
# On the Computational Complexity of Deep Learning

Shai Shalev-Shwartz

School of CS and Engineering,
The Hebrew University of Jerusalem

"Optimization and Statistical Learning",
Les Houches, January 2014

Based on joint work with:
Roi Livni and Ohad Shamir,
Amit Daniely and Nati Linial,
Tong Zhang

# PAC Learning

Goal (informal): Learn an accurate mapping $h : \mathcal{X} \to \mathcal{Y}$ based on examples $((x_1, y_1), \ldots, (x_n, y_n)) \in (\mathcal{X} \times \mathcal{Y})^n$

# PAC Learning

Goal (informal): Learn an accurate mapping $h : \mathcal{X} \to \mathcal{Y}$ based on examples $((x_1, y_1), \ldots, (x_n, y_n)) \in (\mathcal{X} \times \mathcal{Y})^n$

PAC learning: Given $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$, probably approximately solve

$$\min_{h \in \mathcal{H}} \left[ \mathbb{P}_{(x,y) \sim \mathcal{D}} [h(x) \neq y] \right] ,$$

where $\mathcal{D}$ is unknown but the learner can sample $(x, y) \sim \mathcal{D}$

# What should be $\mathcal{H}$ ?

$$\min_{h \in \mathcal{H}} \left[ \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}} [h(x) \neq y] \right]$$

1. Expressiveness

   Larger $\mathcal{H}$ $\Rightarrow$ smaller minimum

# What should be $\mathcal{H}$ ?

$$\min_{h \in \mathcal{H}} \left[ \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}} [h(x) \neq y] \right]$$

1. Expressiveness
   Larger $\mathcal{H}$ $\Rightarrow$ smaller minimum
2. Sample complexity
   How many samples are needed to be $\epsilon$-accurate?

# What should be $\mathcal{H}$ ?

$$\min_{h \in \mathcal{H}} \left[ \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}} [h(x) \neq y] \right]$$

1. Expressiveness
   Larger $\mathcal{H}$ $\Rightarrow$ smaller minimum
2. Sample complexity
   How many samples are needed to be $\epsilon$-accurate?
3. Computational complexity
   How much computational time is needed to be $\epsilon$-accurate ?

# What should be $\mathcal{H}$ ?

$$\min_{h \in \mathcal{H}} \left[ \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}} [h(x) \neq y] \right]$$

1. Expressiveness
   Larger $\mathcal{H}$ $\Rightarrow$ smaller minimum
2. Sample complexity
   How many samples are needed to be $\epsilon$-accurate?
3. Computational complexity
   How much computational time is needed to be $\epsilon$-accurate ?

No Free Lunch: If $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ then the sample complexity is $\Omega(|\mathcal{X}|)$.

# What should be $\mathcal{H}$ ?

$$\min_{h \in \mathcal{H}} \left[ \mathop{\mathbb{P}}_{(x,y)\sim\mathcal{D}} [h(x) \neq y] \right]$$

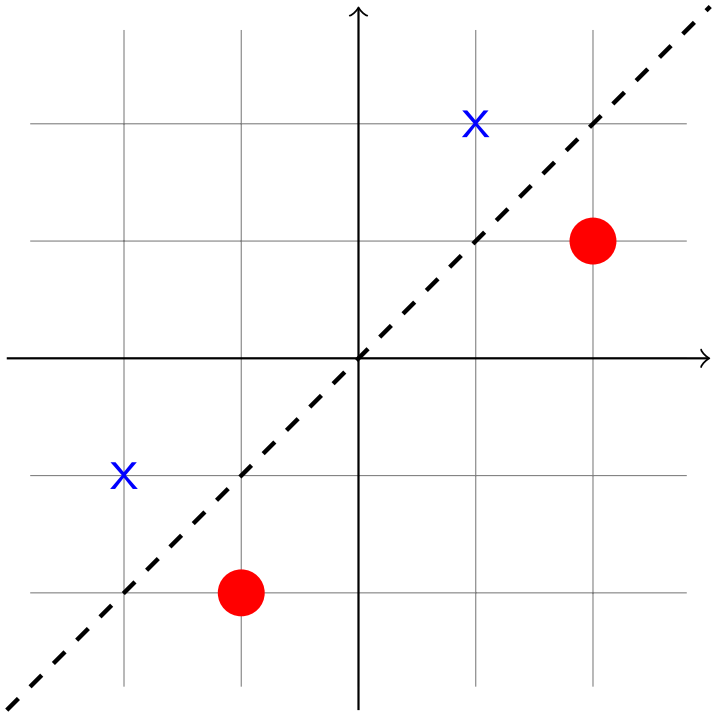
1. Expressiveness
   Larger $\mathcal{H}$ $\Rightarrow$ smaller minimum
2. Sample complexity
   How many samples are needed to be $\epsilon$-accurate?
3. Computational complexity
   How much computational time is needed to be $\epsilon$-accurate ?

No Free Lunch: If $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ then the sample complexity is $\Omega(|\mathcal{X}|)$.

Prior Knowledge:
We must choose smaller $\mathcal{H}$ based on prior knowledge on $\mathcal{D}$

# Prior Knowledge

- SVM and AdaBoost learn a halfspace on top of features, and most of the practical work is on finding good features
- Very strong prior knowledge



# Weaker prior knowledge

- Let $\mathcal{H}_T$ be all functions from $\{0,1\}^p \to \{0,1\}$ that can be implemented by a Turing machine using at most $T$ operations.

# Weaker prior knowledge

- Let $\mathcal{H}_T$ be all functions from $\{0,1\}^p \to \{0,1\}$ that can be implemented by a Turing machine using at most $T$ operations.
- Very expressive class

# Weaker prior knowledge

- Let $\mathcal{H}_T$ be all functions from $\{0,1\}^p \to \{0,1\}$ that can be implemented by a Turing machine using at most $T$ operations.
- Very expressive class
- Sample complexity ?

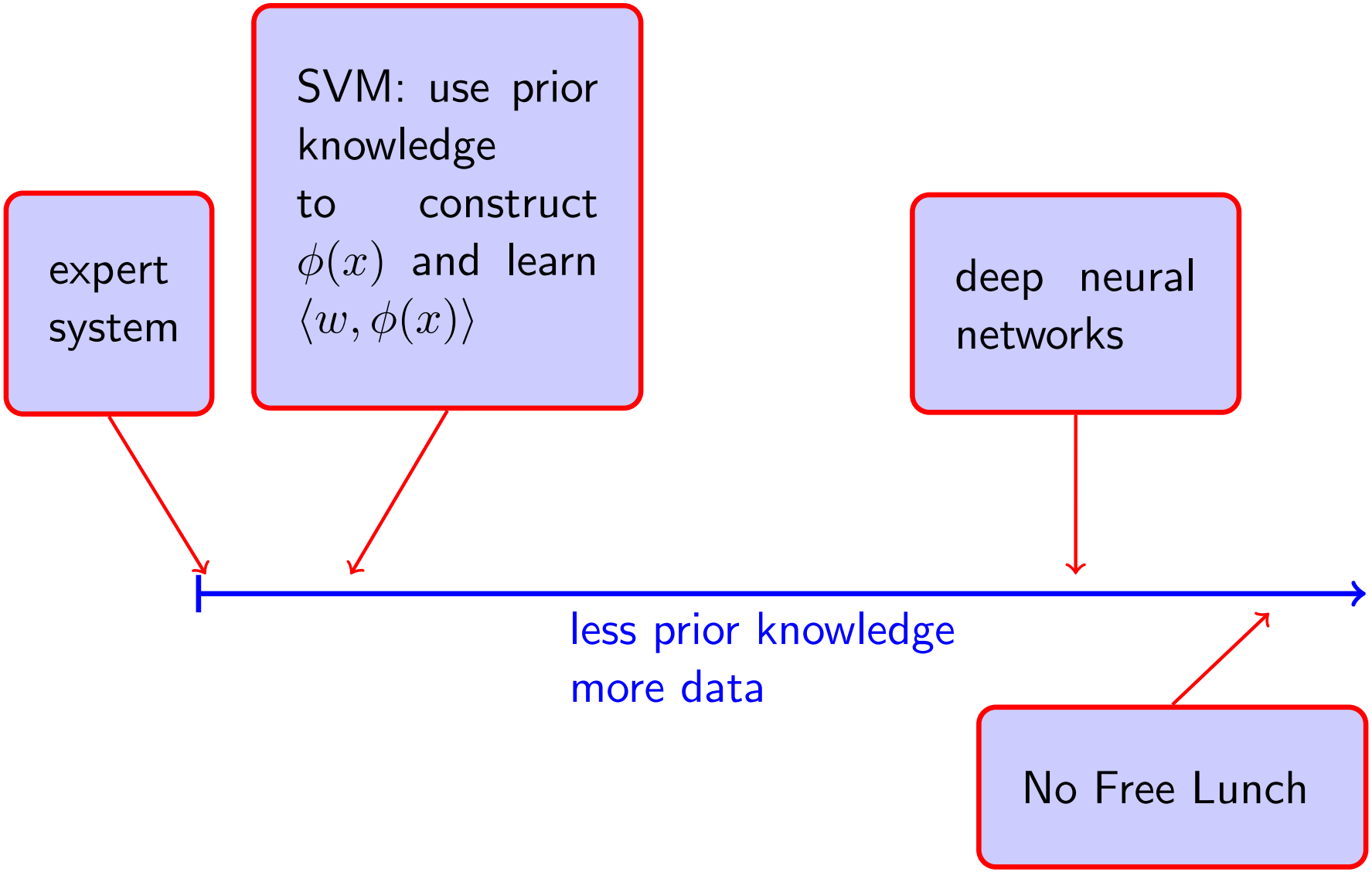
# Weaker prior knowledge

- Let $\mathcal{H}_T$ be all functions from $\{0,1\}^p \to \{0,1\}$ that can be implemented by a Turing machine using at most $T$ operations.
- Very expressive class
- Sample complexity ?

**Theorem**

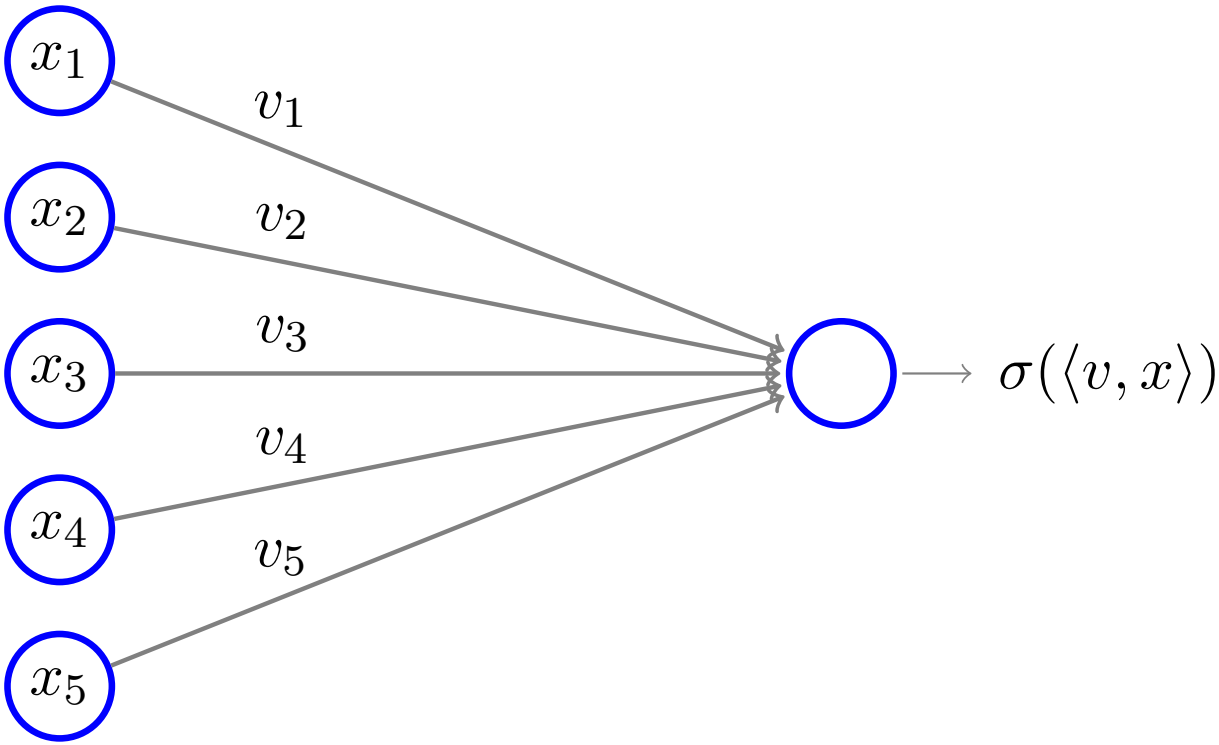
- *$\mathcal{H}_T$ is contained in the class of neural networks of depth $O(T)$ and size $O(T^2)$*
- *The sample complexity of this class is $O(T^2)$*

# The ultimate hypothesis class

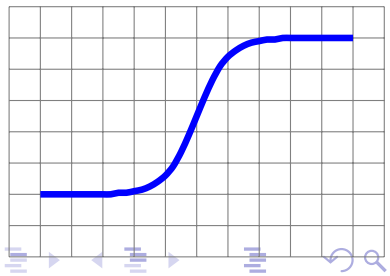


# Neural Networks

- A single neuron with activation function $\sigma : \mathbb{R} \to \mathbb{R}$

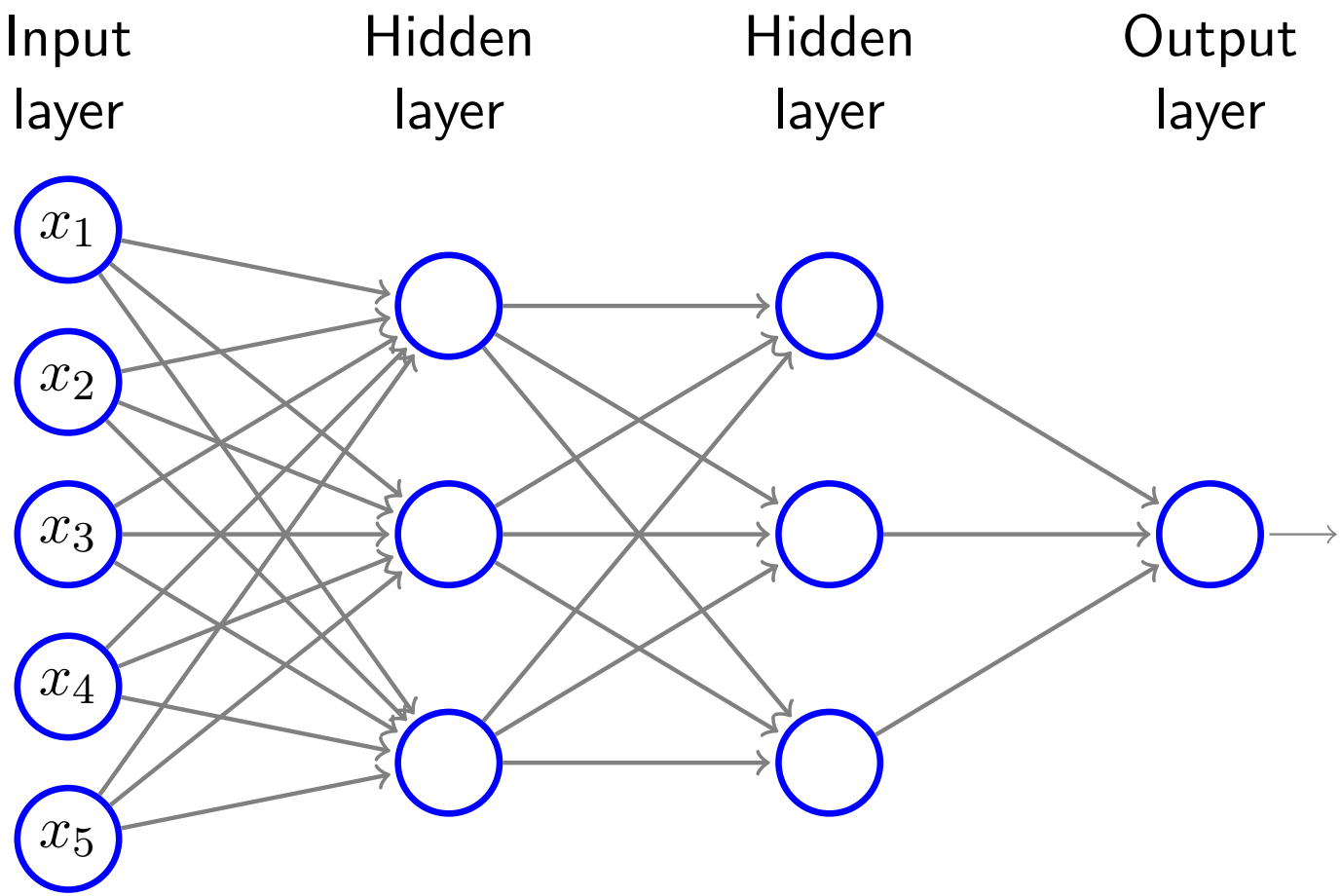



- E.g., $\sigma$ is a sigmoidal function



# Neural Networks

- A multilayer neural network of depth $3$ and size $6$



# Brief history

- Neural networks were popular in the 70's and 80's
- Then, suppressed by SVM and Adaboost on the 90's
- In 2006, several deep architectures with unsupervised pre-training have been proposed
- In 2012, Krizhevsky, Sutskever, and Hinton significantly improved state-of-the-art without unsupervised pre-training
- Since 2012, state-of-the-art in vision, speech, and more

# Computational Complexity of Deep Learning

- By fixing an architecture of a network (underlying graph and activation functions), each network is parameterized by a weight vector $w \in \mathbb{R}^d$, so our goal is to learn the vector $w$

# Computational Complexity of Deep Learning

- By fixing an architecture of a network (underlying graph and activation functions), each network is parameterized by a weight vector $w \in \mathbb{R}^d$, so our goal is to learn the vector $w$
- Empirical Risk Minimization (ERM):
  Sample $S = ((x_1, y_1), \ldots, (x_n, y_n)) \sim \mathcal{D}^n$ and approximately solve

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$$

# Computational Complexity of Deep Learning

- By fixing an architecture of a network (underlying graph and activation functions), each network is parameterized by a weight vector $w \in \mathbb{R}^d$, so our goal is to learn the vector $w$
- Empirical Risk Minimization (ERM):
  Sample $S = ((x_1, y_1), \ldots, (x_n, y_n)) \sim \mathcal{D}^n$ and approximately solve

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$$

- Realizable sample: $\exists w^*$ s.t. $\forall i,\ h_{w^*}(x_i) = y_i$

# Computational Complexity of Deep Learning

- By fixing an architecture of a network (underlying graph and activation functions), each network is parameterized by a weight vector $w \in \mathbb{R}^d$, so our goal is to learn the vector $w$
- Empirical Risk Minimization (ERM):
  Sample $S = ((x_1, y_1), \ldots, (x_n, y_n)) \sim \mathcal{D}^n$ and approximately solve

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$$

- Realizable sample: $\exists w^*$ s.t. $\forall i,\ h_{w^*}(x_i) = y_i$
- Blum and Rivest 1992: Distinguishing between realizable and unrealizable $S$ is NP hard even for depth $2$ networks with $3$ hidden neurons (reduction to $k$ coloring)
  Hence, solving the ERM problem is NP hard even under realizability

# Computational Complexity of Deep Learning

## The argument of Pitt and Valiant (1988)

If it is NP-hard to distinguish realizable from un-realizable samples, then properly learning $\mathcal{H}$ is hard (unless RP=NP)

# Computational Complexity of Deep Learning

## The argument of Pitt and Valiant (1988)

If it is NP-hard to distinguish realizable from un-realizable samples, then properly learning $\mathcal{H}$ is hard (unless RP=NP)

Proof: Run the learning algorithm on the empirical distribution over the sample to get $h \in \mathcal{H}$ with empirical error $< 1/n$:

- If $\forall i, h(x_i) = y_i$, return "realizable"
- Otherwise, return "unrealizable"

# Improper Learning


- Allow the learner to output $h \notin \mathcal{H}$

# Improper Learning

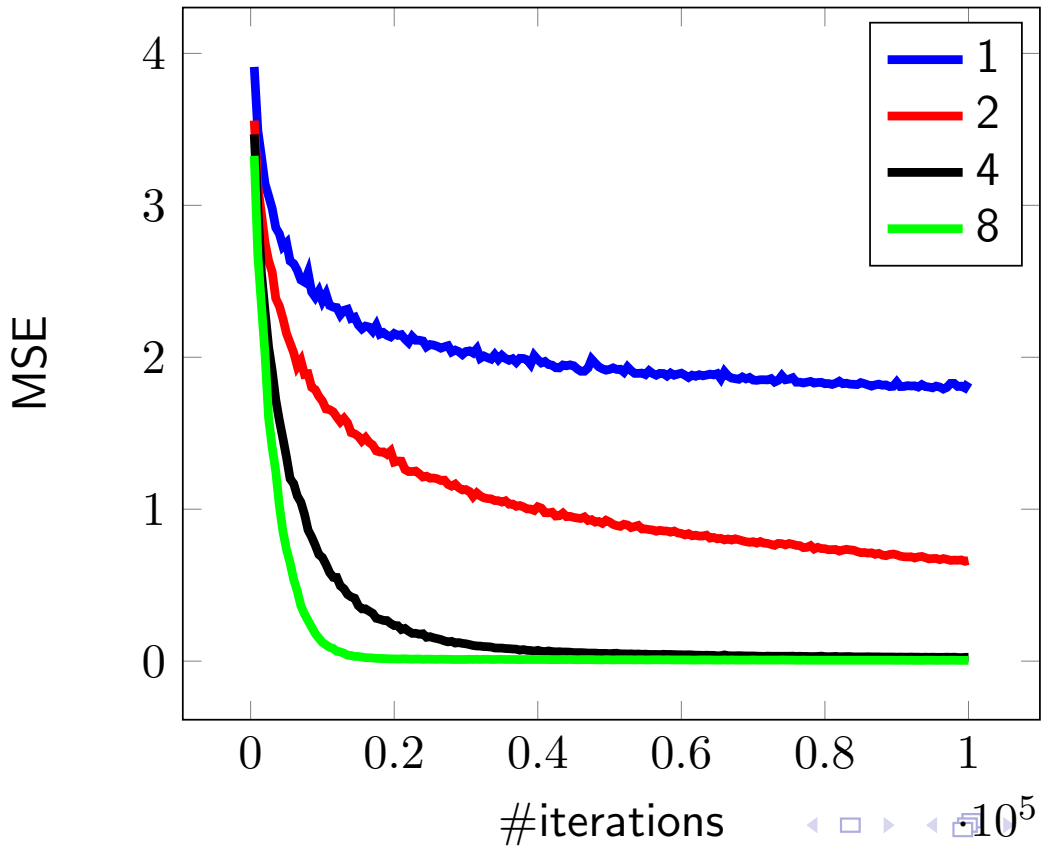

- Allow the learner to output $h \notin \mathcal{H}$
- The argument of Pitt and Valiant fails because the algorithm may return consistent $h$ even though $S$ is unrealizable by $\mathcal{H}$

# Improper Learning


- Allow the learner to output $h \notin \mathcal{H}$
- The argument of Pitt and Valiant fails because the algorithm may return consistent $h$ even though $S$ is unrealizable by $\mathcal{H}$
- Is deep learning still hard in the improper model ?

# Hope ...

- Generated examples in $\mathbb{R}^{150}$ and passed them through a random depth-2 network that contains $60$ hidden neurons with the ReLU activation function.
- Tried to fit a new network to this data with over-specification factors of $1, 2, 4, 8$



# How to show hardness of improper learning?

- The argument of Pitt and Valiant fails for improper learning because improper algorithms might perform well on unrealizable samples

# How to show hardness of improper learning?

- The argument of Pitt and Valiant fails for improper learning because improper algorithms might perform well on unrealizable samples

## Key Observation

- If a learning algorithm is computationally efficient its output must come from a class of "small" VC dimension
- Hence, it cannot perform well on "very random" samples

# How to show hardness of improper learning?

- The argument of Pitt and Valiant fails for improper learning because improper algorithms might perform well on unrealizable samples

## Key Observation

- If a learning algorithm is computationally efficient its output must come from a class of "small" VC dimension
- Hence, it cannot perform well on "very random" samples

Using the above observation we conclude:

Hardness of distinguishing realizable form "random" samples implies hardness of improper learning of $\mathcal{H}$

# Deep Learning is Hard

Using the new technique and under a natural hardness assumption we can show:

- It is hard to improperly learn intersections of $\omega(1)$ halfspaces
- It is hard to improperly learn depth $\geq 2$ networks with $\omega(1)$ neurons, with the threshold or ReLU or sigmoid activation functions

# Theory-Practice Gap

- In theory: it is hard to train even depth $2$ networks
- In practice: Networks of depth $2 - 20$ are trained successfully

# Theory-Practice Gap

- In theory: it is hard to train even depth $2$ networks
- In practice: Networks of depth $2 - 20$ are trained successfully

How to circumvent hardness?

- Change the problem ...

# Theory-Practice Gap

- In theory: it is hard to train even depth $2$ networks
- In practice: Networks of depth $2 - 20$ are trained successfully

How to circumvent hardness?

- Change the problem ...
- Add more assumptions

# Theory-Practice Gap

- In theory: it is hard to train even depth $2$ networks
- In practice: Networks of depth $2 - 20$ are trained successfully

How to circumvent hardness?

- Change the problem ...
- Add more assumptions
- Depart from worst-case analysis

# Change the activation function

- Simpler non-linearity — replace sigmoidal activation function by the square function $\sigma(a) = a^2$
- Network implements polynomials, where the depth correlative to degree
- Is this class still very expressive ?

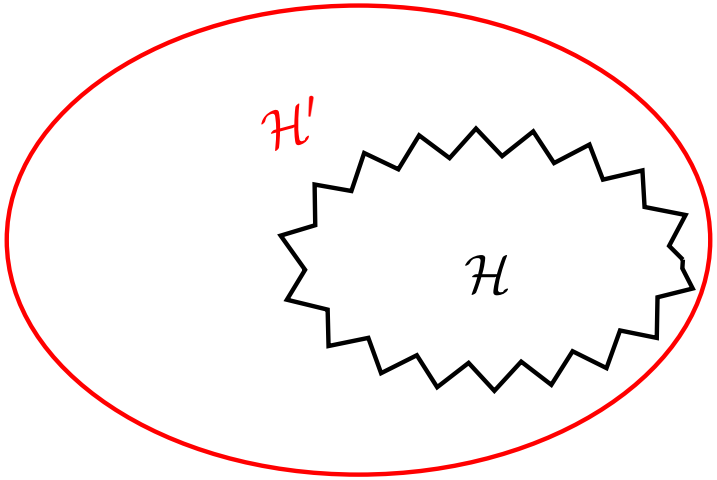
# Change the activation function

- Simpler non-linearity — replace sigmoidal activation function by the square function $\sigma(a) = a^2$
- Network implements polynomials, where the depth correlative to degree
- Is this class still very expressive ?

## Expressiveness of polynomial networks

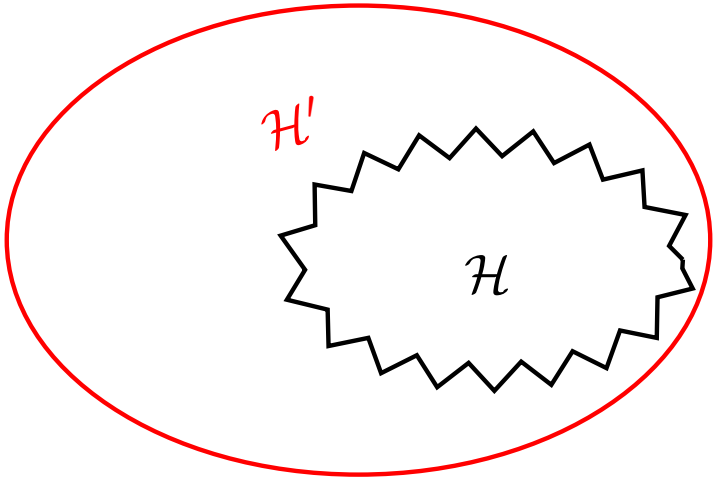
Recall the definition of $\mathcal{H}_T$ (functions that can be implemented by $T$ operations of a turing machine). Then, $\mathcal{H}_T$ is contained in the class of polynomial networks of depth $O(T \log(T))$ and size $O(T^2 \log^2(T))$

# Computational Complexity of Polynomial Networks



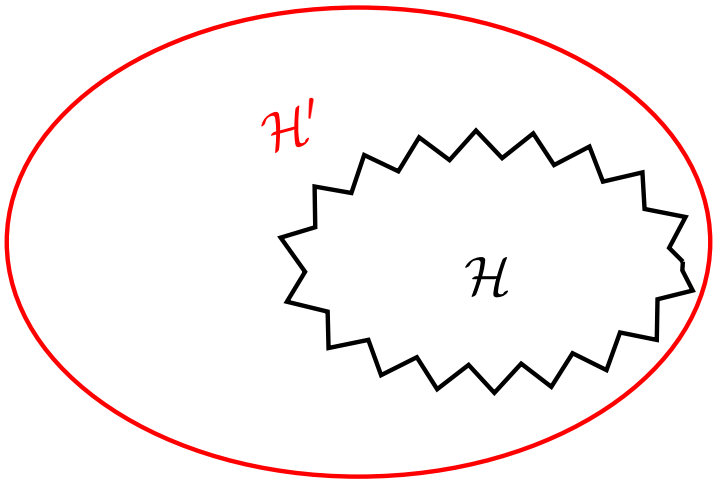

- Proper learning is still hard even for depth $2$

# Computational Complexity of Polynomial Networks



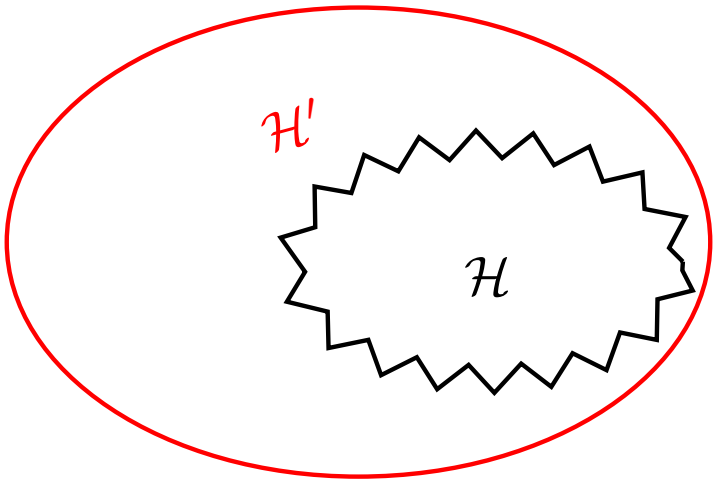

- Proper learning is still hard even for depth $2$
- But, for constant depth, improper learning works

# Computational Complexity of Polynomial Networks




- Proper learning is still hard even for depth $2$
- But, for constant depth, improper learning works
  - Replace original class with a linear classifier over all degree $2^{\text{depth}-1}$ monomials

# Computational Complexity of Polynomial Networks




- Proper learning is still hard even for depth $2$
- But, for constant depth, improper learning works
  - Replace original class with a linear classifier over all degree $2^{\text{depth}-1}$ monomials
- Size of the network is very large. Can we do better?

# Forward Greedy Selection for Polynomial Networks

- Consider depth $2$ polynomial networks
- Let $\mathcal{S}$ be the Euclidean sphere of $\mathbb{R}^d$
- Observation: Two layer polynomial networks equivalent to mappings from $\mathcal{S}$ to $\mathbb{R}$ with sparse support
- Apply forward greedy selection for learning the sparse mapping
- Main caveat: at each greedy iteration we need to find $v$ that approximately solve

$$\operatorname*{argmax}_{v \in \mathcal{S}} |\nabla_v R(w)|$$

- Luckily, this is an eigenvalue problem

$$\nabla_v R(w) = v^\top \left( \mathop{\mathbb{E}}_{(x,y)} \ell' \left( \sum_{u \in \mathrm{supp}(w)} w_u \langle u, x \rangle^2, y \right) x x^\top \right) v$$

# Back to Sigmoidal (and ReLU) Networks

- Let $\mathcal{H}_{t,n,L,\text{sig}}$ be the class of sigmoidal networks with depth $t$, size $n$, and bound $L$ on the $\ell_1$ norm of the input weights of each neuron
- Let $\mathcal{H}_{t,n,\text{poly}}$ be defined similarly for polynomial networks

# Back to Sigmoidal (and ReLU) Networks

- Let $\mathcal{H}_{t,n,L,\text{sig}}$ be the class of sigmoidal networks with depth $t$, size $n$, and bound $L$ on the $\ell_1$ norm of the input weights of each neuron
- Let $\mathcal{H}_{t,n,\text{poly}}$ be defined similarly for polynomial networks

**Theorem**

$$\forall \epsilon, \quad \mathcal{H}_{t,n,L,sig} \subset_\epsilon \mathcal{H}_{t\log(L(t-\log\epsilon)),nL(t-\log\epsilon),poly}$$

# Back to Sigmoidal (and ReLU) Networks

- Let $\mathcal{H}_{t,n,L,\text{sig}}$ be the class of sigmoidal networks with depth $t$, size $n$, and bound $L$ on the $\ell_1$ norm of the input weights of each neuron
- Let $\mathcal{H}_{t,n,\text{poly}}$ be defined similarly for polynomial networks

## Theorem

$$\forall \epsilon, \quad \mathcal{H}_{t,n,L,sig} \subset_\epsilon \mathcal{H}_{t\log(L(t-\log\epsilon)),nL(t-\log\epsilon),poly}$$

## Corollary

- *Constant depth sigmoidal networks with $L = O(1)$ are efficiently learnable !*

# Back to Sigmoidal (and ReLU) Networks

- Let $\mathcal{H}_{t,n,L,\text{sig}}$ be the class of sigmoidal networks with depth $t$, size $n$, and bound $L$ on the $\ell_1$ norm of the input weights of each neuron
- Let $\mathcal{H}_{t,n,\text{poly}}$ be defined similarly for polynomial networks

**Theorem**

$$\forall \epsilon, \quad \mathcal{H}_{t,n,L,sig} \subset_\epsilon \mathcal{H}_{t\log(L(t-\log\epsilon)),nL(t-\log\epsilon),poly}$$

**Corollary**

- *Constant depth sigmoidal networks with $L = O(1)$ are efficiently learnable !*
- *It is hard to learn polynomial networks of depth $\Omega(\log(d))$ and size $\Omega(d)$*

# Back to the Theory-Practice Gap

- In theory:
  - Hard to train depth $2$ networks

# Back to the Theory-Practice Gap

- In theory:
  - Hard to train depth $2$ networks
  - Easy to train constant depth networks with constant bound on the weights

# Back to the Theory-Practice Gap

- In theory:
  - Hard to train depth $2$ networks
  - Easy to train constant depth networks with constant bound on the weights
- In practice:
  - Provably correct algorithms are not practical ...

# Back to the Theory-Practice Gap

- In theory:
  - Hard to train depth $2$ networks
  - Easy to train constant depth networks with constant bound on the weights
- In practice:
  - Provably correct algorithms are not practical ...
  - Networks of depth $2 - 20$ are trained successfully with SGD (and strong GPU and a lot of patient)

# Back to the Theory-Practice Gap

- In theory:
  - Hard to train depth $2$ networks
  - Easy to train constant depth networks with constant bound on the weights
- In practice:
  - Provably correct algorithms are not practical ...
  - Networks of depth $2 - 20$ are trained successfully with SGD (and strong GPU and a lot of patient)

How to circumvent hardness?

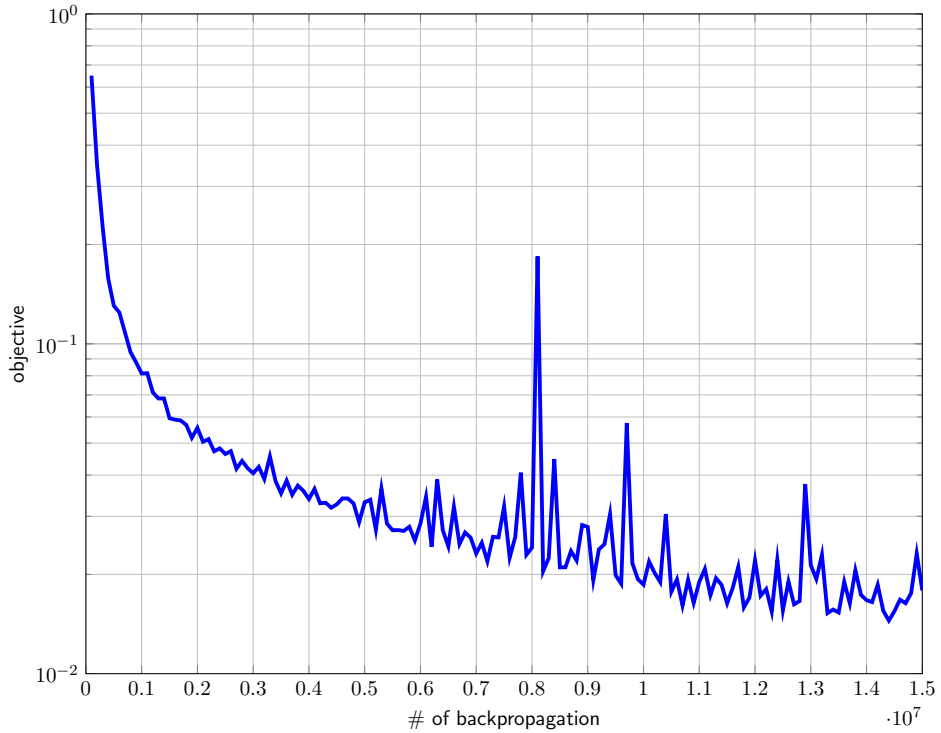
- Change the problem ...
- Add more assumptions
- Depart from worst-case analysis
  When does SGD work ? Can we make it better ?

# SGD for Deep Learning

- **Advantages:**
  - Works well in practice
  - Per iteration cost independent of $n$

# SGD for Deep Learning

- **Advantages:**
  - Works well in practice
  - Per iteration cost independent of $n$
- **Disadvantage:** slow convergence



# How to improve SGD convergence rate?

1. Variance Reduction
   - SAG, SDCA, SVRG
   - Same per iteration cost as SGD
     ... but **converges exponentially faster**
   - Designed for convex problems
     ... but can be **adapted to deep learning**

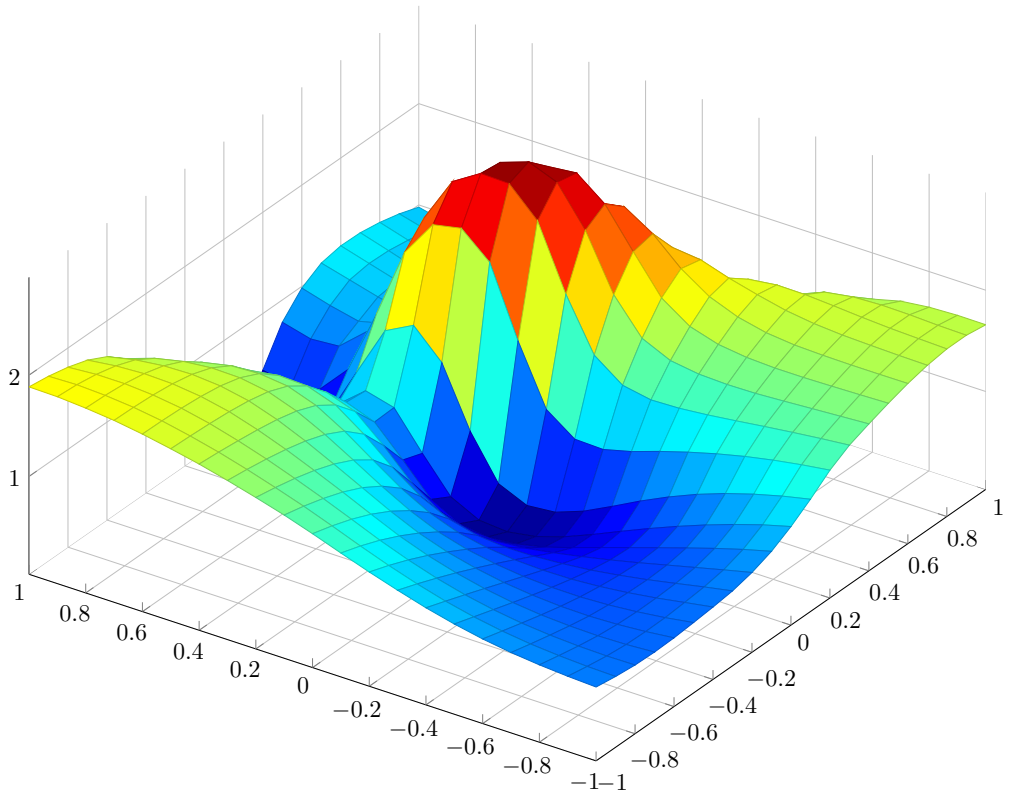
# How to improve SGD convergence rate?

1. Variance Reduction
   - SAG, SDCA, SVRG
   - Same per iteration cost as SGD
     ... but **converges exponentially faster**
   - Designed for convex problems
     ... but can be **adapted to deep learning**

2. SelfieBoost:
   - AdaBoost, with SGD as weak learner, **converges exponentially faster** than vanilla SGD
   - But yields an ensemble of networks — very expensive at prediction time
   - A new boosting algorithm that boost the performance of the same network
   - Faster convergence under some "SGD success" assumption

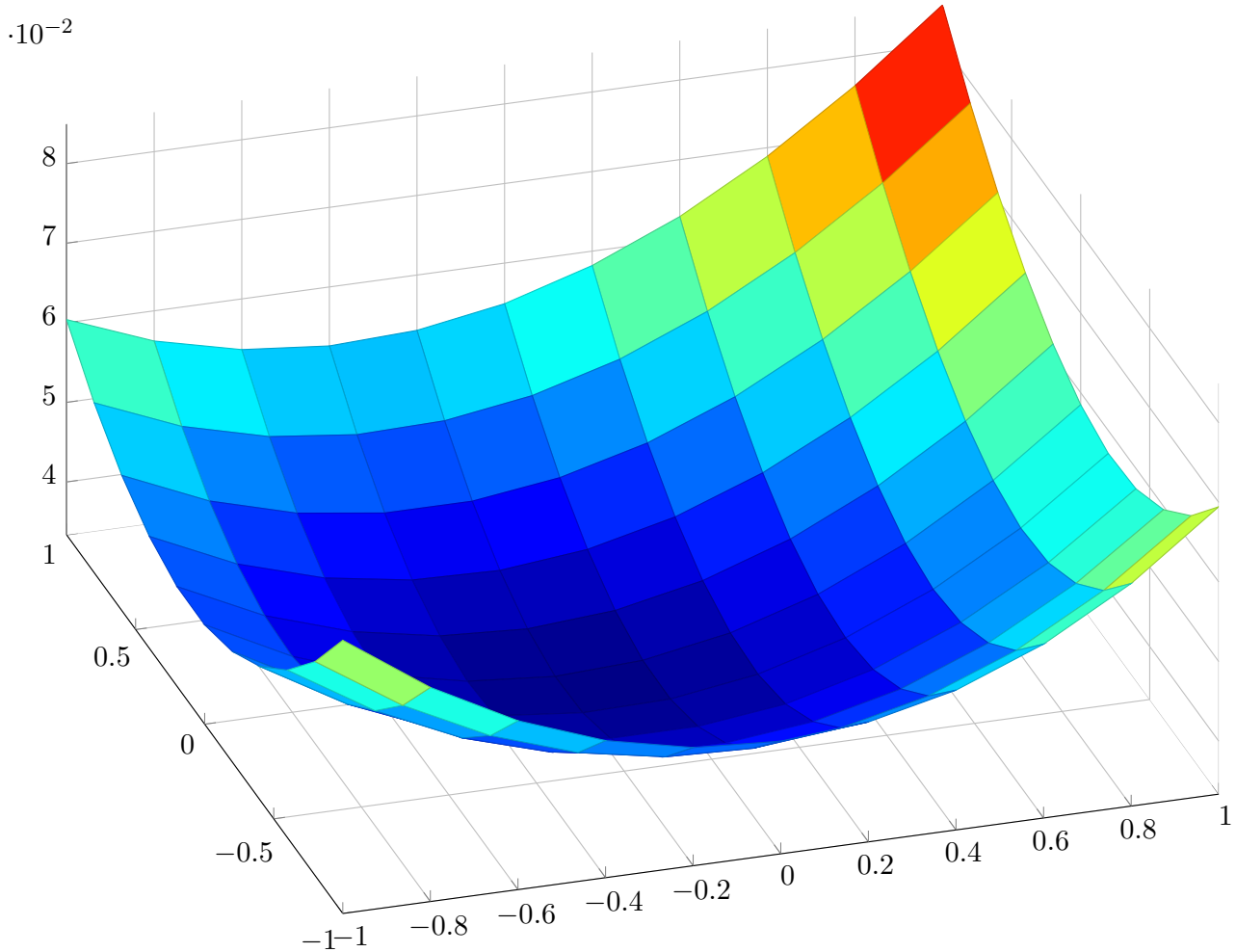
# Deep Networks are Non-Convex

- A 2-dim slice of a network with hidden layers $\{10, 10, 10, 10\}$, on MNIST, with the clamped ReLU activation function and logistic loss.
- The slice is defined by finding a global minimum (using SGD) and creating two random permutations of the first hidden layer.



# But Deep Networks Seem Convex Near a Miminum

- Now the slice is based on 2 random points at distance 1 around a global minimum



# SDCA for Deep Learning

$$\min_{w\in\mathbb{R}^d} \quad P(w) \;:=\; \frac{1}{n}\sum_{i=1}^{n}\phi_i(w) \;+\; \frac{\lambda}{2}\|w\|^2$$

- SDCA is motivated by duality, which is meaningless for non-convex functions, but yields an algorithm we can run without duality:

$$\text{"Dual" update:} \quad \alpha_i^{(t)} = \alpha_i^{(t-1)} - \eta\lambda n\left(\nabla\phi_i(w^{(t-1)}) + \alpha_i^{(t-1)}\right)$$

$$\text{"Primal dual" relationship:} \quad w^{(t-1)} = \frac{1}{\lambda n}\sum_{i=1}^{n}\alpha_i^{(t-1)}$$

$$\text{Primal update:} \quad w^{(t)} = w^{(t-1)} - \eta\left(\nabla\phi_i(w^{(t-1)}) + \alpha_i^{(t-1)}\right)$$

- Converges rate (for convex and smooth): $\left(n+\frac{1}{\lambda}\right)\log\left(\frac{1}{\epsilon}\right)$

# SDCA is SGD

- Recall that SDCA primal update rule is

$$w^{(t)} = w^{(t-1)} - \eta \underbrace{\left( \nabla \phi_i(w^{(t-1)}) + \alpha_i^{(t-1)} \right)}_{v^{(t)}}$$

and that $w^{(t-1)} = \frac{1}{\lambda n} \sum_{i=1}^{n} \alpha_i^{(t-1)}$.

# SDCA is SGD

- Recall that SDCA primal update rule is

$$w^{(t)} = w^{(t-1)} - \eta \underbrace{\left( \nabla \phi_i(w^{(t-1)}) + \alpha_i^{(t-1)} \right)}_{v^{(t)}}$$

  and that $w^{(t-1)} = \frac{1}{\lambda n} \sum_{i=1}^{n} \alpha_i^{(t-1)}$.

- Observe: $v^{(t)}$ is unbiased estimate of the gradient:

$$\begin{aligned}
\mathbb{E}[v^{(t)} | w^{(t-1)}] &= \frac{1}{n} \sum_{i=1}^{n} \left( \nabla \phi_i(w^{(t-1)}) + \alpha_i^{(t-1)} \right) \\
&= \nabla P(w^{(t-1)}) - \lambda w^{(t-1)} + \lambda w^{(t-1)} \\
&= \nabla P(w^{(t-1)})
\end{aligned}$$

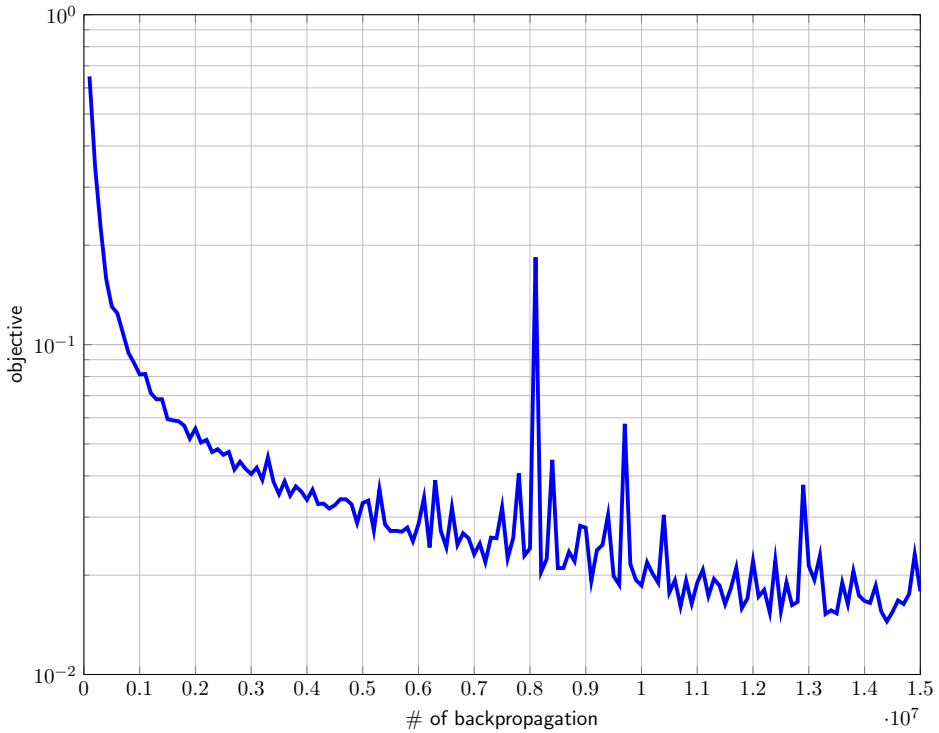
# SDCA is SGD, but better

- The update step of both SGD and SDCA is $w^{(t)} = w^{(t-1)} - \eta v^{(t)}$ where

$$v^{(t)} = \begin{cases} \nabla \phi_i(w^{(t-1)}) + \lambda w^{(t-1)} & \text{for SGD} \\ \nabla \phi_i(w^{(t-1)}) + \alpha_i^{(t-1)} & \text{for SDCA} \end{cases}$$

- In both cases $\mathbb{E}[v^{(t)} | w^{(t-1)}] = \nabla P(w^{(t)})$
- What about the variance?
- For SGD, even if $w^{(t-1)} = w^*$, the variance of $v^{(t)}$ is still constant
- For SDCA, it can be shown that the variance of $v^{(t)}$ goes to zero as $w^{(t-1)} \to w^*$

# How to improve SGD?




Why SGD is slow at the end?

- High variance, even close to the optimum
- Rare mistakes: Suppose all but $1\%$ of the examples are correctly classified. SGD will now waste $99\%$ of its time on examples that are already correct by the model

# SelfieBoost Motivation

- For simplicity, consider a binary classification problem in the realizable case
- For a fixed $\epsilon_0$ (not too small), few SGD iterations find an $\epsilon_0$-accurate solution
- However, for a small $\epsilon$, SGD requires many iterations
- Smells like we need to use boosting ....

# First idea: learn an ensemble using AdaBoost

- Fix $\epsilon_0$ (say $0.05$), and assume SGD can find a solution with error $< \epsilon_0$ quite fast
- Lets apply AdaBoost with the SGD learner as a weak learner:
  - At iteration $t$, we sub-sample a training set based on a distribution $D_t$ over $[n]$
  - We feed the sub-sample to a SGD learner and gets a weak classifier $h_t$
  - Update $D_{t+1}$ based on the predictions of $h_t$
  - The output of AdaBoost is an ensemble with prediction $\sum_{t=1}^{T} \alpha_t h_t(x)$
- The celebrated Freund & Schapire theorem states that if $T = O(\log(1/\epsilon))$ then the error of the ensemble classifier is at most $\epsilon$
- Observe that each boosting iteration involves calling SGD on a relatively small data, and updating the distribution on the entire big data. The latter step can be performed in parallel
- Disadvantage of learning an ensemble: at prediction time, we need to apply many networks

# Boosting the Same Network

- Can we obtain "boosting-like" convergence, while learning a single network?

The SelfieBoost Algorithm:

- Start with an initial network $f_1$
- At iteration $t$, define weights over the $n$ examples according to $D_i \propto e^{-y_i f_t(x_i)}$
- Sub-sample a training set $S \sim D$
- Use SGD for approximately solving the problem

$$f_{t+1} \approx \operatorname*{argmin}_g \sum_{i \in S} y_i (f_t(x_i) - g(x_i)) + \frac{1}{2} \sum_{i \in S} (g(x_i) - f_t(x_i))^2$$

# Analysis of the SelfieBoost Algorithm

- Lemma: At each iteration, with high probability over the choice of $S$, there exists a network $g$ with objective value of at most $-1/4$
- Theorem: If at each iteration, the SGD algorithm finds a solution with objective value of at most $-\rho$, then after

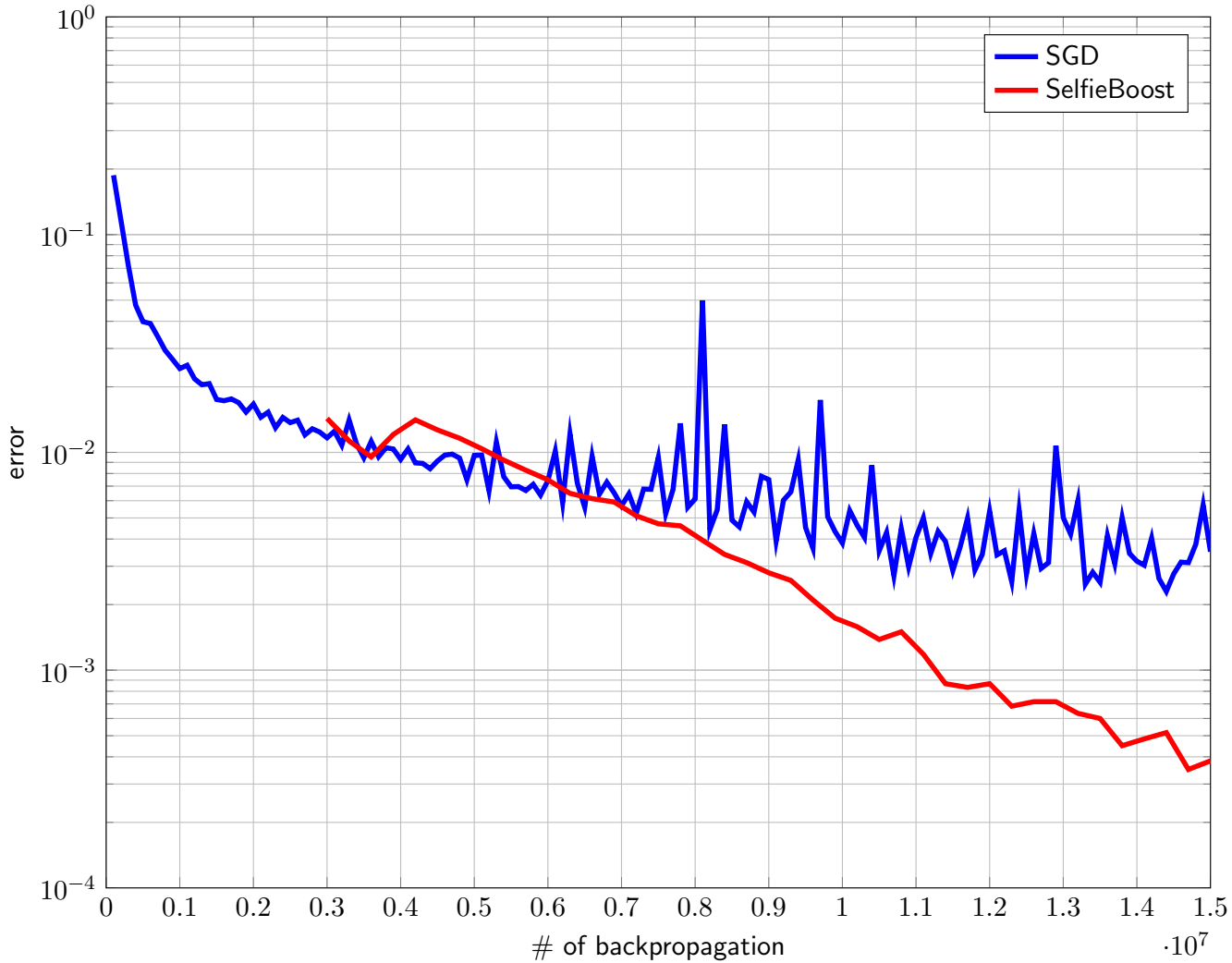
$$\frac{\log(1/\epsilon)}{\rho}$$

SelfieBoost iterations the error of $f_t$ will be at most $\epsilon$

- To summarize: we have obtained $\log(1/\epsilon)$ convergence assuming that the SGD algorithm can solve each sub-problem to a fixed accuracy (which seems to hold in practice)

# SelfieBoost vs. SGD

- On MNIST dataset, depth 5 network



# Summary

- Why deep networks: Deep networks are the ultimate hypothesis class from the statistical perspective
- Why not: Deep networks are a horrible class from the computational point of view
- This work: Deep networks with bounded depth and $\ell_1$ norm are not hard to learn
- Provably correct theoretical algorithms are in general not practical.
- Why SGD works ???
- How can we make it better ?