10-605/10-805: Machine Learning with Large Datasets

Fall 2022

Large-Scale Optimization

Announcements

- HW4 part A due today
  - *remember to terminate your instances when you are finished!!!*
- (Recorded) recitation this week will be the TensorFlow tutorial

Key course topics

Data preparation
- Data cleaning
- Data summarization
- Visualization
- Dimensionality reduction

Training
- Distributed ML
- **Large-scale optimization**
- Scalable deep learning
- Efficient data structures
- Hyperparameter tuning

Inference
- Hardware for ML
- Techniques for low-latency inference (compression, pruning, distillation)

Infrastructure / Frameworks
- Apache Spark
- TensorFlow
- AWS / Google Cloud / Azure

Advanced topics
- Federated learning
- Neural architecture search
- Productionizing ML

Outline

1. Stochastic gradient descent (SGD)
2. Mini-batch SGD
3. Divide & conquer methods
4. Local-updating methods

RECALL

Empirical risk minimization

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

- A popular approach in supervised ML
- Given a loss ℓ and data (x_1, y_1), … (x_n, y_n), we estimate a predictor *f* by minimizing the *empirical risk*
- We typically restrict this predictor to lie in some class, *F*
  - Could reflect our prior knowledge about the task
  - Or may be for computational convenience

LINEAR REGRESSION VIA

Empirical risk minimization

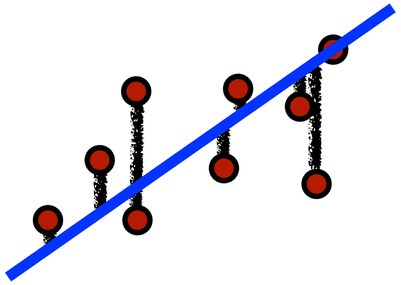
$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

linear (actually, affine) functions

square loss

- A popular approach in supervised ML
- Given a loss ℓ and data (x_1, y_1), … (x_n, y_n), we estimate a predictor *f* by minimizing the *empirical risk*
- We typically restrict this predictor to lie in some class, *F*
  - Could reflect our prior knowledge about the task
  - Or may be for computational convenience

EXAMPLE: LINEAR REGRESSION



Given *n* training points with *k* features, we define:

- $\mathbf{X} \in \mathbb{R}^{n \times k}$: matrix storing points
- $\mathbf{y} \in \mathbb{R}^{n}$: real-valued labels
- $\hat{\mathbf{y}} \in \mathbb{R}^{n}$: predicted labels, where $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$
- $\mathbf{w} \in \mathbb{R}^{k}$: regression parameters / model to learn

Least Squares Regression: Learn mapping ($\mathbf{w}$) from features to labels that minimizes residual sum of squares:

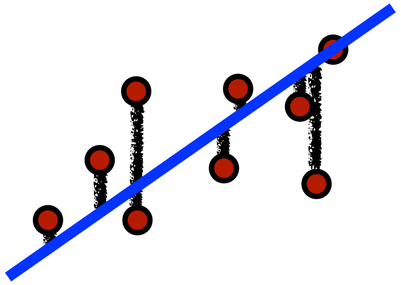
$$\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

Equivalent $\min_{\mathbf{w}} \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)})^2$ by definition of Euclidean norm

RECALL

Given *n* training points with *k* features, we define:

- $\mathbf{X} \in \mathbb{R}^{n \times k}$: matrix storing points
- $\mathbf{y} \in \mathbb{R}^{n}$: real-valued labels
- $\hat{\mathbf{y}} \in \mathbb{R}^{n}$: predicted labels, where $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$
- $\mathbf{w} \in \mathbb{R}^{k}$: regression parameters / model to learn



Ridge Regression: Learn mapping ($\mathbf{w}$) that minimizes residual sum of squares along with a regularization term:

Training Error | Model Complexity

$$\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_2^2$$

Closed-form solution: $\mathbf{w} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_k)^{-1}\mathbf{X}^\top\mathbf{y}$

free parameter trades off between training error and model complexity

Regularized Empirical Risk Minimization

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i) + \lambda R(\theta)$$

- Key idea: modify ERM objective to penalize complex models
- Helps to prevent overfitting
- Note: have re-parameterized our hypothesis, *f*, in terms of θ
- Larger λ
  - more regularization, reduce likelihood of overfitting
  - more bias, less variance

TODAY

Regularized Empirical Risk Minimization

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i) + \lambda R(\theta)$$

- How to solve ERM objectives with optimization methods?
- What techniques exist for *large-scale optimization*?

Techniques for large-scale optimization

Reduce computation:

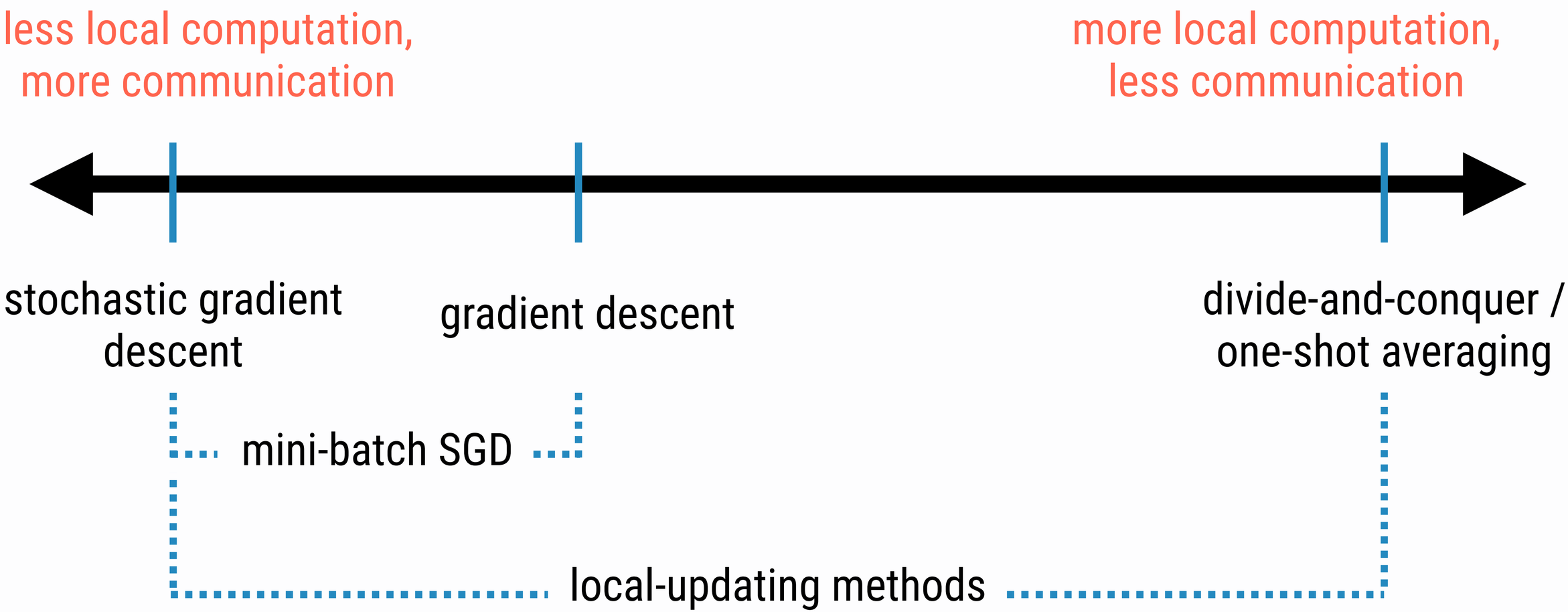
- Distribute computation
- Use first-order methods (i.e., no 2nd-order or higher gradients)
- Use stochastic methods

***Reduce communication*:**

- Keep large objects local
- Reduce iterations

Note: these techniques may be at odds with one another!

Methods for distributed optimization



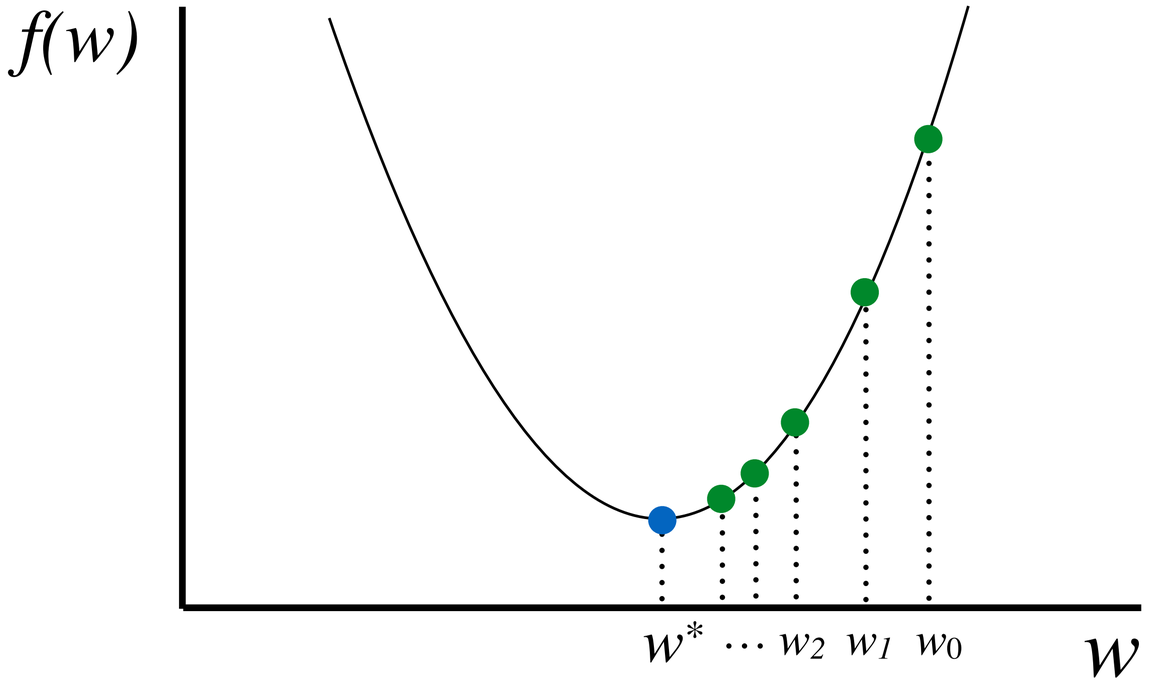
Outline

1. Stochastic gradient descent (SGD)
2. Mini-batch SGD
3. Divide & conquer methods
4. Local-updating methods

RECALL

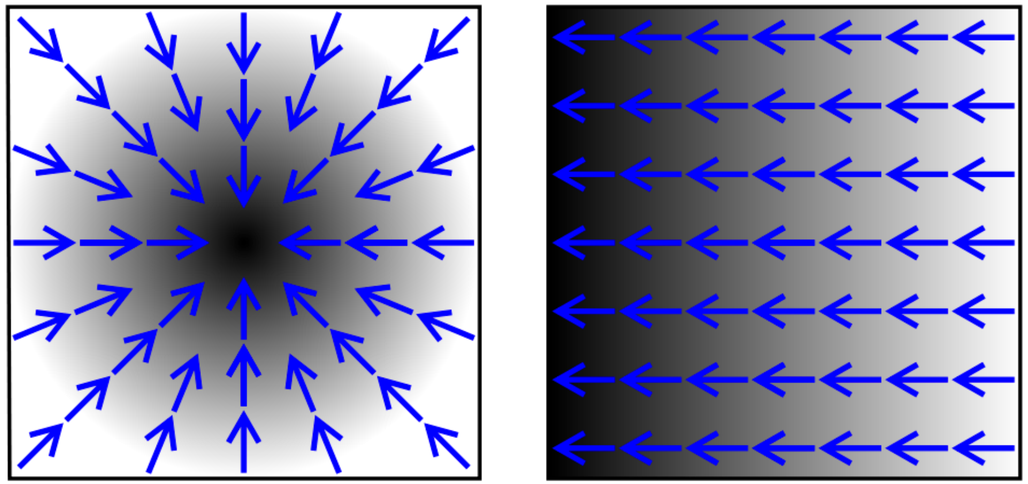
Gradient descent

Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied



RECALL

Choosing a descent direction



"Gradient2" by Sarang. Licensed under CC BY-SA 2.5 via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Gradient2.svg#/media/File:Gradient2.svg

2D Example:

- Function values are in black/white and black represents higher values
- Arrows are gradients

We can move anywhere in $\mathbb{R}^d$

Negative gradient is direction of *steepest* descent!

Step Size

Update Rule: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$

Negative Slope

RECALL

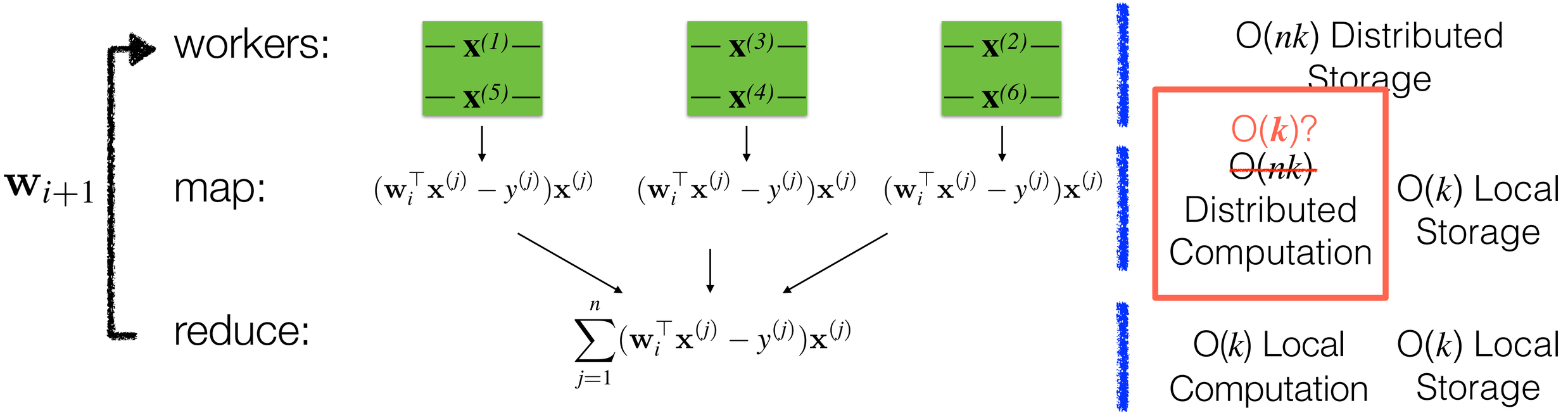
Parallel gradient descent for least squares

Vector Update: $$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$$

Compute summands in parallel!

note: workers must all have $\mathbf{w}_i$

Example: $n = 6$; 3 workers



[FOR LEAST SQUARES REGRESSION]

Stochastic gradient descent

Gradient Descent Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$

How can we reduce computation?

*Idea: approximate full gradient with **just one observation***

Stochastic Gradient Descent (SGD) Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$

the sum is gone!

MORE GENERALLY

Stochastic gradient descent

Objective we want to solve:

$$\min_{\mathbf{w}} f(\mathbf{w}) \quad \text{where} \quad f(\mathbf{w}) := \sum_{j=1}^{n} f_j(\mathbf{w})$$

Gradient Descent Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$$

Stochastic Gradient Descent (SGD) Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f_j(\mathbf{w}_i)$$

with j sampled at random

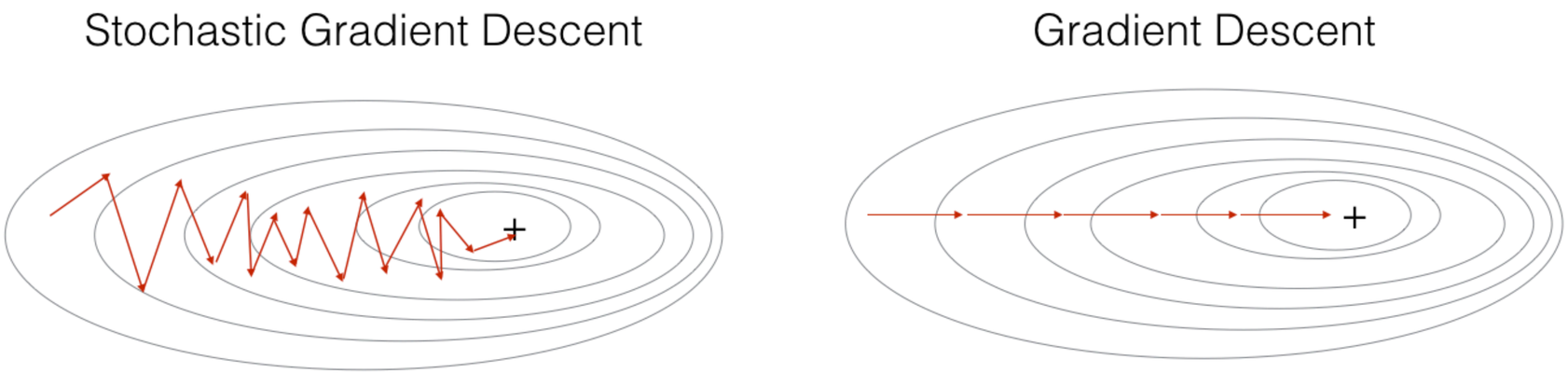
Stochastic gradient descent

What could go wrong?

- Gradient with respect to one random example might point in the *wrong direction*

Why would we expect it to work?

- On average, gradient of random examples will approximate the full gradient
- Method should converge in expectation



Stochastic gradient descent

Pros:

- Less computation
- *n* times cheaper than gradient descent at each iteration
- This faster per-iteration cost might lead to faster overall convergence in terms of FLOPS/ wall-clock time

Cons:

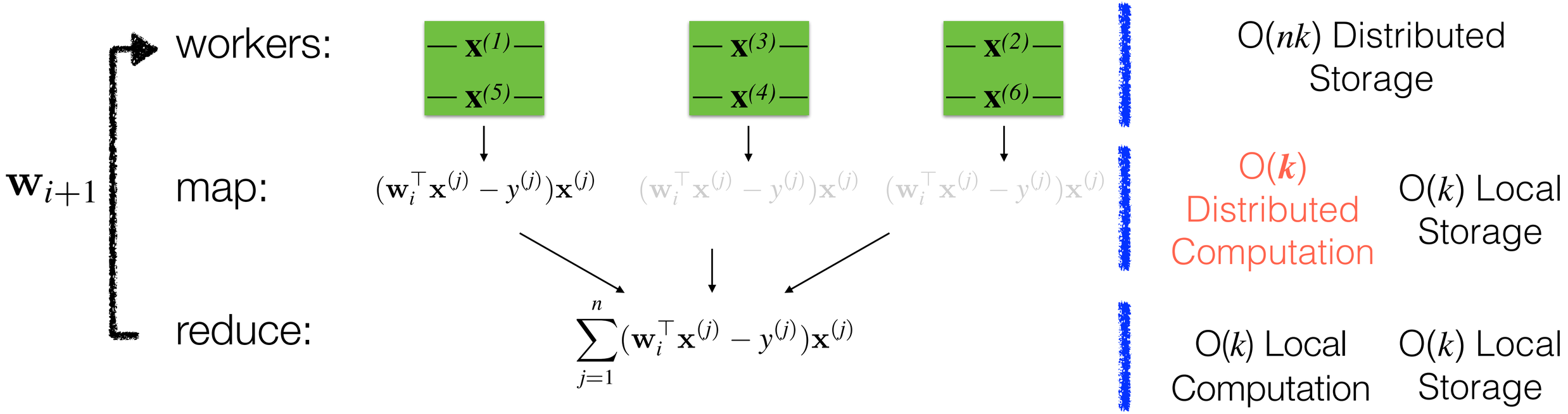
- Less stable convergence than gradient descent
- In terms of iterations: slower convergence than gradient descent
- ***More communication!***

Parallel *stochastic gradient descent* for least squares

Vector Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$

the sum is gone!

Example: $n = 6$; 3 workers



QUESTION:

Is this a good idea for distributed computing?

Issue:

- While we've reduced computation, we may require more iterations (i.e., more communication) to converge
- Additionally, if we only process *one observation* at each iteration, some machines will be idle

Another idea:

- Compute an approximate gradient with respect to a few (more than one) observations [mini-batch methods]

Outline

1. Stochastic gradient descent (SGD)
2. Mini-batch SGD
3. Divide & conquer methods
4. Local-updating methods

[FOR LEAST SQUARES REGRESSION]

Mini-batch SGD

Gradient Descent Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$$

Stochastic Gradient Descent (SGD) Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$$

use a single observation

Mini-batch SGD Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j \in \mathcal{B}_i} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$$

sum over a *mini-batch* of observations

MORE GENERALLY

Mini-batch SGD

Objective we want to solve:

$$\min_{\mathbf{w}} f(\mathbf{w}) \quad \text{where} \quad f(\mathbf{w}) := \sum_{j=1}^{n} f_j(\mathbf{w})$$

Gradient Descent Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$$

Stochastic Gradient Descent (SGD) Update:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f_j(\mathbf{w}_i)$$

with j sampled at random

Mini-batch SGD Update:

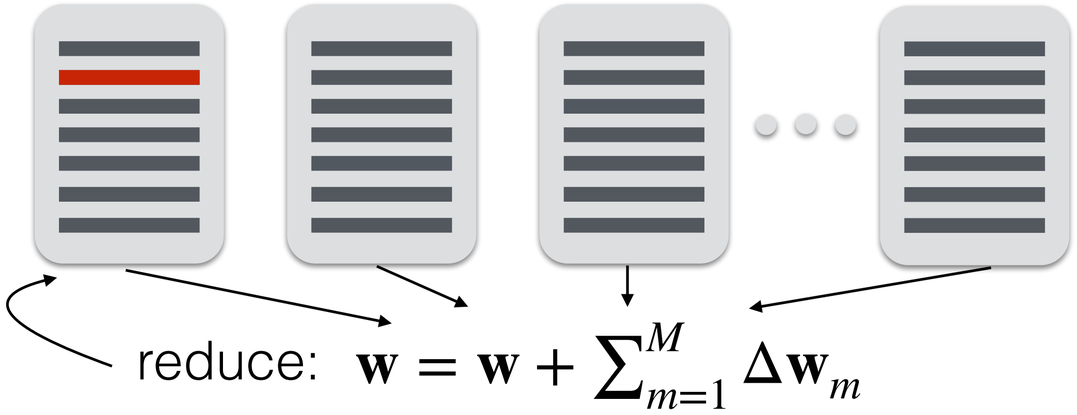
$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f_{\mathcal{B}_i}(\mathbf{w}_i)$$

with mini-batch $\mathcal{B}_i \subseteq \{1, \ldots n\}$ sampled at random

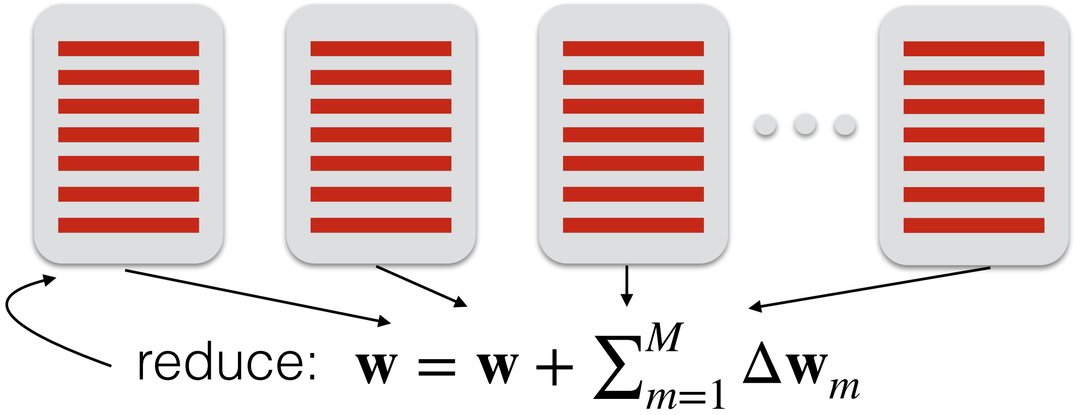
IN PICTURES

Mini-batch SGD

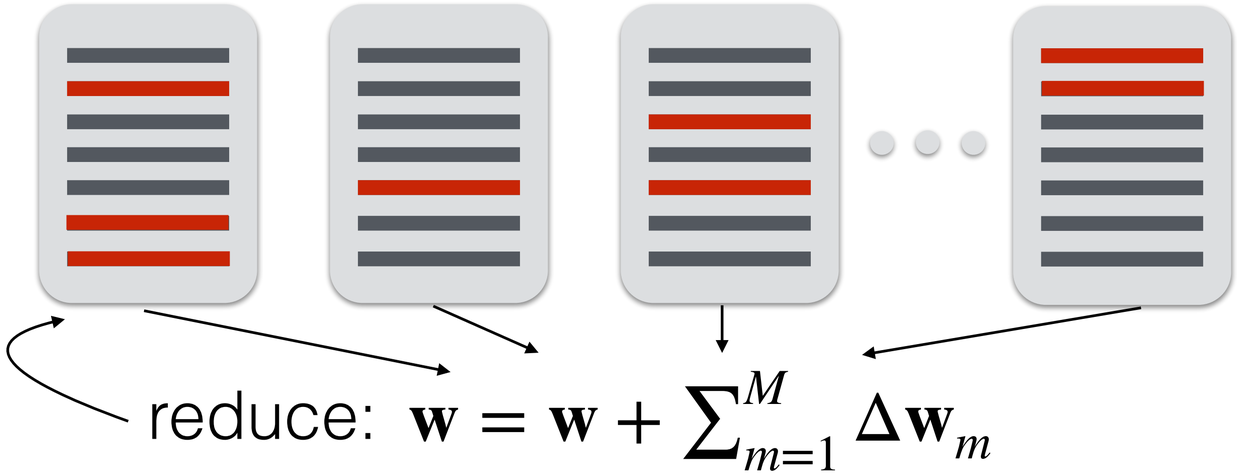
SGD: one observation



gradient descent: all observations

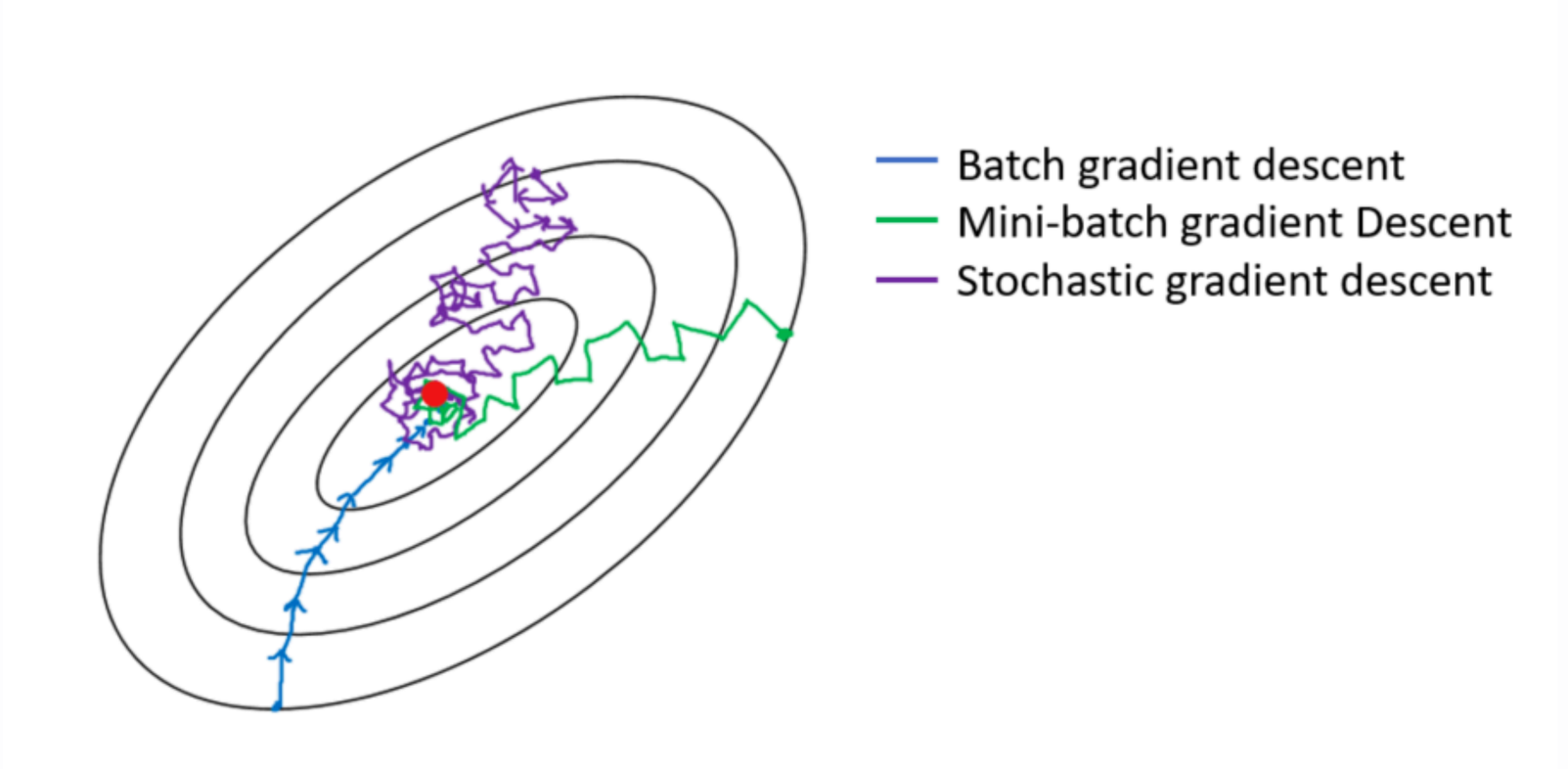


mini-batch SGD: some observations



IN PICTURES

Mini-batch SGD



https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Mini-batch SGD

Pros:

- In contrast to SGD: parallelizable
- Less computation than GD (might lead to faster overall convergence in terms of FLOPS/wall-clock time)
- Can tune computation vs. communication depending on batch size

Cons:

- In terms of iterations: slower convergence than gradient descent
- Another parameter to tune (batch size)
- Still might be too much communication …

EXAMPLE

Logistic regression

For $(\mathbf{x}_i, y_i) \in \mathbb{R}^k \times \{-1, 1\}, \ i = 1, \ldots, n,$ we aim to minimize:

$$f(\mathbf{w}) := \frac{1}{n} \sum_{i=1}^{n} \ell_{log} \left(y_i \cdot \mathbf{w}^\top \mathbf{x}_i \right)$$

Computing the gradient $\sum_{i=1}^{n} (\sigma(y_i \mathbf{w}^\top \mathbf{x}_i) - 1) y_i \mathbf{x}_i,$ is feasible when n is small to moderate, but not when n is massive

- One batch (i.e., full gradient) update costs $O(nk)$
- One mini-batch update costs $O(bk)$
- One SGD update costs $O(k)$

Credit: Ryan Tibshirani

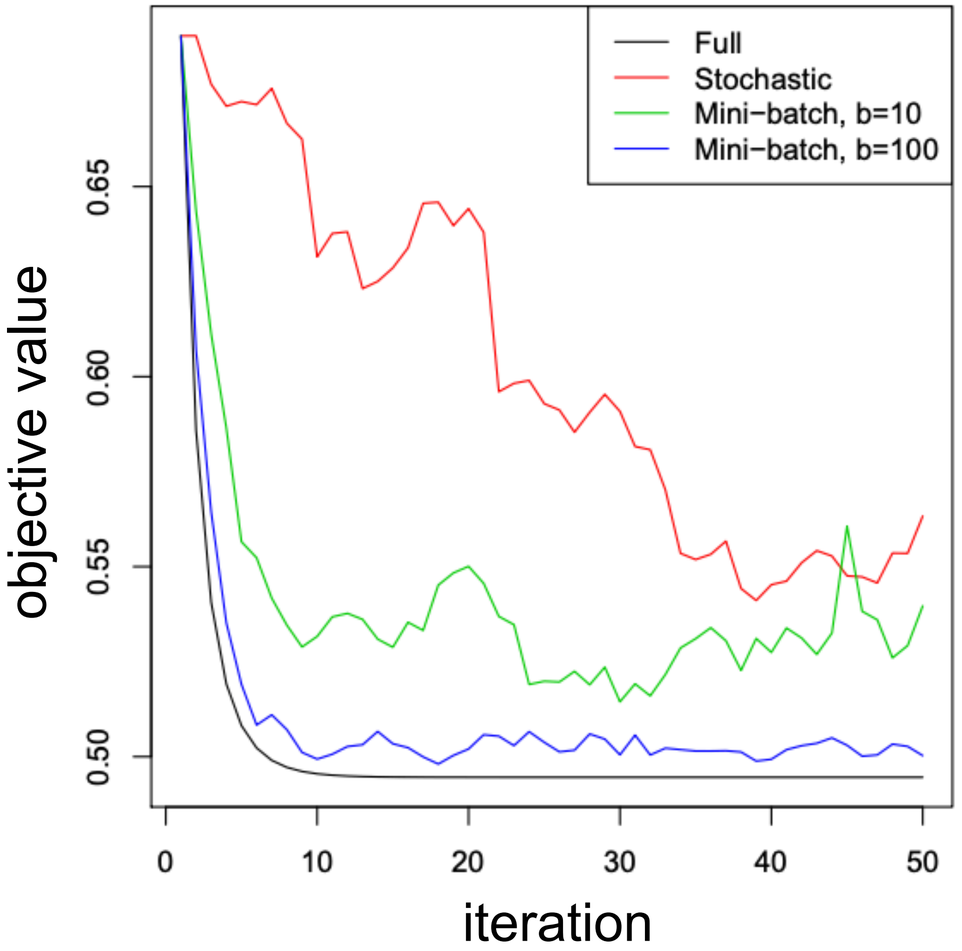
EXAMPLE: LOGISTIC REGRESSION

SGD vs. Mb-SGD vs. GD

Let's compare number of iterations vs. objective value

- Logistic regression objective with $n = 10{,}000, k = 20$
- All methods used fixed step sizes

→ *Larger batches improve convergence speed (in iterations), make convergence more stable*



Credit: Ryan Tibshirani

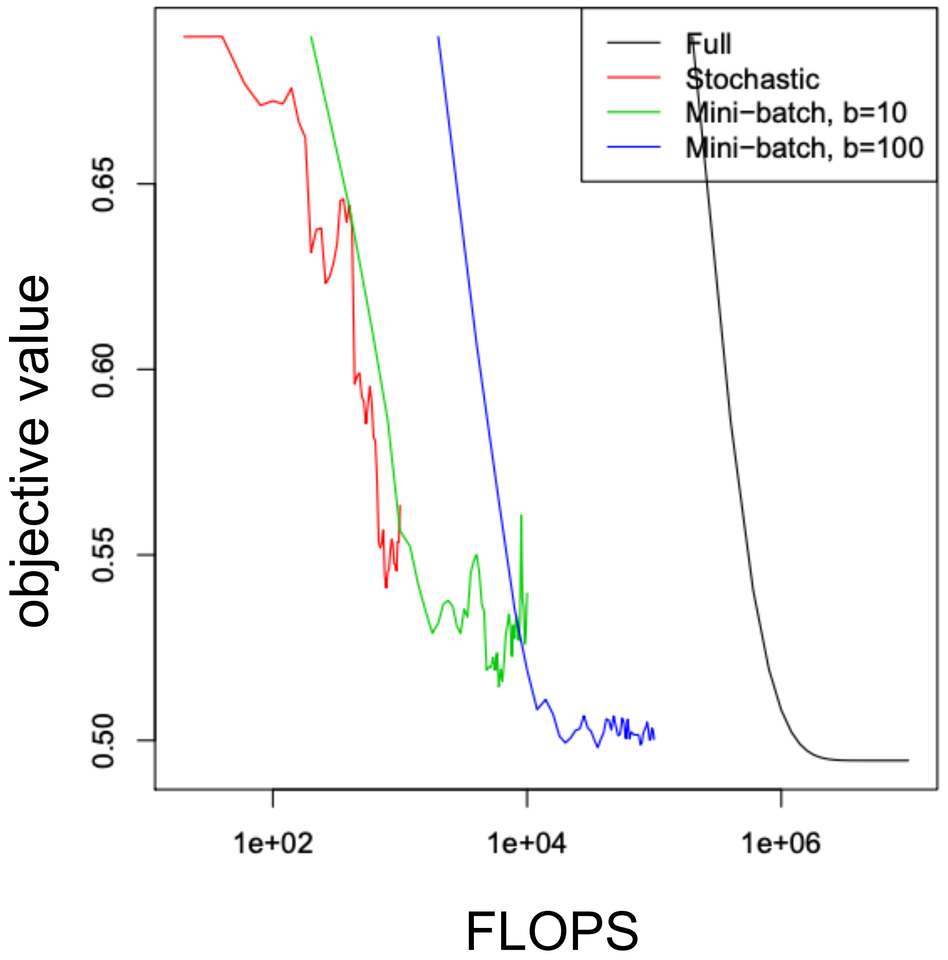
EXAMPLE: LOGISTIC REGRESSION

SGD vs. Mb-SGD vs. GD

What about FLOPS vs. objective value?

- FLOPS (floating point operations) measures the computational cost

→ *Although Mb-SGD converges more slowly than GD terms of iterations, it may converge more quickly in terms of the total computation*



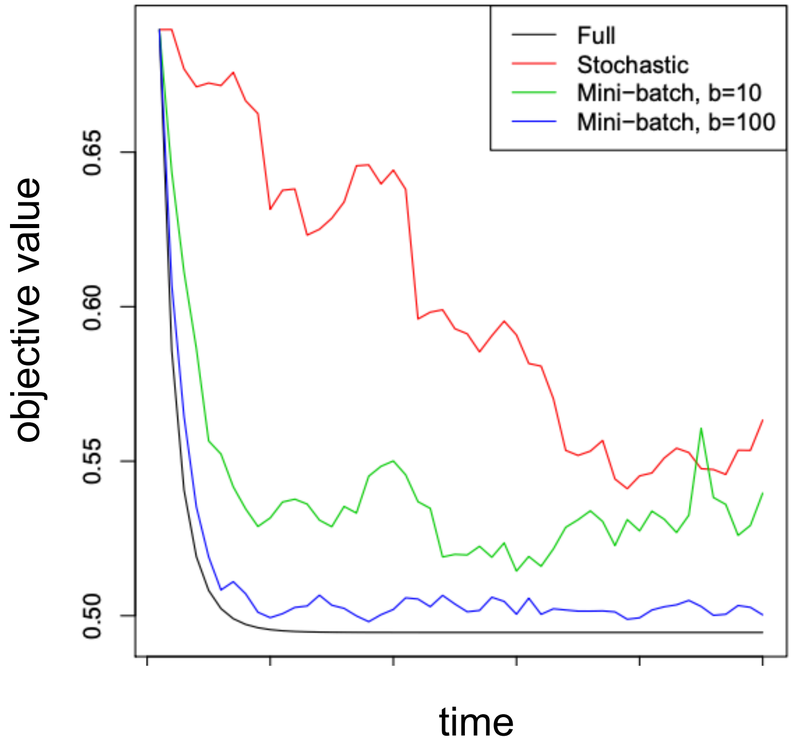
Credit: Ryan Tibshirani

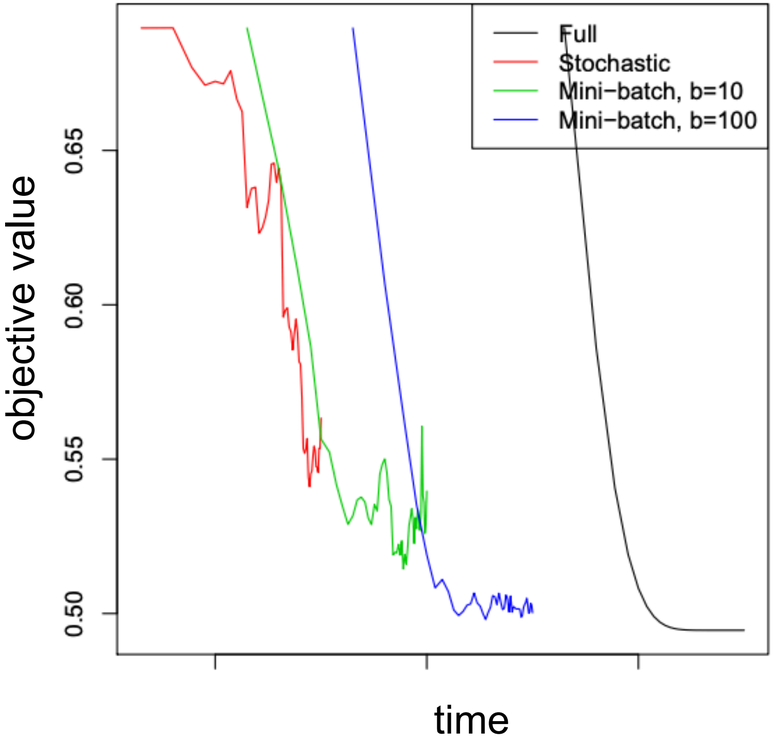
EXAMPLE: LOGISTIC REGRESSION

SGD vs. Mb-SGD vs. GD

mini-batch size can be used to tune communication vs. computation

What about parallel/distributed settings?





→ *if communication costs are high, will prefer to do more local work (i.e., GD will dominate) [left]*

→ *if communication costs are low to moderate, we will prefer to do less computation [right]*

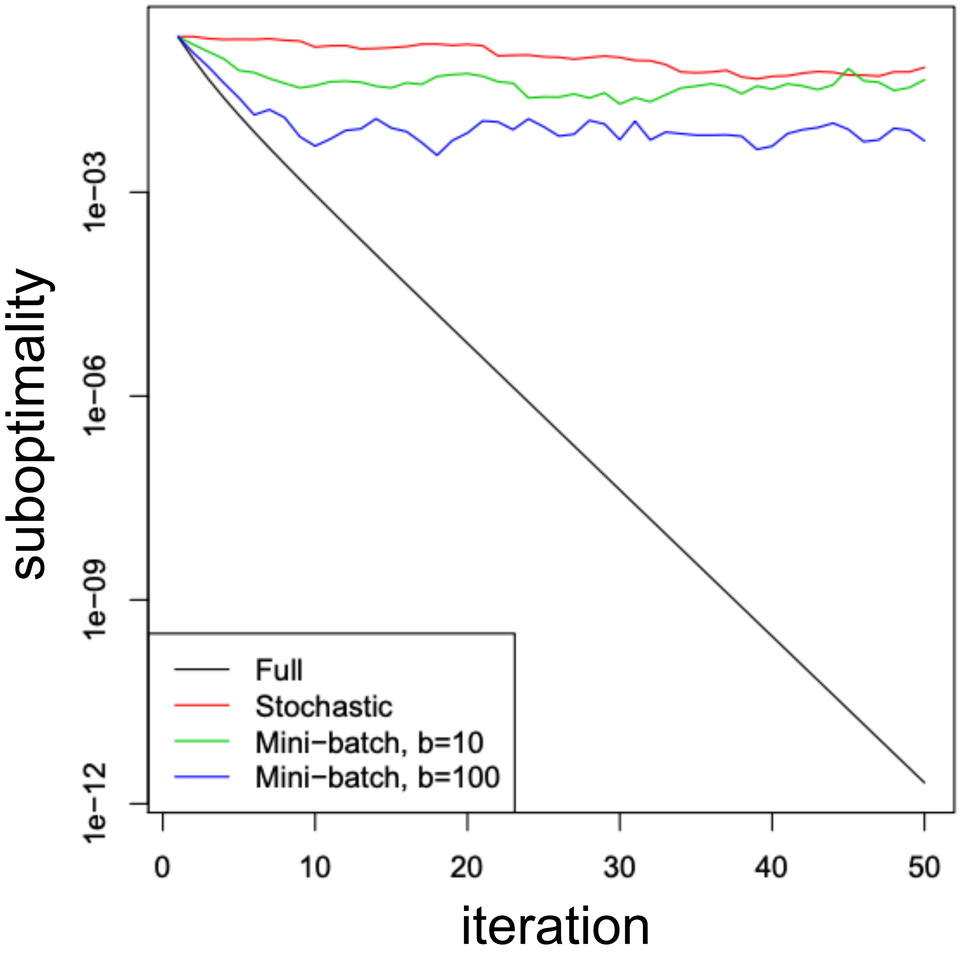
EXAMPLE: LOGISTIC REGRESSION

SGD vs. Mb-SGD vs. GD

What about number of iterations vs. suboptimality?

- 'Suboptimality' measures distance from the optimal objective

→ *Gradient descent is more precise: with fixed step sizes, mini-batch SGD (and SGD) converge within some ball of the optimal solution*



Credit: Ryan Tibshirani

Convergence rates

Iterations needed to reach ϵ-accurate solution

Condition	GD Rate	SGD Rate
Convex	$O\left(1/\epsilon^2\right)$	$O\left(1/\epsilon^2\right)$
+ Lipschitz Gradient	$O\left(1/\epsilon\right)$	$O\left(1/\epsilon^2\right)$
+ Strongly convex	$O\left(\log(1/\epsilon)\right)$	$O\left(1/\epsilon\right)$

Notes:

- In GD, we can take fixed step sizes in the two latter cases
- In SGD, we always take diminishing step sizes to control variance
- Although popular in practice, theory is mixed for Mb-SGD (more on this later)

Convergence rates

assumption: $\mu I \leq \nabla^2 f(\mathbf{w}) \leq \mathrm{L} I$

GD, fixed step size:

$$f(\mathbf{w}_t) - f(\mathbf{w}^\star) \leq \gamma^t \frac{L}{2} \|\mathbf{w}_0 - \mathbf{w}^\star\|_2^2$$

SGD, fixed step size[1]:

$$\mathbb{E}\left[f(\mathbf{w}_t) - f(\mathbf{w}^\star)\right] \leq (1 - \alpha\mu)^t (f(\mathbf{w}_0) - f(\mathbf{w}^\star)) + \frac{\alpha L \sigma_g^2}{2\mu}$$

SGD, diminishing step size[2]:

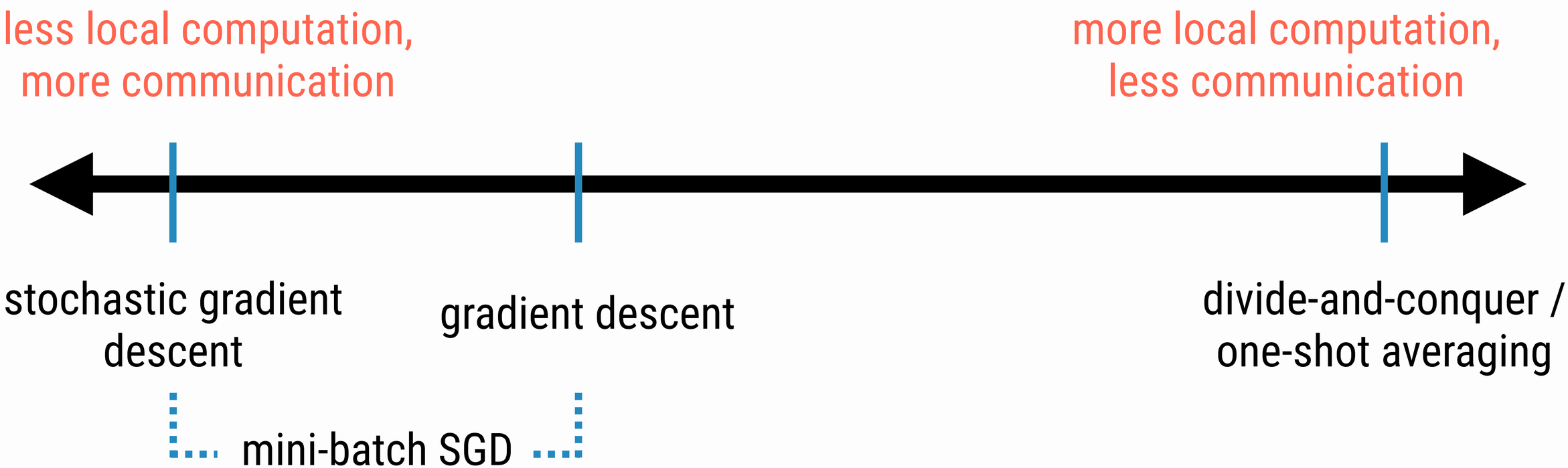
$$\mathbb{E}\left[f(\mathbf{w}_t) - f(\mathbf{w}^\star)\right] \leq \frac{\nu}{\gamma + t}$$

[1] Bottou et al, Theorem 4.6; [2] Bottou et al, Theorem 4.7

Outline

1. Stochastic gradient descent (SGD)
2. Mini-batch SGD
3. Divide & conquer methods
4. Local-updating methods

Methods for distributed optimization

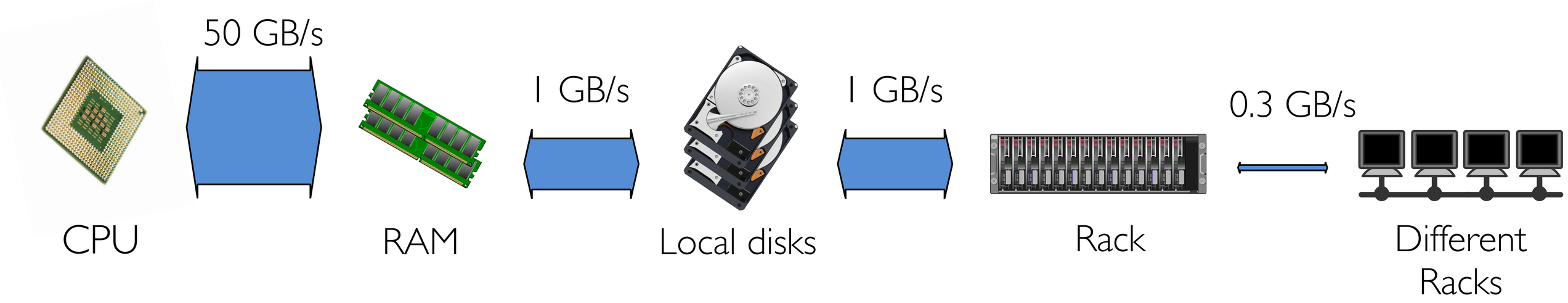


RECALL

Communication hierarchy

Access rates fall sharply with distance

- ***Parallelism makes computation fast***
- ***Network makes communication slow***



Be mindful of this hierarchy when developing parallel/distributed algorithms!

3rd Rule of thumb ("10-605/805 principles")
Minimize network communication — *reduce iterations*

Extreme: **Divide-and-conquer / One-shot averaging**

- Fully process each partition locally, communicate final result
- Single iteration; minimal communication
- Approximate results

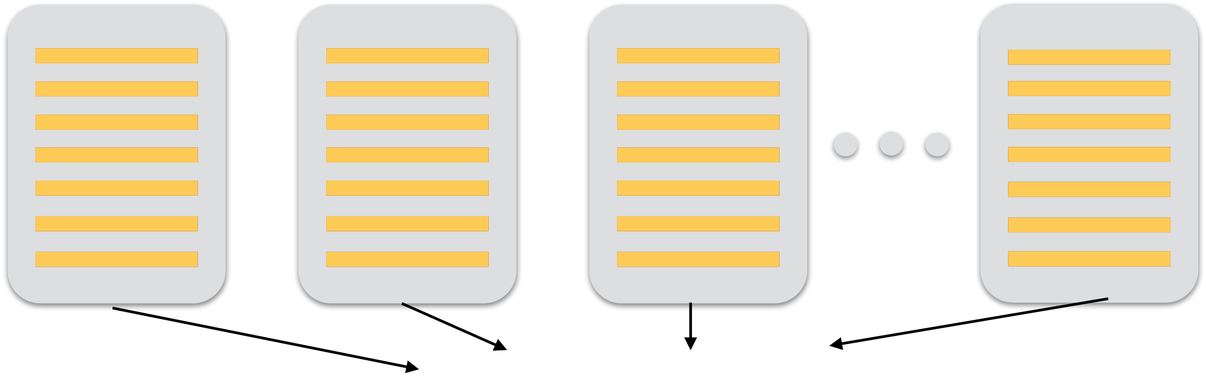
```
> w = train.mapPartitions(localLinearRegression)
                        .reduce(combineLocalRegressionResults)
```

```
> for i in range(numIters):
      alpha_i = alpha / (n * np.sqrt(i+1))
      gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
      w -= alpha_i * gradient
```

IN PICTURES

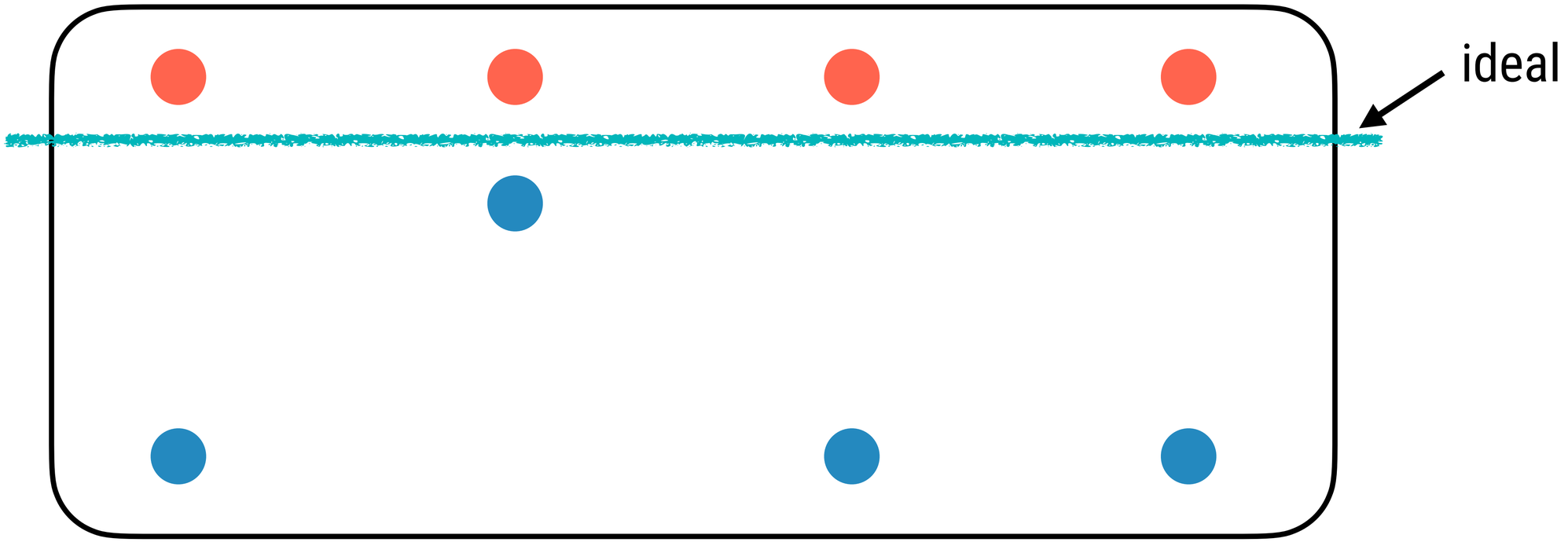
One-shot averaging

map: find optimal local model $\mathbf{w}_m^\star$

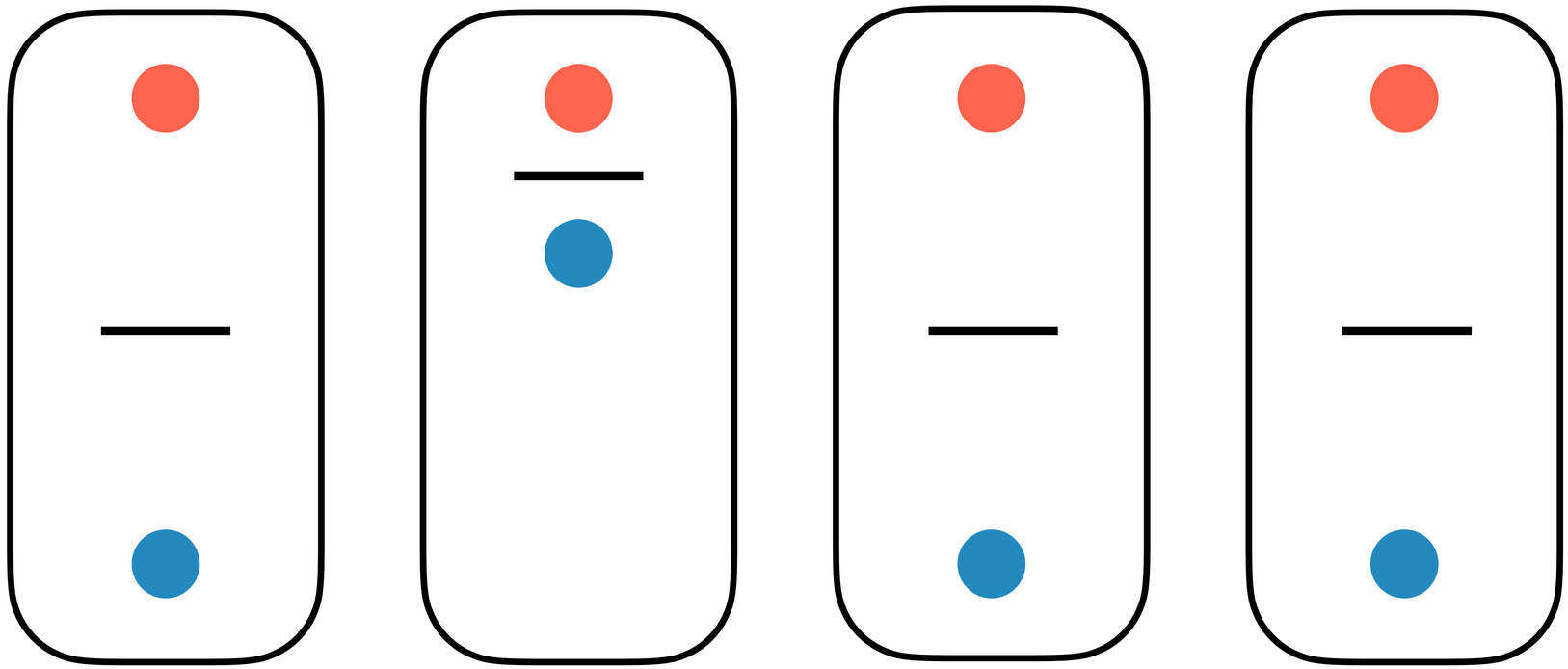


average: $\mathbf{w} := \frac{1}{M} \sum_{m=1}^{M} \mathbf{w}_m^\star$

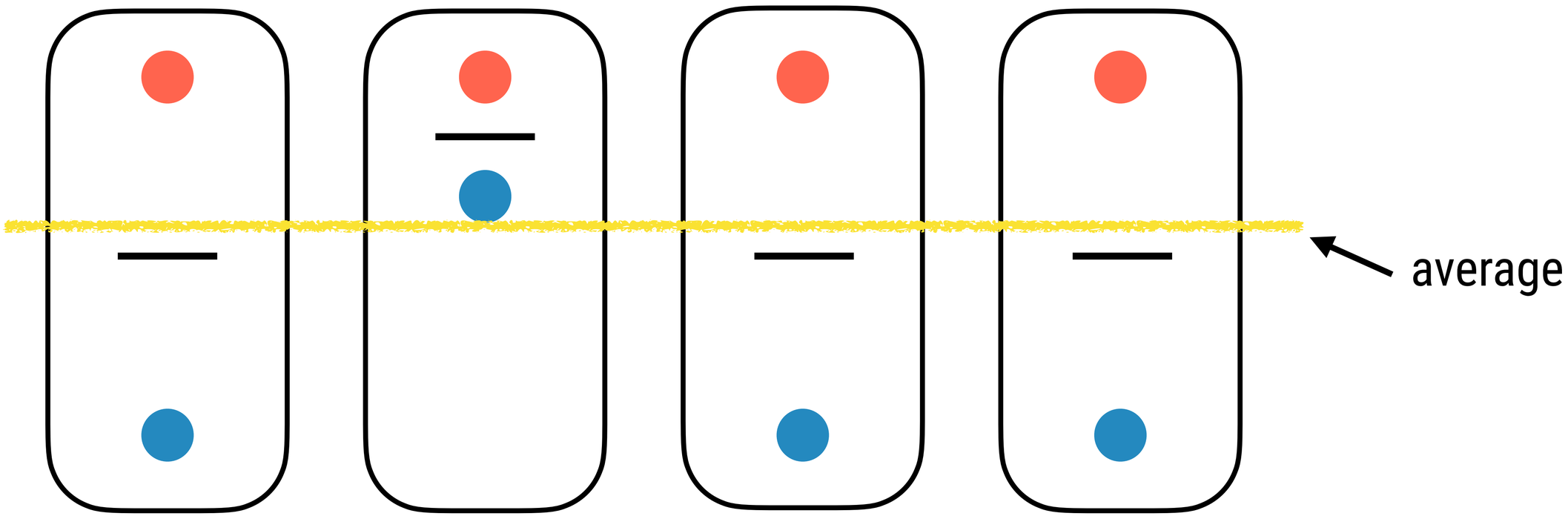
Challenge: One-shot averaging might not work



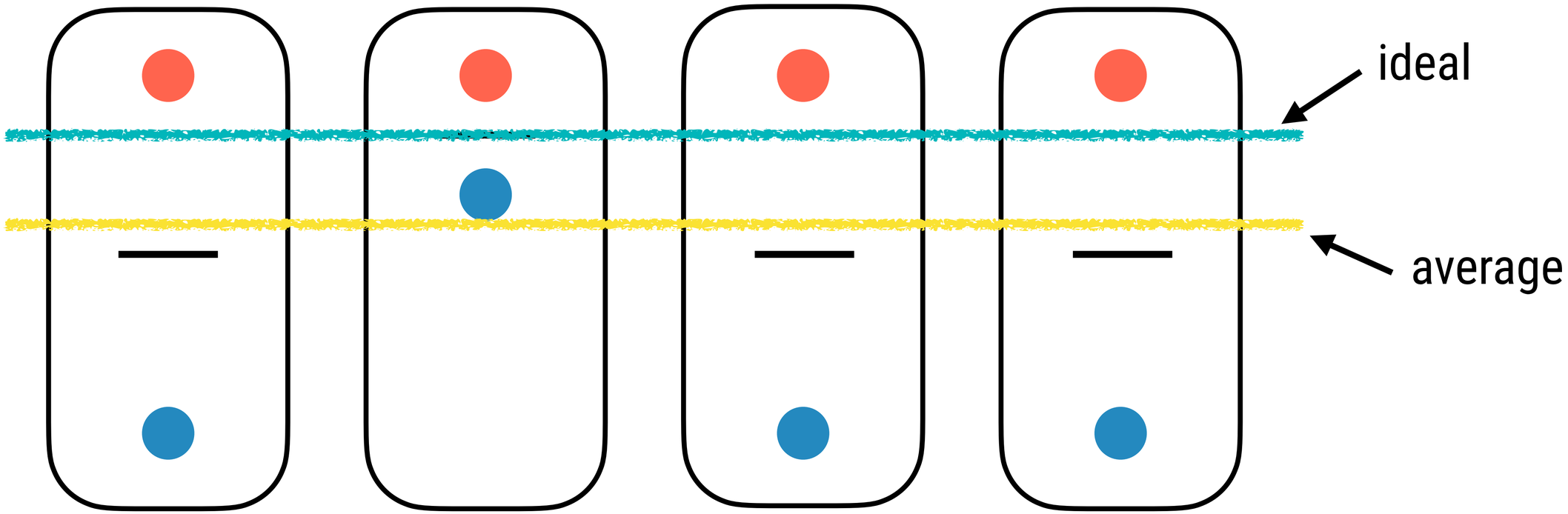
Challenge: One-shot averaging might not work



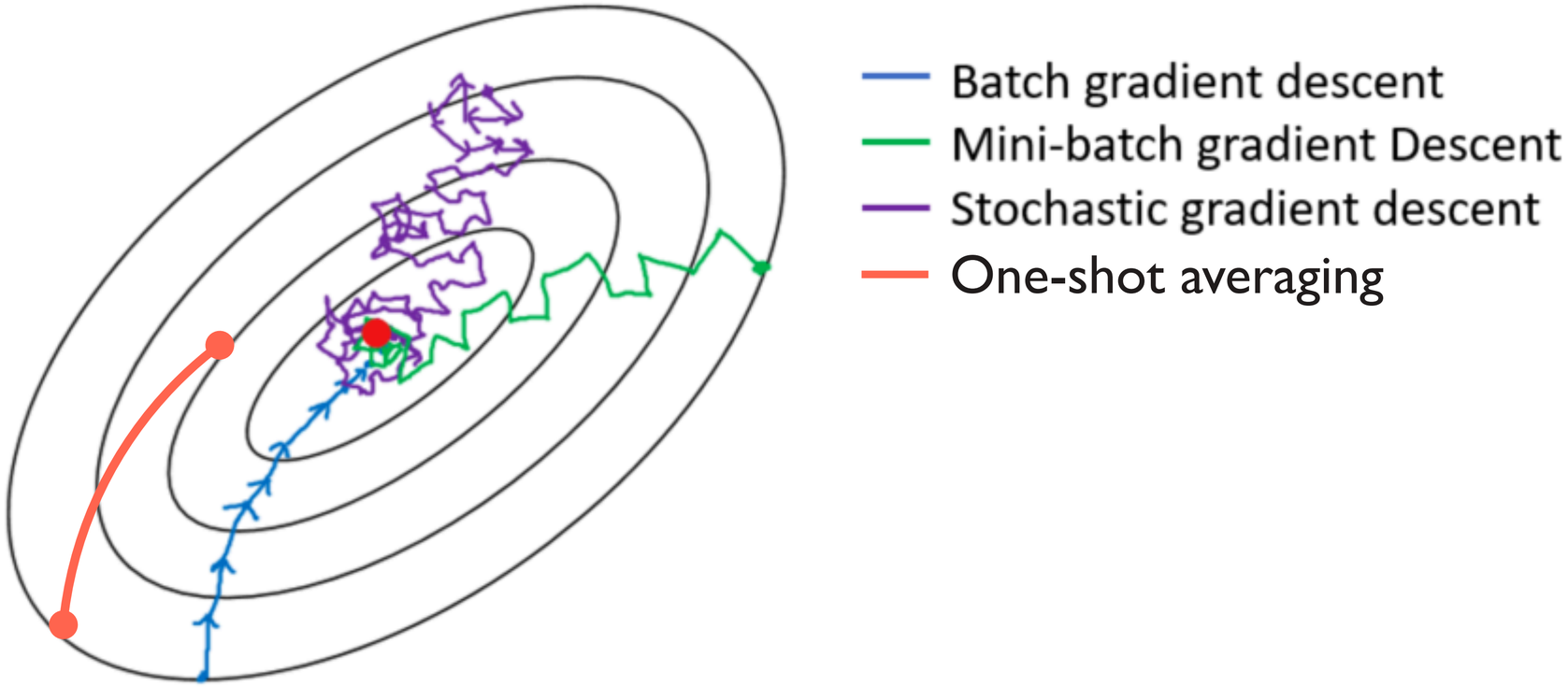
Challenge: One-shot averaging might not work



Challenge: One-shot averaging might not work



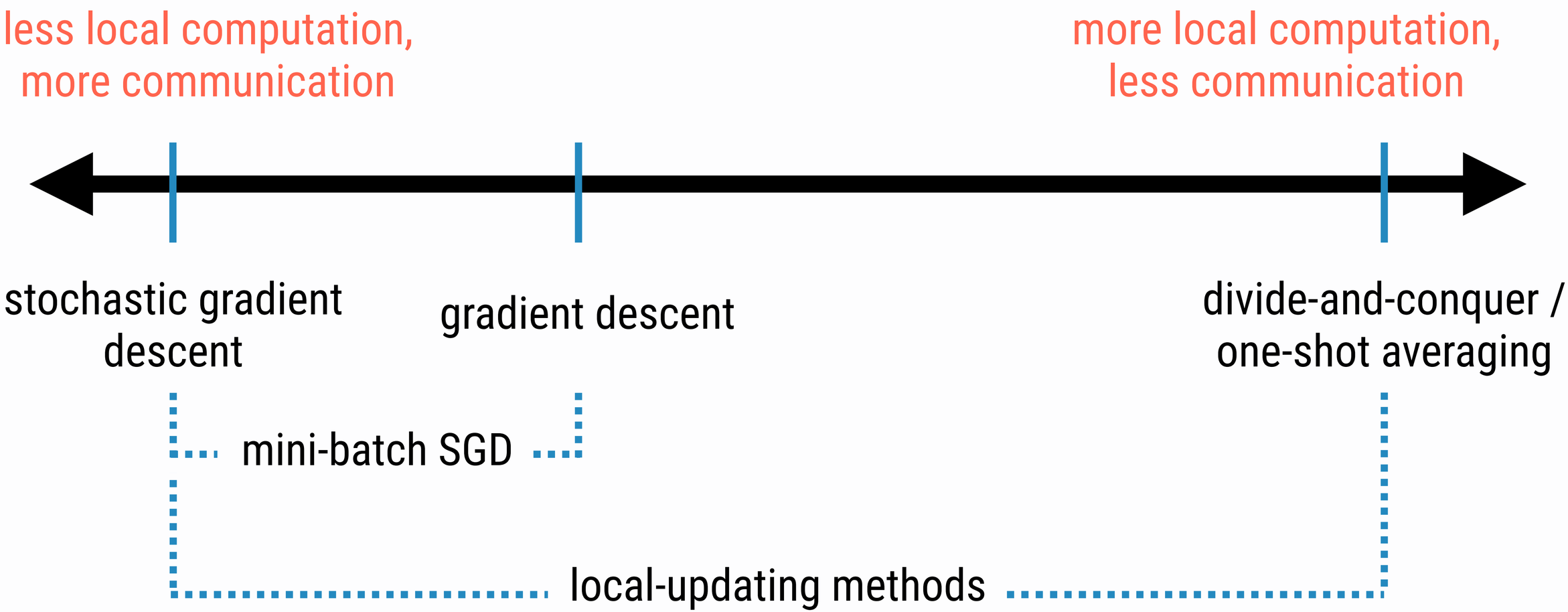
Challenge: One-shot averaging might not work



Outline

1. Stochastic gradient descent (SGD)
2. Mini-batch SGD
3. Divide & conquer methods
4. Local-updating methods

Methods for distributed optimization



3rd Rule of thumb

Minimize network communication — *reduce iterations*

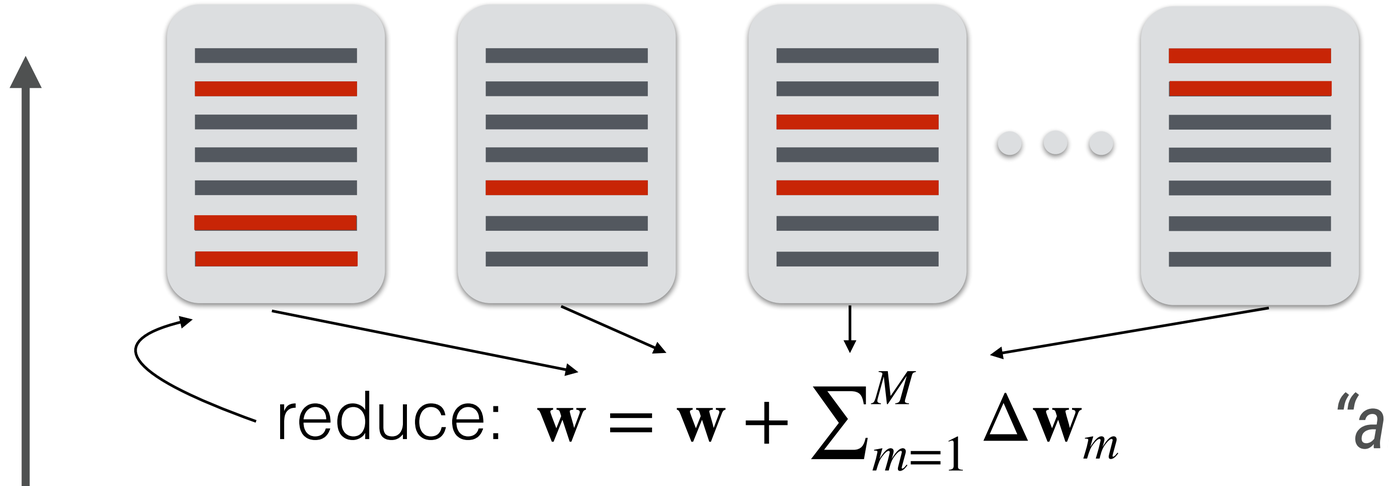
Less extreme: **Local-updating methods**

- Do more work locally than gradient descent before communicating
- Will discuss one particular method: CoCoA

```
> for i in range(fewerIters):
    update = train.mapPartitions(doSomeLocalGradientUpdates)
                  .reduce(combineLocalUpdates)
    w += update
```

```
> for i in range(numIters):
    alpha_i = alpha / (n * np.sqrt(i+1))
    gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
    w -= alpha_i * gradient
```

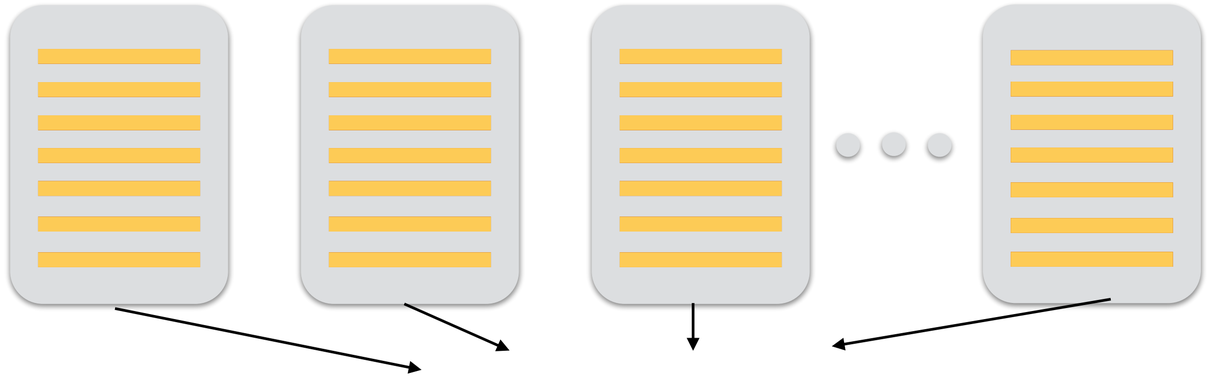
prior work: distributed optimization



reduce: $\mathbf{w} = \mathbf{w} + \sum_{m=1}^{M} \Delta\mathbf{w}_m$

✔ convergence guarantees
✗ slow

"always communicate"

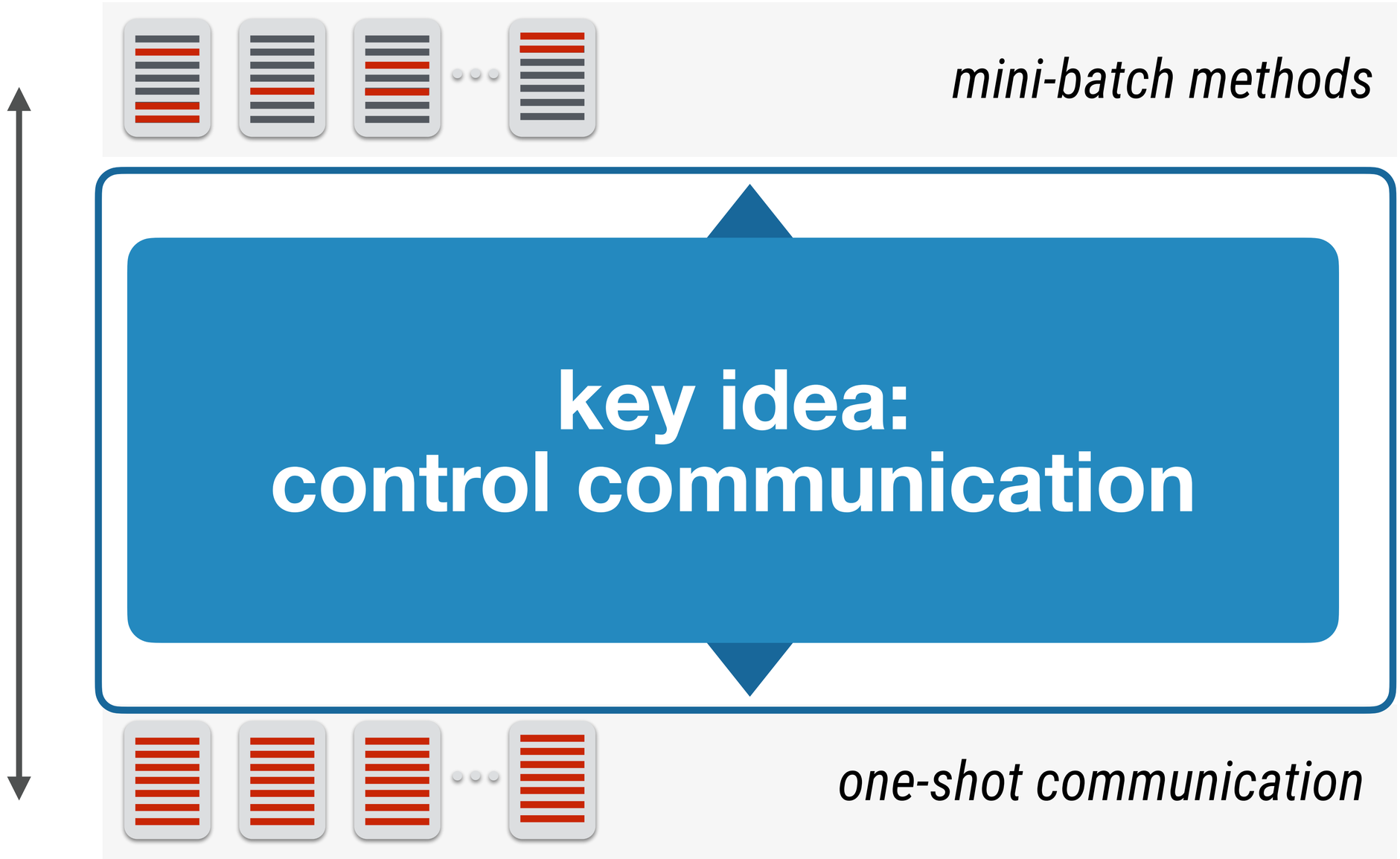


average: $\mathbf{w} := \frac{1}{M} \sum_{m=1}^{M} \mathbf{w}_m^{\star}$

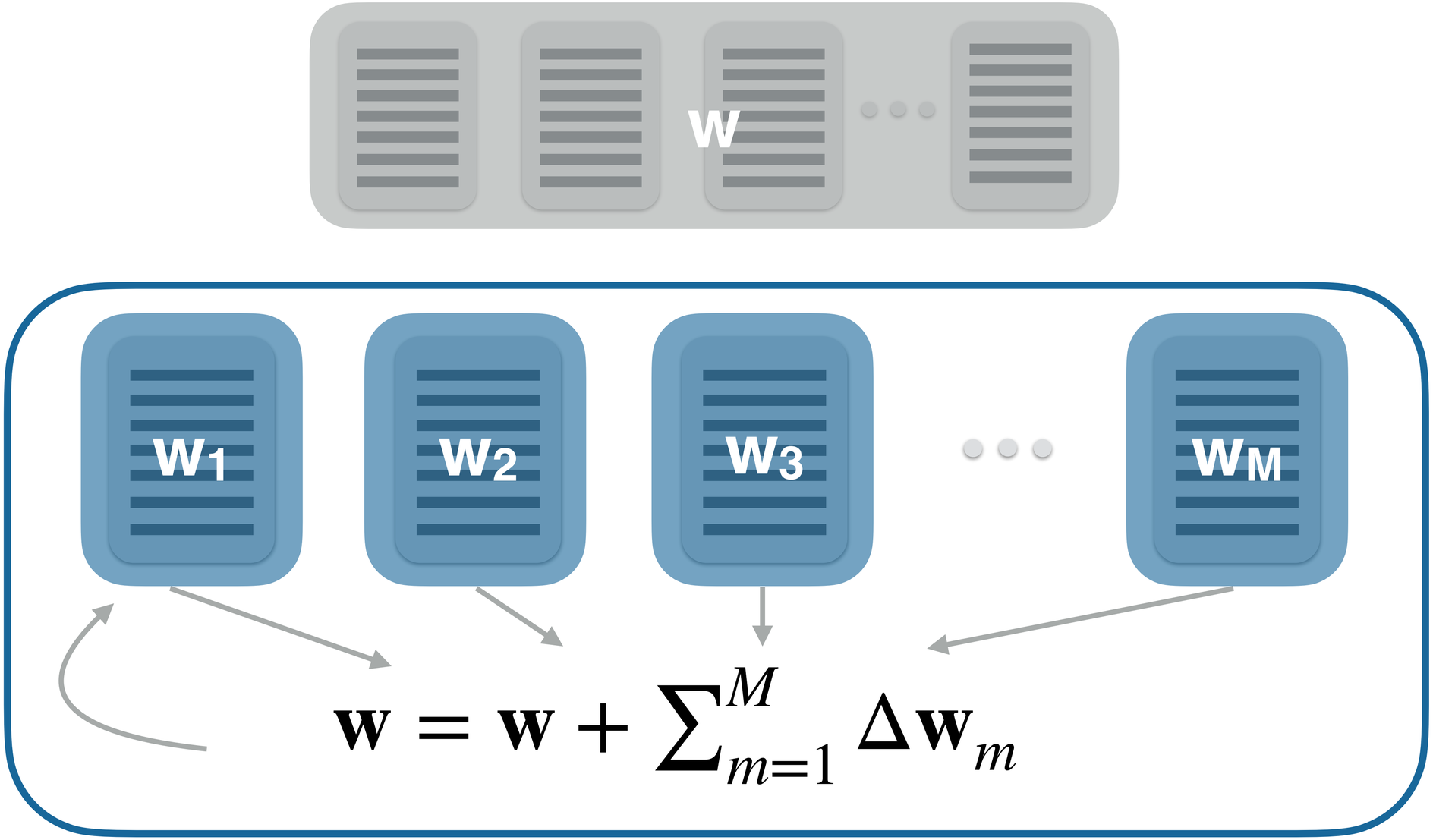
✔ fast
✗ convergence not guaranteed

"never communicate"

CoCoA: communication-efficient distributed optimization



CoCoA: subproblems



CoCoA: communication parameter

Main assumption:
each subproblem is solved to accuracy Θ

$$\Theta \in [0, 1) \approx$$

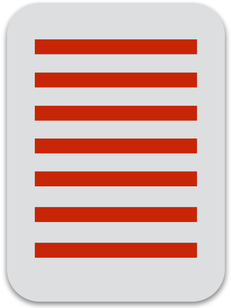
amount of local computation vs. communication

exactly solve

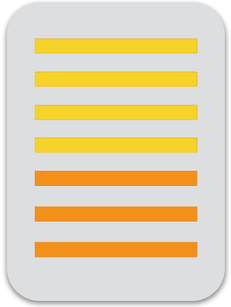
inexactly solve

KEY IDEA: MAKE COMMUNICATION A PARAMETER

Mini-batch local iterations
in (0,n]



CoCoA local iterations
in (0,∞)



✔ CoCoA allows full flexibility

[AT A HIGH LEVEL]

CoCoA Framework

Input: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ distributed over K machines

for $t = 1, 2, \ldots, T$

  for ***machines*** $k = 1, 2, \ldots, K$ ***in parallel***

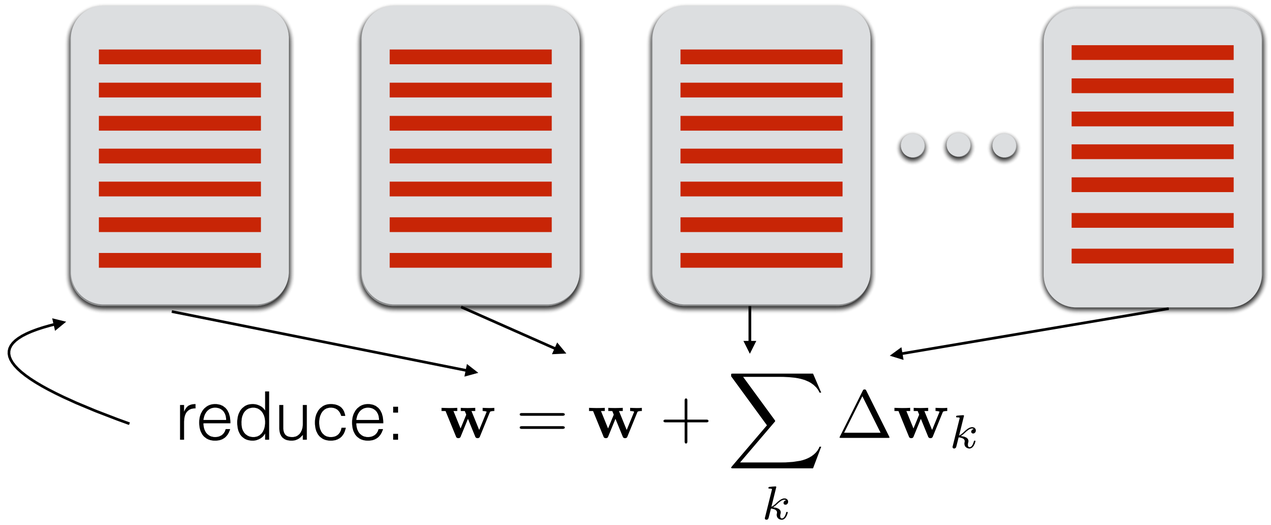
    compute θ-approximate solution to local subproblem

    return $\Delta\mathbf{w}_k$

  end

  reduce: $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \sum_{k=1}^K \Delta\mathbf{w}_k$

end



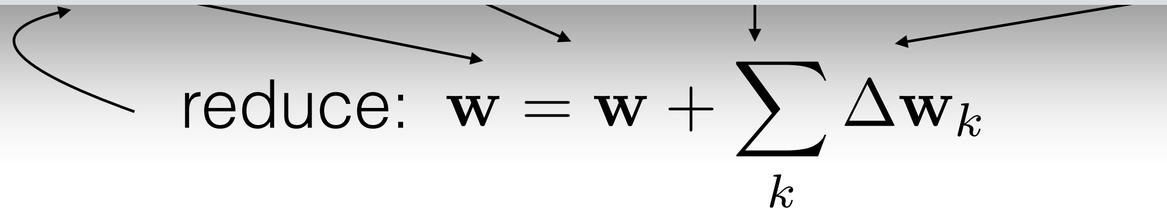
[AT A HIGH LEVEL]

CoCoA Framework

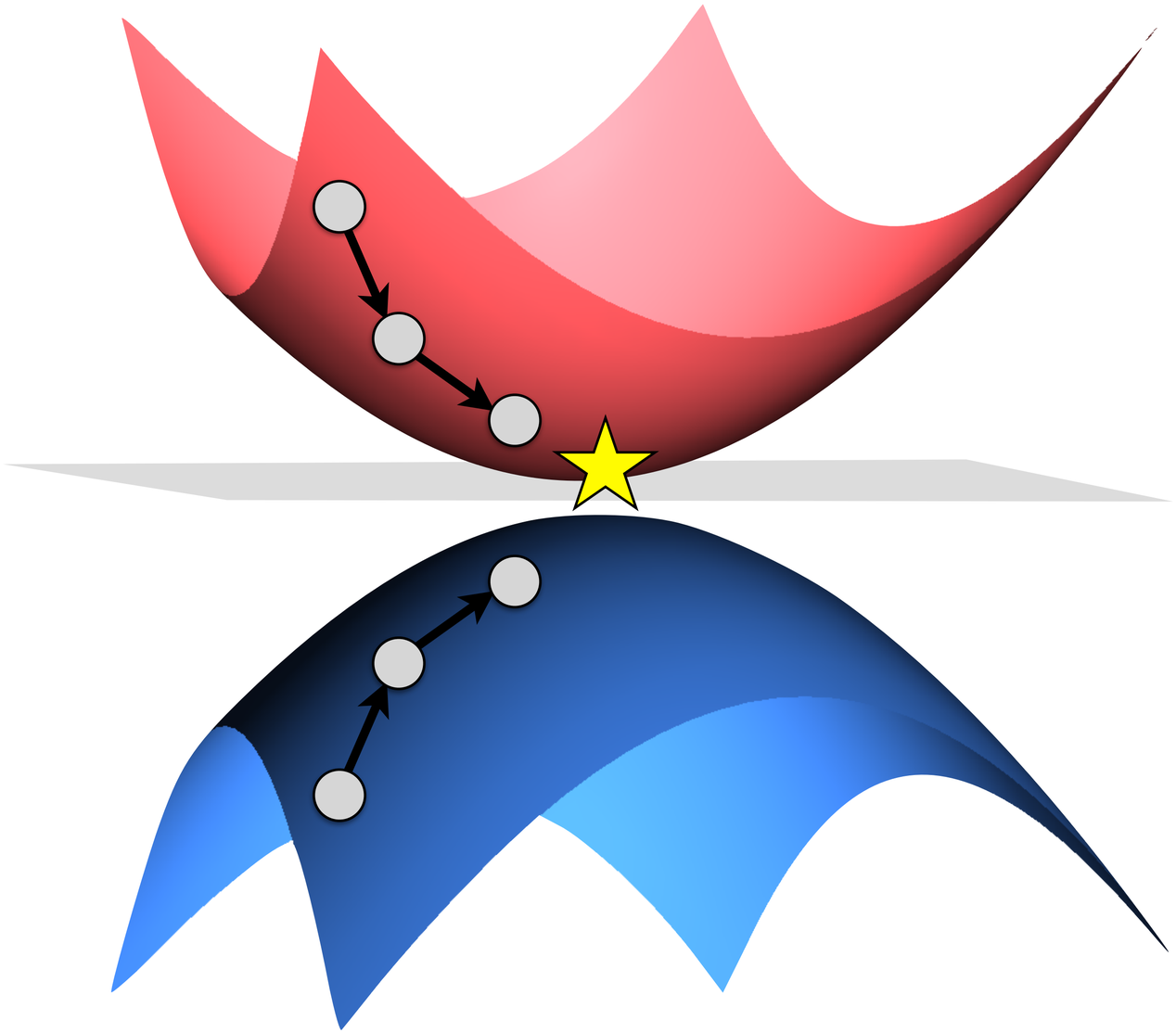
✔ framework can reuse single machine methods

✔ communication is a parameter

✔ easy to implement



HOW TO DEVISE SUBPROBLEMS? KEY IDEA: DUALITY



PRIMAL-DUAL FRAMEWORK

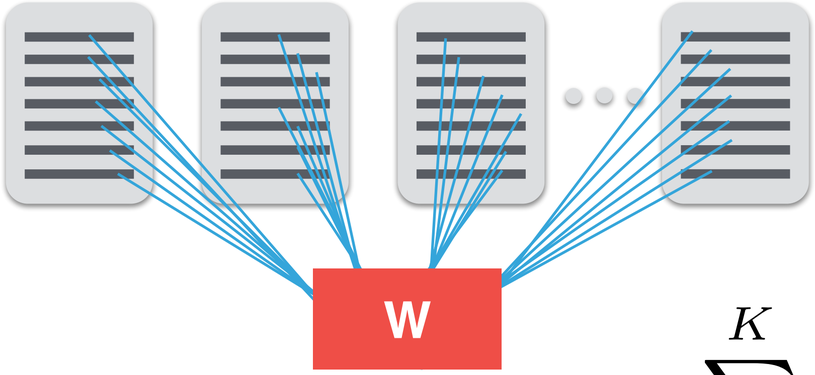
PRIMAL $\geq$ DUAL

$$\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w}^T x_i) + \lambda g(\mathbf{w}) \qquad \max_{\boldsymbol{\alpha}\in\mathbb{R}^n} -\frac{1}{n}\sum_{i=1}^{n} \ell^*(-\alpha_i) - \lambda g^*(X, \boldsymbol{\alpha})$$

- Stopping criteria given by duality gap
- Dual easier to separate across machines (one dual variable per datapoint)

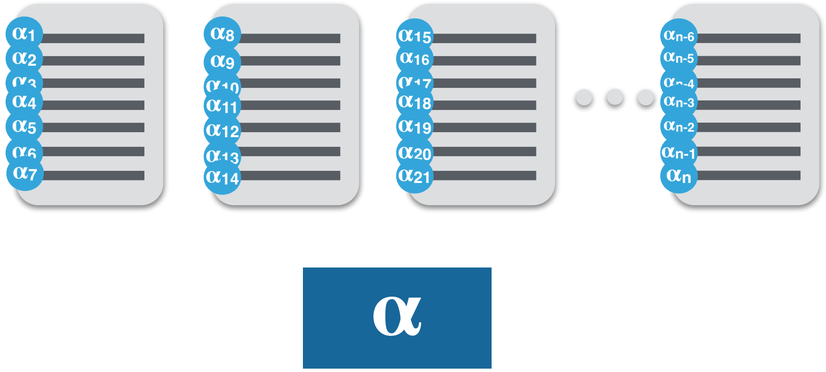
PRIMAL

$$\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w}^T x_i) + \lambda g(\mathbf{w})$$



DUAL

$$\max_{\boldsymbol{\alpha}\in\mathbb{R}^n} -\frac{1}{n}\sum_{i=1}^{n} \ell^*(-\alpha_i) - \lambda g^*(X, \boldsymbol{\alpha}) \quad \sum_{k=1}^{K} \tilde{g}^*(X_{[k]}, \boldsymbol{\alpha}_{[k]})$$



[AT A HIGH LEVEL]

CoCoA Framework

Input: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ distributed over K machines

for $t = 1, 2, \ldots, T$

  for ***machines*** $k = 1, 2, \ldots, K$ ***in parallel***

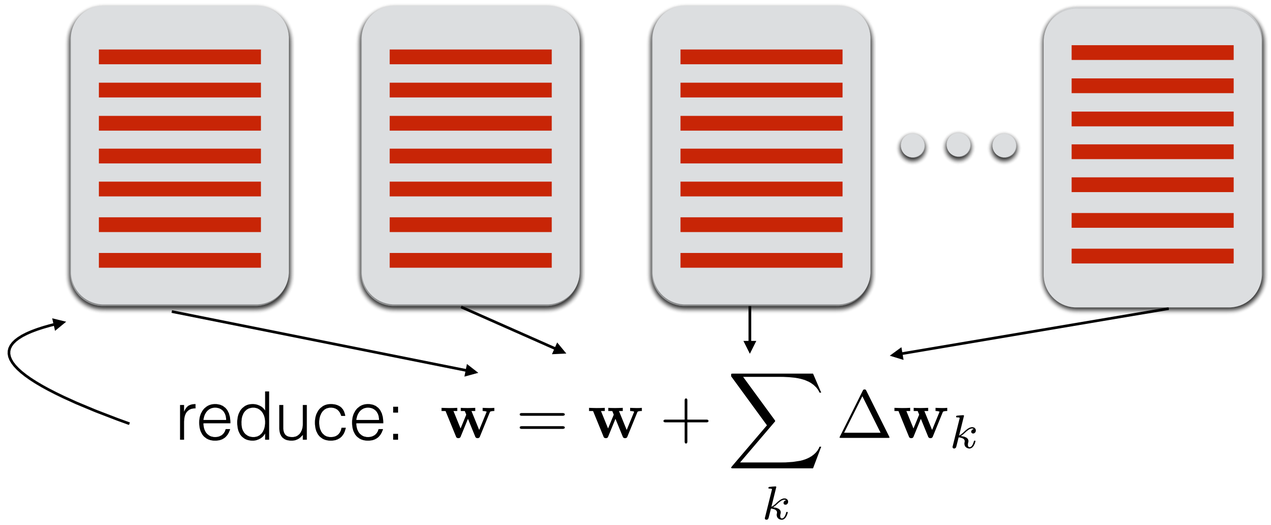
    compute θ-approximate solution to local subproblem

    return $\Delta\mathbf{w}_k$

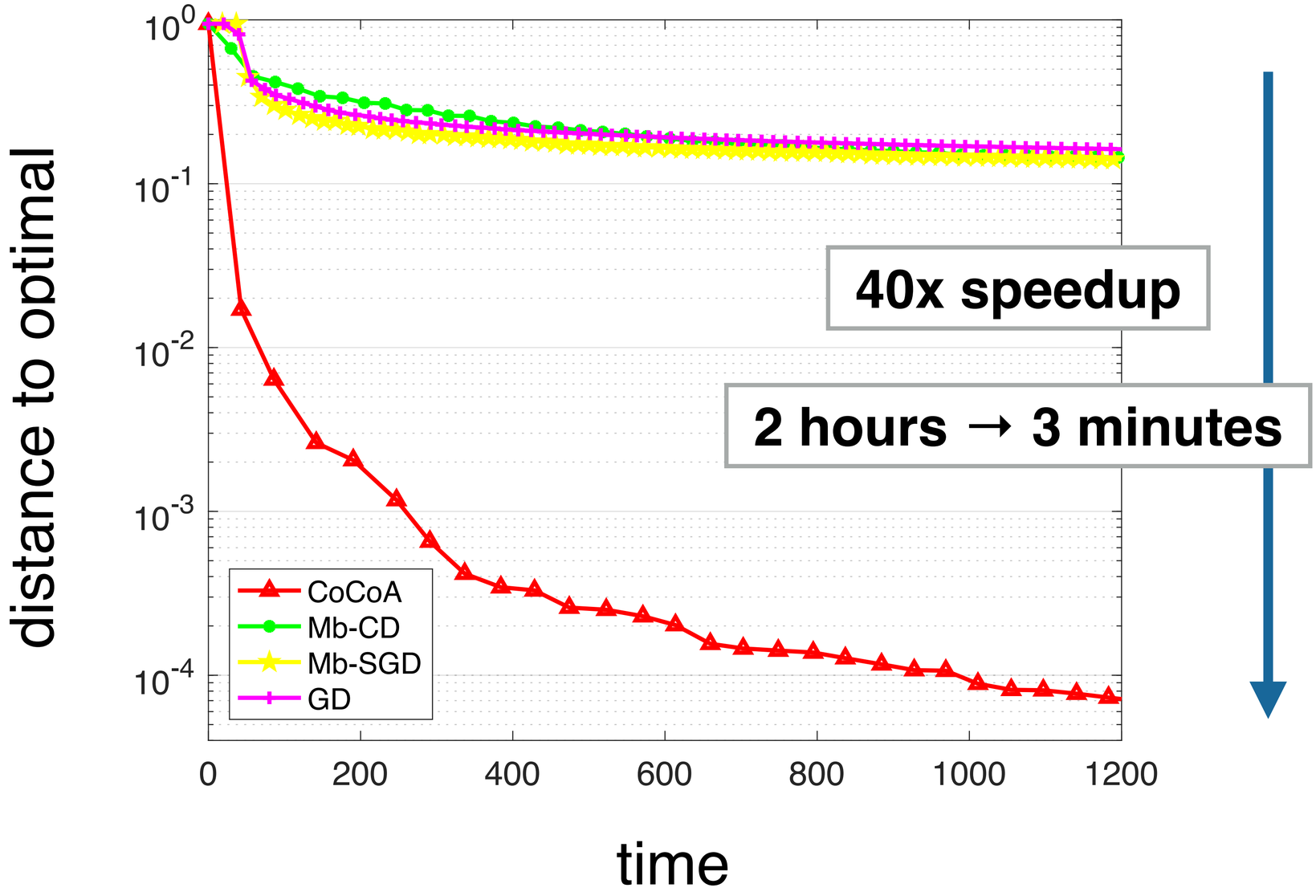
  end

  reduce: $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \sum_{k=1}^K \Delta\mathbf{w}_k$

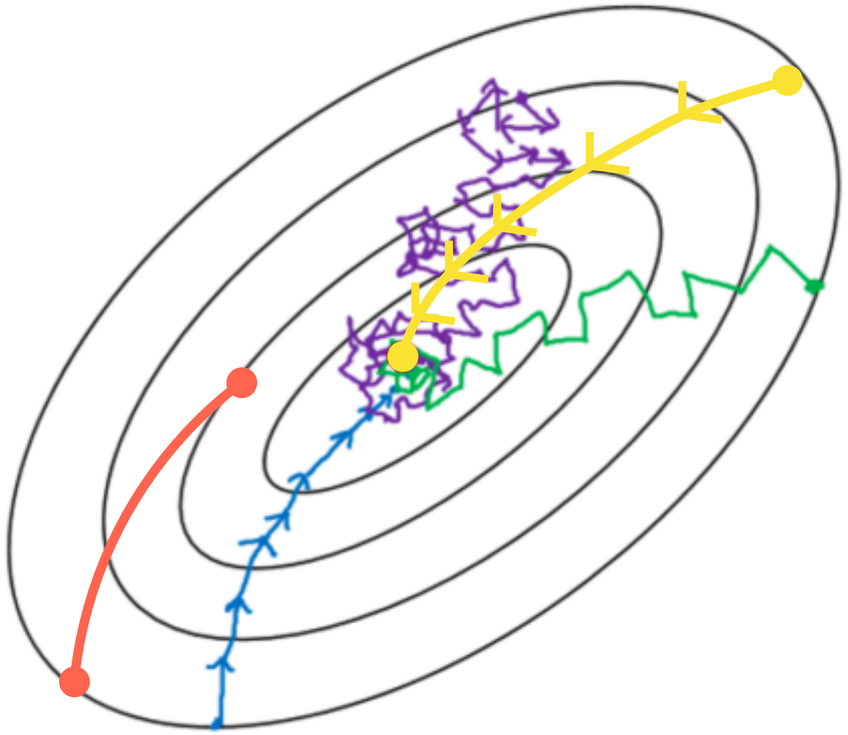
end



EXAMPLE: TEXT DATASET, 16-MACHINE CLUSTER

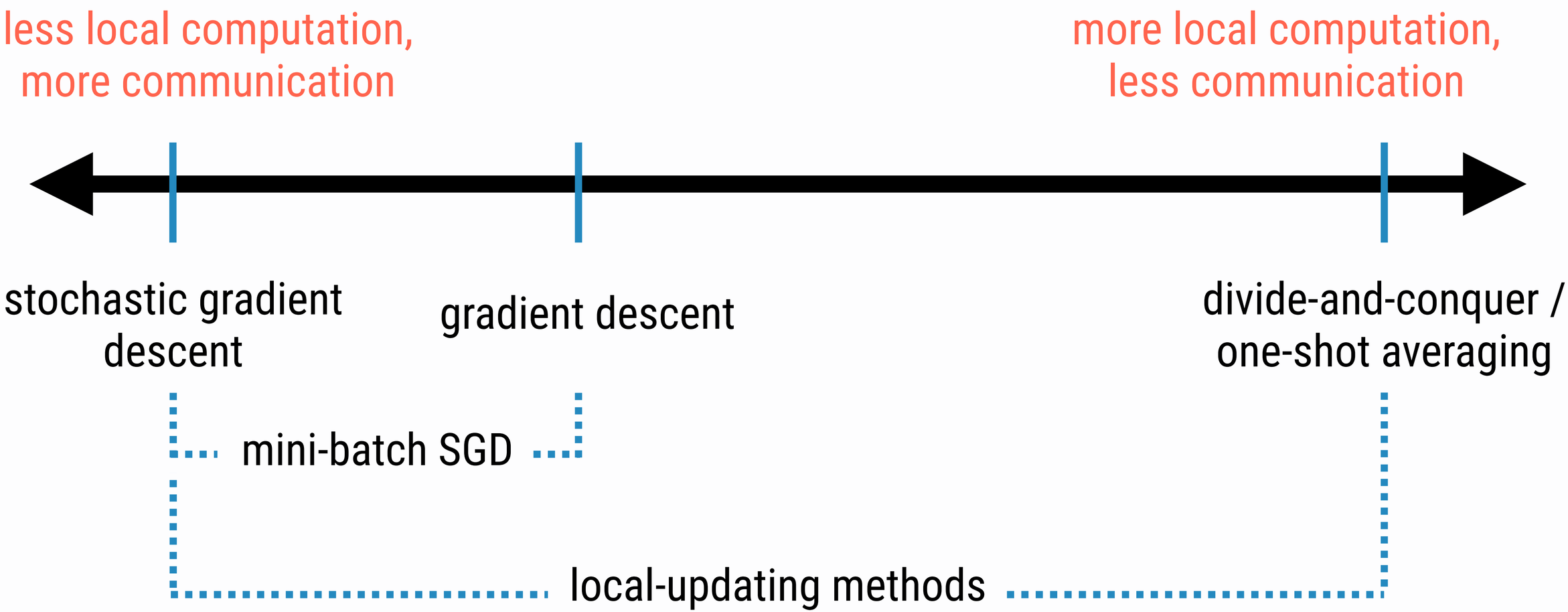


Local-updating methods



Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent
One-shot averaging
*Local-updating methods

Methods for distributed optimization



Local-updating methods

Pros:

- Fully flexible in terms of computation vs. communication
- Allows re-use of local solvers

Cons:

- Another parameter to tune
- Not well-understood for non-convex objectives [current active area of research!]

will talk more about these methods later in the course (federated learning)

Additional Reading

- Optimization for large-scale machine learning: https://leon.bottou.org/publications/pdf/tr-optml-2016.pdf
  - Chapters 3, 4

- CoCoA: http://www.jmlr.org/papers/volume18/16-512/16-512.pdf