COMP6208: Advanced Machine Learning

Machine Learning with Big Data

Jonathon Hare
jsh2@ecs.soton.ac.uk

Contents

- Going to look at two case-studies
  - looking at how we can make machine-learning algorithms work on "**big data**"
    - **"data that is too large to be effectively stored on a single machine or processed in a time-efficient manner by a single machine."**
- Specifically:
  - **K-Means clustering** using Hadoop
  - **Stochastic Gradient Descent** using Spark

Distributing K-Means with MapReduce

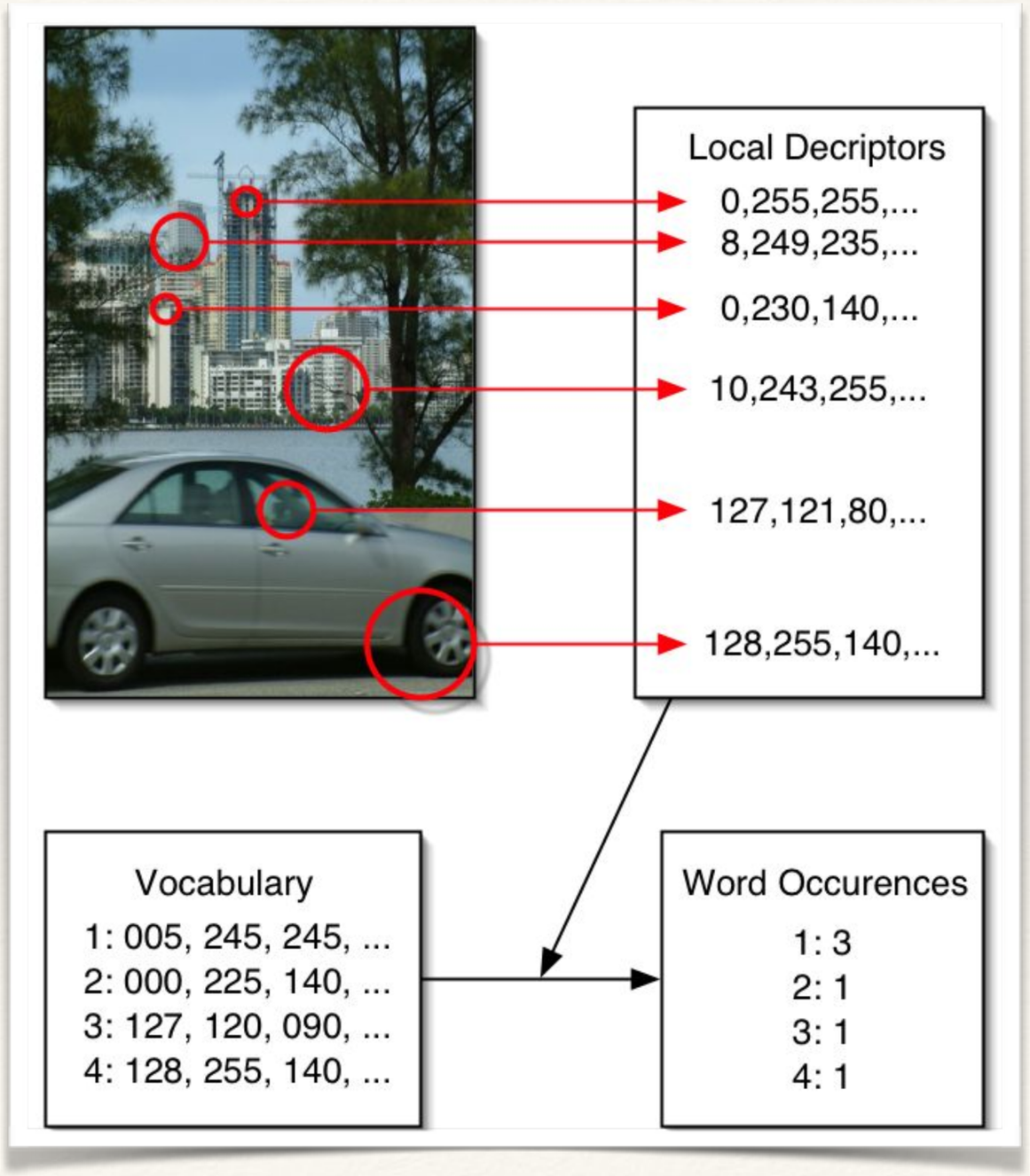
Recap: K-Means Clustering

- K-Means is a classic clustering algorithm for grouping data into *K* groups with each group represented by a *centroid*:
  - The value of *K* is chosen
  - *K* initial cluster centres are chosen
  - Then the following process is performed iteratively until the centroids don’t move between iterations (or max allowed iterations is reached):
    - Each point is assigned to its closest centroid
    - The centroid is recomputed as the mean of all the points assigned to it. If the centroid has no points assigned it is randomly re-initialised to a new point.
  - The final clusters are created by assigning all points to their nearest centroid.

K-Means Demo

K-Means with Big Data

- Let's assume that we're building a visual image search system for millions of images (ala Google "Goggles").
  - We'd do this by decomposing an each image into a histogram of visual words that can be effectively indexed.
  - A visual word vocabulary can be created by clustering SIFT features…



K-Means with Big Data

- Typical image might have ~3000 SIFT Features
  - Each one is a 128-dimensional vector
  - Assuming double-precision fp (64bits/element) each image has ~3MB of features
- This kind of image search requires **big** vocabularies
  - Say 1 million
- Assume we've got 1 million images:
  - We want to cluster 3B features into 1M clusters!
  - Features take up **~3TB** storage; centroids **~1GB**.

How can K-Means be parallelised?

- Fundamentally the algorithm has two steps that **repeat**:
  - **Centroid assignment**
  - **Centroid update**
- These are **independent**, and both can be parallelised

Parallelised Assignment

- During assignment the centroids are static, so given enough processing resource each vector could be assigned in parallel.
  - The process of searching for the closest centroid can also be parallelised…
    - …by computing distance to each centroid in parallel.

- This looks a lot like a *Map* operation:
  `Map(<null, vector>) -> <centroid, vector>`

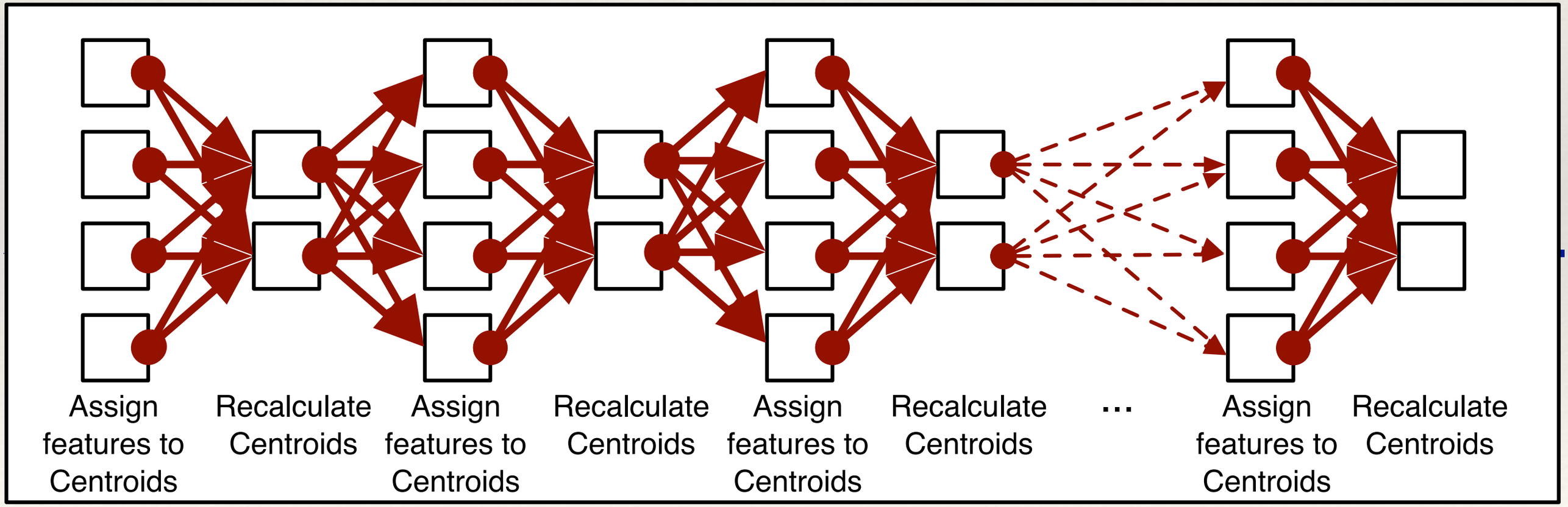
Parallelised Update

- Each centroid update is independent of the others, so they can also be performed in parallel.
  - Computing the new centroid is a matter of accumulating the corresponding elements of all the assigned vectors, and dividing by the number of assigned vectors.
    - This could (*potentially*) be parallelised on a per-element basis (although there might be practical limitations).

- This looks a lot like a *Reduce* operation:
  `Reduce(<centroid, list(vector)>) -> <null, centroid>`

MapReduce K-Means

- K-Means ***almost*** fits the MapReduce model



MapReduce K-Means Caveats

- Map function needs to have **access to the centroids** in addition to its normal inputs
- Can use a fixed number of iterations
  - but if we want to stop on **convergence**, the Reduce functions need to be able to **communicate** this

Basic Hadoop MapReduce K-Means

- Hadoop **Mapper** can load the current centroids from HDFS into memory during setup.
  - Allows centroids to be used over all vectors belonging to the InputSplit
- Map function doesn't need to actually output the centroid vector
  - an ID is fine (and much smaller)
  - **Unfortunately, you're still emitting every input vector!**
    - **This is less of an issue because the cost of data transfer is likely to be outweighed by the cost of finding the closest centroid (assuming many centroids).**

Basic Hadoop MapReduce K-Means

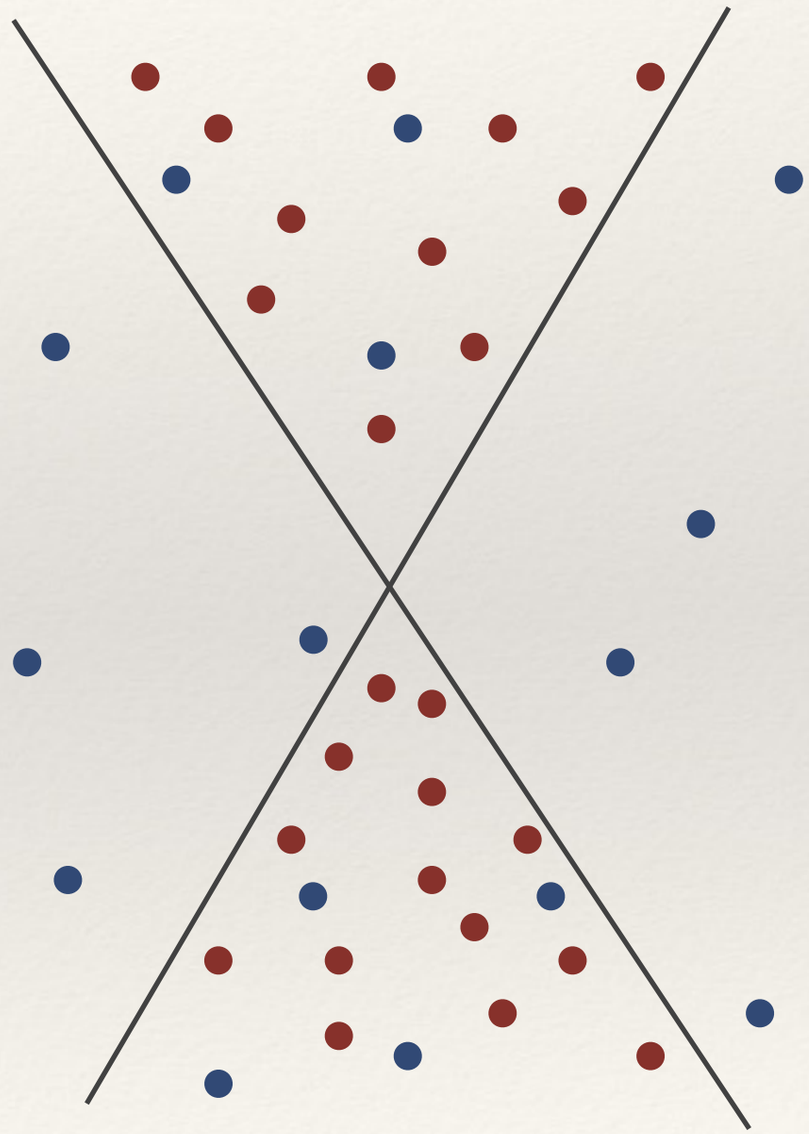
- The reducer doesn't actually care about the centroid vector (hence use of IDs in the map isn't a problem)
  - just needs to know those vectors that have been assigned to a centroid in order to compute the new centroid vector
- For convergence checking, we could have a single **Reducer** that aggregates convergence of all reduces, or we could write convergence state to HDFS.

K-Means Initialisation

- To start K-Means, you need to have an initial set of *K* centroids….
- How can these be generated efficiently and effectively?

Random Generated Centroids

- In some cases, you can just generate some random vectors as a starting point.
  - This only works well if the vectors you generate cover the space of the data vectors.
  - In the case of SIFT features, the space, although bounded, is highly **non-uniform** and contains many areas with **very low density**, and others with very high density.



Random Sampled Centroids

- Choosing the centroids by randomly uniformly sampling the actual data is often a good starting point that works well in practice.
- How can we efficiently sample in our big data scenario?
  - We don't necessarily know exactly how many features we have
    - Finding out is relatively expensive because it means reading through all the data.

Hadoop MapReduce Sampling

Synopsis: sample *K* items from an unknown amount of data

Input: collection of files containing feature-vectors; read one vector at a time.

Mapper:

Map function:

Input: <*Key:* null, *Value:* vector>

Outputs: *nothing*

Read the input vectors into the mapper memory.

Output on shutdown: <*Key:* null, *Value:* vector>

Output K uniformly randomly sampled vectors on shutdown of the mapper.

Reduce function:

Input: <*Key:* centroid_id, *Value:* [vectors of the items assigned to the cluster]>

Outputs: <*Key:* Null, *Value:* centroid vector>

Average the vectors to compute the new centroid, and emit this with a null key.

Canopy Clustering

- Fast algorithm to compute **approximate** clusters based on a **pair of distance thresholds**.
  - Can be implemented in a single Hadoop MapReduce round:
    - Each Mapper processes a subset of the data and uses the canopy algorithm to produce a set of clusters ("canopies").
    - Canopy clustering is applied to the centroids of all the mapper canopies (in a single Reducer) to produce the final set of initial centroids.

Better Hadoop MapReduce K-Means

- We've dealt with initialisation and a basic Hadoop MapReduce K-Means implementation, but there are still two coupled problems:
- Finding the closest centroid (nearest-neighbour search) is slow.
- Every round of maps is emitting all the vectors to the reducers, resulting in massive amounts of data transfer.

Efficient Neighbour Search

- Rather than brute-force search (computing the distance of every point to every centroid), we can use an **approximate** method.
- for example, an ensemble of KD-Trees works well for SIFT-like vectors.
  - Reduces computation from O(NK) to O(N log(K))

Reducing Data Transfer

- The Hadoop MapReduce framework has an additional concept called a **Combiner**.
  - Combiners aim to reduce the data sent to the reducer by performing aggregation of data before it is sent to the reducer.
  - A combiner behaves like a reducer that runs on the data output from **mappers on a single machine**.
  - Because the combiner and mappers run on the same machine, there is no network I/O.

Reducing Data Transfer

- We can modify the MapReduce K-Means implementation to use a Combiner
  - Output average vector and the number of samples used to produce it:

```
Map(<null, vector>) -> <centroidID, <vector, 1>>

Combine(<centroidID, list(<vector, 1>)>) ->
   <centroidID, <local_centroid_vector, count>>

Reduce(<centroidID, list(<vector, count>)>) ->
   <null, centroid_vector>
```

Distributed Stochastic Gradient Descent with Spark

Gradient Descent

- Gradient Descent is a set of common **first order optimisation algorithms**.
- Attempts to find a (local) minimum of a **differentiable function.**
- Take steps proportional to the **negative gradient** of the function at the **current point**.

Machine Learning

- In machine learning we consider the problem of minimising an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w)$$

where w is the parameter vector that you're trying to estimate, and each summand function Q_i is associated with the i-th observation of the training set.

Batch Gradient Descent (BGD)

- Each iteration of optimisation has the following form:

$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w)$$

learning rate

- The summation can be performed in parallel

Example: Linear Regression

- Problem: fit a (straight) line ($y=w_0x+w_1$) to some data, minimising the error between the points and the line.
- Objective function is:

$$Q(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (w_0x_i + w_1))^2$$

- Or, in terms of the summand function:

$$Q_i(w) = \frac{1}{N}(y_i - (w_0x_i + w_1))^2$$

Example: Linear Regression...

- Differentiating w.r.t. w gives:

$$\frac{\delta Q_i(w)}{\delta w_0} = \frac{-2x_i(y_i - (w_0 x_i + w_1))}{N}$$

$$\frac{\delta Q_i(w)}{\delta w_1} = \frac{-2(y_i - (w_0 x_i + w_1))}{N}$$

- So we now have a simple way to iteratively compute an estimate for w given some data

BGD Demo

Stochastic Gradient Descent (SGD)

- BGD becomes very expensive with large datasets
  - Each iteration has to sum over all summand function gradients.
- SGD uses only a single observation to estimate the gradient at each step:

$$w := w - \alpha \nabla Q_i(w)$$

- This scales better, although doesn't converge as smoothly
  - Unfortunately, it's not easy to parallelise

SGD Demo

Mini-Batch SGD (MBSGD)

- MBSGD is a compromise between BGD and SGD
  - It uses a small number of observations at each iteration
    - The small sample is called a “mini-batch”
- The gradients of the summand for each observation in the mini-batches can be computed in parallel.
- Convergence is smoother than SGD

MBSGD Demo

Why is this useful?

- You've used Gradient Descent before:
  - The backpropagation algorithm!
- Directly applicable to many ML approaches:
  - Training SVMs
  - Training Neural Nets (inc. Deep Learning)
  - Matrix factorisation
    - "Funk SVD" used in Netflix
  - …

Big Data Gradient Descent

- Assume we have a **big** (distributed) dataset of observations and want to perform some kind of ML using gradient descent.
  - Mini-batches could be formed making use of data locality to allow for data-local processing.
- But, processing of mini-batches is sequential:
  - process batch -> update params -> process batch -> update params -> …
  - Not making effective use of computational resources

Downpour SGD

- Approach developed by google for training very large deep NNs using SGD
- Distributed processing based on a master-worker model.
  - Master maintains current parameters, and updates them on receipt of gradients from a worker.
  - Workers compute gradients of mini-batches

Downpour SGD

- Before starting to compute the gradients, each worker asks the master for the current parameters.
  - Gradients are returned to the master as soon as the worker has finished, and the master immediately applies them to update the parameters (using **Adagrad** for learning rates).
    - Obviously this means that many of the workers will be using out-of-date parameters.
      - Strangely enough, this doesn't seem to matter!

Downpour Demo

Downpour SGD in Spark

- Implementation in Spark is trivial:
  - Dataset represented by an **RDD**
  - The "master" is the node on which you run the program; it sets up a server that provides methods to get the current parameters and update the parameters with new gradients.
    - Communications could use anything: HTTP, RMI, even MPI…
    - The update method takes a gradient vector and uses it to alter the parameters of the model.

Downpour SGD in Spark

- The master then uses the **mapPartitions action** with a function that:
  - Fetches the current parameters from the master
  - Computes the gradients of each mini-batch (the mini-batch being defined as a RDD partition – that is basically equivalent to a block of the underlying file on HDFS).
  - Sends a message to the master informing it of the computed gradients.

Summary

- Machine learning with big data is **challenging**.
  - Many **considerations** need to be taken into account.
    - **Minimising data transfer; optimising computation.**
  - Hadoop-MapReduce and Spark provide programming frameworks in which you can implement different types of algorithms.