Mechanisms for Parallel Online Learning

John Langford

(on joint work with Daniel Hsu & Alex Smola & Martin Zinkevich)

How should we write fast learning algorithms?

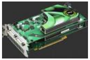
How should we write fast learning algorithms?

1. Speed up slow learning algorithms.

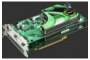
How should we write fast learning algorithms?

1. Speed up slow learning algorithms.
   Canonical Example = Map-Reduce applied to statistical query algorithms. 

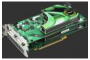
How should we write fast learning algorithms?

1. Speed up slow learning algorithms.
   Canonical Example = Map-Reduce applied to statistical query algorithms. 
2. Switch architectures. Canonical example = GPU  Useful for computationally intense learning, but...

How should we write fast learning algorithms?

1. Speed up slow learning algorithms.
   Canonical Example = Map-Reduce applied to statistical query algorithms. 
2. Switch architectures. Canonical example = GPU
   Useful for computationally intense learning, but... doesn't address problems with large amounts of data.

How should we write fast learning algorithms?

1. Speed up slow learning algorithms.
   Canonical Example = Map-Reduce applied to statistical query algorithms. 
2. Switch architectures. Canonical example = GPU  Useful for computationally intense learning, but... doesn't address problems with large amounts of data.
3. Start with a fast learning algorithm and design a new parallel algorithm that competes with it. Canonical fast algorithm = linear prediction via online gradient descent.

Core Problems for Parallel algorithms = Bandwidth & Latency

1 Gb/s ethernet = 450GB/hour $\Rightarrow$ 1 Terafeature is reasonable.
Latency = 1ms $\simeq 10^6$ cycles.

Core Problems for Parallel algorithms = Bandwidth & Latency

1 Gb/s ethernet = 450GB/hour $\Rightarrow$ 1 Terafeature is reasonable.
Latency = 1ms $\simeq 10^6$ cycles.
Many tricks to reduce the problem:

1. Sparse feature representation
2. Implicit feature representation
3. Compressed format

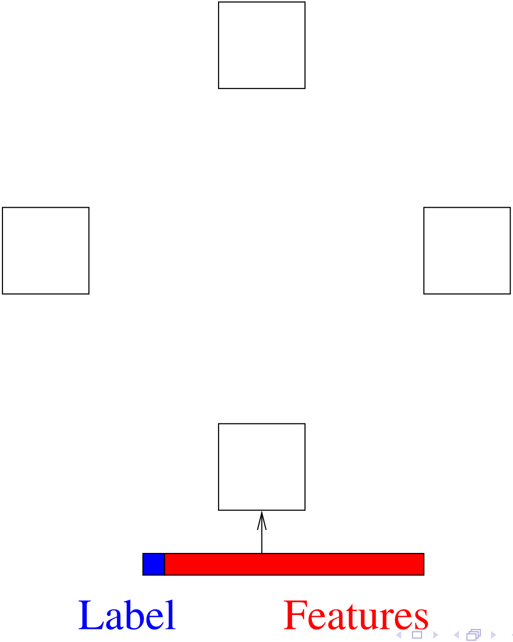
(Vowpal Wabbit has all of these.)
But in the end the problem must be addressed.

Bad news: Delay is pretty bad.

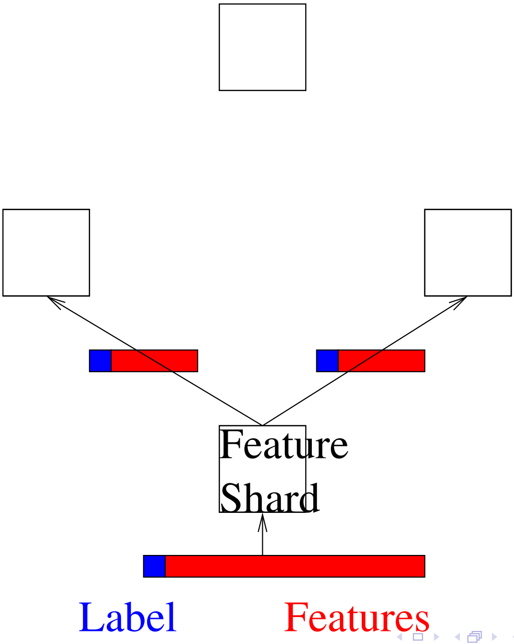
Theorem: (Mesterharm 2005) Delayed updates reduce convergence by delay factor in worst case for expert algorithms.
Theorem: (LSZ 2009) Same for linear predictors.
(Caveat: there are some special cases where you can do better.)

What is an architecture for minimum latency delayed updates?

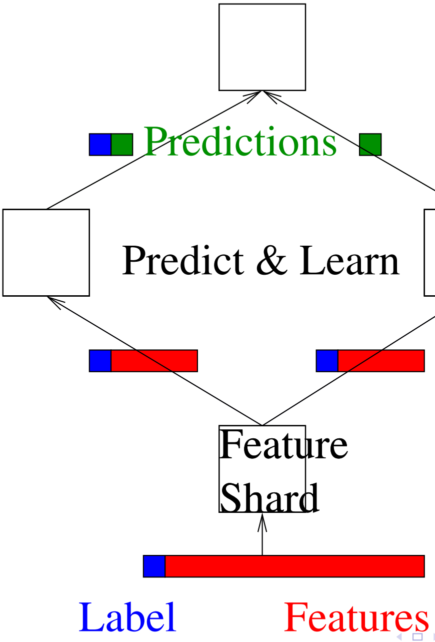
How can we avoid delay?



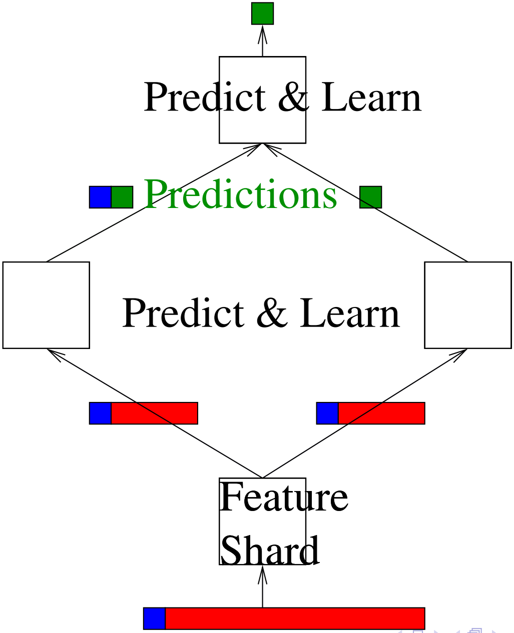
How can we avoid delay?



How can we avoid delay?



How can we avoid delay?



Observations about Feed Forward

1. No longer the same algorithm—it's designed for parallel environments.
2. Bandwidth $=$ few bytes per example, per node $\Rightarrow$ Tera-example feasible with single master, arbitrarily more with hierarchical structure.
3. No delay.
4. Feature Shard is stateless $\Rightarrow$ parallelizable & cachable.

Bad News: Feed Forward can't compete with general linear predictors.

Probability	y	x_1	x_2	x_3
0.25	1	1	1	0
0.125	1	1	0	1
0.125	1	0	1	1
0.25	0	0	0	1
0.125	0	1	0	0
0.125	0	0	1	0

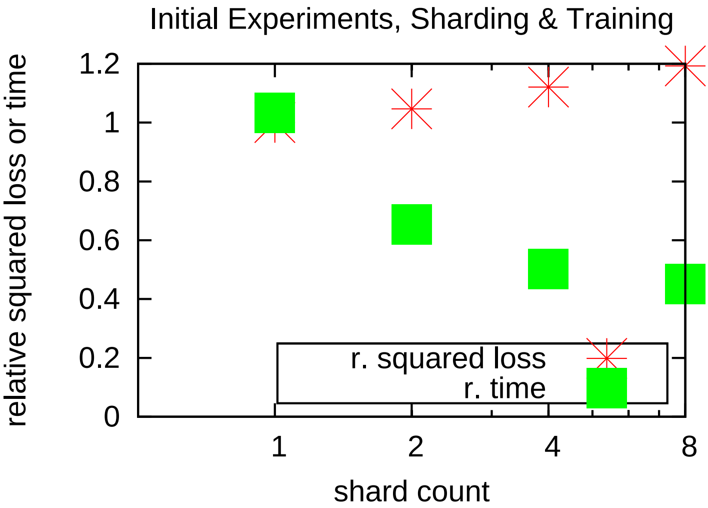
Features 1&2 are imperfect predictors. Feature 3 is uncorrelated with truth. Optimal predictor = majority vote on all 3 features.

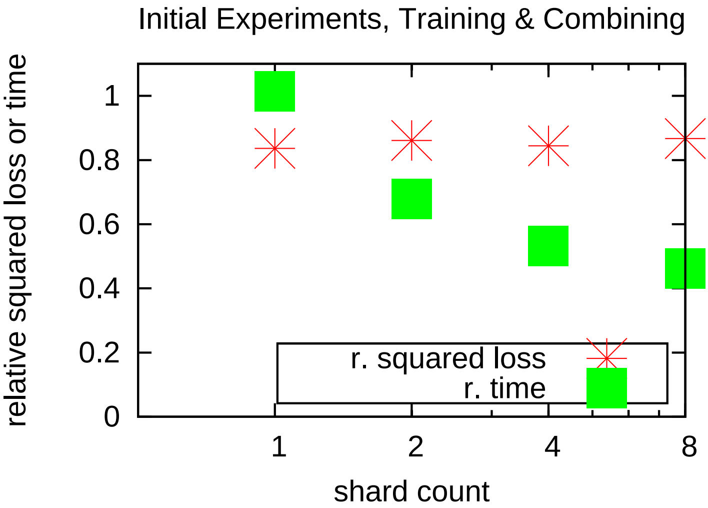
Good news: If Naive Bayes holds $P(x_1|y)P(x_2|y) = P(x_1, x_2|y)$, you win.
Better news: $x_1 =$ first shard, $x_2 =$ second shard
Even better: There are more complex problem classes for which this also works.

Initial experiments on a medium size text Ad dataset @ Yahoo!

1. $\sim$100G when gzip compressed.
2. $\sim$10M examples.
3. $\sim$125G nonzero features
4. Computational constraint weakly active due to implicit features.

Relative progressive validation squared loss & relative wall-clock time reported.


Initial Experiments, Sharding & Training
relative squared loss or time
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
shard count
r. squared loss
r. time


Initial Experiments, Training & Combining
relative squared loss or time
1
0.8
0.6
0.4
0.2
0
1
2
4
8
shard count
r. squared loss
r. time

Final thoughts

We compete with & beat multicore parallelized online gradient descent.

This general approach, unlike averaging approaches, is fully applicable to nonlinear systems.

Backprop & Delayed GD coming soon.

Code at: http://github.com/JohnLangford/vowpal_wabbit

Patches welcome.

Tutorial@2pm (No skiing for me!)

Some further discussion @ http://hunch.net