corresponding to the largest of the 10 activations (top-1 accuracy). The resulting logic circuits have 35 to 52 million And/Xor gates and 5500 to 6000 logic levels. (These sizes along with the need to fit random data dictated the choice of architecture and benchmark.)

Expt. 1: Effect of Simple CFS. Figure 2a shows the training accuracies obtained through l-CFS for each of the three networks as l varies from 1 to 1024 (which is about 1.7% of the number of training examples). We call these plots *CFS curves*. As l increases, i.e., as more patterns become rare and get perturbed, the accuracy falls eventually reaching chance. However, it is interesting that the drop in accuracy is highest for `nn-random` (e.g. at l = 64, the drop is about 45%), somewhat less for `nn-real-100` (20%) and the least for `nn-real-2` (1.4%). Thus the falloff in accuracy with l is an indicator of the level of overfit of a network.

It is remarkable that even when a neural network is represented at a very low level as a logic circuit, relative overfit can be detected using an intuitive algorithm with no hyperparameters to tune.

Expt. 2: Impact of Architecture. There are many different ways in which a neural network can be compiled down into logic gates. In Expt. 1, we made certain architectural choices for the circuit, but what if we had chosen differently? To evaluate that, here we replace the multipliers used in Expt. 1 with array multipliers (i.e., multipliers based on the elementary algorithm for multiplication). Figure 2b shows the resultant CFS curves (with dashed lines) as well as the original curves (solid lines) for reference. The curves do not coincide indicating that the result of CFS depends on the structure of the circuit and not just on the function implemented by the circuit (since the function is the same in both cases). However, for the same choice of architecture, we find that the falloff in CFS curves are again indicative of the degree of overfit.

Expt. 3: Impact of Choice of Primitives. Even at the lowest level of abstraction, we can choose what primitives to work with. To see how this choice impacts CFS curves, in this experiment we

[2]While evaluating `random` for training accuracy (with or without CFS), the permuted labels are used.

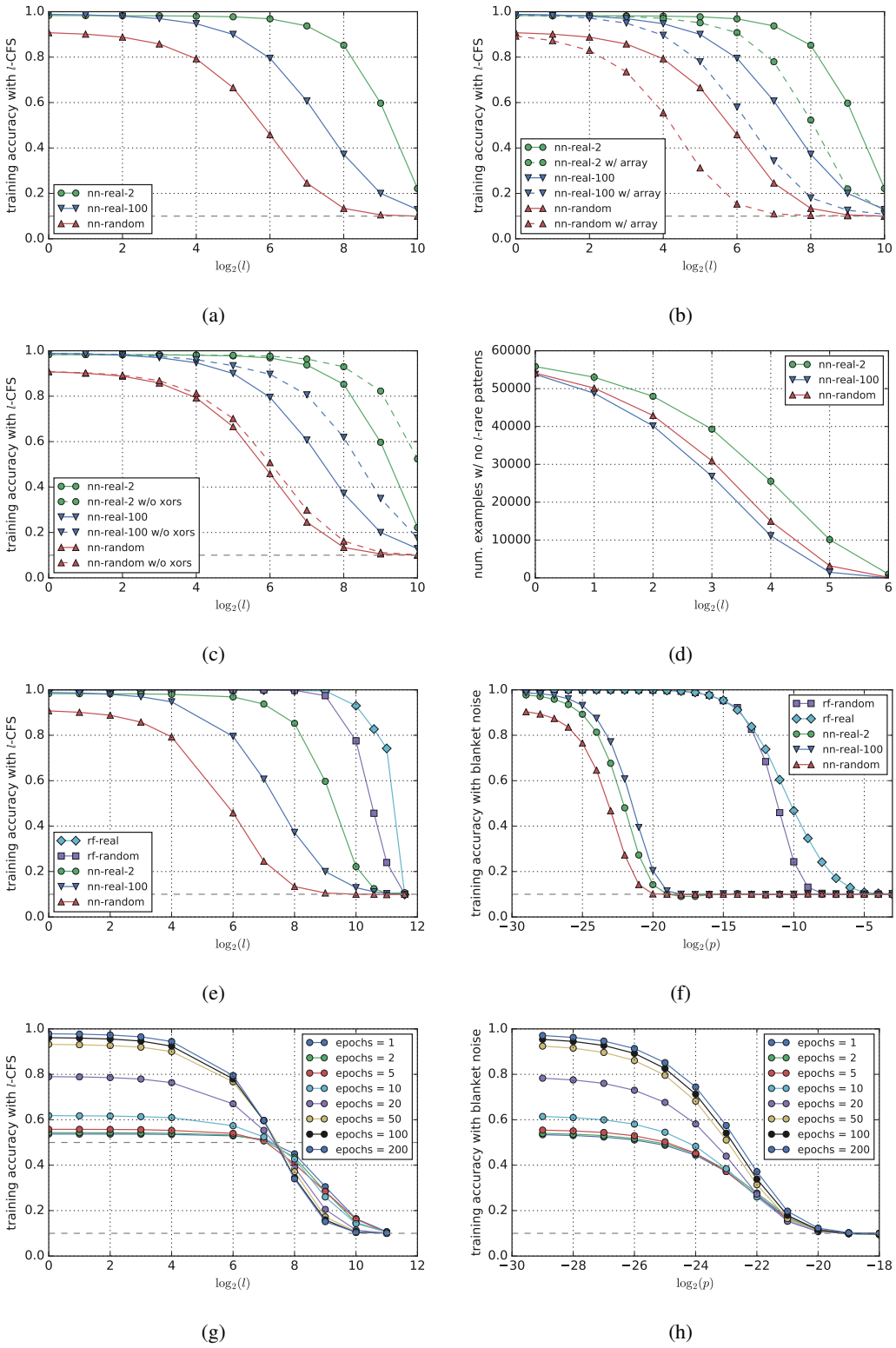

Figure 2: The results of experiments in Section 3. Plot (a) shows the CFS curves for 3 networks with different amounts of overfitting; (b) and (c) show the impact of the choice of multipliers and of primitive logic gates respectively on CFS curves; (d) shows how many examples are unaffected by l-CFS since they do not have any l-rare patterns; (e) shows CFS curves for random forests; (f) shows training accuracies when a signal is randomly flipped with probability p; and (g) and (h) show the differences between CFS and random flips for 8 networks trained on a dataset with high label noise.

disallow Xor gates as primitives (thus requiring that only And gates and inverters be used). Figure 2c shows the resulting CFS curves (dashed) as well as the originals (solid). Once again, we see that the curves do not coincide indicating that the choice of primitives matters but for the same choice of primitives, the falloff in CFS curves are again indicative of the degree of overfit.

Expt. 4: Count of Rare Patterns. Figure 2d shows for each value of l, how many examples have *no* l-rare patterns, i.e., cannot be possibly affected by l-CFS. We observe in particular, that for `nn-random`, there are 54,094 examples (about 90% of the training set) that have no 1-rare patterns. This is in sharp contrast to a simple lookup table where every example would have a 1-rare pattern, and in fact slightly more than `nn-real-100`. Comparing these curves of counts to the CFS curves in Figure 2a indicates that perturbation is an important part of CFS and that rarity by itself is a cruder measure of overfitting since it may not be observable.

Expt. 5: Random Forests. Since CFS works on the circuit level, it can check random forests for overfit. Two random forests were trained using version 0.19.1 of Scikit-learn [Pedregosa et al., 2011]. Each forest has 10 trees and is trained using the default settings, except for bootstrapping (to avoid non-uniform weights during inference). The first forest (`rf-real`) was trained on MNIST whereas the second forest (`rf-random`) was trained on MNIST with the output labels pseudo-randomly permuted (as before with `nn-random`). Both forests reach perfect training accuracy. `rf-real` gets 95.58% validation accuracy whereas as expected `rf-random` gets no better than chance. (`rf-real` has about 14K nodes per tree whereas `rf-random` has about 70K nodes per tree.)

The forests are compiled down to circuits in a straightforward manner. Each tree is compiled separately and produces 10 16-bit outputs (one per class). The corresponding outputs are added across all 10 trees and the class output by the forest corresponds to the class with the maximum value. Each internal node in a tree maps to a multiplexer controlled by a 8-bit comparator to implement the threshold, and each leaf node corresponds to 10 16-bit constants representing the number of examples that occupy each class in that leaf (thus most entries are zero). The circuit for `rf-real` has about 700K nodes whereas `rf-random` has 3M nodes. Both are less than 250 logic levels deep. (These are much smaller than the circuits for the neural networks.)

Figure 2e shows the CFS curves for the two random forests. Once again, we see that the overfit model (`rf-random`) degrades faster than the model with better generalization (`rf-real`) confirming that CFS is effective even for models that are fundamentally different from neural networks. However, it is interesting to see that if we compare *across* model families i.e. between the neural networks from Expt. 1 (repeated in Figure 2e for convenience) and the random forests, CFS is not effective at distinguishing overfit. In particular, `nn-real-2` which is not overfit degrades more rapidly that `rf-random` which is highly overfit. We discuss this in greater detail in Section 4.

Expt. 6: Blanket Noise. CFS may be seen as adding a targeted noise. Here, instead, we add blanket noise by simulating the training set while randomly flipping the node values with probability p. As p varies from 2^{-30} to 2^{-5}, the resulting *noise curves* (Figure 2f) are similar to the CFS curves (Figure 2e). However, the more overfit `nn-real-100` does *not* fall faster than `nn-real-2`. With CFS, these curves are well separated, and the gap between neural nets and forests is much larger.

Expt. 7: Sensitivity. To better understand the sensitivity difference between CFS and blanket noise, we trained 38 neural networks (with the same architecture as before but different max epochs) on MNIST with exactly one half of labels randomized (so maximum accuracy possible is about 55%). We show the CFS curves and the noise curves for 8 representative networks in Figures 2g and 2h respectively. Note the crossover of the CFS curves that indicates a larger falloff for overfit networks compared to the more uniform degradation of the noise curves. It is fascinating that all the CFS curves cross over at a single point with an accuracy of about 50%. This is discussed in Section 4.

4 Discussion

Structure Dependence. Expt. 2 and 3 show that the results of Simple CFS depend not just on the function but on the structure of the circuit. (Other CFS variants we investigated show this behavior as well.) A small example provides some insight. Consider the Boolean function $f(a, b, c) = a$ evaluated on the training set comprising the full Boolean cube (i.e., all 8 combinations of $a, b, c \in \{0, 1\}$). In addition to the direct implementation, f can also be implemented (redundantly) as $a \cdot b \cdot c + a \cdot \neg b \cdot c + a \cdot b \cdot \neg c + a \cdot \neg b \cdot \neg c$. It is easy to see that under 1-CFS, the direct implementation

is unchanged (there are no 1-rare patterns) but the redundant implementation maps to constant 0 (the output of each conjunction is 1 only once).

Although this is not a problem when the compilation process can be controlled, this is bad news in the adversarial setup. A good model with a poor implementation may show steeper degradation under CFS than a more overfit model with better implementation (e.g. c.f. `nn-real-2` with array v/s `nn-real-100` without xors at $l = 256$). Ideally, we would like to find a variant of CFS that does not depend on structure but only on the function.[3] In the absence of that ideal, we view the structure of the circuit as a certificate of how well the dataset is learnt, and make it Merlin's responsibility to find and present the most convincing structure. From this perspective, in the above example, the direct implementation of f (which is not impacted by 1-CFS) is more convincing than the redundant implementation of f (which is severely impacted).

Adversarial Attack on CFS. Based on the discussion above it is easy to design a way to arbitrarily degrade the performance of a circuit under CFS. But is the opposite possible? Can Merlin fit an arbitrary function on the inputs but compile it down to a circuit which does not degrade under CFS? Expt. 5. offers a clue. The overfit model `rf-random` fitted on random labels falls off more slowly than `nn-real-2` which generalizes well. What is going on? The short answer is that although each tree in `rf-random` is extremely overfit with most leaves containing only a single example, the circuit nodes have few rare patterns due to the observability don't cares introduced by the muxes.

Again a simple example is instructive. Let f be the parity function on n bits, i.e., $f(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \ldots \oplus x_n$. Consider an tree implementation of this function obtained using Shannon decomposition which has a multiplexer at the top controlled by x_1 and with $1 \oplus x_2 \oplus x_3 \ldots \oplus x_n$ and $x_2 \oplus x_3 \ldots \oplus x_n$ as its data inputs. If the training set is the full Boolean cube, i.e., $\{0,1\}^n$ it is easy to see that there are no l-rare signals for $l << n$ since each input to the multiplexer is balanced, i.e., has equal number of 0s and 1s. Since Shannon decomposition is recursive, a similar argument holds for the lower levels of the tree until we get to the leaves which are constant 0 and constant 1 which though unbalanced have no rare patterns.

This example suggests a way to break CFS. Build a mux tree by recursively splitting on one variable at a time, but ensuring that each branch has roughly equal number of examples of each class. Note the contrast with usual decision tree construction heuristics which favor unbalanced splits. (The unbalanced splits are likely why CFS works on random forests.)

Comparison with Blanket Noise. Based on Expt. 6 and 7, we believe that blanket noise is less sensitive than CFS. Our results here add to the extensive literature on noise, generalization and fault tolerance in neural networks (e.g., see Bernier et al. [2001] and the references therein) by extending them to the circuit level (where distinctions between activation or weight noise, or additive or multiplicative noise disappear) and to other model families such as random forests. Furthermore, Expt. 6 presents a direct comparison of the fault-tolerance of neural nets and forests where forests are seen to be about 1000x more fault-tolerant to bit flips. This is likely due to the redundancy from ensembling. It also suggests that noise-based intrinsic methods could be easily fooled by an adversary by adding redundancy.

Generalization in Deep Learning. Why do neural nets trained with SGD generalize when they have sufficient (effective) capacity to memorize their training set? This is an open research question [Zhang et al., 2017, Arpit et al., 2017, Bartlett et al., 2017, Arora et al., 2018, Neyshabur et al., 2018]. Expt 5. shows that this question is not limited to nets—the same could be asked for random forests as well. One (informal) answer for forests is that decision tree construction procedures look for common patterns between examples. When examples share commonality, they are combined into common leaf nodes and the model generalizes, whereas when there is little commonality, each example is its own leaf (so the training set is fit well) but the model fails to generalize. Could the same thing be going on with SGD and networks? CFS on `nn-random` and Expt 4. provide *direct* evidence that even on random data, nets do not "brute-force memorize" but identify common patterns in the data (a question raised in Zhang et al. [2017] and discussed in Arpit et al. [2017]).

In this context, it is interesting to conjecture why the CFS curves for the different networks in Expt 7. intersect at a common point corresponding approximately to the achievable accuracy (additional

[3] In principle, one could canonize the circuit structure before applying CFS (e.g. by building ROBDDs) but that would be computationally prohibitive. Alternatively, one could lightly optimize the circuit before CFS but that may not be enough.

experiments – see the supplement – indicate that this holds for other fixed ratios of randomly permuted labels): Roughly half of the examples are easy since they have correct labels and are learnt in the first few epochs. The remaining examples with corrupt labels are harder and learnt only in later epochs by the models that are trained longer. With CFS, the accuracy of those models breaks down earlier since the hard examples have more rare patterns than the easy examples, and the accuracy on the easy examples thus forms a limiting curve for all models. If this conjecture is true, this would provide more (and direct) evidence for the claim in Arpit et al. [2017, §1] that "SGD learns simpler patterns first before memorizing." Furthermore, we could identify "simpler patterns" as examples that have fewer rare patterns and "memorizing" as what is required for examples that have more rare patterns. Thus, learning simpler patterns and memorization are not fundamentally different but lie at two ends of a spectrum. We plan to investigate this for the final version of the paper and also look at differences between different initializations v/s snapshots from the training trajectory.

Related Work. One may be tempted to view margin as an intrinsic measure to estimate the generalization of a model. However, when we have models with intermediate representations, the notion of margin by itself is not adequate since an adversary can overfit to a favorable intermediate representation that is easily linearly separated (but otherwise arbitrary). However, recent work in this area (e.g. Bartlett et al. [2017]) has focussed on margins normalized by spectral complexity (i.e., a measure related to the Lipschitz constant of the network) and in that case the above argument does not obviously apply. We have not studied if normalized margin can be exploited by an adversary. Similarly, most measures based on the shallowness of minima are not adequate since they are not scale-invariant Dinh et al. [2017] and we have not investigated if more recent work on scale-invariant measures [Rangamani et al., 2019] can be exploited. In comparison to these and other approaches for neural networks [Arora et al., 2018, Neyshabur et al., 2018], CFS is fundamentally more discrete, which makes it applicable to a larger class of models. However, in contrast to the other approaches, we do not have any theoretical bounds yet while our results indicate that without further refinements to CFS itself, any generalization bounds from l-CFS would likely be vacuous in practice.

5 Conclusion and Future Work

Our main result is that CFS based on adding small amounts of targeted noise at the logic circuit level can detect overfit. This is remarkable because at this level of representation we have lost most aspects of the structure of the model, such as the distinction between weights and activations, or even whether the model is a neural network, a random forest, or a lookup table. Furthermore, variations such as perturbing only rare patterns in single signals or across the fanins of a gate lead to qualitatively similar results and there are variants (such as Simple CFS) that are naturally free of hyper-parameters.

By studying rare patterns, we find that SGD does not lead to "brute force" memorization, but finds common patterns (whether training is done with randomized labels or actual labels), and neural networks are not unlike forests in this regard. By adding blanket noise, random forests are found to be about 1000x more resilient to noise than neural networks which could be useful when implementing machine learning systems with unreliable low level components.

There are several directions for future work. We analyze flat circuits, but with a clever implementation that constructs the circuit on-the-fly from higher level specifications, the computations can scale to larger models. We could also apply CFS at higher levels of abstraction (perhaps as part of the model evaluation process in frameworks like Scikit Learn and Tensorflow Estimators) though at that level there are more degrees of freedom in the implementation (e.g., what kind of noise to add).

Finally, based on insights from our analysis of Simple CFS, we would like to continue the search for an intrinsic method that does not depend on the model structure and is adversarially robust, or to show that such a method does not exist, even for learning tasks of practical interest. As a generalization of this idea, it is interesting to contemplate learning algorithms that produce certificates of generalization, much like a Boolean satisfiability solver can produce a certificate of satisfiability or of unsatisfiability.

Acknowledgments

The first author thanks Michele Covell, Ali Rahimi, Alex Alemi, Shumeet Baluja, Sergey Ioffe, Tomas Izo, Shankar Krishnan, Rahul Sukthankar, and Jay Yagnik for helpful discussions.

References

Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. *CoRR*, abs/1802.05296, 2018. URL http://arxiv.org/abs/1802.05296.

Devansh Arpit, Stanislaw K. Jastrzebski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron C. Courville, Yoshua Bengio, and Simon Lacoste-Julien. 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*, pages 233–242, 2017. URL http://proceedings.mlr.press/v70/arpit17a.html.

Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6240–6249. Curran Associates, Inc., 2017.

Jose Bernier, Julio Ortega, Eduardo Ros Vidal, Ignacio Rojas, and Alberto Prieto. A quantitative study of fault tolerance, noise immunity, and generalization ability of mlps. *Neural Computation*, 12:2941–2964, 01 2001. doi: 10.1162/089976600300014782.

Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Comput.*, 10(7):1895–1923, October 1998. ISSN 0899-7667. doi: 10.1162/089976698300017197. URL http://dx.doi.org/10.1162/089976698300017197.

Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. *CoRR*, abs/1703.04933, 2017. URL http://arxiv.org/abs/1703.04933.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015. ISSN 0036-8075. doi: 10.1126/science.aaa9375. URL https://science.sciencemag.org/content/349/6248/636.

Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. Towards understanding the role of over-parametrization in generalization of neural networks. *CoRR*, abs/1805.12076, 2018. URL http://arxiv.org/abs/1805.12076.

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

Akshay Rangamani, Nam H. Nguyen, Abhishek Kumar, Dzung Phan, Sang H. Chin, and Trac D. Tran. A Scale Invariant Flatness Measure for Deep Network Minima. *arXiv e-prints*, art. arXiv:1902.02434, Feb 2019.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *Proceedings of the International Conference on Learning Representations ICLR*, 2017.