The Complexity of Explaining Neural Networks Through (group) Invariants

Danielle Ensign DANIENSIGN@GMAIL.COM
School of Computing, University of Utah

Scott Neville DROP.SCOTT.N@GMAIL.COM
School of Computing, University of Utah

Arnab Paul ARNAB.PAUL@INTEL.COM
Intel Labs, Oregon

Suresh Venkatasubramanian SURESH@CS.UTAH.EDU
School of Computing, University of Utah

Editors: Steve Hanneke and Lev Reyzin

Abstract

Ever since the work of Minsky and Papert, it has been thought that neural networks derive their effectiveness by finding representations of the data that are invariant with respect to the task. In other words, the representations eliminate components of the data that vary in a way that is irrelevant. These invariants are naturally expressed with respect to group operations, and thus an understanding of these groups is key to explaining the effectiveness of the neural network. Moreover, a line of work in deep learning has shown that explicit knowledge of group invariants can lead to more effective training results.

In this paper, we investigate the difficulty of discovering anything about these implicit invariants. Unfortunately, our main results are negative: we show that a variety of questions around investigating invariant representations are NP-hard, even in approximate settings. Moreover, these results do not depend on the kind of architecture used: in fact, our results follow as soon as the network architecture is powerful enough to be universal. The key idea behind our results is that if we can find the symmetries of a problem then we can solve it.

Keywords: Neural Networks, NP-completeness, Group Theory

1. Introduction

What does a neural network learn? One perspective on the power of deep learning is that a neural network learns a succinct and multi-scale *representation* of data. This representation is then a suitable input for the actual learning task. But what is this representation? Starting with the early work of Minsky and Papert (1972) and continuing to this very day is the idea that the learned representation can be expressed in terms of a group of operations that it is *invariant* to. For example, a convolutional neural network learns a representation of an image that is invariant under rotations and translations (the group operations).

Motivated by this principle, a number of researchers have investigated how one might encode explicit invariances in a neural network (Giles and Maxwell, 1987; Shawe-Taylor, 1993; Anselmi et al., 2015b; Paul and Venkatasubramanian, 2015; Gens and Domingos, 2014) or even directly learn representations that are invariant under some group action (Cohen and Welling, 2014).

However, the basic question: "can we identify the invariants that a particular network admits" has not been studied as extensively. Minsky and Papert (1972) showed that a network that admits invariant representation must be expressible in terms of orbits of the group (See Section 3 for definitions) but were silent on the problem of actually finding these invariants given a network.

This question is important because the invariants define the representation. If we could take any neural network and extract some information about what group operators it is invariant to, then we could build a better representation of the data directly – much like the work of Cohen and Welling (2014) and (Zaheer et al., 2017).

A separate and urgent concern comes from the area of transparency in machine learning (FATML). As deep learning gets deployed in more and more decision-making settings, the ability to *explain* the results (or decision) becomes more crucial, especially given the evidence of harm caused by black-box decision-making systems (Angwin et al., 2016).

1.1. Our Contributions

We present a number of intractability results relating to extracting the invariant group for a network. We show that the problem of identifying which group or subgroup a network is invariant to is NP-hard. We first establish this in a setting when the inputs are boolean and G is contained in the symmetric group, *i.e.,* the group of permutations. We then show that the decision version of KNAPSACK can be reduced to the problem of finding permutations to which a network is invariant (Section 5). Our next reduction builds on it, showing that an approximate version of the same is NP-hard as well (Section 5.1). Next, we extend this to continuous inputs and also to more interesting groups, such as $GL_n(\mathbb{R})$, the group of invertible matrices over $\mathbb{R}^n$ (Section 6). Our results do not depend on the kind of activation function, they work for threshold gates, sigmoids, rectified linear units (ReLUs), etc: we prove a general result to this effect in Section 7. This result essentially shows that any powerful-enough class of networks resists the extraction of invariants, suggesting that there is an explanation-expressiveness tradeoff in deep learning.

2. Related Work

The connection between groups and neural networks goes back to Minsky and Papert (1972). They proved a group invariance theorem saying that if a neural network was invariant to some group, then its output could be expressed as functions of the *orbits* of the group. While they used this group invariance theorem to prove a limit on what a perceptron could learn, other researchers used the same idea to try and construct neural networks that explicitly encoded desirable invariants (Giles and Maxwell, 1987; Shawe-Taylor, 1993; Gens and Domingos, 2014). This idea was extended further in the context of invariants for images (Anselmi et al., 2015a,b; Poggio, 2011; Smale et al., 2010; Bouvrie et al., 2009) and speech (Evangelopoulos et al., 2014). These ideas were generalized to Lie groups by Cohen and Welling (2014, 2015).

A closely related concept is group equivariance — where a function and a group operator commute. This is a similarly powerful idea that can be used for the hidden layers of a network, so that the group action can still be used by the network, but only after some preprocessing is performed. Identifying useful equivariant groups allows one to generalize deep CNNs for immediate improvements (Cohen and Welling, 2016, 2017). This is yet another case where a group informed restriction on the neural network leads to better performance.

A recent paper by Zaheer et al. (2017) sets out to construct a network whose output is permutation invariant[1], and attempts to characterize all functions that can be expressed this way. Unfortunately, they provide only a sufficient condition for the architecture to be permutation invariant, and not a necessary one. They also illustrate the advantages we could gain if we knew what invariances we could encode. A perfectly symmetric problem (such as finding an outlier in a set) can be formatted as they outline, allowing for significantly fewer parameters, a smaller internal representation, better recall, etc. It seems reasonable to expect similar potential improvements from other identified invariances, or even subsets of permutations.

Our results complement earlier work by Maurer (2011) showing that even if one were to be presented with a candidate permutation as well as a network, checking if the network is invariant to the permutation is coNP-complete. Note that this is not equivalent to our result, which shows that even finding a single permutation that the network is invariant to is NP-hard.

We note that while the group-theoretic perspective on representation learning has been studied extensively, there are other approaches to understanding the representations yielded by a neural network. Among them are approaches based on connections to kernels (Hazan and Jaakkola, 2015; Cho and Saul, 2009) and probabilistic models (Patel et al., 2015). We will not discuss them further in this work.

3. Preliminaries

We start by introducing some notation. The input space for a task will be denoted by D, the set of all n-bit binary strings $\{0,1\}^n$ unless otherwise specified. We will denote the labels for the task by the set $\mathcal{L}, |\mathcal{L}| = L$: equivalently, we think of $\mathcal{L}$ as the outputs of the neural network under consideration. Without loss of generality we assume that each label is an ℓ-bit binary string where $\ell = \lceil \log L \rceil$. Let $\varphi \colon D \to \mathcal{L}$ denote the function that a given neural network implements.

Definition 1 (G-set) *Let G be a group with identity e. Let D be a set. We say that D is a G-set if there exists an operator $\circ \colon G \times D \to D$ such that for all $d \in D, g, g' \in G$*

- $e \circ d = d$
- $g \circ (g' \circ d) = (gg') \circ d$

The operator $\circ$ is called the group action *of G on D.*

1. As of writing, the authors mistakenly assert a necessary and sufficient condition for a layer to be permutation invariant, but actually give a sufficient condition for equivariance, and then pool appropriately on all of their final layers. The authors have been notified.

Group actions give us a way of defining equivalence relations and invariants on G-sets. Let us define the relation $\sim \subset D \times D$ as $d \sim d'$ if and only if there exists $g \in G$ such that $d = g \circ d'$. Since G is a group and its action on D is closed, $\sim$ is an equivalence relation and we can denote its set of equivalence classes as the quotient D/G. These classes are called the *orbits*: in particular, the orbit of an element $d \in D$ is the set $O_G(d) = \{g \circ d | g \in G\}$ which is the equivalence class that d belongs to. Thus, orbits are the *invariants* of D under G: for any orbit $O_G(d) \in D/G$ and $g \in G$, we know that $g \circ O_G(d) = O_G(d)$.

Definition 2 (G-invariance) *A function $\varphi\colon D \to \mathcal{L}$ is G-invariant if*

1. *D is a G-set (with action $\circ$)*
2. *$\forall g \in G, d \in D : \varphi(g \circ d) = \varphi(d)$*

The idea of G-invariance formalizes the idea of a neural network being able to ignore certain kinds of variation. For example, a properly trained CNN would be G-invariant if G is the class of image rotations and translations.

4. Neural networks and group invariants

With the machinery of G-sets, group actions and G-invariance we can now capture the idea of invariant representations. Suppose $\varphi : D \to \mathcal{L}$ is G-invariant. Let $h\colon D \to D/G$ be the function that maps each element $d \in D$ to its orbit. Then there exists a function $\varphi'\colon D/G \to \mathcal{L}$ such that

$$\varphi(d) = \varphi'(h(d))$$

and thus any such function φ can be identified with a related function that operates only on the invariant subsets. If we think of φ as the function expressed by a trained network, then the function φ' is an *explanation* of what the network is doing and the function $h\colon D \to D/G$ is the learned invariant mapping. Thus understanding the invariant representations captured by a neural network amounts to understanding the way in which φ is expressed in terms of φ' and h.

Such a representation must exist to be meaningful. We first establish that for *any* classification task there is a group such that we can map the invariant subsets of D to the labels $\mathcal{L}$ directly, with every orbit having a unique label. In words, for every finite classification task there is a group whose orbits give the labels, and those orbits are given by a group of invariances.

Let $\phi : D \to \mathcal{L}$ be a classification task on a finite input D.

Lemma 3 *For every classification task ϕ there exists a group G_ϕ such that:*

1. *D is a G_ϕ-set*
2. *There exists a function $\psi\colon D/G_\phi \to \mathcal{L}$ such that:*
   (a) *ψ is injective*
   (b) *For all $x \in D$: $\psi(O_{G_\phi}(x)) = \phi(x)$*

We call G_ϕ the group of invariants of ϕ.

Proof We will construct a particularly inefficient group. Let G be an abelian group with $|\mathcal{L}| = L$ generators $\{g_1, \cdots g_L\}$. Let $O_i = \{x \mid \phi(x) = i\}$ be the set of all elements labeled i by ϕ. We let g_i be a *circle permutation* of O_i[2].

We first observe that D is a G-set. Then $\forall y \in O_i$, $G \circ y = O_i$. As only g_i acts on y, and since g_i is a circle permutation, it will eventually send y to any other element in O_i.

We can construct the map ψ via our group operation. Since we have L orbits (one per generator), we select some $y \in O_G(x)$ and set $\psi(O_G(x)) = \phi(y)$. Since there are at most L inputs, each with unique output, ψ is injective. ■

We can now think of *invariant learning* as a mapping from D to D/G. And using the universal approximation properties of neural nets (Hornik et al., 1989; Cybenko, 1989) we can always build a network that realizes such a mapping.

Lemma 4 *For any group G and G-set D, there exists a neural network $\varphi : D \to D/G$ such that*

$$\forall x \in D, \forall g \in G : \varphi(x) = \varphi(g \circ x)$$

Proof Consider the function $\phi : D \to D/G$ that sends each element d to its orbit $O_G(d)$. We can construct this function directly with a hidden layer node h_i for each possible input string $i_1, \ldots, i_n$, attached to each input k with weight of 1 if $i_k = 1$, and -1 if $i_k = 0$, with bias of PARITY(i) - the number of 1's in their binary representation. We connect each of these with weight 1 to the correct label, and give each label node a bias of 1. Clearly only one hidden layer node can ever activate for some input, and thus only one label will be applied. Since neural networks are universal function approximators, it is guaranteed that one can create a network N_ϕ that approximates ϕ arbitrarily closely. ■

The previous two lemmas establish a complete pipeline for the invariant-based view of learning. Given a classification task ϕ, Lemma 3 assures us that it can be expressed in terms of an invariant-based representation of the data. Lemma 4 then tells us how to build a neural network to extract this invariant representation.

What we lack is an *explicit* representation of the group G_φ associated with the invariant representation of φ. The next lemma tells us how we can learn about G_φ.

Lemma 5 *Given a neural network $\varphi : D \to \mathcal{L}$, we can find an element $g \in G_\varphi$ in time $O(L = |\mathcal{L}|)$.*

Proof Take $L + 1$ distinct inputs, and run φ on each input. By the pigeonhole principle, some pair of inputs d, d' will have the same label. Return the group operator g which permutes d, d' and fixes all other inputs. Clearly our network is invariant to this operator, as:

2. In particular, we fix some cyclic ordering of the set $\{x \mid \phi(x) = i\}$ and then define g_i as merely taking each element to the next in the ordering. Therefore g_i has order $|O_i|$.

1. $\forall d'' \in D \setminus \{a, b\} : \varphi(g \circ d'') = \varphi(d'')$ as $g \circ d'' = d''$
2. $\varphi(g \circ d) = \varphi(d') = \varphi(d) = \varphi(gd')$ by our choice of d, d'.

■

Note that any network is a function that takes $O(\log D)$ inputs and produces $O(\log L)$ outputs (unless there is some mechanism enforcing 1-hot encoding), so this algorithm potentially runs in exponential time with respect to the size of the network.

5. Finding an invariant permutation is hard

Lemma 5 shows us how to extract an element of the group G_ϕ. However, the algorithm presented in Lemma 5 is inefficient and tells us very little about G_ϕ. The inefficiency appears to be unavoidable since writing down an arbitrary group element that acts on D may require $|D| \log |D| = n2^n$ bits (since it may need to describe the result of the group action for each element of D). Suppose we limit ourselves to group operators that are compactly representable. Is it possible to extract such an element efficiently?

We show that the answer is negative. We start with an easy case where we merely ask if there is a permutation of the *bits* of the input that the network is invariant to. Note that this is a much smaller class of permutations, requiring only $n \log n$ bits to describe the group action for a fixed group element.

Firstly, recall that a *threshold gate* is a gate that takes inputs $x_1, x_2, \ldots, x_k$ and has a fixed *threshold* τ associated with it. It has a single output that is 1 if $\sum x_i \geq \tau$ and is 0 otherwise.

Problem 5.1 (FINDP) ***Input:*** *A circuit (i.e a directed acyclic graph) φ of threshold gates that takes n inputs and returns ℓ outputs.*

Output: *A permutation $\pi : [n] \to [n]$ (with $\pi \neq e$) of the input bits such that φ is π-invariant (i.e for all $d \in D$, $\varphi(d) = \varphi(\pi \circ d)$) or else* DNE *if such a permutation does not exist.*

Theorem 6 FINDP *is NP-hard*

Proof

An instance of KNAPSACK consists of n items $\{1, \ldots, n\}$ each with a weight w_i and value v_i, a knapsack capacity C and a total value V. The goal is to determine whether some subset $S \subset [n]$ has total weight at most C and value at least V.

The proof proceeds via a Turing reduction from KNAPSACK. We will build a circuit encoding a verifier for KNAPSACK and via repeated invocations of FINDP will either produce a solution or conclude that none exists.

Each intermediate circuit will have the following form. It takes k inputs, each denoted by a variable $b_{S_j}, S_j \subset [n]$. It has $n+2$ hidden threshold gates. These consist of two gates we will call $\mathcal{W}$ and $\mathcal{V}$ and n gates $g_i, i \in [n]$. Finally, it has one output threshold gate $\mathcal{O}$.

Each variable b_S is connected to $\mathcal{W}$, $\mathcal{V}$ and all $g_i, i \in S$. The edge from b_S to $\mathcal{W}$ has weight $-\sum_{i \in S} w_i$ and $\mathcal{W}$ has threshold $-W$. The edge from b_S to $\mathcal{V}$ has weight $\sum_{i \in S} v_i$ and $\mathcal{V}$ has threshold V. The edge from b_S to $g_i, i \in S$ has weight -1 and each g_i has threshold -1.

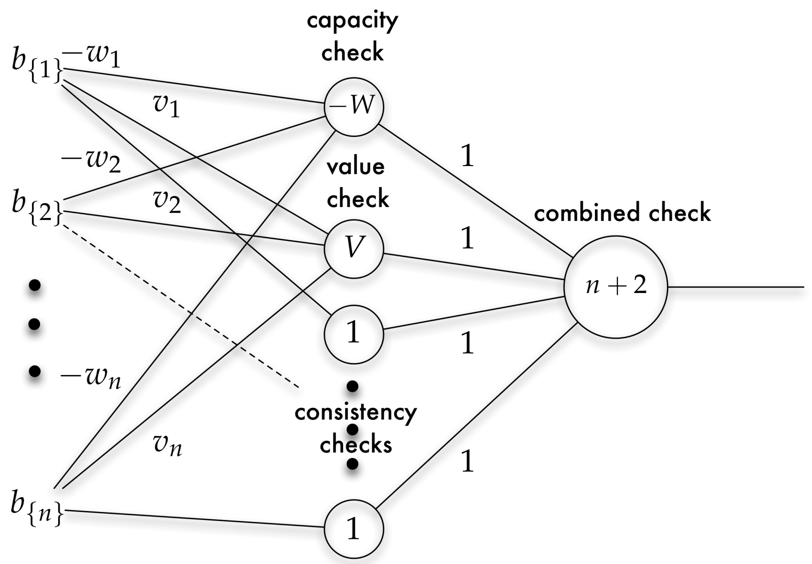

Figure 1: Knapsack Recognizer

There is one edge from each of the $n+2$ threshold gates to $\mathcal{O}$. Each edge has weight 1 and $\mathcal{O}$ has threshold $n+2$. The entire network is shown in Figure 1.

The proof relies on the following structural lemma.

Lemma 7 *Any assignment of values to the b_{S_j} that yields an output of* 1 *yields a feasible solution to* KNAPSACK.

Proof Consider any two input nodes $b_S, b_{S'}$ where $S \cap S' \neq \emptyset$. Let $i \in S \cap S'$. Since g_i has a threshold of -1, at most one of these two input nodes can be set to 1. Therefore, the set of all b_{S_j} that are set to 1 must be a disjoint collection of sets whose union we denote by $\mathcal{S} \subset [n]$. Now pick all elements $a_i, i \in \mathcal{S}$ as a solution to KNAPSACK. By construction, this set must be feasible. ■

This defines our strategy. We will use repeated calls to FINDP to progressively construct an assignment of values to the b_{S_j} that yield an output of 1, thus solving KNAPSACK.

Suppose FINDP returns a permutation π. Since π is not the trivial identity permutation, it has at least one nontrivial *cycle*: a sequence of indices $i_1, \ldots, i_\ell$ such that for all $1 \leq j < \ell$, $\pi(i_j) = \pi(i_{j+1})$ and $\pi(i_\ell) = i_1$.

Lemma 8 *Let i, j be two elements in a cycle of π. For any solution $\mathbf{b} = b_1, \ldots, b_k$ that causes the network to output a* 1 *there is another solution $\mathbf{b}' = b'_1, b'_2, \ldots, b'_k$ such that one of the following is true.*

1. $b'_i = b'_j = 1$

2. $b'_i = b'_j = 0$

3. $b'_i = 1, b'_j = 0$

Proof Take a solution in which $b_i = 0, b_j = 1$. Then because of the invariance of π, the assignment $\pi(\mathbf{b})$ is also a solution. Let k be such that $\pi^k(i) = j$ (such a k must exist since i, j are in the same cycle of π). Then in the assignment $\mathbf{b}' = \pi^k(\mathbf{b}$, we have $\mathbf{b}'_i = b_j = 1$ and then we are in either the first or third case above. ■

We can now describe the reduction. We start with a network consisting of the inputs $b_{\{i\}}, i \in [n]$ with connections to the hidden layers as described above. At each step, we invoke FINDP. Suppose it returns a permutation π and one of its cycles contains the nodes $b_S, b_{S'}$. Then one of the following two cases occurs:

S, S' **are disjoint.** We construct a new network in which the input $b_{S'}$ is replaced by an input $b_{S \cup S'}$, adjusting the edge weights and connections appropriately.

Note that the only assignment from the original network that cannot be expressed here is one in which $b_S = 0, b_{S'} = 1$. However, by Lemma 8 if there is any solution, then the original network is guaranteed to have at least one solution without this assignment, and so the new network has a solution as well. Conversely, any solution in the new network can be expressed in terms of the old one: this is because b_S and $b_{S \cup S'}$ cannot both be set to 1 in a solution due to the hidden nodes $g_i, i \in S$, and all other possible assignments to b_S and $b_{S \cup S'}$ can be expressed in terms of b_S and $b_{S'}$.

S, S' **are not disjoint.** We first observe that in any solution, $b_S, b_{S'}$ cannot both be set to 1. Suppose one of them is 0 and the other is 1. By Lemma 8 we can assume that $b_S = 1, b_{S'} = 0$. Alternately, both $b_S = 0$ and $b_{S'} = 0$. In either case, we have $b_{S'} = 0$ and can remove that node from the input, adjusting all weights accordingly.

Alternatively, FINDP returns DNE. Suppose for contradiction that this KNAPSACK instance has no satisfying assignments. Then all assignments to the inputs must return 0, and FINDP should have been able to return any permutation of the inputs. Therefore it must be the case that $\forall \pi \in S_m$, there is some input assignment d such that $\varphi(d) \neq \varphi(\pi \circ d)$. But since the network only returns two values, we know that either $\varphi(d)$ or $\varphi(\pi \circ d)$ must return 1 and so by Lemma 7 this input must yield a feasible solution to KNAPSACK.

What remains is to show that this reduction runs in polynomial time. We first observe that after each invocation of FINDP, the number of inputs either reduces by one or stays the same. Since the network is initialized with n inputs, it has at most n inputs in each intermediate step.

After each invocation of FINDP, there is at most a linear amount of work as we update the edge weights for a deleted or replaced input node. Thus, what remains is to determine the number of invocations of FINDP.

Let us call the operation of replacing the pair $b_S, b_{S'}$ by $b_S, b_{S \cup S'}$ a *merge* operation. We say that each element of S' is *associated* with $b_{S \cup S'}$. If we remove a node b_S, we call this a *deletion* and say that each element of S *participates* in the deletion. Each element $i \in [n]$ starts off being associated with the input node $b_{\{i\}}$. At any stage, each such element is associated with exactly one node, and its association can change at most n times. Once a node is deleted, all elements associated with that node are never associated with any node again. Charge each element one unit each time its association changes. Each element is charged at most n times, and there can be at most n deletion operations. Therefore, the total number of invocations of FINDP is at most $O(n^2)$, and at each stage the network has at most $O(n^2)$ edges in it.

Let the running time for an invocation of FINDP on a network with E edges be FINDP(E). Then the running time of the algorithm for KNAPSACK is at most $O(n^2(\text{FINDP}(n^2)+n))$ which is polynomial if FINDP can be implemented in polynomial time. ■

5.1. Approximate group operators are hard to find

In practice, there are many symmetries that aren't perfectly invariant. For example, we expect an image recognizer to have trouble if we translate the image too far, or if the original subject was at the edge of its view. Likewise, there may be special cases where it would make sense for some invariance to fail, but we would expect these cases to be relatively sparse, and perhaps even unimportant for the task we're interested in. In this section we introduce two separate problems to deal with these cases, and show both are hard.

Recall that a group operation π is said to be invariant if for all inputs $x \in D : \varphi(x) = \varphi(\pi \circ x)$.

Definition 9 (Deficient invariant) *A group element π is said to be a* deficient invariant *if it is invariant on all of D except some small subset $Q \subset D$, that is: $\forall x \in D \setminus Q, \varphi(x) = \varphi(\pi \circ x)$, where $|Q| = O(poly(n))$. If π is a deficient invariant of φ, we say φ is deficiently π-invariant.*

Problem 5.2 (DEFICIENTFINDP) ***Input:*** *A circuit (i.e a directed acyclic graph) φ of threshold gates that takes n inputs and returns ℓ outputs.*

Output: *A permutation $\pi : [n] \to [n]$ (with $\pi \neq e$) of input bits such that φ is deficiently π-invariant, or* DNE *if such a permutation does not exist.*

Deficient invariants are promising, as they make it easier for a particular permutation to be an invariant, and make it somewhat more difficult to show a specific permutation is not an invariant. They also can be robust to some mistakes by the network — perhaps those inputs in Q are actually mislabeled.

Definition 10 (Approximate invariant) *Let $\epsilon > 0$. We say a group operation π is an ϵ-approximate invariant if $\forall x \in D : |\varphi(x) - \varphi(\pi \circ x)|_\infty \leq \epsilon$.*

Approximate invariants reduce to exact invariants for discrete label spaces. When we instead allow $\mathcal{L} = [0,1]^n$, and use continuous activation functions like the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, the question now becomes nontrivial.

Problem 5.3 (APPROXFINDP$_\epsilon$)

Input: *A circuit (i.e a directed acyclic graph) φ of sigmoid gates that takes n inputs and returns ℓ outputs.*

Output: *A permutation π such that φ is approximately invariant to π, or DNE if none exists.*

We now state and sketch proofs for the related theorems.

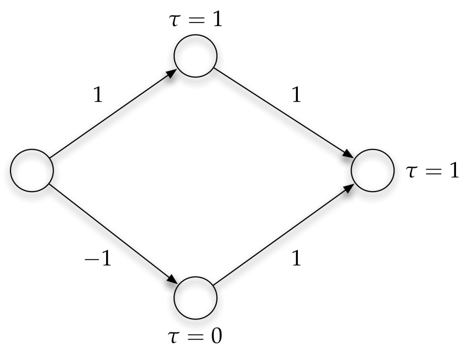

Figure 2: checking that a $[0, 1]$ variable is in $\{0, 1\}$

Theorem 11 DEFICIENTFINDP *is* NP*-hard.*

Proof [Sketch] The proof follows along the lines of Theorem 6. The main idea is to create *padded* instances (both on input and output) of the verifier for KNAPSACK and run the self-reducibility protocol as before. This has the effect of multiplying solutions into many more solutions, so that DEFICIENTFINDP acts as FINDP. The details are presented in Appendix A. ■

Theorem 12 APPROXFINDP$_\epsilon$ *is* NP*-hard.*

Proof [Sketch] The main idea here is to take the threshold gates that form an instance to FINDP and replace them by appropriately weighted sigmoid gates. By setting parameters appropriately (as a function of ϵ) we can show that a solution to APPROXFINDP will also be a valid solution to FINDP. The details are presented in Appendix A.2. ■

5.2. Continuous domains are also hard

Our results extend to the case when the inputs are continuous, rather than discrete (for example, in the case of images).

Theorem 13 FINDP *is NP-hard on continuous input neural networks.*

We briefly sketch the main idea. The key insight is the simple network shown in Figure 2. When this network is given any real value r within [0,1], the top node activates when $r \cdot 1 \geq 1$, while the bottom node activates when $-r \geq 0 \implies r \leq 0$. Our output node activates only when one of these two activate, and thus we have a 'discrete recognizer' built into a continuous neural network. During any of the previous reductions, one can change our input constraints from boolean to the continuous interval $[0, 1]$, add these 'discrete recognizer' hidden layer nodes to the construction, and all results still hold.

This same idea can be generalized to inputs on all of $\mathbb{R}$. We defer the details to an extended version of this abstract.

6. Finding invariants in $GL_n(\mathbb{R})$ is hard

Thus far, we have considered generic groups of invariants. It is conceivable that by adding more structure to the group we are searching over, the problem might get easier. Consider for example the general linear group $GL_n(\mathbb{R})$ defined over continuous inputs. While this class contains permutations as a subgroup, it also contains other groups including the group of rotations, and one might hope that in a larger class it is easier to find *some* invariant.

Problem 6.1 (FINDGL) ***Input:*** *A neural network φ of threshold gates that takes n inputs and returns ℓ outputs.*

Output: *A matrix $M \in GL_n(R)$ (with $M \neq \mathcal{I}$) such that φ is M-invariant (i.e for all $d \in D, \varphi(d) = \varphi(Md)$) or else* DNE *if no such matrix exists.*

Unfortunately, this problem is also hard.

Theorem 14 FINDGL *is NP-hard*

Proof [Sketch] Unfortunately, the larger class of invariants does not help. We prove this result via reduction from FINDP. In effect, we show that we can modify a given instance of FINDP so that the only possible invariants are permutations. Invoking a call to FINDGL then yields such a permutation if it exists, solving FINDP. We defer the proof to Appendix B. ■

7. General Networks

Of course, most neural networks used in practice are not threshold networks, and often have weight tying or other particular structures. Perhaps some of these make recovering permutations easier? While it is difficult to show that all such architectures are insufficient, there are several properties that we expect from a good architecture that imply recovering is difficult. The next definition outlines these properties.

Definition 15 (General Network Architectures) *Let NN be a set of functions ϕ, which we think of as all neural networks sharing some common architecture. We call a neural net architecture NN* general *if it satisfies the following criteria:*

Universality. *For every circuit c, $\forall \epsilon > 0$, there exists $f \in NN$ such that $|f(x) - c(x)| < \epsilon$ for all inputs x.*

Succinctness. *All such $f(x)$ use weights, nodes, layers, etc of size $POLY(\frac{1}{\epsilon}, n)$.*

Constructiveness. *There is an oracle O which takes a circuit and epsilon, and generates a satisfying $f \in NN$.*

It is not hard to show that networks consisting of activations such as threshold functions, sigmoid functions, tanh functions and rectified linear units (commonly denoted ReLUs) (Pan and Srikumar, 2016) are all general. We will show that it is hard to find invariants of any general architecture.

Theorem 16 APPROXFINDP$_\epsilon$ *is hard on any family of General Networks.*

Proof

We follow exactly the reduction from KNAPSACK outlined in 6, but we use our assumptions on NN rather than direct modification of the neural network. We use APPROXFINDP$_\epsilon$ to find an invariant, then apply the oracle to find the updated circuit with merged (or removed items) as suggested by Lemmas 7 and 8. Because of succinctness, each of these networks are polynomially sized, and so total runtime will be polynomial. ■

8. Conclusions and Future Work

Our results establish that while a neural network may be able to identify group invariance underlying a classification task, it cannot also be used at the same time to find them. We have shown that extracting these invariants is hard in the cases of permutations, even approximate ones, and $GL_n(\mathbb{R})$, as well as for a large class of general networks. While neural networks are certainly encoding this information, it does not appear to be efficiently extractable.

It should be noted our proofs can likely be generalized, and it would be interesting to explore other potential groups. It is likely that, at least for finite input spaces, finding any group operator that acts on a significant fraction of the inputs will be hard.

Another interesting direction is to identify special subclasses of networks where invariants *can* be efficiently discovered. For example, can we identify the invariants for a perceptron relatively efficiently?

Acknowledgments

This research is funded in part by the NSF under grant IIS-1633724.

References

Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. Machine bias. *ProPublica*, May 23, 2016.

Fabio Anselmi, Joel Z. Leibo, Lorenzo Rosasco, Jim Mutch, Andrea Tacchetti, and Tomaso Poggio. Unsupervised learning of invariant representations. *Theoretical Computer Science*, pages –, 2015a. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2015.06.048. URL http://www.sciencedirect.com/science/article/pii/S0304397515005587.

Fabio Anselmi, Lorenzo Rosasco, Cheston Tan, and Tomaso A. Poggio. Deep convolutional networks are hierarchical kernel machines. *CoRR*, abs/1508.01084, 2015b. URL http://arxiv.org/abs/1508.01084.

Jake V. Bouvrie, Lorenzo Rosasco, and Tomaso A. Poggio. On invariance in hierarchical models. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada.*, pages 162–170. Curran Associates, Inc., 2009. ISBN 9781615679119. URL http://papers.nips.cc/paper/3732-on-invariance-in-hierarchical-models.

Youngmin Cho and Lawrence K. Saul. Kernel Methods for Deep Learning. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *NIPS*, pages 342–350. Curran Associates, Inc., 2009.

Taco Cohen and Max Welling. Learning the irreducible representations of commutative lie groups. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1755–1763, 2014. URL http://jmlr.org/proceedings/papers/v32/cohen14.html.

Taco Cohen and Max Welling. Transformation properties of learned visual representations. In *Proc. International Conference on Learning Representations*, 2015.

Taco S Cohen and Max Welling. Group equivariant convolutional networks. *arXiv preprint arXiv:1602.07576*, 2016.

Taco S Cohen and Max Welling. Steerable cnns. *arXiv preprint arXiv:1612.08498*, 2017.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL http://dx.doi.org/10.1007/BF02551274.

Georgios Evangelopoulos, Stephen Voinea, Chiyuan Zhang, Lorenzo Rosasco, and Tomaso A. Poggio. Learning an invariant speech representation. *CoRR*, abs/1406.3884, 2014. URL http://arxiv.org/abs/1406.3884.

FATML. Fairnes, acccountability and transparency in machine learning. http://fatml.org.

Robert Gens and Pedro M Domingos. Deep symmetry networks. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2537–2545. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5424-deep-symmetry-networks.pdf.

C. Lee Giles and Tom Maxwell. Learning, invariance, and generalization in high-order neural networks. *Appl. Opt.*, 26(23):4972–4978, Dec 1987. doi: 10.1364/AO.26.004972. URL http://ao.osa.org/abstract.cfm?URI=ao-26-23-4972.

Tamir Hazan and Tommi Jaakkola. Steps Toward Deep Kernel Methods from Infinite Neural Networks. *arXiv:1508.05133 [cs]*, August 2015. URL http://arxiv.org/abs/1508.05133. arXiv: 1508.05133.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Peter M Maurer. Why is symmetry so hard? *Baylor University Technical Report*, 2011.

M.L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. Mit Press, 1972. ISBN 9780262130431. URL https://books.google.com/books?id=Ow1OAQAAIAAJ.

Xingyuan Pan and Vivek Srikumar. Expressiveness of rectifier networks. In *International Conference on Machine Learning*, pages 2427–2435, 2016.

Ankit B Patel, Tan Nguyen, and Richard G Baraniuk. A probabilistic theory of deep learning. *arXiv preprint arXiv:1504.00641*, 2015.

Arnab Paul and Suresh Venkatasubramanian. Why does deep learning work? - A perspective from group theory. *Workshop at International Conference of Learning Representation(ICLR)*, abs/1412.6621, 2015. URL http://arxiv.org/abs/1412.6621.

Tom Poggio. The computational magic of ventral stream. *Nature Proceedings*, 2011. URL http://precedings.nature.com/documents/6117/version/3.

John Shawe-Taylor. Symmetries and discriminability in feedforward network architectures. *Neural Networks, IEEE Transactions on*, 4(5):816–826, 1993.

Steve Smale, Lorenzo Rosasco, Jake V. Bouvrie, Andrea Caponnetto, and Tomaso A. Poggio. Mathematics of the neural response. *Foundations of Computational Mathematics*, 10 (1):67–91, 2010. doi: 10.1007/s10208-009-9049-1. URL http://dx.doi.org/10.1007/s10208-009-9049-1.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

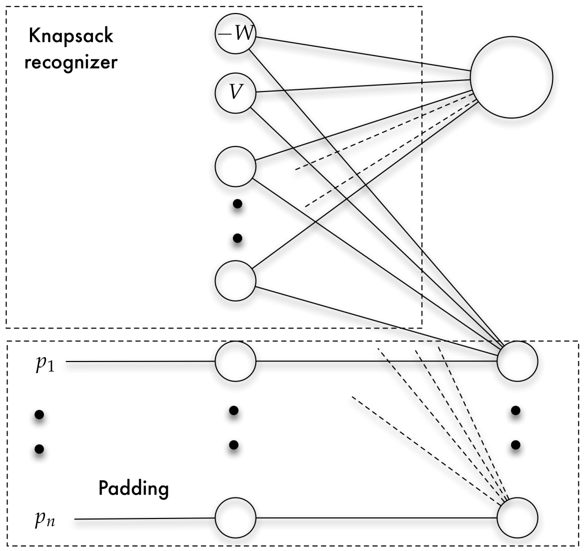

Figure 3: Knapsack recognizer with padding

Appendix A. Hardness Proofs for Deficient and Approximate Invariants

A.1. Proof of Theorem 11

Proof We call π *honest*[3] on an input d if $\varphi(\pi \circ d) = \varphi(d)$ - that is, d is does not need to be in Q. First, construct our knapsack φ exactly as we did in the FINDP proof above. We will modify φ so that every solution corresponds more than $|Q|$ solutions, and use this to force DEFICIENTFINDP to return an honest operator.

We add m new *pad* inputs p_i, with $2^m > |Q|$, then modify our knapsack φ to have m additional output nodes $o_1, \cdots, o_m$, and send the pad inputs directly into those output nodes. By this we mean that for each given pad input we add a hidden layer node h_i with threshold 0.5 which then takes in p_i with weight 1. For each o_i we add an edge from h_i with weight 1, we also add an edge from each hidden layer node in our knapsack verifier with weight $\frac{1}{n+2}$. We give each o_i a bias of $\frac{n+2}{n+3}$. These values are chosen for the next Lemma.

Lemma 17 *For all i, $o_i = p_i$ unless the input satisfies our original* KNAPSACK *verifier, in which case $o_i = 1$*

Proof Let d be some unsatisfying selection of inputs, so there must be some hidden layer node in our original knapsack that isn't activated. The input has either taken too many of a single item, exceeded our capacity constraints, or has taken too little value. There are 2 cases:

$p_i = 1$. Then the total weight coming into o_i is greater than or equal to $p_i = 1 \geq \frac{n+2}{n+3}$ so $o_i = 1$.

3. This is different from the group operation "honest", by honest we mean that π is an actual invariant for d, and not a 'dishonest' invariant (due to Q).

$p_i = 0$. Then the total weight coming into o_i is at most $(n+1)\frac{1}{n+2} = \frac{n}{n+1} < \frac{n+2}{n+3}$ because not all $n+2$ hidden layer nodes activated, so $o_i = 0$.

Note in either case, $o_i = p_i$. Hence any input that does not satisfy the KNAPSACK instance will have label $(0, p_1, \cdots, p_m)$.

Now, suppose we have a satisfying input. In this case every hidden layer node of the original KNAPSACK verifier is activated, all $n+2$ of them. Thus the total weight going into o_i is $\geq (n+2)\frac{1}{n+2} = 1 \geq \frac{n+2}{n+3}$, so all $o_i = 1$ regardless of p_i. ■

Thus every solution now corresponds to 2^m solutions, each with label $(1, 1, \ldots, 1)$. The following lemma will be important in the analysis.

Lemma 18 *if either of:*

- *π permutes one of the p_i*
- *our network is not invariant to any π and we have $\geq 3 + m$ input nodes*

then there must be a solution to the KNAPSACK *instance.*

Proof Suppose π permutes one of the p_i. Since there can only be $|Q|$ dishonest inputs, if we pick $|Q| + 1$ inputs we're guaranteed to have at least one input such that $nn(\pi \circ d) = nn(d)$. Pick these such that the p_i input is $\neq$ our p_i input after we apply π, then π has changed p_i, and thus must have changed the output label unless d and $\pi \circ d$ are solutions.

In our second case, suppose there are no invariant permutations. Then $\forall \pi \in S_{n+m}$ there must be $|Q| + 1$ inputs d such that $\varphi(d) \neq \varphi(\pi \circ d)$. Suppose π does not permuate any of the p_i, and consider any d such that $\varphi(d) \neq \varphi(\pi \circ d)$. If d is a solution, we're done, so suppose it's not. Note that applying π to an input string that does not satisfy our KNAPSACK instance either doesn't change any of the outputs (since π doesn't permute any of the p_i), or we have $\varphi(\pi \circ d) = (1, 1, \cdots, 1)$, and $\pi \circ d$ is a solution. Hence we know there is a solution to the orginal KNAPSACK instance.

■

Now, we run DEFICIENTFINDP on our network. The analysis follows.

1. Suppose there are no permutations. This can only happen if there are solutions to our knapsack, so we return TRUE.
2. Suppose π permutes one of the pad variables p_i. Then return TRUE.
3. Finally, our permutation can act entirely on the original variables. In this case we update as we did for section 5.1 (the FINDP proof).

Because we can iteratively create smaller and smaller networks using permutations, eventually the knapsack instance is small enough we can brute force the solution. At first glance, one might think the approximation lets us lose solutions or find invalid permutations, however if s is a solution to our original knapsack problem, we now have 2^m possible pad inputs that, along with s, are valid solutions, and not all of them can be inside Q. Thus

π must act honestly on all but a few of these inputs, but note they're all identical — π doesn't act on our pad bits. Thus for π to act on only our knapsack items, it must be honest on all solutions, and our reduction continues the same as before.

Note our construction is at most a logarithmic factor larger than the construction for FINDP, and the runtime analysis is exactly the same. ■

A.2. Proof of Theorem 12

Proof The proof follows from a reduction from FINDP. Given a threshold φ, we can construct an approximation φ' for φ using sigmoid activations instead of thresholds. We can adjust the biases so that every threshold gate in φ is only queried for values strictly above or below the threshold by finding the highest precision value in the original threshold network, say 2^{-p}, and rounding all biases down to the nearest multiple of 2^{-p-1}. Note in the proof for FINDP, we were able to use only integer weights and $\mathbb{Z}\frac{1}{2}$ biases, rounding all biases to the nearest $\mathbb{Z} + \frac{1}{2}$ will suffice. For simplicity we'll assume the same here.

Choose any $\epsilon < \frac{1}{4}$. We want to ensure the approximation is within $\frac{\epsilon}{2}$ so that the combined error of 2 inputs will not exceed ϵ. To do this we need only scale all weights and biases by $O(\frac{1}{\epsilon})$, with the constent depending on the precision of the biases and weights. Once we've constructed φ', we run APPROXFINDP$_\epsilon$. There are two cases:

1. Suppose we find π, an ϵ-approximate invariant of φ' then π is also an invariant of φ, since our outputs were $0, 1$, our new outputs are $0 \pm \epsilon, 1 \pm \epsilon$, and so for all x, $|\varphi'(x) - \varphi(\pi^n \circ x)| < \epsilon \implies \varphi(x) = \varphi(\pi^n \circ x)$.
2. Suppose we do not find any ϵ-approximate invariant π. Then no π can be an invariant of φ as we would have for all $x \in D$, $\varphi(x) = \varphi(\pi x) \implies |\varphi'(x) - \varphi(\pi x)| < \epsilon$, so π would also be ϵ-approximately invariant.

Hence we've reduced FINDP to APPROXFINDP. Note φ' uses the same sized network as φ with $O(-\log(\epsilon))$ longer weights, so this reduction is polynomially sized. Additionally, our reduction requires only a single call to APPROXFINDP, and $O(|\varphi|^2)$ multiplications, so if APPROXFINDP were in P, then FINDP would be too.

■

We remark that sigmoid functions are not critical to this proof. Any activation function that can concisely approximate a threshold gate with reasonable coefficients will give the same result.

Appendix B. Proof of Theorem 14

Proof We prove this via reduction from FINDP. Suppose we have an oracle for FINDGL. We will take a network instance φ for FINDP and create a new threshold network φ' such that its only possible invariants are permutations. We make the following changes to φ to construct φ':

1. relax the inputs of φ to $\mathbb{R}^n$

2. add a new label B to ℓ (to denote the label *bad*), and a corresponding output node to φ'.

3. For each input node x_i, we add 4 new hidden layer nodes:

   (a) $g1_i = \text{threshold}\ (x_i \geq 1)$, which is made by attaching x_i to $g1_i$ with weight 1, and giving $g1_i$ a bias of 1.
   (b) $g0_i = \text{threshold}\ (x_i \geq 0)$, which is made by attaching x_i to $g0_i$ with weight 1, and giving $g0_i$ a bias of 0.
   (c) $l1_i = \text{threshold}\ (x_i \leq 1)$, which is made by attaching x_i to $l1_i$ with weight -1, and giving $l1_i$ a bias of -1.
   (d) $l2_i = \text{threshold}\ (x_i \leq 0)$, which is made by attaching x_i to $l0_i$ with weight -1, and giving $l0_i$ a bias of 0.

   Note 3 of these activate if and only if $x_i \in \{0,1\}$, otherwise only 2 activate.

4. Let $w_{i,o}$ denote the weight from the ith node in φ to o (which is 0 if no such edge exists), and b_o the bias of o. For each output node o in φ, let $K_o = 1 + |b_o| + \sum_i |w_{i,o}|$, ie K_o one greater than the maximum magnitude φ can input to o. For each output node o we add edges from $g1_i, g0_i, l1_i, l0_i$ to o with weight K_o, and increase the bias of o by $3nK$. Thus if any input is not binary, then every output except B will be 0, and if the input is binary, then the outputs of φ' will behave as φ.

5. Connect B only to our new hidden layer nodes $g1_i, g0_i, l1_i, l0_i$ for all i, each with weight -1 and give B bias $0.5 - 3n$.

Note that φ' has only $O(n)$ new nodes and $O(\log|\varphi|)$ larger weights, so this is a polynomially bigger instance.

Lemma 19 *φ' is identical to φ for $x \in \{0,1\}^n$, and φ' labels all other inputs B.*

Proof If $x \notin \{0,1\}^n$ then less than $3n$ of the new hidden layer nodes will be active, so B will be activated, and no output node will activate as:

$$\sum_{h\in\varphi} a_h + b_o + K\sum_i (g1_i(x) + g0_i + l1_i + l0_i) - 3nK < K + 3(n-1)K + 2K - 3nK = 0$$

by our choice of K_o.

Similarly, if $x \in \{0,1\}^n$ then $3n$ of our hidden layer nodes will be active, and since $-3n \leq 0.5 - 3n$, we will not apply the label B. Each of the outputs of φ' compute:

$$3nK_o + \sum_i a_i \leq 3nK_o + b_o \leftrightarrow \sum_i a_i \leq b_o$$

thus all $\{0,1\}^n$ inputs keep their previous labels. ■

Now, we run FINDGL on φ'. Suppose we get some matrix M. M find must fix $\{0,1\}^n$, as otherwise there would exist some $d \in \{0,1\}^n$ such that Md is not $\{0,1\}^n$, but then

$\varphi'(Md) = B \neq \varphi'(d)$. Further, note the only matrices in GL that fix $\{0,1\}^n$ are permutations, so M must be a permutation matrix, and we return it as the answer to FINDP. Now, suppose FINDGL returns DNE. Because any permutation preserves $\{0,1\}^n$ and $\varphi(x) = \varphi'(x)$ for $x \in \{0,1\}^n$, it must be that φ is not invariant to any permutation, so we can return DNE for FINDP as well.

Similarly, if there is a valid permutation P such that φ unrelaxed is P invariant then the modified network for FINDGL is also P invariant, as every input in B remains in B under permutation (there's still some non-$\{0,1\}$ input) and every input in $\{0,1\}^n$ has kept their original label.

Since the reduction consists only of computing relatively short sums, $O(n)$ new nodes and a single call to FINDGL, the reduction is polynomial. ■

B.0.1. A LOWER BOUND FOR WRITING AN ARBITRARY GROUP ACTION

We differ for a moment to discuss a lower bound on specifying an arbitrary group action. Fix D, a finite space of size 2^n. Suppose D is a G-set for some group G. How much space do we need to speficy the action of a single operator $g \in G$ on D without any assumptions about G?

The only constraint is that $g\circ$ is a bijection, so that it can have an inverse.

Proposition 20 *$g\circ$ is invertible.*

Proof This follows from the fact that g is invertible in G, hence:

$$d = e \circ d = g^{-1}g \circ d = g^{-1} \circ (g \circ d)$$

by definition, and G-set properties. So $g^{-1}\circ$ is the inverse of $g\circ$. ■

Since we have no assumptions about G, g can have any order (and it will always have an order, any bijection on a finite set does). Further, any bijection $g\circ$ associates with itself, so any bijection is possible as a group action.

How many bijections are there? Order the elements of D. The first element could be sent anywhere (including itself), so we have $|D|$ choices. The second can be mapped to anywhere *except* the element the first was mapped to, so we have $|D| - 1$ choices. In general, for the ith choice we have $|D| - i + 1$ remaining options. The number of options is independent of all other choices, so the total number of such bijections is $|D|!$, hence we must have $\log(|D|!) \sim |D| \log(|D|) - |D|$ bits (stirlings approximation) to specify an arbitrary group operators action on D.