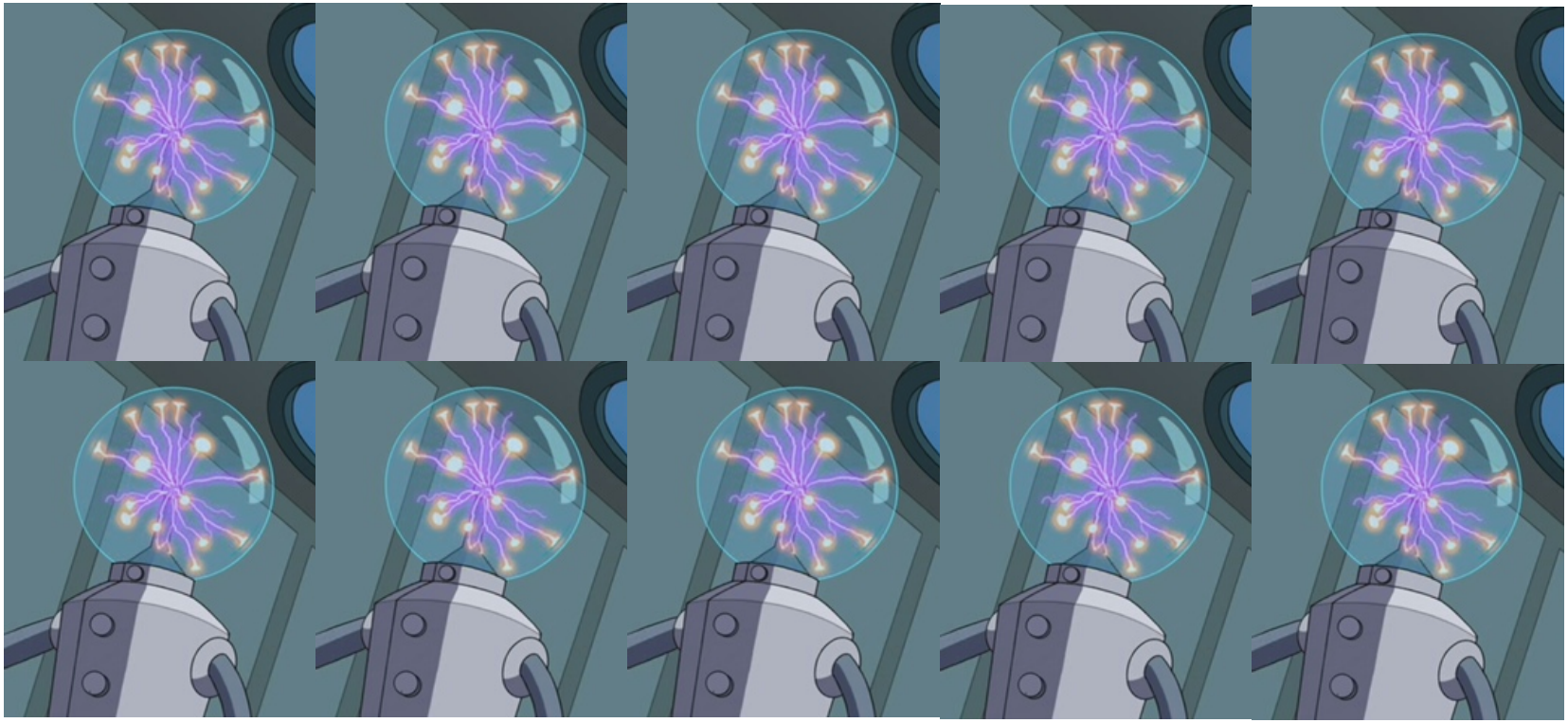
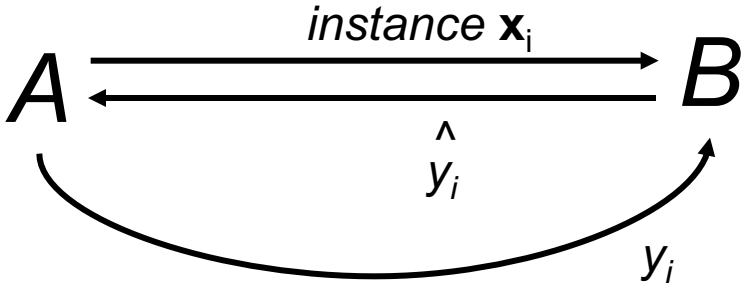
Parallel Perceptrons and Iterative Parameter Mixing



Recap: perceptrons

The perceptron



Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

Margin γ. A must provide examples that can be separated with some vector $\mathbf{u}$ with margin $\gamma > 0$, ie

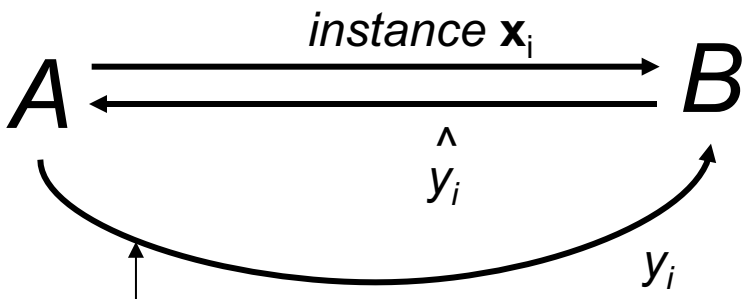
$$\exists \mathbf{u} : \forall (\mathbf{x}_i, y_i) \text{ given by } A, (\mathbf{u} \cdot \mathbf{x}) y_i > \gamma$$

and furthermore, $\|\mathbf{u}\| = 1$.

Radius R. A must provide examples "near the origin", ie

$$\forall \mathbf{x}_i \text{ given by } A, \|\mathbf{x}\|^2 < R^2$$

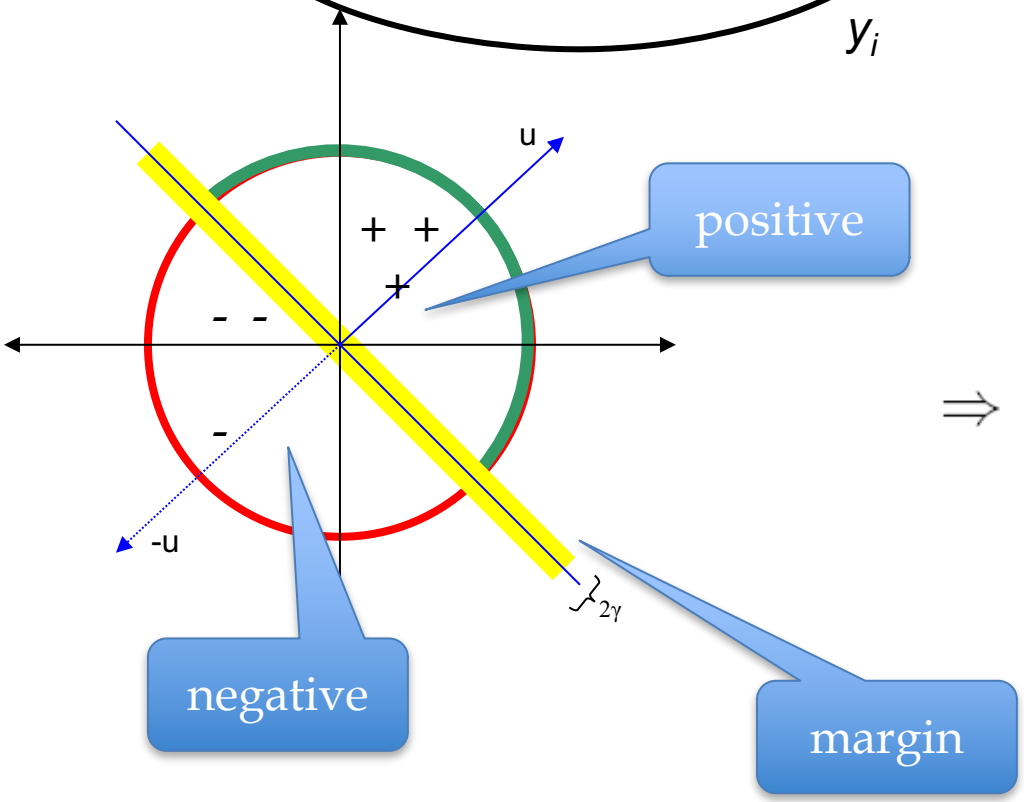
The perceptron



Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

A lot like SGD update for logistic regression!



Mistake bound:

$$\Rightarrow \quad k \leq \frac{R}{\gamma^2}^2 = \left(\frac{R}{\gamma}\right)^2$$

2

$$P(\text{error in x}) = \sum_k P(\text{error on x}|\text{picked } \mathbf{v}_k)P(\text{picked } \mathbf{v}_k)$$

$$= \sum_k \frac{1}{m_k}\frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m}$$

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	√ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	√
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	√
6	$\mathbf{v}_2$	$\mathbf{x}_6$	√
7	$\mathbf{v}_2$	$\mathbf{x}_7$	√
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	√
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

$m_1=3$

$m_2=4$

$m=10$

1. Pick a $\mathbf{v}_k$ at random according to m_k/m, the fraction of examples it was used for.
2. Predict using the $\mathbf{v}_k$ you just picked.
3. (Actually, use some sort of deterministic approximation to this).

predict using sign(v*. **x**)

$$\mathbf{v}_* = \sum_k \left(\frac{m_k}{m}\mathbf{v}_k\right)$$

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	√ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	√
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	√
6	$\mathbf{v}_2$	$\mathbf{x}_6$	√
7	$\mathbf{v}_2$	$\mathbf{x}_7$	√
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	√
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

m_1=3

m_2=4

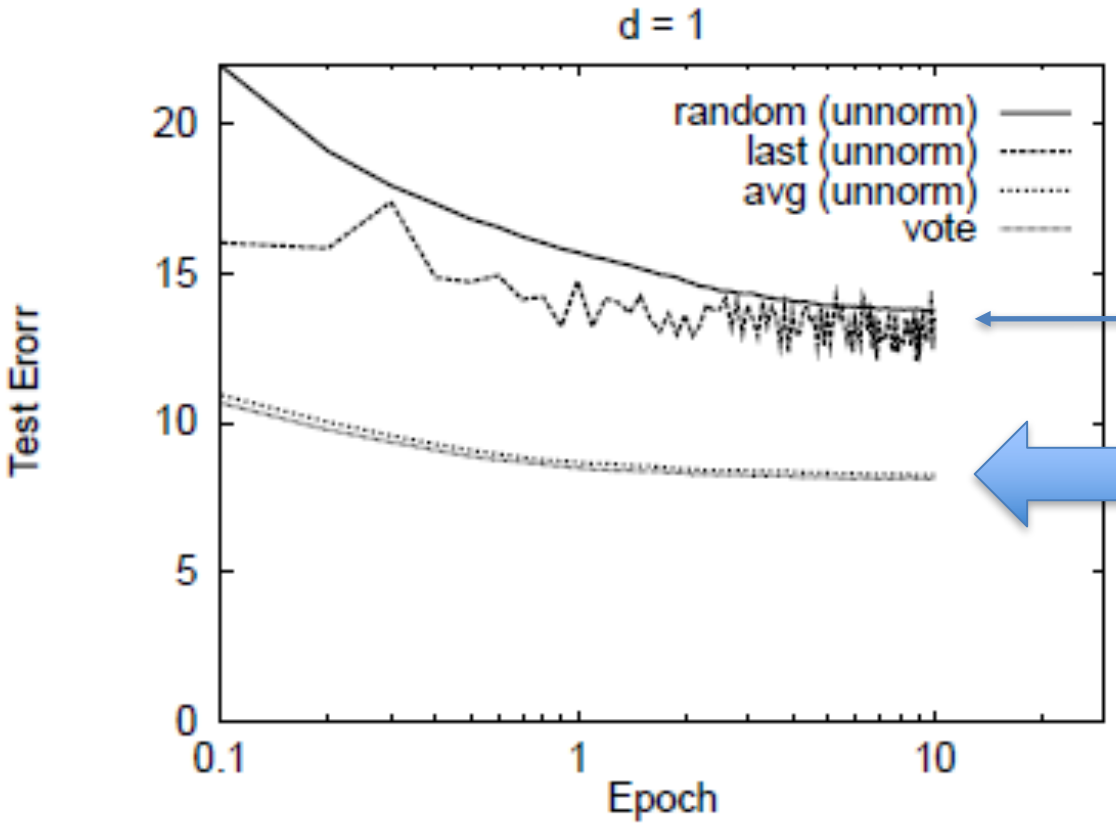
m=10

1. Pick a $\mathbf{v}_k$ at random according to m_k/m, the fraction of examples it was used for.
2. Predict using the $\mathbf{v}_k$ you just picked.
3. (Actually, use some sort of deterministic approximation to this).

predict using sign(v*. **x**)

$$\mathbf{v}_* = \sum_k \left(\frac{m_k}{m}\mathbf{v}_k\right)$$

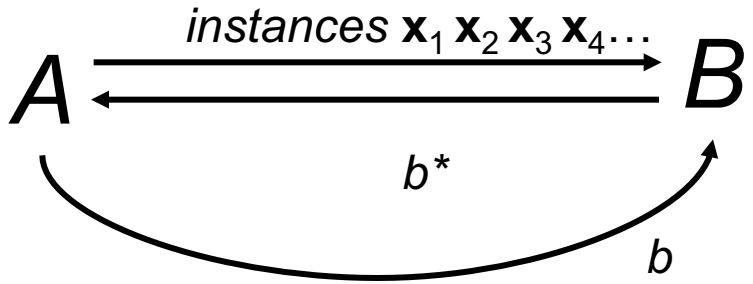
Also: there's a sparsification trick that makes learning the averaged perceptron fast



Last perceptron

Averaging/voting

The voted perceptron *for ranking*



Compute: $y_i = \hat{\mathbf{v}}_k \cdot \mathbf{x}_i$
Return: the index b^* of the "best" $\mathbf{x}_i$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_b - \mathbf{x}_{b^*}$

Margin γ. *A* must provide examples that can be correctly ranked with some vector $\mathbf{u}$ with margin $\gamma > 0$, ie

$$\exists \mathbf{u} : \forall \mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,n_i}, \ell \text{ given by } A, \forall j \neq \ell,\ \mathbf{u} \cdot \mathbf{x}_\ell - \mathbf{u} \cdot \mathbf{x}_j > \gamma$$

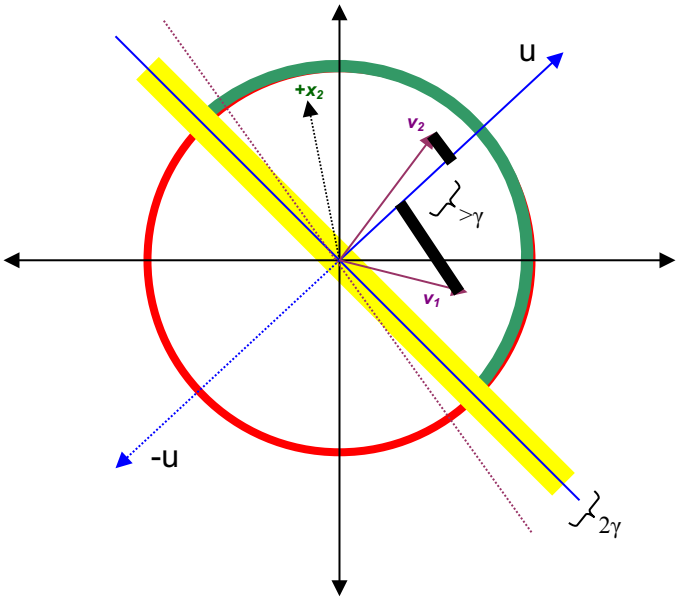
and furthermore, $\|\mathbf{u}\|^2 = 1$.

Radius *R*. *A* must provide examples "near the origin", ie

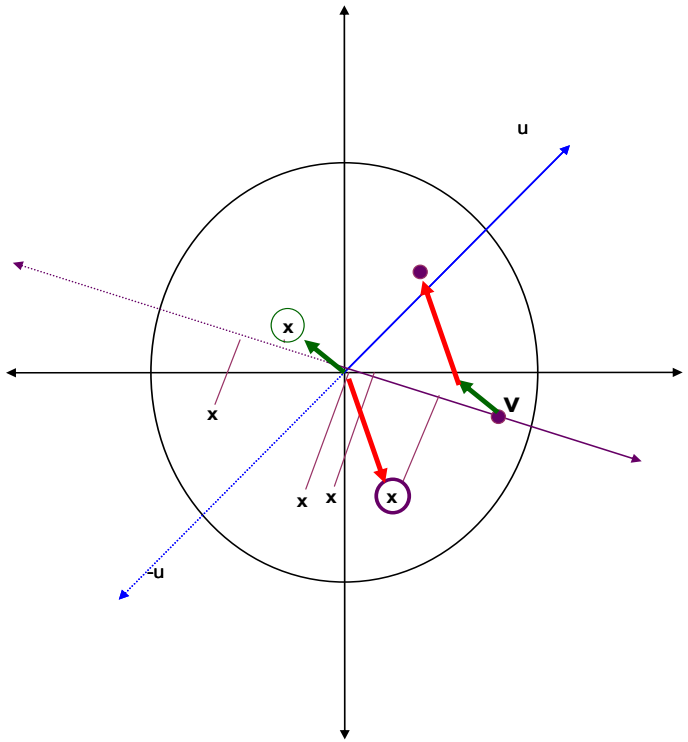
$$\forall \mathbf{x}_i \text{ given by } A,\ \|\mathbf{x}\|^2 < R^2$$

The voted perceptron *for ranking*

Normal perceptron



Ranking perceptron



Ranking perceptrons ➔ structured perceptrons

- Ranking API:
  - A sends B a (maybe **huge**) set of items to rank
  - B finds the single **best** one according to the current weight vector
  - A tells B which one was actually best
- Structured classification API:
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

Ranking perceptrons → structured perceptrons

- Structured → ranking API:
  - A sends B the word sequence **x**
  - B finds the single **best y** according to the current weight vector (using dynamic programming)
  - A tells B which **y** was actually best
  - This is equivalent to ranking pairs g=(**x**,**y'**)

- Structured classification on a sequence
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

But implements structured classification's API

Ranking perceptrons → structured perceptrons

- Structured → ranking API:
  - A sends B the word sequence **x**
  - B finds the single **best y** according to the current weight vector (using dynamic programming)
  - A tells B which **y** was actually best
  - This is equivalent to ranking pairs g=(**x**,**y'**)
- Structured classification on a sequence
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

Distributed Training Strategies for the Structured Perceptron

Ryan McDonald **Keith Hall** **Gideon Mann**
Google, Inc., New York / Zurich
{ryanmcd|kbhall|gmann}@google.com

NAACL 2010



Parallel Structured Perceptrons

- Simplest idea:
  - Split data into S "shards"
  - Train a perceptron on each shard independently
    - weight vectors are $\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, …
  - Produce a weighted average of the $\mathbf{w}^{(i)}$'s as the final result

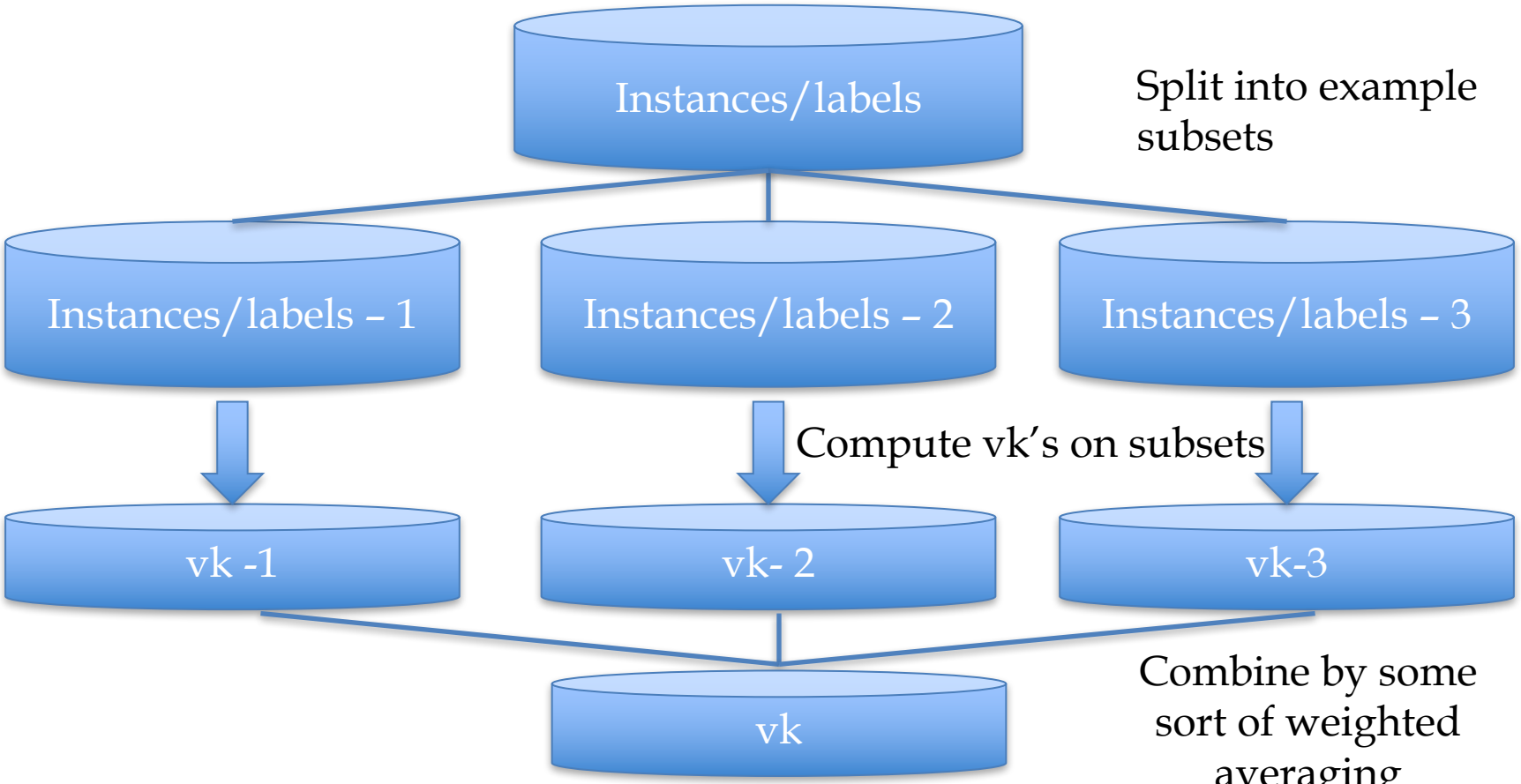
aka parameter mixture

PerceptronParamMix($\mathcal{T} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|\mathcal{T}|}$)

1. Shard $\mathcal{T}$ into S pieces $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_S\}$
2. $\mathbf{w}^{(i)} = \text{Perceptron}(\mathcal{T}_i)$ †
3. $\mathbf{w} = \sum_i \mu_i \mathbf{w}^{(i)}$ ‡
4. return $\mathbf{w}$

Figure 2: Distributed perceptron using a parameter mixing strategy. † Each $\mathbf{w}^{(i)}$ is computed in parallel. ‡ $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_S\}$, $\forall \mu_i \in \boldsymbol{\mu} : \mu_i \geq 0$ and $\sum_i \mu_i = 1$.

Parallelizing perceptrons



Parallel Perceptrons

- Simplest idea:
  - Split data into S "shards"
  - Train a perceptron on each shard independently
    - weight vectors are $\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, …
  - Produce some weighted average of the $\mathbf{w}^{(i)}$'s as the final result

- Theorem: this doesn't always work.
- Proof: by constructing an example where you can converge on every shard, and still have the averaged vector *not* separate the full training set – no matter *how* you average the components.

PerceptronParamMix($\mathcal{T} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|\mathcal{T}|}$)

1. Shard $\mathcal{T}$ into S pieces $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_S\}$
2. $\mathbf{w}^{(i)} = \text{Perceptron}(\mathcal{T}_i)$ †
3. $\mathbf{w} = \sum_i \mu_i \mathbf{w}^{(i)}$ ‡
4. return $\mathbf{w}$

Figure 2: Distributed perceptron using a parameter mixing strategy. † Each $\mathbf{w}^{(i)}$ is computed in parallel. ‡ $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_S\}$, $\forall \mu_i \in \boldsymbol{\mu} : \mu_i \geq 0$ and $\sum_i \mu_i = 1$.

Parallel Perceptrons – take 2

PerceptronIterParamMix($\mathcal{T} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|\mathcal{T}|}$)
1. Shard $\mathcal{T}$ into S pieces $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_S\}$
2. $\mathbf{w} = \mathbf{0}$
3. for $n : 1..N$
4. $\quad \mathbf{w}^{(i,n)} = \text{OneEpochPerceptron}(\mathcal{T}_i, \mathbf{w})$ †
5. $\quad \mathbf{w} = \sum_i \mu_{i,n} \mathbf{w}^{(i,n)}$ ‡
6. return $\mathbf{w}$

OneEpochPerceptron($\mathcal{T}$, $\mathbf{w}^*$)
1. $\mathbf{w}^{(0)} = \mathbf{w}^*$; $k = 0$
2. for $t : 1..T$
3. $\quad$ Let $\mathbf{y}' = \arg\max_{\mathbf{y}'} \mathbf{w}^{(k)} \cdot \mathbf{f}(\mathbf{x}_t, \mathbf{y}')$
4. $\quad$ if $\mathbf{y}' \neq \mathbf{y}_t$
5. $\quad\quad \mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{f}(\mathbf{x}_t, \mathbf{y}_t) - \mathbf{f}(\mathbf{x}_t, \mathbf{y}')$
6. $\quad\quad k = k + 1$
7. return $\mathbf{w}^{(k)}$

Figure 3: Distributed perceptron using an iterative parameter mixing strategy. † Each $\mathbf{w}^{(i,n)}$ is computed in parallel. ‡ $\boldsymbol{\mu}_n = \{\mu_{1,n}, \ldots, \mu_{S,n}\}$, $\forall \mu_{i,n} \in \boldsymbol{\mu}_n$: $\mu_{i,n} \geq 0$ and $\forall n$: $\sum_i \mu_{i,n} = 1$.

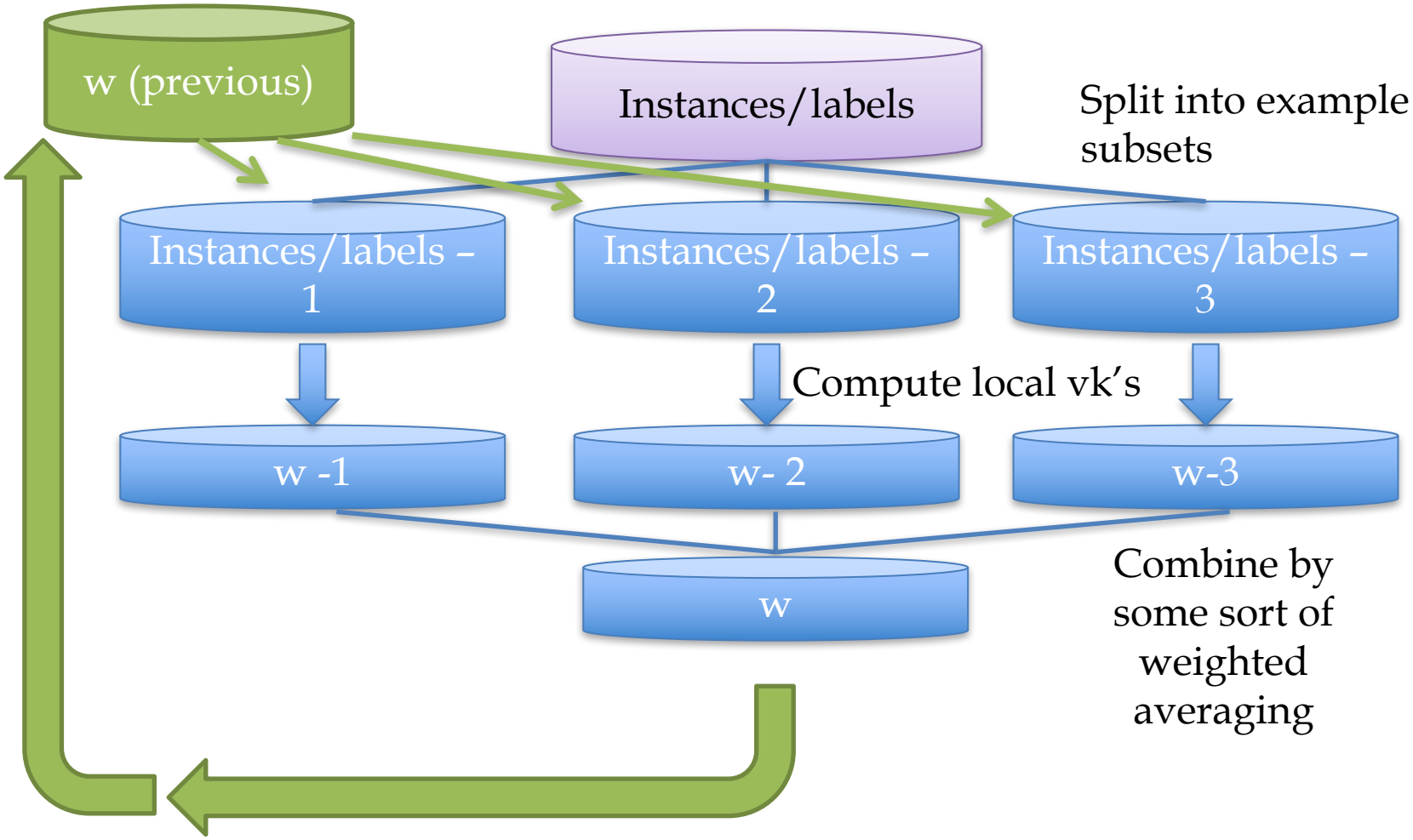
Idea: do the simplest possible thing iteratively.

- Split the data into shards
- Let **w = 0**
- For n=1,…
  - Train a perceptron on each shard with one pass *starting with* **w**
  - Average the weight vectors (somehow) and let **w** be that average

All-Reduce

Extra communication cost:
- redistributing the weight vectors
- done less frequently than if fully synchronized, more frequently than if fully parallelized

Parallelizing perceptrons – take 2



A theorem

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

$$\sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n} \leq \frac{R^2}{\gamma^2}$$

Corollary: if we weight the vectors uniformly, then the number of mistakes is still bounded.

I.e., this is "enough communication" to guarantee convergence.

What we know and don't know

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

uniform mixing…μ=1/S

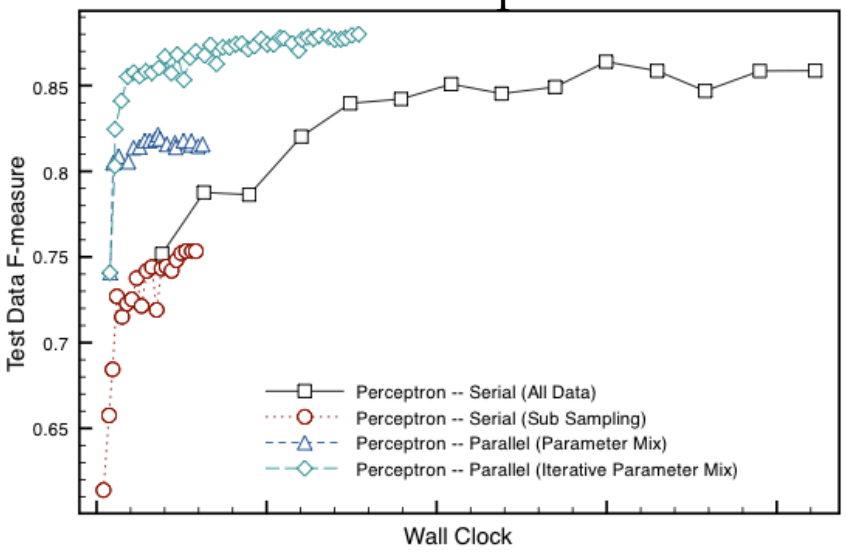
$$\sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n} \leq \frac{R^2}{\gamma^2} \quad \Longrightarrow \quad \sum_{n=1}^{N}\sum_{i=1}^{S}k_{i,n} \leq S \times \frac{R^2}{\gamma^2}$$

could we lose our speedup-from-parallelizing to slower convergence?

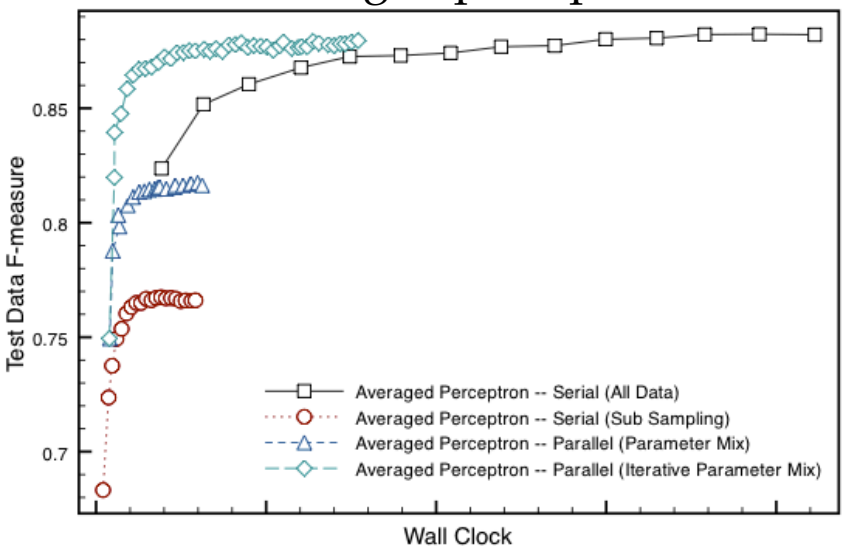
speedup by factor of S is cancelled by slower convergence by factor of S

Results on NER

Perceptron



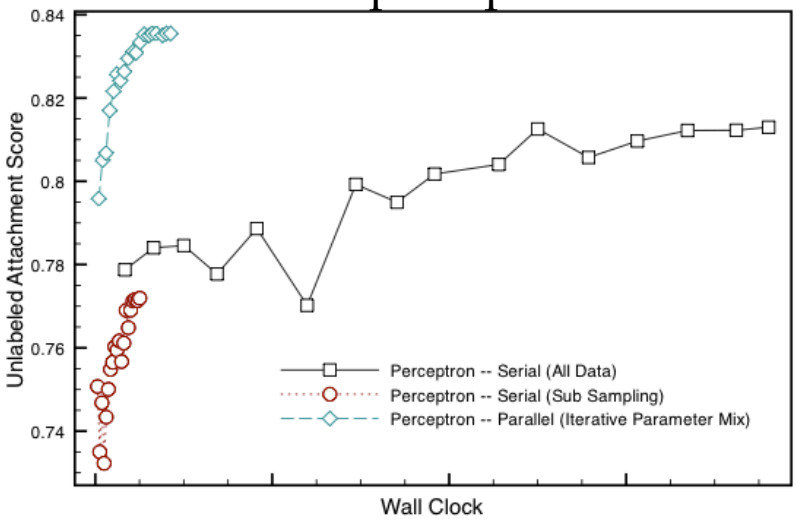
Averaged perceptron



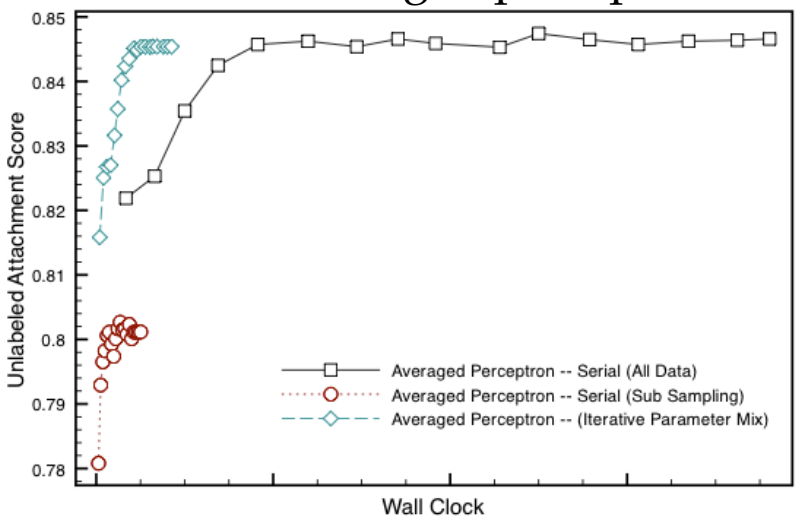
	Reg. Perceptron F-measure	Avg. Perceptron F-measure
Serial (All Data)	85.8	88.2
Serial (Sub Sampling)	75.3	76.6
Parallel (Parameter Mix)	81.5	81.6
Parallel (Iterative Parameter Mix)	87.9	88.1

Results on parsing

perceptron



Averaged perceptron



	Reg. Perceptron Unlabeled Attachment Score	Avg. Perceptron Unlabeled Attachment Score
Serial (All Data)	81.3	84.7
Serial (Sub Sampling)	77.2	80.1
Parallel (Iterative Parameter Mix)	83.5	84.5

The theorem…

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

$$\sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n} \leq \frac{R^2}{\gamma^2}$$

$$\mathbf{w}^{(\text{avg},n)} = \sum_{i=1}^{S}\mu_{i,n}\mathbf{w}^{(i,n)}$$

$$\begin{aligned}
\mathbf{u}\cdot\mathbf{w}^{(i,n)} &= \mathbf{u}\cdot\mathbf{w}^{([i,n]-1)} \\
&\quad + \mathbf{u}\cdot(\mathbf{f}(\mathbf{x}_t,\mathbf{y}_t) - \mathbf{f}(\mathbf{x}_t,\mathbf{y}')) \\
&\geq \mathbf{u}\cdot\mathbf{w}^{([i,n]-1)} + \gamma \\
&\geq \mathbf{u}\cdot\mathbf{w}^{([i,n]-2)} + 2\gamma \\
\ldots &\geq \mathbf{u}\cdot\mathbf{w}^{(\text{avg},n-1)} + k_{i,n}\gamma \quad \text{(A1)}
\end{aligned}$$

The theorem…

Perceptron($\mathcal{T} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|\mathcal{T}|}$)

1. $\mathbf{w}^{(0)} = \mathbf{0}$; $k = 0$
2. for $n : 1..N$
3. for $t : 1..T$
4. Let $\mathbf{y}' = \arg\max_{\mathbf{y}'} \mathbf{w}^{(k)} \cdot \mathbf{f}(\mathbf{x}_t, \mathbf{y}')$
5. if $\mathbf{y}' \neq \mathbf{y}_t$
6. $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{f}(\mathbf{x}_t, \mathbf{y}_t) - \mathbf{f}(\mathbf{x}_t, \mathbf{y}')$
7. $k = k + 1$
8. return $\mathbf{w}^{(k)}$

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

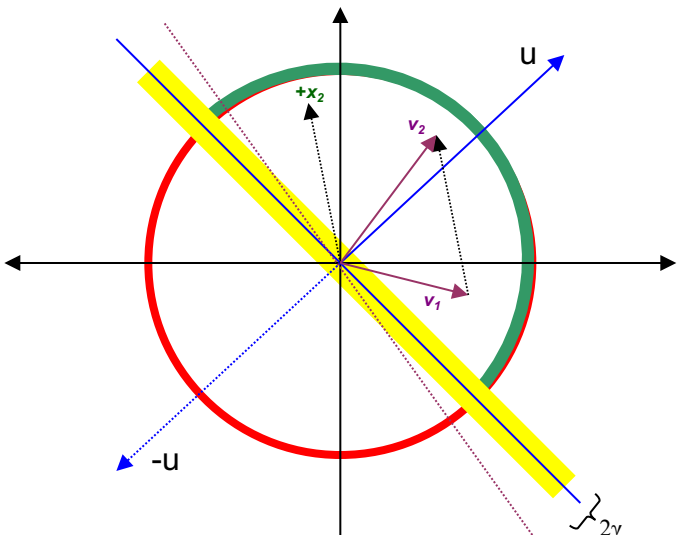
$$\sum_{n=1}^{N}\sum_{i=1}^{S} \mu_{i,n} k_{i,n} \leq \frac{R^2}{\gamma^2}$$

$$\mathbf{w}^{(\text{avg},n)} = \sum_{i=1}^{S} \mu_{i,n} \mathbf{w}^{(i,n)}$$

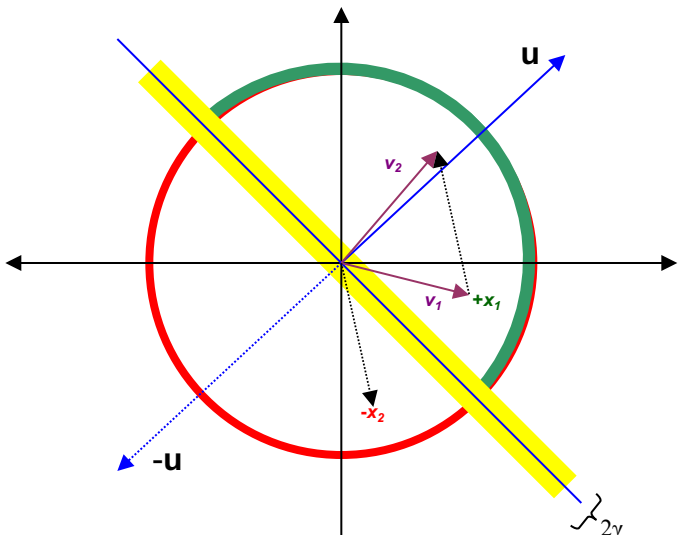
$$\begin{aligned}
\|\mathbf{w}^{(i,n)}\|^2 &= \|\mathbf{w}^{([i,n]-1)}\|^2 \\
&\quad + \|\mathbf{f}(\mathbf{x}_t, \mathbf{y}_t) - \mathbf{f}(\mathbf{x}_t, \mathbf{y}')\|^2 \\
&\quad + 2\mathbf{w}^{([i,n]-1)}(\mathbf{f}(\mathbf{x}_t, \mathbf{y}_t) - \mathbf{f}(\mathbf{x}_t, \mathbf{y}')) \\
&\leq \|\mathbf{w}^{([i,n]-1)}\|^2 + R^2 \\
&\leq \|\mathbf{w}^{([i,n]-2)}\|^2 + 2R^2 \\
\ldots &\leq \|\mathbf{w}^{(\text{avg},n-1)}\|^2 + k_{i,n}R^2 \quad \text{(A2)}
\end{aligned}$$

This is not new….

(3a) The guess $\mathbf{v}_2$ after the two positive examples: $\mathbf{v}_2=\mathbf{v}_1+\mathbf{x}_2$



(3b) The guess $\mathbf{v}_2$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$



Lemma 2 *$\forall k$, $\|\mathbf{v}_k\|^2 \leq kR$. In other words, the norm of $\mathbf{v}_k$ grows "slowly", at a rate depending on R.*

Proof:

$$\mathbf{v}_{k+1} \cdot \mathbf{v}_{k+1} = (\mathbf{v}_k + y_i\mathbf{x}_i) \cdot (\mathbf{v}_k + y_i\mathbf{x}_i)$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_{k+1}\|^2 + 2y_i\mathbf{x}_i \cdot \mathbf{v}_k + y_i^2\|\mathbf{x}\|^2$$

If mistake: $y_i\,\mathbf{x}_i\,\mathbf{v}_k < 0$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_{k+1}\|^2 + [\text{something negative}] + 1\|\mathbf{x}\|^2$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_{k+1}\|^2 + \|\mathbf{x}\|^2$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_{k+1}\|^2 + R^2 \qquad \forall \mathbf{x}_i \text{ given by } A,\ \|\mathbf{x}\|^2 < R^2$$

$$\Rightarrow \quad \|\mathbf{v}_k\|^2 \leq kR^2$$

Using A1/A2 we prove two inductive hypotheses:

$$\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)} \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \qquad \text{(IH1)}$$

$$\|\mathbf{w}^{(\text{avg},N)}\|^2 \leq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2 \qquad \text{(IH2)}$$

The base case is $\mathbf{w}^{(\text{avg},1)}$, where we can observe:

$$\mathbf{u} \cdot \mathbf{w}^{\text{avg},1} = \sum_{i=1}^{S} \mu_{i,1} \mathbf{u} \cdot \mathbf{w}^{(i,1)} \geq \sum_{i=1}^{S} \mu_{i,1} k_{i,1} \gamma$$

Follows from: $\boldsymbol{u.w}^{(i,1)} >= k_{1,i}\gamma$

This is new …. We've never considered averaging operations before

IH1 inductive case:

$$\mathbf{w}^{(\text{avg},n)} = \sum_{i=1}^{S} \mu_{i,n} \mathbf{w}^{(i,n)}$$ IH1

$$\begin{aligned}
\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)} &= \sum_{i=1}^{S} \mu_{i,N} \left(\mathbf{u} \cdot \mathbf{w}^{(i,N)}\right) \\
&\geq \sum_{i=1}^{S} \mu_{i,N} \left(\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N-1)} + k_{i,N}\gamma\right) \qquad \text{From A1} \\
&= \cancel{\textstyle\sum_i \mu_{i,N}}\, \mathbf{u} \cdot \mathbf{w}^{(\text{avg},N-1)} + \sum_{i=1}^{S} \mu_{i,N} k_{i,N}\gamma \qquad \text{Distribute} \\
&\qquad \text{μ's are } \textit{distribution} \\
&\geq \left[\sum_{n=1}^{N-1} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\gamma\right] + \sum_{i=1}^{S} \mu_{i,N} k_{i,N}\gamma \\
&= \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\gamma
\end{aligned}$$

$$\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)} \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\gamma \qquad \text{(IH1)}$$

The first inequality uses A1, the second step $\sum_i \mu_{i,N} = 1$ and the second inequality the inductive hypothesis IH1.

Using A1/A2 we prove two inductive hypotheses:

$$\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)} \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \qquad \text{(IH1)}$$

$$\|\mathbf{w}^{(\text{avg},N)}\|^2 \leq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2 \qquad \text{(IH2)}$$

IH2 proof is similar

IH1 implies $\|\mathbf{w}^{(\text{avg},N)}\| \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma$
since $\mathbf{u} \cdot \mathbf{w} \leq \|\mathbf{u}\| \|\mathbf{w}\|$ and $\|\mathbf{u}\| = 1$.

IH1, IH2 together imply the bound (as in the usual perceptron case)

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

$$\sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \leq \frac{R^2}{\gamma^2}$$

What we know and don't know

Theorem 3. *Assume a training set $\mathcal{T}$ is separable by margin γ. Let $k_{i,n}$ be the number of mistakes that occurred on shard i during the nth epoch of training. For any N, when training the perceptron with iterative parameter mixing (Figure 3),*

uniform mixing…

$$\sum_{n=1}^{N}\sum_{i=1}^{S}\mu_{i,n}k_{i,n} \leq \frac{R^2}{\gamma^2} \quad \Longrightarrow \quad \sum_{n=1}^{N}\sum_{i=1}^{S}k_{i,n} \leq S \times \frac{R^2}{\gamma^2}$$

could we lose our speedup-from-parallelizing to slower convergence?

Formally – yes; the algorithm converges but could be S times slower
Experimentally – no

How robust are those experiments?

What we know and don't know

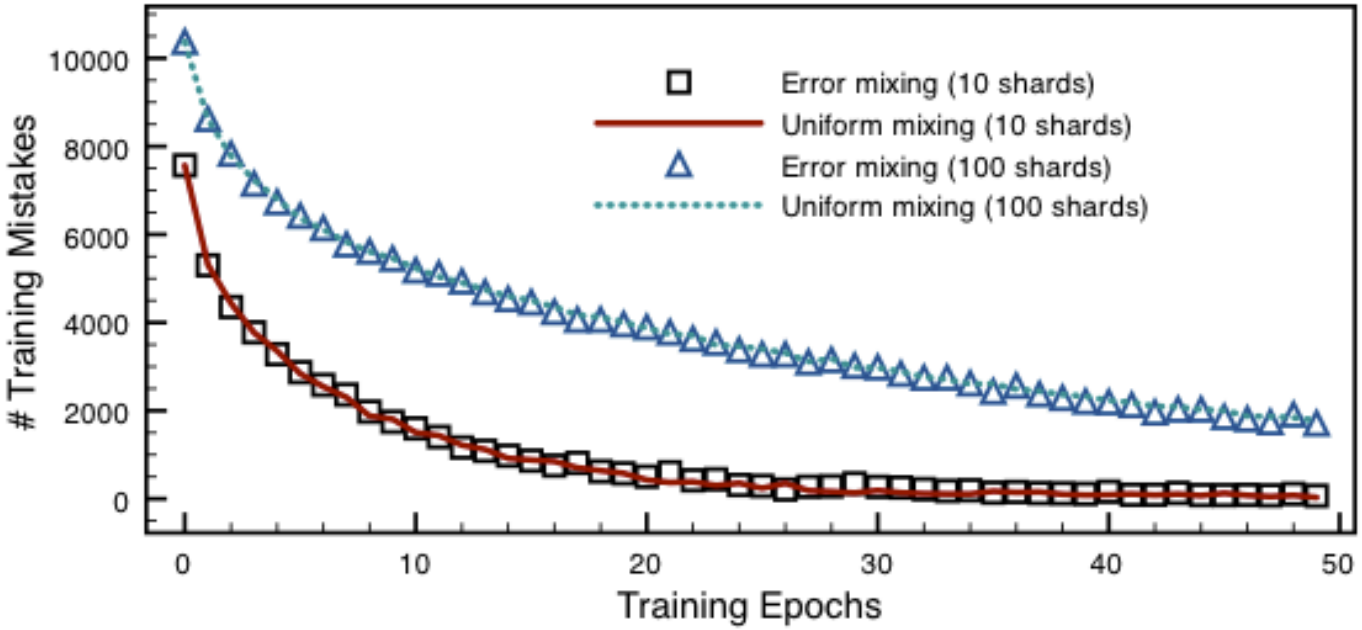


Figure 6: Training errors per epoch for different shard size and parameter mixing strategies.

Is there a tipping point, where cost of parallelizing outweighs benefits?

What we know and don't know

Thus, for cases where training errors are uniformly distributed across shards, it is possible that, in the worst-case, convergence may slow proportional the the number of shards. This implies a trade-off between slower convergence and quicker epochs when selecting a large number of shards. In fact, we observed a tipping point for our experiments in which increasing the number of shards began to have an adverse effect on training times, which for the named-entity experiments occurred around 25-50 shards. This is both due to reasons described in this section as well as the added overhead of maintaining and summing multiple high-dimensional weight vectors after each distributed epoch.

What we know and don't know

In this paper we have investigated distributing the structured perceptron via simple parameter mixing strategies. Our analysis shows that an iterative parameter mixing strategy is both guaranteed to separate the data (if possible) and significantly reduces the time required to train high accuracy classifiers. However, there is a trade-off between increasing training times through distributed computation and slower convergence relative to the number of shards.

Followup points

- On beyond perceptron-based analysis
  - delayed SGD, the theory
- All-Reduce in Map-Reduce
- How bad is Take 1 of this paper?

Regret analysis for on-line optimization

Slow Learners are Fast

John Langford JL@HUNCH.NET
Alexander J. Smola ALEX@SMOLA.ORG
Martin Zinkevich MAZ@YAHOO-INC.COM
Yahoo! Labs, Great America Parkway, Santa Clara, CA 95051 USA

2009

Algorithm 1 Delayed Stochastic Gradient Descent

Input: Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule η_t and delay $\tau \in \mathbb{N}$
Initialization: set $x_1 \ldots, x_\tau = 0$ and compute corresponding $g_t = \nabla f_t(x_t)$.
for $t = \tau + 1$ **to** $T + \tau$ **do**
  Obtain f_t and incur loss $f_t(x_t)$
  Compute $g_t := \nabla f_t(x_t)$
  Update $x_{t+1} = \text{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)
end for

f is loss function, x is parameters

1. Take a gradient step: $x' = x_t - \eta_t g_t$
2. If you've restricted the parameters to a subspace X (e.g., must be positive, …) find the closest thing in X to x': $x_{t+1} = \text{argmin}_X \; dist(x - x')$
3. But…. you might be using a "stale" g (from τ steps ago)

Algorithm 1 Delayed Stochastic Gradient Desce

Input: Feasible space $X \subseteq \mathbb{R}^n$, annealing sched … elay $\tau \in \mathbb{N}$

Initialization: set $x_1 \ldots, x_\tau = 0$ and compute c… ng $g_t = \nabla f_t(x_t)$.

for $t = \tau + 1$ **to** $T + \tau$ **do**

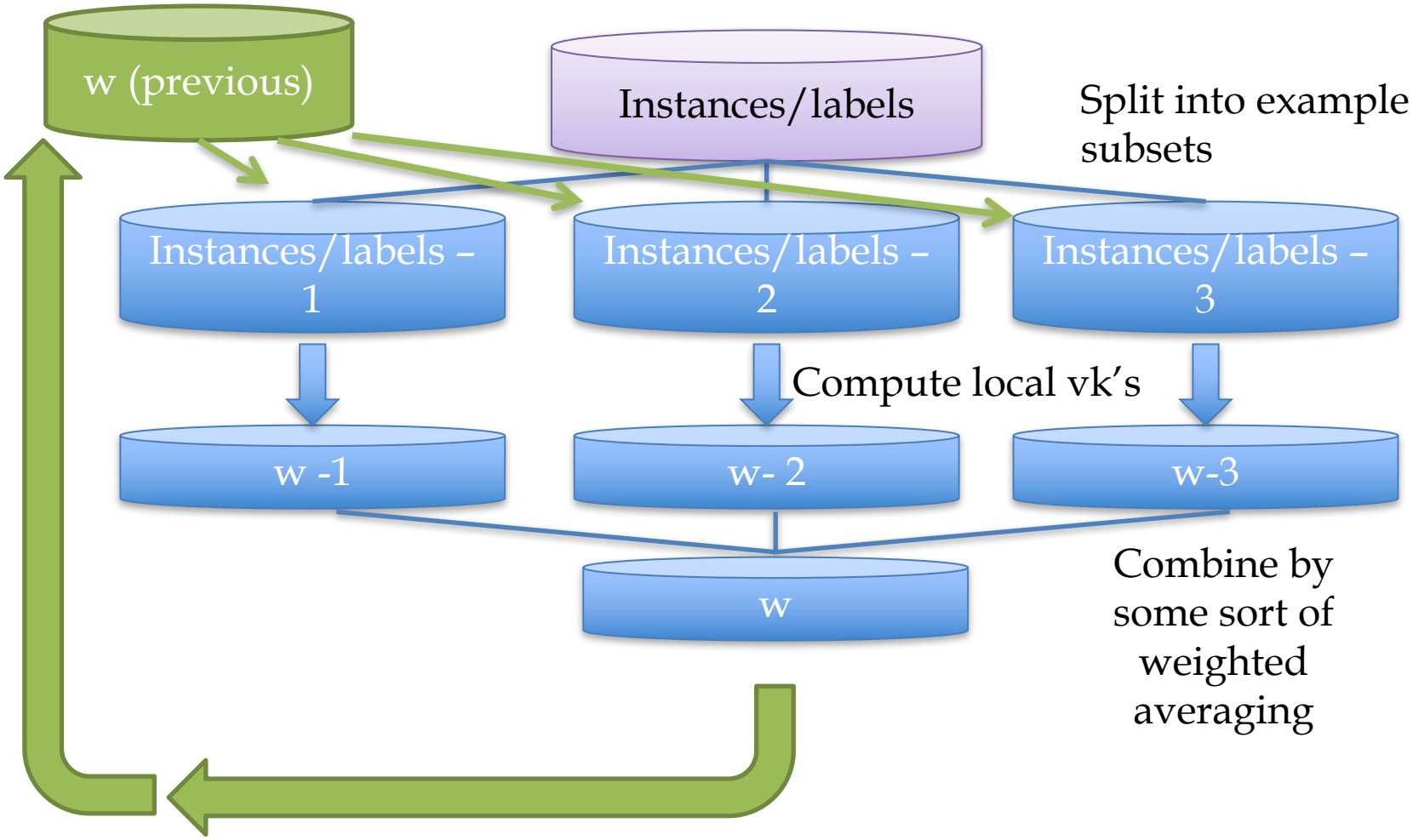
  Obtain f_t and incur loss $f_t(x_t)$

  Compute $g_t := \nabla f_t(x_t)$

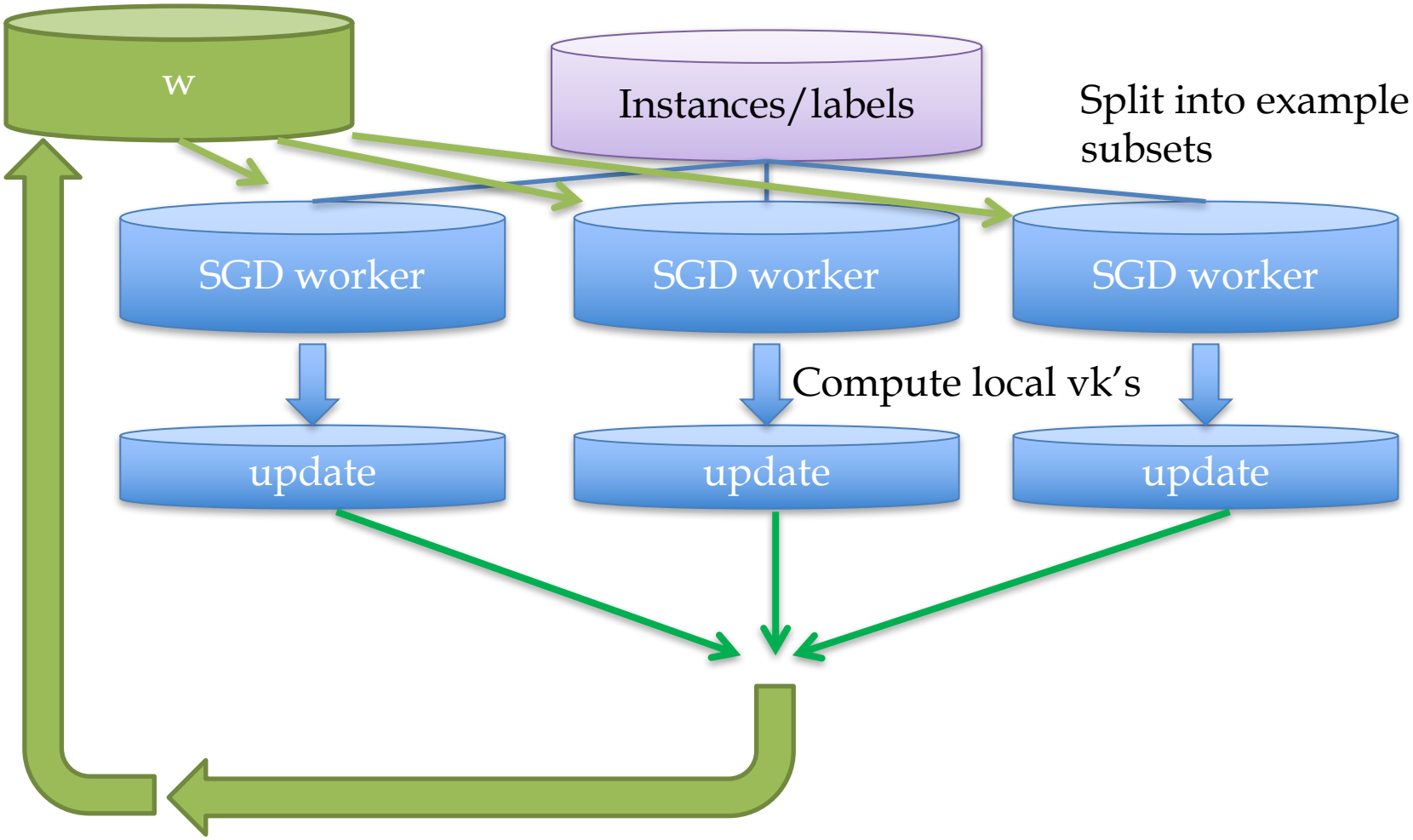
  Update $x_{t+1} = \text{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)

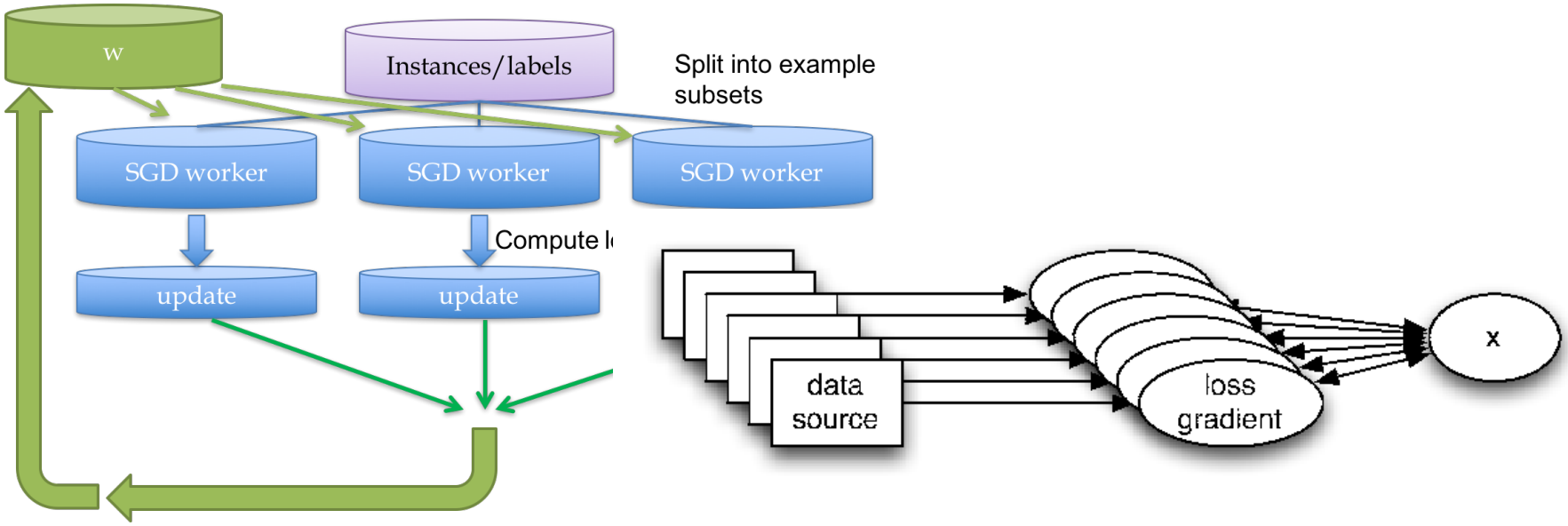
end for

Parallelizing perceptrons – take 2



Parallelizing perceptrons – take 2





Algorithm 1 Delayed Stochastic Gradient Descent

Input: Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule η_t and delay $\tau \in \mathbb{N}$

Initialization: set $x_1 \ldots, x_\tau = 0$ and compute corresponding $g_t = \nabla f_t(x_t)$.

for $t = \tau + 1$ **to** $T + \tau$ **do**

Obtain f_t and incur loss $f_t(x_t)$

Compute $g_t := \nabla f_t(x_t)$

Update $x_{t+1} = \text{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)

end for

Regret: how much loss was incurred **during learning**, over and above the loss incurrent with an optimal choice of x

$$R[X] := \sum_{t=1}^{T} f_t(x_t) - f_t(x^*).$$

Special case:

- f_t is 1 if a mistake was made, 0 otherwise
- $f_t(x^*) = 0$ for optimal x^*

Regret = # mistakes made in learning

Theorem: you can find a learning rate so that the regret of delayed SGD is bounded by

$$R[X] \leq 4FL\sqrt{\tau T}$$

T = # timesteps
τ= staleness > 0

$$\max_{x,x' \in X} D(x\|x') \leq F^2$$

$$D(x\|x') := \tfrac{1}{2}\|x - x'\|^2$$

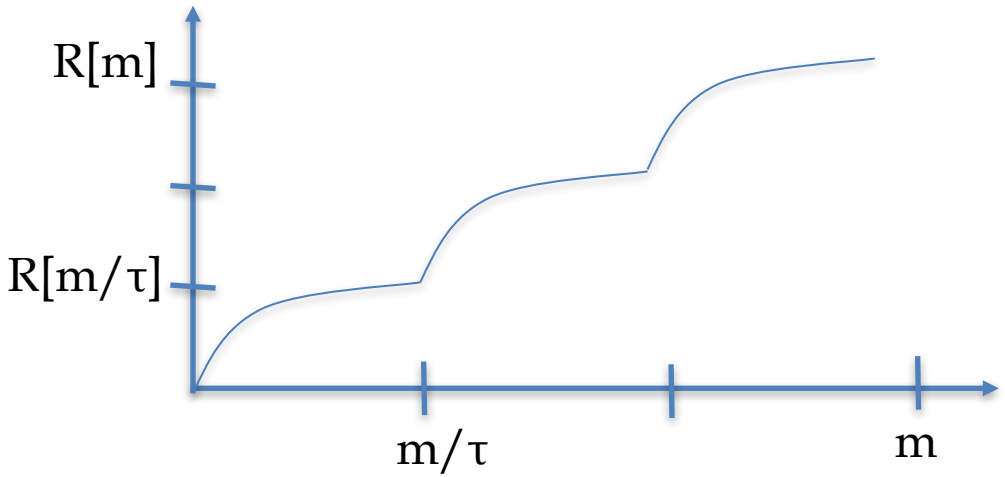
$$\|\nabla f_t(x)\| \leq L$$

Examples are "near each other"; generalizes "all are close to origin"

Gradient is bounded; no sudden sharp changes

Lemma: if you have to use information from at least τ time steps ago then $R[m] \geq \tau R[m/\tau]$

$R[m]$ = regret after m instances



the worse case!

In this case a parallel algorithm is no faster than a sequential code.

Theorem: you can find a learning rate so that the regret of delayed SGD is bounded by

$$R[X] < 4FL\sqrt{\alpha\tau T}$$

T = # timesteps
τ= staleness > 0

α is a measure of how much gradients at different examples are correlated

$$\|\nabla f_t(x) - \nabla f_t(x')\| \leq H \|x - x'\|$$

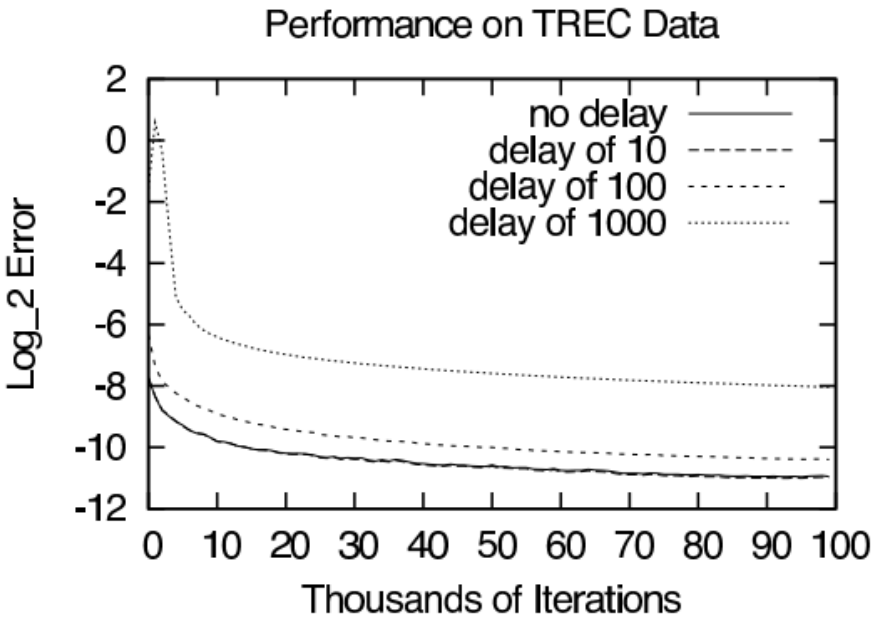
Theorem: you can do better if you assume the gradients change slowly when you change x:

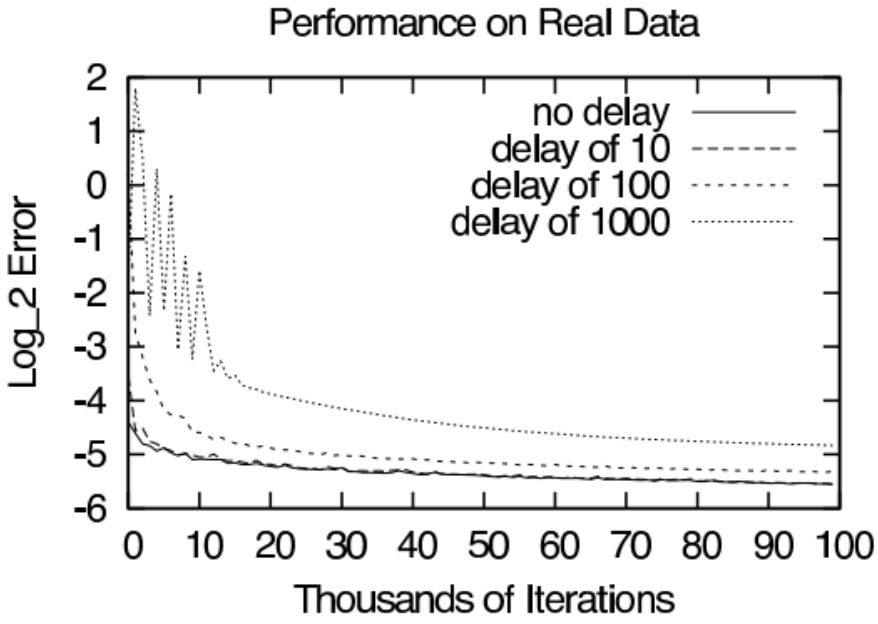
$$\mathbf{E}[R[X]] \leq \left[28.3F^2H + \frac{2}{3}FL + \frac{4}{3}F^2H \log T\right]\tau^2 + \frac{8}{3}FL\sqrt{T}.$$

dominant term

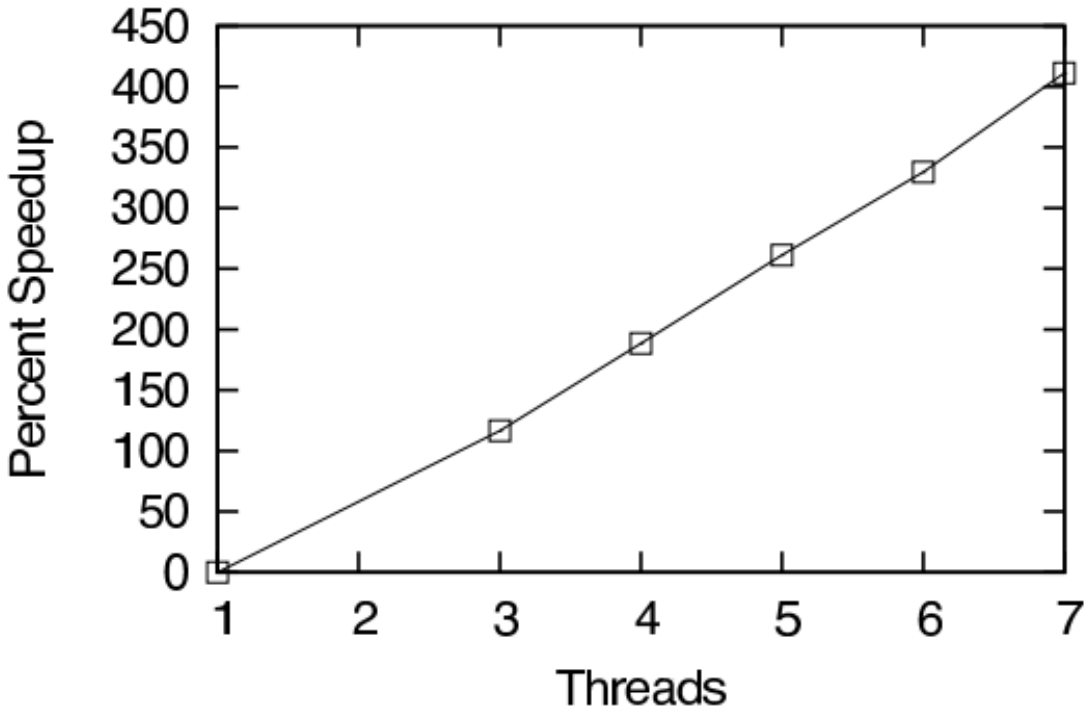
No-delay loss

Experiments?





Experiments



But: this speedup required using a quadratic kernel which took 1 ms/example

Followup points

- On beyond perceptron-based analysis
  - delayed SGD, the theory
- All-Reduce in Map-Reduce
- How bad is Take 1 of this paper?

ALL-REDUCE

Introduction

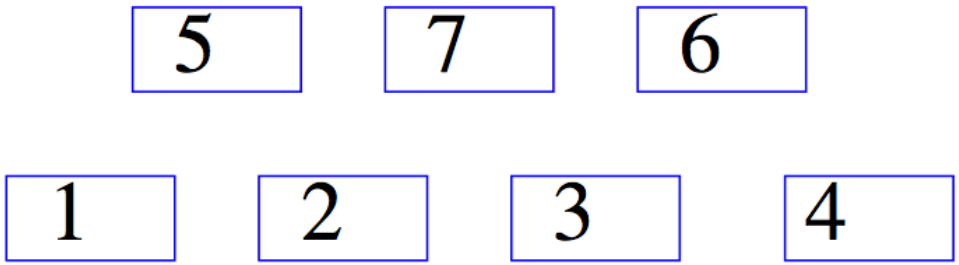
- Common pattern:
  - do some learning in parallel
  - aggregate local changes from each processor
    - to shared parameters
  - distribute the new shared parameters
    - back to each processor
  - and repeat….

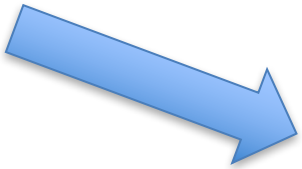
MAP

ALLREDUCE

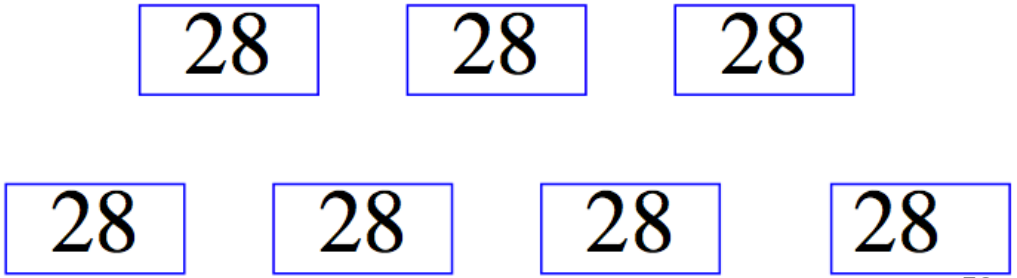
- AllReduce implemented in MPI, also in VW code (John Langford) in a Hadoop/compatible scheme

Allreduce initial state

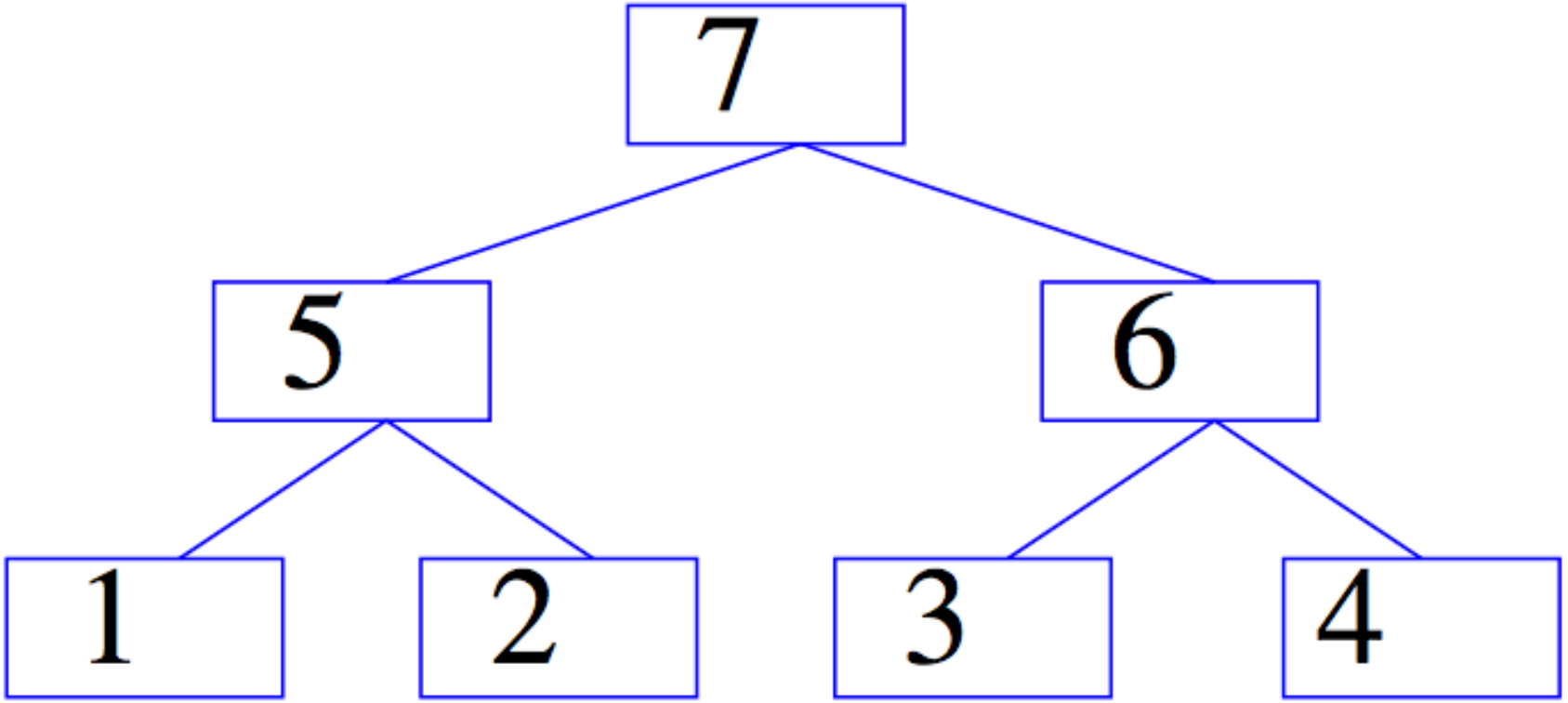




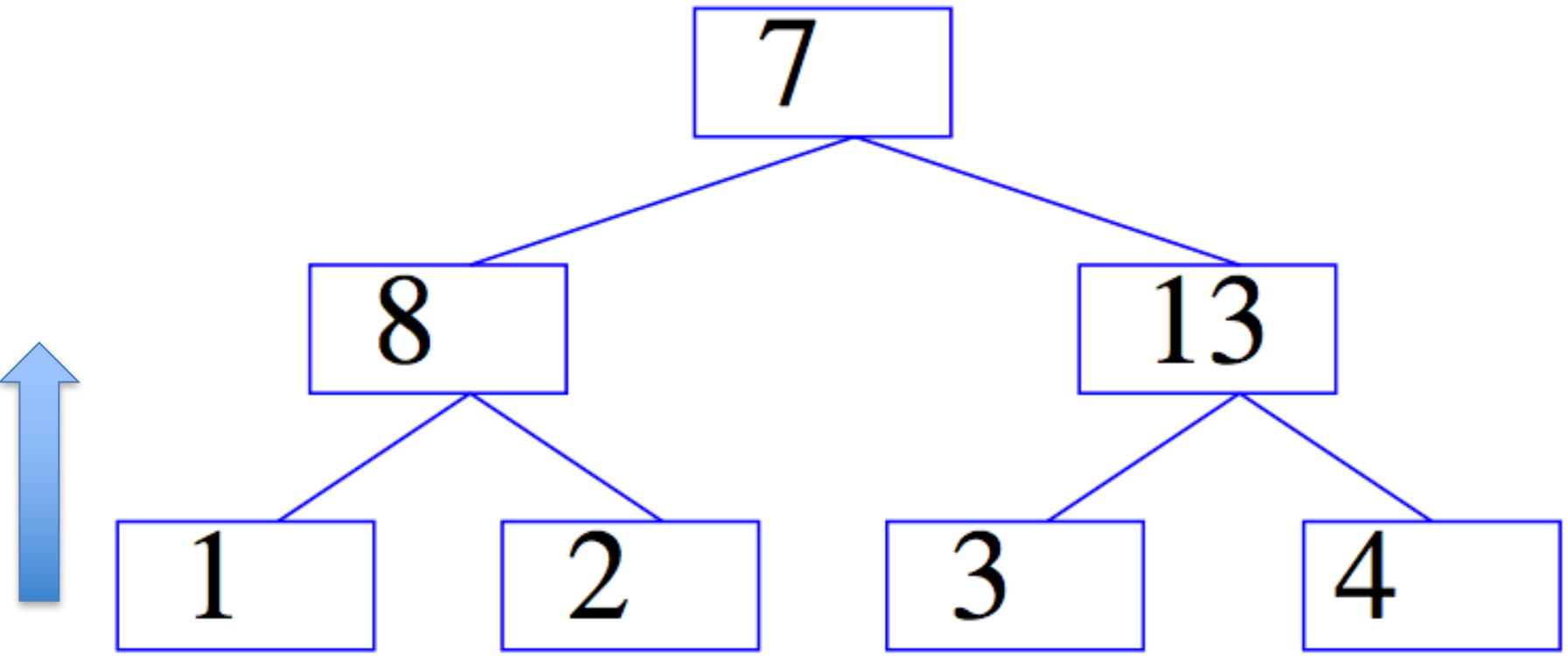
Allreduce final state



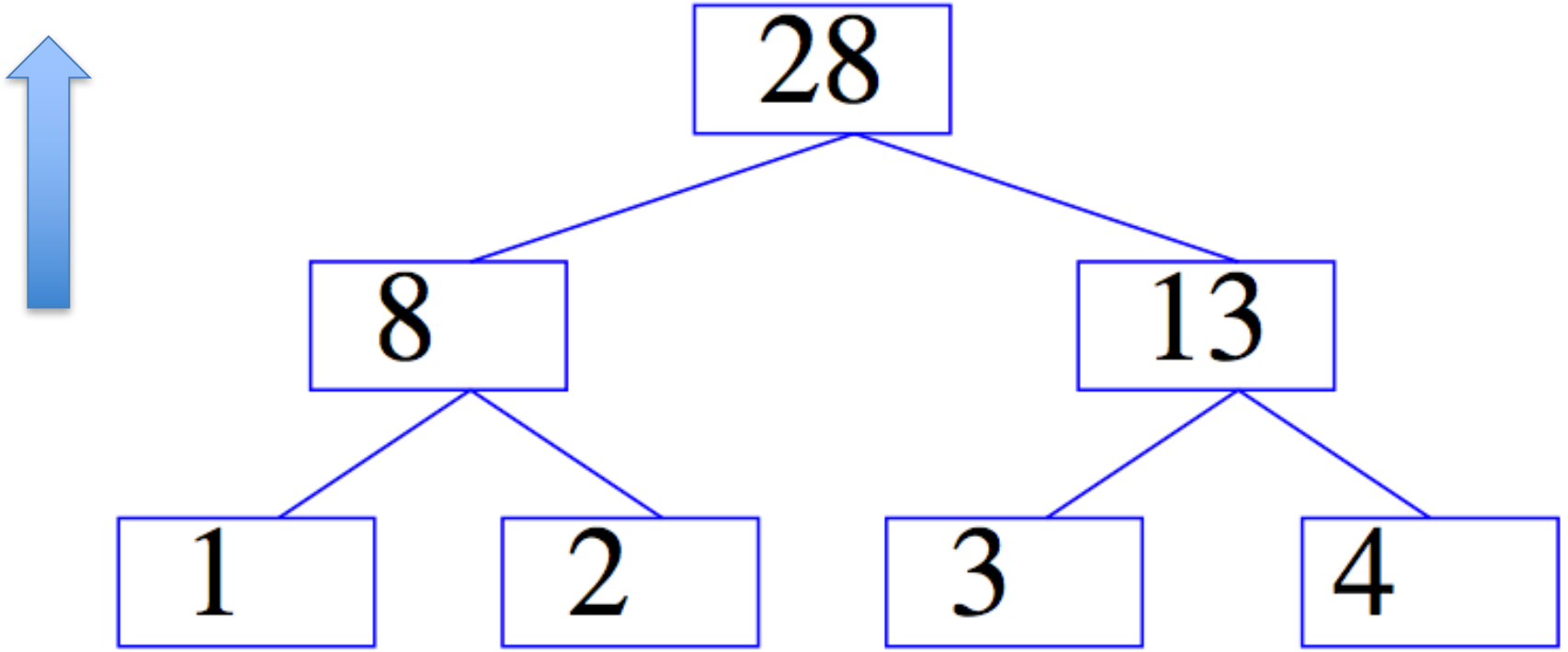
Create Binary Tree



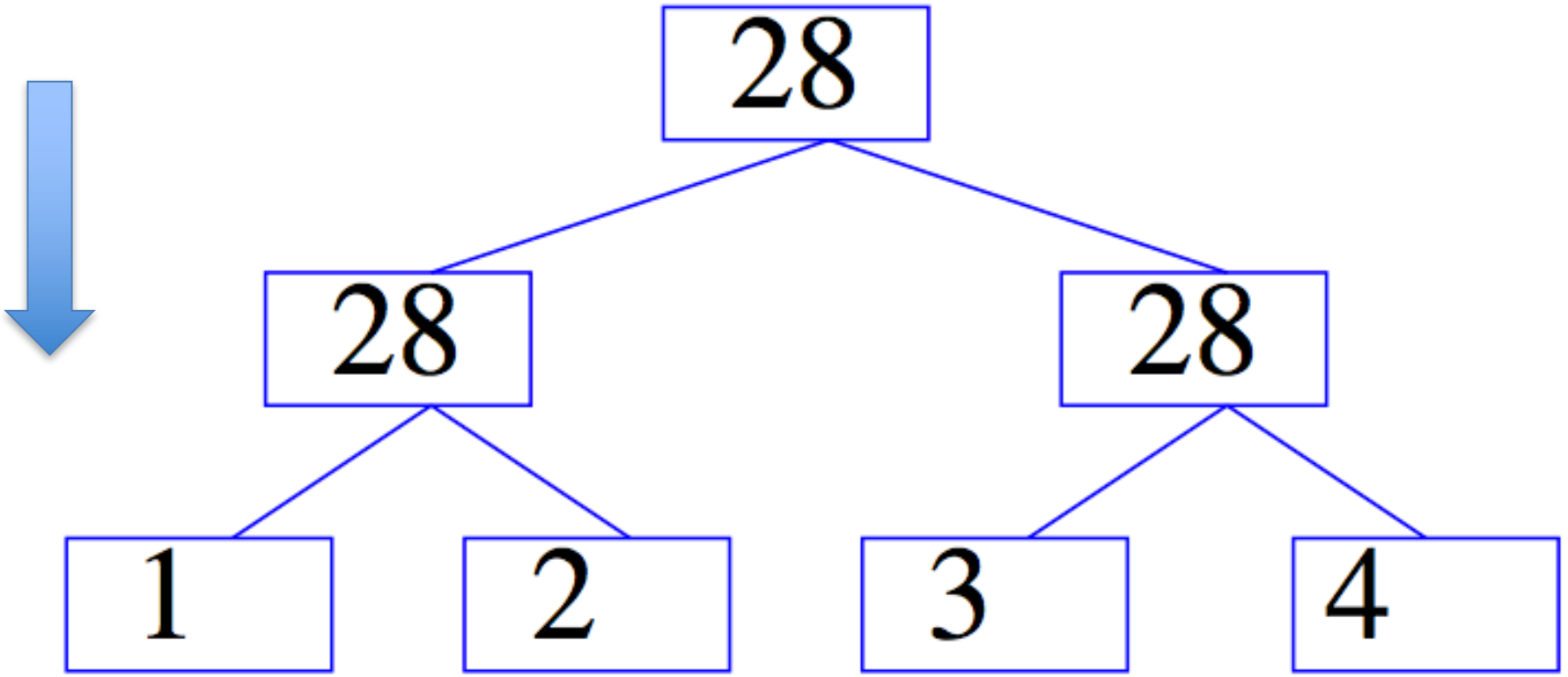
Reducing, step 1



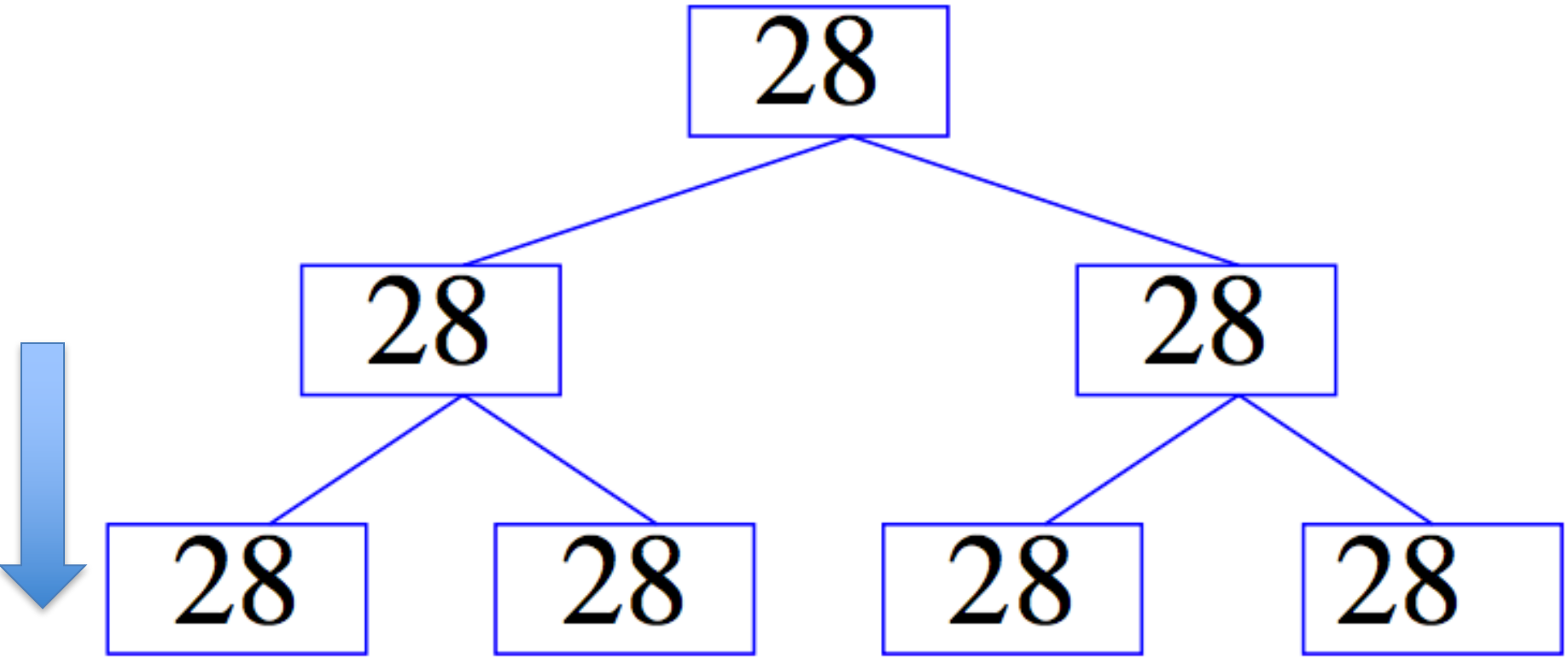
Reducing, step 2



Broadcast, step 1



Allreduce final state

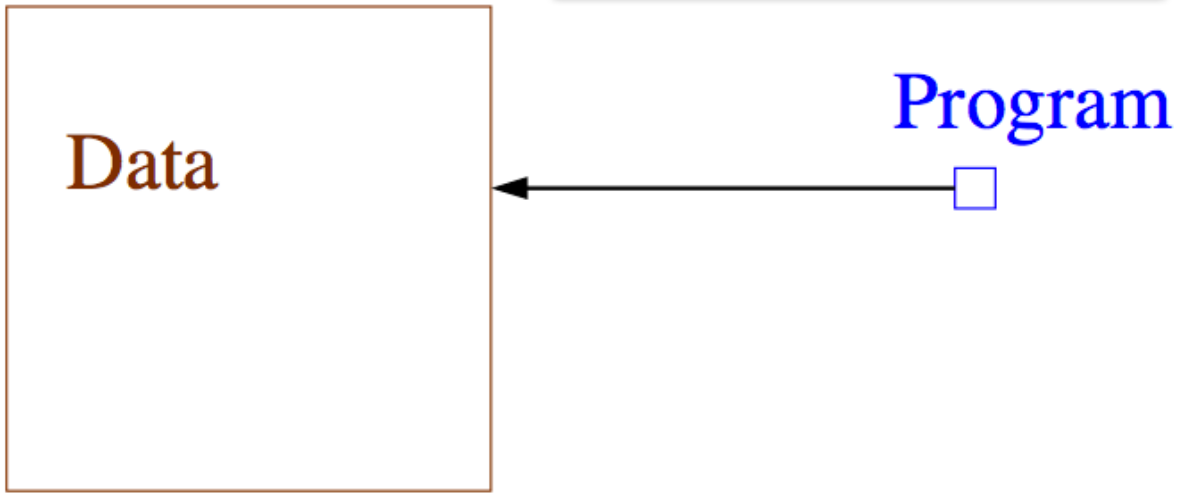


AllReduce = Reduce+Broadcast

Gory details of VW Hadoop-AllReduce

- Spanning-tree server:
  - Separate process constructs a spanning tree of the *compute nodes in the cluster* and then acts as a server
- Worker nodes ("fake" mappers):
  - Input for worker is locally cached
  - Workers all connect to spanning-tree server
  - Workers all execute the same code, which might contain AllReduce calls:
    - Workers **synchronize** whenever they reach an all-reduce

Hadoop AllReduce



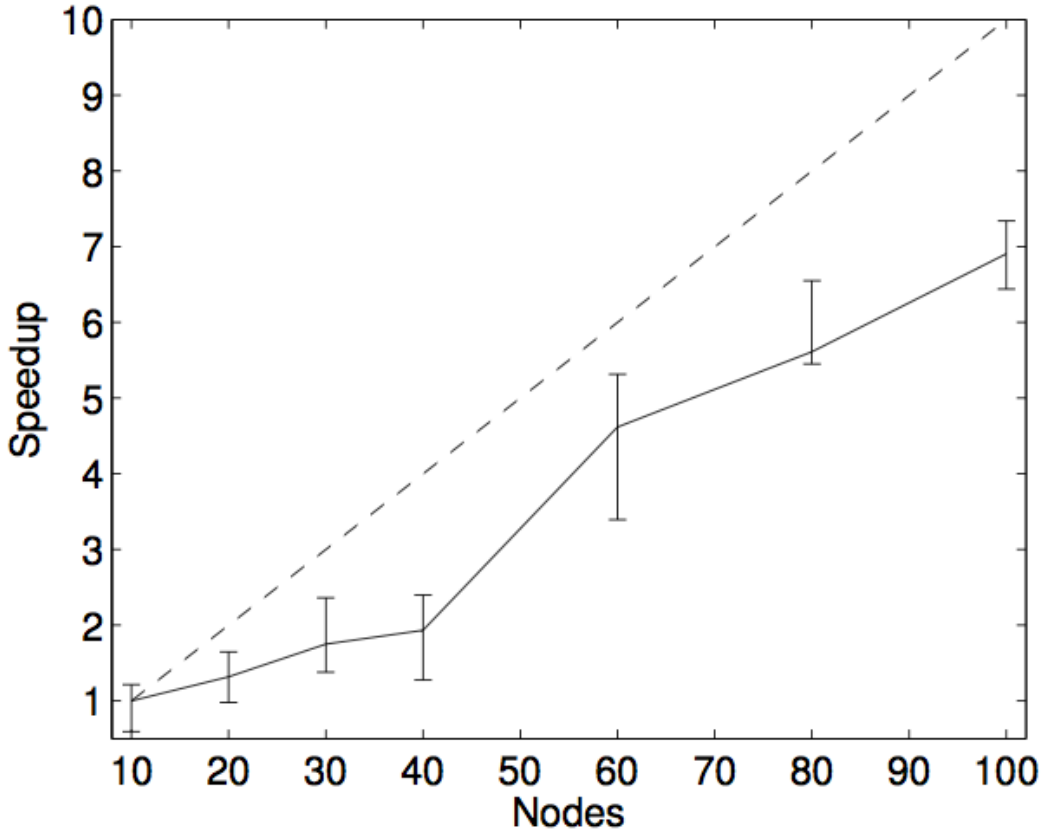
1. "Map" job moves program to data.
2. Delayed initialization: Most failures are disk failures. First read (and cache) all data, before initializing allreduce. Failures autorestart on different node with identical data.
3. Speculative execution: In a busy cluster, one node is often slow. Hadoop can speculatively start additional mappers. We use the first to finish reading all data once.

don't wait for duplicate job

1. Optimize hard so few data passes required.
    1. Normalized, adaptive, safe, online, gradient descent.
    2. L-BFGS
    3. Use (1) to warmstart (2).
2. Use map-only Hadoop for process control and error recovery.
3. Use AllReduce code to sync state.
4. Always save input examples in a cachefile to speed later passes.
5. Use hashing trick to reduce input complexity.

Second-order method - like Newton's method

Open source in Vowpal Wabbit 6.1. Search for it.



2^{24} features

~=100 non-zeros/example

2.3B examples

example is user/page/ad and conjunctions of these, positive if there was a click-thru on the ad

Figure 2: Speed-up for obtaining a fixed test error, on the display advertising problem, relative to the run with 10 nodes, as a function of the number of nodes. The dashed corresponds to the ideal speed-up, the solid line is the average speed-up over 10 repetitions and the bars indicate maximum and minimal values.

Table 3: Computing time on the splice site recognition data with various number of nodes for obtaining a fixed test error. The first 3 rows are average per iteration (excluding the first one).

Nodes	100	200	500	1000
Comm time / pass	5	12	9	16
Median comp time / pass	167	105	43	34
Max comp time / pass	462	271	172	95
Wall clock time	3677	2120	938	813

50M examples

explicitly constructed kernel ➔ 11.7M features

3,300 nonzeros/example

old method: SVM, 3 days: reporting time to get to fixed test error

Table 5: Average training time per iteration of an internal logistic regression implementation using either MapReduce or AllReduce for gradients aggregation. The dataset is the display advertising one and a subset of it.

	Full size	10% sample
MapReduce	1690	1322
AllReduce	670	59

Followup points

- On beyond perceptron-based analysis
  - delayed SGD, the theory
- All-Reduce in Map-Reduce
- How bad is Take 1 of this paper?

Parallel Structured Perceptrons

- Simplest idea:
  - Split data into S "shards"
  - Train a perceptron on each shard independently
    - weight vectors are $\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, …
  - Produce a weighted average of the $\mathbf{w}^{(i)}$'s as the final result

aka parameter mixture

PerceptronParamMix($\mathcal{T} = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|\mathcal{T}|}$)

1. Shard $\mathcal{T}$ into S pieces $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_S\}$
2. $\mathbf{w}^{(i)} = \text{Perceptron}(\mathcal{T}_i)$ †
3. $\mathbf{w} = \sum_i \mu_i \mathbf{w}^{(i)}$ ‡
4. return $\mathbf{w}$

Figure 2: Distributed perceptron using a parameter mixing strategy. † Each $\mathbf{w}^{(i)}$ is computed in parallel. ‡ $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_S\}$, $\forall \mu_i \in \boldsymbol{\mu} : \mu_i \geq 0$ and $\sum_i \mu_i = 1$.

Parallelized Stochastic Gradient Descent

Martin A. Zinkevich
Yahoo! Labs
Sunnyvale, CA 94089
maz@yahoo-inc.com

Markus Weimer
Yahoo! Labs
Sunnyvale, CA 94089
weimer@yahoo-inc.com

Alex Smola
Yahoo! Labs
Sunnyvale, CA 94089
smola@yahoo-inc.com

Lihong Li
Yahoo! Labs
Sunnyvale, CA 94089
lihong@yahoo-inc.com

Algorithm 1 SGD($\{c^1, \ldots, c^m\}, T, \eta, w_0$)

for $t = 1$ **to** T **do**
  Draw $j \in \{1 \ldots m\}$ uniformly at random.
  $w_t \leftarrow w_{t-1} - \eta \partial_w c^j(w_{t-1})$.
end for
return w_T.

Algorithm 2 ParallelSGD($\{c^1, \ldots c^m\}, T, \eta, w_0, k$)

for all $i \in \{1, \ldots k\}$ **parallel do**
  $v_i = \text{SGD}(\{c^1, \ldots c^m\}, T, \eta, w_0)$ on client
end for
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} v_i$ and **return** v

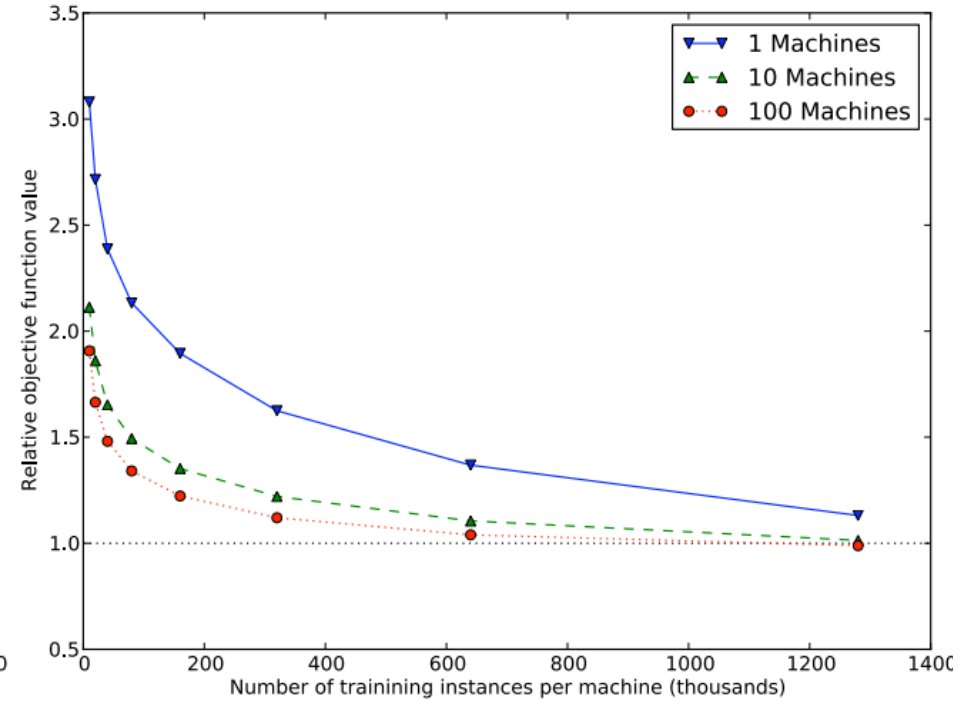
Algorithm 3 SimuParallelSGD(Examples $\{c^1, \ldots c^m\}$, Learning Rate η, Machines k)

Define $T = \lfloor m/k \rfloor$
Randomly partition the examples, giving T examples to each machine.
for all $i \in \{1, \ldots k\}$ **parallel do**
  Randomly shuffle the data on machine i.
  Initialize $w_{i,0} = 0$.
  for all $t \in \{1, \ldots T\}$: **do**
    Get the tth example on the ith machine (this machine), $c^{i,t}$
    $w_{i,t} \leftarrow w_{i,t-1} - \eta \partial_w c^i(w_{i,t-1})$
  end for
end for
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} v_i$ and **return** v

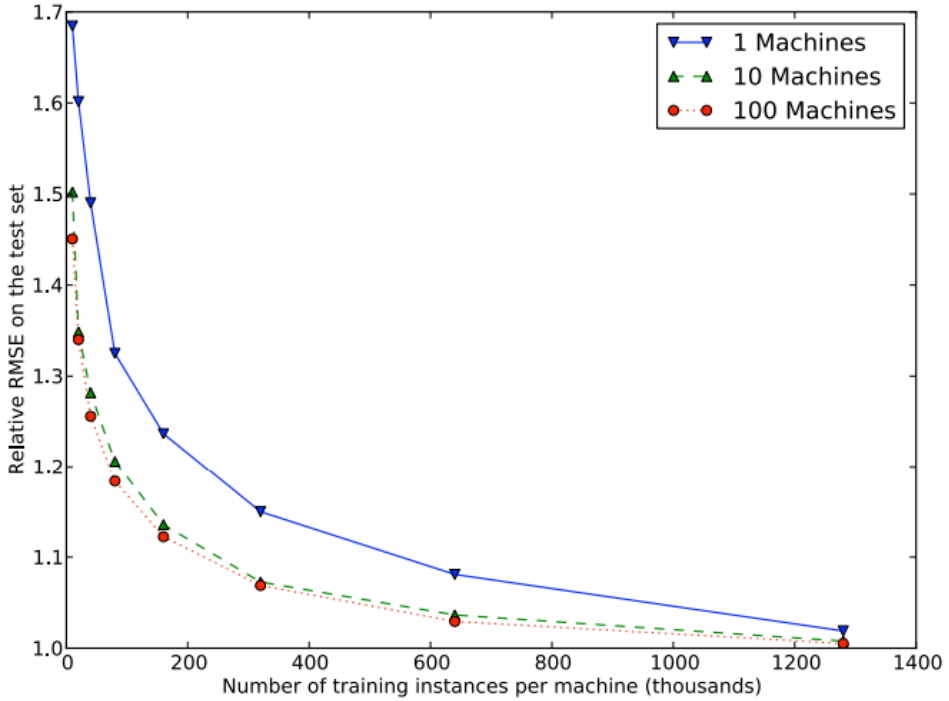
The data is *not* sharded into disjoint sets: **all** machines get **all*** the data

* Not actually **all** but as much as it will have time to look at

Think of this as an ensemble with limited training time…..



Training loss (L2)



Test loss (L2)

There is also a formal bound on regret relative to the best classifier