It's Not What Machines Can Learn, It's What We Cannot Teach

Gal Yehuda
Technion – Israel Institute of Technology
ygal@cs.technion.ac.il

Moshe Gabel
University of Toronto
mgabel@cs.toronto.edu

Assaf Schuster
Technion – Israel Institute of Technology
assaf@cs.technion.ac.il

Abstract

Can deep neural networks learn to solve any task, and in particular problems of high complexity? This question attracts a lot of interest, with recent works tackling computationally hard tasks such as the traveling salesman problem and satisfiability. In this work we offer a different perspective on this question. Given the common assumption that *NP* $\neq$ *coNP* we prove that any polynomial-time sample generator for an *NP*-hard problem samples, in fact, from an easier sub-problem. We empirically explore a case study, Conjunctive Query Containment, and show how common data generation techniques generate biased datasets that lead practitioners to over-estimate model accuracy. Our results suggest that machine learning approaches that require training on a dense uniform sampling from the target distribution cannot be used to solve computationally hard problems, the reason being the difficulty of generating sufficiently large and unbiased training sets.

1. Introduction

Applying deep learning methods to solve computationally hard problems has gained popularity in recent years. Examples include attempts to solve the satisfiability problem (Selsam et al., 2018), the traveling salesman problem (TSP) (Prates et al., 2019; Milan et al., 2017) and symbolic integration (Lample & Charton, 2020). There has also been recent interest in developing dedicated architectures for learning how to perform algorithmic tasks from solved instances, such as the Neural Turing Machine (Graves et al., 2014), the Differentiable Neural Computer (Graves et al., 2016), and the Neural GPU (Kaiser & Sutskever, 2015).

The *expressive power* of deep neural networks, which represents the breadth of functions deep models are able to compute, has been an active area of research since the rise of deep learning (Siegelmann & Sontag, 1991; Raghu et al., 2017; Lu et al., 2017). We know that recurrent neural networks and many modern architectures are Turing complete (Prez et al., 2019), meaning they are capable of performing any computation that a Turing machine can do. This raises the intriguing possibility of discovering efficient approximate solvers by using machine learning to train a model on solved instances of a given problem. However, even if a model is expressive enough in theory, we must also be able to train it to arrive at the correct solution.

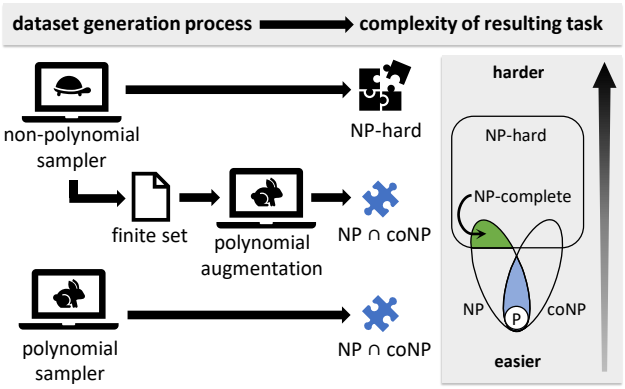

Figure 1: Assuming *NP* $\neq$ *coNP*, any polynomial-time data generator for a *NP*-hard classification task will output data from an easier *NP* $\cap$ *coNP* task, even when starting with a sample generated by a non-polynomial deterministic process and augmenting it.

The difficulty emerges in acquiring a suitable dataset. Large, diverse and densely sampled datasets are essential for the learning ability of deep learning models (Chollet, 2017). Existing datasets for computational tasks tend to be application specific; such datasets may also be biased towards a subset of the problem space, which may be easy and unrepresentative. For example, training a model to answer the 3-SAT problem using a dataset where all examples follow a simple pattern may yield high accuracy for similar data without capturing the full essence and difficulty of the problem in the resulting model. A trivial example of such a pattern is

when all the positive instances are shorter than the negative instances. A more subtle case is when all samples are instances of an easy problem that is hard to identify at first glance yet for which an efficient solution is known, such as 2-colorability. In both cases, solving the problem on these datasets does not mean solving the broader problem. Since the performance of such models is measured empirically, a biased, possibly easy dataset may lead us to falsely believe the models are solving the general problem.

For abstract computational tasks such as 3-SAT and TSP, a popular alternative to using existing datasets is generating solved instances (Selsam et al., 2018; Prates et al., 2019). Such dataset generators can generate as many samples as we wish, which is particularly appealing when training models that require large training sets. Moreover, performance evaluation can be more precise since we can generate as many samples as we need to reduce the generalization gap.

Dataset generators, however, are not without issues. Labeling datasets for *NP*-hard classification tasks requires deterministic solvers whose runtime grows exponentially or worse with the problem size (Kovács & Voronkov, 2013), which is impractical for the large training sets needed by popular ML approaches. Instead, practitioners turn to alternative approaches that run in polynomial, often linear time. One common approach is starting with a random example and carefully applying transformations so that the label is known by construction (Lample & Charton, 2020). Another approach is data augmentation: start with a seed set of deterministically-labeled samples and apply class-preserving rewrites (Selsam et al., 2018). It is not uncommon for test sets to be generated using the same procedure.

Our Contributions We show that polynomial-time dataset generators cannot be used to train models in solving *NP*-hard problems. If a classification task is an *NP*-hard decision problem, any *efficient* (polynomial time) procedure generates biased, unrepresentative data sets of solved instances, unless *NP* = *coNP*. In other words, when starting with an *NP*-hard problem, the data sampling procedure leaves us with an easier problem that we train the model on. Figure 1 illustrates this result. Finally, we show an example of the worst case scenario: an *NP*-hard language for which any polynomial-time dataset generator creates a trivial classification task.

Specifically, under the commonly accepted assumption that *NP* $\neq$ *coNP* we prove the following:

1. No polynomial-time data generation procedure can ever sample from the full problem space.
2. The classification task that a polynomial-time data generator can sample from is an *NP* $\cap$ *coNP* decision problem, strictly easier than the original problem.
3. There is a language that is *NP*-hard to decide, yet any polynomial-time procedure that generates samples from it creates samples that can with high probability be classified using a superficial feature.

As a case study, we consider the *NP*-complete problem of Conjunctive Query Containment,or CQC (Chandra & Merlin, 1977; Chirkova, 2018). We use a data augmentation approach to quickly generate large training sets of solved CQC instances, and train a neural network model to solve it. We demonstrate how training on the generated dataset is not enough for solving the original CQC task, and that using the same procedure to generate the test set can lead us to overestimate model performance.

In summary, we show for *NP*-hard problems, even if we had the right model architecture and training algorithm, we cannot feasibly obtain the data required in order to train them. Though a trained model may appear to solve the task on an efficiently generated dataset, it does not mean the trained model has learned to solve the original task.

2. Case Study: Leaning an *NP*-hard Problem

In this section we demonstrate how common and seemingly reasonable data generation approaches can cause us to overestimate model performance. We describe a representative case study: modeling, training, and evaluating a solver for the Conjunctive Query Containment (CQC) problem. CQC is a central problem in the theory of databases (Chirkova, 2018), motivated by both practical and theoretical interests, with applications in query minimization and optimization (Jarke & Koch, 1984), verifying data integrity (Florescu et al., 1999), cache management (Draper et al., 2001) and querying incomplete databases (Imieliński & Lipski, 1988).

2.1. Problem Definition

The problem of query containment is to decide, given two database queries p, q, if for every database D the results of p on D are contained in the results of q on D. For clarity, we focus on a simpler yet *NP*-complete version of this problem, with up to 2 relations and no projections.

A *database* $D = \{R_1, ..., R_k\}$ is a collection of tables, where each table R_i is collection of rows (tuples) of length 3. A *conjunctive query* q over the database D is a first order predicate of the form

$$\exists x_1, ..., x_n : R_{i_1}(\ell_1, \ell_2, \ell_3) \wedge \cdots \wedge R_{i_s}(\ell_{3s-2}, \ell_{3s-1}, \ell_{3s})$$

Where $x_1, ..., x_n$ are variables, and ℓ_j is either a variable (some x_u) or a constant c_w. We assume that all variables and constants take value from a finite set Σ. Given a conjunctive query q, we denote by $vars(q)$ the set of all variables in q. For example, the following is a query with 3 conjunctions:

$$q : R_1(x_4, x_3, x_1) \wedge R_1(x_1, x_2, x_1) \wedge R_2(x_5, x_1, x_2)$$

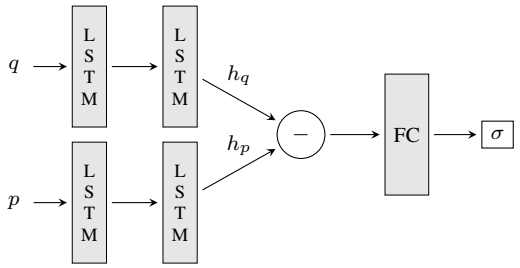

Figure 2: Model architecture

A tuple $(c_1, ..., c_n)$ *satisfies* the query q for database D if when assigning c_i to x_i the predicate is true. The *evaluation* of a query q on a database D, denoted by $q(D)$, is the collection of all tuples which satisfy q.

Conjunctive Query Containment (CQC) is the set of all pairs (p, q) of conjunctive queries such that $p(D) \subseteq q(D)$ for every database D; we denote such pairs by $p \subseteq q$. Deciding whether a query pair (p, q) is in CQC is *NP*-complete (Chandra & Merlin, 1977).

2.2. An RNN Model for CQC

Exact containment is *NP*-complete, so instead we aim to give an approximation using supervised learning: we will train a model to discriminate between query pairs. Given two queries q and p as a sequence of tokens, it will output 1 if $q \subseteq p$ or 0 if $q \not\subseteq p$.

Input Encoding Given a pair of conjunctive queries (p, q) and a binary label, we tokenize each query and represent it as a fixed length sequence of one-hot vectors with 42 dimensions (the number of tokens in our dictionary). The sequence length is 95, since this is the longest possible query with our parameters. We pad shorter queries with zero vectors. The full table of token encodings is available in the Appendix.

Model Architecture Since we aim to map sequences (query pairs) to scalars, we choose to use Recurrent Neural Networks with Long Short-Term Memory (LSTM) units[1], which excel at such tasks and have been shown to be computationally expressive (Prez et al., 2019; Weiss et al., 2018).

Figure 2 shows the network architecture for the model. We encode each query into a w-dimensional vector using LSTM layers with ReLU activations: two layers for p and two layers for q. The length of each layer is n, and the internal dimension (width) of the LSTM units is w. The final LSTM state vectors h_p and h_q are then subtracted from each other, resulting in the w-dimensional vector $v = h_q - h_p$. Finally, The vector v is fed to a fully connected layer that reduces v to a single scalar (i.e., a dot product), followed by a sigmoid activation function $\sigma(x)$ to normalize the output to the range $[0, 1]$. When $p \subseteq q$ the label will be 1, or 0 otherwise.

[1] We emphasize that our main results in Section 3 do not depend on any particular modeling choice, and apply equally to all approaches that require dense sampling. Nevertheless, we have also explored alternatives including Transformers and learned embeddings, with no meaningful difference in empirical performance or generalization. We discuss hybrid architectures in Section 5.

2.3. Data Generation

Simply generating random query pairs and labeling them using a determinstic CQC solver is not feasible, given the number of pairs we need and the large size of each query, and may also result in an unbalanced training set.

One common approach is to generate one input from the pair and work forwards or backwards to the other input by applying a sequence of rewrites that guarantee the pair's class (Lample & Charton, 2020). However this approach risks introducing superficial features and biasing the data towards unrealistic examples (Davis, 2019).

Instead, we aim to sample query pairs directly. We address class imbalance by sampling (p, q) from a special distribution μ such that $\Pr[p \subseteq q] \approx 0.5$, yet both positive and negative instances have the same structure (size, number of variables, etc.). We first generate a small "seed" set of query pairs by sampling from μ and labeling them using a deterministic theorem prover. We then use *data augmentation* to generate large training sets – a common approach for this problem (Selsam et al., 2018).

Generating Balanced Dataset When drawing samples from parametrized distribution, many *NP*-complete languages such as 3-SAT and TSP exhibits a *phase transition* phenomenon: the likelihood of a random sample drawn from a special parametric distribution to be in the language is determined by where the distribution's parameter α is in relation to constant c (Gent & Walsh, 1994; Zhang, 2004; Prates et al., 2019). We exploit a similar phenomena in CQC to draw balanced samples.

We define a parametric family of query pairs $\mu(m_1, m_2)$ such that sampling (p, q) from $\mu(m_1, m_2)$ with $m_1 \geq m_2$ guarantees the following properties. First, p has m_1 conjunctions and q has m_2 conjunctions. Second, the probability that $p \subseteq q$ is approximately 0.5. Finally, the process for generating positive and negative examples is the same. The definition and details of $\mu(m_1, m_2)$ are available in the Appendix.

We generate instances of (p, q), both positive and negative, where the number of conjunctions in p is 1–10, and the number of conjunctions in q is 1–8. We first choose $m_1 \sim U(1, 8)$ and $m_2 \sim U(1, \min\{m_1, 8\})$, and then sample

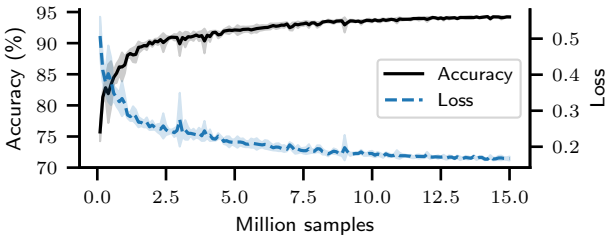

Figure 3: Average model accuracy and cross entropy loss throughout the learning process for the unseen AUG test set comprised of 500K queries. Lines show average of 5 models, bands show standard deviations.

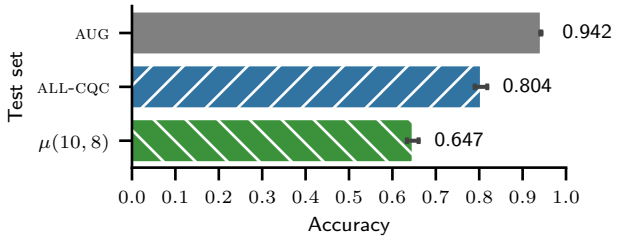

Figure 4: Average final accuracy for different test sets. Error bars show standard deviation. The high performance on the test set generated by data augmentation method does not translate to high performance on the other test sets.

$(p, q) \sim \mu(m_1, m_2)$. For each conjunction we choose a relation at random from $R = \{R_1, R_2\}$, with 3 variables or constants sampled uniformly with repetition from the set $\{x_0, \ldots, x_{32}, 0, 1\}$. Using $R = \{R_1, R_2\}$ is sufficient to make the problem *NP*-complete. We use the Vampire theorem prover (Kovács & Voronkov, 2013) to obtain the label for each sample.

Data Augmentation Though the time complexity of sampling from μ is linear, generating large training sets this way is infeasible since the deterministic theorem prover runs in exponential time in the worst case.

Instead, we augment every labeled sample in the seed set to create 99 additional samples with the same label. Starting with the original sample (p, q), we apply a sequence of up to 3 randomly selected class-preserving rewrites, yielding a new pair (p', q') with the same label. We repeat the process 98 more times, each time starting from the last (p', q'). Since the original seed set was balanced, this results in a dataset $\times 100$ larger with roughly half positive and half negative instances. Data augmentation runs in linear time.

An example of a class preserving rewrite is variable merging: if $p \subseteq q$, then merging two variables in p to a single variable will preserve the containment. The full list of all class-preserving rewrites for $p \subseteq q$ and $p \not\subseteq q$ is available in the Appendix.

2.4. Experimental Results

We trained 5 models using the Adam optimizer (Kingma & Ba, 2014), with binary cross entropy loss. We set the dimensionality of the LSTM output space to $w = 256$, and learning rate was set to 0.00105 by tuning on a separate validation set. Adam's hyperparameter β_1 was set to 0.9 and β_2 was set to 0.999. We train each model for 150 steps: in each step we generate 100K query pairs and train with mini-batch size of 500. We used a 3.3GHz Intel i9-7900X machine with two Nvidia GeForce GTX 1080 Ti GPUs.

Figure 3 shows average performance during training, measured on the AUG test set: a balanced test set of 500K instances generated by applying the data augmentation procedure to a new seed set (Section 2.3). The average final accuracy after 15 million samples is 94.2% (SD 0.6%).

Generalization While the model appears to perform very well on the unseen test set, we were suspicious. Is it really possible that such a straightforward model results in such high accuracy?

To test generalization, we generated two additional test sets. The first one, denoted ALL-CQC, is the set of all 537,477,120 conjunctive query pairs with 2 conjunctions in p and 2 conjunctions in q, labeled by a deterministic solver. The second dataset, denoted $\mu(10, 8)$, contains 250K queries sampled from μ, where p had 10 conjunctions and q had 8 conjunctions, again labeled by the solver.

Figure 4 shows the average accuracy of the trained model on the two new test sets, as well as the original test set based on mutating pairs of conjunctive queries. The high accuracy obtained on the original test set is not preserved when testing it on the entire space. Additionally, it is worth noting that ALL-CQC is unbalanced: classifying everything as 0 would result in accuracy above 90%. Performance on the balanced $\mu(10, 8)$ dataset is even lower, even though its class balance matches that of the training set.

2.5. Discussion

What went wrong? Clearly the model has learned something: it performs very well on an unseen test set created by our data generator. This suggests the issue is not improper learning schedule or a poor choice of model. Instead, the model did not learn how to solve CQC but rather how to exploit a property of the generation method. Moreover, by generating the test set using the same procedure, we overestimate performance on the full problem space. Had we not tested on ALL-CQC and $\mu(10, 8)$, we might have remained convinced that the model has learned to solve

CQC.

In the next section, we show that the issue indeed lies with data generation, and that it would be difficult to overcome for any modeling approach that requires large training sets. Any polynomial-time data generation method for an *NP*-hard problem results in an easier sub-problem.

3. Inherent Bias in Efficient Samplers

Supervised learning requires obtaining a training set: instances of the problem with known labels. When the training set is biased or the label leaks via superficial features, such as sample length or range of attributes, the resulting model may be of no value.

In this section we show that any polynomial-time data generation method for an *NP*-hard decision problem is not just inherently biased, it is also biased in a way that precludes training a model to solve the original problem.

We study the data generation problem for binary classification tasks under the assmumption that *NP* $\neq$ *coNP*, and show that for an *NP*-hard decision problem, no efficient method can produce every possible labeled instance. Moreover, the dataset generated by any efficient data generation method for *NP*-hard problems will provably generate data from an easier sub-problem of the original classification task. In addition, we show an example of a problem for which any efficient data generation procedure only generates data that is trivially solvable with arbitrarily high probability, whereas the original problem cannot be solved in polynomial time.

3.1. Complete Efficient Samplers Do Not Exist

We first discuss desirable properties for data generation methods for classification tasks, and show that under the assumption that *NP* $\neq$ *coNP*, it is impossible to obtain both efficient and representative generators for *NP*-hard problems.

A *language* L is a set of strings. Every language L induces a binary classification task: the positive class contains all the strings in L, and the negative class contains all the strings in L's complement L^C. For example, $L =$ CQC is the set of all strings w such that $w = (p, q)$ for two conjunctive queries p, q and $p \subseteq q$. The classification task induced by L is to decide, given a string w, whether w is in L.

A *sampler* S_L for the classification problem induced by the language L is a randomized algorithm (an algorithm that can flip coins during its execution) which generates labeled samples from both the negative and the positive classes. Our goal is to obtain a dataset as representative as possible in the following sense: achieving high accuracy on the sampled instances should indicate high accuracy on the entire space of instances.

Hence, a reasonable property of a representative sampler is *completeness*: the ability to generate every instance with a non-zero probability (otherwise a classifier trained using the sampler may have low accuracy on the unsampled parts of the problem space). In addition, since modern machine learning methods such as deep neural networks require large datasets, the sampler is used to generate millions of labeled instances. Thus, a sampler should be *efficient*, which we define as polynomial run time complexity[2].

The first question we address is the existence of such efficient and complete samplers for *NP*-hard problems. Alas, such samplers do not exist: under the plausible assumption that *NP* $\neq$ *coNP*, we will now show that it is impossible to obtain efficient and complete data samplers for *NP*-hard languages. Without loss of generality and for technical convenience, we separate our discussion between a *positive* and a *negative* sampler.

Definition 3.1 (Positive Sampler)**.** *A positive sampler S_L^+ for the language L, is a randomized algorithm which on input n (represented in unary), outputs a string w such that $|w| = n$ and $w \in L$, or outputs $\emptyset$ if no such string exists.*

Definition 3.2 (Negative Sampler)**.** *A negative sampler S_L^- for a language L is a positive sampler for L^C: on input n S_L^- outputs a string w such that $|w| = n$ and $w \notin L$, or outputs $\emptyset$ if no such string exists.*

Optionally, a sampler (negative or positive) can also output the string of random bits drawn by the sampler when generating w. For a language L with both a positive and a negative samplers, we define a *sampler* for L.

Definition 3.3 (Sampler)**.** *A sampler S_L for a language L is a randomized algorithm such that on input n (represented in unary) it samples a word w using either S_L^+ or S_L^- and returns w and the corresponding label 1 or 0.*

Note that this definition matches any data generation algorithm for L, regardless of method, since we do not limit how S_L chooses which sampler to use. It can even run both S_L^+ and S_L^-, and only then choose which word to output.

We denote by $S_L(n)$ the set of strings of length n that can be generated by the sampler S_L. A sampler is *complete* if it can generate every example: for every sufficiently large n, for every $w \in L$ of length n it holds that $w \in S_L(n)$. A sampler S_L is called *efficient* if it runs in polynomial time. For clarity, if $w \in S_L(|w|)$, in other words if w can be generated by S_L, we denote it by $w \in S_L$.

The notion of complete efficient sampler is related to the notion of Nondeterministic Test Instance Construction Method (NTICM), as defined by Sanchis (1990). The NTICM for a language L is a nondeterministic Turing machine M such

[2] In reality, this is hardly sufficient for real use, but as we see even this permissive requirement is too demanding for samplers.

that on input 1^n outputs a string from L, and that for every string in L there is a computational path of M which outputs it. As proven in (Sanchis, 1990), NTICMs for *coNP*-complete languages do not exist unless *NP* = *coNP*.

We now show that the existence of efficient complete sampler implies the existence of NTICM. It follows that efficient complete negative samplers for *NP*-complete languages do not exist, hence efficient complete samplers do not exist.

Theorem 1. *If L is NP-hard, then there is no efficient complete sampler for it, unless NP = coNP.*

Proof. Assume by contradiction that there exists an efficient complete sampler S_L for an *NP*-hard language L. Denote by S_L^- the negative sampler used by S_L. Define the following nondeterministic Turing machine M: M runs S_L^- on 1^n, and each time S_L^- flips a coin, M decides nondeterministicly to which branch of S_L^- to proceed. M is a NTICM for L^C, which is *coNP*-complete since we assumed L is *NP*-complete, in contradiction to Proposition 2.1 in (Sanchis, 1990). □

Theorem 1 shows that it is impossible to obtain an efficient complete sampler for both the negative and the positive classes of an *NP*-hard language L. We note that even the existence of efficient complete *positive* samplers for all languages in *NP* is an open problem: the existence of a language $L \in$ *NP* with no efficient complete positive sampler would imply that *P* $\neq$ *NP* (Sanchis, 1990).

We next show that every *incomplete* efficient data samplers for *NP*-hard languages are biased towards an easier subset.

3.2. Incomplete Efficient Samplers are Biased

We now show that not only an efficient sampler cannot generate the entire space of labeled instances for an *NP*-complete problem, but also the instances it does generate are *easier* to decide than the original problem.

Definition 3.4. *The classification task induced by S_L, denoted by $C(S_L)$, is task of classifying instances generated by S_L: given an instance w generated by S_L (without its label), determine if $w \in L$.*

$C(S_L)$ may be easier than the original decision problem. For example, let L be the MAX-CLIQUE problem: given a graph G and a number k, does G has a clique of size k? Consider a sampler S_L that generates instances of (G, k) with matching labels, but can only generate planar graphs. In this case $C(S_L)$ would be the problem of deciding if G has a clique of size k, where G is a planar graph. While the original classification task of deciding L is *NP*-complete, the task $C(S_L)$ is in *P* (Chiba & Nishizeki, 1985).

It turns out that if L is *NP*-hard, $C(S_L)$ is always easier for any polynomial-time S_L, assuming *NP* $\neq$ *coNP*.

Lemma 2. *If S_L is an efficient sampler for a language L, then the classification task $C(S_L)$ is in NP ∩ coNP.*

Proof. Recall that $L \in$ *NP* if for every word $w \in L$ there exists a string z with length polynomial in $|w|$ (a *certificate*) such that a deterministic Turing machine (*verifier*) that given w and z can verify in polynomial time that $w \in L$. Similarly, $L \in$ *coNP* if there exist polynomial verification for every $w \notin L$. Note it is enough to prove that the certificate and verifier exist, even if we do not know what they are.

Given a string w generated by S_L with label 1, let z be the sequence of random bits used by S_L to generate w. We can now build a deterministic Turing machine M that, given w and z, verifies $w \in L$. At each step, M operates as S_L would; whenever S_L needs to draw a random bit, M will use the next bit from z. Since S_L runs in polynomial time, it must use at most polynomial number of random bits. Once S_L ends, M verifies that its output is w and the label returned by S_L is 1. Thus, for every $w \in S_L$ there exists a polynomial verification z for $w \in L$. Note this verifier only applies to w generated by S_L, not to every $w \in L$.

Similarly, given a negative string w generated by S_L, we can use the verifier M and the sequence of random bits used by S_L to verify in polynomial time that $w \notin L$.

We thus conclude that $C(S_L) \in$ *NP* ∩ *coNP*, completing the proof. □

Lemma 2 bounds the complexity of the classification problem over *any* efficient sampler. In particular, under the assumption that *coNP* $\neq$ *NP*, even if the original problem is *NP*-hard, after sampling the classification task *cannot* be *NP*-hard: there is no polynomial time reduction between solving $C(S_L)$ and solving L.

It immediately follows efficient samplers for *NP*-hard languages sample from a strictly easier sub-problem $C(S_L)$. The proof follows from Lemma 2 when L is *NP*-hard.

Corollary 3. *If L is an NP-hard problem and assuming coNP $\neq$ NP, then for* any *efficient sampler S_L for L the classification task over S_L is* not *NP-hard: $C(S_L) \notin$ NP-hard.*

Corollary 3 shows that the sampled sub-problem is easier. It also implies that even a machine learning model learns to correctly classify instances from $C(S_L)$, that model does not necessarily solve L, meaning that test sets generated by S_L cannot be used to evaluate performance on L.

We next show an extreme example of a language L_0 with severe bias. Any efficient sampler generates trivial examples, yet L_0 is a difficult problem.

3.3. An *NP*-hard Language with Trivially Decidable Instances

Lemma 2 gives an upper bound on the difficulty of $C(S_L)$. But what of the other direction? Given that L is *NP*-hard, how easy can $C(S_L)$ be, and can we meaningfully train a model to classify it? In general, this depends on the language L and the specific efficient sampler S_L.

However, we now give an example of a worst-case scenario: an *NP*-hard language L_0 where if S_{L_0} is an efficient sampler, then $C(S_{L_0})$ can be classified easily and with very high accuracy. More precisely, we will show that any w generated by any efficient sampler S_L can be classified in constant time using a superficial feature.

This is somewhat surprising. No matter how we implement an efficient sampler for L_0, the resulting training set will be useless to us: any such model trained on it simply learn to look at the superficial feature. Though deciding L_0 may not seem immediately practical, we conjecture that many *NP*-hard languages may exhibit similar, though less extreme, properties: samplers that generate superficial features, or $C(S_{L_0})$ that is always be in *P*.

To prove this result, we first prove the following Lemma.

Lemma 4. *There exists an NP-hard language L_1 and a function $\delta(n) \to 0$ as $n \to \infty$, such that for any sufficiently long w generated by any randomized polynomial process,* $\Pr[w \in L_1] \leq \delta(n)$.

A full proof of Lemma 4 is included the Appendix. Here we describe a sketch of the proof.

Let $M_1, M_2, \ldots$ be an enumeration for all Turing machines. We construct a randomized algorithm P that runs in super-polynomial time: given size n, it chooses a Turing machine between $M_1 \ldots M_{g(n)}$ (where $g(n)$ grows slowly towards infinity), runs it, and returns its output w. We then apply a result by Itsykson et al. (2016) to show there is a process P^* that is slower than P, but can with high probability $1 - \epsilon(n)$ generate words that P cannot. Since the output of P includes any polynomial process up to $g(n)$, we show that the probability for P to generate w is below $g(n)\epsilon(n) \to 0$ for $g(n)$ that grows sufficiently slowly.

We now use Lemma 4 to to prove the following Theorem.

Definition 3.5. *Let L be an NP-hard language. The polynomial sampler S_L is* trivial *if there exists m such that for any word w generated by S_L where $|w| \geq m$, with high probability $w \in L$ if and only if the first bit of w is* 1.

Theorem 5. *There exists an NP-hard language L_0 for which every polynomial sampler S_{L_0} for L_0 is trivial.*

Proof. Let L_1 be the language from Lemma 4. Define the language L_0 using the string concatenation operator $\circ$:

$$L_0 = \{1 \circ u \mid u \notin L_1\} \cup \{0 \circ v \mid v \in L_1\} .$$

Let S_{L_0} be a polynomial sampler for L_0, and let w of length n be a word generated by S_{L_0}. Denote by b the first bit of w, and by x the last $n-1$ bits of w. Since S_{L_0} runs in polynomial time, it follows from Lemma 4 that with probability greater than $1 - \delta(n-1)$, $x \notin L_1$.

We now show that S_{L_0} is *trivial*: positive samples generated by the sampler S_{L_0} start with $b = 1$ with high probability, and that negative examples start with $b = 0$ with high probability. As n grows, $\Pr[x \in L_1]$ shrinks. Thus positive examples generated by the sampler will be, with high probability, of the form $\{1 \circ u \mid u \notin L_1\}$. Conversely, negative examples generated by S_{L_0} are from the form $\{1 \circ u \mid u \in L_1\} \cup \{0 \circ v \mid v \notin L_1\}$. As n grows the probability that w is of the form $\{1 \circ u \mid u \in L_1\}$ shrinks, thus with high probability $w \in \{0 \circ v \mid v \notin L_1\}$.

The *NP*-hardness of L_0 follows from the *NP*-hardness of L_1. A reduction R from L_1 to L_0 simply concatenates 0 to a word w, $R(w) = 0 \circ w$. Then $w \in L_1$ if and only if $R(w) \in L_0$, which implies that L_0 is *NP*-hard. □

It follows from Theorem 5 that for every polynomial sampler S_{L_0} for L_0, there is a simple algorithm that obtains arbitrarily high accuracy on the instances generated by S_{L_0}: return the first bit of the input.

4. Related Work

Using neural networks to solve computationally hard problems has been studied for many years, with early works attempting approximation of combinatorial optimization problems (Peterson & Anderson, 1988; Budinich, 1997; Smith, 1999). Recent efforts on using deep neural networks to solve *NP*-complete problems include 3-SAT (Selsam et al., 2018), graph problems (Khalil et al., 2017; Prates et al., 2019), symbolic mathematics (Lample & Charton, 2020), and learning to solve routing problems (Kool et al., 2018). An alternative research direction is hybrid architectures that incorporate deterministic solvers. Solving *NP*-complete problems with a differential solver layer was studied in (Wang et al., 2019; Ferber et al., 2019). Selsam & Bjørner (2019) propose integrating deep learning models with deterministic solver in order to improve heuristics used by deterministic SAT solvers.

Data generation in these works was done either by deterministic solvers, which are impractical for large training sets, or by data augmentation heuristics. For example, in recent work Lample & Charton (2020) generate instances of symbolic integration problems by applying transformation to random samples, and by discovering new samples

from existing ones (data augmentation). As noted by Davis (2019), these specific techniques are biased: the generated instances are not diverse and do not represent the difficulty of the problem. They might also leak information via the relative size of function pairs.

Though we focus on classic computational tasks, when real life decisions made by machine learning models trained on biased datasets, such models can perpetuate the bias in future decisions (Yapo & Weiss, 2018). The deleterious effects of dataset bias have been further documented when machine learning is used for healthcare (Oakden-Rayner et al., 2019), recidivism prediction (Dressel & Farid, 2018), predicting criminal behaviour (Yapo & Weiss, 2018) and job performance (Cawley & Talbot, 2010). A survey on bias in machine learning can be found in (Mehrabi et al., 2019).

The study of generating solved instance for computationally hard problems was initiated by Sanchis (1990), who studied the ability to generate optimization problems which are difficult for deterministic solvers. This line of research focuses on generating particularly difficult (slow to compute) instances for deterministic solvers (Selman et al., 1996; Horie & Watanabe, 1997; Cook & Mitchell, 1997; Xu et al., 2005; Haanpää et al., 2006), for example to benchmark solvers (Escamocher et al., 2019). In contrast, when generating data to train machine learning models we generally prefer unbiased, representative samples.

5. Discussion

Recent years have seen many attempts to use machine learning (ML), and in particular, deep neural networks (DNNs), to solve intractable computational tasks, if only approximately. In parallel, there has been much discussion on what DNNs can do, and what they can learn. We show that it is not enough to worry about the representation power of the network and the properties of the loss surface, but also the procedure used to generate the data we need to train it.

We prove that, under the common assumption that $NP \neq coNP$, any efficient sampling technique for an *NP*-hard problem is hopelessly biased: the probability of sampling from certain parts of the problem space is zero – no efficient sampler is complete. Worse, the resulting sub-problem that we do sample from is strictly easier than the original *NP*-hard problem. Thus, common approaches to increasing training set size such as data augmentation result in a training set that does not reflect the full problem. Any ML model trained on such datasets does not learn to solve nor approximate the full NP-hard problem – only the easier sub-problem. Moreover, the sub-problem may in fact be trivial to solve using superficial features of the dataset: we give an example for an NP-hard problem where the data generated by any efficient sampler is trivially easy to classify. Finally, we empirically demonstrate the pitfalls of such approaches when applied to Conjunctive Query Containment, showing how biased data generation leads us to overestimate performance.

We discuss implications and limitations of our results.

Can we teach current DNNs to solve or approximate NP-hard problems? In practice, it is hard to see how. Accurate labeling requires non-polynomial samplers such as deterministic solvers, and the problem space grows exponentially large. Moreover, experience shows that such models generalize poorly when we increase the problem size (Graves et al., 2014; Prates et al., 2019), meaning that even if we succeed, we would need to obtain new training data for larger problems.

Does this mean DNNs cannot learn to approximate NP-hard problems? It does not. We only discuss the difficulty of obtaining training and testing data, and do not say what DNN can or cannot learn. If we somehow obtain an accurately labeled and sufficiently large training set and use the right optimization procedure, we might be able to teach a DNN model to solve such a problem.

Can semi-supervised learning help? Unfortunately, in so far as these methods are efficient samplers, our results apply – meaning they are similarly biased. For example, popular semi-supervised learning approaches such as MixMatch (Berthelot et al., 2019) and ADASYN (He et al., 2008) essentially perform data augmentation: they take samples from the training set and mutate them.

Are the sub-problems trivial to solve? They can be, as shown in Section 3.3, but not necessarily. However, our experience with another NP-hard problem leads us to suspect many NP-hard problems do suffer from this issue to some extent. We intend to explore this question in future work.

Is it impossible to use ML to solve hard problems? Not at all. First, not all hard problems are NP-hard, and even when they are, the application might not require solving the full NP-hard problem. For many applications, solving an easier sub-problem may be sufficient. For example, optimal elastic image matching is NP-complete (Keysers & Unger, 2003), yet ML techniques excel at computer vision tasks. Second, while many ML approaches require a dense sampling from the modeled distribution, this does not necessarily apply to all approaches. A model that can learn from a very sparse sampling of the problem space could, presumably, be trained using a smaller training set generated by a deterministic solver. However, we conjecture that such models must incorporate a non-polynomial deterministic solver of some sort. Rather than learning the problem directly, they could learn a polynomial reduction from the original problem to one the solver layer can solve. In particular, we believe hybrid architectures such as the differentiable SAT solver (Wang et al., 2019) are a promising direction.

Where to go from here? As mentioned, we believe hybrid models that incorporate deterministic solvers or provable approximations might be the way forward. However, improvements should not be limited to model architectures, but look to sampling methods as well. We intend to study the connection between the hardness of sampling and hardness of solving, and to quantify the hardness of the resulting sub-problem.

References

Berthelot, D., Carlini, N., Goodfellow, I., Papernot, N., Oliver, A., and Raffel, C. A. Mixmatch: A holistic approach to semi-supervised learning. In *Advances in Neural Information Processing Systems 32*, pp. 5050–5060. Curran Associates, Inc., 2019.

Budinich, M. Neural networks for NP-complete problems. *Nonlinear Analysis: Theory, Methods & Applications*, 30 (3):1617–1624, 1997.

Cawley, G. C. and Talbot, N. L. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11 (Jul):2079–2107, 2010.

Chandra, A. K. and Merlin, P. M. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 77–90. ACM, 1977.

Chiba, N. and Nishizeki, T. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

Chirkova, R. *Query Containment*, pp. 2981–2985. Springer New York, New York, NY, 2018.

Chollet, F. *Deep Learning with Python*. Manning Publications Co., USA, 1st edition, 2017. ISBN 1617294438.

Cook, S. A. and Mitchell, D. G. Finding hard instances of the satisfiability problem: A survey. 1997.

Davis, E. The use of deep learning for symbolic integration: A review of (Lample and Charton, 2019). *arXiv preprint arXiv:1912.05752*, 2019.

Draper, D., Halevy, A. Y., and Weld, D. S. The nimble XML data integration system. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pp. 155–160. IEEE, 2001.

Dressel, J. and Farid, H. The accuracy, fairness, and limits of predicting recidivism. *Science Advances*, 4(1), 2018.

Escamocher, G., O'Sullivan, B., and Prestwich, S. D. Generating difficult SAT instances by preventing triangles. *CoRR*, abs/1903.03592, 2019.

Ferber, A., Wilder, B., Dilina, B., and Tambe, M. MIPaaL: Mixed integer program as a layer. *arXiv preprint arXiv:1907.05912*, 2019.

Florescu, D., Levy, A., and Suciu, D. Verifying integrity constraints on web sites. In *In IJCAI*, 1999.

Gent, I. P. and Walsh, T. The SAT phase transition. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pp. 105–109, 1994.

Graves, A., Wayne, G., and Danihelka, I. Neural Turing machines. *CoRR*, abs/1410.5401, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Haanpää, H., Järvisalo, M., Kaski, P., and Niemelä, I. Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):27–46, 2006.

He, H., Bai, Y., Garcia, E. A., and Li, S. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pp. 1322–1328, June 2008.

Horie, S. and Watanabe, O. Hard instance generation for SAT. In *International Symposium on Algorithms and Computation*, pp. 22–31. Springer, 1997.

Imieliński, T. and Lipski, W. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pp. 342–360. Elsevier, 1988.

Itsykson, D., Knop, A., and Sokolov, D. Complexity of Distributions and Average-Case Hardness. In Hong, S.-H. (ed.), *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 38:1–38:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Jarke, M. and Koch, J. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

Kaiser, Ł. and Sutskever, I. Neural GPUs learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Keysers, D. and Unger, W. Elastic image matching is NP-complete. *Pattern Recognition Letters*, 24(1):445 – 453, 2003.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pp. 6348–6358, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kool, W., Van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Kovács, L. and Voronkov, A. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pp. 1–35. Springer, 2013.

Lample, G. and Charton, F. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020. To appear.

Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pp. 6231–6239, 2017.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., and Galstyan, A. A survey on bias and fairness in machine learning. *arXiv preprint arXiv:1908.09635*, 2019.

Milan, A., Rezatofighi, S. H., Garg, R., Dick, A., and Reid, I. Data-driven approximations to NP-hard problems. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Oakden-Rayner, L., Dunnmon, J., Carneiro, G., and Ré, C. Hidden stratification causes clinically meaningful failures in machine learning for medical imaging. *arXiv preprint arXiv:1909.12475*, 2019.

Peterson, C. and Anderson, J. Neural networks and Np-complete optimization problems; a performance study on the graph bisection problem. *Complex Syst.*, 2(1):5989, February 1988.

Prates, M., Avelar, P. H., Lemos, H., Lamb, L. C., and Vardi, M. Y. Learning to solve NP-complete problems: A graph neural network for decision TSP. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4731–4738, 2019.

Prez, J., Marinkovi, J., and Barcel, P. On the Turing completeness of modern neural network architectures, 2019.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Dickstein, J. S. On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2847–2854. JMLR. org, 2017.

Sanchis, L. A. On the complexity of test case generation for NP-hard problems. *Information Processing Letters*, 36 (3):135–140, 1990.

Selman, B., Mitchell, D. G., and Levesque, H. J. Generating hard satisfiability problems. *Artificial intelligence*, 81 (1-2):17–29, 1996.

Selsam, D. and Bjørner, N. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 336–353. Springer, 2019.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

Siegelmann, H. T. and Sontag, E. D. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

Smith, K. A. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

Wang, P.-W., Donti, P., Wilder, B., and Kolter, Z. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6545–6554, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

Weiss, G., Goldberg, Y., and Yahav, E. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 740–745, Melbourne, Australia, July 2018. Association for Computational Linguistics.

Xu, K., Boussemart, F., Hemery, F., and Lecoutre, C. A simple model to generate hard satisfiable instances. *arXiv preprint cs/0509032*, 2005.

Yapo, A. and Weiss, J. Ethical implications of bias in machine learning. 2018.

Zhang, W. Phase transitions and backbones of the asymmetric traveling salesman problem. *Journal of Artificial Intelligence Research*, 21:471–497, 2004.

Appendix

Proof of Lemma 4.

Lemma 4. *There exists an NP-hard language L_1 and a function $\delta(n) \to 0$ as $n \to \infty$, such that for any sufficiently*

long w generated by any randomized polynomial process,

$$\Pr[w \in L_1] \leq \delta(n)\,.$$

The proof is similar to the proof of Theorem 1 in (Itsykson et al., 2016). The main difference is that we construct a decidable language, in contrast to the language generated in (Itsykson et al., 2016).

Proof. For every n, the output of a randomized algorithm P is a random variable P_n: for $w \in \{0,1\}^n$, $\Pr[P_n = w]$ is the probability that given the length n, P outputs w. Let $K \subseteq \{0,1\}^n$ be a set of words of length n; $\Pr[P_n \in K]$ is the probability that a random word w drawn by P_n is in K.

Given two random variables X, Y such that X, Y take values in $\{0,1\}^n$, the *statistical distance* between X and Y is defined as (Itsykson et al., 2016):

$$\Delta(X,Y) = \max_{K \subseteq \{0,1\}^n} |P[X \in K] - P[Y \in K]|\ .$$

Using Theorem 9 in (Itsykson et al., 2016) when $a = \frac{1}{2}$ and $b = 1$ we obtain the following corollary.

Corollary 5. *For every randomized algorithm P that runs in time $O(n^{\log^{0.5} n})$ there exist infinitely many words that P can only generate with probability less than $\epsilon(n)$, where $\epsilon(n) \to 0$ as $n \to \infty$.*

We construct the randomized algorithm P as follows. Let $\mathcal{M}$ be an enumeration of all probabilistic Turing machines $\mathcal{M} = M_1, M_2, M_3, ...$, under a standard enumeration of Turing machines, and let $g(n)$ be a function that satisfies $g(n)\epsilon(n) \to 0$ and $g(n) \to \infty$ (where $\epsilon(n)$ is the function from Corollary 5). Example of such function is $g(n) = \frac{1}{\log(\epsilon(n))}$. We define $\delta(n) = g(n)\epsilon(n)$, by the definition of $g(n)$, $\delta(n) \to 0$.

On input n, the algorithm P uniformly chooses M_i for $1 \leq i \leq g(n)$ and runs M_i on the input n (with the random bits M_i needs) for $O(n^{\log^{0.5} n})$ steps. If M_i returned a word $w < n$, P pads it with $n - |w|$ zeros and returns the result. If M_i returned a word $w > n$, P trims $|w| - n$ characters from w and returns it. Finally, if M_i did not halt, P returns $w = 1^n$.

P satisfies the following properties:

1. For every randomized polynomial algorithm P' and for every $w \in \{0,1\}^n$ when n is large enough,

$$\Pr[P_n = w] \geq \frac{1}{g(n)} \Pr[P'_n = w]\,.$$

2. P runs in time $O(n^{\log^{0.5} n})$.

We show that the first property holds as follows. Let P' be a randomized polynomial algorithm that runs in time $O(n^c)$, and let n_0 be the first index that P' appears in the enumeration $\mathcal{M}$. For w, $|w| = n \geq g(n_0)$ and $n^{\log^{0.5} n} \geq n^c$, the probability of P to generate w is at least the probability to choose the machine P', $\frac{1}{g(n)}$, multiplied by the probability that the machine P' generates w: $\Pr[P'_n = w]$. Note we give P' enough time to complete the computation by choosing n such that $n^{\log^{0.5} n} \geq n^c$.

The second property holds by the definition of P.

By Corollary 5 there exists a randomized algorithm P^* such that for infinitely many n's $n_1, n_2, n_3, ...$, it holds that $\Delta(P^*_n, P_n) \geq 1 - \epsilon(n)$. It means that for each such n, there exists a set of strings K_n such that $\Pr[P_n \in K_n] \leq \epsilon(n)$.

Define L_1 as the union of all K_n.

Let $w \in L_1$ of length n for sufficiently large n, and let P' be a randomized polynomial algorithm.

$$\Pr[w = P'_n] \leq g(n) \Pr[w = P_n] \quad (1)$$
$$\leq g(n)\epsilon(n) \quad (2)$$
$$= \delta(n) \to 0\,. \quad (3)$$

Where (1) follows from the first property of P, (2) follows from the definition of L, and (3) is the definition of $\delta(n)$.

□

Additional Details on CQC

For reproducibility, we include full details of our case study on Conjunctive Query Containment (CQC).

Encoding Query Tokens Table 1 shows the mapping between query tokens and their representation as one-hot vectors.

Table 1: Token representation. Each token with index j is mapped to a vector with 1 in position j and all other elements are zero. The dictionary size and the length of the vectors is $d = 42$.

Type	Tokens	Index range
Variables	`x0 ... x32`	6–11, 14-40
Relations	`Q R0 R1`	12, 5, 4
Operators	`∧ :`	1, 13
Parentheses	`( )`	2, 3
Constants	`0 1`	41, 42

Sampling Balanced Query Pairs from μ We exploit the the phase transition phenomenon to define a parametric family of query pairs $\mu(m_1, m_2)$ such that sampling (p, q) from $\mu(m_1, m_2)$ with $m_1 \geq m_2$ guarantees the following:

- p has m_1 conjunctions and q has m_2 conjunctions.
- The probability that $p \subset q$ is approximately 0.5.
- The process for generating positive and negative examples is the same.

Intuitively, for a conjunctive query p with a fixed number of conjunctions, the fewer variables is uses, the more "constrained" it is. For example, let $p(x_1) = R_1(x_1, x_2, x_3)$ and $q(x_1) = R_1(x_1, x_1, x_2)$. While every tuple in R_1 will satisfy p, only tuples whose first and second element are the same will satisfy q.

Given a fixed set of relations R, we define the distribution $G(X, m)$ over conjunctive queries with m conjunctions, where X is a set of variables as follows: first, choose m relations from R uniformly and with repetitions; then, conjunction variables for each conjunction uniformly and with repetitions from X. The *constraintness* of $G(X, m)$ is defined as $\alpha = \frac{m}{n}$.

Let $p \sim G(X_1, m_1)$ and $q \sim G(X_2, m_2)$ be a query pair, and let α_1 and α_2 be the respective constraintness. We observe that the probability of $p \subseteq q$ depends on the ratio of α_2 and α_1. When $\frac{\alpha_2}{\alpha_1} \gg c$ for a constant c, with high probability $p \subseteq q$, when $\frac{\alpha_2}{\alpha_1} \ll c$ with high probability $p \not\subseteq q$, and when $\frac{\alpha_2}{\alpha_1} \approx c$, the probability of $p \subseteq q$ is approximately 0.5. We empirically determined that for $m_1 \geq m_2$, $c \approx \frac{2}{15}$.

Finally, we define the distribution $\mu(m_1, m_2)$ over pairs of conjunctive queries (p, q) as sampling $p \sim G(X_1, m_1)$ and $q \sim G(X_2, m_2)$ with X_1 and X_2 such that $\frac{\alpha_2}{\alpha_1} \approx c$. Since positive and negative samples are generated with the same structure and the same constraintness, syntactic features alone are unlikely to help classification.

Data Augmentation for Conjunctive Query Pairs Given a query q, we define the following rewrites:

- `MergeVar(`q`)`: Pick two variables $x, y \in vars(q)$, replace every occurrence of y by x.
- `SplitVar(`q`)`: Pick a new variable $w \notin vars(q)$, and a variable $x \in vars(q)$. Each occurrence of x is unchanged with probability 0.5 or replaced with w.
- `AddConj(`q`)`: Pick a conjunction $R(\ell_1, \ell_2, \ell_3)$ and add it to q.
- `DelConj(`q`)`: Pick a conjunction in p and remove it.
- `Shuffle(`q`)`: Shuffle the order of conjunctions in p.

For (p, q) where $p \subseteq q$, we use the following set of class-preserving rewrites: $(\texttt{MergeVar}(p), q)$, $(p, \texttt{SplitVar}(q))$, $(\texttt{AddConj}(p), q)$, $(p, \texttt{DelConj}(q))$, $(\texttt{Shuffle}(p), q)$, and $(p, \texttt{Shuffle}(q))$. For (p, q) where $p \not\subseteq q$, we use the following class-preserving rewrites: $(p, \texttt{MergeVar}(q))$, $(\texttt{SplitVar}(p), q)$, $(p, \texttt{AddConj}(q))$, $(\texttt{Shuffle}(p), q)$, and $(p, \texttt{Shuffle}(q))$.