Modularity in Query-Based Concept Learning

Benjamin Caulfield BCAULFIELD@BERKELEY.EDU
unaffiliated

Sanjit A. Seshia SSESHIA@EECS.BERKELEY.EDU
University of California, Berkeley, CA, USA

Editors: G. Pappas, P. Ravikumar, S. A. Seshia

Abstract

We define and study the problem of modular concept learning, which is learning a concept that is a cross-product (i.e., Cartesian product) of component concepts. The theory of concept learning provides a framework for analyzing algorithms for inductive synthesis of programs and systems, which involves synthesis from examples and queries. Modular concept learning is important for systems that can be broken into subsystems that each act independently of the other. We analyze this problem with respect to different types of queries that are made to an oracle, formalized as an oracle interface. We show that if a given oracle interface cannot directly answer questions about the components, learning can be difficult, even when the components are easy to learn with the same type of oracle queries. Specifically, we show that learning from membership, equivalence, or subset queries is hard. However, these problems become tractable when oracles are given a positive example and are allowed to ask membership queries.

Keywords: Active Learning, Exact Learning, Inductive Synthesis, Query-Based Learning, Modularity

1. Introduction

Concept learning uses examples and queries to find the best hypothesis (i.e., concept) from a prescribed set of valid hypotheses. In *exact concept learning*, the learned concept must exactly match a target "correct" concept. It is often not possible to do this from examples alone, and learners often need to make specific queries, such as membership queries that ask "is this example consistent with the target concept" (i.e., is it an element of the target set) (Angluin, 1988).

This field of exact concept learning has been particularly relevant for inductive synthesis, which aims to synthesize programs (concepts) from examples or other observations. Inductive synthesis has found application in formal methods, program analysis, software engineering, and related areas, for problems such as invariant generation (e.g. (Garg et al., 2014)), program synthesis (e.g., (Solar-Lezama et al., 2006)), compositional reasoning (e.g. (Cobleigh et al., 2003)), and software analysis (e.g., (Vaandrager, 2017; Howar and Steffen, 2018)). The special nature of query-based learning for formal synthesis, where a program is automatically generated to fit a high-level specification through interaction with oracles, has also been formalized as *oracle-guided inductive synthesis* by Jha and Seshia (2017).

However, the existing theory of query-based concept learning does not exploit the modularity present in programming languages and formalisms. Researchers in the field of exact concept learning will often fix a concept class and then study which queries are needed to learn concepts in that class. When a different concept class is studied, an entirely new algorithm will need to be developed from scratch, even if it is composed of already studied concept classes.

We introduce the idea of modular concept learning, which allows us to break down concept classes into their components and study the learnability of the larger class in terms of the learnability of the components. We focus on the case when concepts are the cross-products (i.e., Cartesian

Query Name	Symbol	Complexity	Oracle Definition
Single Positive Query	$1Pos$	n/a	Return a fixed $x \in c^*$
Positive Query	Pos	$\#Pos$	Return an $x \in c^*$ that has not yet been given as a positive example (if one exists)
Membership Query	*Mem*	$\#Mem$	Given element x, return 'true' iff $x \in c^*$
Equivalence Query	*EQ*	$\#EQ$	Given $c \in C$, return 'true' if $c = c^*$ otherwise return $x \in (c \backslash c^*) \cup (c^* \backslash c)$
Subset Query	*Sub*	$\#Sub$	Given $c \in C$, return 'true' if $c \subseteq c^*$ otherwise return some $x \in c \backslash c^*$
Superset Query	*Sup*	$\#Sup$	Given $c \in C$, return 'true' if $c \supseteq c^*$ otherwise return some $x \in c^* \backslash c$
Example Query	$EX_{\mathcal{D}}$	$\#EX(\mathcal{D})$	Samples x from distribution $\mathcal{D}$ and returns x with a label indicating whether $x \in c^*$

Table 1: Types of queries studied in this paper. The expression under "Complexity" is a variable representing how many instances of that query the algorithm needs to learn the target concept. It is used, for example, in Figure 2.

products) of component concepts. This occurs when a system can be broken into subsystems that act independently of each other. We describe below two problem instances where such modular learning can be useful: one in program synthesis via "sketching" (Solar-Lezama et al., 2006), and another on learning automata, where algorithms such as Angluin's algorithm for learning deterministic finite automata (DFAs) has had particular success (Angluin, 1987) . For example, Angluin's algorithm uses membership and equivalence queries. We show that when an automaton is made of several independent components, our results can reduce the number of equivalence queries exponentially in the number of components.

We will focus on the queries given in Table 1. The results are summarized in Table 2, and include both upper and lower bounds. Learning cross-products from equivalence queries or subset queries is intractable, while learning from just membership queries is polynomial, though somewhat expensive. We show that when a learning algorithm is allowed to make membership queries and is given a single positive example, previously intractable problems become tractable. This situation of being given one (or more) positive examples is motivated by practice: for example, the LoopInvGen invariant synthesis tool generates positive examples by executing the program (Padhi et al., 2019), and the use of concept learning for synthesizing the switching logic for hybrid automata (Seshia, 2012). Appendix B shows that learning cross-products from superset queries is no more difficult than learning each individual concept seperately. Appendix F studies the complexity of Probably Approximately Correct learning (PAC learning) and shows how it can be improved when membership queries are allowed.

1.1. Sample Application: Program Synthesis by Sketching

To illustrate the learning problem, consider the sketching problem given in Figure 1. Here we want to find the set of possible initial values for x and y that can replace the ?? values so that the program satisfies Φ, using Φ as a black-box oracle mapping x and y inputs to 'true' and 'false'.

Looking at the structure of this program and specification, we can see that the correctness of these two variables are independent of each other. Correct x values are correct independent of y and vice-versa. Therefore, the set of settings will be the cross-product of the acceptable settings

```
int x = ??;
int y = ??;
int a = f(x);
int b = g(y);
```

$\Phi = \phi_1(a) \wedge \phi_2(b)$

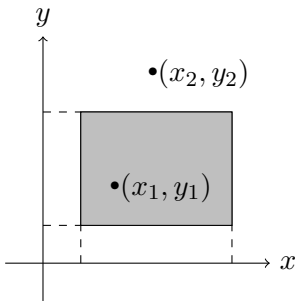

Figure 1: A simple partial program to be synthesized to satisfy a specification Φ (left) and the correct set of initial values for x and y (right).

for each variable. If an oracle can answer queries about correct x or y values separately, then the oracle can simply learn the acceptable values separately and take their cross-product.

If these sets of correct values are continuous intervals, the correct settings will look something like the rectangle shown in Figure 1. If we had access to an algorithm that could learn intervals from queries, we could try to use this algorithm as a blackbox in order to create a "product-learner" to learn their cross-product (i.e., the rectangle). This product-learner would begin instances A_1 and A_2 of the interval-learning algorithm, which we call the "sublearners", and try to answer the queries the sublearners ask. The product-learner would only have access to an oracle that can answer queries about the rectangle, not the intervals. For example, if both sublearners need a positive example, the product-learner can query the oracle for a positive example. Given the positive example (x_1, y_1) as shown in the figure, the product-learner can then pass x_1 and y_1 to each of the sublearners as positive examples.

However, this does not apply to negative examples, such as (x_2, y_2) in the figure. In this example, x_2 is in its target interval, but y_2 is not. When just given the negative example (x_2, y_2), the product-learner has no way of knowing which dimension a negative element fails on. Handling negative counterexamples is one of the main challenges of this paper.

1.2. Sample Application: Learning Automata

Consider the problem of learning a finite automaton that is the product of two or more component automata. Algorithm 1 of this paper can make black-box queries to Angluin's well-known algorithm for learning finite automata (Angluin, 1987) from membership and equivalence queries, without knowing anything of her original algorithm.

To see why it may be useful to learn the product of two component automata, consider an automaton representing the proper input/output interactions of a simple vehicle. Assume the vehicle can be split into two systems: the headlights and the motor controls.

It may be reasonable to assume that these two systems act independently. So the correct input/output sequence for motor controls is true regardless of the state of the headlights, and vice-versa. This allows us to make inferences like "The brakes should always activate after the brake pedal is pushed, regardless of whether the high-beams are on".

Now consider the problem of learning this automaton from oracle interactions or labelled examples (i.e., examples that are labelled with a 'yes' or 'no'). If the labels and oracle answers are subsystem-specific (e.g., "The headlights are incorrect in this example"), then it might be possible to learn each subsystem separately.

But there are a few reasons why this subsystem-specific feedback might not be possible:

1. If the oracle answers queries by running simulations, it might not be obvious which subsystem is responsible for a fault. (e.g., "Did the car crash because the high-beams were on or because the breaks didn't function correctly?")

2. When only learning from labelled examples, the data might not include this subsystem information.

3. The oracle may be implemented by an existing model which does not have this subsystem information.

This last case is relevant in system deobfuscation, where black-box queries are made to a complex model in order to learn a simple representation of that model. That simpler representation might be used to explain the model to a human, or it might be checked against a logical specification.

In particular, recent work has focused on using Angluin's algorithm to learn finite automata from RNNs Weiss et al. (2017). If the RNN could be split into independent subsystems, but was not trained in a modular fashion using data with subsystem-specific information, then a naive application of Angluin's algorithm would require learning the product automaton.

If the headlights-automaton has n states and the motor control automaton has m states, then the product of these two automata might be of size mn. Learning this product automaton from Angluin's algorithm would require $O(mn)$ equivalence queries. However, we can leverage the learning algorithms from this paper to learn the same automaton in $O(m + n)$ equivalence queries. If an automaton is made of k components of size n, this process can reduce the number of equivalence queries from $O(n^k)$ to $O(kn)$, resulting in an exponential increase in efficiency. The appendix develops these ideas in further detail.

1.2.1. REUSING COMPONENT ALGORITHMS

Another important application of this work is the reuse of existing learning algorithms for previously unstudied concept classes. For example, assume that we have a different model of a car, which uses an interval of acceptable internal temperatures and an automaton representing the car's motor controls. Again, we might assume the car's motor controls are correct regardless of its temperature and vice-versa. This model might be represented as the cross-product of an interval of integers (representing the acceptable temperatures) and a finite automaton (representing motor controls). Although it is unlikely that a learning algorithm for this particular type of model would have been studied, individual learning algorithms for intervals and automata have been. A researcher could then use these two algorithms as the black-box sublearners to learn the entire model.

2. Notation

This paper deals with *concept learning*, a type of machine learning where hypotheses are represented as sets. We use X to represent the "universe" over which learning is done, called the *instance space* (or just *space*). A concept over X is a set $c \subseteq X$. A *concept class* C over X is a set of concepts over X. The goal of concept learning is to find a hypothesis (i.e., concept) c in C that best fits a *target concept* c^* in C. Excluding the section on PAC learning, we focus on *exact learning*, meaning the algorithm should return the target concept c^*. For example, when learning regular languages, the instance space X might be the set $\{a, b\}^*$ (i.e., all finite strings using a and b) and the concept class C would be the set of regular languages defined using only a and b.

In the following proofs, we assume we are given concept classes $C_1, C_2, \ldots, C_k$ defined over instance spaces $X_1, X_2, \ldots, X_k$. Each target concept c_i^* in each C_i is learnable from algorithm A_i (called the*sublearner*) using queries to an oracle that can answer any queries in a set $\mathcal{Q}_{subc}$. This set $\mathcal{Q}_{subc}$ contains the available types of queries, which are taken from the list of queries shown in Table 1. For example, if $\mathcal{Q}_{subc} = \{Mem, EQ\}$, then each A_i can make membership and equivalence queries to its corresponding oracle.

<table>
<tr><th rowspan="2">$\mathcal{Q}_{subc} \downarrow$</th><th rowspan="2">$\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$
#q</th><th colspan="2">$\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{Mem, 1Pos\}$</th></tr>
<tr><th>#Mem</th><th>#q</th></tr>
<tr><td>Pos</td><td>Not Possible</td><td>Not Possible</td><td>Not Possible</td></tr>
<tr><td>Sup</td><td>$\sum \#Sup_i$</td><td>0</td><td>$\sum \#Sup_i$</td></tr>
<tr><td>Mem</td><td>$(max_i\{\#Mem_i\})^k$</td><td>$\sum \#Mem_i$</td><td>$\sum \#Mem_i$</td></tr>
<tr><td>Sub</td><td>$k^{\sum \#Sub_i}$</td><td>$lg(k) \sum \#Sub_i$</td><td>$\sum \#Sub_i$</td></tr>
<tr><td>EQ</td><td>$k^{\sum \#EQ_i}$</td><td>$lg(k) \sum \#EQ_i$</td><td>$\sum \#EQ_i$</td></tr>
<tr><td>{ EQ, Mem}</td><td><table><tr><th>#Mem</th><th>#EQ</th></tr><tr><td>$(max_i\{\#Mem_i + \#EQ_i\})^k$</td><td>$\sum \#EQ_i$</td></tr></table></td><td><table><tr><th>#Mem</th><th>#EQ</th></tr><tr><td>$\sum \#Mem_i$
$+ lg(k) \sum \#EQ_i$</td><td>$\sum \#EQ_i$</td></tr></table></td><td></td></tr>
</table>

Figure 2: Final collection of query complexities for learning cross-products. The rows represent the set $\mathcal{Q}_{subc}$ of queries needed to learn each C_i. The columns determine whether the cross-product is learned from queries in just $\mathcal{Q}_{subc}$ or $\mathcal{Q}_{subc} \cup \{Mem, 1Pos\}$. In the latter case, the column is separated to track the number of membership queries and queries in $\mathcal{Q}_{subc}$ that are needed. The value k denotes the number of dimensions (i.e., concept classes) included in the cross-product. If $|\mathcal{Q}_{subc}| = 1$, then $\#q$ is the number of queries to the one element of $\mathcal{Q}_{subc}$ that are needed to learn the cross-product. In the case when $\mathcal{Q}_{subc} = \{Mem, EQ\}$, the meaning of $\#q$ is not defined, so the complexity of each case is split into $\#Mem$ and $\#EQ$.

For each query $q \in \mathcal{Q}_{subc}$, we say algorithm A_i makes $\#q_i$ (or $\#q_i(c_i^*)$) many q queries to the oracle in order to learn concept c_i^*, dropping the index i when unambiguous. We replace the term $\#q$ with a more specific term when the type of query is specified. For example, an algorithm A might make $\#Mem$ many membership queries to learn c.

Unless otherwise stated, we will assume any index i or j ranges over the set $\{1 \dots k\}$. We write $\prod S_i$ or $S_1 \times \cdots \times S_k$ to refer to the k-ary Cartesian product (i.e., cross-product) of sets S_i. We use S^k to refer to $\prod_{i=1}^k S$.

We use vector notation $\boldsymbol{x}$ to refer to a vector of elements $(x_1, \dots, x_k)$, $\boldsymbol{x}[i]$ to refer to x_i, and $\boldsymbol{x}[i \leftarrow x_i']$ to refer to $\boldsymbol{x}$ with x_i' replacing value x_i at position i. The concept class formed by taking the cross product of the subclasses is defined by $\boxtimes_{i=1}^k C_i := \{\prod c_i \mid c_i \in C_i, i \in \{1, \dots, k\}\}$. We write $\boldsymbol{c}$ or $\prod c_i$ for any element of $\boxtimes_{i=1}^k C_i$ and will often denote $\boldsymbol{c}$ by $(c_1, \dots, c_k)$ in place of $\prod c_i$. Thus the target concept, c^* in $\boxtimes_{i=1}^k C_i$ can be represented as $(c_1^*, \dots, c_k^*)$.

The *product oracle* is able to answer queries about the target concept c^*. The types of queries this oracle can answer are in the set $\mathcal{Q}_{prod}$, which are taken from the queries in Table 1. We now have enough to state the problem of this paper.

Problem Statement: For different sets of queries, $\mathcal{Q}_{subc}$ and $\mathcal{Q}_{prod}$, can we bound the number of queries needed to learn a concept in $\boxtimes C_i$ as a function of each query complexity, $\#q_i$, for each $q \in \mathcal{Q}_{subc}$?

There are far too many combinations of sets $\mathcal{Q}_{subc}$ and $\mathcal{Q}_{prod}$ to consider in one paper. In this paper we will mostly focus on the cases when $|\mathcal{Q}_{subc}| = 1$ and $\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$ or $\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{1Pos, Mem\}$, as these cases are more likely to appear in practice.

3. Simple Lower Bound

We will start with a simple lower bound on learnability from *EQ*, *Sub*, and *Mem*. See Figure 3 for a visual representation of this proposition. We will see later that this lower bound is tight when learning from membership queries, but not equivalence or subset queries. The appendix shows that learning from superset queries is in fact easy.

Proposition 1 *For any positive integer n, there exists a concept class C_1 that is learnable from n many queries posed to $\mathcal{Q}_{subc} \subseteq \{\text{Mem}, \text{EQ}, \text{Sub}\}$ such that learning C_1^k requires at least n^k many queries when $\mathcal{Q}_{prod} = \mathcal{Q}_{subc}$.*

Proof Let $C_1 := \{\{j\} \mid j \in \{0 \dots n\}\}$.

We can learn C_1 in n membership, subset, or equivalence queries by querying, for all j between 0 and n, either $j \in c^*$, $\{j\} \subseteq c_1^*$, or $\{j\} = c_1^*$, respectively. However, a learning algorithm for the product class C, defined by $C := C_1^k$, requires more than n^k queries. To see this, note that C_1^k contains all singletons in a space of size $(n+1)^k$.

So for each subset query $\{x\} \subseteq c^*$, if $\{x\} \neq c^*$, the oracle will return x as a counterexample, giving no new information. Likewise, for each equivalence query $\{x\} = c^*$, if $\{x\} \neq c^*$, the oracle can return x as a counterexample. Therefore, any learning algorithm must query $x \in c^*$, $\{x\} \subseteq c^*$, or $\{x\} = c^*$ for $(n+1)^k - 1$ values of x in $\{0, \dots, n\}^k$. ■

4. Learning From Membership Queries and One Positive Example

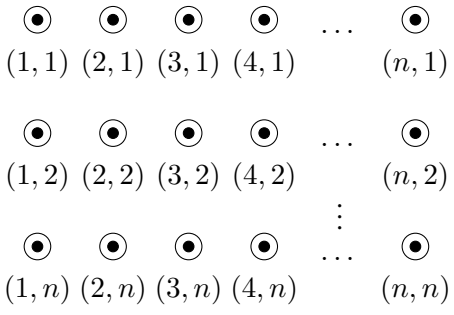

Figure 3: Representation for $C \times C$ in Proposition 1, when $k = 2$. The circle around each point represents a singleton set in $C \times C$.

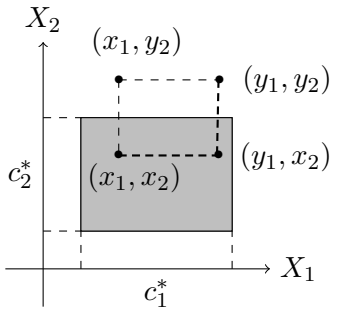

Figure 4: The figure for Example 1 on handling counter-examples with membership queries.

Ideally, learning the cross-product of concepts should be about as easy as learning all the individual concepts. The last section showed this is not the case when learning with equivalence, subset, or membership queries. However, when the learner is given a single positive example and allowed to make membership queries, the number of queries becomes tractable. This is due to the following simple observation.

Observation 1 *Fix sets $S_1, S_2, \ldots, S_k$, points $x_1, x_2, \ldots, x_k$ and an index i. If $x_j \in S_j$ for all $j \neq i$, then $(x_1, x_2, \ldots, x_k) \in \prod S_i$ if and only if $x_i \in S_i$.*

This suggests a simple method for handling counterexamples. Given a positive example $\boldsymbol{p} \in c^*$ and a counterexample $\boldsymbol{x} \notin c^*$, query $\boldsymbol{p}[j \leftarrow x_j] \in c^*$ for each j. Recall that $\boldsymbol{p}[j \leftarrow x_j]$ is the result of replacing the jth element of $\boldsymbol{p}$ with x_j. So by the above observation, there will be some j such that $p[j \leftarrow x_j] \notin c^*$ and we can infer that $x_j \notin c^*_j$. The appendix shows that we can use binary search to find such a j in $lg(k)$ queries, but we will focus on using k queries here. This process closely matches that of empirical sciences, where variables are fixed by a control and then a single variable is changed. This process is better explained in the following example.

Example 1 *Figure 4 shows an example of using membership queries to handle a counter-example. The rectangle represents the target hypothesis $c^* = c^*_1 \times c^*_2$. The point $\boldsymbol{p} = (x_1, x_2)$ is a positive example and (y_1, y_2) is a negative counter-example. The algorithm wants to find whether (y_1, y_2) is a negative example because $y_1 \notin c^*_1$ or $y_2 \notin c^*_2$. It then constructs $\boldsymbol{p}[1 \leftarrow y_1]$ (i.e., (y_1, x_2)) and queries $\boldsymbol{p}[1 \leftarrow y_1] \in c^*$. The oracle returns 'true', so $y_1 \in c^*_1$. It then repeats the process and queries $\boldsymbol{p}[2 \leftarrow y_2] \in c^*$. The oracle returns 'false', so $y_2 \notin c^*_2$, and the oracle passes y_2 to learner A_2 as a counterexample.*

We now have enough information to present the following theorem.

Theorem 2 *Assume a single positive example $\boldsymbol{p} \in c^*$ is given.*

1. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Mem}\}$, then c^* is learnable in $\sum \#\text{Mem}_i$ membership queries.*
2. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{EQ}\}$, then c^* is learnable in $lg(k) \cdot \sum \#\text{EQ}_i$ membership queries and $\sum \#\text{EQ}_i$ equivalence queries.*
3. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Sub}\}$, then c^* is learnable in $lg(k) \cdot \sum \#\text{Sub}_i$ membership queries and $\sum \#\text{Sub}_i$ subset queries.*
4. *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Mem}, \text{EQ}\}$, then c^* is learnable in $lg(k) \cdot \sum \#\text{EQ}_i + \sum \#\text{Mem}_i$ membership queries and $\sum \#\text{EQ}_i$ equivalence queries.*

Proof (Sketch) See Appendix C for full proofs and algorithm descriptions.

Item 1 The algorithm learns by simulating each A_i in sequence, moving on to A_{i+1} once A_i returns a hypothesis c_i. For any membership query M_i made by A_i, $M_i \in c^*_i$ if and only if $\boldsymbol{p}[i \leftarrow M_i] \in c^*$ by Observation 1. Therefore the algorithm is successfully able to simulate the oracle for each A_i, yielding a correct hypothesis c_i.

Item 2 The algorithm asks for an equivalence query, $c_i = c^*_i$, from each learner A_i. It then queries $\prod c_i$ to the oracle. Negative counter-examples are handled as in Example 1. If a positive counter-example $\boldsymbol{x}$ is given, then $\boldsymbol{x}[i]$ is passed as a positive counter-example to each A_i.

Item 3 (Similar to Item 2) The algorithm asks for a subset query, $c_i \subseteq c^*_i$, from each learner A_i. It then queries $\prod c_i$ to the oracle. Negative counter-examples are handled similarly to in Example 1. The logarithmic factor can be achieved by a simple binary-search-like handling of queries (See Algorithm C in Appendix C).

Item 4 See Algorithm 5. The learning algorithm answers membership queries for each A_i as in Item 1 until an equivalence query is asked. The equivalence queries are then combined as in Item 2 and passed to the oracle. ■

5. Learning From Only Membership Queries

We have seen that learning with membership queries can be made significantly easier if a single positive example is given. If no positive example is given, then Proposition 1 gives a lower bound on the number of membership, subset, or equivalence queries needed. This section gives an algorithm showing that this bound is tight when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Mem}\}$ or $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Mem}, \textit{EQ}\}$. The algorithm uses membership queries so that either a positive example is found or the target concept is learned. Once a positive example is found, the learning algorithm from Section 4 can be used.

Somewhat surprisingly, even if $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Mem}, \textit{EQ}\}$, only membership queries are needed to find a positive example. We present the algorithm for learning when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Mem}, \textit{EQ}\}$ in Algorithm 1, since the algorithm is essentially the same if no equivalent queries are allowed. Unlike the other algorithms in this paper, this algorithm assumes that we can choose an element in each hypothesis. There might not be an efficient algorithm for this choosing. However, choosing an arbitrary element will likely be more efficient than checking the equivalence of two concepts. We only need to choose elements from $k \cdot max_i\{\#\textit{EQ}_i\}$ hypotheses, so this requirement is not too restrictive.

Proposition 3 *If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Mem}, \text{EQ}\}$, then Algorithm 2 can find a positive example using $max_i\{\#\text{Mem}_i + \#\text{EQ}_i\}^k$ membership queries or at most one equivalence query.*

Proof (Sketch: see Appendix D for full proof) Algorithm 2 answers the queries posed by each sublearner. For each sublearner A_i, there are two possible cases: (I) every answer to that sublearner is correct or (II) some answer to that sublearner is incorrect. In case (I), the sublearner will eventually return the correct concept (and an element from that concept is added to T_i). In case (II), the answer $x_i \notin c_i^*$ (in response to a membership query) or the counterexample $x_i \in S_i \backslash c_i^*$ (in response to an equivalence query) must be incorrect. Either way, x_i is in c_i^* and x_i is added to T_i. So in either case each T_i will eventually contain a positive element. Since a membership query is made to every possible element in $\prod T_i$, an element of c^* will be found. ■

As mentioned before, finding a positive example when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Mem}\}$ is essentially the same process as Algorithm 2. The main difference is that we make the initial query $\emptyset = c^*$ at the beginning of the algorithm. Appendix D shows that when $\emptyset \in \boxtimes C_i$, there is no way to determine if $c^* = \emptyset$ using membership queries. Algorithm 2 will find a positive example if one exists, or run indefinitely otherwise. In all other cases, Algorithm 2 works with only membership queries.

6. Learning from Equivalence or Subset Queries is Hard

The previous section showed that learning cross-products of membership queries requires at most $O(max_i\{\#\textit{Mem}_i\}^k)$ membership queries. A natural next question is whether this can be done for equivalence and subset queries (i.e., when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{EQ}\}$ or $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\textit{Sub}\}$).

In this section, we answer that question in the negative. We will construct a concept class $\mathfrak{C}$ that can be learned from n equivalence or subset queries but which requires at least k^n queries to learn $\mathfrak{C}^k$. We define $\mathfrak{C}$ to be the set $\{\mathfrak{c}(s) \mid s \in \mathbb{N}^*\}$, where $\mathfrak{c}(s)$ is defined over strings such that $\mathfrak{c}(\lambda) := \{\lambda\} \times \mathbb{N}$, $\mathfrak{c}(s) := (\{s\} \times \mathbb{N}) \cup \mathfrak{c}_{sub}(s)$, and $\mathfrak{c}_{sub}(s \cdot a) := (\{s\} \times (\mathbb{N} \backslash \{a\})) \cup \mathfrak{c}_{sub}(s)$. For example, $\mathfrak{c}(1 \cdot 2) = (\{1 \cdot 2\} \times \mathbb{N}) \cup (\{1\} \times (\mathbb{N} \backslash \{2\})) \cup (\{\lambda\} \times (\mathbb{N} \backslash \{1\}))$. Here, $1 \cdot 2$ refers to the concatenation of symbols 1 and 2.

To learn $\mathfrak{c}(s)$, it is enough to find the underlying string s. This can be done by constructing longer prefixes of s from the counter-examples given by an oracle. So, an algorithm learning $\mathfrak{c}(1 \cdot 2 \cdot 3)$ from equivalence queries might start by querying $\mathfrak{c}(\lambda)$ and getting a negative counter-example $(\lambda, 1)$. It can then infer that 1 is a prefix of the target string s. It then queries $\mathfrak{c}(1)$ and

LearnMemEQ()

Get positive example p
while *some* A_i *has not completed* **do**
  for *each* A_i **do**
    // Answer *Mem* queries until an *EQ* query is made
    while A_i *queries* $x_i \in C_i$ **do**
      Query $\boldsymbol{p}[i \leftarrow x_i] \in c^*$
      Return answer to A_i
    Receive learned concept S_i or a query $S_i = c_i^*$ from A_i
  Query $\prod S_i = c^*$
  if $\prod S_i = c^*$ **then**
    return $\prod S_i$
  else
    Receive c.e. $\boldsymbol{x}$
  if $\boldsymbol{x} \in \prod S_i$ **then**
    for $i \in \{1, \ldots, k\}$ **do**
      Query $\boldsymbol{p}[i \leftarrow x_i] \in c^*$
      if $\boldsymbol{p}[i \leftarrow x_i] \notin c^*$ **then**
        Pass c.e. x_i to A_i
  else
    // $\boldsymbol{x} \in c^* \setminus \prod S_i$
    for $i \in \{1, \ldots, k\}$ **do**
      if $x_i \notin S_i$ **then**
        Pass c.e. x_i to A_i
return $\prod S_i$

Algorithm 1: Learn cross-product with queries $\mathcal{Q}_{subc} = \{EQ, Mem\}$ and $\mathcal{Q}_{prod} = \{EQ, Mem, 1Pos\}$

FindPos()

if $\emptyset \in C$ **then**
  Query $\emptyset = c^*$
  return counterexample
Initialize all $T_i := \emptyset$
while *True* **do**
  for $i \in \{1, \ldots, k\}$ **do**
    Ask A_i for query
    if *No possible query* **then**
      Pass
    if A_i *returns* c_i **then**
      Choose $y_i \in c_i$
      Add y_i to T_i
    if A_i *queries* $x_i \in c_i^*$ **then**
      Add x_i to T_i
      Pass "False" to A_i
    else
      A_i queries $S_i = c_i^*$
      Choose $y_i \in S_i$
      Pass y_i as c.e. to A_i
      Add y_i to T_i
  for *Unqueried* $\boldsymbol{y} \in \prod T_i$ **do**
    Query $\boldsymbol{y} \in c^*$
    if $\boldsymbol{y} \in c^*$ **then**
      return $\boldsymbol{y}$

Algorithm 2: Finds positive example when $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem, EQ\}$.

gets a negative counter-example $(1, 2)$. It then queries $\mathfrak{c}(1 \cdot 2)$ and gets a negative counter-example $(1 \cdot 2, 3)$. Finally, it queries the correct concept $\mathfrak{c}(1 \cdot 2 \cdot 3)$ and is done. Appendix E proves the following proposition and gives the full algorithm description.

Proposition 4 *There exist algorithms for learning from equivalence queries or subset queries such that any concept* $\mathfrak{c}(s) \in \mathfrak{C}$ *can be learned from* $|s|$ *queries.*

We will now demonstrate a lower bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. This will imply that $\mathfrak{C}^k$ is hard to learn from equivalence queries, since an adversarial equivalence query oracle can give the exact same answers and counterexamples as a subset query oracle.

It is easy to learn $\mathfrak{C}$, since each new counterexample gives at least one more character in the target string s. When learning a concept, $\prod \mathfrak{c}(s_i)$, it is not clear which dimension a given counterexample applies to. Specifically, a given counterexample $\boldsymbol{x}$ could have the property that $\boldsymbol{x}[i] \in \mathfrak{c}(s_i)$ for all $i \neq j$, but the learner cannot infer the value of this j. It must then proceed by considering all possible values of j, requiring exponentially more queries for longer s_i.

To see this, consider the following example, where a learner must learn $\mathfrak{c}(s_1) \times \mathfrak{c}(s_2)$, when $|s_1| + |s_2| = 2$.

Example 2 *First, the learner queries* $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$ *to the oracle and receives a counter-example* $((\lambda, 1), (\lambda, 2))$. *We now know either* s_1 *starts with* 1 *or* s_2 *starts with* 2. *The learner queries* $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ *and receives counterexample* $((1, 3), (\lambda, 4))$. *If* s_1 *starts with* 1, *then either* $s_1 = 1 \cdot 3$ *or* s_2 *starts with* 4. *If* s_1 *doesn't start with* 1, *then we've learned nothing. The learner queries* $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$ *and receives counterexample* $((\lambda, 5), (2, 6))$. *At this point, there are four possible values of* c^**:* $(\mathfrak{c}(1), \mathfrak{c}(4))$, $(\mathfrak{c}(1 \cdot 3), \mathfrak{c}(\lambda))$, $(\mathfrak{c}(5), \mathfrak{c}(2))$ *and* $(\mathfrak{c}(\lambda), \mathfrak{c}(2 \cdot 6))$. *The learner must query all but one of these in order to find the correct concept.*

A learning algorithm that uses the above strategy would need $2^{(|s_1|+|s_2|)}$ queries to learn a concept $\mathfrak{c}(s_1) \times \mathfrak{c}(s_2)$ of arbitrary length. In Appendix E, we show that this is in fact the best strategy. Namely, we demonstrate an adversarial oracle such that any learner that does not use the above strategy must fail. That oracle and the corresponding proofs yield the following theorem.

Theorem 5 *Any algorithm learning* $\mathfrak{C}^k$ *from subset (or equivalence) queries requires at least* k^r *queries to learn a concept* $\prod \mathfrak{c}(s_i)$, *where* $r = \sum |s_i|$. *Equivalently, the algorithm takes* $k^{\sum \#\text{Sub}_i}$ *subset (or* $k^{\sum \#\text{EQ}_i}$ *equivalence) queries.*

7. Related Work on Modularity

Modularity has been fairly well-studied in statistical machine learning with uses in neural networks (Bottou and Gallinari, 1991; Anand et al., 1995; Auda and Kamel, 1999) and cyber-physical systems (Bradley, 2010). However, there has been less work on studying modularity in computational learning theory. The main work on this subject, by Bshouty et. al., has studied the problem of learning compositions of concept classes using equivalence queries (Ben-David et al., 1997; Bshouty, 1998). Here the composition for a class C is the set C^* built from all boolean combinations of concepts from C. Their work is orthogonal to our own, as the current paper allows for composition of many different classes but does not account for all boolean combinations. Moreover, their work only studies PAC-learning and learning from equivalence queries.

8. Conclusion

This paper is a first look at the problem of modular concept learning, with a focus on learning the cross-product of concept classes. It shows that learning cross-products can become exponentially hard when using membership, equivalence, and subset queries. These learning problems become easier when a single positive example is given and the learner is allowed to use membership queries. We also demonstrate a method for systematically finding a positive example from membership queries alone. Further results on PAC-learnability of cross-products can be found in Appendix F.

Future work in this area can study more permutations of usable queries $\mathcal{Q}_{subc}$ and $\mathcal{Q}_{prod}$. Additionally, as the specification in Figure 1 demonstrates, the product concept class represents a conjunction of its subclasses. It may be interested to study modular learning of other fixed boolean operations, such as 'or' or 'implication'.

Acknowledgments

This work was largely done while the first author was affiliated with UC Berkeley. It was supported in part by NSF grant 1139138, by Microsoft Research, and by the iCyPhy center.

References

Rangachari Anand, Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. Efficient classification for multiclass problems using modular neural networks. *IEEE Transactions on Neural Networks*, 6(1):117–124, 1995.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75 (2):87–106, 1987.

Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

Gasser Auda and Mohamed Kamel. Modular neural networks: a survey. *International Journal of Neural Systems*, 9(02):129–151, 1999.

Shai Ben-David, Nader H Bshouty, and Eyal Kushilevitz. A composition theorem for learning algorithms with applications to geometric concept classes. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 324–333, 1997.

Léon Bottou and Patrick Gallinari. A framework for the cooperation of learning algorithms. In *Advances in neural information processing systems*, pages 781–788, 1991.

David M Bradley. Learning in modular systems. Technical report, Carnegie Mellon Univ., Pittsburgh PA, 2010.

Nader H Bshouty. A new composition theorem for learning algorithms. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 583–589, 1998.

Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

Falk Howar and Bernhard Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54 (7):693–726, 2017.

Saswat Padhi, Rahul Sharma, and Todd Millstein. LoopInvGen: A loop invariant generator based on precondition inference. *ArXiv e-prints*, 2019.

Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the Design Automation Conference (DAC)*, pages 356–365, June 2012.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017. doi: 10.1145/2967606. URL https://doi.org/10.1145/2967606.

Aad Van Der Vaart and Jon A Wellner. A note on bounds for VC dimensions. *Institute of Mathematical Statistics collections*, 5:103, 2009.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. *arXiv preprint arXiv:1711.09576*, 2017.

Appendix A. Automata Learning

This section further develops the applications on learning automata that were stated in the introduction.

Given a finite set of alphabet Σ, a set $\Gamma \subseteq \Sigma$, and a string $s \in \Sigma^*$, the projection of s to Γ, written $\pi_\Gamma(s)$ is formed by removing all symbols from s that are not in Γ. More formally, for the empty string λ, $\pi_\Gamma(\lambda) = \lambda$. For all $t \in \Sigma^*$ and $a \in \Sigma$, $\pi_\Gamma(ta) = \pi_\Gamma(t)a$ if $a \in \Gamma$ and $\pi_\Gamma(ta) = \pi_\Gamma(t)$ if $a \notin \Gamma$. Projection can be extended to languages $L \subseteq \Sigma^*$ by $\pi_\Gamma(L) := \{\pi_\Gamma(s) \mid s \in L\}$.

The following definition formalizes the concept of independence discussed in the introduction.

Definition 6 *Let Σ be a finite alphabet and let $P = \{\Sigma_1, \ldots, \Sigma_k\}$ be a partitioning of Σ into disjoint subsets. We say P is independent on the language $L \subseteq \Sigma^*$ if for each $s \in \Sigma^*$, $s \in L$ if and only if for all $i \in \{1, \ldots, k\}$, $\pi_{\Sigma_i}(s) \in \pi_{\Sigma_i}(L)$.*

To understand this definition, consider the following example. Let $|s|_a$ be the number of occurrences of a character a in the string s (similar for b). Define $L := \{s \mid |s|_a \equiv 0 \bmod 2 \text{ and } |s|_b \equiv 0 \bmod 3\}$ for $L \subseteq \{a, b\}^*$. Then the partition $P = \{\{a\}, \{b\}\}$ acts independently on L.

We can restate this concept of independence in term of cross products. The following proposition follows immediately from the definition of independence.

Proposition 7 *For any finite alphabet Σ and formal language $L \subseteq \Sigma^*$ and partition $P = \{\Sigma_1, \ldots, \Sigma_k\}$ on Σ. If P act independently on L, then for any $s \in \Sigma^*$, $(\pi_{\Sigma_1}(s), \ldots, \pi_{\Sigma_k}(s)) \in \prod \pi_{\Sigma_i}(L)$ if and only if $s \in L$.*

An algorithm to learn these cross products can be constructed from any oracle answering questions about the language L. Each query about cross products is translated to an equivalent query about L. The query $(s_1, \ldots, s_k) \in \prod \pi_{\Sigma_i}(L)$ is translated to $s_1 \cdot \cdots \cdot s_k \in L$. To answer the query $\prod L_i = \prod \pi_{\Sigma_i}(L)$, let L' be the largest set in Σ such that for all i, $\pi_{\Sigma_i}(L) = L_i$. To construct this set, let M_i be the automaton accepting each L_i. For each M_i add self-loops to each state labelled with the characters in $\Sigma \backslash \Sigma_i$. The intersection of the resulting languages is the set L', and $\prod L_i = \prod \pi_{\Sigma_i}(L)$ if and only if $L' = L$. These constructions can also be used to answer subset, superset, and example queries.

Appendix B. Easy Results

We show two simple results, learning from positive examples is not always possible and learning from superset queries is easy.

Proposition 8 *There exist concepts C_1 and C_2 that are each learnable from constantly many positive queries, such that $C_1 \times C_2$ is not learnable from any number of positive queries.*

Proof Let $C_1 := \{\{a\}, \{a, b\}\}$ and set $C_2 := \{\mathbb{N}, \mathbb{Z} \backslash \mathbb{N}\}$. To learn the set in C_1, pose two positive queries to the oracle, and return $\{a, b\}$ if and only if both a and b are given as positive examples. To learn C_2, pose one positive query to the oracle and return $\mathbb{N}$ if and only if the positive example is in $\mathbb{N}$. An adversarial oracle for $C_1 \times C_2$ could give positive examples only in the set $\{a\} \times \mathbb{N}$.

Each new example is technically distinct from previous examples, but there is no way to distinguish between the sets $\{a\} \times \mathbb{N}$ and $\{a, b\} \times \mathbb{N}$ from these examples. ■

The next proposition makes use of the following simple observation:

Observation 2 *For sets $S_1, S_2, \ldots, S_k$ and $T_1, T_2, \ldots, T_k$, assume $\prod S_i \neq \emptyset$. Then $\prod S_i \subseteq \prod T_i$ if and only if $S_i \subseteq T_i$, for all i.*

Proposition 9
If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{\text{Sup}\}$, then there is an algorithm that learns any concept $c^* \in \prod C_i$ in $\sum \#\text{Sup}_i(c_i^*)$ queries.

Proof Algorithm B learns $\boxtimes C_i$ by simulating the learning of each A_i on its respective class C_i. The algorithm asks each A_i for superset queries $S_i \supseteq c_i^*$, queries the product $\prod S_i$ to the oracle, and then uses the answer to answer at least one query to some A_i. Since at least one A_i receives an answer for each oracle query, at most $\sum \#Sup_i(c_i^*)$ queries must be made in total.

We will now show that each oracle query results in at least one answer to an A_i query (and that the answer is correct). The oracle first checks if the target concept is empty and stops if so. If no concept class contains the empty concept, this check can be skipped. At each step, the algorithm poses query $\prod S_i$ to the oracle. If the oracle returns 'yes' (meaning $\prod S_i \supseteq c^*$), then $S_i \supseteq c_i^*$ for each i by Observation 2, so the oracle answers 'yes' to each A_i. If the oracle returns 'no', it will give a counterexample $\boldsymbol{x} = (x_1, \ldots, x_k) \in c^* \setminus \prod S_i$. There must be at least one $x_i \notin S_i$ (otherwise, $\boldsymbol{x}$ would be in $\prod S_i$). So the algorithm checks $x_j \in S_j$ for all x_j until an $x_i \notin S_i$ is found. Since $\boldsymbol{x} \in c^*$, we know $x_i \in c_i^*$, so $x_i \in c_i^* \setminus S_i$, so the oracle can pass x_i as a counterexample to A_i.

Note that once A_i has output a correct hypothesis c_i, S_i will always equal c_i, so counterexamples must be taken from some $j \neq i$. ■

Result: Learn $\prod C_i$ from Superset Queries
if $\emptyset \in C_i$ *for some* i **then**
  Query $\emptyset \supseteq c^*$
  if $\emptyset \supseteq c^*$ **then**
    return $\emptyset$
for $i = 1 \ldots k$ **do**
  Set S_i to initial subset query from A_i
while *Some* A_i *has not completed* **do**
  Query $\prod S_i$ to oracle
  if $\prod S_i \supseteq c^*$ **then**
    Answer $S_i \supseteq c_i^*$ to each A_i
    Update each S_i to new query
  else
    Get counterexample $\boldsymbol{x} = (x_1, \ldots, x_k)$
    for *i = 1 . . . k* **do**
      if $x_i \notin S_i$ **then**
        Pass counterexample x_i to A_i
        Update S_i to new query
  for *i = 1 . . . k* **do**
    if A_i *outputs* c_i **then**
      Set $S_i := c_i$
return $\prod c_i$

Algorithm 3: Algorithm for learning from Superset Queries

Appendix C. Learning From Membership Queries and One Positive Example

We prove results relating to learning from membership queries and one positive example.

Proposition 10 *Let $\boldsymbol{x}$ be a negative example. Given a positive example, Algorithm C returns an i such that $x_i \notin c^*_i$ using at most $lg(k)$ membership queries.*

Proof We show by induction that at each step of the algorithm, there is an index $i \in [low, 2 \cdot up - low]$ such that $x_i \notin c^*_i$. Base: Since $up = \lceil k/2 \rceil$ and $low = 1$ the range $[low, 2 \cdot up - low]$ equals $[1, k]$. By Observation 1 there is some $x_i \notin c^*_i$, since $\boldsymbol{x} \notin c^*$. Inductive: assume this has held for all previous steps. So either there is an i in $[low, up]$ or in $[up, 2 \cdot up - low]$ such that $x_i \notin c^*_i$. By the construction of $\boldsymbol{p'}$ and Observation 1, if $i \in [low, up]$, then $\boldsymbol{p'} \in c^*$. So the new up (call it up') will be set to $low + \lceil (up - low)/2 \rceil$, so $[low, up] = [low, 2 \cdot up' - low]$ and the property still holds. If $i \notin [low, up]$, then $i \in [up, 2 \cdot up - low]$ and $\boldsymbol{p'} \notin c^*$. In this case, the new low and up (low' and up') will be up and $up + \lceil (up - low)/2 \rceil$, respectively. So $[up, 2 \cdot up - low] = [low', 2 \cdot (up') - low']$ and the property still holds. Since the size of $up - low$ decreases by one half each round, after $lg(k)$ rounds $low = up$ and so $i \in [low, 2 \cdot up - low] = [low, low]$. ■

Input: $\boldsymbol{p}$: positive example (in c^*) $\boldsymbol{x}$: counter example
Output: i such that $\boldsymbol{x}[i] \notin c^*_i$
HandleCounterexample *($\boldsymbol{p}$, $\boldsymbol{x}$):*

```
// lower and upper bounds
low := 1  up := ⌈k/2⌉
for j = 1 ... lg(k) do
    p' := p
    p'[i] := x[i] for low ≤ i ≤ up
    size := up − low
    Query p' ∈ c*
    if p' ∉ c* then
        low := up
    up := low + ⌈size/2⌉
return low
```

Algorithm 4: Find dimension on which counterexample fails

Theorem 2: Assume a single positive example $\boldsymbol{p} \in c^*$ is given.

- If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem\}$, then c^* is learnable in $\sum \#Mem_i(c^*_i)$ membership queries.
- If $\mathcal{Q}_{subc} = \{EQ\}$ (respectively $\mathcal{Q}_{subc} = \{Sub\}$) and $\mathcal{Q}_{prod} = \{EQ, Mem\}$ (respectively $\mathcal{Q}_{prod} = \{Sub, Mem\}$), then c^* is learnable in $lg(k) \cdot \sum \#q_i(c^*_i)$ membership queries and $\sum \#EQ_i(c^*_i)$ equivalence queries (respectively $\sum \#Sub_i(c^*_i)$ subset queries).
- If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem, EQ\}$, then c^* is learnable in $lg(k) \cdot \sum \#EQ_i(c^*_i) + \sum \#Mem_i(c^*_i)$ membership queries and $\sum \#EQ_i(c^*_i)$ equivalence queries.

Proof

Proof of Item 1 See Algorithm C in this appendix. The algorithm learns by simulating each A_i in sequence, moving on to A_{i+1} once A_i returns a hypothesis c_i. For any membership query M_i made by A_i, $M_i \in c^*_i$ if and only if $\boldsymbol{p}[i \leftarrow M_i] \in c^*$ by Observation 1. Therefore the algorithm is successfully able to simulate the oracle for each A_i, yielding a correct hypothesis c_i.

Proof of Item 2 The learning process for either subset or equivalence queries is described in Algorithm C, with differences marked in comments. In either case, once the correct c_j is found for any j, S_j will equal c_j for all future queries, so any counterexamples must fail on an $i \neq j$.

We separately show for each type of query that a correct answer is given to at least one learner A_i for each subset (resp. equivalence) query to the cross-product oracle. Moreover, at most $lg(k)$ membership queries are made per subset (resp. equivalence) query, yielding the desired bound.

Subset Queries: For each subset query $\prod S_i \subseteq c^*$, the algorithm either returns 'yes' or gives a counterexample $\boldsymbol{x} = (x_1, \ldots, x_k) \in \prod S_i \backslash c^*$. If the algorithm returns 'yes', then by Observation 2 $S_i \subseteq c_i^*$ for all i, so the algorithm can return 'yes' to each A_i. Otherwise, $\boldsymbol{x} \notin c^*$, so there is an i such that $x_i \notin c_i^*$. Algorithm C is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries.

Equivalence Queries: For each equivalence query $\prod S_i = c^*$, the algorithm either returns 'yes', or gives a counterexample $\boldsymbol{x} = (x_1, \ldots, x_k)$. If the algorithm returns 'yes', then a valid target concept is learned. Otherwise, either $\boldsymbol{x} \in \prod S_i \backslash c^*$ or $\boldsymbol{x} \in c^* \backslash \prod S_i$. In the second case, as with superset queries, Algorithm C is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries. Once the $x_i \notin c_i^*$ is found it is given to A_i as a counterexample.

Proof of Item 3 The learning algorithm is described in Algorithm 1. The algorithm uses the positive example to answer membership queries. By Observation 1, for any membership query x_i made by A_i, $x_i \in c_i^*$ if and only if $\boldsymbol{p}[i \leftarrow x_i] \in c^*$. So each membership query posed by an A_i is answered with one membership query posed by the cross-product learner.

Membership queries are answered until each A_i poses an equivalence query $S_i = c_i^*$ (if A_i has terminated with the correct answer, we just assume S_i equals c_i^*). The learning algorithm then queries $\prod S_i = c^*$ and receives a counterexample $\boldsymbol{x} := (x_1, \ldots, x_k)$ or it receives a 'yes' and terminates. The algorithm checks if $\boldsymbol{x} \in \prod S_i$ and handles each case seperately.

If $\boldsymbol{x} \in \prod S_i$: then $\boldsymbol{x} \in (\prod S_i) \backslash c^*$. So there is an i such that $x_i \notin c_i^*$. Algorithm C is used to find the $x_i \notin c_i^*$ in $lg(k)$ queries. Since $\boldsymbol{x} \in \prod S_i$, $x_i \in S_i \backslash c_i^*$, so x_i is passed to A_i as a counterexample to the query $S_i = c_i^*$. This takes at most k membership queries.

If $\boldsymbol{x} \notin \prod S_i$: then $\boldsymbol{x} \in c^* \backslash \prod S_i$, since it is a counterexample. So there is an i such that $x_i \notin S_i$. Since the algorithm has access to each S_i, it can check this explicitly without using any counter-examples.

In either case, at least one A_i receives a counterexample to its equivalence query, so this process is done at most $\sum \#EQ_i(c_i^*)$ times, using at most k membership queries per process. This yields the stated bound on query complexity. ■

for $i = 1 \dots k$ **do**
  Set S_i to initial query from A_i
while *Some A_i has not completed* **do**
  Query $\prod S_i$ to oracle
  if *the oracle returns 'yes'* **then**
    Pass 'yes' to each A_i `// If` $\mathcal{Q}_{subc} = \{EQ\}$ `each sublearner will immediately complete`
  else
    Get counterexample $\boldsymbol{x} = (x_1, \dots, x_k)$
    if $\boldsymbol{x} \in c^* \setminus \prod S_i$ **then**
      `// Only happens if` $\mathcal{Q}_{subc} = \{EQ\}$
      for $i = 1 \dots k$ **do**
        if $x_i \notin S_i$ **then**
          Pass counterexample x_i to A_i
          Update S_i to new query from A_i
    else
      for $i = 1 \dots k$ **do**
        Query $\boldsymbol{p}[i \leftarrow x_i] \in c^*$ **if** $\boldsymbol{p}[i \leftarrow x_i] \notin c^*$ *and* $x_i \in S_i$ **then**
          Pass counterexample x_i to A_i
          Update S_i to new query from A_i
Each A_i returns some c_i **return** $\prod c_i$

Algorithm 5: Learn when $\mathcal{Q}_{subc} = \{EQ\}$ (or $\mathcal{Q}_{subc} = \{Sub\}$) and $\mathcal{Q}_{prod} = \mathcal{Q}_{subc} \cup \{Mem, 1Pos\}$

Input: $\boldsymbol{p}$: Positive Example in X
Learn *($\boldsymbol{p}$):*
  for $i = 1 \dots k$ **do**
    while A_i *has not completed* **do**
      Get query $x_i \in c_i^*$ from A_i
      Query $\boldsymbol{p}[i \leftarrow x_i] \in c^*$
      Pass answer to A_i
      if A_i *returns guess* c_i **then**
        Break
  return $\prod c_i$

Algorithm 6: Learn from Membership Queries and One Positive Example

Appendix D. Learning From Only Membership Queries

Observation 3 *In general, Algorithm 5 cannot distinguish determine if $c^* = \emptyset$ using only membership queries, even when subconcepts are learnable with only membership queries*

Proof Consider the concept classes $C_1 = \{\{1\}, \emptyset\}$ and $C_2 = \{\{j\} \mid j \in \mathbb{N}\}$. We can find the correct concepts in C_1 or in C_2 using membership queries. For any finite number of membership queries, there is no way to distinguish between the sets $\emptyset$ and $\{(1, j)\}$ for some j that has yet to be queried. ■

Proposition 3 If $\mathcal{Q}_{subc} = \mathcal{Q}_{prod} = \{Mem, EQ\}$, then Algorithm 5 can find a positive example using $max_i\{(\#Mem_i + \#EQ_i)\}^k$ membership queries and at most one equivalence query.

Proof

Since the algorithm might give inconsistent answers to the sublearners, there is no guarantee that A_i can always give a new query. This is handled in the case marked "If no possible query" in the algorithm.

At the start of the algorithm, if the concept class includes the empty concept, the algorithm queries $\emptyset = c^*$. If $\emptyset = c^*$, the concept is learned. Otherwise, some positive counter-example from c^* must be given. The rest of the algorithm then assumes that is not a valid hypothesis.

The remaining part of the algorithm simulates each A_i in parallel until every T_i contains an element of c_i^*. At this point, the algorithm will stop, since all possible elements of $\prod T_i$ are posed as membership queries.

For each A_i, either every answer to a query from A_i is correct or at least one answer is incorrect. We will discuss each case separately. Every answer is correct: In this case A_i will eventually query the correct hypothesis c_i^*. Since $c_i^* \neq \emptyset$, some element of c_i^* will then be added to T_i. Some answer to A_i is incorrect: If an incorrect answer to a membership query is given, then for some query $x_i \in c_i^*$, the answer "False" is incorrect. So $x_i \in c_i^*$ and T_i will contain a positive example. If an incorrect answer to an equivalence query is given, either the counter-example is incorrect or the statement that they are not equivalent is incorrect. If an incorrect counterexample y_i to the query $S_i = c_i^*$ is given, then $y_i \in c_i^*$, since we already know $y_i \in S_i$. If the statement $S_i \neq c_i^*$ is incorrect, then some element of c_i^* will then be added to T_i.

The algorithm adds at most one element to T_i per query and must stop once every A_i has made enough queries to learn c_i^*, yielding the stated bound.

■

Appendix E. Learning From Equivalence and Subset Queries is Hard

This appendix proves the propositions and theorems showing learning from equivalence or subset queries is hard. We will start by showing $\mathfrak{C}$ is easy to learn.

An important part of the construction of $\mathfrak{C}$ is that for any two strings $s, s' \in \mathbb{N}$, we have that $\mathfrak{c}(s) \subseteq \mathfrak{c}(s')$ if and only if $s = s'$. This implies that a subset query will return true if and only if the true concept has been found. Moreover, an adversarial oracle can always give a negative example for an equivalence query, meaning that oracle can give the same counterexample if a subset query were posed. So we will show that $\mathfrak{C}$ is learnable from equivalence queries, implying that it is learnable from subset queries.

Proposition 4 There exist algorithms for learning from equivalence queries or subset queries such that any concept $\mathfrak{c}(s) \in \mathfrak{C}$ can be learned from $|s|$ queries.

Proof Algorithm E shows the learning algorithm for equivalence queries, and Figure 5 show the decision tree. This algorithm starts by querying $\mathfrak{c}(\lambda)$ to an oracle. When learning $\mathfrak{c}(s)$ for any $s \in \mathbb{N}^*$, the algorithm will construct s by learning at least one new element of s per query. Each new query to the oracle is constructed from a string that is a substring of s If a positive counterexample is given, this can only yield a longer substring of s and so learning is done in less than $|s|$ time. ■

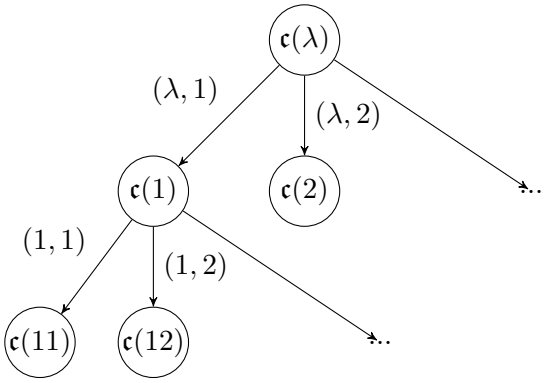

Figure 5: A tree representing Algorithm E. Nodes are labelled with the queries made at each step, and edges are labelled with the counterexample given by the oracle.

Result: Learns $\mathfrak{C}$
Set $s = \lambda$ **while** *True* **do**
  Query $\mathfrak{c}(s)$ to Oracle **if** *Oracle returns 'yes'* **then**
    return $\mathfrak{c}(s)$
  end
  if *Oracle returns* $(s', m) \in c^* \backslash \mathfrak{c}(s)$ **then**
    Set $s = s'$
  end
  if *Oracle returns* $(s, m) \in \mathfrak{c}(s) \backslash c^*$ **then**
    Set $s = sm$
  end
end

Algorithm 7: Learning $\mathfrak{C}$ from equivalence queries.

We will prove a lower-bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. This will imply that $\mathfrak{C}^k$ is hard to learn from equivalence queries, since an adversarial equivalence query oracle can give the exact same answers and counterexamples as a subset query oracle.

First, we need a couple definitions.

A concept $\prod \mathfrak{c}(s_i)$ is *justifiable* if one of the following holds:

- For all i, $s_i = \lambda$
- There is an i and an $a \in \mathbb{N}$ and $w \in \mathbb{N}^*$ such that $s_i = wa$, and the k-ary cross-product $\mathfrak{c}(s_1) \times \cdots \times \mathfrak{c}(w) \times \cdots \times \mathfrak{c}(s_k)$ was justifiably queried to the oracle and received a counterexample $\boldsymbol{x}$ such that $\boldsymbol{x}[i] = (w, a)$.

A concept is *justifiably queried* if it was queried to the oracle when it was justifiable.

For any strings $s, s' \in \mathbb{N}^*$, we write $s \leq s'$ if s is a substring of s', and we write $s < s'$ if $s \leq s'$ and $s \neq s'$. We say that the *sum of string lengths* of a concept $\prod \mathfrak{c}(s_i)$ is of size r if $\sum |s_i| = r$

Proving that learning is hard in the worst-case can be thought of as a game between learner and oracle. The oracle can answer queries without first fixing the target concept. It will answer queries so that for any n, after less than k^n queries, there is a concept consistent with all given

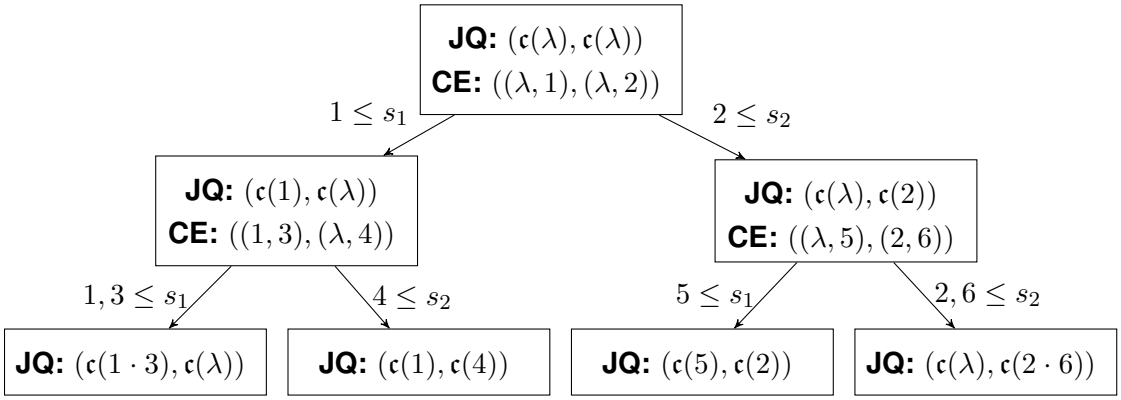

Figure 6: The tree of justifiable queries used in Example 2. Each node lists the justifiable query (JQ) and counterexample (CE) given for that query. The edges below each node are labelled with the possible inferences about s_1 and s_2 that can be drawn from the counterexample.

oracle answers that the learning algorithm will not have guessed. The specific behavior of the oracle is defined as follows:

- It will always answer the same query with the same counterexample.
- Given any query $\prod \mathfrak{c}(s_i) \subseteq c^*$, the oracle will return a counterexample $\boldsymbol{x}$ such that for all i, $\boldsymbol{x}[i] = (s_i, a_i)$, and a_i has not been in any query or counterexample yet seen.
- The oracle never returns 'yes' on any query.

The remainder of this section assumes that queries are answered by the above oracle. An example of answers by the above oracle and the justifiable queries it yields is given below.

Example 3 *Consider the following example when $k = 2$. First, the learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$ to the oracle and receives a counter-example $((\lambda, 1), (\lambda, 2))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$. The learner queries $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and receives counterexample $((1, 3), (\lambda, 4))$. The learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$ and receives counterexample $((\lambda, 5), (2, 6))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(4))$, $(\mathfrak{c}(1 \cdot 3), \mathfrak{c}(\lambda))$, $(\mathfrak{c}(5), \mathfrak{c}(2))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2 \cdot 6))$. At this point, these are the only possible solutions whose sum of string lengths is 2. The graph of justifiable queries is given in Figure 6.*

The following simple proposition can be proven by induction on sum of string lengths.

Proposition 11 *Let $\prod \mathfrak{c}(s_i)$ be a justifiable concept. Then for all w_1, w_2, ..., w_k where for all i, $w_i \leq s_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle.*

Proposition 12 *If all justifiable concepts $\prod \mathfrak{c}(s_i)$ with sum of string lengths equal to r have been queried, then there are k^{r+1} justifiable queries whose sum of string lengths equals $r + 1$*

Proof This proof follows by induction on r. When $r = 0$, the concept $\prod \mathfrak{c}(\lambda)$ is justifiable.For induction, assume that there are k^r justifiable queries with sum of string lengths equal to r. By construction, the oracle will always chose counterexamples with as-yet unseen values in $\mathbb{N}$. So querying each concept $\prod \mathfrak{c}(s_i)$ will yield a counterexample $\boldsymbol{x}$ where for all i, $\boldsymbol{x}[i] = (s_i, a_i)$ for new a_i. Then for all i, this query creates the justifiable concept $\prod \mathfrak{c}(s'_j)$, where $s'_j = s_j$ for all

$j \neq i$ and $s'_i = \mathfrak{c}(s_i \cdot a_i)$. Thus there are k^{r+1} justifiable concepts with sum of string lengths equal to $r+1$. ■

We are finally ready to prove the main theorem of this section.

Theorem 5 Any algorithm learning $\mathfrak{C}^k$ from subset (or equivalence) queries requires at least k^r queries to learn a concept $\prod \mathfrak{c}(s_i)$, whose sum of string lengths is r. Equivalently, the algorithm takes $k^{\sum \#q_i}$ subset (or equivalence) queries.

Proof Assume for contradiction that an algorithm can learn with less than k^r queries and let this algorithm converge on some concept $c = \prod \mathfrak{c}(s_i)$ after less than k^r queries. Since less than k^r queries were made to learn c, by Proposition 12, there must be some justifiable concept $c' = \prod \mathfrak{c}(s'_i)$ with sum of string lengths less than or equal to r that has not yet been queried. By Proposition 11, we can assume without loss of generality that for all $w_i \leq s'_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle. We will show that c' is consistent with all given oracle answers, contradicting the claim that c is the correct concept. Let $c_v := \prod \mathfrak{c}(v_i)$ be any concept queried to the oracle, and let $\boldsymbol{x}$ be the given counterexample. If for all i, $v_i \leq s'_i$, then by construction, there is a j with $\boldsymbol{x}[j] = (v_j, a_j)$ such that $v_j \cdot a_j \leq s'_j$, so $\boldsymbol{x}$ is a valid counterexample. Otherwise, there is an i such that $v_i \not\leq s'_i$. So $(\{v_i\} \times \mathbb{N}) \cap \mathfrak{c}(s'_i) = \emptyset$, so $\boldsymbol{x}$ is a valid counterexample. Therefore, all counterexamples are consistent with c' being correct concept, contradicting the claim that the learner has learned c. ■

Appendix F. PAC Learning

This section discusses the problem of PAC-learning the cross-products of concept classes.

Previously, Van Der Vaart and Wellner (2009) have shown the following bound on the VC-dimension of cross-products of sets:

$$\mathcal{VC}(\prod C_i) \leq a_1 log(k a_2) \sum \mathcal{VC}(C_i)$$

Here a_1 and a_2 are constants with $a_1 \approx 2.28$ and $a_2 \approx 3.92$. As always, k is the number of concept classes included in the cross-product.

The VC-dimension gives a bound on the number of labelled examples needed to PAC-learn a concept, but says nothing of the computational complexity of the learning process. This complexity mostly comes from the problem of finding a concept in a concept class that is consistent with a set of labelled examples. We will show that the complexity of learning cross-products of concept classes is a polynomial function of the complexity of learning from each individual concept class, for a fixed VC-dimension. The algorithm's complexity increases exponentially with the VC-dimension.

First, we will describe some necessary background information on PAC-learning.

F.1. PAC-learning Background

Definition 13 *Let C be a concept class over a space X. We say that C is efficiently PAC-learnable if there exists an algorithm A with the following property: For every distribution $\mathcal{D}$ on X, every $c^* \in C$, and every $\epsilon, \delta \in (0,1)$, if algorithm A is given access to $EX(c^*, \mathcal{D})$ then with probability $1-\delta$, A will return a $c' \in C$ such that $error(c') \leq \epsilon$. A must run in time polynomial in $1/\epsilon$, $1/\delta$, and $size(c^*)$.*

The oracle $EX(c^*, \mathcal{D})$ samples elements from X according to distribution $\mathcal{D}$ and labels whether each element is in c^*. We will refer to ϵ as the 'accuracy' parameter and δ as the 'confidence' parameter. The value of $error(c)$ is the probability that for an x sampled from $\mathcal{D}$ that $c(x) \neq c^*(x)$. PAC-learners have a *sample complexity* function $m_C(\epsilon, \delta) : (0,1)^2 \to \mathbb{N}$. The sample complexity is the number of samples an algorithm must see in order to PAC-learn a concept with parameters ϵ and δ.

Given a set S of labelled examples in X, we will use $A(S)$ to denote the concept the algorithm A returns after seeing set S .

A learner A_i is an *empirical risk minimizer* (ERM) if $A_i(S)$ returns a $c \in C$ that minimizes the number of misclassified examples (i.e., it minimizes $|\{(x, b) \in S \mid c(x) \neq b\}|$). A well-known theorem restated in the next subsection shows that an ERM can PAC-learn a concept from polynomially many examples. Our goal is thus to show how to construct an ERM for the cross-products from the learning algorithms A_1 and A_2.

F.2. PAC-Learning Cross-Products

We now have enough background to describe the strategy for PAC-learning cross-products. For ease of explanation, we will just describe learning the cross-product of two concepts. As above, assume concept classes C_1 and C_2 and PAC-learners A_1 and A_2 are given. We define $T_i(\epsilon, \delta)$ as the runtime of the sublearner A_i to PAC-learn with accuracy parameter ϵ and confidence parameter δ.

Assume that C_1 and C_2 have VC-dimension d_1 and d_2, respectively. We can use the bound from van Der Vaart and Weller to get an upper bound d on the VC-dimension of their cross-product. Assume the algorithm is given an ϵ and δ and there is a fixed target concept $c^* = c_1^* \times c_2^*$. Theorem 18, which is well known in the PAC-literature, is restated below (Theorem 6.7 from Shalev-Shwartz and Ben-David (2014)). It gives a bound on the sample complexity $m_{C_1 \times C_2}(\epsilon, \delta)$.

Theorem 14 *If the concept class C has VC dimension d, then there is a constant, b, such that applying an Empirical Risk Minimizer A to $m_C(\epsilon, \delta)$ samples will PAC-learn in C, where*

$$m_C(\epsilon, \delta) \leq b \frac{d \cdot log(1/\epsilon) + log(1/\delta)}{\epsilon}$$

The algorithm will take a sample of labelled examples of size $m_{C_1 \times C_2}(\epsilon, \delta)$. Our goal is to construct an Empirical Risk Minimizer for $C_1 \times C_2$. In our case, the target concepts c_1^* and c_2^* are in C_1 and C_2, respectively. Therefore, for any sample S, an Empirical Risk Minimizer will yield a concept in $C_1 \times C_2$ that is consistent with S. This process is shown in Algorithm F.3.

We will now argue that Algorithm F.3 is correct. Let S be any such sample the algorithm takes. This set can easily be split into positive examples S^+ and negative examples S^-, both in $X_1 \times X_2$. The algorithm works by maintaining sets labeled samples L_1 and L_2 for each dimension. For any $(x_1, x_2) \in S^+$, it holds that $x_1 \in c_1^*$ and $x_2 \in c_2^*$ so $(x_1, \top)$ and $(x_2, \top)$ are added to L_1 and L_2 respectively. For any $(x_1, x_2) \in S^-$, we know that $x_1 \notin c_1^*$ or $x_2 \notin c_2^*$ (or both), but it is not clear which is true. However, since the goal is only to create an Empirical Risk Minimizer, it is enough to find any concepts C_1 and C_2 that are consistent with these samples. Let $S_1^- := \{x \mid \exists y, (x, y) \in S^-\}$, let $m = |S_1^-|$ and order the elements of S_1^- by $x_1, x_2, \ldots, x_m$. The following lemma gives a bound on the number of concepts consistent with these examples.

Lemma 15 $|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq (em/d)^d$

Proof By the definition of growth function, $|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq G_{C_1}(m)$. By lemma 19, $G_{C_1}(m) \leq (em/d)^d$. ■

In other words, there are less than $(em/d)^d$ assignments of truth values to elements of S_1^- that are consistent with some concept in C_1. If the algorithm can check every $c_1 \in C_1$ consistent with S^+ and S_1^-, it can then call A_2 to see if there is any $c_2 \in C_2$ such that $(c_1 \times c_2)$ assigns true to every element in S^+ and false to every element in S^-.

Finding these consistent elements of C_1 is made easier by the fact that we can check whether partial assignments to S_1^- are consistent with any concept in C_1. As mentioned above, it starts

by creating the sets L_1 and L_2 containing all samples in the first and second dimension of S^+, respectively. It then iteratively adds labeled samples from S^-. At each step, the algorithm chooses one element $(x_1, x_2) \in S^-$ at a time and checks which possible assignments to x_1 are consistent with L_1. If $(x_1, \bot)$ is consistent, it adds $(x_1, \bot)$ to L_1 and calls $RecursiveFindSubconcepts$ on L_1 and L_2. If $(x_1, \top)$ is consistent with C_1, then the algorithm adds $(x_1, \top)$ to L_1 and $(x_2, \bot)$ to L_2 and calls RFS (RecursiveFindSubconcepts). In either case, if an assignment is not consistent, no recursive call is made. We can summarize these results in the following theorem.

Theorem 16 *Let concept classes C_1 and C_2 have VC-dimension d_1 and d_2, respectively. There exists a PAC-learner for $C_1 \times C_2$ that can learn any concept using a sample of size $m = ((d_1+d_2)\cdot log(1/\epsilon)+log(1/\delta))/\epsilon$. The learner requires time $O(m^{d_1}(T_1(1/m, log(\delta)) + T_2(1/m, log(\delta))))$.*

F.3. Efficient PAC-learning with Membership Queries

Although polynomial for a fixed VC-dimension, the complexity of PAC-learning cross-products from a *EX* oracle is fairly expensive. We will show that when a learner is allowed to make membership queries, PAC-learning cross-products becomes much more efficient. This is due to the previously shown technique, which uses membership queries and a single positive example to determine on which dimensions a negatively labelled example fails.

In this case, assuming that $\emptyset \in \boxtimes C_i$, we can ignore the assumption that a positive example is given. If no positive example appears in a large enough labeled sample, the algorithm can return $\emptyset$ as the learned hypothesis.

If S does contain a positive example $\boldsymbol{p}$, then S can be broken down into labeled samples for each dimension i. The algorithm initialize the sets of positive and negative examples to $S_i^+ := \{\boldsymbol{x}[i] \mid (\boldsymbol{x}, \top) \in S\}$ and $S_i^- := \{\}$, respectively. For each $(\boldsymbol{x}, \bot) \in S$, a membership queries $\boldsymbol{p}[i \leftarrow \boldsymbol{x}[i]] \in c^*$. If so, $\boldsymbol{x}[i]$ is added to S_i^+. Otherwise it is added to S_i^-. This labelling is correct by Observation 1. The set of labelled examples $S_i := (S_i^+ \times \{\top\}) \cup (S_i^- \times \{\bot\})$ is then passed to the sublearner A_i. A_i is run on S_i with accuracy parameter $\epsilon' := \epsilon/k$ and confidence parameter $\delta' := \delta/k$.

Proposition 17 *The algorithm described above PAC-learns from the concept class $\boxtimes C_i$ with accuracy ϵ and confidence δ. It makes $m_C(\epsilon, \delta)$ queries to* EX*, $k \cdot m_C(\epsilon, \delta)$ membership queries, and has runtime $O(\sum T_i(\epsilon/k, \delta/k))$.*

Result: Find Subconcepts Consistent with Sample
Input: S^+: Set of positive examples in $X_1 \times X_2$
S^-: Set of negative examples in $X_1 \times X_2$
δ : Confidence parameter in $(0, 1)$
FindSubconcepts *(S^+, S^-, δ)*
  $\delta' := \delta/(|S^-|G_{C_1}(|S^-|) + G_{C_2}(|S^-|))$
  $\epsilon' := 1/|S|$
  $L_1 := \{(x_1, \top) \mid \exists y, (x_1, y) \in S^+\}$
  $L_2 := \{(x_2, \top) \mid \exists y, (y, x_2) \in S^+\}$
  $U := S^-$
  return RFS(L_1, L_2, U, ϵ', δ')
Algorithm 8: PAC Learning

```
// Recursive Find Subconcepts
```
RFS *(L_1, L_2, U, ϵ', δ'):*
  if $U = \emptyset$ **then**
    if $A_2(L_2, \epsilon', \delta')$ **then**
      return $(A_1(L_1, \epsilon', \delta'), A_2(L_2, \epsilon', \delta'))$
    else
      return $\perp$
  Get $(x_1, x_2) \in U$
  $U := U \backslash \{(x_1, x_2)\}$
  `// Tries to label` x_1 `false`
  if $A_1(L_1 \cup \{(x_1, \perp)\}, \epsilon', \delta') \neq \perp$ **then**
    $L_1' := L_1 \cup \{(x_1, \perp)\}$
    $c := RFS(L_1', L_2, U, \epsilon', \delta')$
    if $c \neq \perp$ **then**
      return c
  `// Tries to label` x_1 `true`
  if $A_1(L_1 \cup \{(x_1, \top)\}, \epsilon', \delta') \neq \perp$ **then**
    $L_1' := L_1 \cup \{(x_1, \top)\}$
    $L_2' := L_2 \cup \{(x_2, \perp)\}$
    $c := RFS(L_1', L_2', U, \epsilon', \delta')$
    if $c \neq \perp$ **then**
      return c

We give relevant theorems and prove a lemma for PAC learning cross-products. The following theorem is well-known in PAC literature (Theorem 6.7 from Shalev-Shwartz and Ben-David (2014))

Theorem 18 *If the concept class C has VC dimension d, then there is a constant, b, such that applying an Empirical Risk Minimizer A to $m_C(\epsilon, \delta)$ samples will PAC-learn in C, where*

$$m_C(\epsilon, \delta) \leq b\frac{d \cdot log(1/\epsilon) + log(1/\delta)}{\epsilon}$$

The *growth function* describes how many distinct assignments a concept class can make to a given set of elements. More formally, for a concept class C and $m \in \mathbb{N}$, the growth function $G_C(m)$ is defined by:

$$G_C(m) = \max_{x_1, x_2, \ldots, x_m} \left|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C\}\right|$$

Each x_i in the above equation is taken over all possible elements of X_i. The VC-dimension of a class C is the largest number d such that $G_C(d) = 2^d$.

We will use the following bound, a corollary of the Perles-Sauer-Shelah Lemma, to bound the runtime of learning cross-products Shalev-Shwartz and Ben-David (2014).

Lemma 19 *For any concept class C with VC-dimension d and $m > d + 1$:*

$$G_C(m) \leq (em/d)^d$$

Lemma 15: Given the elements $x_1, \ldots, x_m$ described in the paper, $|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq (em/d)^d$

Proof By the definition of growth function, $|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq G_{C_1}(m)$. By lemma 19, $G_{C_1}(m) \leq (em/d)^d$. ■