

Large Scale Machine Learning

Part 2: Distributed Logistic Regression

Previously on Large Scale ML

- Definition of Large Scale ML
- Overview of Large Scale ML software and hardware paradigms
- Large Scale ML on your machine

This episode

- Distributed Logistic Regression
 - Synchronous Distributed Gradient Descent
 - Second order methods
- Other distribution strategies
 - Parameter Server
 - AllReduce

Distributed Computing

- Using multiple networked computation nodes (computers)
- Nodes communicate and coordinate their actions by passing data on the network
- Distributed ML
 - Designing algorithms that work efficiently on distributed systems

Logistic Regression

- We are going to use logistic regression as a show case for some of the techniques shown
 - Most of what we are going to say applied to most parametric models that can be optimized with Gradient Descent
- Logistic Regression is used everywhere
 - Most common classification algorithm, can be quite very flexible
 - Criteo trains thousands of LR models per day and uses them to do billions of predictions

Logistic Regression

- Let's start with a well known algorithm to show case the design ideas
- M data points $(x, y)_i - i = 1..M$
- Parametrized model function $f_{\theta}(x) = \hat{y}$
- Loss function
 - For each data point

$$l(y, \hat{y}) = l(y, f_{\theta}(x)) = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y})$$

- For the whole dataset

$$L(\theta) = \frac{1}{M} \sum_{i=1}^M l(y_i, \hat{y}_i)$$

- Objective

$$\operatorname{argmin}_{\theta} L(\theta)$$

Logistic Regression

- Gradient descent
 - Repeat until convergence

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(\theta)$$

- Stochastic gradient descent
 - Repeat until convergence
 - Sample random i from $\{1..M\}$

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla_{\theta} l(y_i, \widehat{y}_i)$$

Logistic Regression

- Mini-batch stochastic gradient descent
 - Repeat until convergence
 - Sample a batch of data of size b

$$L_b(\theta) = \frac{1}{b} \sum_{i=1}^b l(y_i, \hat{y}_i)$$

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla_{\theta} L_b(\theta)$$

- Stochastic and mini-batch gradient descent rely on

$$E(\nabla_{\theta} l(y_i, \hat{y}_i)) = E(\nabla_{\theta} L_b(\theta)) = \nabla_{\theta} L(\theta)$$

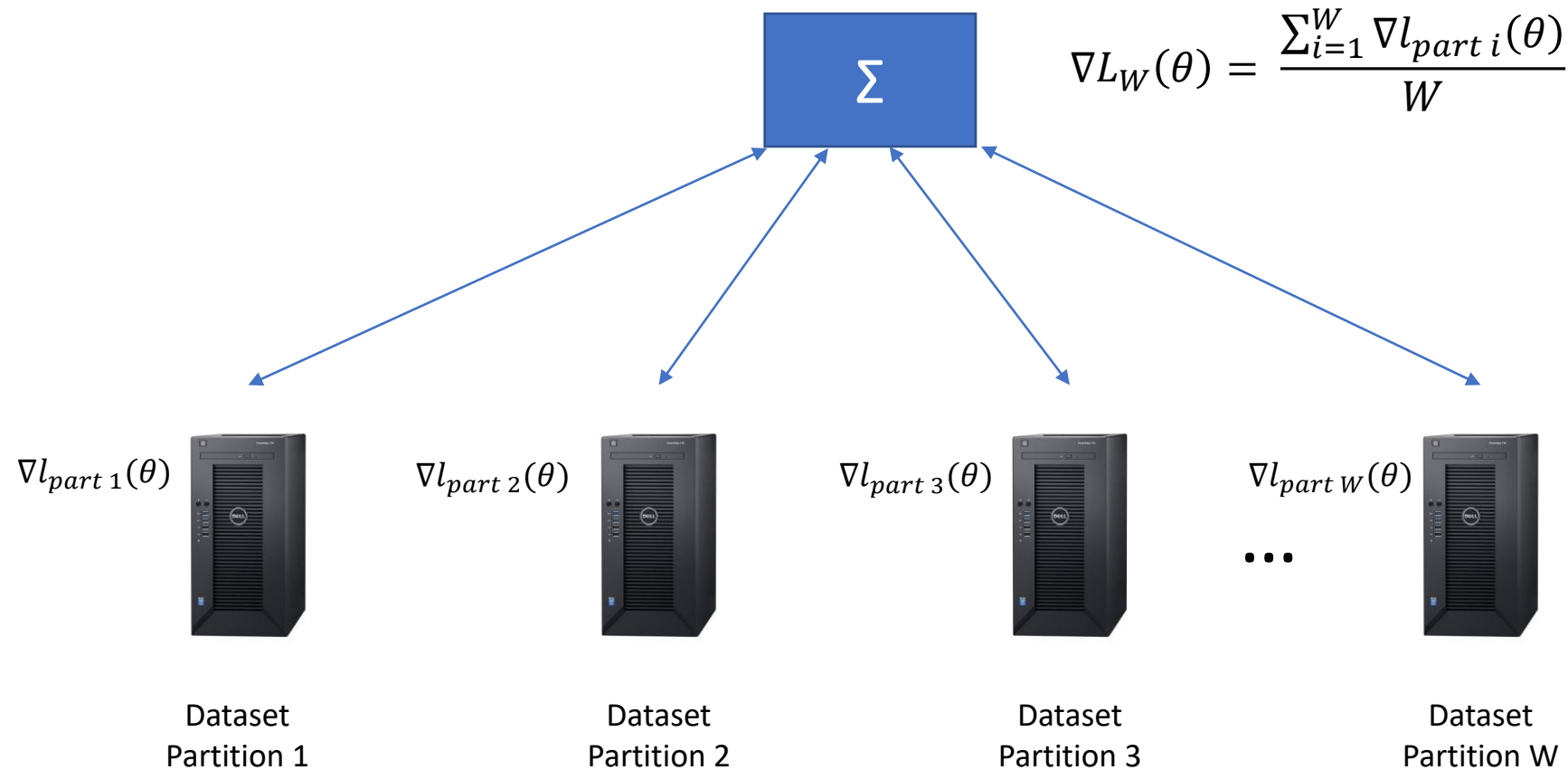
Distributed Gradient Descent 101

- W workers
- Each worker
 - Reads a partition of data
 - Computes a local gradient
 - Sends the gradient over to be aggregated
 - Model is updated and re-pushed to the workers
 - Rinse and repeat

Distributed Gradient Descent 101

- In mini-batch SGD gradients will only be computed on a batch of data
 - Each worker will process a small batch of data
- In Full Gradient Descent it will be computed on the whole dataset
 - Assuming n workers, each worker will process $1/n$ of the dataset

Distributed Gradient Descent 101

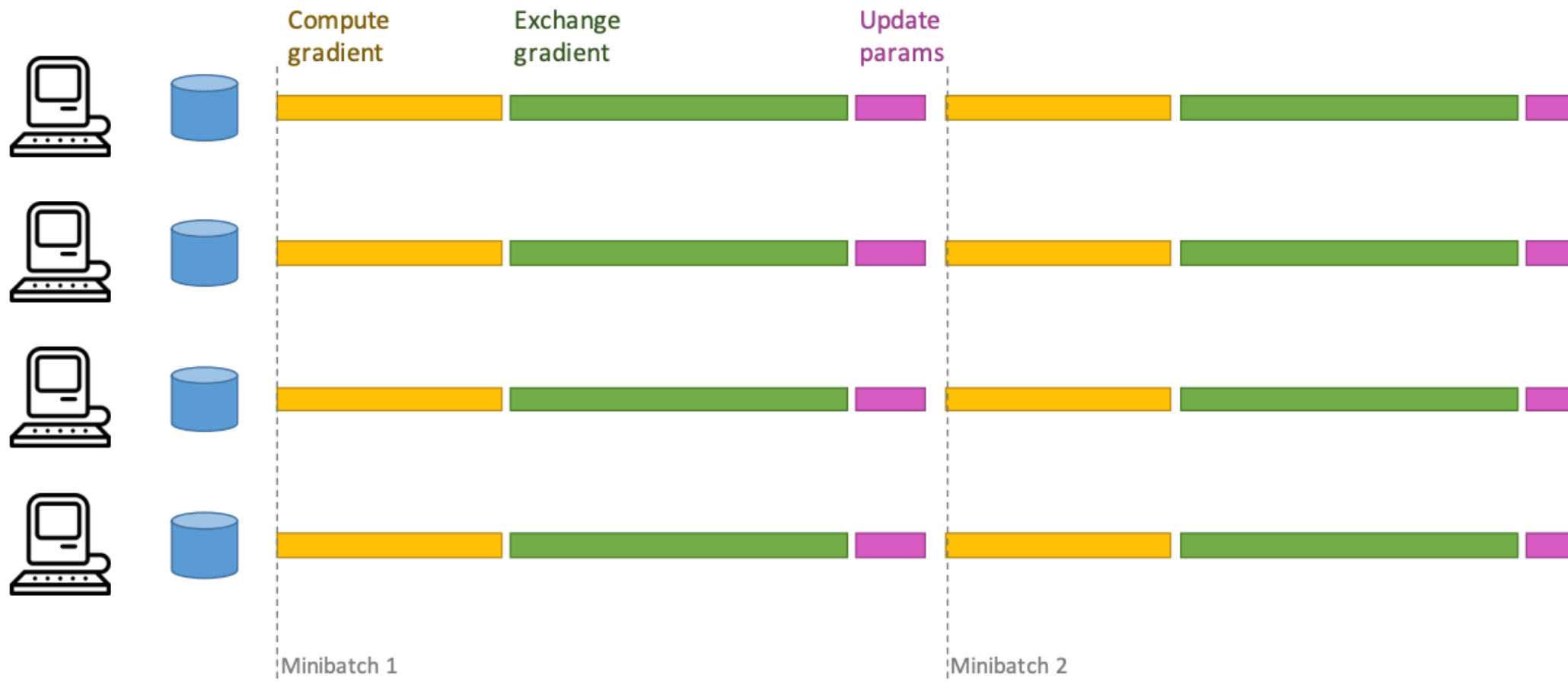


Distributed Gradient Descent 101

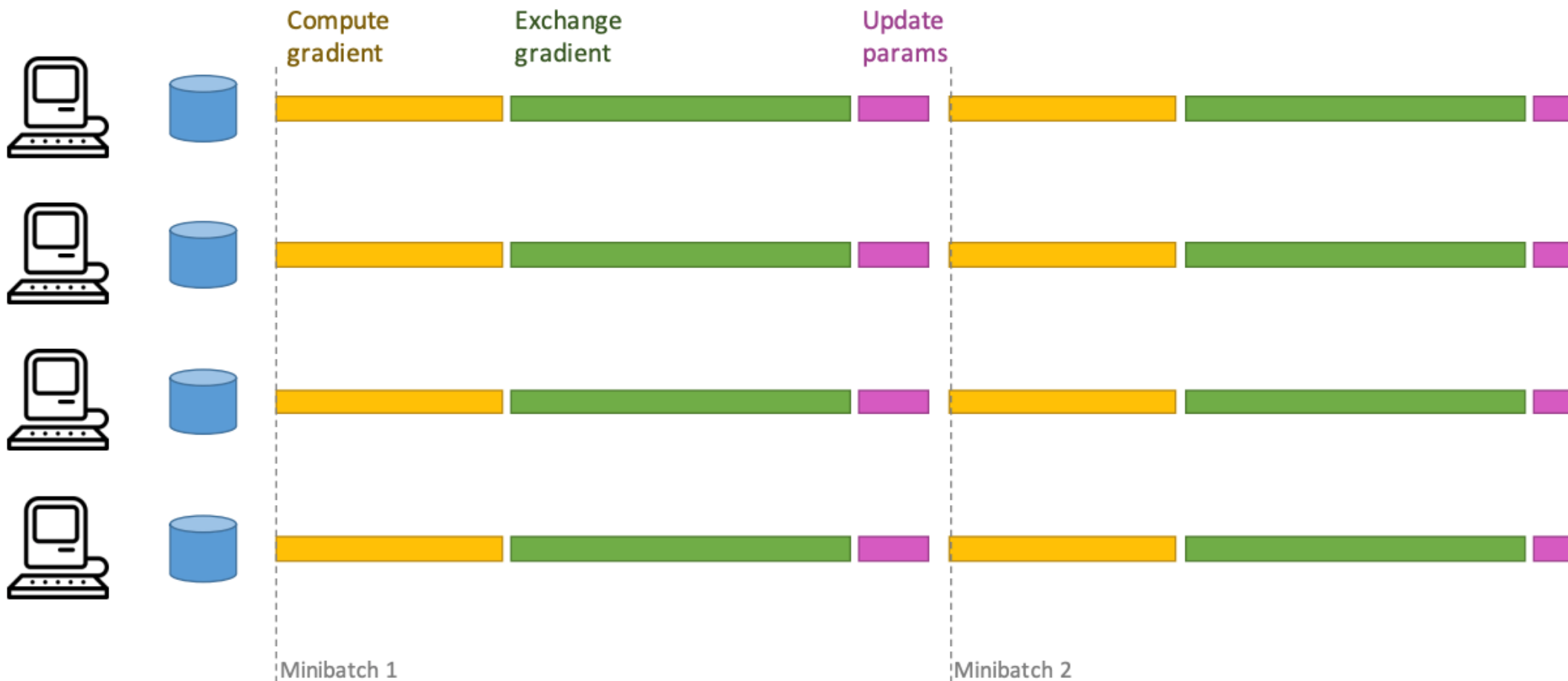
- Easy to implement a simple version in Spark

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

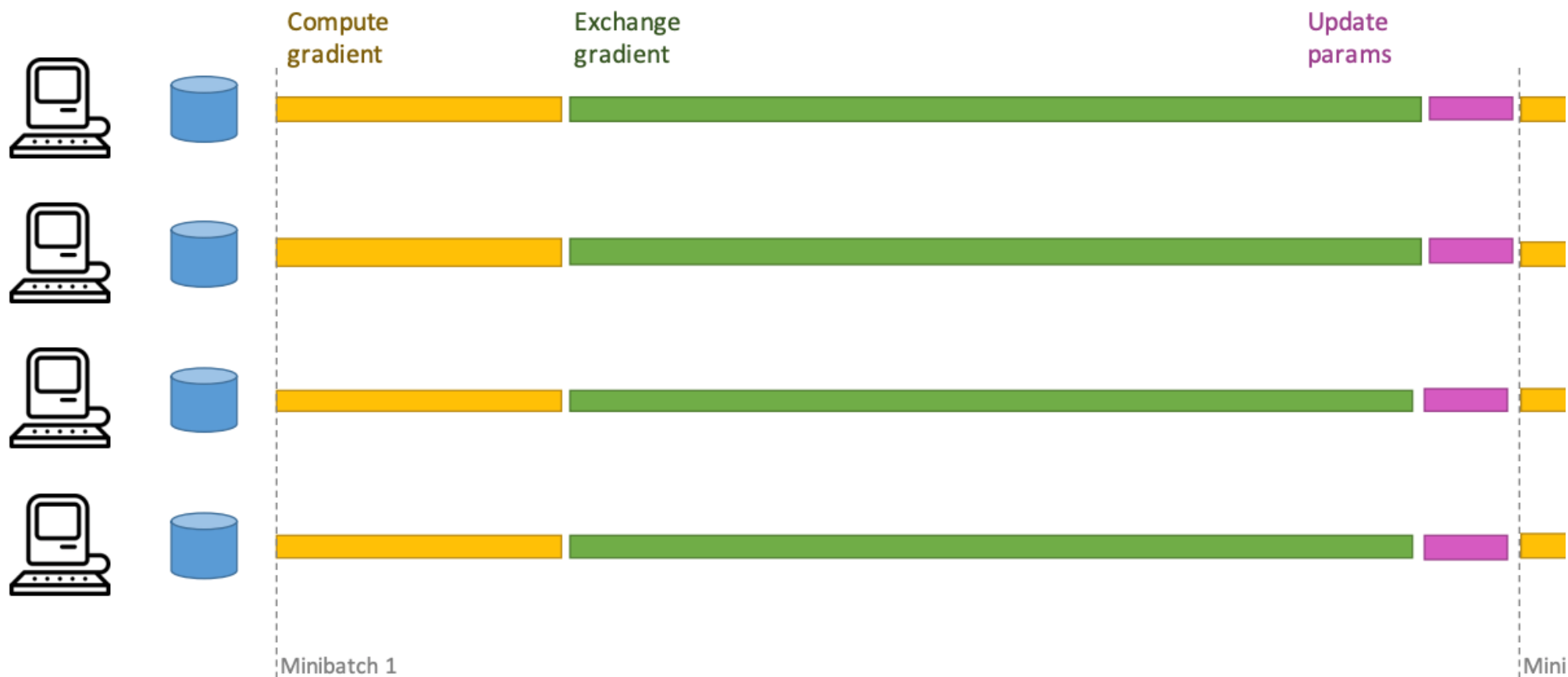
Synchronous Data Parallel SGD



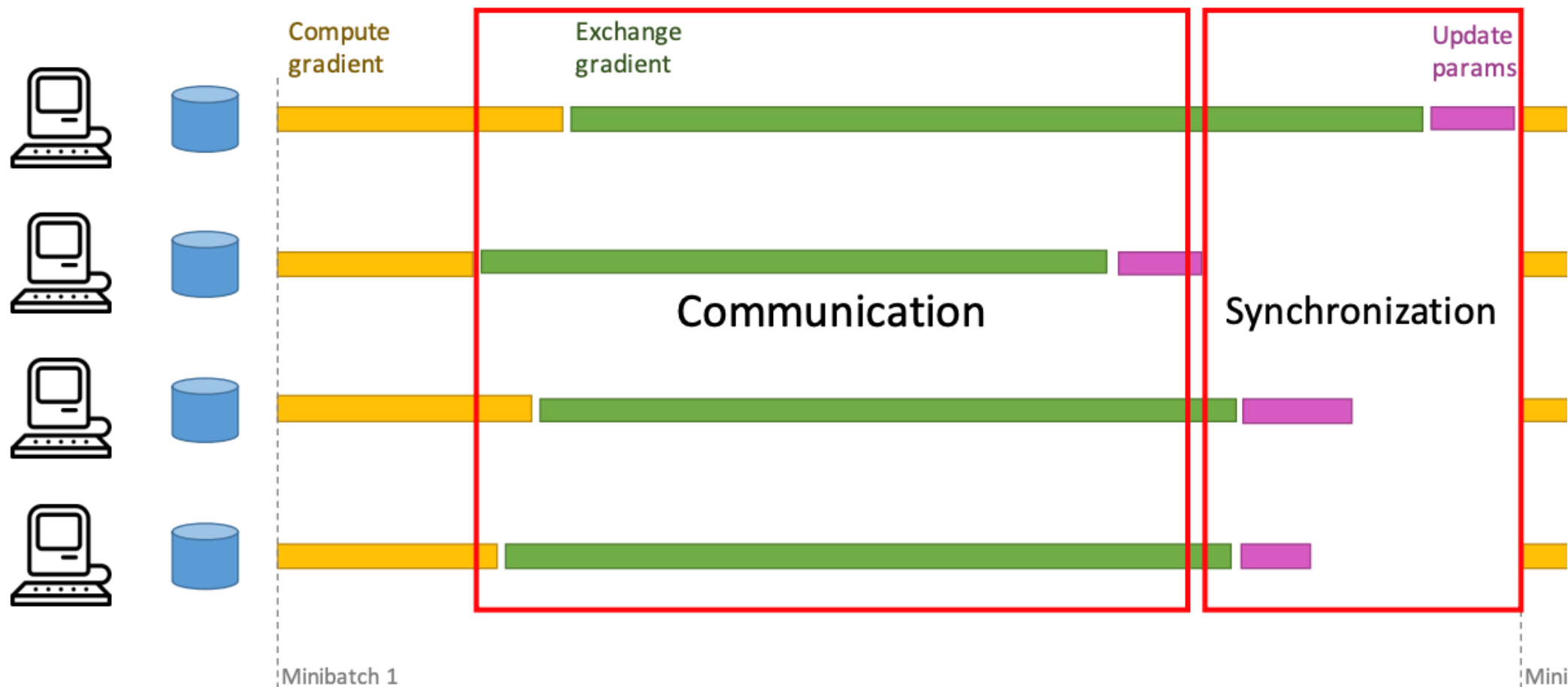
Synchronous Data Parallel SGD



Synchronous Data Parallel SGD



Synchronous Data Parallel SGD



Synchronous parallel SGD

- Objective: compute the gradient in parallel
- Two main overheads
 - Communication: sending gradient updates to the driver / other workers
 - Synchronization: waiting for all the workers to execute the current iteration
- This what is preventing from getting the theoretical speed up
 - With n workers
 - Gradient computation: T / n
 - Comm and sync cost: does not decrease with n but increase !

Communication cost

- Sparse model example
 - Total model weights M
 - But only $f \ll M$ non zero features per row
 - For the log-loss defined earlier, gradient will be sparse and needs to be represented by a sparse vector
- This is the key to efficiency
 - Gradient update is sparse + sparse operation

Why is the gradient sparse

$$\nabla_{\theta} l(y, \hat{y}) = - \nabla_{\theta} [y \cdot \log f_{\theta}(x) + (1 - y) \cdot \log(1 - f_{\theta}(x))]$$

$$\nabla_{\theta} l(y, \hat{y}) = \left(y - \frac{1}{1 + e^{-x^T \theta}} \right) x$$

$$\nabla_{\theta} l(y, \hat{y}) = \kappa x$$

Sparse feature vector = sparse gradient

Some bad news

- What about regularization ?
 - L2 Regularization term $\frac{1}{2} \lambda \theta^T \theta$

$$\nabla_{\theta} \frac{1}{2} \lambda \theta^T \theta = \lambda \theta$$

- The update rule becomes
 - $\theta \leftarrow \theta - \alpha (\kappa x + \lambda \theta)$
 - It's dense now ☹ no matter how sparse the feature vector is

Regularized SGD with Sparse Updates

- Let's concentrate on the first dimension of the model θ_0
- Suppose we have an update with the first dimension being zero

$$\theta_0 \leftarrow \theta_0 - \alpha (\kappa x_0 + \lambda \theta)$$

$$\theta_0 \leftarrow \theta_0 - \alpha \lambda \theta$$

$$\theta_0 \leftarrow (1 - \alpha \lambda) \theta_0$$

- After two updates

$$\theta_0 \leftarrow (1 - \alpha \lambda)^2 \theta_0$$

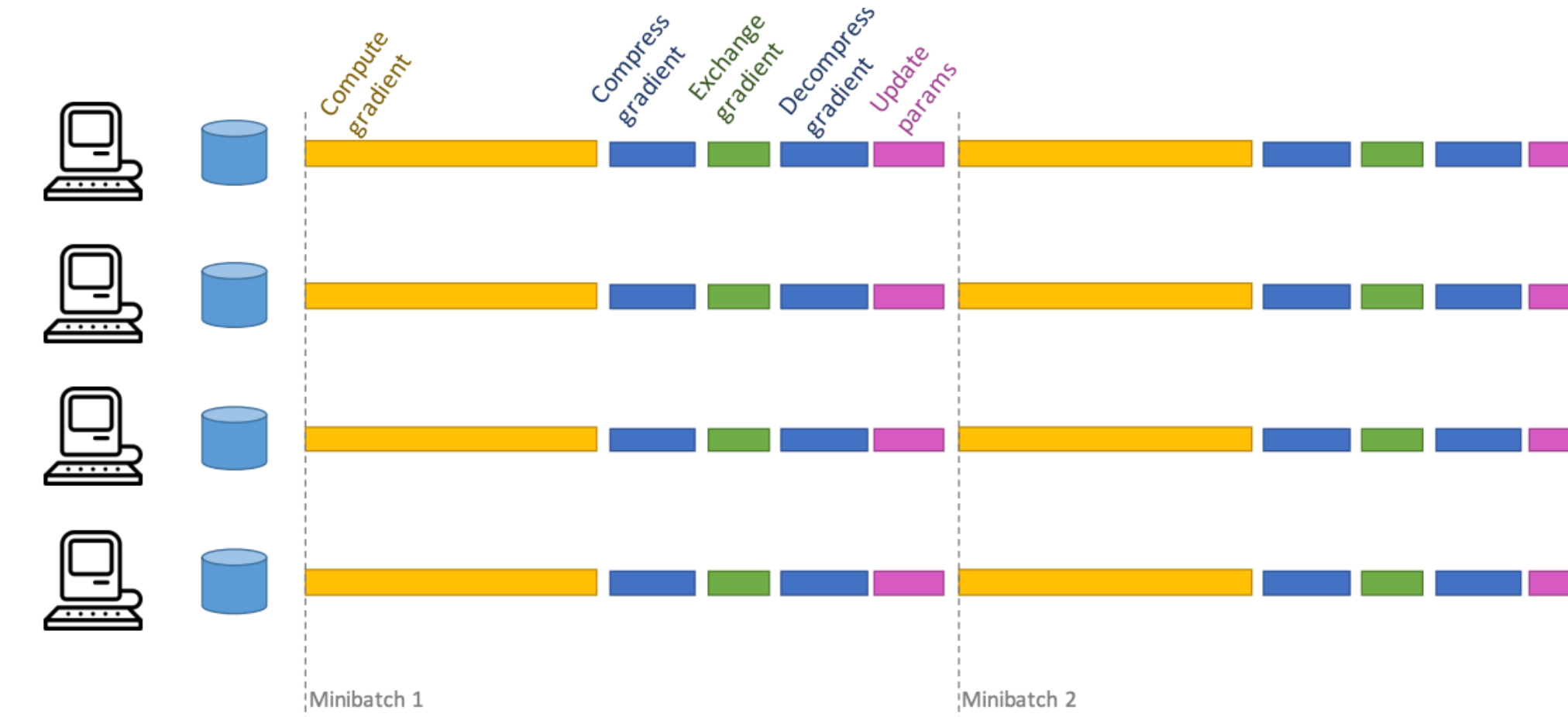
Regularized SGD with Sparse Updates

- We can then compress K computations

$$\theta_0 \leftarrow (1 - \alpha\lambda)^K \theta_0$$

- We remember how many steps we didn't update a particular dimension
- Catch up when needed

More general pattern for reducing communication cost

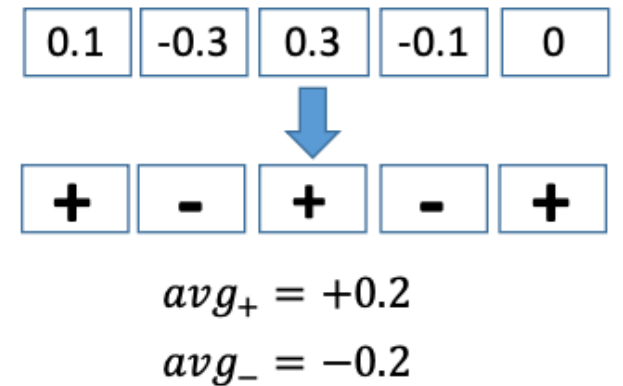


How can we compress a vector ?

- Lossy compression
- Quantization function
 - \mathbf{v} vector of real values

$$Q_i(v) = \begin{cases} avg_+ & \text{if } v_i \geq 0, \\ avg_- & \text{otherwise} \end{cases}$$

where $avg_+ = \text{mean}([v_i \text{ for } i: v_i \geq 0])$, $avg_- = \text{mean}([v_i \text{ for } i: v_i < 0])$



Why this shouldn't work

- Remember: Mini-batch gradient descent rely on

$$E(\nabla_{\theta} L_b(\theta)) = \nabla_{\theta} L(\theta)$$

- But

$$E(Q(\nabla_{\theta} L_b(\theta))) \neq E(\nabla_{\theta} L_b(\theta))$$

- We don't have an unbiased estimate anymore

Nice trick: Stochastic Quantization

- Quantization function

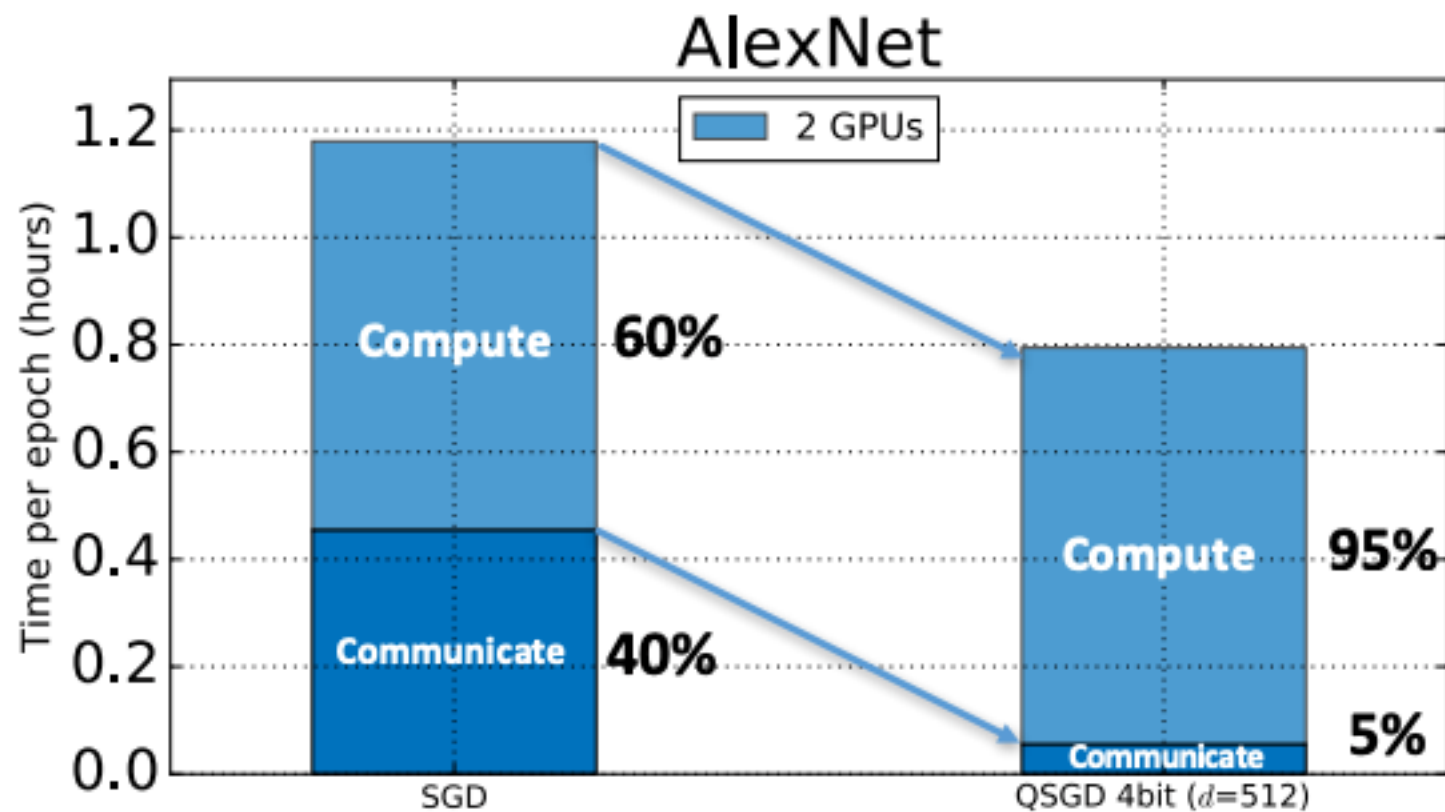
$$Q(v_i) = \|v\|_2 \cdot \text{sgn}(v_i) \cdot \xi_i(v_i)$$

where $\xi_i(v_i) = 1$ with probability $|v_i|/\|v\|_2$ and 0 otherwise.

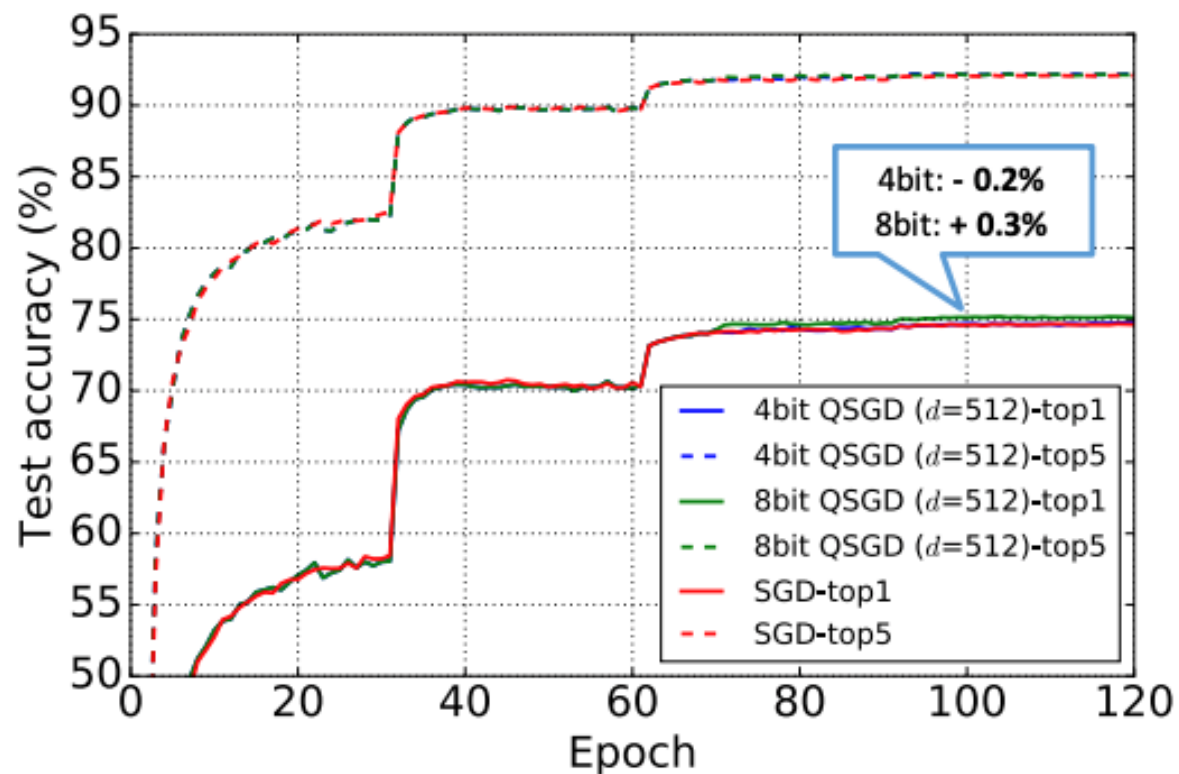
- Now it's unbiased

$$\mathbf{E}[Q[v_i]] = \|v\|_2 \cdot \text{sgn}(v_i) \cdot |v_i|/\|v\|_2 = \text{sgn}(v_i) \cdot |v_i|$$

Nice trick: Stochastic Quantization



Nice trick: Stochastic Quantization



ResNet50 on ImageNet
8 GPU nodes

How about synchronization ?

- We will talk about it a bit later but here is simple solution
 - Drop the gradients of slow workers by imposing a timeout to the computation

