Skalierbare Methoden der Künstlichen Intelligenz

0 Einführung

- Warum skalierbare KI?
  - Rechenzeit (sequentielle Berechnung dauert zu lange) → Beschleunigung
  - Speicherbedarf (Daten zu groß für Speicher eines Knotens) → Befähigung

1 Grundbegriffe

- Deep Learning (NNs mit mehr als einer versteckten Schicht) ⊆ ML (Programmierung aus Daten) ⊆ KI (Imitation intelligenten Verhaltens)
- Self-supervised: Lernen aus nicht annotierten Daten durch Lösen selbstgestellter Probleme (bspw. Daten rekonstruieren): Dimensionsreduktion, Autoencoder
- Strategien (Skalierung): Härter arbeiten (Taktfrequenz erhöhen → physikalisch limitiert), Schlauer arbeiten (bessere Algorithmen → nicht immer möglich), Hilfe holen (Skalierung der Berechnung)
- Vertikale Skalierung (scale up): Einzelne Knoten vergrößern (mehr CPUs, etc.)
- Horizontale Skalierung (scale out): Mehr Knoten verwenden (mehr Verbindungen)
- Nebenläufigkeit (concurrency): Unabhängig und in beliebiger Reihenfolge
- Parallelisierung (parallelization): Gleichzeitig und unabhängig
- High-Throughput Computing (HTC, Maximierung des Durchsatzes): Effiziente nebenläufige Ausführung unabhängiger Aufgaben (hauptsächlich Inferenz)
- High-Performance Computing (HPC): Parallelisierung stark abhängiger Vorgänge (hauptsächlich Training)
- Implizite Verktorisierung (bspw. durch Compiler, zwingend nebenläufig), explizite Vektorisierung (zwingend parallel)
- (Symmetric) Multiprocessing (SMP): Knoten hat mehrere (gleiche) CPUs

- Chip Multiprocessing (CMP): CPU besteht aus mehreren Kernen
- Simultaneous Multithreading (SMT): Mehrere Threads pro Kern
- Prozess: Getrennter Adressraum, Message Passing
- Thread: Gemeinsamer Adressraum, explizite Synchronisation (Mutex)
- Beschleuniger: spezialisierte Hardware (GPU), meist Vektorrechner, programmiert über eigene Sprache (VHDL) oder Dialekt (CUDA), Datentransfer notwendig
- Hierarchisches Softwaredesign (bspw. NumPy + Python): Kern in hardwarenahen Sprachen (C++), High-level Script API (Python)
- Multiple Programs Multiple Data (MPMD): unterschiedliche Programme (Server-Client, Dask, Map-Reduce/Spark), häufig zur Fehlertoleranz
- Single Program Multiple Data (SPMD): gleiches Programm mit Fallunterscheidungen (MPI, Facebook Gloo, NVidia NCCL), häufig zur maximalen Performanz
- Batchprocessing: Jobs auf Clustersystemen mit Batch Scheduler (stellt Ressourcen bereit, priorisiert, bspw. SLRUM, LSF, PBS)
- Starke Skalierung: Problem konstanter Größe durch mehr Ressourcen schneller lösen. Zielmetrik: Speedup (Problem: $1/T_{\text{seriell}}$ ist Asymptote für Speed-up)
- Schwache Skalierung: Proportional größer werdendes Problem durch mehr Ressourcen in gleicher Zeit lösen. Zielmetrik A: skalierter Speedup: $S_{\text{skaliert}} = \frac{T_{\text{skaliert}} + P \cdot T_{\text{parallel}}}{T_P}$; Zielmetrik B: Effizienz $E = \frac{S_{\text{skaliert}}}{P} = \frac{T_{\text{seriell}} + P \cdot T_{\text{parallel}}}{P \cdot T_P}$

2 K-means Clustering

- Assozation: Regeln finden, die Korrelationen zwischen Datenpunkten beschreiben (Dendrogram?)
- Dimensionsreduktion: Reduktion der Daten auf wesentliche Variablen
- Cluster-Analyse: Auffinden von Ähnlichkeitsstrukturen
- K-Means: Minimierung der Cluster-Varianz $J = \sum_{i=1}^{k} \sum_{\vec{x}_j \in S_i} ||\vec{x}_j - \vec{\mu}_i||^2$
- Lloyd Algorithmus: Zufällige Zentroiden aus Daten wählen, Datenpunkte nächstem Zentroiden zuordnen, Zentroiden neu berechnen, wiederholen bis Konvergenz
- K-Means++: initiale Zentroiden sind möglichst weit voneinander entfernt
- Motivation für Parallelisierung: Mehr Daten (bspw. Cluster von Sternen), Curse of Dimensionality: Komplexität wächst propertional zu Anzahl Cluster/Datenpunkte

- **Parallelisierung an Sample-Achse**: Jeder Prozessor berechnet Distanzen und nächsten Zentroiden für seine Datenpunkte: $\min_k \frac{1}{f} \sum_{i=1}^{f} (x_j - \mu_k)^2$. Neue Zentroiden als Mittelwert über alle Prozessoren hinweg.
  - Kommunikation: Reduktion für Summe (Kommunikation $\mathcal{O}(k)$)
  - Synchronisation: Nach jeder Iteration um Zentroiden zu aktualisieren
- **Parallelisierung an Feature-Achse**: Jeder Prozessor p hält Datenmatrix $[n \times f/p]$ und berechnet Abstandsquadrate seiner Komponenten: $d_p(\vec{x}_j, k) = \sum_{i=p \cdot f/P}^{(p+1) \cdot f/P} (x_j^{(i)} - \mu_k^{(i)})^2$. Distanzmatrix über alle Prozessoren hinweg. Jeder bestimmt nächsten Zentroiden (Redundanz). Jeder berechnet neue Zentroiden für seine Komponenten.
  - Kommunikation: Reduktion für Summe $D_{j,k} = \frac{1}{P} \sum_p d_p(\vec{x}_j, k)$ (hoher Kommunikationsaufwand $\mathcal{O}(k \cdot n)$)
  - Synchronisation: Zusammenführen der Zentroiden-Komponenten
- Dynamic Task Scheduling: Beispiel Dask (Austausch von Python Objekten)
  - Verzögerte Ausführung: Aufgaben sammeln → optimiert verteilen (Task Graph)
  - Task Scheduler: Zentrale Instanz (Verteilen, Zusammenführen)
  - Kommunikation nur zwischen Task Scheduler und Worker
- MapReduce: Nebenläufige Berechnung: Map (Prozess-lokale Operation auf einzelnen Elementen), Shuffle, Reduce (Zusammenfassen der Map über alle Prozesse)
- Problem: Input/Output wird von Festplatte gelesen bzw. geschrieben (zeitintensiv) → moderne MapReduce Software umgeht das
- K-means MapReduce: Map: bestimmte nächsten Zentroiden für jedes $\vec{x}$ (Distanzen + Minimum) → Key-Value Paar: Cluster Index i als Schlüssel, $(x, 1)$ als Wert; Reduce: Summe über Werte des gleichen Schlüssels
- K-means MapReduce für Feature-Aufteilung: MapReduce zwei Mal ausführen
- K-medians: Median der Punkte ⇔ Minimierung bzgl. L_1-Norm: $||\vec{x}||_1 = \sum_{i=1}^{f} |x^{(f)}|$
- Median benötigt Sortieren: Parallelisierung an Sample-Achse → verteiltes Sortieren
- PSRS: Parallel Sorting by Regular Sampling
  1. Sortiere lokale n/p Daten, sample Datenpunkte mit Index $1, \ldots, (p-1)w+1$ mit $w = n/p^2$ → Gather-Operation für p^2 Samples (jeweils p)
  2. Sequentiell: Sortiere Samples und wähle $(p-1)$ Pivots an Indizes $p+\rho, \ldots, (p-1)p+\rho$ mit $\rho = p/2$, Brodcast für Pivots

     Parallel: Bilde p Partitionen anhand der Pivots (Split-Stellen)
  3. Austausch: Jeder Prozessor i behält Partition i und verteilt $p-1$ Partitionen

4. Merge: Jeder Prozessor fügt p erhaltenen Partitionen in eine Liste zusammen

   Konkatenation der einzelnen Listen ist die final sortierte Liste

- PSRS: Reguläres Sampling für Load Balancing: reguläre Samples approximierten Werteverteilung der Keys; moderater Kommunikationsaufwand; kann mit verschiedenen Verteilungen umgehen; ohne Duplikate bearbeitet jeder Prozessor maximal $2n/p$ Elemente (Schranke wächst linear mit der Anzahl Duplikate)

3 DBSCAN

- Density-Based Spatial Clustering for Applications with Noise
- Grundidee: Cluster $\rightarrow$ hohe Dichte; Rauschen $\rightarrow$ geringe Dichte
- Parameter: räumlicher Suchradius ε; Punkteanzahl für hohe Dichte minPoints
- Ein Punkt ist ein Clusterkern, wenn größer gleich minPoints Punkte in seiner Epsilonnachbarschaft liegen.
- Punkt ist Randpunkt, wenn seine Epsilonnachbarschaft kleiner minPoints ist, er aber in der Epsilonnachbarschaft eines Kerns ist (Directly Density Reachable)
- Ein Punkt ist Rauschen, wenn er von keinem Kern erreichbar ist.
- Rechenkomplexität: $\mathcal{O}(n^2)$ klassisch, $\mathcal{O}(n \cdot \log(n))$ mit räumlicher Indexstruktur
- Speicherkomplexität: $\mathcal{O}(n \cdot \text{minPoints})$
- **PDSDBSCAN**: Parallel Disjoint-Set DBSCAN
  1. Daten uniform partitioniert laden
  2. Alle Epsilonnachbarschaften austauschen
  3. Modifiziertes Disjoint-Set DBSCAN auf lokaler Partition ausführen
  4. Disjoint-Sets zusammenführen
- Disjoint-Set Datenstruktur: Mengen haben Repräsentant (bel. Element), *find()*- und *union()*-Operation, Zusammenführung durch Umsetzen von Referenzen
- **HPDBSCAN**: Highly Parallel DBSCAN
  1. Daten uniform partitioniert laden
  2. Parallel lokal mit regelmäßigem Gitter indizieren (Zellen haben ε Ausdehnung in jede Dimension)
  3. Indizierte Daten umverteilen, damit disjunkte Unterräume inkl. Halos (Zellen am Rand eines Unterraums) entstehen; Lastbalancierung durch Heuristik, die Anzahl Berechnungen pro Unterraum misst und dementsprechend aufteilt: $S_{\text{Cell}}(c) = |c| \times |N_\varepsilon(c)|$ und $S_{\text{Total}} = \sum_{c \in \text{Cells}} S_{\text{Cells}}(c)$

4. Lokales iteratives DBSCAN ausführen. Nutze atomicMin um Clusterlabel als Minimum von allen Labels im Cluster setzen (parallel, damit Merge funktioniert)
5. Cluster über Regeln fusionieren (parallel mit atomicMin)
6. *Optional*: initiale Reihenfolge wiederherstellen

4 Spektrales Clustering

- Graphentheoretisch: Knoten = Sample $\vec{x}$, Kanten = Ähnlichkeit $s_{ij} = d(\vec{x}_i, \vec{x}_j)$
- Laplace-Matrix $\mathcal{L} = D - A$ mit Adjazenz-/Ähnlichkeitsmatrix A und Gradmatrix $D = [\deg(\vec{v}_i)] = [\sum_j d(\vec{x}_i, \vec{x}_j)]$
- Multiplizität Eigenwert 0 von $\mathcal{L}$ entspricht Anzahl unabhängiger Komponenten
- Spektraler Abstand (Spectral Gap): Finde k kleinste Eigenwerte; zugehörige Eigenvektoren sind spektrale Einbettung (Spectral Embedding); Clustern in niederdimensionaler Darstellung (bspw. k-Means)
- Spektrales Clustering
  1. Berechne Adjazenzmatrix $A \in \mathbb{R}^{n \times n}$ mittels $d(\vec{x}_i, \vec{x}_j)$
  2. Erstelle die Laplace-Matrix $\mathcal{L}$ mithilfe von A
  3. Berechne Eigenwertzerlegung von $\mathcal{L}$
  4. Stelle Matrix $U \in \mathbb{R}^{n \times k}$ auf, mit Eigenvektoren $\vec{u}_1, \ldots, \vec{u}_k$ der k kleinsten Eigenwerte als Spalten
  5. k-Means Clustering von reduzierter Datendarstellung U
- Distanzmetriken: Manhatten ($\sum_f |x_i^{(f)} - x_j^{(f)}|$), euklidisch, radiale Basisfunktion (RBF-Kernel, $\exp(-\frac{1}{2\sigma} \sum_f (x_i^{(f)} - x_j^{(f)})^2)$)
- Graph-Darstellungen: ε-Nachbarschaft (ungewichtet: $A_{ij} = 1\{s_{ij} \geq \varepsilon\}$, gewichtet: $A_{ij} = s_{ij}\{s_{ij} \geq \varepsilon\}$); k-nächste Nachbarn (ungewichtet: $A_{ij} = 1\{v_j \in \text{kNN}(v_i)\}$, gewichtet: $A_{ij} = s_{ij}\{v_j \in \text{kNN}(v_i)\}$); voll vernetzt (gewichtet: $A_{ij} = s_{ij}$)
- Laplace-Matrix: nicht-normalisiert ($\mathcal{L} = D - A$), normalisiert ($\mathcal{L}_{\text{rw}} = D^{-1}\mathcal{L}$), normalisiert symmetrisch ($\mathcal{L}_{\text{rw}} = D^{-1/2}\mathcal{L}D^{-1/2}$)
- Parallelisierung der Berechnung von A: sequentiell $\mathcal{O}(n^2 \cdot f)$ Rechenzeit und $\mathcal{O}(n^2)$ Speicher $\rightarrow$ verteilen an Sample-Achse
- **Parallel spectral clustering in distributed systems**: Jeder Prozessor berechnet s_{ij} zwischen lokalen N/P Samples $\vec{x}_i$ und gesamten Datensatz $\vec{x}_j$: $d(\vec{x}_i, \vec{x}_j) = ||\vec{x}_i - \vec{x}_j||^2 = ||\vec{x}_i||^2 + ||\vec{x}_j||^2 - 2\vec{x}_i^\top \vec{x}_j$ (Vorberechnung von $||\vec{x}_i||^2$ lokal und lokale Matrix-Multiplikation für $||\vec{x}_j||^2$ und $2\vec{x}_i^\top \vec{x}_j$)

- Falls X nicht in RAM passt: Teile X in Blöcke auf und berechne Distanzen einzeln (wiederholte Festplatten-I/O) $\rightarrow$ deswegen verteilte paarweise Distanzen
- **Generalisierte verteilte paarweise Distanzen** (für nicht-Minkowski Distanzen)
  1. Iteration 0: Jeder berechnet paarweise Distanzen lokal vorliegender Samples c_p
  2. Für i in $1, \ldots, (P+1)/2$ Iterationen:
     a) p schickt c_p and $(p+i)$ mod P
     b) Berechne Block $d(c_p, c_{p+i})$ (Nebendiagonale)
     c) Wegen Symmetrie schickt p sein Ergebnis an $(p-i)$ mod P zurück
- Parallelisierung der Berechnung der Eigenwerte: Numerische Verfahren rechenintensiv für dichtbesetzte Matrizen $\rightarrow$ effizientere Methoden für sparse Matrizen
- Transformiere in sparse Matrix gleicher Größe (t-nächste Nachbarn, ε-Nachbarschaft, Random Sampling) $\rightarrow$ paralleler Eigenwert-Löser (Arnoldi-/Lanzcos-Verfahren)
- **Lanczos Algorithmus**: Iteratives Verfahren zur Bestimmung extremer Eigenwerte für symmetrische hermitesche Matrizen (nicht geeignet für vollvernetzte Graphen)
  - In m Iterationen erstelle tridiagonale, symmetrische Matrix $T \in \mathbb{R}^m \times \mathbb{R}^m = V^* A V$ und V mit orthonormalen Spalten ($m \ll n$).
  - $T\vec{x} = \lambda\vec{x} \Rightarrow \vec{y} = V\vec{x}$ ist EV von A.
- Ansatz für vollvernetzt: **Nyström Methode** (Nähern der dichten Ähnlichkeitsmatrix durch Sub-matrix kleinerer Größe $[m \times m]$)
  - Ziehe zufällig l Reihen aus Matrix S_d und berechne Distanzen $A : [l \times l]$
  - $B : [l \times (n-l)]$ Distanzen zwischen l gezogenen und $n-l$ übrigen Datenpunkten
  - Nyström-Methode: Näherung von S_d mittels A und B.
  - $W = \begin{bmatrix} A \\ B^\top \end{bmatrix}$, $S_d \approx W A^{-1} W^\top$
  - Eigenwertzerlegung $A = V_A \Sigma_A V_A^\top \Rightarrow$ Nähere Eigenwerte von S_d an: $\tilde{\Sigma} = \frac{n}{l}\Sigma_A$, $\tilde{V} = \sqrt{\frac{l}{n}} W V_A \tilde{\Sigma}$
  - Berechne die normalisierte Matrix $\tilde{U}_{ij} = \frac{\tilde{V_{ij}}}{\sqrt{\sum_{r=1}^{k} \tilde{V}_{ir}^2}}$
  - Partitioniere die n Reihen von $\tilde{U}$ mittels k-Means in k Cluster.

*	Erklärung	Komplexität
	Clustering	
k-Means (Sample-Achse)	Teile Samples uniform auf, berechne näheste Zentroiden lokal, Reduktion für neue Zentroiden.	Kommunikation $\mathcal{O}(k)$
k-Means (Feature-Achse)	Teile Features uniform auf, berechne Teildistanz, Reduktion für Gesamtdistanz, lokal Teilzentroid berechnen.	Kommunikation $\mathcal{O}(k \cdot n)$
k-Means (MapReduce)	Map: Cluster mit kleinstem Abstand (Key), Shuffle nach Key, Reduktion für Zentroid pro Cluster.	*
k-Medians: PSRS	Daten uniform aufteilen, lokal sortieren, $p \cdot p$ Samples an root, Samples auf root sortieren $p-1$ Split-Points verteilen pro Split an Prozessor schicken, Einzellisten auf Prozessoren mergen, dann alle Teillisten konkatenieren	Ohne Duplikate bearbeitet jeder Prozessor maximal $2n/p$ Elemente (wächst linear mit Duplikaten)
PDSDBSCAN	Daten uniform laden, Epsilonnachbarschaften austauschen, Disjoint-Set DBSCAN lokal ausführen, Disjoint-Sets mergen.	Parallelisierungsanteil 97%
HPDBSCAN	Daten uniform laden, in Grid einteilen, nach Heuristik umsortieren, DBSCAN mit atomicMin lokal ausführen, Cluster (parallel) über Regeln fusionieren.	Parallelisierungsanteil 99%
Parallel spectral clustering	Jeder Prozessor berechnet für N/P Samples die Abstände zu allen Samples	Falls X nicht in RAM passt $\rightarrow$ wiederholte Festplatten-IO
Generalisierte verteilte paarweise Distanzen	Daten uniform laden, lokale Distanzen berechnen, pro Iteration Daten an anderen Prozessor senden, Nebendiagonale berechnen.	*
Lanczos Algorithmus	Extreme EWs für dünne Matrizen: Erstelle tridiagonales T mit gleichen EWs wie A.	*
Nyström Methode	EW annäheren für dichte Matrizen: Wähle Submatrix $A : [l \times l]$ und $B : [l \times (n-l)]$ und nähere damit Submatrix $C : [(n-l) \times (n-l)]$ an: $C \approx B^\top A^{-1} B$. Berechne EWZ von $A = V_A \Sigma_A V_A^\top \rightsquigarrow$ EWZ $\tilde{S} = \tilde{V} \tilde{\Sigma} \tilde{V}^\top$	*
	Lineare Regression	

Merry Go Round	Jeder bekommt Teilmenge der Zeilen von A und Spalten von B, berechnet iterativ Block C_{ii}, gibt Spalten von B einen Prozessor weiter, berechnet nächsten Block bis alle Spalten von B verarbeitet wurden	Kommunikation $p \cdot n^2$
SUMMA	Jeder bekommt Block aus A und B und kommuniziert Teilspalten aus A mit seiner Zeile in A und Teilzeilen aus B mit seiner Spalte in B. Berechnet für jeder Teilzeile aus A und Teilspalte aus B das Outer Product und summiert sie.	Kommunikation $\sqrt{p} \cdot n^2$
	Support Vector Machines	
SVM Optimierung	$\min_{\vec{w},b}\{\frac{1}{2}\|\vec{w}\|^2\}$ N.B. $y_n(\langle \vec{w}, \vec{x}_n \rangle + b) \geq 1$, $n = 1, \ldots, N$	
SVM mit Slack	$\mathcal{L}(\vec{w}, b, \vec{\alpha}, \vec{\mu}) = \frac{1}{2}\|\vec{w}\|^2 + C\sum_{n=1}^{N} \xi_n - \sum_{n=1}^{N} \alpha_n [y_n(\langle \vec{w}, \vec{x}_n \rangle + b) - 1 + \xi_n] - \sum_{n=1}^{N} \mu_n \xi_n$	
Coordinate Ascend	Optimiere alle α einzeln nacheinander iterativ, bis konvergiert	
Sequential Minimal Optimization CA	Optimiere alle α (je zwei) nacheinander iterativ. $\sum_{n=1}^{N} \alpha_n y_n = 0$ kann nicht eingehalten werden, wenn nur ein α geändert wird.	Rechenaufwand $\mathcal{O}(N^2 \cdot T)$ T: Anzahl Iterationen
Inkrementelle SVMs	Daten partioniert laden, jeweils SVM trainieren und nur Stützvektoren an nächsten Prozessor weitergeben.	Performant nur für IID. $\mathcal{O}(T \cdot \|SV\|)$ für Kommunikation
Cascade SVMs	Daten partioniert laden, jeweils SVM trainieren und nur Stützvektoren an Elternknoten weitergeben. Root gibt finale Stützvektoren zurück an Blätter und Blatt entscheidet, ob das ein alternativer Stützvektor ist (bis Konvergenz)	Rechenzeit/ Kommunikation stark abhängig von Verhältnis Stüztvektoren zu Trainingssample. Verbessert durch randomisierte Vernetzung. $\mathcal{O}(T \cdot \|SV\|)$ für Kommunikation
Alternating Direction Method of Multipliers (ADMM)	Daten partioniert laden. Dezentrale Optimierungsprobleme sind durch Konsens-Bedingung für $\vec{v}$ gekoppelt. Optimiere nach lokalem $\vec{v}$ und versuche es nah an globalem $\vec{\omega}$ zu halten durch Austausch mit Nachbarn.	$\mathcal{O}(T \cdot P)$, vertikal Kommunikation in $\mathcal{O}(T \cdot n)$

Incremental Least Squares SVM	Fehler-Minimierung von ξ statt $\geq$ in der Nebenbedingung $\rightsquigarrow$ LGS statt quadratisches Optimierungsproblem. Jeder Prozessor berechnet Teilergebnis, Koordinator summiert Teilergebnisse auf und löst globales LGS	$\mathcal{O}(P^2)$
Vertically Distributed Core Vector Machines	Features aufteilen. Jeweils Kugel um Daten einer Klasse herum bilden (immer mit weitesten entfernten Datenpunkt. Hier wird kommuniziert). Iteration beenden, wenn Kugelgröße annähernd konstant bleibt.	Kommunikation in $\mathcal{O}(T^2)$
Privacy-Preserving SVMs	Features aufteilen. Berechne lokale Kernelmatrix, kommuniziere an root und root erstellt gesamte Kernelmatrix (Addition/-Komponentenweises Produkt je nach Kernel). Root löst Optimierungsproblem.	Kommunikation $\mathcal{O}(n^2)$
Seperable SVMs	Löse primäres SVM Problem auf jedem Prozessor mit SGD. Finde Gewichtung für Kombination lokaler Vorhersagen (wie ein Ensemble).	
	Neuronale Netze	
DPNN Allgemein	Daten IID partioniert laden, jeder macht lokal Forward-Backward-Pass. Dazwischen werden Netzwerkgewichte synchronisiert (Kommunikation und Mittelung von Gradienten $\rightarrow$ evtl. anderer Gradient wegen ungleichmäßiger Aufteilung)	Alle Gewichtsupdates müssen synchronisiert werden (Parameter-Server oder Allreduce)
Verteiltes Mischen	Jeder mischt lokalen Chunk, kommuniziert Teile des lokalen Chunks (all-to-all, Ringkommunikation, etc.) und mischt neuen lokalen Chunk erneut.	
Asynchronous Stochastic Gradient Descent (ASGD)	Parameter-Server hält Modellgewichte, empfängt Gradienten (asynchron). Worker bekommt zu Beginn des Forward-Passes aktuelle Modellgewichte. Stale Gradients beeinträchtigen Konvergenz.	Überlappung von Kommunikation und Berechnung $\rightarrow$ Beschleunigung.
Tree Allreduce	Stufe 1: Paarweise Reduktion von Elementen. Stufe 2: Paarweise Verteilung des Ergebnis. Reduktor zwingend assoziativ.	Anzahl Nachrichten $\mathcal{O}(2(2p-2))$, Datendurchsatz $\mathcal{O}(np)$, Latenz $\mathcal{O}(2\log p)$

Ring Allreduce	Teilen-Reduktions-Phase (Operation auf Element anwenden und weiterschicken). Nur-Teilen-Phase: Alle erhalten reduziertes Ergebnis.	Anzahl Nachrichten $\mathcal{O}(2(p-1)p)$, $\mathcal{O}(\frac{n}{p}(p-1))$ für Datendurchsatz, $\mathcal{O}(2(p-1))$ für Latenz
Fully Connected Layer Parallel	Forward-Pass: Verteile Matrixmultiplikation. Backpropagation: Allgather für Gradienten, Speicher evtl. transponieren.	
Scalable Massively Parallel Artificial Neural Networks	Neuronen jeder Schicht gleicherverteilt auf Prozessoren (lokal vollvernetzt), Kommunikation nur über Geister-Neuronen am Rand. Alle Prozessoren bekommen gesamten Input.	
Cannon's Matrix Multiplikation	Jeder Prozessor hält je einen Block von A und B, shifte Blöcke in Zeile von A/Spalte von B so, dass jeder Prozesser die Blöcke bekommt, die er für Berechnung seines Blockes in C braucht.	
Verteilte Channel CNNs	Jeder Prozessor hat C/P der Filter. Jeden Filter auf gesamten Output der vorgehenden Schicht anwenden (Allgather für Output. Allreduce, da Ergebnis Summe über alle Channel ist). Backward-Pass: Allgather für Gradienten (Aktivierungen zwischenspeichern oder neu berechnen).	Häufig ineffizient, da viel Kommunikation bei großen Featuremaps.
Verteilte Featuremaps CNNs	Jeder Prozessor hält Teil der Featuremaps (Austausch von Halos). Je Faltung für eigene Featuremap berechnen.	Kommunikation nur mit nächstem Nachbarn (paarweise nichtblockierend $\rightsquigarrow$ Überlappung Kommunikation und Berechnung)
GPipe	Splitte Mini-Batch in Micro-Batches, berechne ersten Micro-Batch auf Prozessor 1 und sende Ergebnis weiter. Fange parallel mit zweitem Micro-Batch an (usw.). Bubble entsteht durch synchrone Gewichtsupdates. Aktivierungen werden im Backward-Pass neuberechnet.	Optimiert für Speichereffizienz

PipeDream	Inter- und Intra-Batch Parallelismus durch asynchrone Gewichtsupdates. Nach Warmup-Phase gibt es theoretisch keine Pipeline-Bubbles mehr. Jedoch wieder Stale Gradients.	Weniger Kommunikation (nur an Grenzen Aktivierungen versenden, nur Nachbarn kommunizieren)
	Hyperparameteroptimierung	
Rastersuche	Kartesisches Raster als Suchraum. Jeden Kandidaten ausprobieren (trivial parallelisierbar)	Fluch der Dimensionalität (steigt exponentiell in Anzahl Features)
Zufallssuche	Lösung zufällig aus Suchraum sampeln. Robuster gegen unkorrelierte Features. Trivial parallelisierbar.	Fluch der Dimensionalität. Kein Checkpoint notwendig.
Sequential Model-based Optimization (SBMO)	Zielfunktion durch Surrogat modellieren, von Surrogat nächsten Kandidaten vorschlagen (Selektions-/Akquisitionsfunktion), Evaluation von Kandidaten, Surrogate updaten, iterativ wiederholen.	
Parallele Bayessche Optimierung	Quasi SBMO mit Gauß-Prozess als Surrogat mit Kernelfunktion als Schätzung der Kovarianz. Cholesky-Zerlegung $\Sigma = BB^\top \rightsquigarrow \begin{pmatrix} M \\ R \end{pmatrix} \sim m + B\mathcal{N}(\vec{0}, I)$. Akquisitionsfunktion: Probability of Improvement/Upper Confidence Bound. Parallelisierung: Alle werten unabhängig Samples aus und aktualisieren den selben Gaussian Process.	
Kreuzvalidierung	Bessere Loss-Schätzung: Trainiere jeweils auf $k-1$ Folds und verwende den letzten als Testdaten. Wiederhole, bis jeder Fold einmal Test war. Zufällig oder stratifiziert (Verteilung in Folds nahezu identisch). Trivial parallelisierbar.	
	Biologisch inspirierte Algorithmen	
Partikelschwarmoptimierung	Partikel mit Trägheit, kognitiver und sozialer Komponente mit zufälliger Gewichtung. Gradientenfrei.	Allreduce für globales Optimum.

Genetische Algorithmen	Nachahmung natürlicher Auslese. Individuen mit Genen bilden Population. Selektion (Roulette: proportional zur Fitness, Elitismus: k beste, Turnier: wähle aus je k zufälligen Individuen stärksten), Mutation (Einzelmutation mit P_{mutate}, Schwund, Bitmutation, bedingte Mutation: bspw. Layer) und Kreuzung (k-Kreuzung an jeweils k zufälligen Schnittpunkten, uniforme Kreuzung (jedes Gen 50:50)). Gradientenfrei. Feingranular (einzelne Individuen parallelisieren), grobgranular (Individuen aufteilen), hybrid.	Nachrichtenanzahl durch Inselmigration reduzieren.
	Ensemble-Methoden	
Bootstrap AGGregatING	Verwende Stichproben der Trainingsdaten um (meistens) verschiedene Entscheidungsbäume zu trainieren.	
Sample-parallele Random Forests	Wie Bagging, aber an einem Split stehen nur eine zufällige Untermenge aller Features zur Verfügung $\rightsquigarrow$ mehr Diversität. Inferenz trivial parallelisierbar. Training über verteilte Daten (IID $\rightsquigarrow$ verteiltes Shuffling). Kommunikation für Output.	Rechenkomplexität $\mathcal{O}(kMN \log N)$
Feature-parallele Random Forests	Feature-Subsets auf verschiedenen Prozessoren. DSI Tabelle für Indizerung der Subsets. Taskparallel mit DAG und Task-Scheduler. Dimension Reduction und Weighted Voting für bessere Accuracy. Parallelisierung via MapReduce.	Rechenkomplexität $\mathcal{O}(k(M \cdot N + m \cdot N \cdot \log(N)))$ mit $m < M$
AdaBoost	Trainiere Entscheidungsstümpfe und verändere Gewichte der Trainingsdaten so, dass schwere Samples beim nächsten Training höheres Gewicht haben. Output ist gewichteter Mittelwert der Stümpfe. Inhärent sequentiell.	
DistBoost	Jeder trainiert Entscheidungsstümpfe anhand lokaler Daten mit globaler Gewichtung. Anhand aller Entscheidungsstümpfe eines Schrittes (summiert) werden die Samples neu gewichtet. IID wird verletzt.	

PreWeak	Jeder Prozessor baut mit AdaBoost lokal T schwache Klassifikatoren und gemeinsam wird Fehlermatrix für jeden Klassifikator berechnet: $E : [(T \cdot K) \times n]$. Master-Server macht AdaBoost ohne Trainieren neuer Klassifikatoren, sondern wählt einen aus, der die gewichtete Fehlermatrix minimiert. Update die Gewichte.	Verhältnismäßig viel Kommunikation (Allgather aller schwachen Klassifikatoren)
AdaSampling	AdaBoost bei jedem Prozessor auf lokalen Daten laufen lassen, nach Gewichten sortieren und die n/K Samples broadcasten mit dem niedrigsten Testerror. Master macht AdaBoost mit n Samples die verteilt wurden.	Weniger Kommunikation als PreWeak
Gradient Boosting	Konstruiere starkes Modell als Gradientabstieg über schwache Modelle (Submodell anstelle von Parametern). $f_t(x) = w_{q(x)}$, w enthält Werte der Blätter, q bestimmt, welches Blatt ausgewählt wird. Die Objective durch Taylor-Entwicklung 2. Ordnung approximieren. Optimiere ein Blatt nach dem anderen.	
XGBoost	Open-Source Software Library für Gradient Boosting Algorithmen.	
Stacking	Jedes Ensemblemitglied ist ein anderer ML-Algorithmus $\rightsquigarrow$ Diversität. Ein Modell kombiniert Vorhersagen.	Modelle können unterschiedliche Rechenkomplexitäten haben $\rightsquigarrow$ Task-Graphen und Scheduler
	Exotische Architekturen	
Quanten-annealing	Formuliere Zielfunktion als Minimierung der Energie eines Systems. Jedes Qubit entspricht einer Variable. Initial Gleichverteilt als ungerichteter Graph mit Qubits als Knoten. System entwickelt sich langsam zu niedrigenergetischem Zustand (Quantentunneln) $\rightsquigarrow$ Lösung des ursprünglichen Optimierungsproblems	

Grover's Search	Ziel: Finde x_0, sodass $f(x_0) = 1$. Initial alle Qubits gleichverteilt. Quantenschaltkreis C (unitär), sodass x_0 geflipt wird und der Rest gleich bleibt. Dann erhöht sich Amplitude von x_0 und andere Ampituden verringern sich. Kann Lernalgorithmen beschelunigen, die auf unstrukturierter Suche basieren (bspw. k-medians)	Erwartet $\frac{\pi}{4}\sqrt{n}$ Queries (klassisch erwartet $n/2$ Queries)
Spiking Neural Networks	Anlehnung ans Gehirn: Impuls wird nur ab Schwellwert weitergeleitet, keine Weiterleitung in Refraktärzeit. SNNs haben Zeit als Konzept für Informationsweiterleitung.	
Direct Random Target Projection (DRTP)	Keine Backpropagation (biologisch implausibel und ineffizient). Stattdessen wird Zufallsvektor mit Target als Ersatz für Gradienten verwendet. Gewichtstransportfrei, Update Locking-frei, direkter Feedbackweg, hohe Trainingsgeschwindigkeit.	
Intelligence Processing Units (IPU)	Hardware mit Knoten als Operatoren/-Daten mit Zustand und Verbindungskanten. Task-Graph kann auf Hardware kompiliert werden, sodass alle Rechen- und Synchronisationsschritte stattfinden (immer all-to-all Kommunikation). Scheduler handhabt alles.	Große Skalierbarkeit