LECTURE 8: SCALABLE ALGORITHMS

EKATERINA MURAVLEVA

OVERVIEW OF THE COURSE

Lecture 1: General course information, CRISP-DM methodology

Lecture 2: Supervised learning/unsupervised learning. Classification/regression problems. Accuracy metrics (precision, recall, ROC-AUC scores). Concept of loss functions, overfitting / underfitting.

Lecture 3: Classical ML: Linear regression, logistic regression, support vector machine

Lecture 4: Classical ML: Decision trees, random forests, boosting.

Lecture 5: Classical ML: Dimensionality reduction: linear, non-linear methods.

Lecture 6: Classical ML: Clustering methods

Lecture 7: Basic neural networks

Lecture 8: Scalable algorithms

RECAP OF LECTURE 7

- Fully connected neural network
- Learning
- Backpropagation
- Types of NN architectures (brief summary)

PLAN OF LECTURE 8

Scalable algorithms

Distributed training

Randomized PCA

Approximate nearest neighbors

Incremental learning

SCALABLE ALGORITHMS

— Scalable algorithms in machine learning refer to the ability of an algorithm to handle **large datasets** and efficiently process them in a timely manner.

— As the volume of data continues to grow exponentially, the need for scalable algorithms has become increasingly important in the field of machine learning.

SCALABLE ALGORITHMS: OPTIONS

There are two options for scalable algorithms

— Use the available hardware as efficient as possible (use GPUs, more efficient parallelization).

We will talk about distributed computations.

— Select other algorithms (maybe more difficult to implement or less accurate) that have better asymptotic with respect to the size of the dataset and dimensionality of the problem.

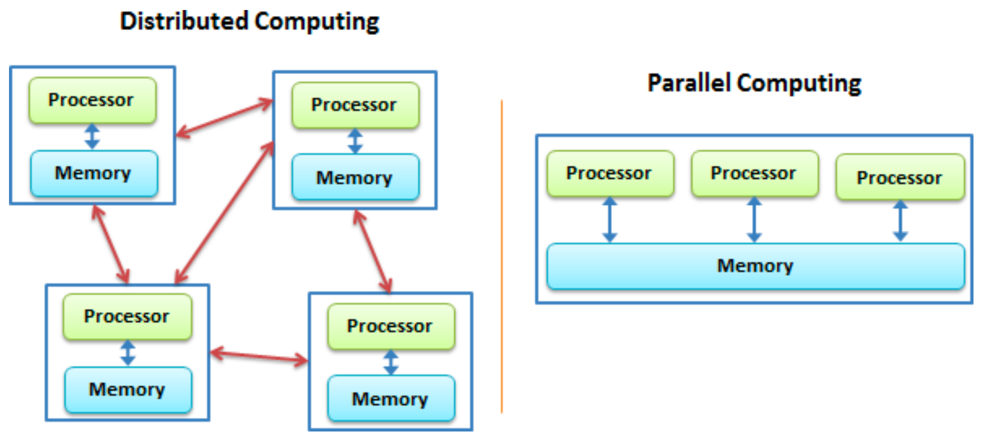
We will talk about randomized PCA and approximate nearest neighbors for clustering.

KEY CHARACTERISTICS (1)

There are several recipes that help to make machine learning algorithms scalable:

— **Parallel Processing**: Scalable algorithms utilize parallel processing techniques to **distribute the workload** among multiple processors or computing nodes. This allows for faster and more efficient processing of large datasets, as the data can be divided and processed simultaneously.

— **Distributed Computing**: In addition to parallel processing, scalable algorithms also use distributed computing techniques **to distribute the data across multiple machines or clusters**. This allows for efficient use of resources and reduces the processing time required for large datasets.



KEY CHARACTERISTICS (2)

Incremental Learning: Scalable algorithms use incremental learning techniques, where the model is continuously updated as new data is fed into the system. This allows for real-time learning and adaptation to changing data patterns, without having to retrain the entire model from scratch.

Sampling Techniques: Another important aspect of scalable algorithms is the use of sampling techniques, where a subset of the data is used for training instead of the entire dataset. This helps in reducing the computational time and resources required for training, while still maintaining a good level of accuracy.

Dimensionality Reduction: High-dimensional datasets can pose a challenge for traditional machine learning algorithms.

Scalable algorithms employ dimensionality reduction techniques to reduce the number of features in the dataset, making it more manageable and easier to process.

LARGE-SCALE COMPUTATIONS: NEED TO USE GPUS

For machine learning, most of the **computations utilize graphical cards**, GPU (Graphical processing units)

They are much more efficient than central processing units (CPUs).

Even old GPUs in Google Colab are much faster than CPUs!

PEAK PERFORMANCE

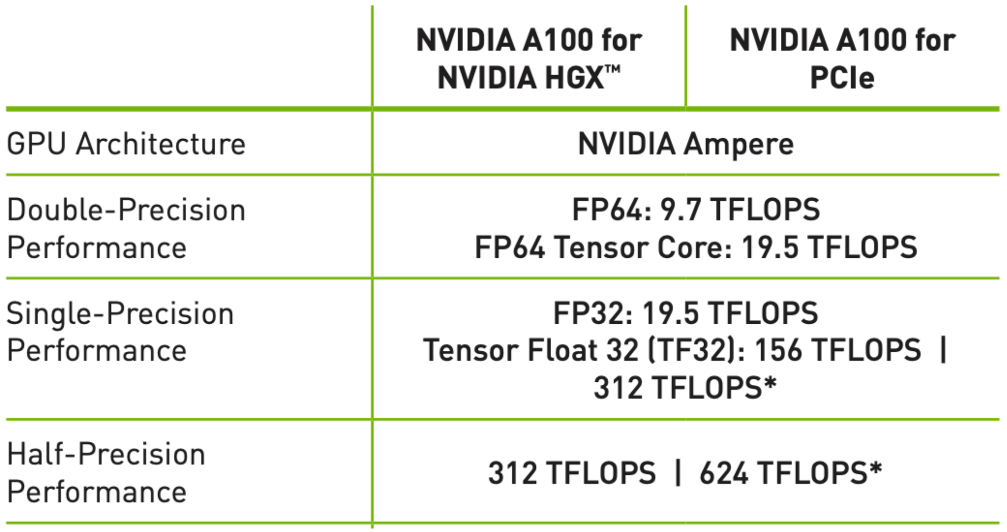
To measure the computational power of the device, we use Flops (floating point operations per second):

Flops is defined as the maximal possible number of operations the device can do per second.

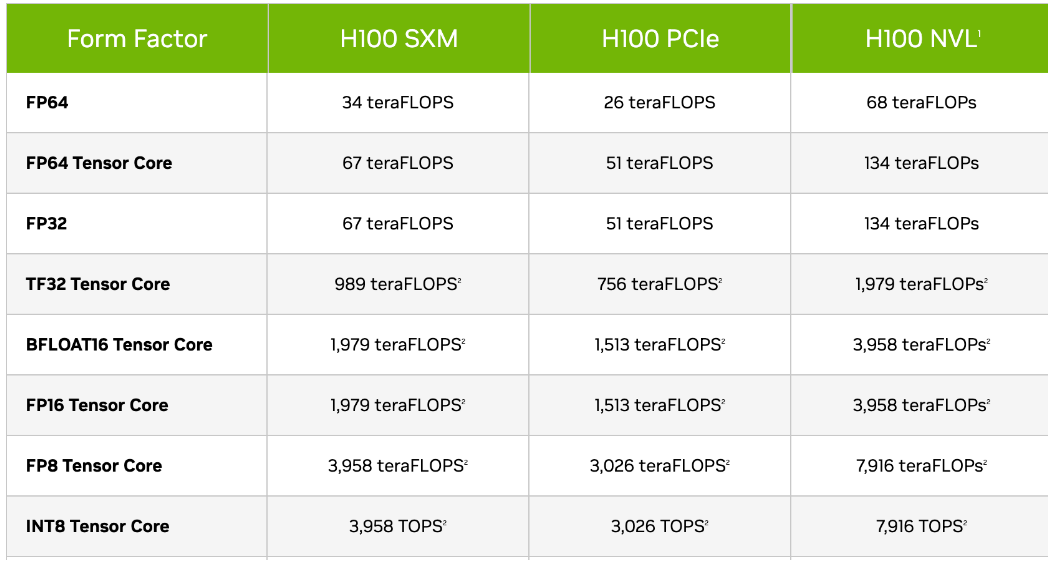
CPU has peak performance of order hundreds of Gigaflops, whereas GPU has peak performance or order tens and hundreds of Teraflops.

Giga = 10^9, Tera = 10^{12}

PEAK PERFORMANCE OF MAIN GPUS



	NVIDIA A100 for NVIDIA HGX™	NVIDIA A100 for PCIe
GPU Architecture	NVIDIA Ampere	
Double-Precision Performance	FP64: 9.7 TFLOPS FP64 Tensor Core: 19.5 TFLOPS	
Single-Precision Performance	FP32: 19.5 TFLOPS Tensor Float 32 (TF32): 156 TFLOPS | 312 TFLOPS*	
Half-Precision Performance	312 TFLOPS | 624 TFLOPS*	



Form Factor	H100 SXM	H100 PCIe	H100 NVL[1]
FP64	34 teraFLOPS	26 teraFLOPS	68 teraFLOPs
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
FP32	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
TF32 Tensor Core	989 teraFLOPS[2]	756 teraFLOPS[2]	1,979 teraFLOPs[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP8 Tensor Core	3,958 teraFLOPS[2]	3,026 teraFLOPS[2]	7,916 teraFLOPs[2]
INT8 Tensor Core	3,958 TOPS[2]	3,026 TOPS[2]	7,916 TOPS[2]

Different floating point formats give different accuracy of the computations;

FP32 — single precision, 32 bit per floating point number

FP64 — double precision, 64 bit per floating point number, rarely used in ML

DISTRIBUTED TRAINING

For larger datasets a single GPU is not enough thus we need to use several GPUs.

To train a sufficiently large model at least 2-4 GPUs are needed.

Thus, we have to effectively distribute computations between different computational nodes.

This is called **distributed training.**

PRACTICAL PERFORMANCE

In practice we get the performance which is much smaller than the peak performance.

Even for very optimized code, used for training large deep neural network models, such as transformer-based model, we have the performance that is about 30-40% of the peak performance.

Now let’s discuss distributed training.

DISTRIBUTED TRAINING

Let’s illustrate the process of distributed training on the example of **stochastic gradient descent** for deep neural network training

The idea is to

a) Keep the parameters of the model on the Parameter Server

b) Use different computational nodes to compute the gradients over different parts of the data

The training can be **synchronous** or **asynchronous**

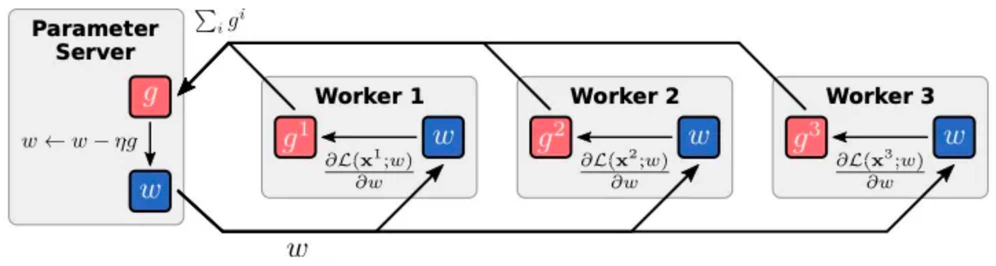
DISTRIBUTED TRAINING: SYNCRONOUS

— **In synchronous training**, the parameter server waits for all gradients from all workers before updating the model parameters.

— This leads to a higher quality parameter update due to less noisy average gradients.

— However if some workers take longer to compute their gradients, other workers are idle and resources are wasted.

— Ideally, every machine should have a task at all times to maximize efficiency.



Parameter Server
$\sum_i g^i$
g
$w \leftarrow w - \eta g$
w
Worker 1
g^1
w
$\frac{\partial \mathcal{L}(\mathbf{x}^1;w)}{\partial w}$
Worker 2
g^2
w
$\frac{\partial \mathcal{L}(\mathbf{x}^2;w)}{\partial w}$
Worker 3
g^3
w
$\frac{\partial \mathcal{L}(\mathbf{x}^3;w)}{\partial w}$
w

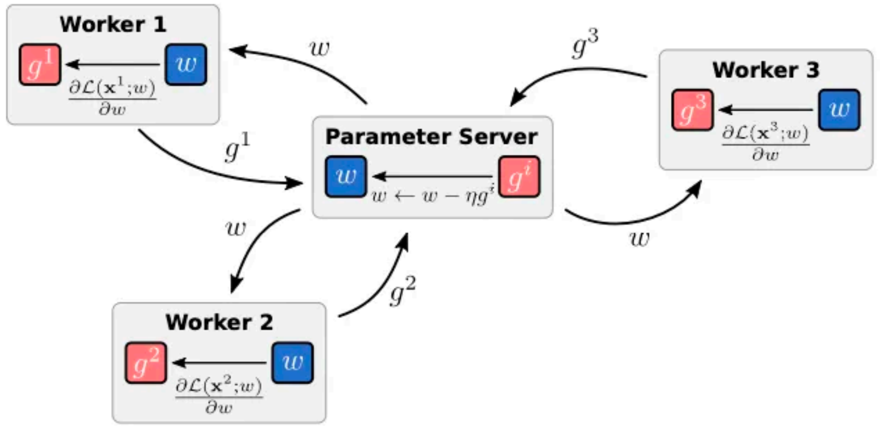
Synchronous data-parallelism. Before sending the updated model parameters, the parameter server aggregates the gradients from all workers. (Image source: Langer et al 2020, link)

DISTRIBUTED TRAINING: ASYNCRONOUS

— We can employ **asynchronous training**, where the parameter server updates the model parameters as soon as it receives a single gradient from a single worker, and sends the updated parameters immediately back to that worker.

— This approach maximizes efficiency by keeping all workers busy at all times, but it can result in lower quality parameter updates due to the noisy average gradients.

— Overall, asynchronous training is a trade-off between efficiency and accuracy, and the optimal choice depends on the specific needs and resources of the training task.



Worker 1
g^1
w
$\frac{\partial \mathcal{L}(\mathbf{x}^1; w)}{\partial w}$
Worker 2
g^2
w
$\frac{\partial \mathcal{L}(\mathbf{x}^2; w)}{\partial w}$
Worker 3
g^3
w
$\frac{\partial \mathcal{L}(\mathbf{x}^3; w)}{\partial w}$
Parameter Server
w
g^i
$w \leftarrow w - \eta g^i$
w
g^1
w
g^2
g^3
w

Asynchronous data-parallelism. Each worker sends their local gradients to the parameter server, and receives the model parameters. (Image source: Langer et al 2020, link)

SUMMARY OF DISTRIBUTED TRAINING

Distributed training is based on the idea of splitting of computation of the sum

$f(\theta) = \sum_{i=1}^{N} l(y_i, \hat{y}_i)$ over different workers and updating the parameters

Different machines works with different parts of the data — **data parallelism**

Modern deep learning models take Gigabytes of memory — may not fit into a single GPU

One can distribute model parameters between different computational nodes — much more complicated.

USING FASTER ALGORITHMS

When we have large and/or high-dimensional datasets, we may use algorithms that have better asymptotic with respect to number of samples and the dimensionality.

Let’s illustrate it on the case of Principle Component Analysis.

PCA: REMINDER

In PCA, we have the data matrix X of size $d \times N$,
where d is the number of features,
and N is the number of data points.

A standard PCA will require forming an $d \times d$ covariance matrix
and computing its *k* leading eigenvectors, or, equivalently,
computing first *k* singular vectors of matrix X.

PCA: COMPLEXITY

Complexity of the SVD-based computation of the PCA will be $\mathcal{O}(nd^2)$,

where Big-O notation means that the total number of arithmetic operations

is bounded by a constant times nd^2

Can we do faster?

There are several fast approximate SVD algorithms,

with the **randomized SVD** being very popular.

RANDOMIZED PCA

For randomized PCA, we do the following **iterative process**

1. We initialize a random matrix Q of size $n \times p$ where p is slightly larger than k
2. We project the data randomly into a p-dimensional subspace as
   $Y = XQ$, matrix Y has size $d \times p$
3. We compute left singular vectors of Y, giving the k principal components.
4. There is a theoretical justification for this algorithm
5. The complexity is now $\mathcal{O}(dnp)$ and if $p \ll d$ this is a significant reduction

SCALABLE CLUSTERING

K-Means is frequently used for exploring large datasets,

since it scales linearly with the number of points.

However, it has limitations:

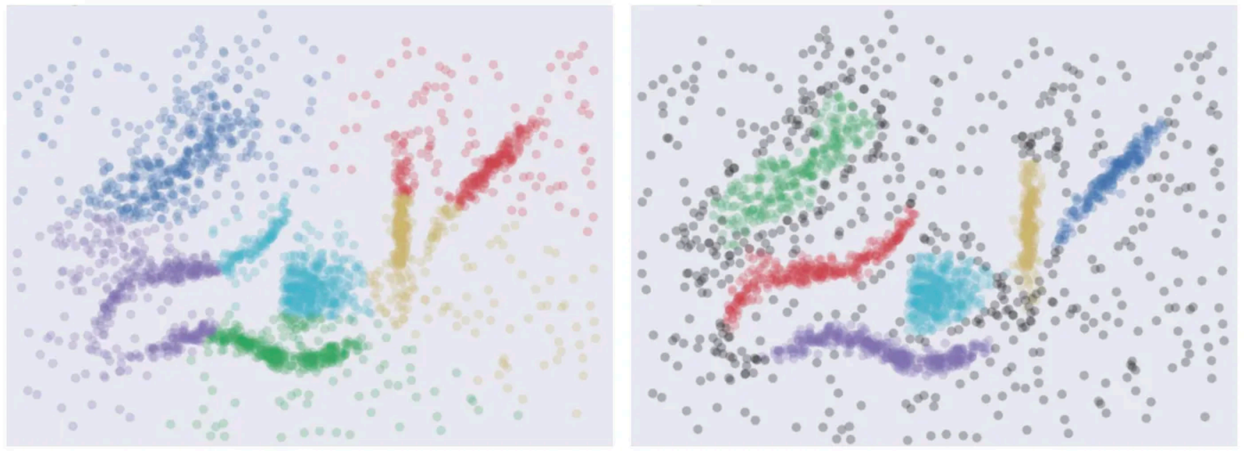
1. The dataset has exactly k clusters
2. Every data point must be in a cluster, real-world data may contain noise.
3. K-means assumes the data have similar size and number of points.

SCALABLE CLUSTERING: HDBSCAN

HDBSCAN stands for
Hierarchical **D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise

It's a clustering algorithm that overcomes many of the limitations of *k*-means. For example, it does not require a difficult-to-determine parameter to be set before it can be used.

- **Hierarchical**: arranges points into hierarchies of clusters within clusters, allowing clusters of different sizes to be identified.
- **Density-Based**: uses density of neighboring points to construct clusters, allowing clusters of any shape to be identified.
- **Applications with Noise**: expects that real-world data is filled with noise, allowing noise points to be identified and excluded from clusters.

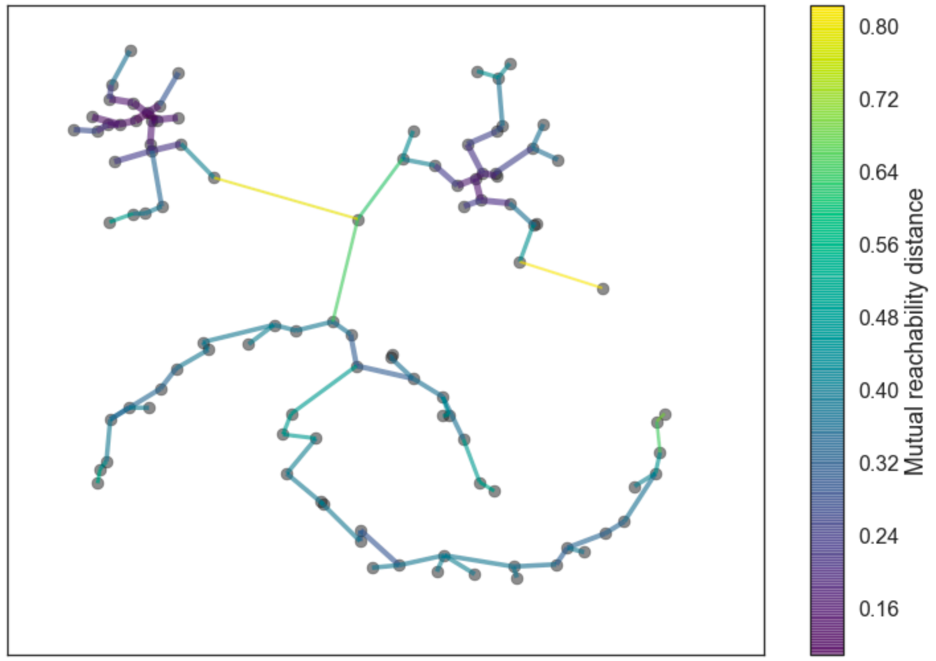


TRADITIONAL IMPLEMENTATION OF HDBSCAN

The traditional implementation of HDBSCAN consists of two main phases
— Construct the *k*-nearest-neighbors (*k*-NN) graph, with an undirected edge connecting each point *p* to the *k* most similar points to *p*. Use the *k*-NN information to define a new metric called the *mutual reachability distance* between all pairs of points in the data

— Find the minimum spanning tree (MST) connecting all points in the data according to the mutual reachability distance. This tree represents a hierarchy of clusters, and the individual clusters can be extracted using a simple heuristic.

Detailed description: https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html



Example of the minimal spanning tree, according to a special metric between points

SCALING

— Construction of k-NN graph scales quadratically with the size of the dataset
— Constructing the minimal spanning tree also scales quadratically

Not applicable to large datasets.

What can we do?

APPROXIMATING THE K-NN GRAPH

To approximate step 1, we can use NN-descent algorithm

It is based on the principle that ***the neighbor of a neighbor is likely to be a neighbor***.

— Each point *p* keeps a list of *k* other points, which are the *k* closest points to *p* that have been found so far.

— In each update step, two randomly chosen *a* and *b* in the neighbor list of a point are compared. If *b* is closer to *a* than the farthest point in *a*'s neighbor list, then the farthest point in *a*'s neighbor list is replaced by *b*.

— Repeating this update improves the accuracy of the *k*-NN graph approximation with each iteration.

APPROXIMATING MINIMAL SPANNING TREE

— To approximate step 2, we find the minimum spanning tree of the graph produced by NN-Descent rather than the complete graph connecting all data points.

— The only important edges of the MST are those which connect points in the same cluster, and those edges should mostly be present in the *k*-NN graph.

— Doing this drastically reduces the size of the graph input to the MST algorithm, and allows the computation to scale to larger datasets.

SUMMARY ON FASTER CLUSTERING

Such kind of approximate nearest neighbors + efficient graph operations

give 5x times improvement already on not very large datasets, with similar quality!

INCREMENTAL LEARNING

The concept of incremental learning:

— The model is continuously updated as new data is fed into the system.

— This allows for real-time learning and adaptation to changing data patterns, without having to retrain the entire model from scratch.

Example: fine-tuning a large model on a small dataset.

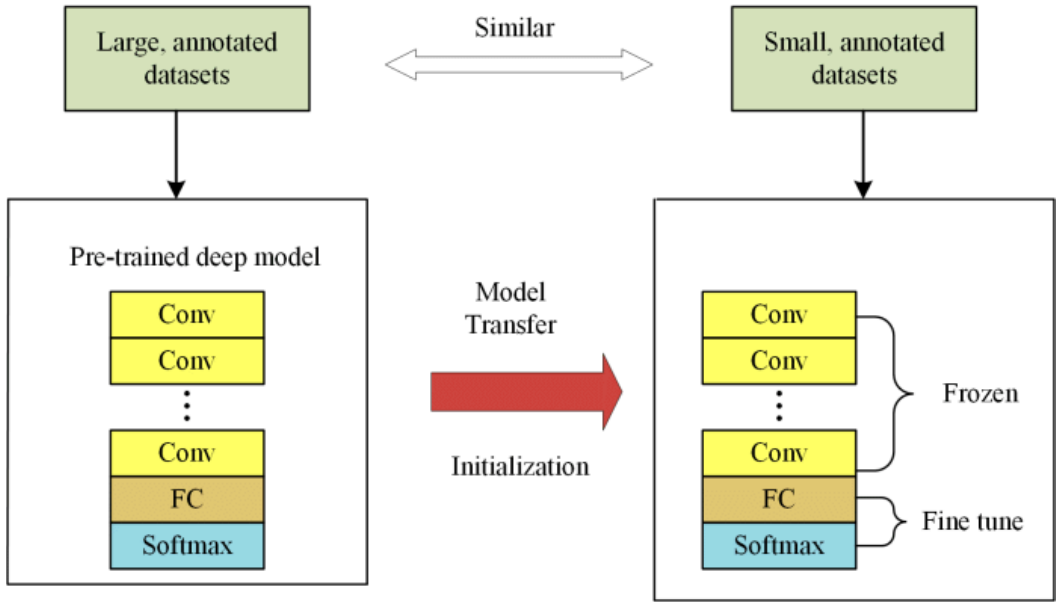
INCREMENTAL LEARNING (2)

A standard pipeline is that you take a deep neural networks consisting of k layers.

The large model is retrained on large, annotated dataset (such as ImageNet)

The new model is trained on a small dataset, but the first layers **are frozen.**

This concept of **pretraining** is very powerful: one can use such pretrained models for many downstream task, and the cost of fine-tuning is negligible.



SCALING OF DIFFERENT ALGORITHMS FOR LARGE DATA

— **k-means clustering**: scales well

— **Random forest**: scales well, since trained on a subset of data

— **Gradient boosting**: similar, scales well

— **Deep learning algorithms**: scales well

— **Kernel methods**: does not scale well due to quadratic memory and cubic complexity with respect to the number of samples, can be solved with selecting smaller sample.

RECAP OF LECTURE 8

— Scalable algorithms: distributed computing

— Randomized PCA

— Approximate nearest neighbors.

NEXT LECTURE

WE ARE DONE WITH LECTURES!