Distributed Deep Learning

Shai Shalev-Shwartz

School of CS and Engineering,
The Hebrew University of Jerusalem

"ICRI-CI 2015",
Intel, Haifa, May 2015

Learning and Deep Learning

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

Learning and Deep Learning

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

- For example, $\mathcal{X}$ is the space of images and $\mathcal{Y}$ is a set of k image labels (dog, cat, person, car, ...).

Learning and Deep Learning

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

- For example, $\mathcal{X}$ is the space of images and $\mathcal{Y}$ is a set of k image labels (dog, cat, person, car, ...).

Deep learning:

- Each mapping $h : \mathcal{X} \to \mathcal{Y}$ can be described by a deep network $\left(= \text{layered graph with input nodes corresponding to } x \in \mathcal{X} \text{ and output nodes corresponding to } y \in \mathcal{Y}\right)$
- The calculation performed by the network is parameterized by a weight vector $w \in \mathbb{R}^d$
- So, our goal is to learn the vector w

Deep Learning: Gradient based optimisation

Most algorithms for learning the vector w relies on gradient information:

- For each example (x, y), we have a loss function $\ell(h_w(x), y)$, that measures how bad is the prediction $h_w(x)$ when the true target is y
- The gradient, $\nabla \ell(h_w(x), y)$, is a vector $v \in \mathbb{R}^d$ such that for small $\epsilon > 0$, if we move from w to $w - \epsilon v$ we obtain the largest decrease in the loss

Deep Learning: Gradient based optimisation

Most algorithms for learning the vector w relies on gradient information:

- For each example (x, y), we have a loss function $\ell(h_w(x), y)$, that measures how bad is the prediction $h_w(x)$ when the true target is y
- The gradient, $\nabla \ell(h_w(x), y)$, is a vector $v \in \mathbb{R}^d$ such that for small $\epsilon > 0$, if we move from w to $w - \epsilon v$ we obtain the largest decrease in the loss

Stochastic Gradient Descent (SGD)

- Start with some initial $w_1 \in \mathbb{R}^d$
- For $t = 1, 2, \ldots,$

Deep Learning: Gradient based optimisation

Most algorithms for learning the vector w relies on gradient information:

- For each example (x, y), we have a loss function $\ell(h_w(x), y)$, that measures how bad is the prediction $h_w(x)$ when the true target is y
- The gradient, $\nabla \ell(h_w(x), y)$, is a vector $v \in \mathbb{R}^d$ such that for small $\epsilon > 0$, if we move from w to $w - \epsilon v$ we obtain the largest decrease in the loss

Stochastic Gradient Descent (SGD)

- Start with some initial $w_1 \in \mathbb{R}^d$
- For $t = 1, 2, \ldots,$ two weeks
  - Pick S_t: a subset of training examples at random
  - Update $w_{t+1} = w_t - \eta_t \sum_{(x,y)\in S_t} \nabla \ell(h_{w_t}(x), y)$

Why distributed learning ?

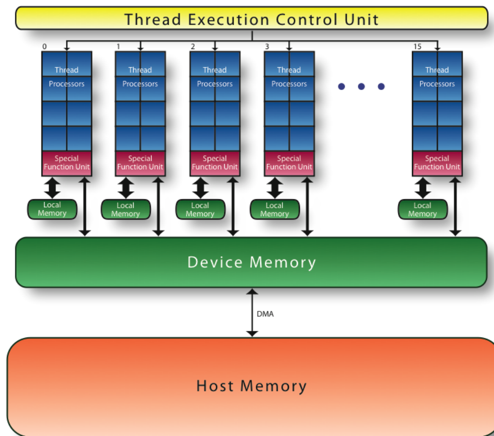
- Speed up training (two weeks is a long time for finding out you have a bug ...)
- Big Data (many many training examples)
- Huge Models (dimension of w is larger than, say, 5G) — not in this talk

Distributed Computing

- Many possible models
- We focus on two simple building blocks:
  1. Shared memory, single device, model
  2. Master-slave distributed system

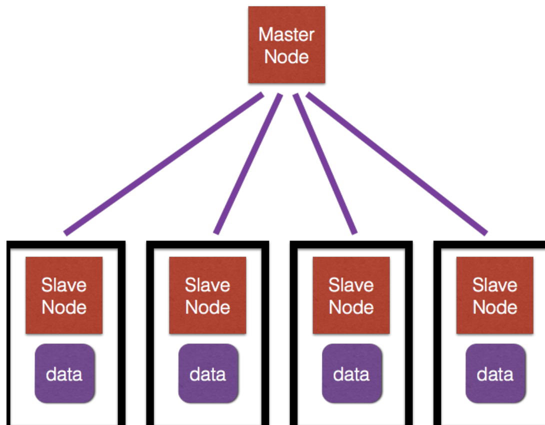
Distributed Computing

- Shared memory, single device, model: (CPU threads, GPU, ...)



Distributed Computing

- Master-slave distributed system: (Hadoop, ...)



Distributed deep learning: Main Challenge

- The most common algorithm is SGD
- SGD is inherently sequential

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples
- SGD with large mini-batches

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples
- SGD with large mini-batches
- Asynchronous SGD

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples
- SGD with large mini-batches
- Asynchronous SGD
- Baseline approaches (often works the best):
    - Hot potato

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples
- SGD with large mini-batches
- Asynchronous SGD
- Baseline approaches (often works the best):
  - Hot potato
  - Train in parallel and average the resulting models / predictions

Distributed deep learning: main approaches

For shared memory devices: SGD with small mini-batches works great

For master-slave systems:

- Forget about "stochasticity" (accelerated batch gradient descent, Hessian free, LBFGS): distribute each gradient calculation over examples
- SGD with large mini-batches
- Asynchronous SGD
- Baseline approaches (often works the best):
  - Hot potato
  - Train in parallel and average the resulting models / predictions
- Few "outer" iterations and many "inner" iterations (e.g. DANE)

Distributed deep learning: highlight of our approach

- Combine the two distributed models
- Master node has also a shared memory device
- Few “outer” iterations
- Each “outer” iteration is composed of two steps
  - Collection of “interesting” examples by the slaves
  - Fast SGD only on the “interesting” examples using the shared memory device of the master

Step I: Simple Example

- Consider a linear classification problem: $x \mapsto \mathrm{sign}(w^\top x)$
- Suppose there exists w^* that achieves zero error (with margin 1)
- We want to find a solution with accuracy ϵ
- Question 1: How many training examples do we need?

$$n = \frac{\|w^*\|^2 \, \mathbb{E}[\|x\|^2]}{\epsilon}$$

- Examples are distributed evenly and randomly over m slaves

Step I: Simple Example

Define: $c = \log(1/\epsilon)$ (practically constant ...)

Method	Iterations	Time	Communication
vanilla SGD	n	n	n

Step I: Simple Example

Define: $c = \log(1/\epsilon)$ (practically constant ...)

Method	Iterations	Time	Communication
vanilla SGD	n	n	n
Hot Potato SGD	n	n	m

Step I: Simple Example

Define: $c = \log(1/\epsilon)$ (practically constant ...)

Method	Iterations	Time	Communication
vanilla SGD	n	n	n
Hot Potato SGD	n	n	m
Acce. Batch GD	$\sqrt{n}$	$n^{1.5}/m$	$\sqrt{n}\,m$

Step I: Simple Example

Define: $c = \log(1/\epsilon)$ (practically constant ...)

Method	Iterations	Time	Communication
vanilla SGD	n	n	n
Hot Potato SGD	n	n	m
Acce. Batch GD	$\sqrt{n}$	$n^{1.5}/m$	$\sqrt{n}\,m$
Our approach	c	$c\,(n\,\epsilon + n/m)$	$c\,n\,\epsilon$

Define: $c = \log(1/\epsilon)$ (practically constant ...)

Method	Iterations	Time	Communication
vanilla SGD	n	n	n
Hot Potato SGD	n	n	m
Acce. Batch GD	$\sqrt{n}$	$n^{1.5}/m$	$\sqrt{n}\, m$
Our approach	c	$c\,(n\,\epsilon + n/m)$	$c\,n\,\epsilon$
Our (for $m = \frac{1}{\epsilon}$)	$\log(m)$	$\log(m)\,n/m$	$\log(m)\,n/m$

Describing the method

- Initialize:
  - Define $k = 4n\epsilon$
  - Master set $Q_1 = Q_t = \ldots = Q_m = n/m$ and $Z = n$
  - Each slave (in parallel) set $q_1^{(j)} = \ldots = q_{n/m}^{(j)} = 1$
- Loop: For $t = 1, 2, \ldots, c$
  - Mater sample k examples by first choosing j w.p. Q_j/Z and then asking the j'th slave to fetch a random example based on the probability $q_i^{(j)}/Q_j$
  - Master trains $h_t(x) = \mathrm{sign}(w_t^\top x)$ using vanilla SGD over the fetched examples
  - Master sends w_t to all slaves
  - Each slave updates $q_i^{(j)} = q_i^{(j)} \exp(-y_i^{(j)} h_t(x_i^{(j)}))$ and $Q_j = \sum_i q_i^{(j)}$
  - Master updates $z = \sum_j Q_j$

Describing the method

- Initialize:
  - Define $k = 4n\epsilon$
  - Master set $Q_1 = Q_t = \ldots = Q_m = n/m$ and $Z = n$
  - Each slave (in parallel) set $q_1^{(j)} = \ldots = q_{n/m}^{(j)} = 1$
- Loop: For $t = 1, 2, \ldots, c$
  - Mater sample k examples by first choosing j w.p. Q_j/Z and then asking the j'th slave to fetch a random example based on the probability $q_i^{(j)}/Q_j$
  - Master trains $h_t(x) = \mathrm{sign}(w_t^\top x)$ using vanilla SGD over the fetched examples
  - Master sends w_t to all slaves
  - Each slave updates $q_i^{(j)} = q_i^{(j)} \exp(-y_i^{(j)} h_t(x_i^{(j)}))$ and $Q_j = \sum_i q_i^{(j)}$
  - Master updates $z = \sum_j Q_j$

Looks familiar ?

What about deep learning ?

- The same approach can work as is for deep learning
- Each slave only needs to do a “forward” on all of its examples
- Master can use a fancy SGD, with GPU, on the fetched examples (which are much smaller than the original data)

What about deep learning ?

- The same approach can work as is for deep learning
- Each slave only needs to do a "forward" on all of its examples
- Master can use a fancy SGD, with GPU, on the fetched examples (which are much smaller than the original data)

Disadvantage:

- The output of the algorithm is $c = \log(1/\epsilon)$ models — at prediction time, we need to apply c networks, which can be costly

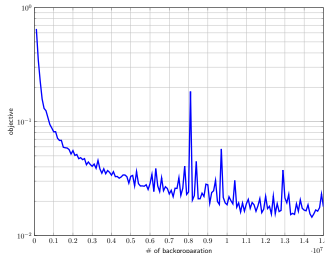
Improvement: SelfieBoost

- SelfieBoost is a boosting algorithm that outputs a single model, but still retain the nice properties of AdaBoost

SelfieBoost Motivation

Why SGD is slow at the end?

- Rare mistakes: Suppose that our current error rate is 2ϵ and we want to achieve an error rate of ϵ
- SGD picks examples at random and therefore will do nothing with probability $(1 - 2\epsilon)$
- It means that the average time until SGD will do something is $1/(2\epsilon)$
- Intuitively, this is exactly the factor $1/\epsilon$ that we are going to save



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

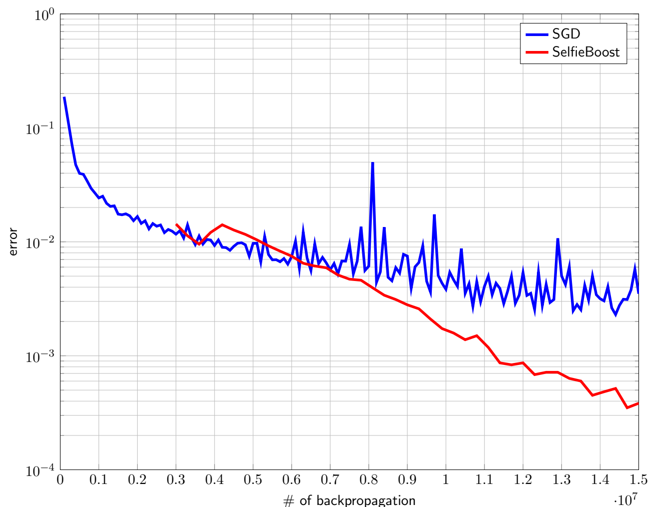
- Lemma: At each iteration, with high probability over the choice of S, there exists a network g with objective value of at most $-1/4$
- Theorem: If at each iteration, the SGD algorithm finds a solution with objective value of at most $-\rho$, then after

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

  SelfieBoost iterations the error of f_t will be at most ϵ
- 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

- Many approaches to distributed deep learning
- The Boosting approach combines a single strong master node and many "weak" slaves that brings "interesting" examples

Future Work and Open Questions:

- Boosting for noisy data
- Speeding up the other factors governing the SGD convergence rate (norm of weight vector and examples)
- Train deep networks over night