Parallel Perceptrons and Iterative Parameter Mixing
Recap: perceptrons
The perceptron

\[ \text{Compute: } y_i = \text{sign}(v_k \cdot x_i) \]

\[ \text{If mistake: } v_{k+1} = v_k + y_i x_i \]

Margin $\gamma$. $A$ must provide examples that can be separated with some vector $u$ with margin $\gamma > 0$, i.e.

\[ \exists u : \forall (x_i, y_i) \text{ given by } A, (u \cdot x)y_i > \gamma \]

and furthermore, $\|u\| = 1$.

Radius $R$. $A$ must provide examples “near the origin”, i.e.

\[ \forall x_i \text{ given by } A, \|x\|^2 < R^2 \]
The perceptron

\[ y_i = \text{sign}(v_k \cdot x_i) \]

If mistake:
\[ v_{k+1} = v_k + y_i x_i \]

Mistake bound:
\[ k \leq \frac{R^2}{\gamma^2} = \left( \frac{R}{\gamma} \right)^2 \]
On-line to batch learning

1. Pick a $v_k$ at random according to $m_k/m$, the fraction of examples it was used for.

2. Predict using the $v_k$ you just picked.

3. (Actually, use some sort of deterministic approximation to this).

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>guess</th>
<th>input</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$v_0$</td>
<td>$x_1$</td>
<td>X (a mistake)</td>
</tr>
<tr>
<td>2</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>√ (correct!)</td>
</tr>
<tr>
<td>3</td>
<td>$v_1$</td>
<td>$x_3$</td>
<td>√</td>
</tr>
<tr>
<td>4</td>
<td>$v_1$</td>
<td>$x_4$</td>
<td>X (a mistake)</td>
</tr>
<tr>
<td>5</td>
<td>$v_2$</td>
<td>$x_5$</td>
<td>√</td>
</tr>
<tr>
<td>6</td>
<td>$v_2$</td>
<td>$x_6$</td>
<td>√</td>
</tr>
<tr>
<td>7</td>
<td>$v_2$</td>
<td>$x_7$</td>
<td>√</td>
</tr>
<tr>
<td>8</td>
<td>$v_2$</td>
<td>$x_8$</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>$v_3$</td>
<td>$x_9$</td>
<td>√</td>
</tr>
<tr>
<td>10</td>
<td>$v_3$</td>
<td>$x_{10}$</td>
<td>X</td>
</tr>
</tbody>
</table>

$m_1 = 3$

$m_2 = 4$

$m = 10$
1. Pick a $v_k$ at random according to $m_k/m$, the fraction of examples it was used for.

2. Predict using the $v_k$ you just picked.

3. (Actually, use some sort of deterministic approximation to this).

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

<table>
<thead>
<tr>
<th>i</th>
<th>guess</th>
<th>input</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$v_0$</td>
<td>$x_1$</td>
<td>$X$ (a mistake)</td>
</tr>
<tr>
<td>2</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>✓ (correct!)</td>
</tr>
<tr>
<td>3</td>
<td>$v_1$</td>
<td>$x_3$</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>$v_1$</td>
<td>$x_4$</td>
<td>$X$ (a mistake)</td>
</tr>
<tr>
<td>5</td>
<td>$v_2$</td>
<td>$x_5$</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>$v_2$</td>
<td>$x_6$</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td>$v_2$</td>
<td>$x_7$</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>$v_2$</td>
<td>$x_8$</td>
<td>$X$</td>
</tr>
<tr>
<td>9</td>
<td>$v_3$</td>
<td>$x_9$</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>$v_3$</td>
<td>$x_{10}$</td>
<td>$X$</td>
</tr>
</tbody>
</table>
predict using $\text{sign}(v^* . x)$

$$v^* = \sum_k \left( \frac{m_k}{m} v_k \right)$$

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

![Graph showing comparison between different methods](image)
The voted perceptron for ranking

Compute: $y_i = \mathbf{v}_k \cdot \mathbf{x}_i$

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

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

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

Radius $R$. $A$ must provide examples "near the origin", i.e.

$$\forall \mathbf{x}_i \text{ given by } A, \|\mathbf{x}\|^2 < R^2$$
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: \( x=(w_1,\ldots,w_n) \)
  – Output: list of labels: \( y=(y_1,\ldots,y_n) \)
  – 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: \( x = (w_1, \ldots, w_n) \)
  - Output: list of labels: \( y = (y_1, \ldots, y_n) \)
  - 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: \( x=(w_1,\ldots,w_n) \)
  - Output: list of labels: \( y=(y_1,\ldots,y_n) \)
  - 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 $w^{(1)}, w^{(2)}, \ldots$
  – Produce a weighted average of the $w^{(i)}$’s as the final result

\[
\text{PerceptronParamMix}(T = \{(x_t, y_t)\}_{t=1}^{\left|T\right|})
\]

1. Shard $T$ into $S$ pieces $T = \{T_1, \ldots, T_S\}$
2. $w^{(i)} = \text{Perceptron}(T_i)$
3. $w = \sum_i \mu_i w^{(i)}$
4. return $w$

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

Instances/labels

- Instances/labels – 1
- Instances/labels – 2
- Instances/labels – 3

vk -1
vk -2
vk -3

vk

Split into example subsets

Compute vk’s on subsets

Combine by some sort of weighted averaging
Parallel Perceptrons

- Simplest idea:
  - Split data into $S$ “shards”
  - Train a perceptron on each shard independently
    - weight vectors are $w^{(1)}, w^{(2)}, \ldots$
  - Produce some weighted average of the $w^{(i)}$'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($T = \{(x_t, y_t)\}_{t=1}^{T}$)
1. Shard $T$ into $S$ pieces $T = \{T_1, \ldots, T_S\}$
2. $w^{(i)} = \text{Perceptron}(T_i)$
3. $w = \sum_i \mu_i w^{(i)}$
4. return $w$
```

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

Idea: do the simplest possible thing iteratively.

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

Extra communication cost:
• redistributing the weight vectors
• done less frequently than if fully synchronized, more frequently than if fully parallelized
Parallelizing perceptrons – take 2

1. Instances/labels
2. Split into example subsets
3. Compute local \(v_k\)'s
4. Combine by some sort of weighted averaging

\[
\begin{align*}
w & \quad (\text{previous}) \\
& \quad \text{Instances/labels} \\
\quad & \quad \text{Instances/labels} - 1 \\
& \quad w -1 \\
\quad & \quad \text{Instances/labels} - 2 \\
& \quad w - 2 \\
\quad & \quad \text{Instances/labels} - 3 \\
& \quad w - 3 \\
\quad & \quad w
\end{align*}
\]
**Theorem 3.** Assume a training set $T$ is separable by margin $\gamma$. Let $k_{i,n}$ be the number of mistakes that occurred on shard $i$ during the $n$th 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 $T$ is separable by margin $\gamma$. Let $k_{i,n}$ be the number of mistakes that occurred on shard $i$ during the $n$th epoch of training. For any $N$, when training the perceptron with iterative parameter mixing (Figure 3),

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

uniform mixing…$\mu=1/S$

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

<table>
<thead>
<tr>
<th></th>
<th>Reg. Perceptron F-measure</th>
<th>Avg. Perceptron F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial (All Data)</td>
<td>85.8</td>
<td>88.2</td>
</tr>
<tr>
<td>Serial (Sub Sampling)</td>
<td>75.3</td>
<td>76.6</td>
</tr>
<tr>
<td>Parallel (Parameter Mix)</td>
<td>81.5</td>
<td>81.6</td>
</tr>
<tr>
<td>Parallel (Iterative Parameter Mix)</td>
<td>87.9</td>
<td>88.1</td>
</tr>
</tbody>
</table>
Results on parsing

perceptron

Averaged perceptron

<table>
<thead>
<tr>
<th></th>
<th>Reg. Perceptron</th>
<th>Avg. Perceptron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unlabeled Attachment Score</td>
<td>Unlabeled Attachment Score</td>
</tr>
<tr>
<td>Serial (All Data)</td>
<td>81.3</td>
<td>84.7</td>
</tr>
<tr>
<td>Serial (Sub Sampling)</td>
<td>77.2</td>
<td>80.1</td>
</tr>
<tr>
<td>Parallel (Iterative Parameter Mix)</td>
<td>83.5</td>
<td>84.5</td>
</tr>
</tbody>
</table>
Theorem 3. Assume a training set $T$ is separable by margin $\gamma$. Let $k_{i,n}$ be the number of mistakes that occurred on shard $i$ during the $n$th epoch of training. For any $N$, when training the perceptron with iterative parameter mixing (Figure 3),

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

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

$$
\mathbf{u} \cdot \mathbf{w}^{(i,n)} = \mathbf{u} \cdot \mathbf{w}^{([i,n]-1)} + \mathbf{u} \cdot (f(x_t, y_t) - f(x_t, y')) \\
\geq \mathbf{u} \cdot \mathbf{w}^{([i,n]-1)} + \gamma \\
\geq \mathbf{u} \cdot \mathbf{w}^{([i,n]-2)} + 2\gamma \\
\ldots \geq \mathbf{u} \cdot \mathbf{w}^{(\text{avg},n-1)} + k_{i,n} \gamma \quad (A1)
$$
The theorem...

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

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

$$
\|w^{(i,n)}\|^2 = \|w^{([i,n]-1)}\|^2 + \|f(x_t, y_t) - f(x_t, y')\|^2 + 2w^{([i,n]-1)}(f(x_t, y_t) - f(x_t, y'))
$$

$$
\leq \|w^{([i,n]-1)}\|^2 + R^2
$$

$$
\leq \|w^{([i,n]-2)}\|^2 + 2R^2
$$

$$
\ldots \leq \|w^{(avg,n-1)}\|^2 + k_{i,n}R^2 \quad (A2)
$$

Perceptron($\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$)

1. $w^{(0)} = 0; \quad k = 0$
2. for $n : 1..N$
3. for $t : 1..T$
4. Let $y' = \text{arg max}_y w^{(k)} \cdot f(x_t, y)$
5. if $y' \neq y_t$
6. $w^{(k+1)} = w^{(k)} + f(x_t, y_t) - f(x_t, y')$
7. $k = k + 1$
8. return $w^{(k)}$
(3a) The guess $v_2$ after the two positive examples: $v_2 = v_1 + x_2$

(3b) The guess $v_2$ after the one positive and one negative example: $v_2 = v_1 - x_2$

Lemma 2 $\forall k, \|v_k\|^2 \leq kR$. In other words, the norm of $v_k$ grows “slowly”, at a rate depending on $R$.

Proof:

$$v_{k+1} \cdot v_{k+1} = (v_k + y_i x_i) \cdot (v_k + y_i x_i)$$

$$\Rightarrow \quad \|v_{k+1}\|^2 = \|v_{k+1}\|^2 + 2y_i x_i \cdot v_k + y_i^2 \|x\|^2$$

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

$$\Rightarrow \quad \|v_{k+1}\|^2 \leq \|v_{k+1}\|^2 + \|x\|^2$$

$$\Rightarrow \quad \|v_{k+1}\|^2 \leq \|v_{k+1}\|^2 + R^2$$

$$\Rightarrow \quad \|v_k\|^2 \leq kR^2$$

If mistake: $y_i x_i v_k < 0$

$\forall x_i$ given by $A$, $\|x\|^2 < R^2$
Using A1/A2 we prove two inductive hypotheses:

\[ u \cdot w^{(avg,N)} \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \]  \hspace{1cm} (IH1)

\[ \|w^{(avg,N)}\|^2 \leq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2 \]  \hspace{1cm} (IH2)

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

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

Follows from: \( u \cdot w^{(i,1)} \geq k_{1,i} \gamma \)

This is new …. We’ve never considered averaging operations before
IH1 inductive case:

\[
\begin{align*}
\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N)} &= \sum_{i=1}^{S} \mu_{i,N}(\mathbf{u} \cdot \mathbf{w}^{(i,N)}) \\
&\geq \sum_{i=1}^{S} \mu_{i,N}(\mathbf{u} \cdot \mathbf{w}^{(\text{avg},N-1)} + k_{i,N}\gamma) \\
&= \mathbf{u} \cdot \mathbf{w}^{(\text{avg},N-1)} + \sum_{i=1}^{S} \mu_{i,N} k_{i,N}\gamma \\
&\geq \left[ \sum_{n=1}^{N-1} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\gamma \right] + \sum_{i=1}^{S} \mu_{i,N} k_{i,N}\gamma \\
&= \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n}\gamma \\
\end{align*}
\]

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

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:

\[ u \cdot w^{(\text{avg,}N)} \geq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} \gamma \]  \hspace{1cm} (IH1)

\[ \|w^{(\text{avg,}N)}\|^2 \leq \sum_{n=1}^{N} \sum_{i=1}^{S} \mu_{i,n} k_{i,n} R^2 \]  \hspace{1cm} (IH2)

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

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

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

uniform mixing...

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

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

Figure 6: Training errors per epoch for different shard size and parameter mixing strategies.
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
Algorithm 1 Delayed Stochastic Gradient Descent

**Input:** Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule $\eta_t$ 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 $x_{t+1} = \operatorname{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)

end for
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 \( x' \): \[ x_{t+1} = \text{argmin}_x \text{dist}(x - x') \]
3. But…. you might be using a “stale” \( g \) (from \( \tau \) steps ago)

---

**Algorithm 1** Delayed Stochastic Gradient Descent

**Input:** Feasible space \( X \subseteq \mathbb{R}^n \), annealing schedule, delay \( \tau \in \mathbb{N} \)

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

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

- Split into example subsets
- Compute local $v_k$'s
- Combine by some sort of weighted averaging
Parallelizing perceptrons – take 2

1. Instances/labels
2. Split into example subsets
3. SGD worker
4. Compute local $v_k$'s
5. Update
Algorithm 1 Delayed Stochastic Gradient Descent

**Input:** Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule $\eta_t$ 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 $x_{t+1} = \arg \min_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)

**end for**
Regret: how much loss was incurred during learning, over and above the loss incurred 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

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

where \( T = \# \text{ timesteps} \) and \( \tau = \text{staleness} > 0 \).

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 \( \tau \) time steps ago then \( R[m] \geq \tau R[m/\tau] \)

\( R[m] = \) regret after \( m \) instances

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
\( \tau = \) staleness > 0

\( \alpha \) is a measure of how much gradients at different examples are correlated.
Theorem: you can do better if you assume the gradients change slowly when you change $x$: 

\[
\|\nabla f_t(x) - \nabla f_t(x')\| \leq H \|x - x'\|.
\]

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

• AllReduce implemented in MPI, also in VW code (John Langford)
  in a Hadoop/compatible scheme
Allreduce initial state

5  7  6

1  2  3  4

Allreduce final state

28  28  28

28  28  28  28  28
Create Binary Tree

```
    7
   / \   /
  5   6
 / \ / \ / \   / \ / \ / \   / \ / \ / \  
1  2 3  4
```
Reducing, step 1
Reducing, step 2
Broadcast, step 1

```
  28
 /   \
28    28
 |    /   \
1    2    3   4
```
Allreduce final state

\[ 28 \]

\[ \begin{array}{c}
  28 \\
  \downarrow \\
  28 \\
  \downarrow \\
  28 \\
  \downarrow \\
  28 \\
  \downarrow \\
  28 \\
\end{array} \]

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
“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. *don’t wait for duplicate job*

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.*
1. Optimize hard so few data passes required.
   - Normalized, adaptive, safe, online, gradient descent.
   - L-BFGS 
   - 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 cache file to speed later passes.

5. Use hashing trick to reduce input complexity.

Open source in Vowpal Wabbit 6.1. Search for it.
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).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm time / pass</td>
<td>5</td>
<td>12</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>Median comp time / pass</td>
<td>167</td>
<td>105</td>
<td>43</td>
<td>34</td>
</tr>
<tr>
<td>Max comp time / pass</td>
<td>462</td>
<td>271</td>
<td>172</td>
<td>95</td>
</tr>
<tr>
<td>Wall clock time</td>
<td>3677</td>
<td>2120</td>
<td>938</td>
<td>813</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>Full size</th>
<th>10% sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>1690</td>
<td>1322</td>
</tr>
<tr>
<td>AllReduce</td>
<td>670</td>
<td>59</td>
</tr>
</tbody>
</table>
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 $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots$
  - Produce a weighted average of the $\mathbf{w}^{(i)}$'s as the final result

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

Figure 2: Distributed perceptron using a parameter mixing strategy. † Each $\mathbf{w}^{(i)}$ is computed in parallel. ‡ $\mu = \{\mu_1, \ldots, \mu_S\}$, $\forall \mu_i \in \mu : \mu_i \geq 0$ 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 $\eta$, 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 $t$th example on the $i$th 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.....
There is also a formal bound on regret relative to the best classifier.