CS 224 Fall 2023 Scribes: Lucia Gordon
November 13, 2023

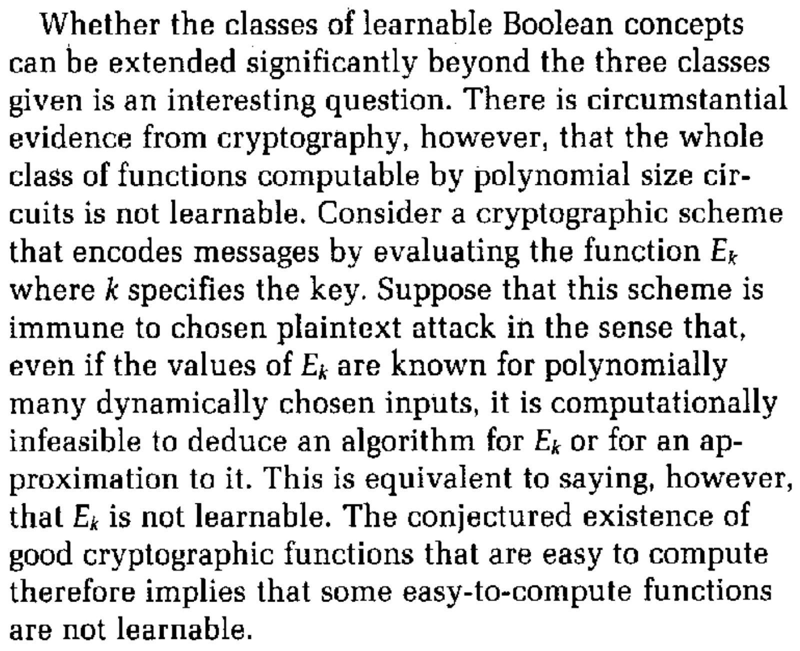
Lecture 19: Cryptographic Reductions and Learning

This is the last unit on the computational complexity of learning. In the last few lectures we talked mainly about statistical query as a recipe for getting lots of different kinds of lower bounds. In general, proving reductions or natural learning problems is hard. The traditional way of proving hardness results is not as successful as using restricted models of computation and proving query lower bounds. In this lecture we look at the reduction side, primarily from the vantage point of connections to cryptography.

1 Early Connections to Cryptography

1.1 Cryptography and Learning

In 1984, L. G. Valiant defined PAC learning [Val84].


Whether the classes of learnable Boolean concepts can be extended significantly beyond the three classes given is an interesting question. There is circumstantial evidence from cryptography, however, that the whole class of functions computable by polynomial size circuits is not learnable. Consider a cryptographic scheme that encodes messages by evaluating the function E_k where k specifies the key. Suppose that this scheme is immune to chosen plaintext attack in the sense that, even if the values of E_k are known for polynomially many dynamically chosen inputs, it is computationally infeasible to deduce an algorithm for E_k or for an approximation to it. This is equivalent to saying, however, that E_k is not learnable. The conjectured existence of good cryptographic functions that are easy to compute therefore implies that some easy-to-compute functions are not learnable.

Suppose we have a black box that is invulnerable to learning, even if we can arbitrarily query the box. If the box is implemented by a polynomial-size circuit, then that implies general polynomial-size circuits are hard to learn. Thus, there are models for which the inference task is easy (e.g., evaluating a neural network), but given examples labeled according to this family of circuits, it would be hard to recover the description of the circuit. Computation of a function and learnability of a function are strictly different things.

1.2 Pseudorandom Function Families (PRF)

Roughly, a pseudorandom function family (PRF) is a family $\mathcal{F}$ of 2^n functions $f\colon \{0,1\}^n \to \{0,1\}^n$ that take n bits to n bits and fool any polynomial-time adversary that tries to distinguish between the following two scenarios:

1. Random: f is sampled uniformly from the set of all Boolean functions
2. Pseudorandom: f is sampled from the PRF $\mathcal{F}$

A pseudorandom function behaves almost as if it were random. The adversary can query f polynomially many times at arbitrary inputs and must distinguish with nontrivial advantage which of the two scenarios we are in. Note that the set of all functions that take n bits to n bits is much larger than the pseudorandom function family since the former is of size 2^{2^n} whereas the latter is of size 2^n, so the number of bits of randomness in the two scenarios are exponentially separate.

This is an even stronger model of learning than what we have considered so far. Previously, we considered the setting where you get a bunch of labeled examples (maybe drawn from a distribution), but here, we have a stronger notion of security. The function is invulnerable even to adversaries that can arbitrarily query the function, not just get random examples with labels according to some function. Thus, if we have a function family that can implement pseudorandom functions, the functions are hard to learn even in the stronger model where you can query the function at whatever input you want.

1.3 PRFs from One-Way Functions

It turns out pseudorandom function families actually exist under very mild assumptions in complexity theory [GGM84]. A one-way function (OWF) is a function $f\colon \{0,1\}^n \to \{0,1\}^m$ that is efficiently computable but hard to invert. This means that given $y = f(x)$ for a random x, it is computationally hard to find x' such that $f(x') = y$. This is the "minimal" assumption necessary for cryptography to be possible.

Theorem 1. *[GGM84] Pseudorandom function families exist if one-way functions exist.*

[GGM84] says you can implement PRFs with a polynomial-size circuit, and these are hard to distinguish from purely random.

Corollary 1. *[Val84] Polynomial-size circuits are hard to PAC learn, even given the ability to query them at arbitrary inputs (not just in the random example setting but learning from membership queries).*

Proof. A polynomial-time PAC learning algorithm would yield a polynomial-time adversary who could determine whether the underlying function is truly random or from a PRF. □

This is some initial evidence that cryptographic primitives provide some hardness of learning results. The pros are that even though the assumption is very mild we can get a hardness result from it, and the learner is very powerful, meaning that even if the learner could query inputs arbitrarily, they still cannot learn. This makes this a great result in the lower bound sense. However, it is not a great result in the sense that it applies to worst-case circuits, but the circuits (pseudorandom functions) labeling data in the real world may not be of this form. Moreover, from the standpoint of proving lower bounds, it is easier to prove hardness results if a function family is very expressive. Here, we would need to prove lower bounds for the set of all polynomial-time-size circuits, which is a very rich family of functions. This is the earliest possible cryptographic lower bound.

1.4 Public-Key Encryption

This makes a stronger assumption than merely the existence of one-way functions, but you get hardness results for a weaker family of functions, making it a better result for proving a lower bound. A public-key cryptosystem consists of three algorithms: gen (generator), enc (encryptor), and dec (decryptor).

- gen(1^n) outputs a public key Pk (given to everyone) and a secret key Sk (only given to the decryptor) given an input of size n
- enc(Pk, m) (randomly) encrypts message $m \in \{0, 1\}$ into poly(n) bits (ciphertext) under Pk
- dec(Sk, c) decrypts a ciphertext c under Pk given Sk to recover m

This method satisfies correctness since dec(Sk, enc(Pk, m)) = m with all but negligible failure probability. It also guarantees security since it is impossible to distinguish between the encryptions of 0 and 1. This is because for all poly-time algorithms A

$$\left|\mathbb{P}_{\text{Pk,Sk}}\left[A(\text{Pk}, \text{enc(Pk)}) = 1\right] - \mathbb{P}_{\text{Pk,Sk}}\left[A(\text{Pk}, \text{enc(Pk)}) = 0\right]\right| \ll \frac{1}{\text{poly}(n)}$$

1.4.1 Trapdoor function

There are various ways of constructing these kinds of cryptosystems, one of which is based on trapdoor functions. A trapdoor function is a one-way function for which there exists an efficient algorithm I that takes as input $f(x)$ along with a trapdoor string t and outputs x' satisfying $f(x) = f(x')$.

Consider the public-key cryptosystem RSA defined by $x \mapsto x^e \mod n$ for $n = pq$ and e relatively prime to $\phi(n) = (p-1)(q-1)$. The trapdoor is $d = e^{-1} \mod \phi(n)$. In the trapdoor setting, the three algorithms work as follows. gen(1^n) has Pk be a trapdoor function f and Sk be a trapdoor t. enc(Pk, m) encrypts a message m by mapping through f. dec(Sk, c) decrypts a ciphertext c by inverting using t. Note this doesn't quite work because we need to define "hard-core predicates."

1.5 Public-Key Encryption + Hardness of Learning

It turns out there is a basic connection between public-key cryptosystems and learning lower bounds.

Lemma 1. *[KV94] Suppose there is a public-key cryptosystem (gen, enc, dec) that you conjecture is secure. Suppose the function class $\mathcal{C}$ contains all the decryption functions dec(Sk,$\cdot$) for all possible choices of secret key. Then $\mathcal{C}$ is hard to learn.*

Proof. We want to break this cryptosystem using the learning algorithm. We train on examples generated as follows.

1. Pick either $m = 0$ or $m = 1$, each with probability 1/2.
2. Then generate a labeled example (enc(Pk, m), m) (example is the bit's encryption (ciphertext) and the label is the decryption (plaintext)).

It is very cheap to generate these once you have the public key, which we assume everyone has access to. Consider running a PAC-learning algorithm on this dataset. We know there is some decryption function that maps enc(Pk, m) to m. If that decryption function were within the function class $\mathcal{C}$, then running the PAC-learning algorithm would learn a good approximation to the decryption function. However, this violates the initial security assumption. □

Note that this hardness result only applies to an adversarial choice of input distribution, so it is only a hardness result for distribution-free PAC learning.

There are several hardness results that were proven using this observation.

- [KV94]: the original paper that showed the reduction, which extended the PRF function family results by showing hardness even for poly-size Boolean formulas, deterministic finite automata of poly-size, and poly-size threshold circuits with constant depth (families of functions less expressive than the family of all possible Boolean circuits)
- [KS06]: on intersections of poly-many halfspaces and also various stylized neural networks over Boolean inputs (e.g., poly-size depth-2 circuits with majority gates, depth-3 poly-size arithmetic circuits)
- [ABW10]: under a certain public-key cryptosystem that they construct out of various average-case assumptions like planted clique, functions that are juntas (only depend on a small number of variables $\log(n)$ in the input) are hard because they can implement the decryption

These results are better than the PRF results in the sense that they apply to more structured families of functions. Even if the function family is less expressive than the family of all Boolean circuits, they are able to demonstrate hardness. The main drawback is that the input distribution over which they show hardness is very unnatural. In the reduction above, the distribution was essentially just ciphertext, but you are probably not going to encounter a dataset whose x values are encryptions of messages according to some public-key cryptosystem. In general, you would expect the distribution to look more like the uniform distribution or something with much more structure.

1.6 Distribution-Specific Hardness: Noise Helps

There is a cryptographic hardness result that works when the input distribution is nice, but the drawback is you need to assume the labels in the data are highly noisy. For learning problems where there is label noise (e.g., agnostic learning), it turns out there are classical ways of proving cryptographic lower bounds, so it is relatively easy to prove hardness even when the input distribution is a known "nice" distribution like Unif($\{\pm 1\}^n$).

For an example, let us make the learning parity with noise (LPN) cryptographic assumption, which we have tried to justify in previous lectures using the perspective of statistical query. Let $\eta > 0$ and $S \subseteq [d]$ be random. We are given a dataset $(x_1, y_1), ..., (x_N, y_N)$ according to $x \sim \{\pm 1\}^d$ and

$$y = \begin{cases} x_S & \text{w.p. } 1-\eta \\ -x_S & \text{otherwise} \end{cases}$$

There are various cryptosystems based on either LPN or a generalization of it called "learning with errors" (LWE) due to larger finite field sizes. These kinds of assumptions form the basis for a variety of cryptosystems that are meant to be secure even against quantum computing. It turns out LPN is useful for trying to show hardness for other problems. This is an instance where there is already noise in the labels, so if we want to show hardness for some other problem with noise in the labels, this is a natural starting point. Moreover, LPN is assumed to be hard even when the inputs are chosen uniformly at random from the hypercube. So it is precisely a setting where the input distribution is nice but there is a lot of noise in the labels.

Lemma 2. *[KKMS08] Learning a halfspace agnostically means we do not assume the data is exactly labeled by a halfspace, but we want to perform as well as possible compared to the best linear classifier. Completing this task, even when the inputs are uniformly random, is hard because if there were an algorithm for the task, it would imply an algorithm for LPN. In other words, the existence of a polynomial-time algorithm for agnostically learning halfspaces over the uniform distribution implies the existence of a polynomial-time algorithm for learning parity with noise.*

Proof. Let us take a particular kind of halfspace: the majority function over a collection of bits, where the majority function is indexed by some subset of the bits and outputs 1 if the majority of the bits in that subset are 1 and otherwise -1. It essentially takes a majority vote over some subset of the input. The majority function correlates with the parity function nontrivially, so if one could approximately learn the former, one could get enough signal to distinguish between a dataset generated by parity with noise and a dataset generated from purely random labels. You could then use this distinguishing to boost your estimate and ultimately get the underlying parity function. The majority function is well approximated by functions we assume are hard to learn in the presence of noise. Thus, if we impose noise on the majority function, it too will be hard to learn. □

A lot of results for agnostic learning are proved in a similar manner. You start with a structured problem that is hard to learn in the presence of noise, and then you try to map your functions of interest back to the structured problem.

1.7 Takeaways

This is all the classical knowledge about hardness of learning. Typically in the past, cryptographic assumptions have been great for proving lower bounds for learning worst-case functions (e.g., implement a decryption function in a public-key cryptosystem or a PRF) either over worst-case input distributions (unnatural) or over benign input distributions but with label noise. Thus, we end up with hard-to-learn problems where either the function is unnatural or the data distribution is unnatural or both. We have talked a lot about highly continuous problems (e.g., learning neural networks over Gaussian inputs), but all of these cryptographic assumptions typically operate over the space of Boolean strings. Thus, these are all specifically discrete settings.

We now transition to working in Gaussian space where our family of functions is (worst-case) neural networks. In particular, we establish a cryptographic lower bound for learning neural networks over Gaussian inputs. This lower bound gets rid of all the caveats above: instead of working with worst-case functions, we are working with smooth neural networks; instead of a worst-case input distribution that is discrete, we have Gaussian inputs; and there is no label noise. We consider a dataset of Gaussian inputs labeled perfectly by some neural network. Even neural networks in some benign setup like smooth analysis where the network is not a worst-case instance still turns out to be hard. We also touch on a cryptographic lower bound for learning mixtures of Gaussians.

2 Crypto Lower Bound for Learning MLPs over Gaussians

2.1 Crypto Hardness for MLPs

Theorem 2. *[DV21] Assuming security of Goldreich's pseudorandom generator, MLPs of depth 3 are hard to learn under Gaussian inputs.*

This is referring to cryptographic hardness rather than statistical query hardness. In particular, we mean cryptographic hardness for learning a real-valued, continuous function over a "nice distribution" without label noise. Unlike the CSQ lower bound we saw previously, this hardness result pertains to arbitrary polynomial-time computation. However, the CSQ lower bound we showed was for depth-2 networks, but here we have to work with more complex, depth-3 networks.

2.2 Goldreich's Pseudorandom Generator (PRG)

A pseudorandom generator is a function that takes a very small amount of randomness and outputs what appears to be a large amount of randomness, which seems counterintuitive from an information-theoretic perspective. However, the output distribution is not close to random in a statistical sense, but rather be indistinguishable from random if you try to use polynomial-time computation. It is a function that maps a small number of bits n to a large number of bits m.

$$F : \{\pm 1\}^n \to \{\pm 1\}^m$$

is a PRG if no polynomial-time algorithm can distinguish between (a) Unif($\{\pm 1\}^m$) (purely random bits) and (b) F(Unif($\{\pm 1\}^m$)) (F applied to a small number of purely random bits) given a polynomial number of samples from both.

We will not assume that PRGs exist but rather that a particular candidate function is a PRG. Define a predicate function that maps a small number of bits to a single bit.

$$P : \{0,1\}^k \to \{0,1\}$$

One example is the XOR-MAJ predicate parameterized by a and b where $k = a + b$.

$$P(z) = \text{XOR-MAJ}_{a,b}(z) = (z_1 \oplus ... \oplus z_a) \oplus \text{Maj}(z_{a+1}, ..., z_{a+b})$$

The majority function outputs 1 or 0 depending on which type of bit has the majority. Goldreich's PRG is constructed as follows. Consider sampling a collection of random subsets $S_1, ..., S_m \subseteq [n]$ of size k. Let us now construct a bipartite graph as follows. Every node corresponding to a subset gets connected to the k bits that are indexed by that subset, where the connections are random. This is visualized in Figure 1. The generator will take as input $z \in \{\pm 1\}^n$ and map it to the m-tuple

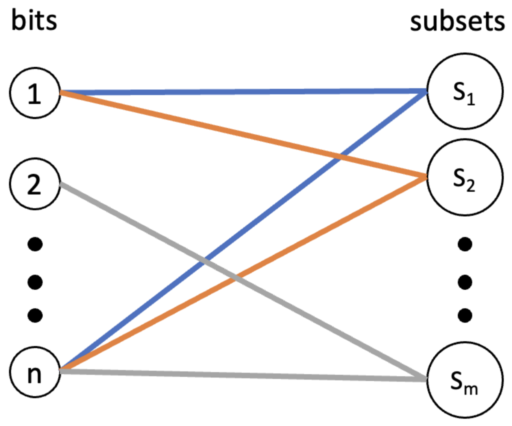


Figure 1: Random bipartite graph with $k = 2$

whose ith entry is given by evaluating the predicate on all the bits connected to the ith node S_i.

$$F(z) = (P(z|_{S_1}), ..., P(z|_{S_m}))$$

where $z|_S$ represents the bits of z indexed by S. We already know constructions of PRGs based on one-way functions, but in practice we want pseudorandom functions that are very simple to compute. For the PRG we just constructed, every output of the function only depends on a constant number of bits, so this is practical to implement. An influential paper [AIK04] delves into constructing cryptographic primitives that are extremely efficient to implement, and Goldreich's PRG is one such example originally proposed by Goldreich. People conjecture it is secure despite its simplicity. The family of hyper-efficient PRGs is called the local PRGs.

We now assume this function F is a PRG. This means for every constant $s > 1$, $\exists$ constant k (number of bits read in by each predicate) and predicate $P : \{0,1\}^k \to \{0,1\}$ such that Goldreich's PRG is a valid PRG.

2.3 Proving the Hardness of Learning MLPs

We use this Goldreich's PRG assumption to prove the hardness of learning MLPs. We do this in three steps. First, (I) we consider why we get hardness in a discrete setting (Boolean-valued inputs $\{0,1\}^n$). Then, (II) we look at a simple way to "lift" this hardness to the Gaussian setting. The lifting is naive and almost works but fails for a crucial reason: the lifting produces an MLP with infinite weights (not a natural function to learn), making the functions discontinuous. Finally, (III) we want to show that even if the function is continuous and over Gaussians, it is still hard to learn, which we show with the Daniely-Vardi gadget [DV21].

(I) How do we go from a PRG to an ML problem? In practice, we are given the PRG, and the security assumption holds even if we know the structure of the graph. If we had an algorithm for an ML problem, that would break the cryptographic assumption, which is a contradiction. To get an ML problem from the PRG, construct a classical ML dataset where the random subsets $S_1, ..., S_m$ are the examples and the labels are given by the output of the PRG $P(Z|_S)$. We have a string z that is either a sample from the PRG or purely random bits. For every S_i, we know the value of the predicate on that subset, so the dataset is given by pairs $(S_i, P(Z|_{S_i}))|_{i=1}^m$.

Treat each S_i as a Boolean string via the following encoding. Given $S = (i_1, ..., i_k) \subseteq [n]$, define an encoding $x^S \in \{0,1\}^{kn}$. Consider k blocks each of length n. In the jth block, the bit string is all 1s except for the i_jth bit that is 0. This means in the first block the 0 goes in i_1, in the second block the 0 goes in i_2, and so on. This is a way of encoding a subset into a bit string. Now we need to show this is computable by a neural network. We want to reduce the problem of distinguishing pseudorandom inputs from random inputs to the problem of learning neural networks over the hypercube. We need to show $\exists$ MLP $N : \{0,1\}^{kn} \to \{0,1\}$ such that $N(x^S) = P(z|_S)\ \forall S$ of size k, which means it outputs the correct label given an encoding for a subset.

Proof. Consider the function mapping $x^S \mapsto P(z|_S)$, which can be implemented via the following DNF.

$$\bigvee_{b\in\{0,1\}^k \text{ s.t. } P(b)\geq 1} \bigwedge_{j\in[k]} \bigwedge_{l \text{ s.t. } z_l\neq b_j}$$

Then you get $x_{j,l}$ by reading the jth block and the lth index. This is a correct implementation of the mapping, and it is a depth-2 Boolean circuit, which is a legitimate MLP. □

Thus, we have shown that learning MLPs over bit strings as constructed above is hard.

(II) We want to be able to convert from Boolean inputs to Gaussian vectors. Observe a distribution over a single block is a distribution over $\{\pm 1\}^n$, which is well approximated by the product distribution Bern$(\frac{n-1}{n})^{\oplus n}$. We want to take this distribution and Gaussianize it. The bits in each block are not independent but rather highly correlated since there is always exactly one zero. Suppose for simplicity that we have hardness even when learning over Bern$(\frac{n-1}{n})^{\otimes kn}$, where the bits are drawn independently. Suppose we want to convert a single uniformly random bit to a Gaussian: Bern$(1/2) \to \mathcal{N}(0,1)$. One way is to output a half-Gaussian.

$$b \in \{0,1\} \to \begin{cases} -\gamma & b = 0 \\ \gamma & b = 1 \end{cases}$$

where $\gamma \sim \mathcal{N}(0,1)|\gamma \geq 0$, a "half Gaussian," shown in Figure 2. We can now

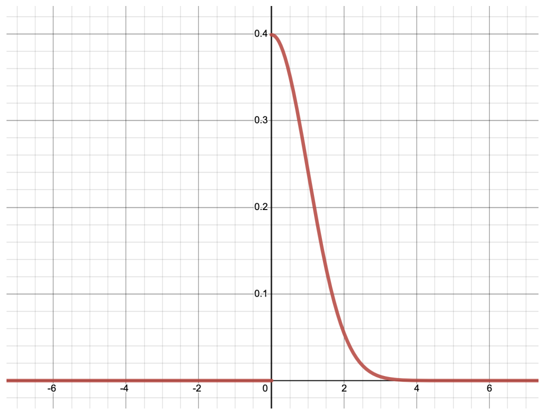


Figure 2: Half Gaussian

consider other kinds of biases instead of 1/2. Suppose we have Bern$(\frac{n-1}{n})$, and we want to output a Gaussian $\mathcal{N}(0,1)$. We can now Gaussianize the bit as follows.

$$b \in \{0,1\} \to \begin{cases} \gamma \sim \mathcal{N}(0,1) | \gamma \geq t & b = 0 \\ \gamma \sim \mathcal{N}(0,1) | \gamma < t & b = 1 \end{cases}$$

This mapping is visualized in Figure 3. This converts the inputs to Gaussian, but

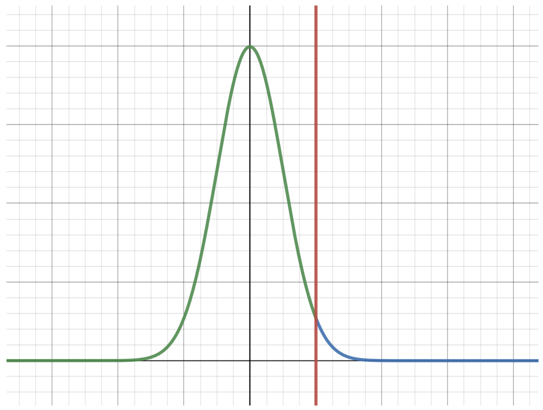


Figure 3: The red line is $x = t$, the blue curve is the $b = 0$ case, and the green curve is the $b = 1$ case.

we also need to ensure the labels are correctly generated. The function we get out of the naive lifting procedure is unfortunately going to be discontinuous. We have gone from $\text{Bern}(\frac{n-1}{n})^{\otimes kn} \to \mathcal{N}(0,1)^{\otimes kn}$, and now we the MLP $N(x)$ and want to get out something that correctly labels the Gaussian dataset. One trivial solution is with $N'(g) = N(\text{threshold}(g))$ for a Gaussian input g and N the function we constructed in the Boolean setting.

$$\text{threshold}(g)_i = \mathbb{1}[g_i \geq t]$$

This function is visualized in Figure 4. We had a process to convert bits to Gaussians, and now we want to take Gaussians and map them back into maps them back to the original distribution on the bits, which is given by the threshold function. This essentially undoes the Gaussianization and is a naive way of lifting to the Gaussian space. However, the threshold function is discontinuous, with a discontinuity at t. Approximating this discontinuity well would require super-polynomially-large weights in the neural network.

(III) Let us try approximating the threshold with a different function that looks like a ramp. We could implement the ramp using a neural network with polynomially-large weights. Consider $N(\text{ramp}(g))$, which is entirely continuous and works for all inputs g except those where a coordinate lands in the "danger

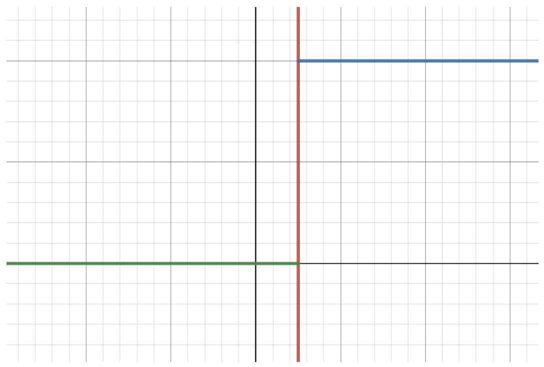


Figure 4: Threshold function with $x = t$ shown in red.

zone." Let us construct a penalty function G_{penalty} that is very large in the danger zone and gradually slopes down to zero on both sides. The penalty function will zero out the label for inputs with a coordinate in the danger zone.

$$H(g) = \text{ReLU}\left(N(\text{ramp}(g)) - \sum_{j \in [k], i \in [n]} G_{\text{penalty}}(g_{ji})\right)$$

The second term (known function $G(g)$) is large if g has a coordinate in the danger zone, making $H(g) = 0$. These functions and zones are displayed in Figure 5. Let us start with the Boolean dataset we want to Gaussianize and draw $(x, m(x))$. We can Gaussianize x to get g. The label is then given by

$$\text{label} = \begin{cases} 0 & g \text{ is in the danger zone} \\ N(x) & g \text{ is in the good zone} \\ \text{ReLU}(N(x) - G(g)) & g \text{ is in the medium zone} \end{cases}$$

We have devised this particular neural network by wrapping a network that does not work with a ReLU and a penalty function. The penalty function is a known neural network, so we can simulate information from it and use that to modify the labels to get out a continuous-valued problem that is hard because the original problem is hard. The main trick was the penalty function that allows us to zero out points in the danger zone, and for the points that are not perfectly Boolean, we can simulate access to those labels because we know the penalty function, completing the reduction.

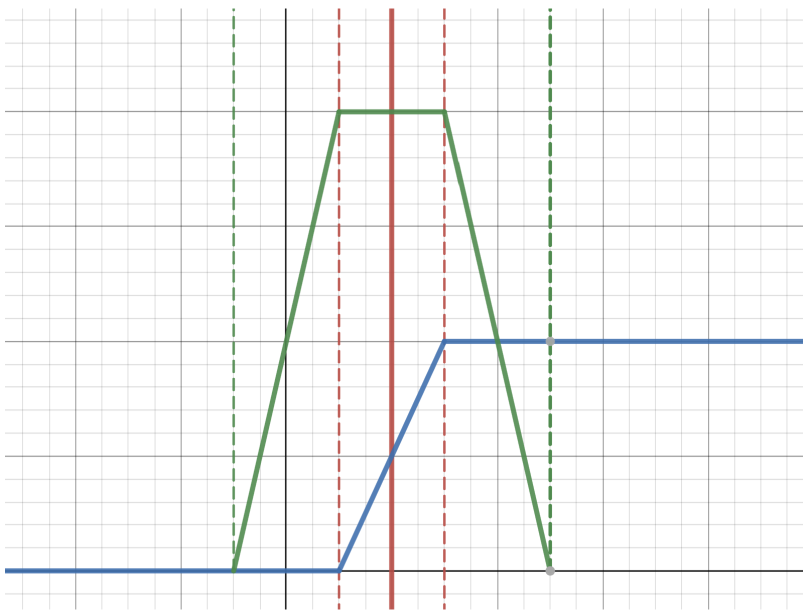


Figure 5: The solid red line is $x = t$. The ramp is shown in blue. The penalty function is shown in solid green. The dotted red lines mark the outer boundaries of the danger zone and the inner boundaries of the medium zone. The dashed green lines mark the outer boundaries of the medium zone and the inner boundaries of the good zone. Points outside the medium and danger zones are in the good zone.

References

[ABW10] Benny Applebaum, Boaz Barak, and Avi Wigderson. Public-key cryptography from different assumptions. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, page 171–180, New York, NY, USA, 2010. Association for Computing Machinery.

[AIK04] B. Applebaum, Y. Ishai, and E. Kushilevitz. Cryptography in nc/sup 0/. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 166–175, 2004.

[DV21] Amit Daniely and Gal Vardi. From local pseudorandom generators to hardness of learning. *CoRR*, abs/2101.08303, 2021.

[GGM84] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. In *25th Annual Symposium on Foundations of Computer Science, 1984.*, pages 464–479, 1984.

[KKMS08] Adam Tauman Kalai, Adam R. Klivans, Yishay Mansour, and Rocco A.

Servedio. Agnostically learning halfspaces. *SIAM Journal on Computing*, 37(6):1777–1805, 2008.

[KS06] Adam R. Klivans and Alexander A. Sherstov. Cryptographic hardness for learning intersections of halfspaces. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 553–562, 2006.

[KV94] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41(1):67–95, jan 1994.

[Val84] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, nov 1984.