PART B. GEAR WORKSHOP II
SCALABLE COMPUTING MODELS

FAQs
- Programming Assignment 2
- 5:00PM March 29 (Adjusted)
- Team submission
- Scores of the workshop critical reviews are available
- Scores of the TP1 are available

Today’s topics
- Optimization for ML
- Distributed optimization for ML

Scalable computing challenges in Analytics
- Effective data access
  - CD-ROM drive, 200 milliseconds
  - Hard drive, 15 milliseconds
  - Random Access Memory (RAM), 70 nanosecond

- Effective computing
  - Optimization of the analytics algorithms

http://www.cs.colostate.edu/~cs535
What is "optimization"?
- Finding one or more minimizer of a function subject to constraints
- Most of machine learning problems are optimization problems in computing
- For example, in k-Means clustering
  - Looks for k-clusters in which each observation belongs to the cluster with the nearest mean
  - In this case, "optimization" is the process to find:
    \[ \text{arg}\min_{\mu_1, \ldots, \mu_k} \sum_{i=1}^{n} \| x_i - \mu_k \| \]

Sometimes, optimization is NOT straightforward
- Minimize \( f(x) \)

Convex optimization
- Convex function
  - A function \( f: \mathbb{R}^n \rightarrow \mathbb{R} \) is convex if for every \( x, y \in \text{dom} \ f \) and any \( a \in [0,1] \),
    \[ f(ax + (1-a)y) \leq af(x) + (1-a)f(y) \]

Convexity optimization
- Theorem
  - If \( x \) is a local minimizer of a convex optimization problem, it is a global minimizer

Optimizations in Spark
- Spark supports
  - Gradient descent
  - Stochastic gradient descent (SGD)
  - Limited-memory BFGS (L-BFGS)

GEAR Workshop II | Scalable Computing Models
Optimization Algorithms
1. Gradient Descent
Gradient Descent

- The simplest method to solve optimization problems
  - Achieve $min_{w \in W} f(w)$
  - Suitable for large scale and distributed computation
- Finds a local minimum of a function by iteratively taking steps in the direction of steepest descent
  - Negative of the derivative (gradient) of the function at the current point

Fitting the linear regression model

- Linear regression model $h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \ldots$
- Example: Predict the student’s science score based on the math score $h(x) = \theta_0 + \theta_1 x$

Objective function (Cost function)

- For a given training set, how do we pick, or learn the parameter $\theta$?
  - Make $h(x)$ close to $y$
    - Make your prediction close to the real observation
  - We define the objective (cost) function
    - Using Mean Squared Error and multiplying $\frac{1}{2}$ for convenience

\[
J(\theta) = \frac{1}{2m} \sum_{i=0}^{m} (h_\theta(x^i) - y^i)^2
\]

Minimization problem

- We have a function $f(\theta_0, \theta_1)$
- We want to find $\min_{\theta_0, \theta_1} f(\theta_0, \theta_1)$
- Goal: Find parameters to minimize the cost (output of the objective function)

- Outline of our approach:
  - Start with some $\theta_0, \theta_1$
  - Keep changing $\theta_0, \theta_1$ to reduce $J(\theta_0, \theta_1)$ until we end up at a minimum

Gradient descent algorithm

Repeat until convergence:

\[
\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)
\]  
(for $j=0$ and $j=1$)

\[
\begin{align*}
\theta_0 &:= \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\
\theta_1 &:= \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)
\end{align*}
\]

Decreasing and increasing $\theta_1$
Decreasing $\theta_1$

- Positive slope

$\theta_1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

Increasing $\theta_1$

- Negative slope

$\theta_1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

Learning rates

- If $\alpha$ is too small, gradient descent can be slow

- If $\alpha$ is too large, gradient descent can overshoot the minimum

Stochastic Gradient Descent (SGD)

- Batch methods

  - Full training set to compute the next update to parameters at each iteration tend to converge very well

  - Advantage
    - Straight forward to get working provided a good off the shelf implementation
    - Very few hyper-parameters to tune

  - Disadvantages
    - Computing the cost and gradient for the entire training set can be very slow
    - Intractable on a single machine if the dataset is too big to fit in main memory
    - No easy way to incorporate new data in an ‘online’

http://www.cs.colostate.edu/~cs535
Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent (SGD)
  - Follows the negative gradient of the objective after seeing only a single or a few training examples.
  - The use of SGD in the neural network setting is motivated by the high cost of running backpropagation over the full training set.
  - Fast convergence.

SGD used in supervised machine learning with Spark

- The loss is written as an average of the individual losses coming from each data point.
- A stochastic subgradient is a randomized choice of a vector.
  - Selects one datapoint \( i \in [l, a] \) uniformly at random, to obtain a stochastic subgradient of \( f(w) \) with respect to \( w \) as follows:
  \[ f'_w = w + \sqrt{\frac{a}{b}} \cdot \mathbf{r}(w) \]
  - Where \( \mathbf{r}(w) \) is a subgradient of the part of the loss function determined by \( i \)-th data point
  - \( f'_w \) is a subgradient of the regularizer \( R(w) \)

SGD used in supervised machine learning with Spark

- Here, the loss part of the optimization problem
  \[ \frac{1}{a} \sum_{i=1}^{a} L(w; x_i, y_i) \]
  - Therefore, the true subgradient:
  \[ \frac{1}{a} \sum_{i=1}^{a} \nabla L_i \]
  - This will require access to the full dataset.

http://www.cs.colostate.edu/~cs535
SGD used in supervised machine learning with Spark

- $|S|$: size of the sampled subset
- $|S| = \text{miniBatchFraction} * n$

- If $|S| = 1$, it is equivalent to the standard SGD
  - In that case, the step direction depends from the uniformly random sampling of the point

Limited-memory BFGS (L-BFGS)

- BFGS (Broyden–Fletcher–Goldfarb–Shanno algorithm)
- Iterative method for solving unconstrained nonlinear optimization problems
- Objective functions are non-linear
- A type of quasi-Newton methods

Limited-memory BFGS (L-BFGS)

- L-BFGS algorithm approximates BFGS algorithm using limited amount of memory
- Stores last $M$ value/gradient pairs and uses them to build positive definite Hessian approximation
  - This approximate Hessian matrix is used to make quasi-Newton step
- If quasi-Newton step does not lead to sufficient decrease of the value/gradient,
  - The algorithm makes line search along direction of this step
- Only last $M$ function/gradient pairs are used
  - $M$ is moderate number smaller than problem size $N$, often as small as 3-10
  - Very cheap iterations, which cost just $O(M)$ operations.

Choosing an optimization method

- Linear methods use optimization internally
  - Linear SVM, logistic regression, regressions (Linear least squares, Lasso)
- Some linear methods in spark.mllib support both SGD and L-BFGS
- Different optimization methods can have different convergence guarantees
  - depending on the properties of the objective function
- In general, when L-BFGS is available, we recommend using it instead of SGD since L-BFGS tends to converge faster (in fewer iterations)
GD and SGD: Implementation in MLlib

- Gradient descent methods including stochastic sub-gradient descent (SGD) as included as a low-level primitive in MLlib

Vector optimize(RDD<scala.Tuple2<Object, Vector>>, data.Vector initialWeights)

- The SGD class GradientDescent sets the following parameters:
  - Gradient
    - A class that computes the stochastic gradient of the function being optimized, i.e., with respect to a single training example, at the current parameter value
    - MLlib includes gradient classes for common loss functions
      - e.g., hinge, logistic, least-squares
    - The gradient class takes as input a training example, its label, and the current parameter value.

Updater

- A class that performs the actual gradient descent step
  - i.e. updating the weights in each iteration, for a given gradient of the loss part
  - The updater is also responsible to perform the update from the regularization part.
  - MLlib includes updaters for cases without regularization, as well as L1 and L2 regularizers

- stepSize
  - A scalar value denoting the initial step size for gradient descent. All updaters in MLlib use a step size at the t-th step equal to \( \frac{\text{stepSize}}{t} \)

- numIterations
  - The number of iterations to run.

This material is built based on

- Jeffrey Dean and Greg S. Corrado and Rajat Monga and Kai Chen and Matthieu Devlin and Quoc V. Le and Mark Z. Mao and Marc'Aurelio Ranzato and Andrew Senior and Paul Tucker and Ke Yang and Andrew Y. Ng, Large Scale Distributed Deep Networks, 2012, NIPS

- Martin Abadi and Paul Barham and Jianmin Chen and Zhifeng Chen and Andy Davis and Jeffrey Dean and Matthieu Devin and Sanjay Ghemawat and Geoffrey Irving and Michael Isard and Martin Juefe and Josh Levenberg and Rajat Monga and Sherry Moore and Derek G. Murray and Benoit Steiner and Paul Tucker and Vijay Vasudevan and Pete Warden and Martin Wicke and Yuan Yu and Xiaoqiang Zheng, TensorFlow: A system for large-scale machine learning, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016

http://www.cs.colostate.edu/~cs535
### Distributed optimization and Synchronization schemes

- Large dataset for training
  - Not only training within a single instance of model
  - Distribute training across multiple model instances
- E.g. SGD is one of the most popular optimization procedure for training deep neural network
  - The traditional formulation of SGD is inherently sequential
  - The time required to move through the data in an entirely serial fashion is prohibitive

### Overview of DistBelief

- Large dataset for training
  - Not only training within a single instance of model
  - Distribute training across multiple model instances
- Downpour SGD
  - Sandblaster L-BFGS
  - Take advantage of the distributed computation within each individual replica
  - Tolerate variance in the processing speed of different model replicas
  - Even the failure of model replicas which may be taken offline or restarted at random

### Downpour SGD

- Objectives of Downpour SGD
  - Applying SGD to extremely large datasets
  - A variant of asynchronous stochastic gradient descent
    - Uses multiple replicas of a single DistBelief model
  - This approach is asynchronous in two distinct aspects
    - The model replicas run independently of each other
    - The parameter server shards also run independently of one another

### Before processing each mini-batch, a model replica asks the parameter server service for an updated copy of its model parameters

- Each machine needs to communicate with just the subset of parameter server shards that hold the model parameters relevant to its partition
  - After receiving an updated copy of its parameters, the DistBelief model replica processes a mini-batch of data to compute a parameter gradient
  - Sends the gradient to the parameter server
  - Then, applies the gradient to the current value of the model parameters

---

http://www.cs.colostate.edu/~cs535
Reducing the communication overhead of Downpour SGD

- Limit each model replica to request updated parameters only every \( n \) \( \text{fetch} \) steps and send updated gradient values only every \( n \) \( \text{push} \) steps.
- Traditional distributed SGD: \( \text{fetch} = \text{push} = 1 \)

Fault tolerance with Downpour SGD

- More robust to machines failures
- Synchronous SGD with failures:
  - If one machine fails, entire training process is delayed
- Asynchronous SGD with failures:
  - If one machine fails, other model replica will continue processing training and updating model parameters

Failures vs. Stochasticity

- No guarantee that:
  - There were the same number of updates in each shard of parameters
  - There were updates applied in the same order
  - There are subtle inconsistencies in the timestamps of parameters
  - Relaxing consistency requirement is effective to enhance stochasticity

Improving Downpour SGD with Adagrad

- Adagrad: Adaptive learning rate procedure
- Let \( \eta_i \) be the learning rate of the \( i \)-th parameter at iteration \( K \) and \( \Delta w_i \) its gradient
- We set: \( \eta_i = \frac{\gamma \sqrt{\sum_j \Delta w_j^2}}{\sqrt{\sum_j \Delta w_j^2} + \Delta w_i} \)
- These learning rates are computed only from the summed squared gradients of each parameter:
  - \( \rightarrow \) Adagrad can be implemented locally within each parameter server shard

- The value of \( \gamma \):
  - The constant scaling factor for all learning rates
  - Generally larger than the best fixed learning rate used without Adagrad

- The use of Adagrad:
  - Extends the maximum number of model replicas that can productively work simultaneously
  - Eliminates stability concerns
  - Combined with a practice of "warmstarting" model training with only a single model replica before unleashing the other replicas