Optimising nested functions using auxiliary coordinates

Miguel Á. Carreira-Perpiñán

Electrical Engineering and Computer Science

University of California, Merced

`http://faculty.ucmerced.edu/mcarreira-perpinan`

work with **Mehdi Alizadeh**, **Ramin Raziperchikolaei**, **Max Vladymyrov** and **Weiran Wang**

Outline

- Nested (deep) and shallow systems
- Training nested systems: chain-rule gradient; greedy layerwise
- The method of auxiliary coordinates (MAC)
  - Design pattern
  - Convergence guarantees
  - Practicalities
  - Related work
  - Model selection “on the fly”
- A gallery of nested models trainable with MAC
  1. Learning low-dimensional features for classification: low-dim SVM
  2. Parametric nonlinear embeddings: parametric t-SNE, EE
  3. Binary hashing for fast image search: binary autoencoder
  4. Learning radial basis function (RBF) networks
  5. Learning feature transformations for decision trees: neural net + tree
  6. Best-subset feature selection
- Distributed optimisation with MAC: ParMAC

Nested systems: examples

Common in computer vision, speech processing, machine learning…

- Object recognition pipeline:

  image pixels $\rightarrow$ SIFT/HoG $\rightarrow$ $\begin{array}{c}k\text{-means}\\ \text{sparse coding}\end{array}$ $\rightarrow$ pooling $\rightarrow$ classifier $\rightarrow$ object category

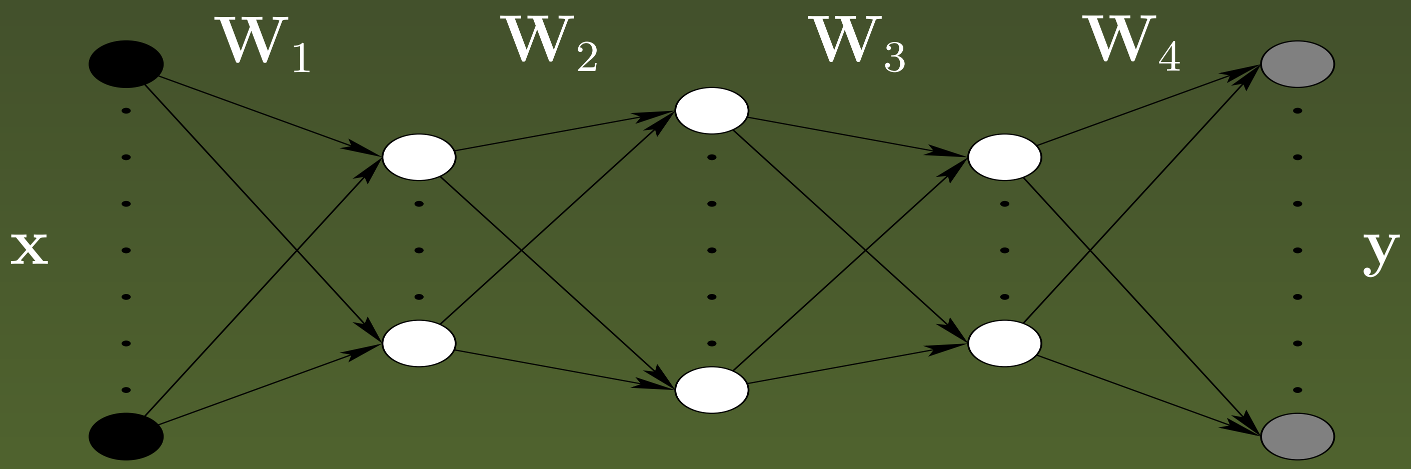
- Phone classification pipeline:

  waveform $\rightarrow$ MFCC/PLP $\rightarrow$ classifier $\rightarrow$ phoneme label

- Preprocessing for regression/classification/search/etc.:

  image pixels $\rightarrow$ PCA/LDA $\rightarrow$ classifier $\rightarrow$ output/label
  image pixels $\rightarrow$ projection $\rightarrow$ thresholding $\rightarrow$ search $\rightarrow$ nearest neighbour index

- Deep net: $\mathbf{x} \rightarrow \{\sigma(\mathbf{w}_i^T\mathbf{x} + a_i)\} \rightarrow \{\sigma(\mathbf{w}_j^T\{\sigma(\mathbf{w}_i^T\mathbf{x} + a_i)\}) + b_j\} \rightarrow \cdots \rightarrow \mathbf{y}$



Nested systems

Mathematically, they construct a (deeply) nested, parametric mapping from inputs to outputs:

$$\mathbf{f}(\mathbf{x};\mathbf{W}) = \mathbf{f}_{K+1}(\dots\mathbf{f}_2(\mathbf{f}_1(\mathbf{x};\mathbf{W}_1);\mathbf{W}_2)\dots;\mathbf{W}_{K+1})$$

$$\begin{tikzcd} \mathbf{x} \arrow[r] & \boxed{\mathbf{f}_1,\mathbf{W}_1} \arrow[r] & \boxed{\mathbf{f}_2,\mathbf{W}_2} \arrow[r] & \boxed{\mathbf{f}_3,\mathbf{W}_3} \arrow[r] & \boxed{\mathbf{f}_4,\mathbf{W}_4} \arrow[r] & \boxed{\mathbf{f}_5,\mathbf{W}_5} \arrow[r] & \mathbf{y} \end{tikzcd}$$

- Each layer (processing stage) has its own trainable parameters (weights) $\mathbf{W}_k$.
- Each layer performs some (nonlinear, nondifferentiable) processing on its input, extracting ever more sophisticated features from it
  (ex.: pixels $\rightarrow$ edges $\rightarrow$ parts $\rightarrow \cdots$)
- Often inspired by biological brain processing
  (e.g. retina $\rightarrow$ LGN $\rightarrow$ V1 $\rightarrow \cdots$)
- The ideal performance is when the parameters at all layers are jointly optimised towards the overall goal (e.g. classification error).
  This work is about how to do this easily and efficiently.

Shallow vs deep (nested) systems

Shallow systems: 0 to 1 hidden layer between input and output.

- Often convex problem: linear function, linear SVM, Lasso, etc.

  ...Or "forced" to be convex: $\mathbf{f}(\mathbf{x}) = \sum_{m=1}^{M} \mathbf{w}_m \phi_m(\mathbf{x})$:
  - RBF network: fix nonlinear basis functions ϕ_m (e.g. k-means), then fix linear weights $\mathbf{w}_m$.
  - Kernel SVM: basis functions (support vectors) result from a QP.
- Practically useful:
  - Linear function: robust (particularly with high-dim data, small samples).
  - Nonlinear function: very accurate if using many BFs (wide hidden layer).
- Easy to train: no local optima; no need for nonlinear optimisation

  linear system, LP/QP, eigenproblem, etc.

Shallow vs deep (nested) systems (cont.)

Deep (nested) systems: at least one hidden layer:

- Examples: deep nets; “wrapper” regression/classification; CV/speech pipelines.
- Nearly always nonconvex.

  The composition of functions is nonconvex in general.
- Practically useful: powerful nonlinear function.

  Depending on the number of layers and of hidden units/BFs.
- May be better than shallow systems for some problems.
- Difficult to train: local optima; requires nonlinear optimisation, or suboptimal approach.

How does one train a nested system?

Training nested systems: backpropagated gradient

- Apply the chain rule, layer by layer, to obtain a gradient wrt all the parameters.

  Ex.: $\frac{\partial}{\partial \mathbf{g}}(\mathbf{g}(\mathbf{F}(\cdot))) = \mathbf{g}'(\mathbf{F}(\cdot))$, $\frac{\partial}{\partial \mathbf{F}}(\mathbf{g}(\mathbf{F}(\cdot))) = \mathbf{g}'(\mathbf{F}(\cdot))\,\mathbf{F}'(\cdot)$.

  Then feed to nonlinear optimiser.

  Gradient descent, CG, L-BFGS, Levenberg-Marquardt, Newton, etc.
- Major breakthrough in the 80s with neural nets (and in the 2010s).

  It allowed to train multilayer perceptrons, and now deep nets, from data.
- Disadvantages:
  - requires differentiable layers in order to apply the chain rule
  - the gradient is cumbersome to compute, code and debug

    this may be avoided with automatic differentiation
  - requires nonlinear optimisation
  - vanishing gradients $\Rightarrow$ ill-conditioning $\Rightarrow$ slow progress even with second-order methods

    this gets worse the more layers we have
  - difficult to parallelise (layer gradients must be computed sequentially).

Training nested systems: layerwise, “filter”

- Fix each layer sequentially from the input to the output (in some way).
- Fast and easy, but suboptimal.

  The resulting parameters are not a minimiser of the joint objective function.
  Sometimes the results are not very good.
- Sometimes used to initialise the parameters and refine the model with backpropagation (“fine tuning”).

Examples:

- Deep nets:
  - Unsupervised pretraining (Hinton & Salakhutdinov 2006)
  - Supervised greedy layerwise training (Bengio et al. 2007)
- RBF networks: the centres of the first (nonlinear) layer’s basis functions are set in an unsupervised way

  k-means, random subset

Training nested systems: layerwise, “filter” (cont.)

“Filter” vs “wrapper” approaches: consider a nested mapping $\mathbf{g}(\mathbf{F}(\mathbf{x}))$ (e.g. $\mathbf{F}$ reduces dimension, $\mathbf{g}$ classifies). How to train $\mathbf{F}$ and $\mathbf{g}$?

Filter approach:

- Greedy sequential training:
  1. Train $\mathbf{F}$ (the “filter”):
     - Unsupervised: use only the input data $\{\mathbf{x}_n\}$

       PCA, k-means, etc.
     - Supervised: use the input and output data $\{(\mathbf{x}_n, \mathbf{y}_n)\}$

       LDA, sliced inverse regression, etc.
  2. Fix $\mathbf{F}$, train $\mathbf{g}$: fit a classifier with inputs $\{\mathbf{F}(\mathbf{x}_n)\}$ and labels $\{\mathbf{y}_n\}$.
- Very popular; $\mathbf{F}$ is often a fixed “preprocessing” or “feature extraction” stage which can be done with existing algorithms.
- Works well if using a good objective function for $\mathbf{F}$.
- . . . But is still suboptimal: the preprocessing may not be the best possible for classification.

Training nested systems: layerwise, “filter” (cont.)

Wrapper approach:

- Train $\mathbf{F}$ and $\mathbf{g}$ jointly to minimise the classification error.
  This is what we would like to do.
- Optimal: the preprocessing is the best possible for classification.
- Even if local optima exist, initialising it from the “filter” result will give a better model.
- Less frequently done in practice.
- Disadvantage: same problems as with backpropagation.
  Requires a chain rule gradient, difficult to compute, nonlinear optimisation, slow.

Training nested systems: model selection

Finally, we also have to select the best architecture:

- Number of units or basis functions in each layer of a deep net; number of filterbanks in a speech front-end processing; etc.
- Requires a combinatorial (grid or random) search, training models for each hyperparameter choice and picking the best
  according to a model selection criterion, cross-validation, etc.
- In practice, this is approximated using expert know-how:
  - Train only a few models, pick the best from there.
  - Fix the parameters of some layers irrespective of the rest of the pipeline.

Very costly in runtime, in effort and expertise required, and leads to suboptimal solutions.

Summary

Nested systems:

- Ubiquitous way to construct nonlinear trainable functions
- Powerful
- Intuitive
- Nonconvex, nonlinear, maybe nondifferentiable so difficult to train:
  - Layerwise: easy but suboptimal
  - Backpropagation: optimal but slow, difficult to implement, needs differentiable layers.

Let us see a different, general way to train nested systems.

The method of auxiliary coordinates (MAC)

M. Carreira-Perpiñán and W. Wang: *Distributed optimization of deeply nested systems*, AISTATS 2014, arXiv:1212.5921.

- A general strategy to train all the parameters of a nested system. Not an algorithm but a meta-algorithm (like EM).
- Allows layerwise training (simple submodels trainable by reusing existing algorithms) and introduces parallelism.

Basic idea (design pattern):

1. Turn the nested problem $E_{\text{nested}}(\mathbf{W})$ into an unnested, constrained problem by introducing new parameters to be optimised over (the auxiliary coordinates $\mathbf{Z}$) that decouple the layers.
2. Optimise the constrained problem with a penalty method.
   Augmented Lagrangian, quadratic penalty, etc.
3. Apply alternating optimisation to the penalised function:
   - Over $\mathbf{W}$: layers step (reuse a single-layer training algorithm, typically)
   - Over $\mathbf{Z}$: coordinates step (needs to be solved specially for each case)

Result: alternate “layerwise training” steps with “coordination” steps.

The nested objective function

Consider for simplicity:

- a single hidden layer: $\mathbf{x} \to \mathbf{F}(\mathbf{x}) \to \mathbf{g}(\mathbf{F}(\mathbf{x}))$
- a least-squares regression for inputs $\{\mathbf{x}_n\}_{n=1}^N$ and outputs $\{\mathbf{y}_n\}_{n=1}^N$:

$$\min E_{\text{nested}}(\mathbf{F}, \mathbf{g}) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{F}(\mathbf{x}_n))\|^2$$

$\mathbf{F}$, $\mathbf{g}$ have their own parameters (weights).
We want to find a local minimiser of E_{nested}.

The MAC-constrained problem

Transform the nested problem into a constrained problem in an augmented space:

$$\min E(\mathbf{F}, \mathbf{g}, \mathbf{Z}) = \frac{1}{2} \sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{g}(\mathbf{z}_n) \right\|^2$$

$$\text{s.t. } \mathbf{z}_n = \mathbf{F}(\mathbf{x}_n) \quad n = 1, \dots, N.$$

- For each data point, we turn the subexpression $\mathbf{F}(\mathbf{x}_n)$ into an equality constraint associated with a new parameter $\mathbf{z}_n$ (the auxiliary coordinates).

  Thus, a constrained problem with N equality constraints and new parameters $\mathbf{Z} = (\mathbf{z}_1, \dots, \mathbf{z}_N)$.
- We optimise over $(\mathbf{F}, \mathbf{g})$ and $\mathbf{Z}$ jointly.
- Equivalent to the nested problem.

The MAC-penalised function

We solve the constrained problem with the quadratic-penalty method: we minimise the following while driving the penalty parameter $\mu \to \infty$:

$$\min E_Q(\mathbf{F}, \mathbf{g}, \mathbf{Z}; \mu) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \underbrace{\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2}_{\text{constraints as quadratic penalties}}$$

We can also use the augmented Lagrangian method instead:

$$\begin{aligned}\min E_{\mathcal{L}}(\mathbf{F}, \mathbf{g}, \mathbf{Z}, \boldsymbol{\Lambda}; \mu) &= \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2 + \sum_{n=1}^{N} \boldsymbol{\lambda}_n^T (\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)) + \frac{\mu}{2} \sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 \\ &= \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n) - \tfrac{1}{\mu}\boldsymbol{\lambda}_n\right\|^2\end{aligned}$$

which does an extra update of the Lagrange multipliers: $\boldsymbol{\lambda}_n \leftarrow \boldsymbol{\lambda}_n - \mu(\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)),\ n = 1, \dots, N$.
For simplicity, we focus on the quadratic-penalty method.

What have we achieved?

- Net effect: unfold the nested objective into shallow additive terms connected by the auxiliary coordinates:

$$E_{\text{nested}}(\mathbf{F},\mathbf{g}) = \frac{1}{2}\sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{g}(\mathbf{F}(\mathbf{x}_n)) \right\|^2 \Longrightarrow$$

$$E_Q(\mathbf{F},\mathbf{g},\mathbf{Z};\mu) = \frac{1}{2}\sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{g}(\mathbf{z}_n) \right\|^2 + \frac{\mu}{2}\sum_{n=1}^{N} \left\| \mathbf{z}_n - \mathbf{F}(\mathbf{x}_n) \right\|^2$$

- All terms equally scaled, but uncoupled.

  Vanishing gradients less problematic.
  Derivatives required are simpler: no backpropagated gradients, sometimes no gradients at all.

- Optimising E_{nested} follows a convoluted trajectory in $(\mathbf{F},\mathbf{g})$ space.
- Optimising E_Q can take shortcuts by jumping across $\mathbf{Z}$ space.

  This corresponds to letting the layers mismatch during the optimisation.

Alternating optimisation of the MAC-penalised function

$(\mathbf{F}, \mathbf{g})$ step, for $\mathbf{Z}$ fixed:

$$\min_{\mathbf{F}} \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 \qquad \min_{\mathbf{g}} \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2$$

- Layerwise training: each layer is trained independently (not sequentially):
  - fit $\mathbf{F}$ to $\{(\mathbf{x}_n, \mathbf{z}_n)\}_{n=1}^N$ (gradient needed: $\mathbf{F}'(\cdot)$)
  - fit $\mathbf{g}$ to $\{(\mathbf{z}_n, \mathbf{y}_n)\}_{n=1}^N$ (gradient needed: $\mathbf{g}'(\cdot)$)

  This looks like a filter approach with intermediate “features” $\{\mathbf{z}_n\}_{n=1}^N$.
- Usually simple fit, even convex.
- Can be done by using existing algorithms for shallow models

  linear, logistic regression, SVM, RBF network, k-means, decision tree, etc.

  Does not require backpropagated gradients.

Alternating opt. of the MAC-penalised function (cont.)

$\mathbf{Z}$ step, for $(\mathbf{F}, \mathbf{g})$ fixed:

$$\min_{\mathbf{z}_n} \quad \frac{1}{2}\|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2 + \frac{\mu}{2}\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 \qquad n = 1, \dots, N$$

- The auxiliary coordinates are trained independently for each point.
  N small problems (of size $|\mathbf{z}|$) instead of one large problem (of size $N|\mathbf{z}|$).
- They "coordinate" the layers.
- The $\mathbf{Z}$ step has the form of a proximal operator:
  $\min_{\mathbf{z}} f(\mathbf{z}) + \frac{\mu}{2}\|\mathbf{z} - \mathbf{u}\|^2$
  The solution has a geometric flavour ("projection").
- Often closed-form (depending on the model).

Alternating opt. of the MAC-penalised function (cont.)

We can view MAC is a “coordination-minimisation” (CM) algorithm:

- M step: minimise (train) layers
- C step: coordinate layers.

The coordination step is crucial: it ensures we converge to a minimum of the nested function (which layerwise training by itself does not do).

MAC is different from pure alternating optimisation over layers:

$$\min E_{\text{nested}}(\mathbf{F},\mathbf{g}) = \frac{1}{2}\sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{g}(\mathbf{F}(\mathbf{x}_n))\right\|^2$$

- Over $\mathbf{g}$ for fixed $\mathbf{F}$: fit $\mathbf{g}$ to $\{(\mathbf{F}(\mathbf{x}_n),\mathbf{y}_n)\}_{n=1}^N$ (needs $\mathbf{g}'(\cdot)$)
- Over $\mathbf{F}$ for fixed $\mathbf{g}$: needs backprop. gradients over $\mathbf{F}$ $(\mathbf{g}'(\mathbf{F}(\cdot))\,\mathbf{F}'(\cdot))$

Pure alternating optimisation $\neq$ “layerwise training”.

MAC in general

MAC applies very generally:

- More complex nesting: K layers, not necessarily feedforward (e.g. recurrent), with shared parameters. . .
- Various loss functions, full/sparse layer connectivity, constraints. . .

The resulting optimisation algorithm depends on:

- the type of layers $\mathbf{f}_k$ used in the nested model

  linear, logistic regression, SVM, RBF network, decision tree. . .
- how/where we introduce the auxiliary coordinates

  at no/some/all layers; before/after the nonlinearity; can even redefine $\mathbf{Z}$ during the optimisation
- how we optimise each step

  choice of penalty function and step optimisation method; no need to update all $\mathbf{W}_k$ or $\mathbf{z}_{kn}$ at each step; inexact steps, warm start, caching factorisations, stochastic updates. . .

The resulting MAC algorithm should handle nondifferentiable problems, introduce parallelism and reuse existing algorithms as a black box, by breaking the nested problem into many small, unnested subproblems.

MAC in general: K layers

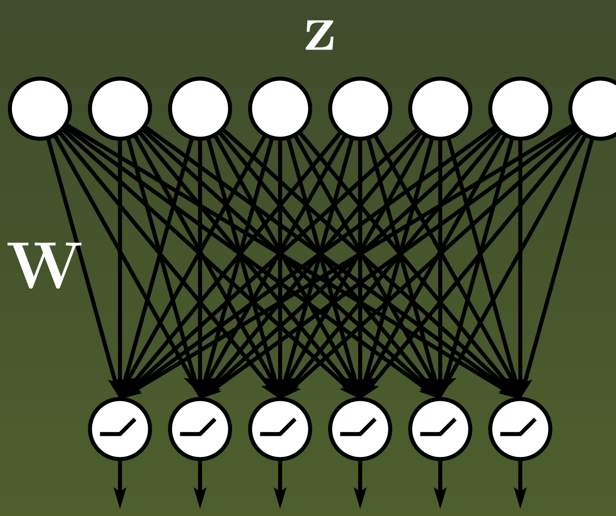
$$\mathbf{y} = \mathbf{f}(\mathbf{x}; \mathbf{W}) = \mathbf{f}_{K+1}(\dots \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}; \mathbf{W}_1); \mathbf{W}_2) \dots ; \mathbf{W}_{K+1})$$

$$\begin{tikzcd}
\mathbf{x} \arrow[r] & \boxed{\mathbf{f}_1, \mathbf{W}_1} \arrow[r] & \boxed{\mathbf{f}_2, \mathbf{W}_2} \arrow[r] & \boxed{\mathbf{f}_3, \mathbf{W}_3} \arrow[r] & \boxed{\mathbf{f}_4, \mathbf{W}_4} \arrow[r] & \boxed{\mathbf{f}_5, \mathbf{W}_5} \arrow[r] & \mathbf{y}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{x} \arrow[r] & \boxed{\mathbf{f}_1, \mathbf{W}_1} \arrow[dr] & & \boxed{\mathbf{f}_2, \mathbf{W}_2} \arrow[dr] & & \boxed{\mathbf{f}_3, \mathbf{W}_3} \arrow[dr] & & \boxed{\mathbf{f}_4, \mathbf{W}_4} \arrow[dr] & & \boxed{\mathbf{f}_5, \mathbf{W}_5} \arrow[r] & \mathbf{y} \\
 & & \boxed{\mathbf{z}_1} \arrow[ur] & & \boxed{\mathbf{z}_2} \arrow[ur] & & \boxed{\mathbf{z}_3} \arrow[ur] & & \boxed{\mathbf{z}_4} \arrow[ur] & &
\end{tikzcd}$$

Example: feedforward neural nets:

- each layer has the form $\mathbf{f}_k(\mathbf{z}; \mathbf{W}_k) = \sigma(\mathbf{W}_k \mathbf{z})$
- each row of $\mathbf{W}_k$ is one hidden unit
- $\sigma(\cdot)$ is an elementwise nonlinearity (sigmoid, tanh, ReLU, etc.).



$$\mathbf{f}_k(\mathbf{z}; \mathbf{W}_k) = \sigma(\mathbf{W}_k \mathbf{z})$$

MAC in general: K layers (cont.)

The nested objective function:

$$E_{\text{nested}}(\mathbf{W}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}(\mathbf{x}_n;\mathbf{W})\|^2, \qquad \mathbf{f}(\mathbf{x};\mathbf{W}) = \mathbf{f}_{K+1}(\dots\mathbf{f}_2(\mathbf{f}_1(\mathbf{x};\mathbf{W}_1);\mathbf{W}_2)\dots;\mathbf{W}_{K+1}).$$

❶ The MAC-constrained problem:

$$E(\mathbf{W},\mathbf{Z}) = \frac{1}{2}\sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n};\mathbf{W}_{K+1})\right\|^2 \quad \text{s.t.} \quad \left\{\begin{array}{l} \mathbf{z}_{K,n} = \mathbf{f}_K(\mathbf{z}_{K-1,n};\mathbf{W}_K) \\ \dots \\ \mathbf{z}_{1,n} = \mathbf{f}_1(\mathbf{x}_n;\mathbf{W}_1) \end{array}\right\} n = 1,\dots,N.$$

❷ The MAC quadratic-penalty function:

$$E_Q(\mathbf{W},\mathbf{Z};\mu) = \frac{1}{2}\sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n};\mathbf{W}_{K+1})\right\|^2 + \frac{\mu}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} \left\|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n};\mathbf{W}_k)\right\|^2.$$

❸ Alternating optimisation:

- **W** step: $\min_{\mathbf{W}_k} \sum_{n=1}^{N} \left\|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n};\mathbf{W}_k)\right\|^2, \; k = 1\dots,K+1$ (if $\mathbf{f}_k = \sigma(\mathbf{W}_k\mathbf{z})$: separates over each row ("hidden unit") of $\mathbf{W}_k$)

  Separates over each of the layers.

- **Z** step: $\min_{\mathbf{z}_n} \left(\frac{1}{2}\left\|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n})\right\|^2 + \frac{\mu}{2}\sum_{k=1}^{K}\left\|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n})\right\|^2\right), \; n = 1,\dots,N$

  Separates over each of the N training points.

$\mathbf{Z}$ step in K layers: “even-odd” alternating opt. ($^{\text{unpublished}}_{\text{results}}$)

$$\mathbf{Z}\text{ step: } \min_{\mathbf{z}_n} \left(\frac{1}{2} \| \mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n}) \|^2 + \frac{\mu}{2} \sum_{k=1}^{K} \| \mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n}) \|^2 \right)$$

$$\begin{tikzcd}
\mathbf{x} \arrow[r] & \boxed{\mathbf{f}_1, \mathbf{W}_1} \arrow[dr] & & \boxed{\mathbf{f}_2, \mathbf{W}_2} \arrow[dr] & & \boxed{\mathbf{f}_3, \mathbf{W}_3} \arrow[dr] & & \boxed{\mathbf{f}_4, \mathbf{W}_4} \arrow[dr] & & \boxed{\mathbf{f}_5, \mathbf{W}_5} \arrow[r] & \mathbf{y} \\
& & \boxed{\mathbf{z}_1} \arrow[ur] & & \boxed{\mathbf{z}_2} \arrow[ur] & & \boxed{\mathbf{z}_3} \arrow[ur] & & \boxed{\mathbf{z}_4} \arrow[ur] & &
\end{tikzcd}$$

$$\|\mathbf{z}_{1,n} - \mathbf{f}_1(\mathbf{x}_n)\|^2 + \|\mathbf{z}_{2,n} - \mathbf{f}_2(\mathbf{z}_{1,n})\|^2 + \|\mathbf{z}_{3,n} - \mathbf{f}_3(\mathbf{z}_{2,n})\|^2 + \|\mathbf{z}_{4,n} - \mathbf{f}_4(\mathbf{z}_{3,n})\|^2 + \frac{1}{\mu}\|\mathbf{y}_n - \mathbf{f}_5(\mathbf{z}_{4,n})\|^2$$

Alternating optimisation over the odd and even layers of $\mathbf{z}_n = \{\mathbf{z}_{k,n}\}$ results in steps where the layers separate:

- over $\{\mathbf{z}_{k,n}\}$ for odd k given all even k: separates over $k = 1, 3, 5 \dots$
- over $\{\mathbf{z}_{k,n}\}$ for even k given all odd k: separates over $k = 2, 4, 6 \dots$

And each subproblem over layer k has the same form (proximal operator):

$$\min_{\mathbf{z}_{k,n}} \left(\|\mathbf{z}_{k+1,n} - \mathbf{f}_{k+1}(\mathbf{z}_{k,n})\|^2 + \|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n})\|^2 \right)$$

Several ways to introduce auxiliary coordinates (unpublished results

With neural net layers of the form $\mathbf{f}_k(\mathbf{z};\mathbf{W}_k) = \sigma(\mathbf{W}_k\mathbf{z})$, we can introduce auxiliary coordinates before, after, or before and after the nonlinearity σ, resulting in subproblems of different form and separability. Consider the nested function $\sigma\big(\mathbf{W}_2\,\sigma(\mathbf{W}_1\mathbf{x})\big)$:

- after: $\overbrace{\sigma\Big(\mathbf{W}_2\;\underbrace{\sigma(\mathbf{W}_1\mathbf{x})}_{\mathbf{f}_1}\Big)}^{\mathbf{f}_2} \Rightarrow \begin{cases} \mathbf{z}_2 = \sigma(\mathbf{W}_2\mathbf{z}_1) \\ \mathbf{z}_1 = \sigma(\mathbf{W}_1\mathbf{x}) \end{cases}$

  $\mathbf{W}_k$: logistic regression problem
  $\mathbf{z}_k$: nonlinear problem.

- before: $\sigma\Big(\overbrace{\mathbf{W}_2\,\sigma(\underbrace{\mathbf{W}_1\mathbf{x}}_{\mathbf{f}_1})}^{\mathbf{f}_2}\Big) \Rightarrow \begin{cases} \mathbf{z}_2 = \mathbf{W}_2\,\sigma(\mathbf{z}_1) \\ \mathbf{z}_1 = \mathbf{W}_1\mathbf{x} \end{cases}$

  $\mathbf{W}_k$: quadratic problem
  $\mathbf{z}_k$: nonlinear problem.

- before and after: $\sigma\Big(\overbrace{\overbrace{\mathbf{W}_2\;\underbrace{\sigma(\underbrace{\mathbf{W}_1\mathbf{x}})}}}\Big) \Rightarrow \begin{cases} \mathbf{u}_2 = \sigma(\mathbf{z}_2) \\ \mathbf{z}_2 = \mathbf{W}_2\mathbf{u}_1 \\ \mathbf{u}_1 = \sigma(\mathbf{z}_1) \\ \mathbf{z}_1 = \mathbf{W}_1\mathbf{x} \end{cases}$

  $\mathbf{W}_k$: quadratic problem;
  $\mathbf{u}_k$: quadratic problem;
  $\mathbf{z}_k$: nonlinear scalar problems,
  can even handle ReLU;
  alt. opt. over $(\mathbf{U},\mathbf{Z})$ separates $\{\mathbf{u}_k\}$ given $\mathbf{Z}$, and $\{\mathbf{z}_k\}$ given $\mathbf{U}$.

"Closest point on a sigmoid": $\arg\min_z\,(y-\sigma(z))^2 + (z-x)^2$ for each element of $\mathbf{z}_k$.

MAC in general: the design pattern

How to train your system using auxiliary coordinates:

Write your nested objective function $E_{\text{nested}}(\mathbf{W})$.

❶ Identify subexpressions and turn them into auxiliary coordinates $\mathbf{Z}$ with equality constraints so the layers decouple.

One auxiliary coordinate vector per training point.

❷ Optimise the constrained problem with a penalty method.

Augmented Lagrangian, quadratic penalty, etc.

❸ Apply alternating optimisation to the penalised function:

- ✦ Over $\mathbf{W}$: layers step (reuse a single-layer training algorithm, typically)
- ✦ Over $\mathbf{Z}$: coordinates step (needs to be solved specially for each case)

  proximal operator; for many important cases closed-form or simple to optimise.

Similar to deriving an EM algorithm.

Define your probability model, write the log-likelihood objective function, identify hidden variables, write the complete-data log-likelihood, obtain E and M steps, solve them.

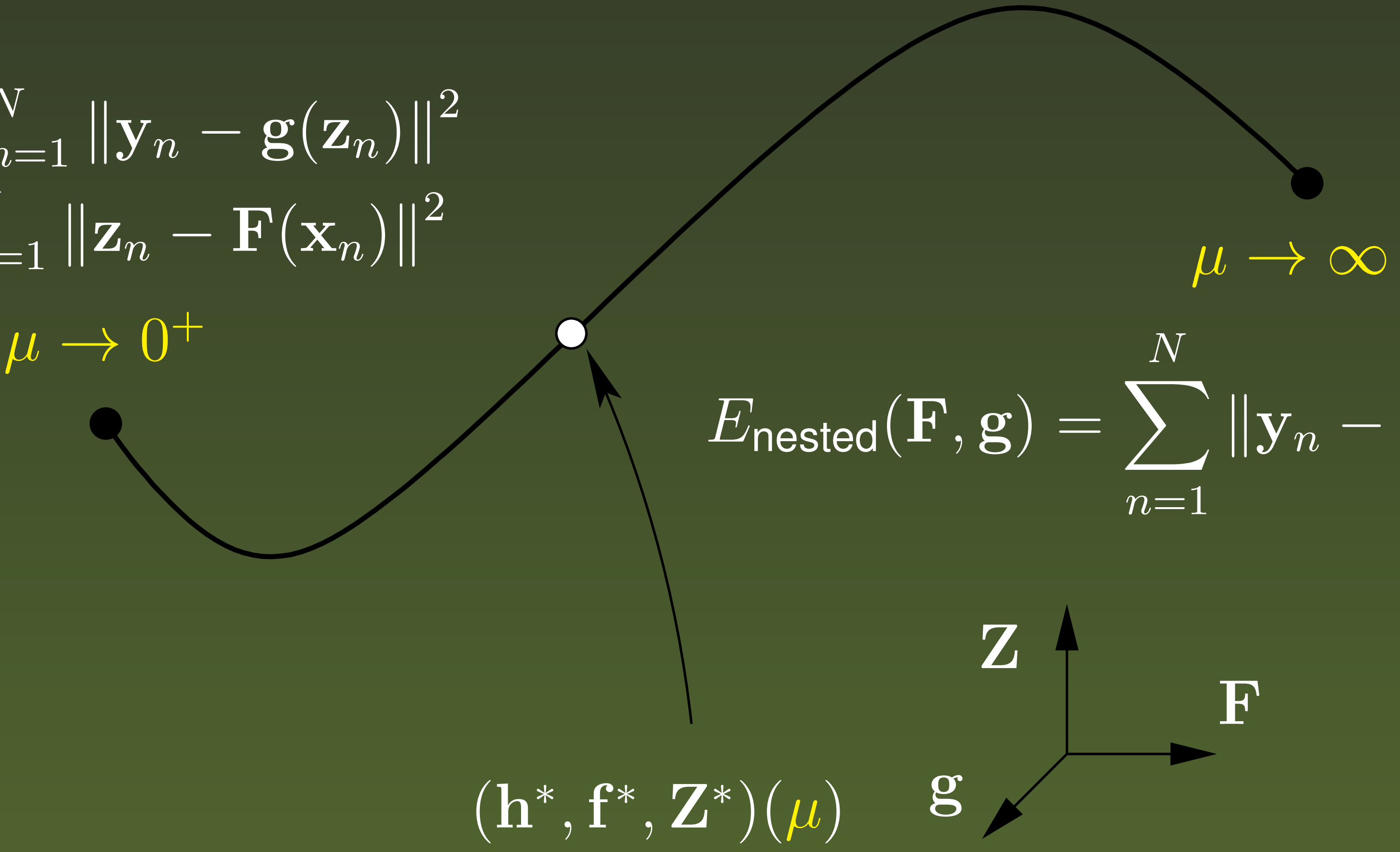
A continuous path induced by μ towards a solution

MAC belongs to the class of homotopy (path-following) methods.

$$E_Q(\mathbf{F},\mathbf{g},\mathbf{Z};\mu) = \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\right\|^2 + \mu \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\right\|^2, \quad \mu \in (0,\infty)$$

$$\min_{\mathbf{g},\mathbf{Z}} \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\right\|^2$$

$$\min_{\mathbf{F}} \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\right\|^2$$



$$E_{\text{nested}}(\mathbf{F},\mathbf{g}) = \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{g}(\mathbf{F}(\mathbf{x}_n))\right\|^2$$

Convergence guarantees

We want a solution (local stationary point) of $E_{\text{nested}}(\mathbf{W})$:

$$\min E_{\text{nested}}(\mathbf{F},\mathbf{g}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{F}(\mathbf{x}_n))\|^2$$

MAC optimises the penalty function E_Q (or $E_{\mathcal{L}}$) by alternating optimisation over $(\mathbf{W},\mathbf{Z})$ while driving $\mu \to \infty$:

$$E_Q(\mathbf{F},\mathbf{g},\mathbf{Z};\mu) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{z}_n)\|^2 + \frac{\mu}{2}\sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2$$

For K layers:

$$E_{\text{nested}}(\mathbf{W}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}(\mathbf{x}_n;\mathbf{W})\|^2, \qquad \mathbf{f}(\mathbf{x};\mathbf{W}) = \mathbf{f}_{K+1}(\dots\mathbf{f}_2(\mathbf{f}_1(\mathbf{x};\mathbf{W}_1);\mathbf{W}_2)\dots;\mathbf{W}_{K+1})$$

$$E_Q(\mathbf{W},\mathbf{Z};\mu) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}_{K+1}(\mathbf{z}_{K,n};\mathbf{W}_{K+1})\|^2 + \frac{\mu}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} \|\mathbf{z}_{k,n} - \mathbf{f}_k(\mathbf{z}_{k-1,n};\mathbf{W}_k)\|^2.$$

Convergence guarantees (cont.)

Does MAC converge to a solution of the nested problem?

It is hard to answer this in general because there are so many different types of nested functions. The layer functions (or constraints) may be convex, nonconvex, non-differentiable or discrete. Let us focus on differentiable functions.

Convergence to a local stationary point of $E_{\text{nested}}(\mathbf{W})$:

1. Would optimising the penalty function over $(\mathbf{W}, \mathbf{Z})$ solve the problem?
   That is, does the path of solutions $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ over $\mu \geq 0$ end at a local stationary point of $E_{\text{nested}}(\mathbf{W})$?
   Yes, for continuously differentiable functions.

2. Would optimising the penalty function by alternating optimisation over $(\mathbf{W}, \mathbf{Z})$ converge to a solution $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ for any given μ?
   Yes, for continuously differentiable functions if the subproblems are nice enough.

Convergence guarantees 1: penalty function

Assuming continuously differentiable functions:

Theorem 1: the nested problem and the MAC-constrained problem are equivalent in the sense that their minimisers, maximisers and saddle points are in a one-to-one correspondence. The KKT conditions for both problems are equivalent.

Optimising the penalty function. For the quadratic penalty E_Q:

Theorem 2: given a positive increasing sequence $(\mu_k) \to \infty$, a nonnegative sequence $(\tau_k) \to 0$, and a starting point $(\mathbf{W}^0, \mathbf{Z}^0)$, suppose the quadratic-penalty method finds an approximate minimizer $(\mathbf{W}^k, \mathbf{Z}^k)$ of $E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k)$ that satisfies $\left\|\nabla_{\mathbf{W},\mathbf{Z}} E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k)\right\| \leq \tau_k$ for $k = 1, 2, \dots$ Then, $\lim_{k\to\infty} (\mathbf{W}^k, \mathbf{Z}^k) = (\mathbf{W}^*, \mathbf{Z}^*)$, which is a KKT point for the nested problem, and its Lagrange multiplier vector has elements $\boldsymbol{\lambda}_n^* = \lim_{k\to\infty} -\mu_k \left(\mathbf{Z}_n^k - \mathbf{F}(\mathbf{Z}_n^k, \mathbf{W}^k; \mathbf{x}_n)\right),\ n = 1, \dots, N$.

That is, MAC defines a continuous path $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ that converges (as $\mu \to \infty$) to a local stationary point of the constrained problem and thus to a local stationary point of the nested problem.

In practice, we follow this path loosely.

The above theorem is for the quadratic penalty. Similar theorems exist for the augmented Lagrangian $E_{\mathcal{L}}$.

Convergence guarantees 2: alternating optimisation

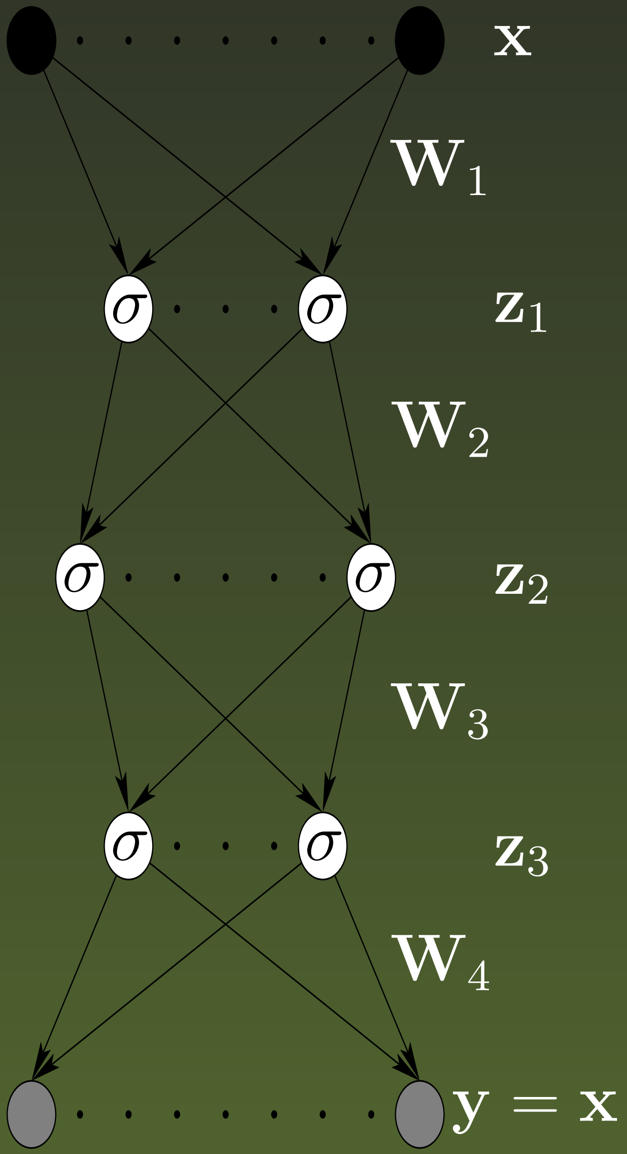
Alternating optimisation of E_Q for fixed μ over $\mathbf{W}$ and $\mathbf{Z}$ does monotonically decrease E_Q (assuming the steps over $\mathbf{W}$ and $\mathbf{Z}$ can be solved, even approximately).
But, does alternating optimisation of E_Q for fixed μ over $\mathbf{W}$ and $\mathbf{Z}$ converge to a local stationary point $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ of E_Q?
That is, can we follow the path over μ as closely as desired by alternating over $\mathbf{W}$ and over $\mathbf{Z}$?

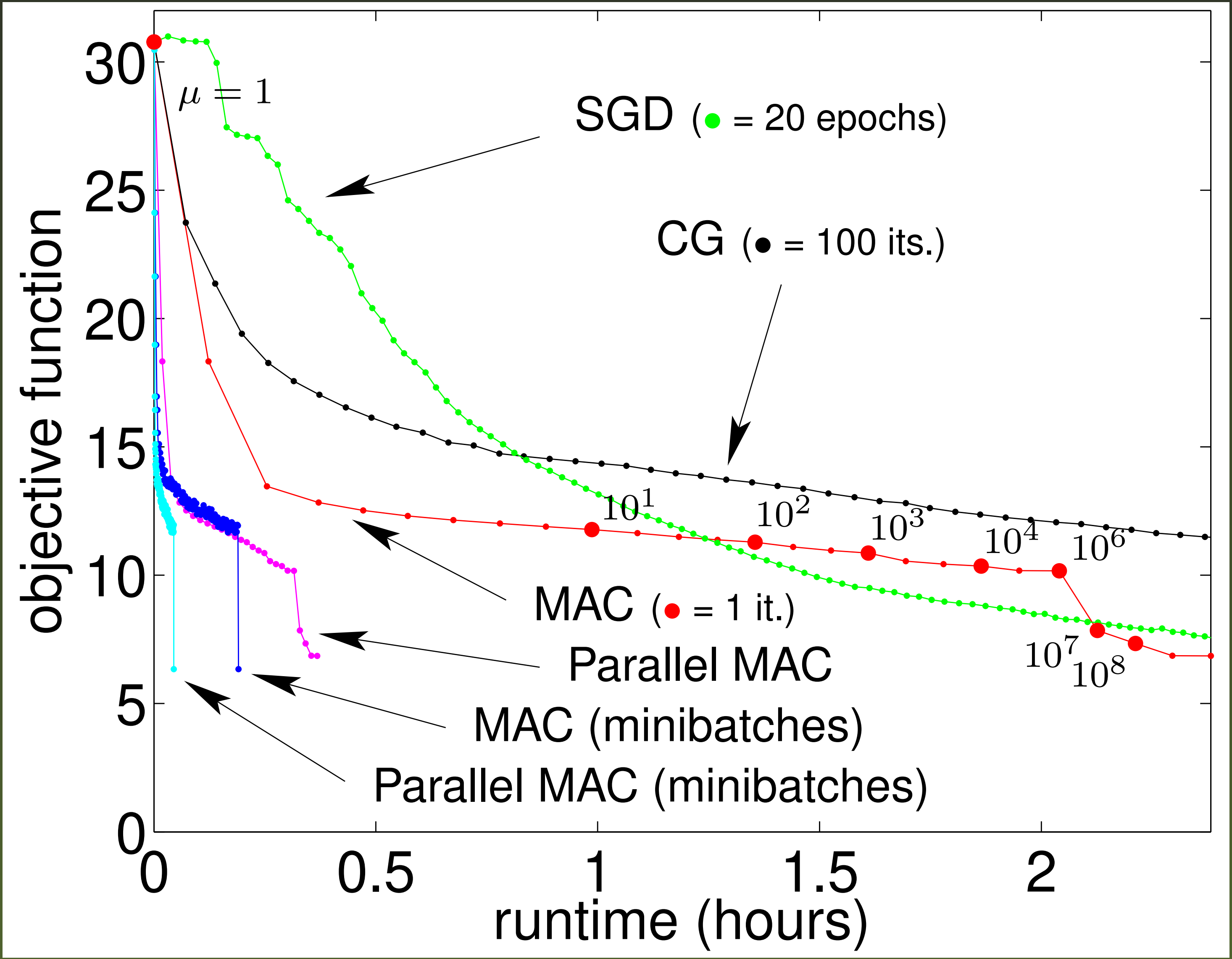
Assuming differentiable functions:

- If E_Q is convex, yes quite generally.
- If E_Q is nonconvex: convergence results are complex and more restrictive. One simple case where convergence occurs is if there is a unique minimizer along each block $\mathbf{W}$ and $\mathbf{Z}$, hence along each layer subproblem $\mathbf{W}_k$ and auxiliary coordinate vector $\mathbf{z}_n$.

Experiment: deep sigmoidal autoencoder

USPS handwritten digits, 256–300–100–20–100–300–256 autoencoder ($K = 5$ logistic layers), auxiliary coordinates at each hidden layer, random initial weights. $\mathbf{W}$ and $\mathbf{Z}$ steps use Gauss-Newton.





Experiment: deep sigmoidal autoencoder (cont.)

Typical behaviour in practice:

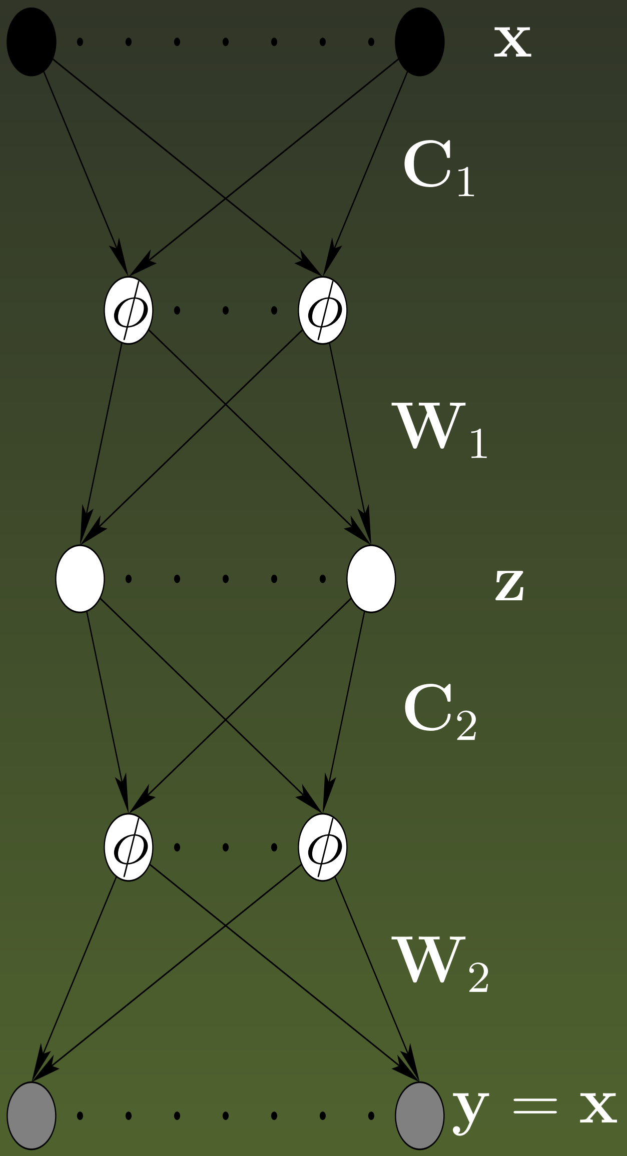
- Very large error decrease at the beginning, causing large changes to the parameters at all layers
  unlike backpropagation-based methods.
- Eventually slows down, slow convergence
  typical of alternating optimisation algorithms.
- “Pretty good net pretty fast”.
- Competitive with state-of-the-art nonlinear optimisers, particularly with many nonlinear layers.

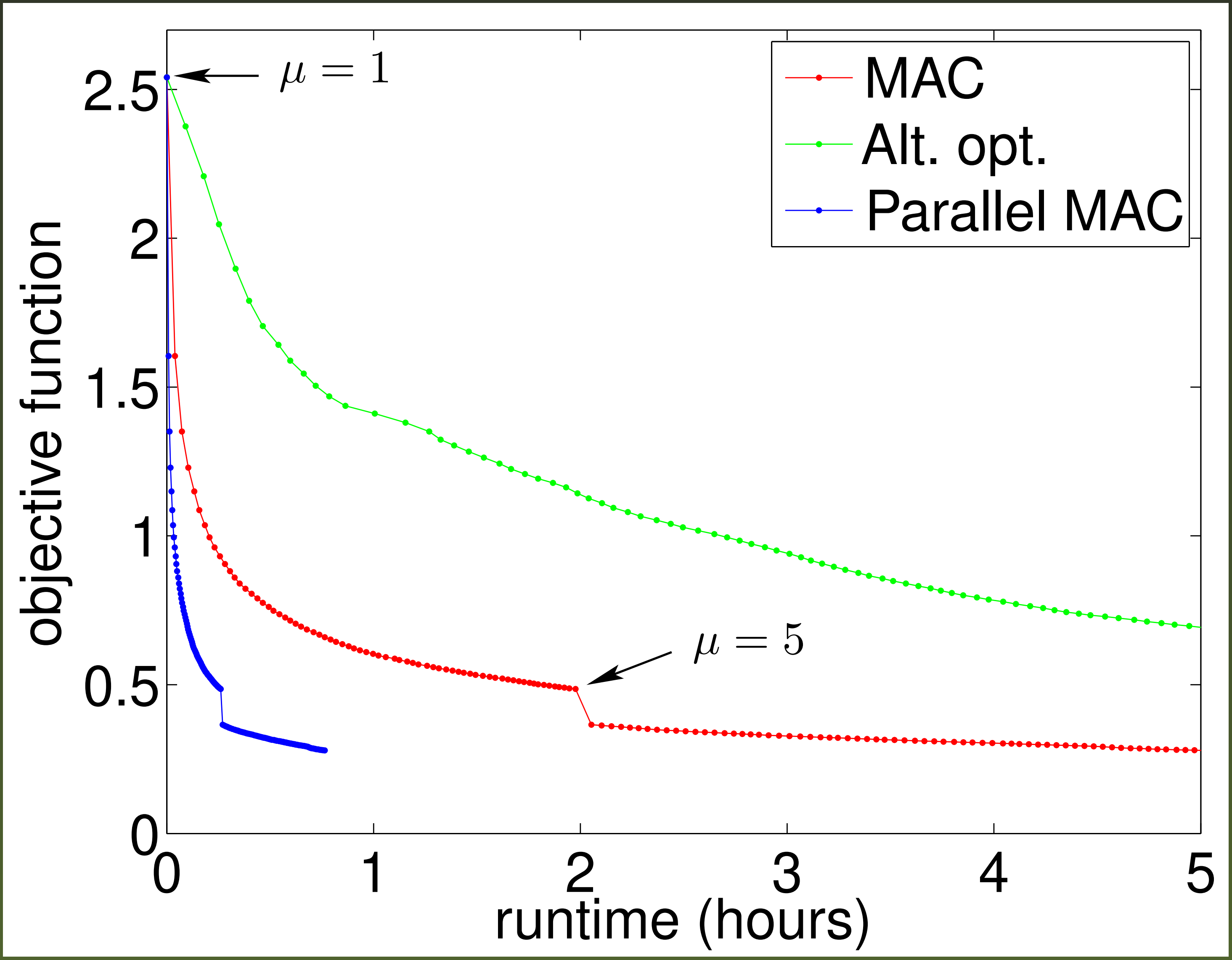
Note: the MAC iterations can be done much faster (see later):

- With better optimisation
- With parallel processing

Experiment: RBF autoencoder

COIL object images, 1024–1368–2–1368–1024 autoencoder ($K = 3$ hidden layers), auxiliary coordinates in bottleneck layer only, initial $\mathbf{Z}$. $\mathbf{W}$ step uses k-means ($\mathbf{C}_k$) + linsys ($\mathbf{W}_k$). $\mathbf{Z}$ step uses Gauss-Newton.





Practicalities

Schedule of the penalty parameter μ:

- Theory: $\mu \to \infty$ for convergence.
- Practice: stop with finite μ.
- Keeping $\mu = 1$ gives quite good results.
- How fast to increase μ depends on the problem.
  An exponential schedule is quite convenient: $\mu_i = \mu_0 a^i$ for $a > 1$ and $i = 0, 1, 2 \dots$
- We increase μ when the error in a validation set increases.

The postprocessing step:
After the algorithm stops, we satisfy the constraints by:

- Setting $\mathbf{z}_{kn} = \mathbf{f}_k(\mathbf{z}_{k-1,n}; \mathbf{W}_k)$, $k = 1, \dots, K$, $n = 1, \dots, N$
  That is, project on the feasible set by forward propagation.
- Keeping all the weights the same except for the last layer, where we set $\mathbf{W}_{K+1}$ by fitting $\mathbf{f}_{K+1}$ to the dataset $(\mathbf{f}_K(\dots(\mathbf{f}_1(\mathbf{X}))), \mathbf{Y})$.
  This provably reduces the error.

Practicalities (cont.)

Choice of optimisation algorithm for the steps:

- $\mathbf{W}$ step: typically, reuse existing single-layer algorithm
  Linear: linsys; SVM: QP; RBF net: k-means + linsys; etc.
  Large datasets: use stochastic updates w/ data minibatches (SGD).
- $\mathbf{Z}$ step: sometimes closed-form, otherwise:
  - Small number of parameters in $\mathbf{z}_n$: Gauss-Newton
    The Gauss-Newton matrix is always positive definite because of the $\|\mathbf{z} - \cdot\|^2$ terms.
  - Large number of parameters in $\mathbf{z}_n$: CG, Newton-CG, L-BFGS…
  - "Even-odd" alternating optimisation if multiple layers.

Standard optimisation and linear algebra techniques apply:

- Inexact steps (solve $\mathbf{W}$ or $\mathbf{Z}$ steps approximately).
- Warm start (initialise from previous iteration's result).
- Caching factorisations (and reuse them over iterations).

Cleverly used, they can make the $\mathbf{W}$ and $\mathbf{Z}$ steps very fast.

Practicalities (cont.)

Defining the auxiliary coordinates:

- With neural nets, we can introduce them
  - after the nonlinearity
  - before the nonlinearity
  - before and after the nonlinearity
- No need to introduce auxiliary coordinates at each layer.

  Spectrum between fully nested (no auxiliary coordinates, pure backpropagation) and fully unnested (auxiliary coordinates at each layer, no chain rule).
- Can even redefine $\mathbf{Z}$ over the optimisation.

The best strategy will depend on the dataset dimensionality and size, and on the model.

Related work: dimensionality reduction

- Given high-dim data $\mathbf{y}_1, \dots, \mathbf{y}_N \in \mathbb{R}^D$, we want to project to latent coordinates $\mathbf{z}_1, \dots, \mathbf{z}_N \in \mathbb{R}^L$ with $L \ll D$.
- Optimise reconstruction error over the reconstruction mapping $\mathbf{f}\colon \mathbf{z} \rightarrow \mathbf{y}$ and the latent coordinates $\mathbf{Z}$:

$$\min_{\mathbf{f},\mathbf{Z}} \sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{f}(\mathbf{z}_n) \right\|^2$$

where $\mathbf{f}$ can be linear (least-squares factor analysis; Young 1941, Whittle 1952...) or nonlinear: spline (Leblanc & Tibshirani 1994), single-layer neural net (Tan & Mavrovouniotis 1995), RBF net (Smola et al. 2001), kernel regression (Meinicke et al. 2005), Gaussian process (GPLVM; Lawrence 2005), etc.

- Problem: nearby $\mathbf{z}$s map to nearby $\mathbf{y}$s, but not necessarily vice versa. This can produce a poor representation in latent space.
- This can be solved by introducing the “inverse” mapping $\mathbf{F}\colon \mathbf{y} \rightarrow \mathbf{z}$.

Related work: dimensionality reduction (cont.)

"Dimensionality reduction by unsupervised regression"
(Carreira-Perpiñán & Lu, 2008, 2010)

$$\min_{\mathbf{f},\mathbf{F},\mathbf{Z}} \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{f}(\mathbf{z}_n)\right\|^2 + \left\|\mathbf{z}_n - \mathbf{F}(\mathbf{y}_n)\right\|^2$$

- Learns both mappings: reconstruction $\mathbf{f}$ and projection $\mathbf{F}$, together with the latent coordinates $\mathbf{Z}$.
- Now nearby $\mathbf{y}$'s also map to nearby $\mathbf{x}$'s

  $\mathbf{f}$ and $\mathbf{F}$ become approximate inverses of each other on the data manifold.
- Special case of MAC to solve the autoencoder problem:

$$\min_{\mathbf{f},\mathbf{F}} \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{f}(\mathbf{F}(\mathbf{x}_n))\right\|^2$$

but with $\mu = 1$ (so biased solution).

Related work: learning good internal representations

Updating weights and hidden unit activations in neural nets:

- Idea originates in 1980s, focused on (single-layer) neural nets.

  Grossman et al. 1988, Saad & Marom 1990, Krogh et al. 1990, Rohwer 1990, Olshausen & Field 1996, Ma et al. 1997, Castillo et al. 2006, Ranzato et al. 2007, Kavukcuoglu et al. 2008, Baldi & Sadowski 2012, etc.
- Learning good internal representations was seen as important as learning good weights.
- Desirable activation values were explicitly generated in different ways: ad-hoc objective function (e.g. to make them sparse), sampling, etc.
- The weights and activations were updated in alternating fashion.
- The generation of activation values wasn't directly related to the nested objective function, so the algorithm didn't converge to a minimum of the latter.

Related work: ADMM

Alternating direction method of multipliers (ADMM):

- Optimisation algorithm for constrained problems with separability.
- Alternates steps on the augmented Lagrangian over the primal and dual variables.
- Often used in consensus problems:

$$\min_{\mathbf{x}} \sum_{n=1}^{N} f_n(\mathbf{x}) \Leftrightarrow \min_{\mathbf{x}_1,\dots,\mathbf{x}_N,\mathbf{z}} \sum_{n=1}^{N} f_n(\mathbf{x}_n) \text{ s.t. } \mathbf{x}_n = \mathbf{z},\ n = 1,\dots,N \Leftrightarrow$$

$$\min \mathcal{L}(\mathbf{X},\mathbf{z},\boldsymbol{\Lambda}) = \sum_{n=1}^{N} \left(f_n(\mathbf{x}_n) + \boldsymbol{\lambda}_n^T(\mathbf{x}_n - \mathbf{z}) + \frac{\mu}{2}\|\mathbf{x}_n - \mathbf{z}\|^2 \right)$$

  The aug. Lag. $\mathcal{L}$ is minimised alternatingly over $\mathbf{X}$, $\mathbf{z}$ and $\boldsymbol{\Lambda}$.
- Can be applied to the MAC-constrained problem as well.

Related work: EM

Expectation-maximisation (EM) algorithm:

- Trains probability models by maximum likelihood.
- Can be seen as bound optimisation (majorisation) or alternating optimisation (of a specially constructed function).
- E step: update the posterior probabilities $\mathbf{p}_1, \dots, \mathbf{p}_N$, in parallel.
  Like MAC's $\mathbf{Z}$ step. The posterior probabilities "coordinate" the individual models.
- M step: update the model parameters, in parallel (the Gaussians for a GM).
  Like MAC's step over the parameters. Each model is trained on its own data and $\mathbf{p}_1, \dots, \mathbf{p}_N$.

EM and MAC have the following properties:

- The specific algorithm is very easy to develop in many cases; intuitive steps where simple models are fit.
- Convergence guarantees (under some assumptions).
- Large initial steps, eventually slower convergence.
- Parallelism.

Related work: automatic differentiation

- Evaluates numerically the gradient (or some other derivative) of a function at a desired point, by applying the chain rule recursively.
  Assuming the function is differentiable.
- Saves us from computing the gradient analytically and coding it.
- Less efficient than a hand-crafted implementation of a particular gradient (expression simplification, etc.).
- It is not a numerical optimisation algorithm.
  We still need to use that gradient with an algorithm (gradient descent, Levenberg-Marquardt. . .), do a line search, select parameters (momentum rate, damping factor. . .), have a stopping criterion, etc.
- In MAC we don't use the gradient of the objective function because we split it into subproblems. Each of these is often a well-known optimisation problem for which efficient code (possibly optimised for our architecture) exists. MAC calls this as a black box by passing to it the data and auxiliary coordinates and receiving the solution.
  We need not compute/code the gradient, do l.s., select parameters, etc.

Model selection "on the fly"

M. Carreira-Perpiñán and W. Wang: *Distributed optimization of deeply nested systems*, AISTATS 2014, arXiv:1212.5921.

We can also optimise over (discrete) hyperparameters of the model (such as the number of hidden units or basis functions) while monotonically decreasing the (penalty) objective function.

Idea: consider discrete variables $x_1, \dots, x_K$ taking M values each, and an objective function f of those variables. Then, the cost of optimising f (by brute-force or enumeration) is:

- if $f(x_1, \dots, x_K)$ is not separable: M^K evaluations of the entire f
- if $f(x_1, \dots, x_K) = f_1(x_1) + \dots + f_K(x_K)$ is separable: M evaluations of each f_k so MK evaluations of one f_k

We can apply this within MAC by having an additional step over the hyperparameters if the penalty objective function separates over the layers corresponding to the hyperparameters.

Model selection "on the fly" (cont.)

- Model selection criteria (AIC, BIC, MDL, etc.) separate over layers:

$$\overline{E}(\mathbf{W}) = E_{\text{nested}}(\mathbf{W}) + C(\mathbf{W}) = \text{nested-error} + \text{model-cost}$$
$$C(\mathbf{W}) \propto \text{total \# parameters} = |\mathbf{W}_1| + \cdots + |\mathbf{W}_K|$$

- Traditionally, a grid search (with M values per layer) means testing an exponential number M^K of nested models.
- In MAC, the cost $C(\mathbf{W})$ separates over layers in the $\mathbf{W}$ step, so each layer can do model selection independently of the others, testing a polynomial number MK of shallow models.

  This still minimises the overall objective $\overline{E}(\mathbf{W})$.
- Instead of a criterion, we can do cross-validation in each layer.

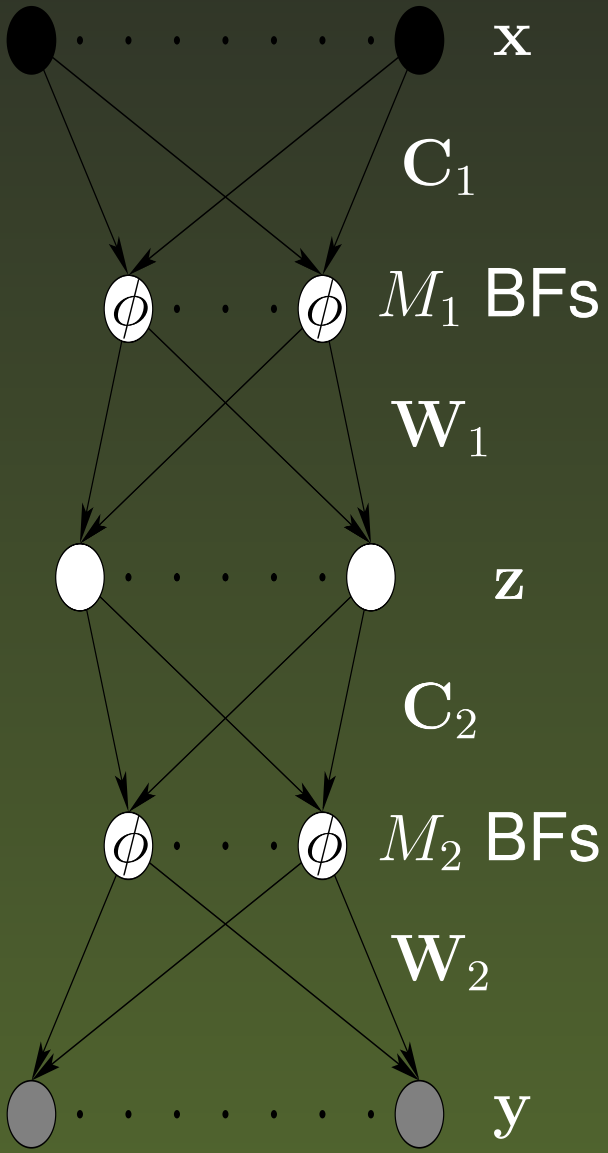
  Partition the data and auxiliary coordinates into training and test.
- In practice, no need to do model selection at each $\mathbf{W}$ step.

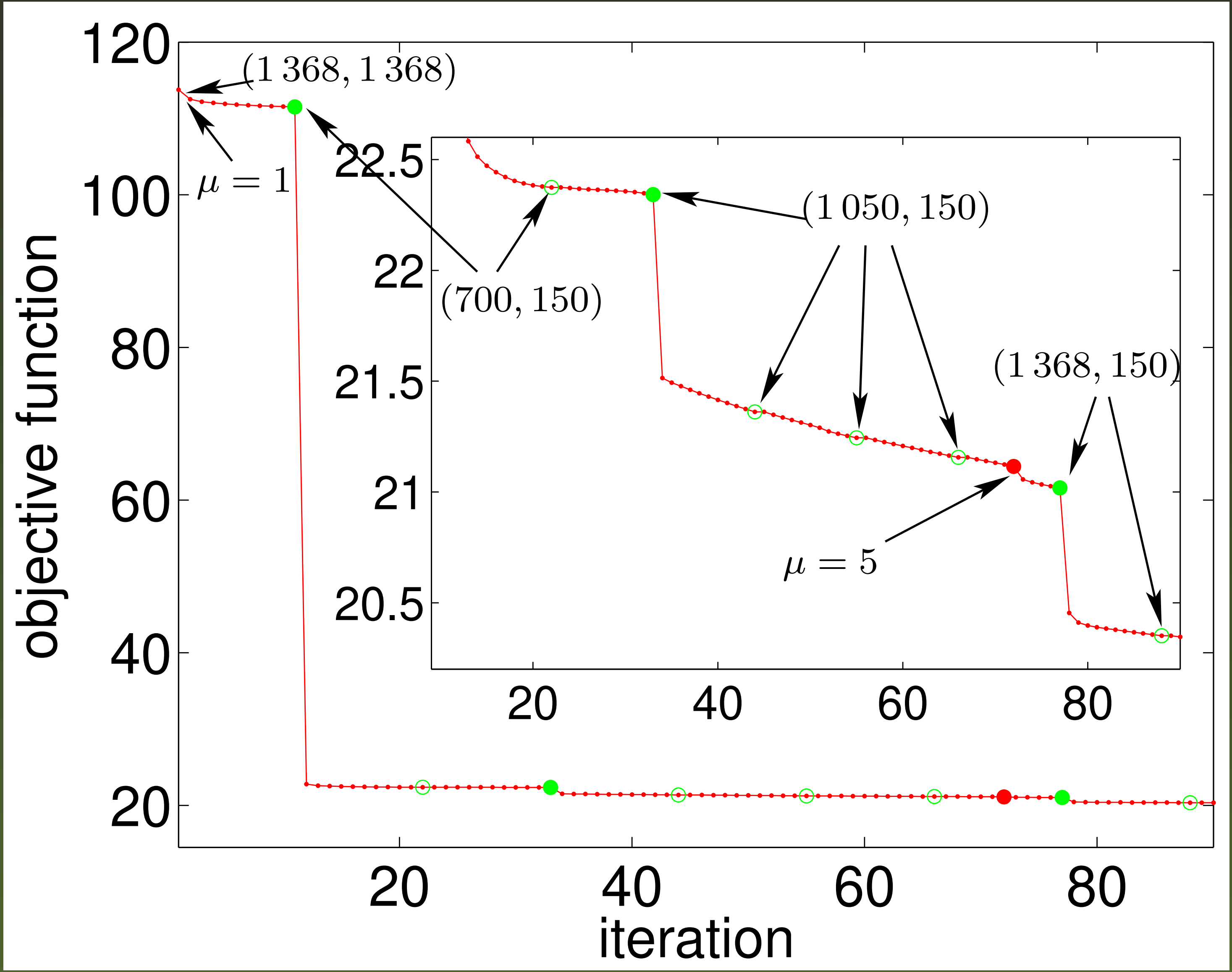
  The algorithm usually settles in a region of good architectures early during the optimisation, with small and infrequent changes thereafter.

This way, MAC searches over the parameter space of the architecture and over the space of architectures itself, in polynomial time, iteratively.

Experiment (model selection "on the fly"): RBF autoenc

COIL object images, 1024–M_1–2–M_2–1024 autoencoder ($K = 3$ hidden layers), AIC model selection over (M_1, M_2) in $\{150, \dots, 1368\}$ (50 values $\Rightarrow 50^2$ possible models).





Parallel/distributed optimisation with MAC

MAC is embarrassingly parallel:

- $\mathbf{W}$ step:
  - all layers separate ($K+1$ independent subproblems)
  - often, all units within each layer separate $\Rightarrow$
    one independent subproblem for each unit's input weight vector
  - the model selection steps also separate
    test each model independently
- $\mathbf{Z}$ step: all points separate (N independent subproblems).

Enormous potential for parallel implementation:

- Unlike other machine learning or optimisation algorithms, where subproblems are not independent (e.g. SGD).
- Suitable for large-scale data and distributed clusters.

Parallel/distributed optimisation with MAC: Matlab

- Shared-memory multiprocessor model using the Matlab Parallel Processing Toolbox: simply change `for` to `parfor` in the $\mathbf{W}$ and $\mathbf{Z}$ loops.

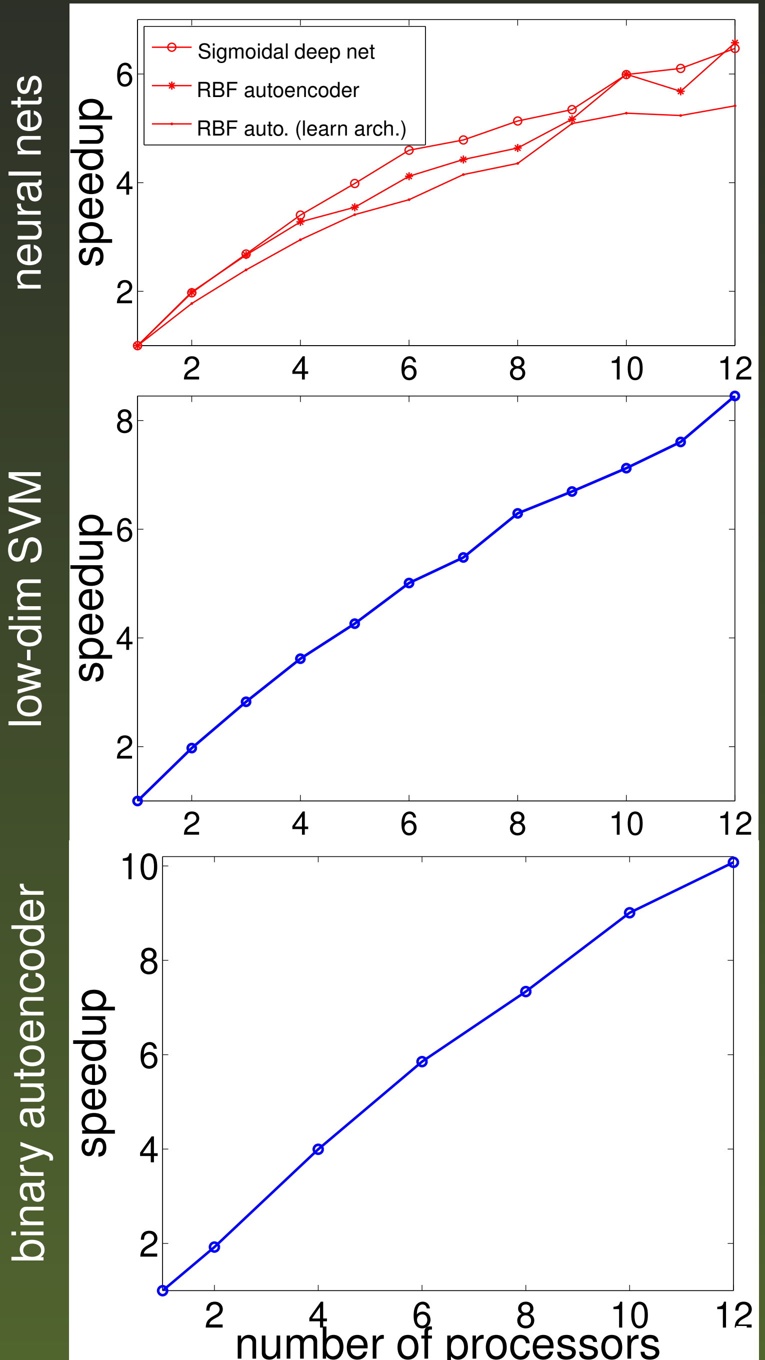
  So Matlab sends each iteration to a different server.
  Our license is limited to 12 processes.

- Near-linear speedups as a function of the number of processors in different nested models.

  Even though the Matlab Parallel Processing Toolbox is quite inefficient.

- We can carry this much farther with ParMAC, a distributed computation model for MAC, implemented in MPI.

  See later.



Outline

- Nested (deep) and shallow systems
- Training nested systems: chain-rule gradient; greedy layerwise
- The method of auxiliary coordinates (MAC)
  - Design pattern
  - Convergence guarantees
  - Practicalities
  - Related work
  - Model selection "on the fly"
- ☞ A gallery of nested models trainable with MAC
  1. Learning low-dimensional features for classification: low-dim SVM
  2. Parametric nonlinear embeddings: parametric t-SNE, EE
  3. Binary hashing for fast image search: binary autoencoder
  4. Learning radial basis function (RBF) networks
  5. Learning feature transformations for decision trees: neural net + tree
  6. Best-subset feature selection
- Distributed optimisation with MAC: ParMAC

A gallery of nested models trainable with MAC

MAC is very widely applicable. Here are a few examples:

1. Learning low-dimensional features for classification:
   - Low-dimensional SVM
   - Low-dimensional logistic regression
2. Parametric nonlinear embeddings:
   - t-SNE/elastic embedding with linear/neural net mapping
3. Binary hashing for fast image search:
   - Binary autoencoder
   - Affinity-based objective function
4. Learning radial basis function (RBF) networks
5. Learning feature transformations for decision trees:
   - Rotation / linear transformation / neural net + tree
6. Best-subset feature selection.

A gallery of nested models trainable with MAC (cont.)

These demonstrate MAC in various settings:

- various types of regressor or classifier
- various optimisation algorithms for each step
- various penalty functions: quadratic penalty, augmented Lagrangian
- differentiable and nondifferentiable (or even discrete) functions
- auxiliary coordinates in objective or constraints
- $\mathbf{Z}$ step separable or coupled on N auxiliary coordinates

1. Learning low-dimensional features for classification

We want to learn a low-dimensional classifier $y = g(\mathbf{F}(\mathbf{x}))$:

- first we reduce the dimensionality of the input $\mathbf{x}$ with a mapping $\mathbf{F}$
  linear, RBF network, neural net, Gaussian process, etc.
- Then we classify the low-dimensional projection with a classifier g
  linear SVM, logistic regression.

We introduce as auxiliary coordinates $\mathbf{z}_n$ the projection of each $\mathbf{x}_n$.
The resulting MAC algorithm alternates:

- train a regressor $\mathbf{F}$ on $(\mathbf{X}, \mathbf{Z})$ and a classifier g on $(\mathbf{Z}, \mathbf{Y})$, in parallel
  done with existing algorithms
- update the N auxiliary coordinates $\mathbf{Z}$, in parallel

What if we want to use a different $\mathbf{F}$ or g?

- $\mathbf{F}$: simply call a different algorithm in $\mathbf{F}$ step
  black box: no need to compute gradients or code anything
- g: call a different algorithm in g step, but the $\mathbf{Z}$ step does change.

1a. Low-dimensional SVM

W. Wang and M. Carreira-Perpiñán: *The role of dimensionality reduction in classification*, AAAI 2014.

Binary classifier g: linear SVM $g(\mathbf{z}) = \mathbf{w}^T\mathbf{z} + b$.

Objective function over the parameters of g and $\mathbf{F}$ given a dataset $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$ of (inputs,labels):

$$\begin{aligned} \min_{\mathbf{F},\mathbf{g},\boldsymbol{\xi}} \quad & \lambda R(\mathbf{F}) + \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^N \xi_n \\ \text{s.t.} \quad & y_n(\mathbf{w}^T\mathbf{F}(\mathbf{x}_n) + b) \geq 1 - \xi_n,\ \xi_n \geq 0,\ n = 1,\dots,N. \end{aligned}$$

- If $\mathbf{F} =$ identity then $y = g(\mathbf{F}(\mathbf{x})) = \mathbf{w}^T\mathbf{x} + b$ is a linear SVM and the problem is a convex quadratic program, easy to solve.
- If $\mathbf{F}$ is nonlinear then $y = g(\mathbf{F}(\mathbf{x})) = \mathbf{w}^T\mathbf{F}(\mathbf{x}) + b$ is a nonlinear classifier and the problem is nonconvex, difficult to solve.

1a. Low-dimensional SVM: auxiliary coordinates

The problem is “nested” because of $g(\mathbf{F}(\cdot))$ in the constraints:

$$\min_{\mathbf{F},\mathbf{g},\boldsymbol{\xi}} \quad \lambda R(\mathbf{F}) + \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N}\xi_n$$

$$\text{s.t.} \quad y_n(\underbrace{\mathbf{w}^T\mathbf{F}(\mathbf{x}_n) + b}_{g(\mathbf{F}(\mathbf{x}_n))}) \geq 1 - \xi_n,\ \xi_n \geq 0,\ n = 1,\dots,N.$$

❶ Break the nesting by introducing an auxiliary coordinate $\mathbf{z}_n$ per point:

$$\min_{\mathbf{F},\mathbf{g},\boldsymbol{\xi},\mathbf{Z}} \quad \lambda R(\mathbf{F}) + \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N}\xi_n$$

$$\text{s.t.} \quad y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1 - \xi_n,\ \xi_n \geq 0,\ \mathbf{z}_n = \mathbf{F}(\mathbf{x}_n),\ n = 1,\dots,N.$$

The auxiliary coordinates correspond to the low-dimensional projections but are separate parameters.

1a. Low-dimensional SVM: alternating optimisation

❷ Solve this constrained problem with a quadratic-penalty method: optimise for fixed penalty parameter $\mu > 0$ and drive $\mu \to \infty$:

$$\min_{\mathbf{F},\mathbf{g},\boldsymbol{\xi},\mathbf{Z}} \quad \lambda R(\mathbf{F}) + \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N}\xi_n + \frac{\mu}{2}\sum_{n=1}^{N}\|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2$$

$$\text{s.t.} \quad y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1 - \xi_n,\ \xi_n \geq 0,\ n = 1,\dots,N.$$

❸ Alternating optimisation of the penalty function for fixed μ:

- g step: fit a linear SVM to $\{(\mathbf{z}_n, y_n)\}_{n=1}^N$
  A classifier using as inputs the auxiliary coordinates.
  Use any desired SVM training algorithm: primal/dual, coordinate descent. . .
- $\mathbf{F}$ step: fit a nonlinear mapping $\mathbf{F}$ to $\{(\mathbf{x}_n, \mathbf{z}_n)\}_{n=1}^N$
  A regressor using as outputs the auxiliary coordinates.
  Use any convenient algorithm to train $\mathbf{F}$.

} in parallel

- $\mathbf{Z}$ step: set $\mathbf{z}_n = \mathbf{F}(\mathbf{x}_n) + \gamma_n y_n \mathbf{w}$, $n = 1,\dots,N$
  A closed-form update for each point's auxiliary coordinate $\mathbf{z}_n$.

} in parallel

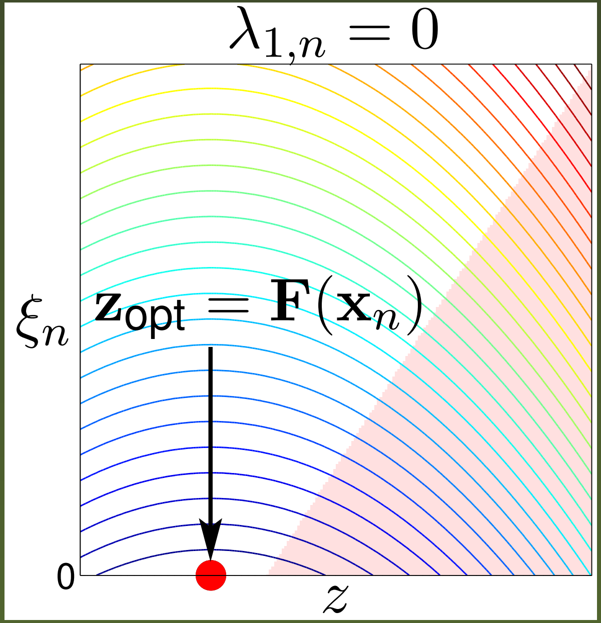
1a. Low-dimensional SVM: solution of the Z step

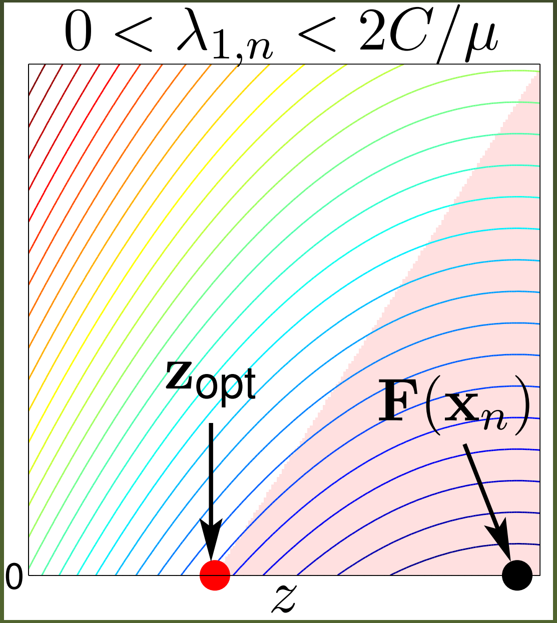
The $\mathbf{Z}$ step decouples into N independent problems, each a quadratic program on L parameters ($L =$ dimension of latent space):

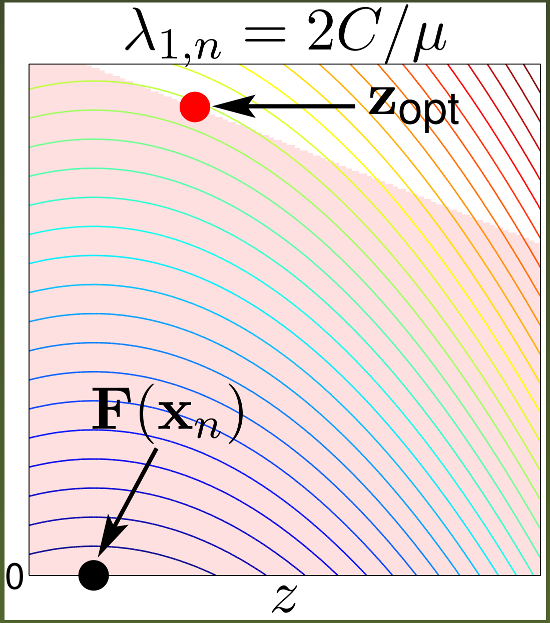
$$\min_{\mathbf{z}_n \in \mathbb{R}^L, \xi_n \in \mathbb{R}} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 + \frac{2C}{\mu}\xi_n \quad \text{s.t.} \quad y_n(\mathbf{w}^T\mathbf{z}_n + b) \geq 1 - \xi_n,\ \xi_n \geq 0$$

with closed-form solution $\mathbf{z}_n = \mathbf{F}(\mathbf{x}_n) + \gamma_n y_n \mathbf{w}$, where $\gamma_n = \frac{1}{2}\min(\lambda_{1,n}, 2C/\mu)$ and $\lambda_{1,n} = \frac{2}{\|\mathbf{w}\|^2}\max(0, 1 - y_n(\mathbf{w}^T\mathbf{F}(\mathbf{x}_n) + b))$.

This corresponds to three possible cases. Cost: $\mathcal{O}(L)$.







1a. Low-dimensional SVM: multiclass case

With K classes, we can use the one-versus-all scheme and have K binary SVMs, each of which classifies whether a point belongs to one class or not.

- Objective function: the sum of that of K binary SVMs.
- $\mathbf{F}$ step: a regressor as before.
- g step: a classifier as before, but with K binary SVMs trained in parallel (one per class).
- $\mathbf{Z}$ step: a QP for each point on $L+K$ variables and $2K$ constraints:

$$\min_{\mathbf{z}_n,\{\xi_n^k\}_{k=1}^K} \quad \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 + \sum_{k=1}^{K} C^k \xi_n^k$$
$$\text{s.t.} \quad y_n^k((\mathbf{w}^k)^T \mathbf{z}_n + b^k) \geq 1 - \xi_n^k,\ \xi_n^k \geq 0, \quad k = 1, \dots, K.$$

This has no closed-form solution, so we solve it numerically.

1a. Low-dimensional SVM: experiment

Reducing dimension with a RBF network $\mathbf{F}(\mathbf{x}) = \sum_{m=1}^{M} \boldsymbol{\alpha}_m \phi_m(\mathbf{x})$ the classifier has the form of a nonlinear SVM $y = \sum_{m=1}^{M} \mathbf{v}_m \phi_m(\mathbf{x}) + b$.

- We have direct control on the classifier complexity through the # BFs M. Unlike in an SVM, where the number of support vectors M is often very large.
- We can trade off classification error vs runtime:
  - Training: linear on sample size N and # BFs M.
  - Testing: linear on # BFs M.

  Error comparable to an SVM's, but with far fewer basis functions.
- The resulting classifier is competitive with nonlinear SVMs but faster to train and at test time.
- The classification accuracy improves drastically as L increases from 1 and stabilizes at $L \approx K - 1$, by which time the training samples are perfectly separated and the classes form point-like clusters approximately lying on the vertices of a simplex.
- MAC algorithm: large progress in first few iterations.

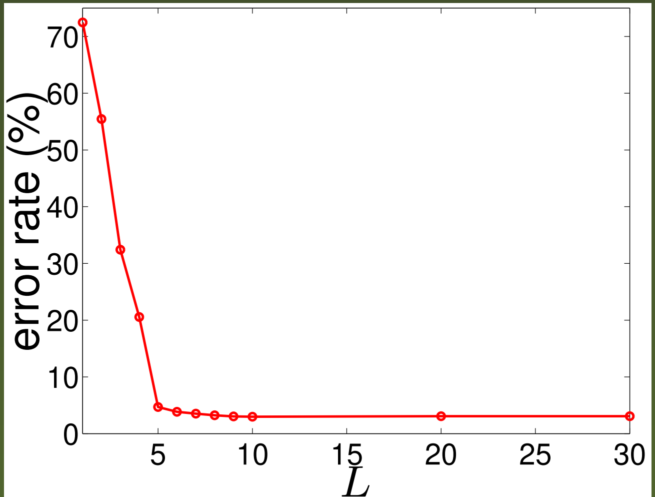
1a. Low-dimensional SVM: experiment (cont.)

10K MNIST handwritten digit images of $28 \times 28 = 784$ dimensions with $K = 10$ classes (one-versus-all).

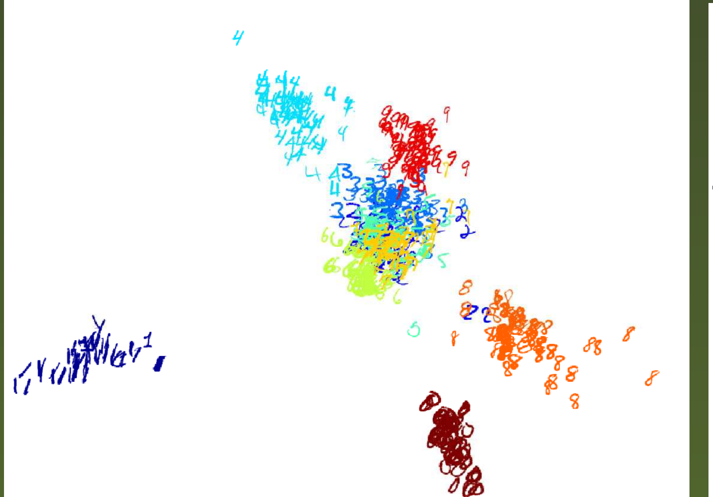
A low-dim SVM ($L = K$ dimensions) does as well as a kernel SVM but using 5.5 times fewer support vectors. The low-dim SVM is also faster to train and parallelizes trivially.

Method	Test error	# BFs
Nearest neighbour	5.34	10 000
Linear SVM	9.20	—
Gaussian SVM	2.93	13 827
low-dim SVM	2.99	2 500
LDA (9) + Gaussian SVM	10.67	8 740
PCA (5) + Gaussian SVM	24.31	13 638
PCA (10) + Gaussian SVM	7.44	5 894
PCA (40) + Gaussian SVM	2.58	12 549
PCA (40) + low-dim SVM	2.60	2 500

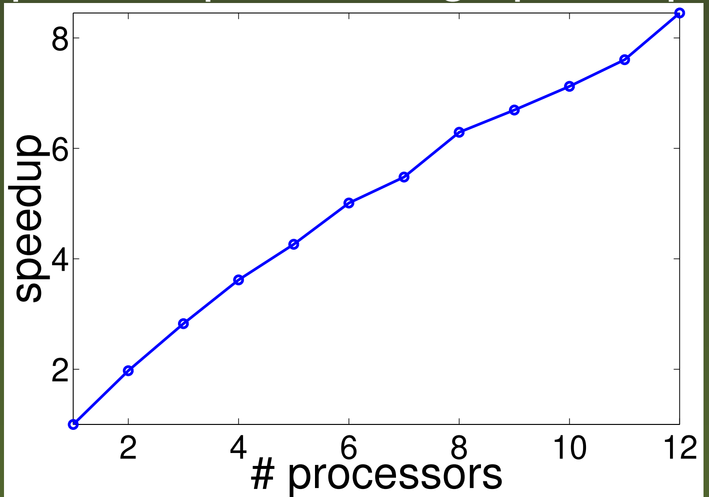
classification error wrt L



L-dim space (+ 2D PCA)



parallel processing speedup



1b. Low-dimensional logistic regression (unpublished results)

Binary classifier g: logistic regression $g(\mathbf{z}) = \sigma(\mathbf{w}^T\mathbf{z}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{z}}}$.

Original (nested) objective function: cross-entropy + regularisation

$$\min_{\mathbf{F},\mathbf{w}} -\sum_{n=1}^{N} \big(y_n \log(\sigma(\mathbf{w}^T\mathbf{F}(\mathbf{x}_n))) + (1-y_n)\log(\sigma(-\mathbf{w}^T\mathbf{F}(\mathbf{x}_n)))\big) + \lambda_{\mathbf{F}} R(\mathbf{F}) + \lambda_{\mathbf{g}} R(\mathbf{w})$$

❶ MAC-constrained problem:

$$\begin{aligned}\min_{\mathbf{Z},\mathbf{F},\mathbf{w}} \quad & -\sum_{n=1}^{N} \big(y_n \log(\sigma(\mathbf{w}^T\mathbf{z}_n)) + (1-y_n)\log(\sigma(-\mathbf{w}^T\mathbf{z}_n))\big) + \lambda_{\mathbf{F}} R(\mathbf{F}) + \lambda_{\mathbf{g}} R(\mathbf{w}) \\ \text{s.t.} \quad & \mathbf{z}_n = \mathbf{F}(\mathbf{x}_n),\ n = 1,\dots,N.\end{aligned}$$

❷ MAC-penalised objective (using the augmented Lagrangian, with Lagrange multipliers $\boldsymbol{\alpha}_n$):

$$\begin{aligned}\min_{\mathbf{Z},\mathbf{F},\mathbf{w}} \quad & -\sum_{n=1}^{N} \big(y_n \log(\sigma(\mathbf{w}^T\mathbf{z}_n)) + (1-y_n)\log(\sigma(-\mathbf{w}^T\mathbf{z}_n))\big) + \lambda_{\mathbf{F}} R(\mathbf{F}) + \lambda_{\mathbf{g}} R(\mathbf{w}) \\ & -\sum_{n=1}^{N} \boldsymbol{\alpha}_n^T(\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)) + \frac{\mu}{2}\sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2\end{aligned}$$

1b. Low-dimensional logistic regression ($^{\text{unpublished}}_{\text{results}}$) (cont.

❸ Alternating optimisation over $(\mathbf{g}, \mathbf{F}, \mathbf{Z})$:

$\left.\begin{array}{l}\text{❖ } \mathbf{g} \text{ step: fit a logistic regression to } (\mathbf{Z}, \mathbf{Y}). \\ \text{❖ } \mathbf{F} \text{ step: fit a nonlinear mapping to } (\mathbf{X}, \mathbf{Z}).\end{array}\right\}$ in parallel

$\left.\begin{array}{l}\text{❖ } \mathbf{Z} \text{ step: solve for each point } n = 1, \dots, N \\ \quad\text{an } L\text{-dim smooth convex problem:}\end{array}\right\}$ in parallel

$$\min_{\mathbf{z}_n \in \mathbb{R}^L} \; -y_n \log(\sigma(\mathbf{w}^T \mathbf{z}_n)) - (1 - y_n) \log(\sigma(-\mathbf{w}^T \mathbf{z}_n)) + \frac{\mu}{2} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 - \boldsymbol{\alpha}_n^T (\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n))$$

✦ Solution: a closed-form update $\mathbf{z}_n = \mathbf{F}(\mathbf{x}_n) + \frac{1}{\mu}\boldsymbol{\alpha}_n - \frac{t}{\mu}\mathbf{w}$

t is the single solution of the scalar equation $t = 1 - y_n - \sigma\left(\frac{\|\mathbf{w}\|^2}{\mu} t - \mathbf{w}^T \left(\mathbf{F}(\mathbf{x}_n) + \frac{1}{\mu}\boldsymbol{\alpha}_n\right)\right)$.

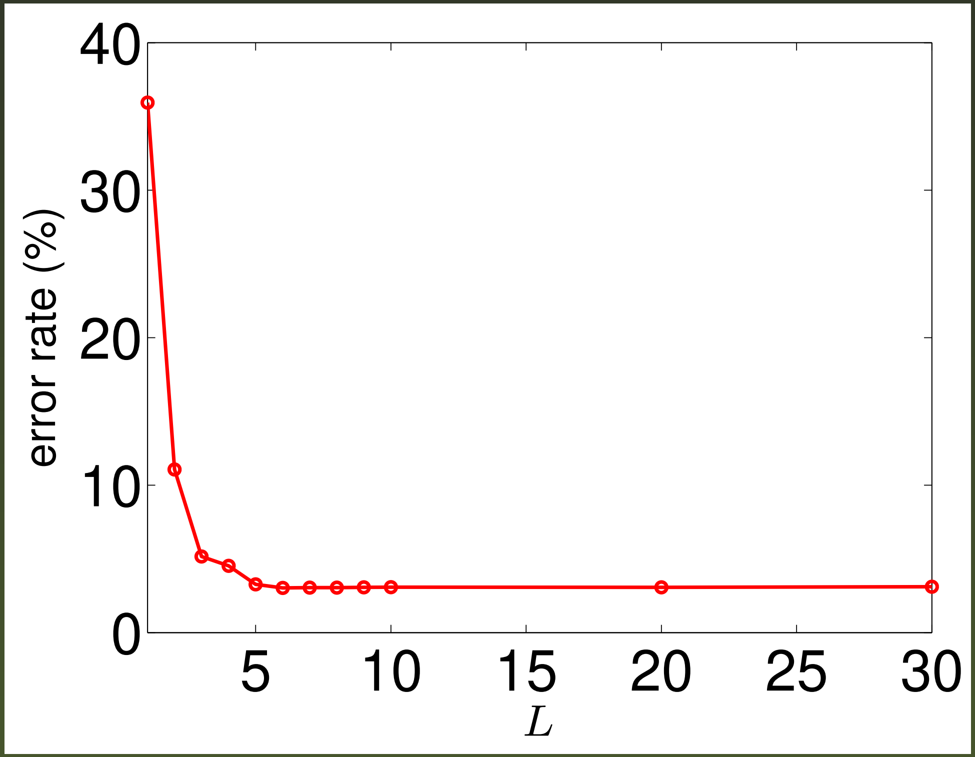
✦ Also convex for multi-class case with softmax

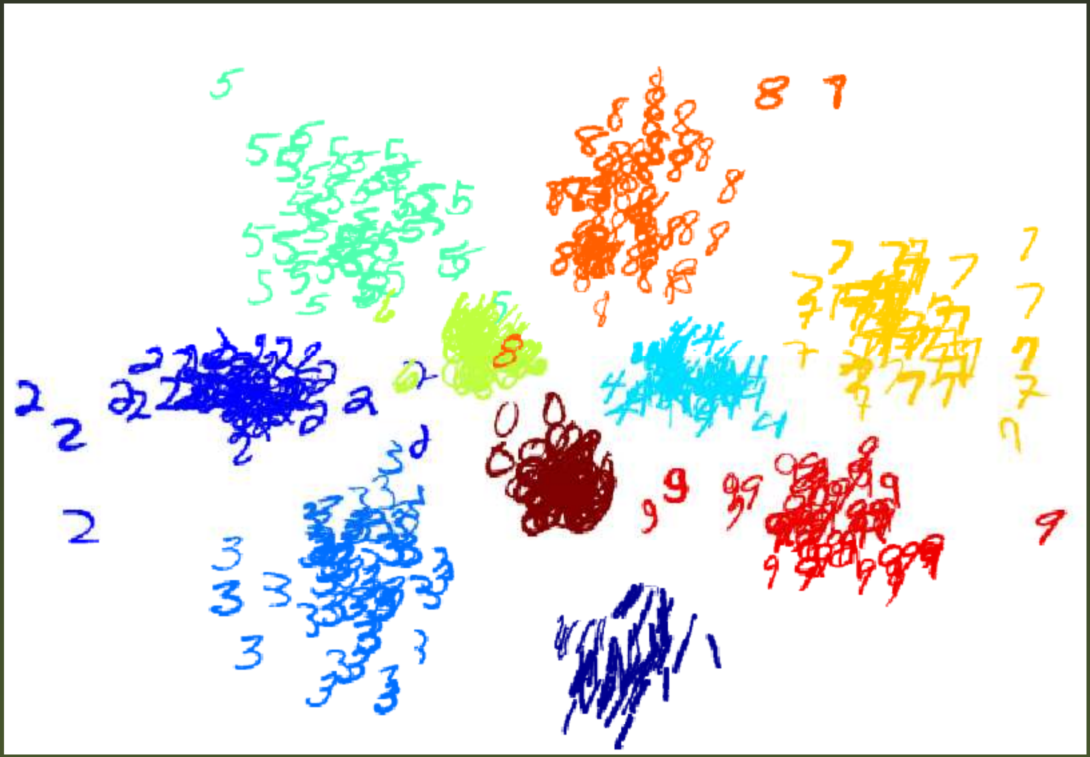
Then, update the Lagrange multipliers:

$$\boldsymbol{\alpha}_n \leftarrow \boldsymbol{\alpha}_n - \mu(\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)), \; n = 1, \dots, N$$

1b. Low-dimensional logistic regression: experiment

Low-dimensional logistic regression ($\mathbf{F}$ = RBF network with $2\,500$ BFs) of MNIST digits ($K = 10$ classes)





classification error over L | embedding $\mathbf{F}(\mathbf{X})$ for $L = 2$

The results are similar to those of the low-dimensional SVM.

2. Parametric nonlinear embeddings

M. Carreira-Perpiñán and M. Vladymyrov: *A fast, universal algorithm to learn parametric nonlinear embeddings*, NIPS 2015.

Nonlinear embedding, e.g. t-SNE or the elastic embedding: finds L-dimensional projections $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathbb{R}^L$ of N high-dimensional data points $\mathbf{y}_1, \dots, \mathbf{y}_N \in \mathbb{R}^D$ by preserving given similarities w_{nm} between pairs of points $(\mathbf{y}_n, \mathbf{y}_m)$:

$$E_{\text{EE}}(\mathbf{X}) = \sum_{n,m=1}^{N} w_{nm} \|\mathbf{x}_n - \mathbf{x}_m\|^2 + \lambda \sum_{n,m=1}^{N} \exp\left(-\|\mathbf{x}_n - \mathbf{x}_m\|^2\right)$$

$$E_{t\text{-SNE}}(\mathbf{X}) = \sum_{n,m=1}^{N} w_{nm} \log\left(1 + \|\mathbf{x}_n - \mathbf{x}_m\|^2\right) + \sum_{n,m=1}^{N} \left(1 + \|\mathbf{x}_n - \mathbf{x}_m\|^2\right)^{-1}$$

- Unsupervised dimensionality reduction problem; input data: $\{w_{nm}\}$.
- A generalisation of multidimensional scaling (MDS).
- Often used to visualise high-dimensional datasets in 2D.

2. Parametric nonlinear embeddings (cont.)

Optimising $E(\mathbf{X})$ is difficult:

- There are NL optimisation variables $\mathbf{x}_1, \dots, \mathbf{x}_N$

  one vector $\mathbf{x}_n \in \mathbb{R}^L$ per data point.
- It is nonlinear and nonconvex.
- It contains $\mathcal{O}(N^2)$ terms (one for every pair of points), so computing $E(\mathbf{X})$ and $\nabla E(\mathbf{X})$ is $\mathcal{O}(N^2)$.

Efficient algorithms proposed in the last few years:

- Iterations that are fast to compute and make a lot of progress using second-order information efficiently: the spectral direction.

  It beats gradient descent, conjugate gradients, fixed-point iteration, L-BFGS and other standard nonlinear optimisation algorithms (Vladymyrov & Carreira-Perpiñán, 2012).
- Gradient approximated using N-body methods, in $\mathcal{O}(N \log N)$ (Barnes-Hut tree) or $\mathcal{O}(N)$ (fast multipole methods).

  (van der Maaten, 2013; Yang et al. 2013; Vladymyrov & Carreira-Perpiñán, 2014).

These methods scale nonlinear embeddings to millions of points.

2. Parametric nonlinear embeddings (cont.)

Parametric embedding: out-of-sample mapping to project points not in the training set ($\mathbf{F}$ can be linear, a neural net, etc.). For EE:

$$P(\mathbf{F}) = \sum_{n,m=1}^{N} w_{nm} \left\| \mathbf{F}(\mathbf{y}_n) - \mathbf{F}(\mathbf{y}_m) \right\|^2 + \lambda \sum_{n,m=1}^{N} \exp\left(- \left\| \mathbf{F}(\mathbf{y}_n) - \mathbf{F}(\mathbf{y}_m) \right\|^2 \right)$$

Also called siamese networks.

The param. embedding objective is a nested function: $P(\mathbf{F}) = E(\mathbf{F}(\mathbf{Y}))$.

Nonlinear, nonconvex, $\mathcal{O}(N^2)$ terms, very slow gradient-based optimisation over the parameters of $\mathbf{F}$.

❶ MAC-constrained problem: introduce the latent projections as auxiliary coordinates:

$$\min_{\mathbf{F},\mathbf{Z}} \quad E(\mathbf{Z}) = \sum_{n,m=1}^{N} w_{nm} \left\| \mathbf{z}_n - \mathbf{z}_m \right\|^2 + \lambda \sum_{n,m=1}^{N} \exp\left(- \left\| \mathbf{z}_n - \mathbf{z}_m \right\|^2 \right)$$

$$\text{s.t.} \quad \mathbf{z}_n = \mathbf{F}(\mathbf{y}_n), \; n = 1, \dots, N.$$

❷ MAC-penalised objective: $\min_{\mathbf{F},\mathbf{Z}} E(\mathbf{Z}) + \frac{\mu}{2} \left\| \mathbf{Z} - \mathbf{F}(\mathbf{Y}) \right\|^2$.

2. Parametric nonlinear embeddings (cont.)

❸ Alternating optimisation over $(\mathbf{F}, \mathbf{Z})$:

- $\mathbf{F}$ step: fit a nonlinear mapping to $(\mathbf{Y}, \mathbf{Z})$
- $\mathbf{Z}$ step: regularised nonlinear embedding

$$\min_{\mathbf{F},\mathbf{Z}} \underbrace{E(\mathbf{Z})}_{\text{embedding } \mathbf{Z}} + \frac{\mu}{2} \underbrace{\|\mathbf{Z} - \mathbf{F}(\mathbf{Y})\|^2}_{\text{pulls } \mathbf{Z} \text{ towards } \mathbf{F}(\mathbf{X})}$$

  - ✦ it does not separate over $\mathbf{z}_1, \dots, \mathbf{z}_N$

    unlike in previous cases, where the objective was a sum of points rather than pairs of points
  - ✦ but we can still reuse the above algorithms to train the embedding efficiently as a black box.

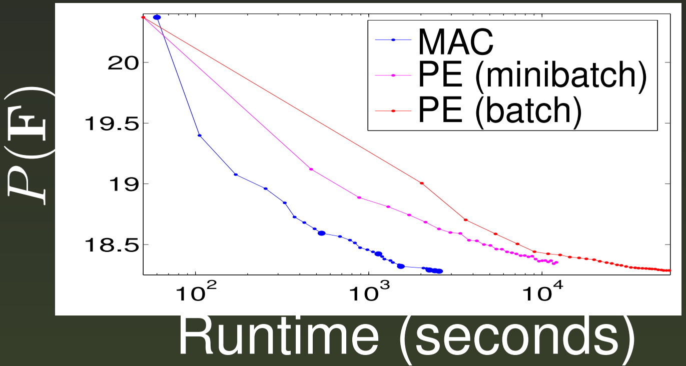
Advantages:

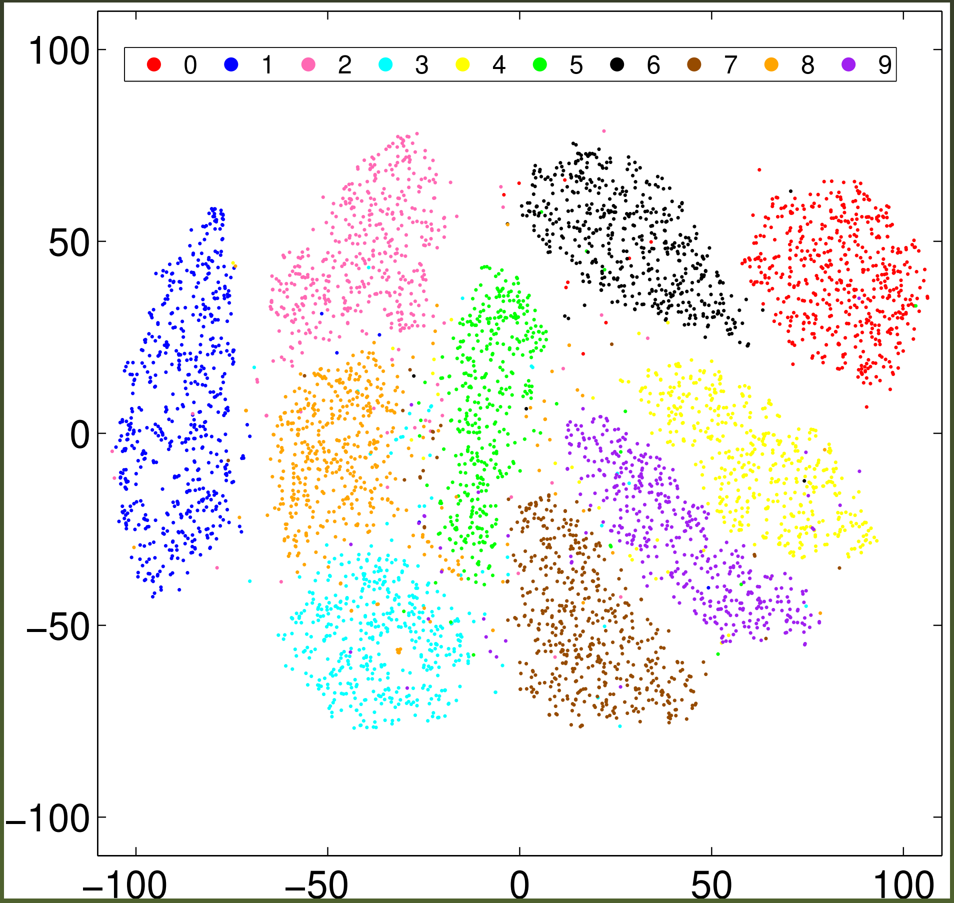
- Much faster than using the chain-rule gradient of P over $\mathbf{F}$

  which does not benefit from the N-body methods.
- If we want to use a different embedding objective $E(\mathbf{X})$ or projection mapping $\mathbf{F}$, we simply call a different embedding or regression routine in the $\mathbf{Z}$ or $\mathbf{F}$ step, resp., as a black box.

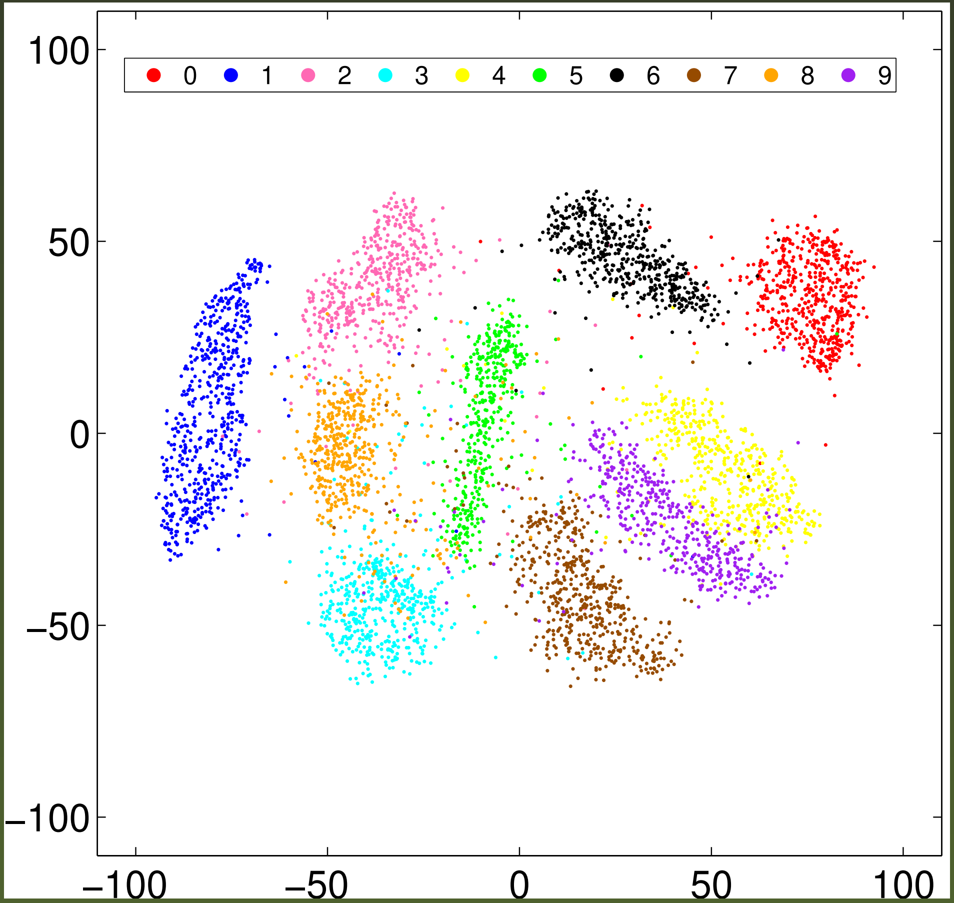
2. Parametric nonlinear embeddings: experiment

MNIST digits visualised with t-SNE using a neural net mapping ((28×28)–500–500–2000–2, initialized with pretraining).





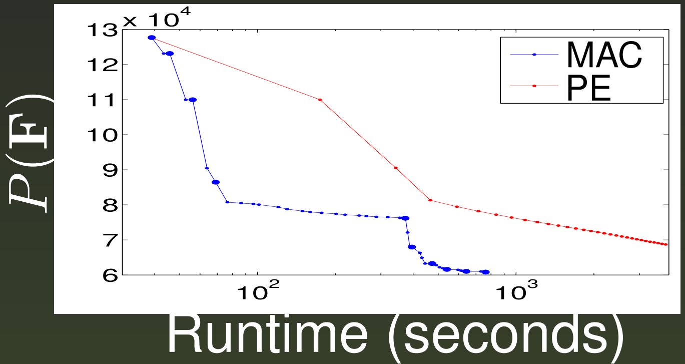
initial, free embedding $E(\mathbf{X})$

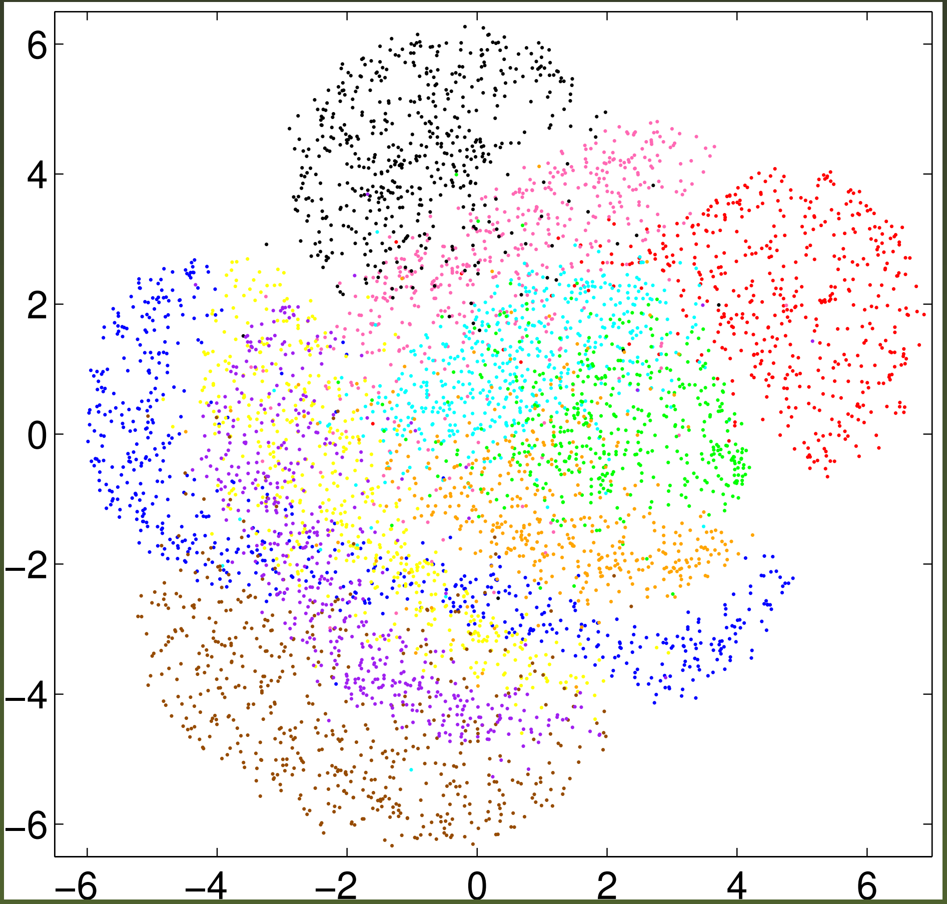


Final parametric embedding $\mathbf{F}(Y)$

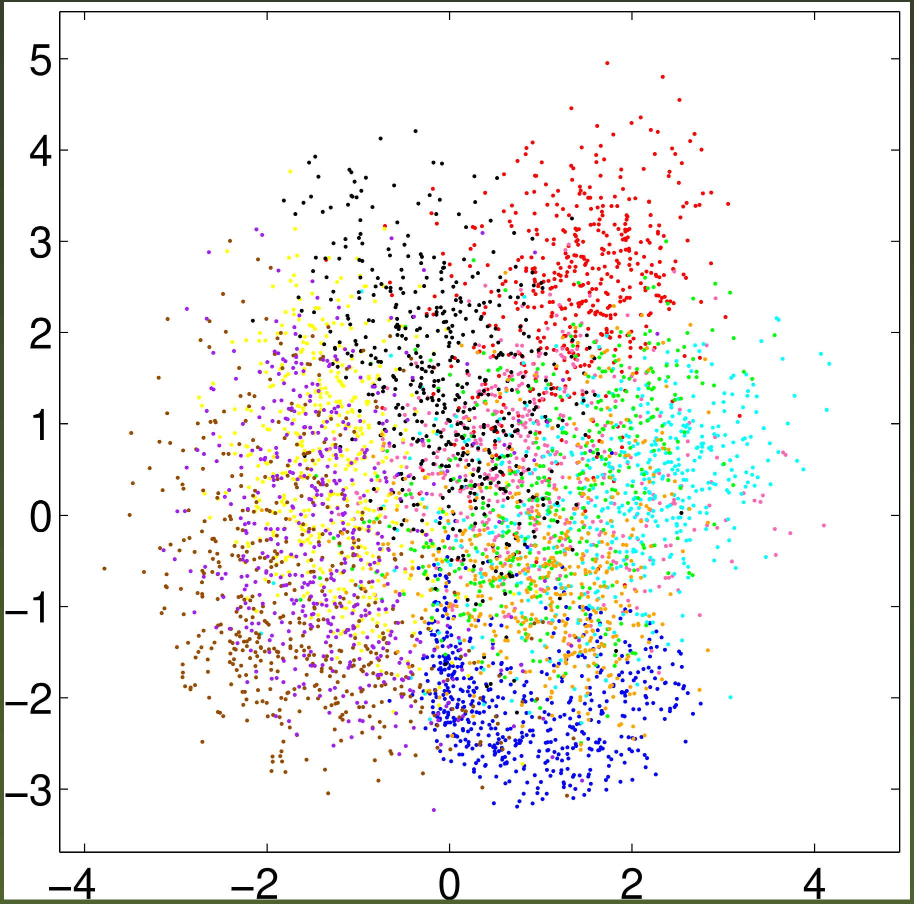
2. Parametric nonlinear embeddings: experiment (cont.)

MNIST digits visualised with EE using a linear mapping ($\mathbf{F}(\mathbf{y}) = \mathbf{A}\mathbf{y} + \mathbf{b}$).





initial, free embedding $E(\mathbf{X})$



Final parametric embedding $\mathbf{F}(Y)$

2. Parametric nonlinear embeddings (cont.)

Three spaces:

- $\mathbf{Z} \in \mathbb{R}^{L \times N}$ = embeddings; nonlinear emb. algorithms operate here.
- $\mathbf{F} \in \mathcal{F}$ = mappings (e.g. linear); chain-rule gradient operates here.
- $(\mathbf{Z}, \mathbf{F})$ = (embeddings,mappings): MAC operates here.

Three embeddings:

- Free embedding: $\mathbf{Z}^* = \arg\min_{\mathbf{Z}} E(\mathbf{Z})$. Best embedding without any constraints.
- Parametric embedding: $\mathbf{F}^* = \arg\min_{\mathbf{F}} P(\mathbf{F}) = E(\mathbf{Z})$ s.t. $\mathbf{Z} = \mathbf{F}(\mathbf{Y})$.
  Best embedding that is realisable by a mapping $\mathbf{F} \in \mathcal{F}$.
- Direct embedding: $\mathbf{F}' = \arg\min_{\mathbf{F}} \left\| \mathbf{Z}^* - \mathbf{F}(\mathbf{Y}) \right\|^2$ (fit $\mathbf{F}$ to free embedding $\mathbf{Z}^*$).
  Equivalent to projecting $\mathbf{Z}^*$ on the feasible set $\{\mathbf{F}(\mathbf{Y}) \in \mathbb{R}^{L \times N} \ \forall \mathbf{F} \in \mathcal{F}\}$. Suboptimal.

These embeddings appear along the path followed by MAC over μ:

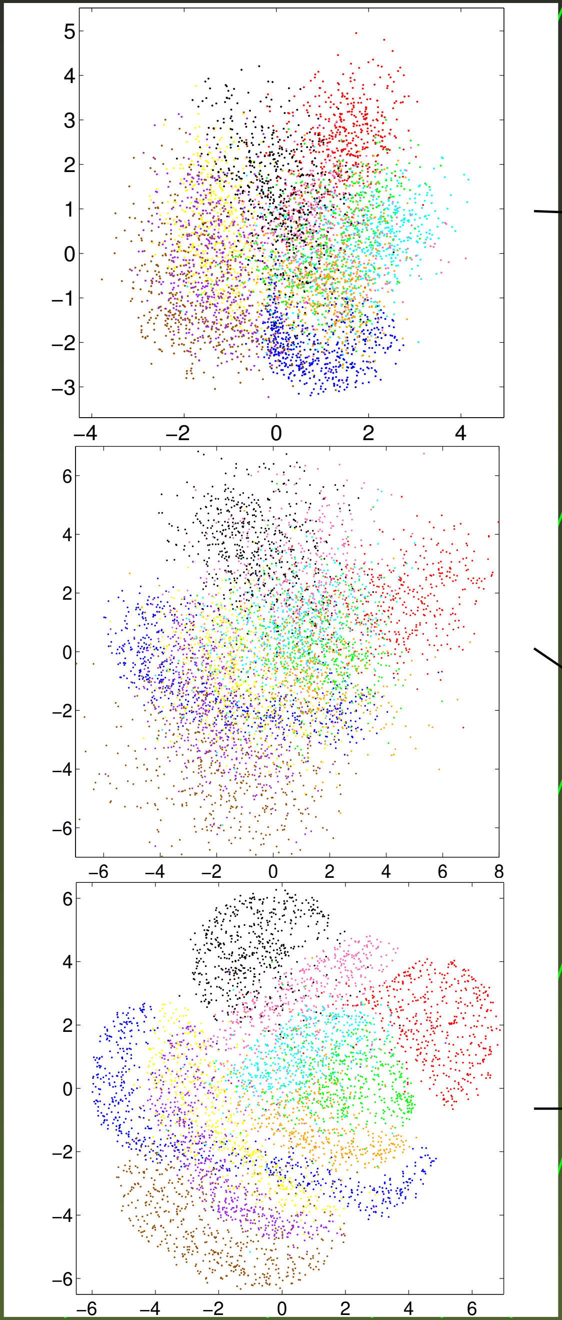
$$\min_{\mathbf{F},\mathbf{Z}} E(\mathbf{Z}) + \frac{\mu}{2} \left\| \mathbf{Z} - \mathbf{F}(\mathbf{Y}) \right\|^2 \begin{cases} \mu = 0, & \text{free embedding} \\ \mu \to 0^+, & \text{direct fit} \\ \mu \to \infty, & \text{parametric embedding.} \end{cases}$$

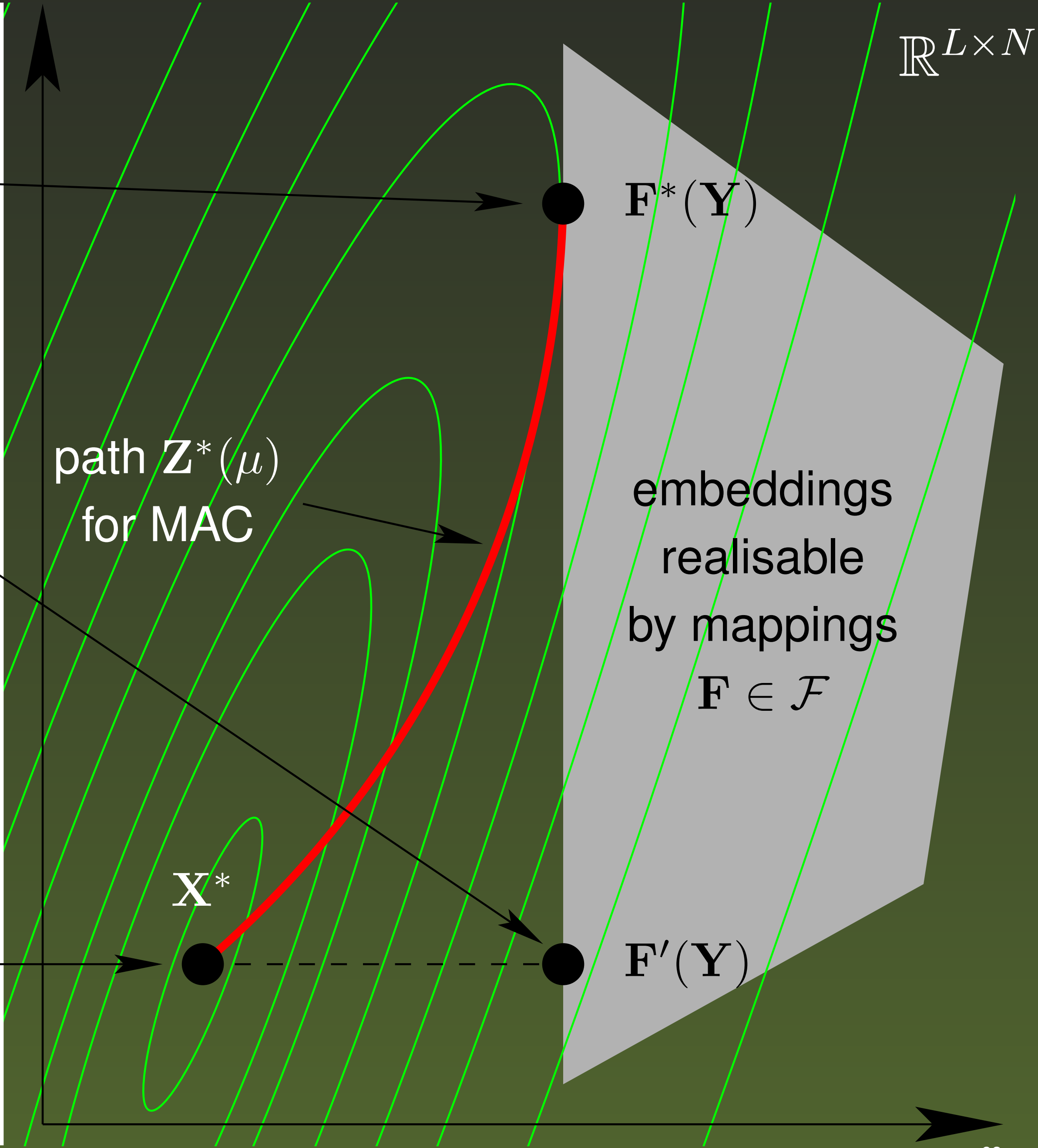
2. Parametric nonlinear embeddings (cont.)

optimal PE
$\mu \to \infty$

direct fit
$\mu \to 0^+$

free embedding
$\mu = 0$





$\mathbb{R}^{L \times N}$
$\mathbf{F}^*(\mathbf{Y})$
path $\mathbf{Z}^*(\mu)$ for MAC
embeddings realisable by mappings $\mathbf{F} \in \mathcal{F}$
$\mathbf{X}^*$
$\mathbf{F}'(\mathbf{Y})$

3. Binary hashing for fast image search

Binary hashing: we want to map a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$ (e.g. an image) to a low-dimensional binary vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{0,1\}^L$ using a binary hash function $\mathbf{h}$, such that Hamming distances in the binary space approximate distances in the original space.

Ex: $\mathbf{h}(\mathbf{x}) = \lrcorner\!\ulcorner(\mathbf{W}\mathbf{x})$ where $\lrcorner\!\ulcorner(\cdot)$ is a step function.

- Application: fast search in image databases
  - Distance calculation fast with hardware support

    XOR then count 1s (POPCNT) instead of floating-point operations
  - Binary database may fit in (fast) memory

    1M points of $D = 300$ floats take 1.2 GB but only 4 MB with a 32-bit code
- In order to learn the hash function $\mathbf{h}$ from data, we formulate an objective function $E(\mathbf{h})$ that preserves distances in some way.

  Many formulations exist.
- Difficult optimisation because E depends on the output of $\mathbf{h}$, which is binary, so E is piecewise constant, nonsmooth and nonconvex. The problem is usually NP-hard. The chain rule does not apply.

3. Binary hashing for fast image search (cont.)

We introduce as auxiliary coordinates $\mathbf{z}_n \in \{0,1\}^L$ the binary code of each $\mathbf{x}_n$. This confines the difficult part of the optimisation to the $\mathbf{Z}$ step. The resulting MAC algorithm alternates:

- training L binary classifiers on $(\mathbf{X}, \mathbf{Z})$, in parallel
  done with existing algorithms, e.g. linear SVM
- updating the N auxiliary coordinates $\mathbf{Z}$
  this is a binary optimisation that we need to solve

as $\mu \rightarrow \infty$ (in fact, we can prove the iterations stop for a *finite* μ).

If we want to use a different hash function $\mathbf{h}$ (linear, neural net...), we simply call a different classifier routine in the $\mathbf{h}$ step as a black box.

We consider two generic types of objective function:

- Binary autoencoder: unsupervised
  preserve (Euclidean) distances between original feature vectors
- Affinity-based objective function: supervised
  neighbourhood information provided by the user (e.g. class labels)

3a. Binary autoencoder

M. Carreira-Perpiñán and R. Raziperchikolaei: *Hashing with binary autoencoders*, CVPR 2015.

Original (nested) objective function (reconstruction error), $\mathbf{W} = (\mathbf{h}, \mathbf{f})$:

$$E_{\text{nested}}(\mathbf{h},\mathbf{f}) = \sum_{n=1}^{N} \|\mathbf{x}_n - \underbrace{\mathbf{f}(\mathbf{h}(\mathbf{x}_n))}_{\text{nesting}}\|^2 \qquad \begin{cases} \text{encoder } \mathbf{h}\text{: } \mathbb{R}^D \rightarrow \{0,1\}^L & \text{(hash functions)} \\ \text{decoder } \mathbf{f}\text{: } \{0,1\}^L \rightarrow \mathbb{R}^D \end{cases}$$

We apply the MAC design pattern:

❶ MAC-constrained problem: auxiliary coordinates $\mathbf{Z} = (\mathbf{z}_1, \dots, \mathbf{z}_N)$:

$$\min_{\mathbf{h},\mathbf{f},\mathbf{Z}} \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 \quad \text{s.t.} \quad \mathbf{z}_n = \mathbf{h}(\mathbf{x}_n),\ \mathbf{z}_n \in \{0,1\}^L,\ n = 1,\dots,N.$$

❷ MAC-penalised objective (e.g. quadratic-penalty function), as $\mu \rightarrow \infty$:

$$\min_{\mathbf{h},\mathbf{f},\mathbf{Z}} \sum_{n=1}^{N} \Big(\underbrace{\|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2}_{\text{unnested, single-layer}}\Big) \quad \text{s.t.} \quad \begin{cases} \mathbf{z}_n \in \{0,1\}^L \\ n = 1,\dots,N. \end{cases}$$

3a. Binary autoencoder (cont.)

❸ Alternating optimisation over $\mathbf{W} = (\mathbf{h}, \mathbf{f})$ and $\mathbf{Z}$:

- $\mathbf{W}$ step (submodels): one submodel per BA unit:
  - $\mathbf{h}$: fit L binary classifiers to $\{(\mathbf{x}_n, \mathbf{z}_n)\}_{n=1}^N$
  - $\mathbf{f}$: fit D regression mappings to $\{(\mathbf{z}_n, \mathbf{x}_n)\}_{n=1}^N$

  } $L + D$ submodels in parallel

  solved using existing algorithms

- $\mathbf{Z}$ step (coordinates): for each point $\{\mathbf{z}_n\}_{n=1}^N$ solve a binary optimisation in L variables:

  } N coordinates in parallel

$$\min_{\mathbf{z}_n \in \{0,1\}^L} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$$

This is usually NP-hard (for example, if $\mathbf{f}$ is linear, it is a binary quadratic problem)
. . . but the number of variables is small.

- If $L \lesssim 10$ we can afford an exhaustive search over the 2^L codes.
- Otherwise, we use alternating optimisation over (groups of) bits.

We can use some other tricks to speed this up (incremental computation, good initialisation, etc.).

3a. Binary autoencoder: MAC algorithm pseudocode

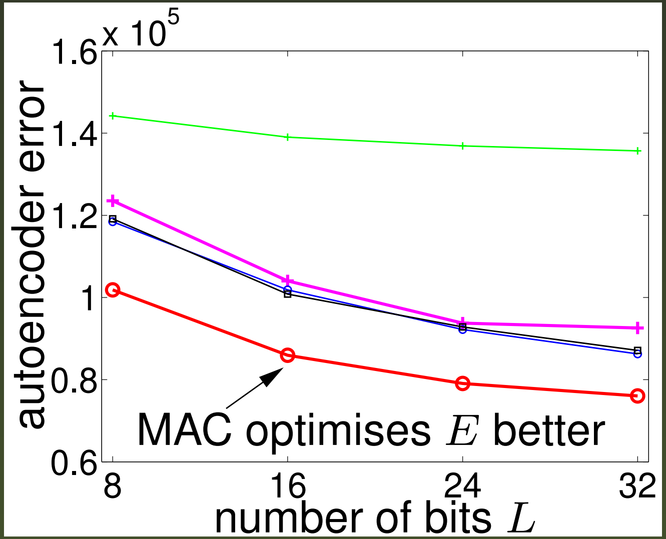
```
input $\mathbf{X}_{D\times N} = (\mathbf{x}_1,\dots,\mathbf{x}_N)$, $L \in \mathbb{N}$
initialise $\mathbf{Z}_{L\times N} = (\mathbf{z}_1,\dots,\mathbf{z}_N) \in \{0,1\}^{LN}$
for $\mu = 0 < \mu_1 < \dots < \mu_\infty$
    parfor $l = 1,\dots,L$                                              $\mathbf{W}$ step: $\mathbf{h}$
        $h_l(\cdot) \leftarrow$ fit SVM to $(\mathbf{X}, \mathbf{Z}_{\cdot l})$
    parfor $d = 1,\dots,D$                                              $\mathbf{W}$ step: $\mathbf{f}$
        $f_d(\cdot) \leftarrow$ least-squares fit to $(\mathbf{Z}, \mathbf{X}_{d\cdot})$
    parfor $n = 1,\dots,N$                                              $\mathbf{Z}$ step
        $\mathbf{z}_n \leftarrow \arg\min_{\mathbf{z}_n \in \{0,1\}^L} \|\mathbf{x}_n - \mathbf{f}(\mathbf{z}_n)\|^2 + \mu\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2$
    if no change in $\mathbf{Z}$ and $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ then stop
return $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$
```

Repeatedly solve: classification ($\mathbf{h}$), regression ($\mathbf{f}$), binarisation ($\mathbf{Z}$).

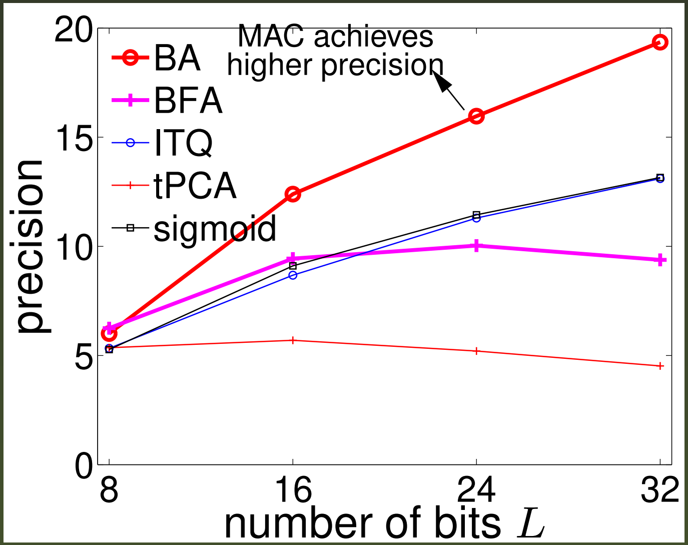
3a. Binary autoencoder: experiment

NUS-WIDE-LITE dataset, $N = 27\,807$ training/ $27\,808$ test images, $D = 128$ wavelet features, linear hash function $\mathbf{h}(\mathbf{x}) = s(\mathbf{W}\mathbf{x})$.

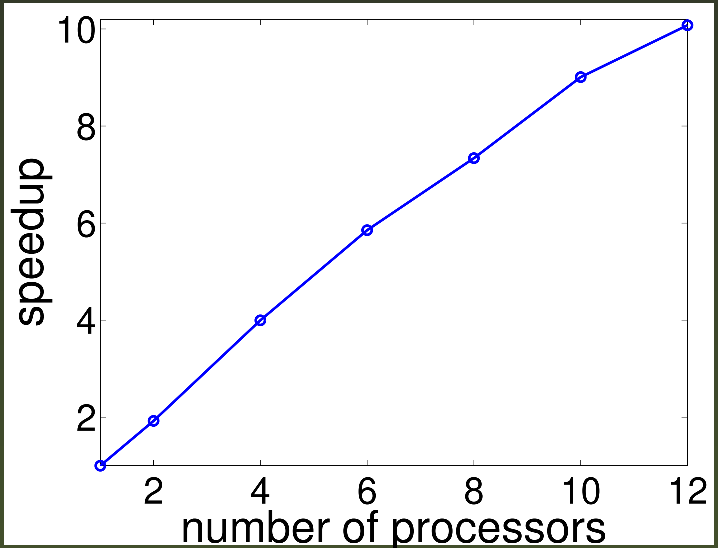
reconstruction error wrt L



precision in retrieval



parallel processing speedup



Optimising the binary autoencoder with MAC achieves both a lower reconstruction error and a better precision/recall in searching than if using other approximate methods to optimise it (e.g. relaxing/truncating, using a sigmoid instead of a step function, etc.).

It also parallelises very well.

3b. Affinity-based objective function

R. Raziperchikolaei and M. Carreira-Perpiñán: *Optimizing affinity-based binary hashing using auxiliary coordinates*, NIPS 2016, arXiv:1501.05352.

Original (nested) objective function: $\mathcal{L}(\mathbf{h}) = \sum_{n,m=1}^{N} L(\mathbf{h}(\mathbf{x}_n), \mathbf{h}(\mathbf{x}_m); w_{nm})$

where L is a loss function that compares the codes for two images with their affinity w_{nm}.

Many formulations exist, e.g. $\begin{cases} w_{nm} = -1/+1/0 \text{ if similar/dissimilar/don't know} \\ L_{\text{KSH}}(\mathbf{z}_n, \mathbf{z}_m; w_{nm}) = (\mathbf{z}_n^T \mathbf{z}_m - L w_{nm})^2. \end{cases}$

❶ MAC-constrained problem:

$$\min_{\mathbf{h},\mathbf{Z}} \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m; w_{nm}) \quad \text{s.t.} \quad \mathbf{z}_n = \mathbf{h}(\mathbf{x}_n),\ \mathbf{z}_n \in \{0,1\}^b,\ n = 1, \dots, N.$$

❷ MAC-penalised objective:

$$\min_{\mathbf{h},\mathbf{Z}} \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m; w_{nm}) + \mu \sum_{n=1}^{N} \|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\|^2 \quad \text{s.t.} \quad \begin{cases} \mathbf{z}_n \in \{0,1\}^L \\ n = 1, \dots, N. \end{cases}$$

3b. Affinity-based objective function: alternating optimis

❸ Alternating optimisation over $(\mathbf{h}, \mathbf{Z})$:

- $\mathbf{h}$ step: fit L binary classifiers to $\{(\mathbf{x}_n, \mathbf{z}_n)\}_{n=1}^N$, in parallel.
- $\mathbf{Z}$ step: minimise the following regularised binary embedding:

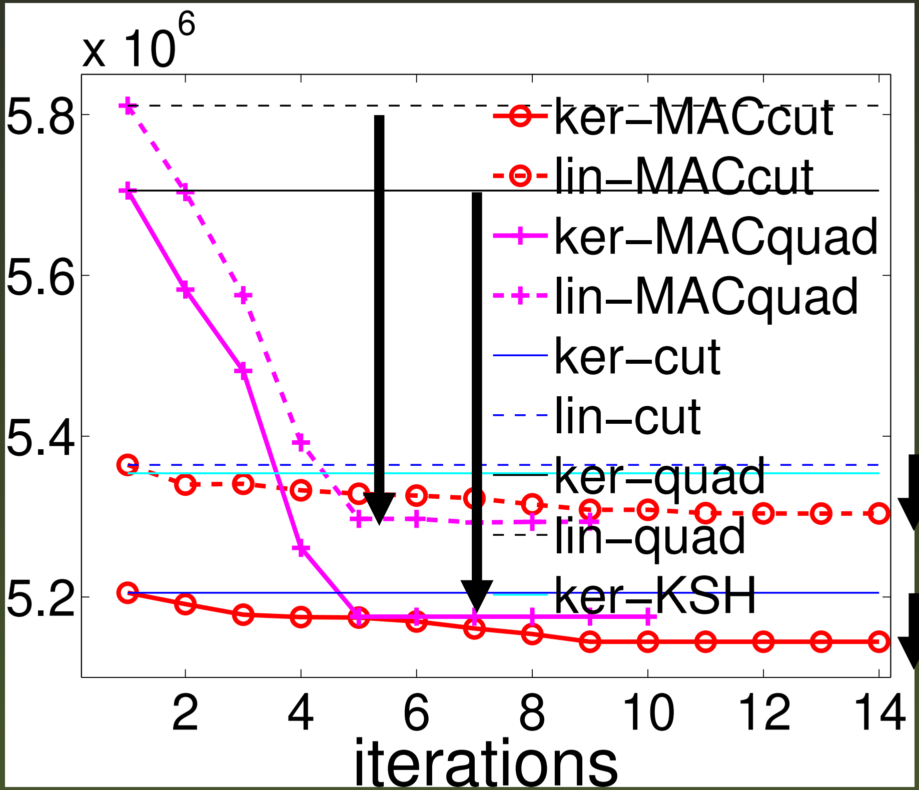
$$\min_{\mathbf{Z}} \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m; w_{nm}) + \mu \sum_{n=1}^{N} \left\| \mathbf{z}_n - \mathbf{h}(\mathbf{x}_n) \right\|^2 \quad \text{s.t.} \quad \begin{cases} \mathbf{z}_n \in \{0,1\}^L \\ n = 1, \dots, N. \end{cases}$$

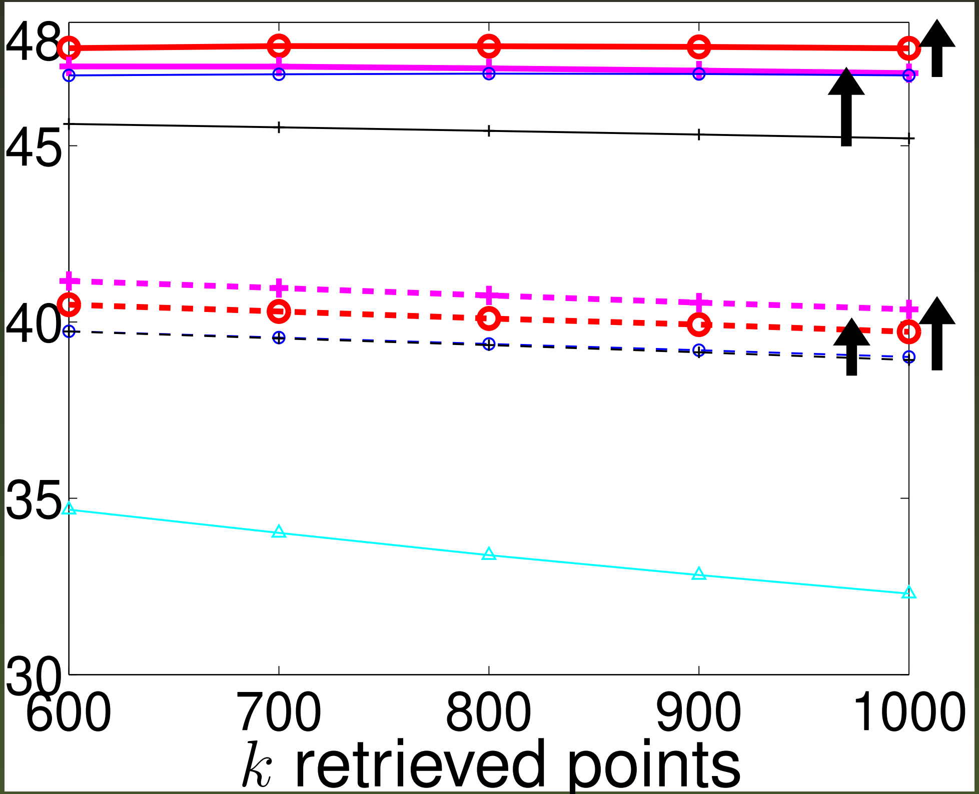
This is a binary optimisation over NL binary variables, usually NP-hard. We reuse existing approximate methods to optimise this.

Alternating optimisation over the lth bit of each code for $l = 1, \dots, L$, with each optimisation over the N binary variables solved approximately by relaxation/truncation or by GraphCut (Lin et al. 2013, 2014).

3b. Affinity-based objective function: experiment

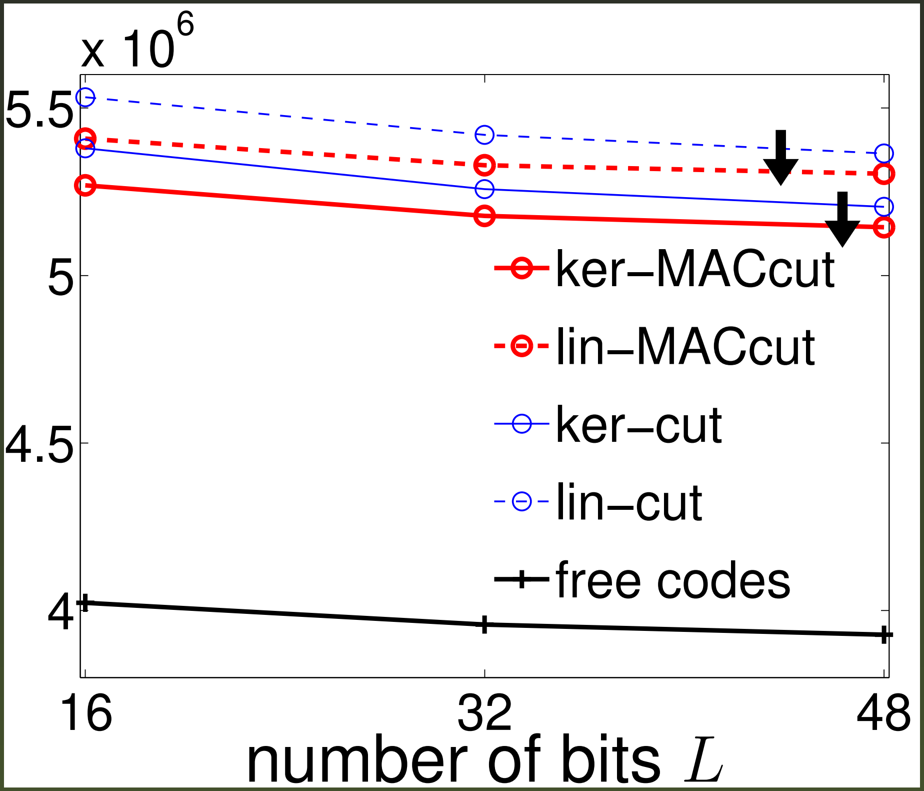
CIFAR dataset, $N = 58\,000$ training/ $2\,000$ test images, $D = 320$ SIFT features, $L = 48$ bits.

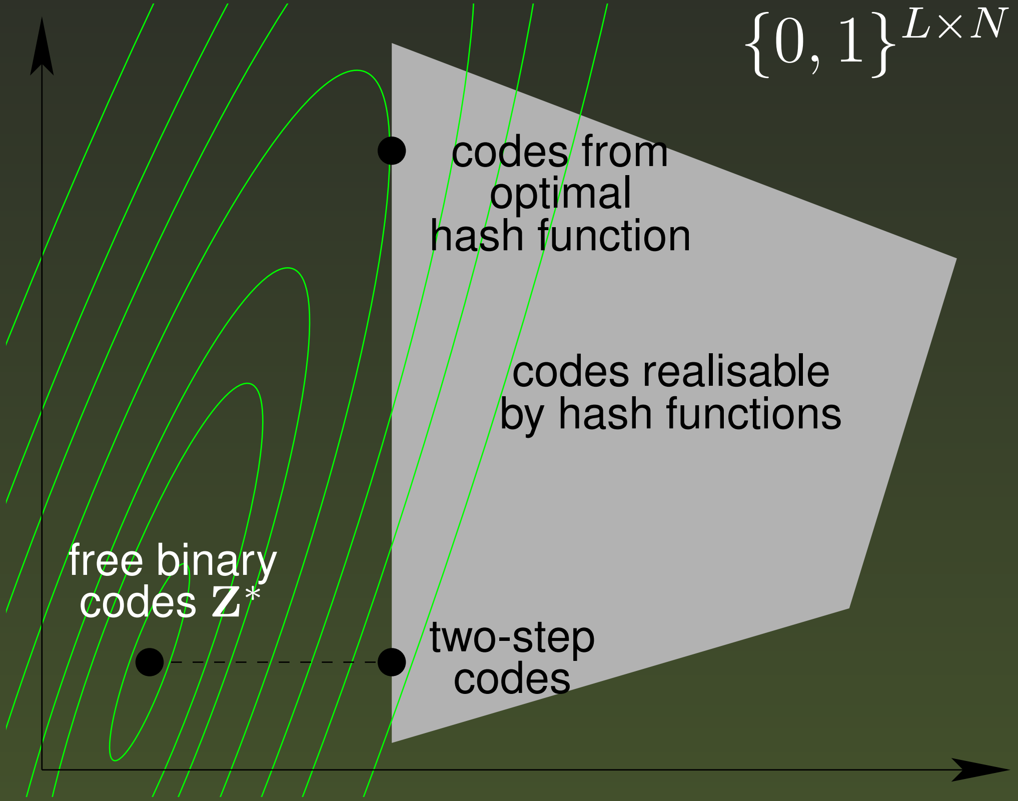




MAC finds hash functions with significantly better objective function value and precision/recall than two-step approaches (which first optimise the binary codes and then fit to them the hash function) and other approximate methods.

3b. Affinity-based objective function: experiment (cont.)





$\{0,1\}^{L\times N}$
codes from optimal hash function
codes realisable by hash functions
free binary codes $\mathbf{Z}^*$
two-step codes

The free binary codes $\mathbf{Z}^*$ are an (approximate) minimiser of the loss $\mathcal{L}(\mathbf{Z})$ without any constraint (i.e., disregarding the hash function). They are usually not realisable by any (linear, say) hash function ($\mathbf{h}(\mathbf{X}) \neq \mathbf{Z}\ \forall\mathbf{h}$). Two-step methods project the free codes onto the feasible set of codes realisable by linear hash functions, which is suboptimal. MAC gradually adjusts both the codes and the hash function so they eventually match, which results in a better solution.

4. Learning radial basis function (RBF) networks (unpublishe results

Original (nested) objective function (regression):

$$E_{\text{nested}}(\mathbf{W}, \boldsymbol{\phi}) = \sum_{n=1}^{N} \|\mathbf{y}_n - \underbrace{\mathbf{W}\boldsymbol{\phi}(\mathbf{x}_n)}_{\text{nesting}}\|^2 + \lambda R(\mathbf{W}), \qquad \boldsymbol{\phi}(\mathbf{x}) = \begin{pmatrix} \phi_1(\mathbf{x}) \\ \dots \\ \phi_M(\mathbf{x}) \end{pmatrix}$$

where $\phi_m(\mathbf{x})$ is the mth basis function, e.g. $\phi_m(\mathbf{x}) = \exp\left(-\frac{1}{2}\left\|\frac{\mathbf{x}-\mathbf{c}_m}{\sigma}\right\|^2\right)$ (Gaussian with centre $\mathbf{c}_m \in \mathbb{R}^D$ and fixed bandwidth $\sigma > 0$).

We apply the MAC design pattern:

❶ MAC-constrained problem: auxiliary coordinates $\mathbf{Z} = (\mathbf{z}_1, \dots, \mathbf{z}_N)$:

$$\min_{\boldsymbol{\phi}, \mathbf{W}, \mathbf{Z}} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{W}\mathbf{z}_n\|^2 + \lambda R(\mathbf{W}) \quad \text{s.t.} \quad \mathbf{z}_n = \boldsymbol{\phi}(\mathbf{x}_n),\ n = 1, \dots, N.$$

❷ MAC-penalised objective (e.g. quadratic-penalty function), as $\mu \to \infty$:

$$\min_{\boldsymbol{\phi}, \mathbf{W}, \mathbf{Z}} \sum_{n=1}^{N} \Big(\underbrace{\|\mathbf{y}_n - \mathbf{W}\mathbf{z}_n\|^2 + \mu\|\mathbf{z}_n - \boldsymbol{\phi}(\mathbf{x}_n)\|^2}_{\text{unnested, single-layer}}\Big) + \lambda R(\mathbf{W})$$

4. Learning radial basis function (RBF) networks (cont.)

❸ Alternating optimisation of the penalty function for fixed μ:

- $\mathbf{W}$ step: fit a linear regression to $\{(\mathbf{z}_n, \mathbf{y}_n)\}_{n=1}^N$

  If $R(\mathbf{W}) = \|\mathbf{W}\|_2^2$: closed-form solution (linear system).
  If $R(\mathbf{W}) = \|\mathbf{W}\|_1$: a 1D Lasso regression per row of $\mathbf{W}$.

- ϕ step: fit basis function $m = 1, \dots, M$ to $\{(\mathbf{x}_n, z_{nm})\}_{n=1}^N$

  A 1D regression per basis function ϕ_m using as outputs the auxiliary coordinates.
  This may be solved in different ways depending on ϕ_m. For example:

  - If $\mathbf{c}_m \in \mathbb{R}^D$ and ϕ_m is differentiable, use a gradient- or Newton-based method. The optimal solution has the form of a weighted average of the training points: $\mathbf{c}_m = \sum_{n=1}^N \alpha_n \mathbf{x}_n / \sum_{n=1}^N \alpha_n$ for certain $\{\alpha_n\}_{n=1}^N \subset \mathbb{R}$.
  - If we constrain $\mathbf{c}_m$ to be one of the data points $\{\mathbf{x}_n\}_{n=1}^N$, we fit $\mathbf{c}_m$ exactly by enumeration over each $\mathbf{x}_n$. This avoids a combinatorial explosion over all N^M possible choices for each of the M centres, instead trying only NM choices.

  } in parallel

- $\mathbf{Z}$ step: set $\mathbf{Z} = (\mathbf{W}^T\mathbf{W} + \mu\mathbf{I})^{-1}(\mathbf{W}^T\mathbf{y}_n + \mu\boldsymbol{\phi}(\mathbf{x}_n))$

  Closed-form solution: linear system.

  } in parallel

5. Learning feature transformations for decision trees

6. Best-subset feature selection ($^{\text{unpublished}}_{\text{results}}$)

Black-box optimisation of nested systems

Many algorithms involving an inner-loop optimisation have benefitted from calling numerical linear algebra subroutines as a black box:

- Linear system, eigenproblem, FFT…
- No need to worry about step sizes, search directions, gradients, Hessians… Just call the subroutine and get the solution.
- Thanks to existing, robust, well-developed code

  LAPACK, BLAS, etc.

As the previous examples of using MAC show, we can also benefit from calling machine learning solvers as a black box in nested systems:

- Logistic regression, SVM classification, linear/nonlinear regression, Lasso, nearest-neighbour, decision tree, enumeration…
  Just call the subroutine and get the solution.
- Robust, well-developed code is becoming gradually available

  `scikit-learn`, TensorFlow, etc.

Black-box optimisation of nested systems (cont.)

Are MAC algorithms the fastest way to optimise a nested system?

- Not necessarily. They do seem competitive in many cases, particularly if parallel processing is available and a high-accuracy solution is not required. In other cases (e.g. with nondifferentiable functions) there may be no simple alternative.
- But the most important advantage is that they are simple to construct and scalable.
- For a given nested system, it is always possible to design a specialised, fast algorithm. But there is an exponential number of nested systems by combining simple models (K layers with M model choices per layer means M^K nested models).

Black-box optimisation of nested systems (cont.)

The same principle is well known in software engineering:

- How compilers are designed:
  - We have N choices of high-level language (C, Fortran. . .).
  - We have M choices of target system (CPU, libraries, etc.).

  Rather than designing a specific compiler for each combination of high-level language and target system (NM combinations) one designs one front-end for each high-level language and one back-end for each target system ($N + M$ choices), connected by an intermediate code representation.
- The Unix philosophy:
  - write programs that do one thing and do it well (modularity)
  - write programs to work together (composition)

  rather than writing a specific, complex program for every task.

Conclusion: the method of auxiliary coordinates (MAC)

- Jointly optimises a nested function over all its parameters.
- Restructures the nested problem into a sequence of iterations with independent subproblems; a coordination-minimisation algorithm:
  - minimisation step: minimise (train) layers
  - coordination step: coordinate layers.
- Advantages:
  - easy to develop, reuses existing algorithms or code for shallow models as a black box
  - convergent under some assumptions
  - introduces embarrassing parallelism
  - can work with nondifferentiable or discrete layers
  - can do model selection “on the fly”.
- Widely applicable in machine learning, computer vision, speech, NLP, etc.

A long-term goal

Develop a software tool where:

- A non-expert user builds a nested system by connecting individual black-box modules from a library, LEGO-like:

  linear, SVM, RBF net, logistic regression, feature selector, decision tree. . .

- The tool automatically:
  - selects the best way to apply MAC

    choice of auxiliary coordinates, choice of optimisation algorithms, etc.
  - reuses training algorithms from a library
  - maps the overall algorithm to a target distributed architecture
  - generates runtime code.

Outline

- Nested (deep) and shallow systems
- Training nested systems: chain-rule gradient; greedy layerwise
- The method of auxiliary coordinates (MAC)
  - Design pattern
  - Convergence guarantees
  - Practicalities
  - Related work
  - Model selection “on the fly”
- A gallery of nested models trainable with MAC
  1. Learning low-dimensional features for classification: low-dim SVM
  2. Parametric nonlinear embeddings: parametric t-SNE, EE
  3. Binary hashing for fast image search: binary autoencoder
  4. Learning radial basis function (RBF) networks
  5. Learning feature transformations for decision trees: neural net + tree
  6. Best-subset feature selection

☞ Distributed optimisation with MAC: ParMAC

Distributed optimisation with MAC: ParMAC

Large dataset stored over P machines (N/P points/machine).

Distributing the computation necessary for faster training or dataset may not fit in a single machine.

Although MAC has inherent parallelism, in the distributed setting the machines must communicate.
How to reduce the communication? What speedups can we expect?

In MAC, the specific algorithm varies from problem to problem. It depends on the model, objective function, how the auxiliary coordinates are introduced, the penalty function, how the steps are optimised… We can achieve steps that are closed-form, convex, nonconvex, binary, etc.

But the following always hold:

- $\mathbf{Z}$ step: N independent subproblems $\mathbf{z}_1, \dots, \mathbf{z}_N$, one per data point.
  For objective functions of the form $\sum_n L(\mathbf{x}_n; \mathbf{W})$.
  Each $\mathbf{z}_n$ depends on all or part of the current model.
- $\mathbf{W}$ step: M independent submodels. Examples:
  - Binary autoencoder: $M =$ # hash functions and linear decoders.
  - Deep net: $M =$ # hidden units.

  Each submodel depends on all or part of the data and coordinates.

Distributed optimisation with MAC: ParMAC (cont.)

Computation vs communication:

- The cost of communicating (through the memory hierarchy or a network) greatly exceeds the cost of computing in time & energy.
- It is essential to limit the amount of communication between machines so it does not obliterate the benefit of parallelism.

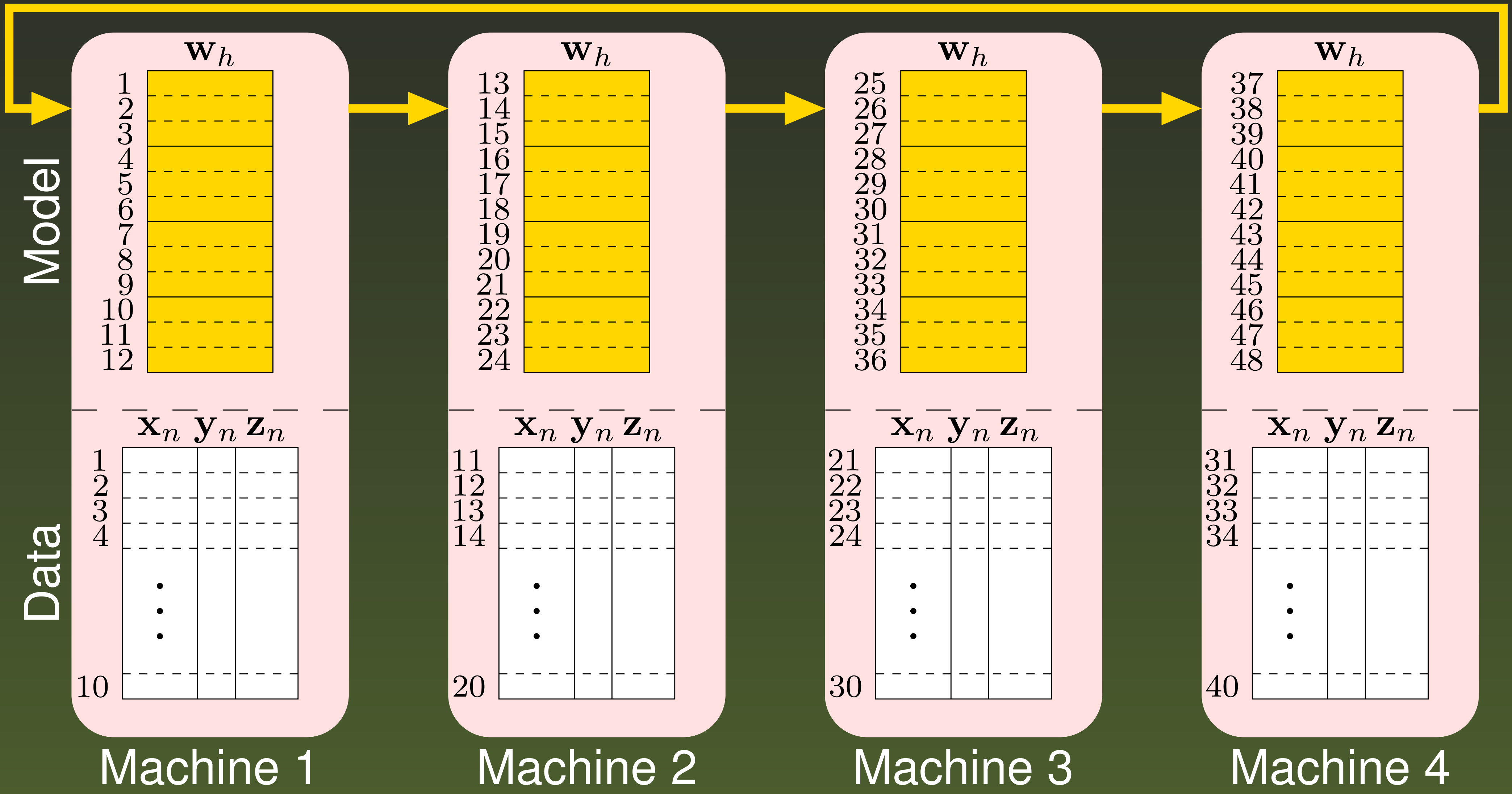
Basic ideas in ParMAC:

- Never communicate training data $(\mathbf{X}, \mathbf{Y})$ or coordinates $\mathbf{Z}$.
- Each machine keeps a disjoint portion of $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ corresponding to its subset of points.
- Only model parameters are communicated, during the $\mathbf{W}$ step:
  - circular topology: implements SGD

How does this affect the $\mathbf{W}$ and $\mathbf{Z}$ steps?

M. Carreira-Perpiñán and M. Alizadeh: *ParMAC: distributed optimisation of nested functions, with application to learning binary autoencoders*, SysML 2019, arXiv:1605.09114.

ParMAC: setup



$P = 4$ nodes, $M = 12$ submodels $\mathbf{w}_h$ and $N = 40$ data points $(\mathbf{x}_n, \mathbf{y}_n)$. Submodels h, $h + M$, $h + 2M$ and $h + 3M$ are copies of submodel h, but only one of them is the most currently updated. At the end of the $\mathbf{W}$ step all copies are identical.

ParMAC: the $\mathbf{Z}$ step

The $\mathbf{Z}$ step behaves just as in ordinary MAC:

- Before the $\mathbf{Z}$ step starts, each machine contains all the (just updated) submodels, as well as its portion of the data and auxiliary coordinates.
- Each machine processes its auxiliary coordinates independently of all other machines.
- No communication occurs.
- At the end of the $\mathbf{Z}$ step, each machine contains its portion of the data and (just updated) auxiliary coordinates, as well as all the submodels.

This step has extremely high parallelism: N independent subproblems, one per data point, **no** communication. It is never a bottleneck in the parallel speedup.

ParMAC: the $\mathbf{W}$ step

This step has high parallelism: M independent subproblems, one per submodel, but **with** communication. It is the speedup bottleneck.

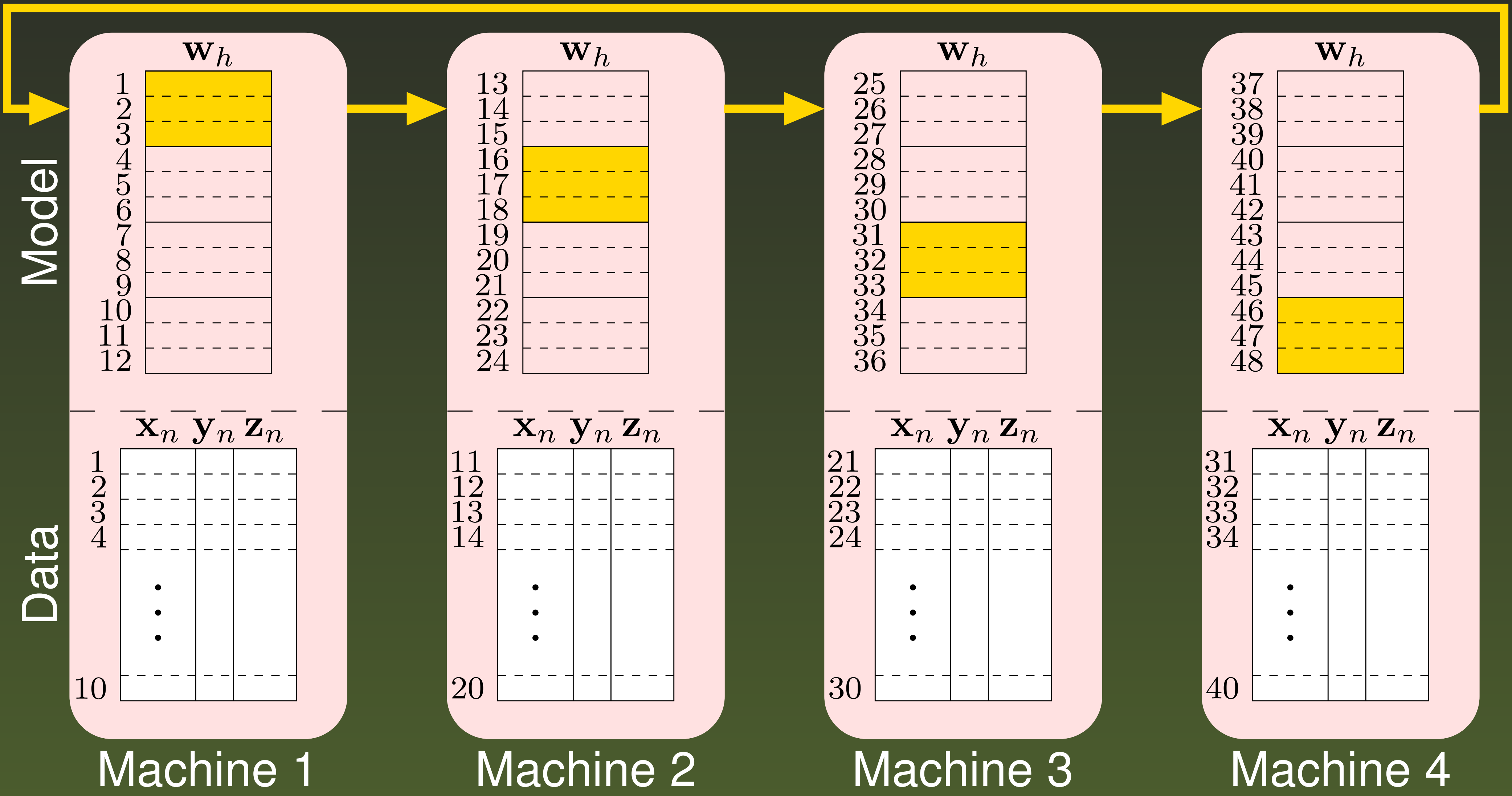
Synchronous version (easier to analyse but less practical):

- Before the $\mathbf{W}$ step starts, each machine contains its portion of the data and (just updated) auxiliary coordinates and all the submodels.
- At each clock tick, in parallel for the P machines, each machine:
  - updates a different portion M/P of the submodels,
  - then sends the submodels updated to its successor.

  After P ticks, each submodel has visited all P machines and completed one "epoch".
- This is repeated e times (for e epochs).
- A final communication-only epoch ensures each machine contains the entire updated model.

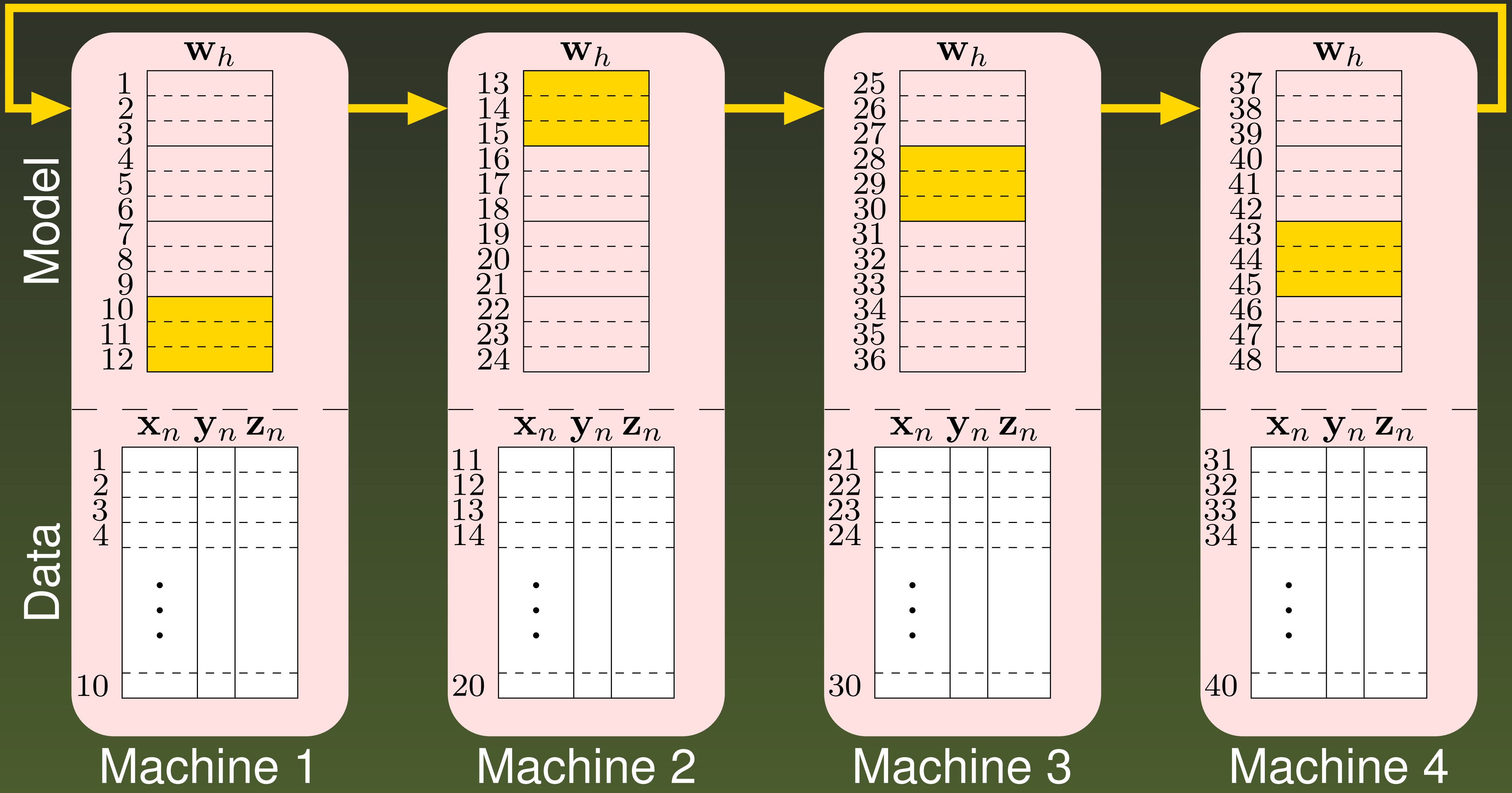
Asynchronous version. Each machine keeps a queue of submodels to be processed, and repeatedly performs the following operations: extract a submodel from the queue, process it (except in epoch $e+1$) and send it to the machine's successor (which will insert it in its queue).

ParMAC: the W step (cont.)



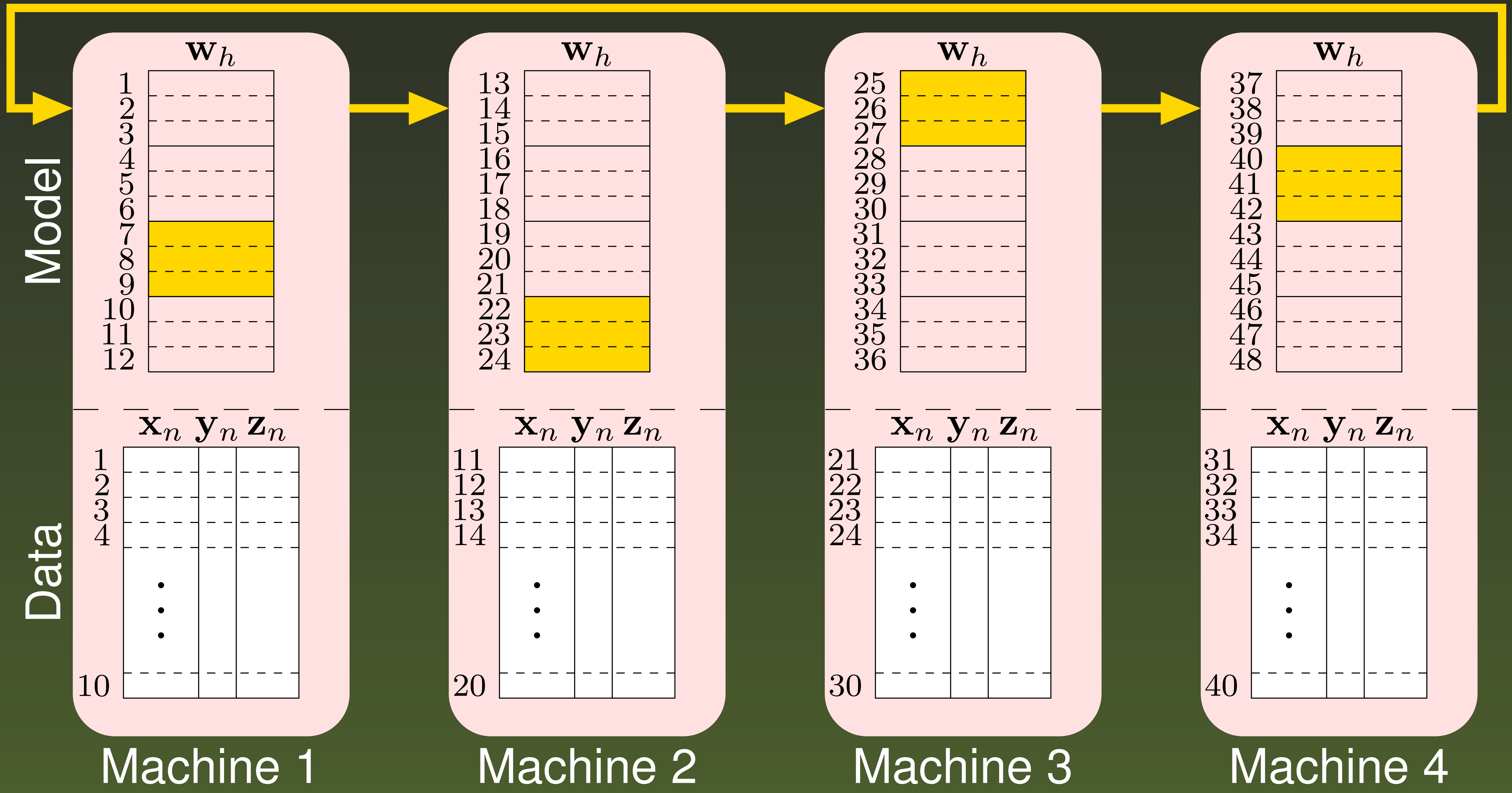
Computation + communication epoch, tick 0

ParMAC: the $\mathbf{W}$ step (cont.)



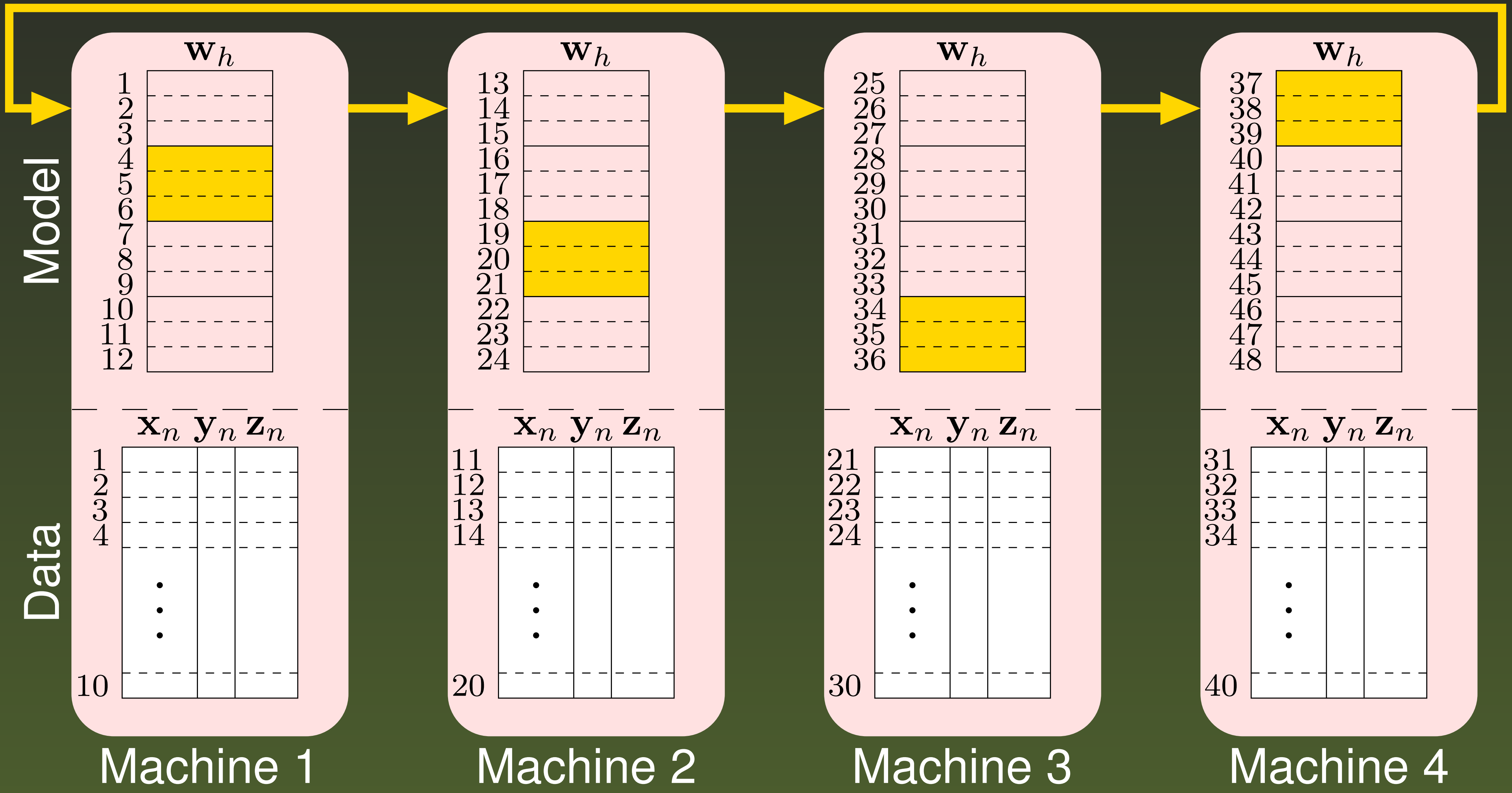
Computation + communication epoch, tick 1

ParMAC: the $\mathbf{W}$ step (cont.)



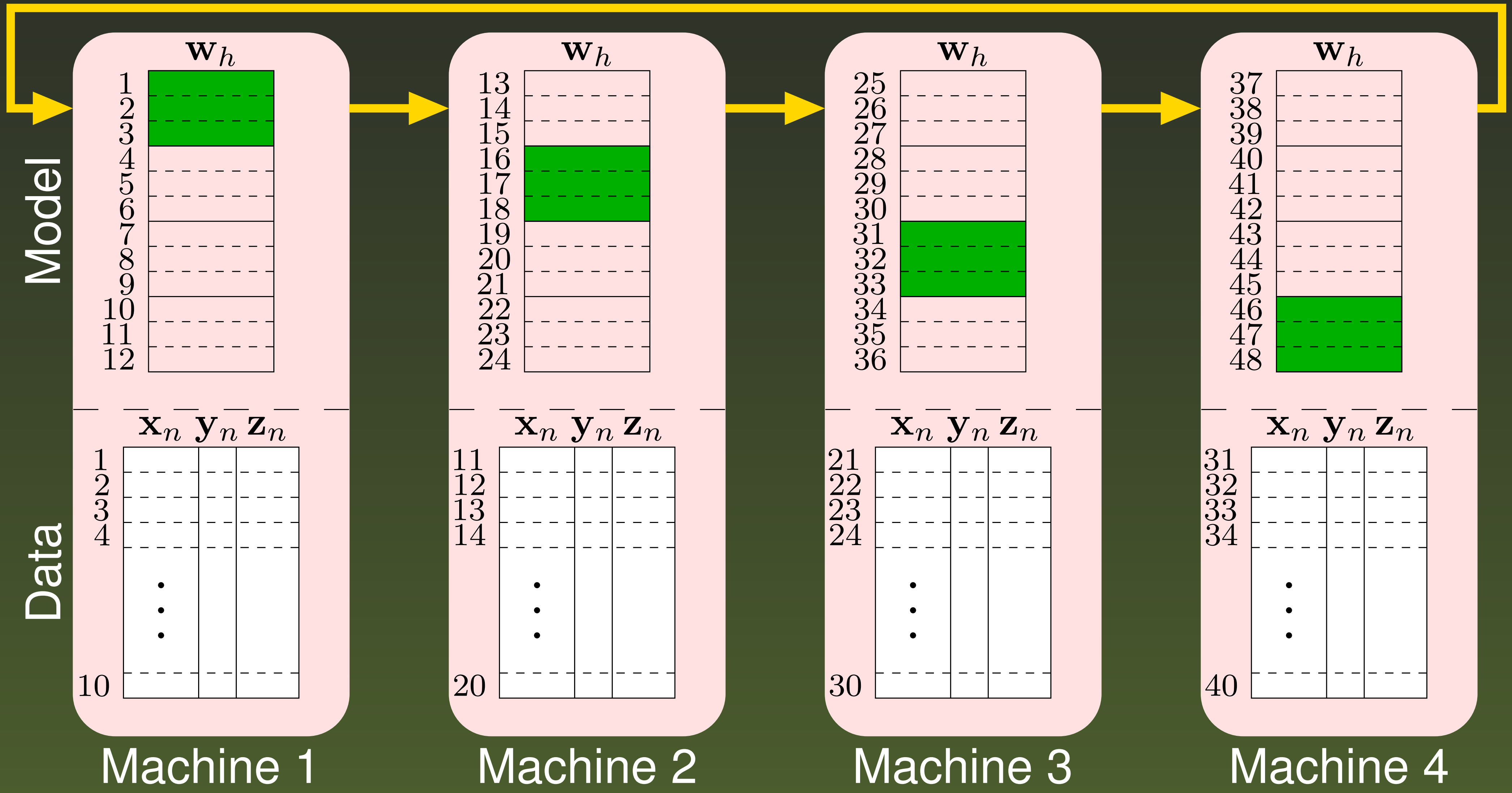
Computation + communication epoch, tick 2

ParMAC: the $\mathbf{W}$ step (cont.)



Computation + communication epoch, tick 3

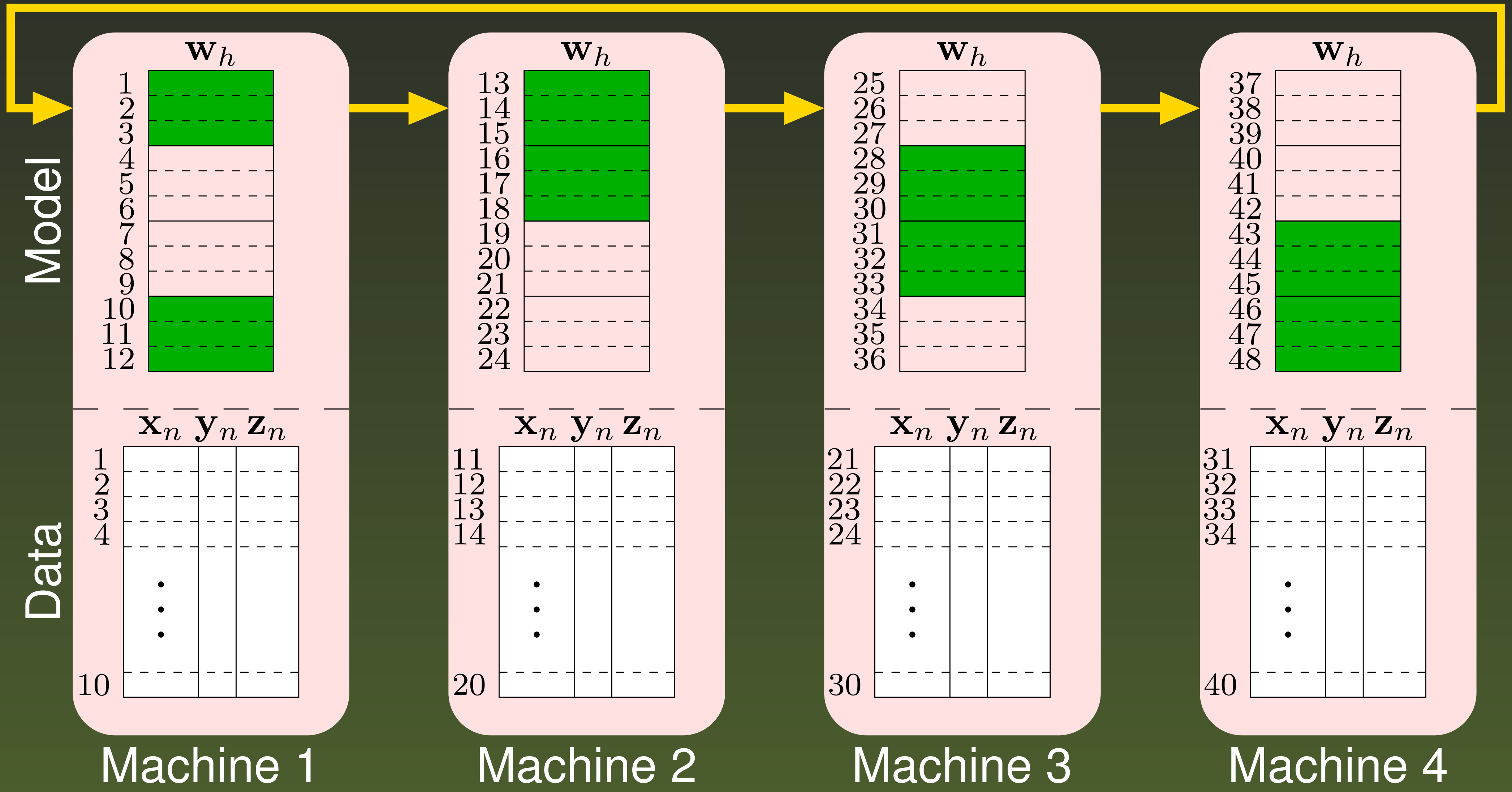
ParMAC: the W step (cont.)



Computation + communication epoch, tick $4 = P$

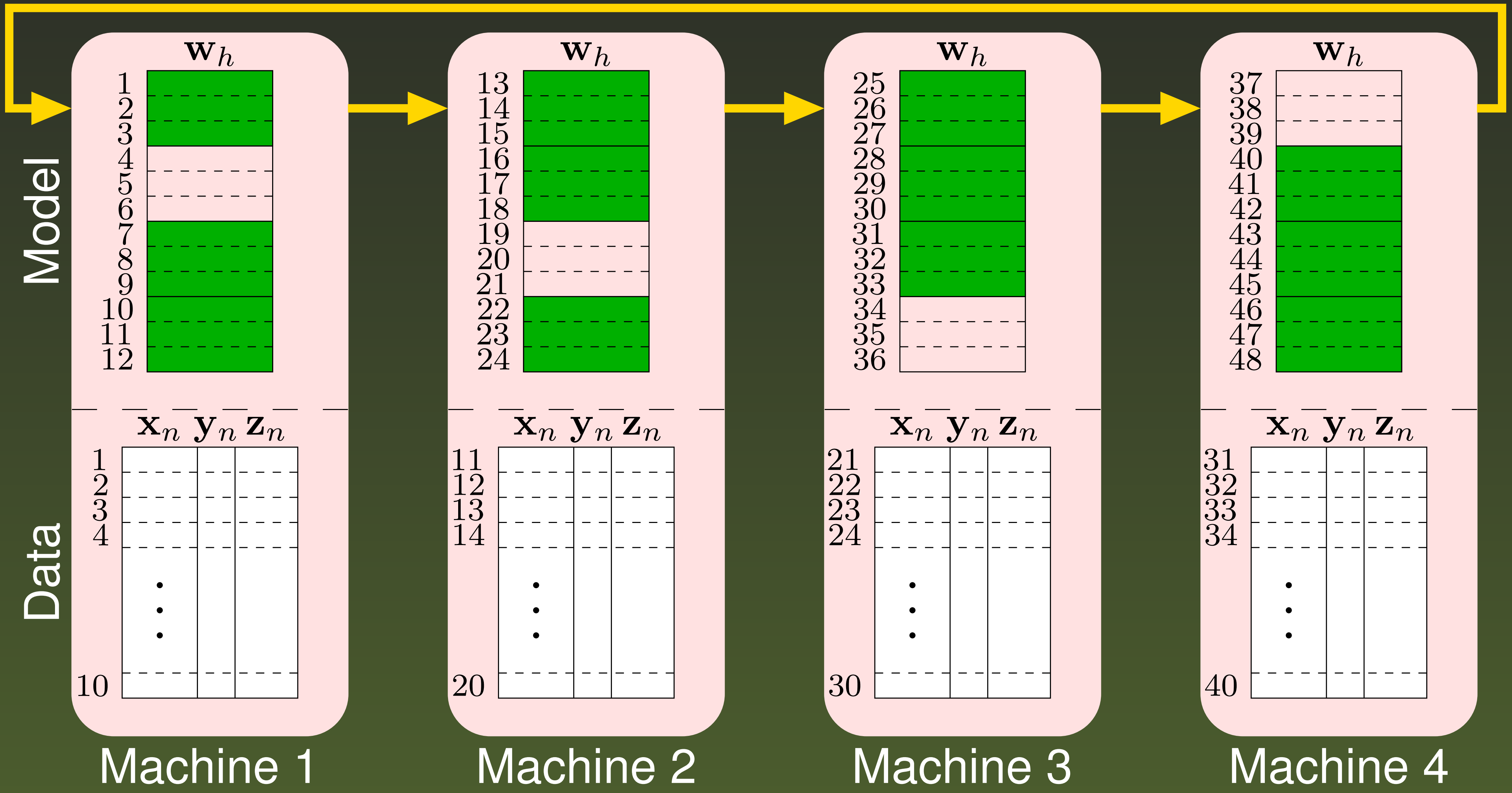
Communication-only (final) epoch, tick 0

ParMAC: the $\mathbf{W}$ step (cont.)



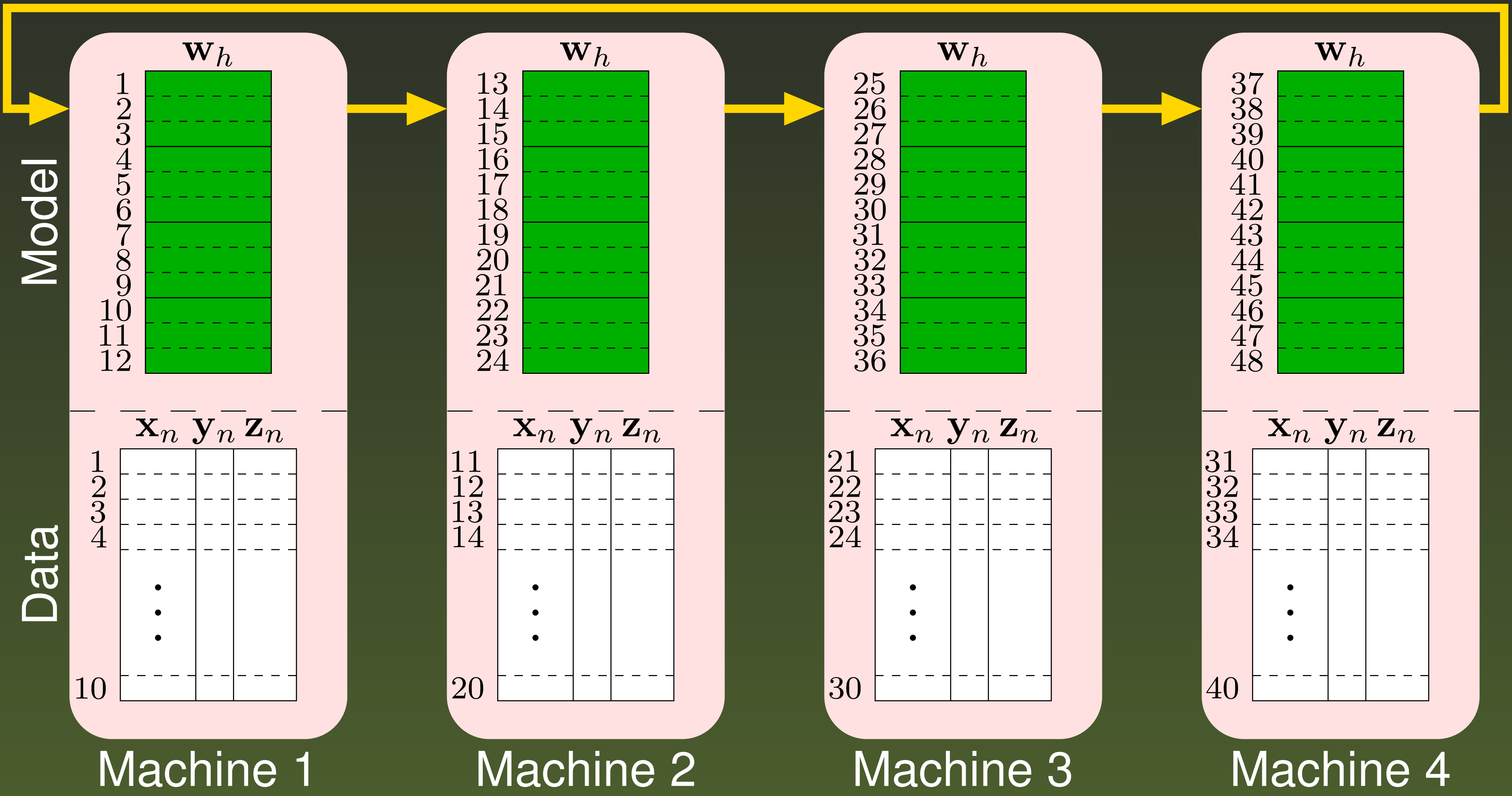
Communication-only (final) epoch, tick 1

ParMAC: the W step (cont.)



Communication-only (final) epoch, tick 2

ParMAC: the W step (cont.)



Communication-only (final) epoch, tick 3 $= P - 1$

ParMAC: the $\mathbf{W}$ step, circular topology (cont.)

Since each submodel is updated as soon as it visits a machine, rather than computing the exact gradient once it has visited all machines and then take a step, the $\mathbf{W}$ step is really carrying out stochastic steps for each submodel in parallel.

- With e epochs, the entire model parameters are communicated $e+1$ times within the $\mathbf{W}$ step.

Practically, two rounds of communication are likely sufficient:

- Have a submodel do e consecutive passes within each machine's data, then communicate it.

  This achieves two rounds of communication with less shuffling, but likely small impact on convergence rate.
- With large datasets we probably need few epochs anyway, particularly with shallow models (logistic regression, linear SVM).

  Even less than one epoch, i.e., a model need not see the whole data to achieve a good enough error.
- Epochs accumulate over MAC iterations (which repeat the $\mathbf{W}$ step).

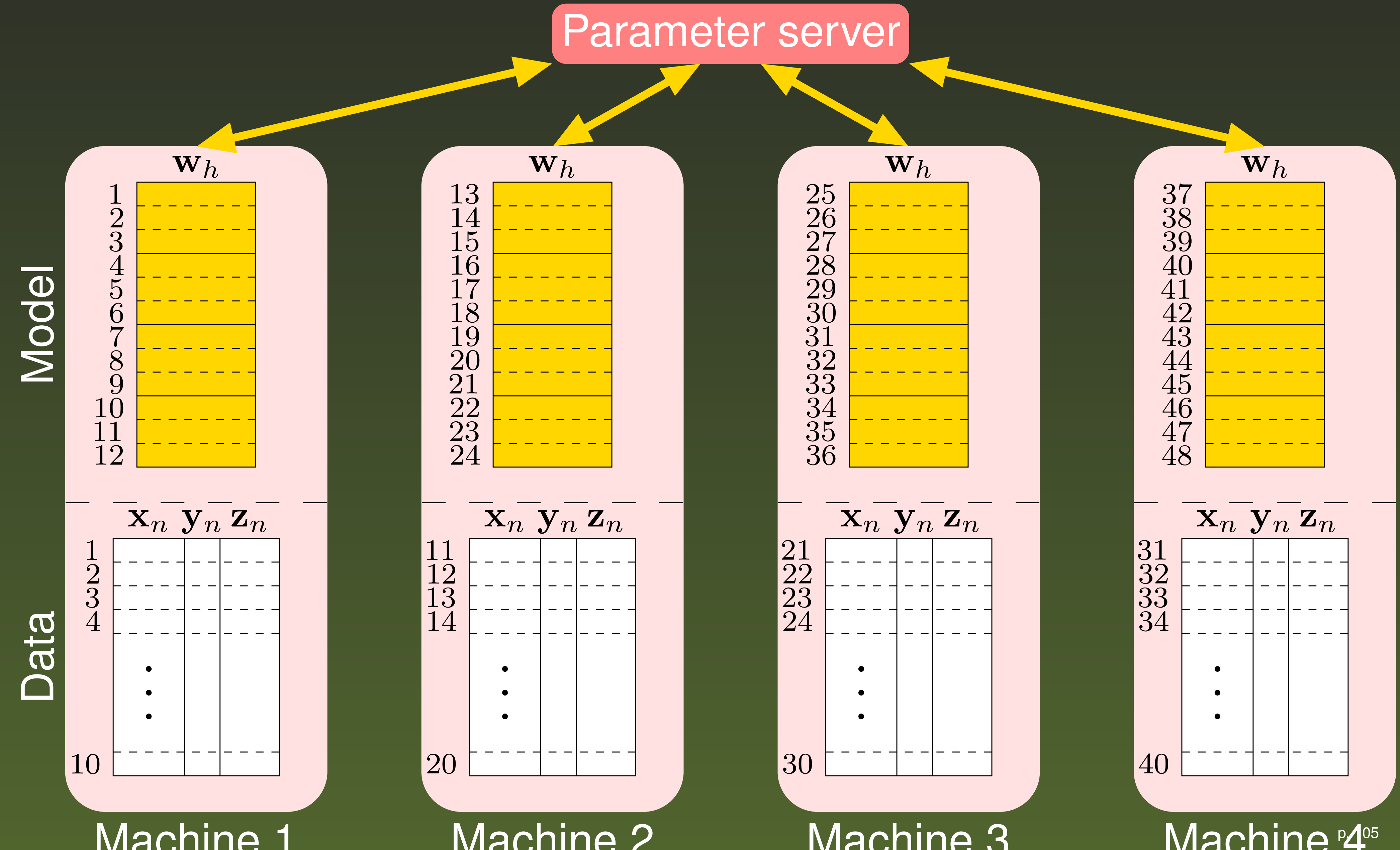
ParMAC, circular topology: extensions

- Data shuffling:
  - Visit data points within each machine at random.
  - Can also randomise the circular topology at each epoch.
- Load balancing:
  - $\mathbf{W}$, $\mathbf{Z}$ steps: work proportional to the number of data points N.
  - If the processing power of machine p is proportional to $\alpha_p > 0$, allocate to it $N\alpha_p/(\alpha_1 + \cdots + \alpha_P)$ data points.
- Streaming: new data added or old data discarded:
  - within a machine, by adding or removing data points to it
  - by adding or removing a machine to the topology

    needs support from the parallel processing library.
- Fault tolerance: similar to removing a machine, but unintended:
  - discard the machine, reconnect circular topology
  - if in $\mathbf{W}$ step, revert submodels lost in the faulty machine to its predecessor's copy of those submodels

ParMAC: the W step, parameter server topology

We do parallel SGD on each submodel independently.



Parallel SGD with a parameter server

- P workers each with N/P points.
- Each worker runs SGD on its own model for a while.
- Workers send their model to the server regularly. The server averages the models (in parameter space) and broadcasts them back to the workers. This can be done synchronously or asynchronously.

Convergence with convex problems:

- For a small enough step size η and some technical conditions.
- Parallelisation stagnates past a certain P.

  Zinkevich et al 2010: expected distance to minimum $\leq \frac{a\eta}{\sqrt{P}} + \frac{b\eta}{P} + c\eta \xrightarrow[P\to\infty]{} c$.
- In practice, the speedups over serial SGD are generally modest.

Convergence with nonconvex problems:

- It does not converge in general. Averaging can fail with local optima.
- With some care, it can work in practice, but this is not easy.

  Average models that are close in parameter space and thus associated with the same optimum.

ParMAC: convergence

MAC theorem (assuming differentiable functions, for quadratic penalty):

> *Theorem 2*: given a positive increasing sequence $(\mu_k) \to \infty$, a nonnegative sequence $(\tau_k) \to 0$, and a starting point $(\mathbf{W}^0, \mathbf{Z}^0)$, suppose the quadratic-penalty method finds an approximate minimizer $(\mathbf{W}^k, \mathbf{Z}^k)$ of $E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k)$ that satisfies $\left\| \nabla_{\mathbf{W},\mathbf{Z}} E_Q(\mathbf{W}^k, \mathbf{Z}^k; \mu_k) \right\| \leq \tau_k$ for $k = 1, 2, \dots$ Then, $\lim_{k\to\infty} (\mathbf{W}^k, \mathbf{Z}^k) = (\mathbf{W}^*, \mathbf{Z}^*)$, which is a KKT point for the nested problem, and its Lagrange multiplier vector has elements $\boldsymbol{\lambda}_n^* = \lim_{k\to\infty} -\mu_k \left(\mathbf{Z}_n^k - \mathbf{F}(\mathbf{Z}_n^k, \mathbf{W}^k; \mathbf{x}_n)\right),\ n = 1, \dots, N$.

The key requirement is to make the gradient of the MAC-penalised function smaller than a set tolerance τ_k, i.e., stay close enough to the path $(\mathbf{W}^*(\mu), \mathbf{Z}^*(\mu))$ that converges (as $\mu \to \infty$) to a local solution.

In MAC, this is achieved by iterating sufficiently the $\mathbf{W}$ and $\mathbf{Z}$ step optimisations with suitable algorithms.

How can this be achieved in ParMAC's $\mathbf{W}$ step?

ParMAC: convergence (cont.)

Circular topology:

- ❖ Under standard conditions for SGD, even if nonconvex submodels.
  Robbins-Monro schedule on the learning rate.
- ❖ Tighter conditions when the subproblems in the $\mathbf{W}$ step are convex.
  Bound the distance to the minimum in objective function value.

$\Rightarrow$ Convergence of ParMAC to a local stationary point guaranteed by the same theorem as MAC, *for convex and nonconvex submodels*, with an added SGD-type condition for the $\mathbf{W}$ step.

Parameter-server topology:

- ❖ Convex $\mathbf{W}$ step: guaranteed by using a parallel SGD condition.
- ❖ Nonconvex $\mathbf{W}$ step: no guarantees.

$\Rightarrow$ Convergence of ParMAC to a local stationary point guaranteed by the same theorem as MAC, *but only for convex submodels*, with an added parallel-SGD-type condition for the $\mathbf{W}$ step.

Distributed EM / k-means

ParMAC (with circular or parameter-server topology) applies just as well to the EM algorithm. Ex: for a mixture of M Gaussians:

- E step: set the posterior probabilites for each point, $\mathbf{p}_1, \dots, \mathbf{p}_N$. Like the $\mathbf{Z}$ step (closed-form solution).
- M step: set the M proportions, means & covariances (submodels) independently from each other as an average over the data. Like the $\mathbf{W}$ step (closed-form form solution in terms of sufficient statistics).

Since the M step is closed-form:

- A single epoch suffices to complete the M step.
- The distributed EM behaves identically to the serial EM, and has the same convergence guarantees.

Circular vs parameter-server topology in the $\mathbf{W}$ step

Run time $\begin{cases} \text{Circular:} & \left(t_r^{\mathbf{W}} \frac{N}{P} + 2t_c^{\mathbf{W}}\right) \lceil \frac{M}{P} \rceil P \\ \text{Parameter-server:} & \left(t_r^{\mathbf{W}} \frac{N}{P} + 2t_c^{\mathbf{W}} \frac{P}{C}\right) M \end{cases}$ where:

- $t_r^{\mathbf{W}}$: computation time per submodel and data point
- $t_c^{\mathbf{W}}$: machine-machine communication time per submodel
- $C \in [1, P]$: # workers the parameter server can communicate with in parallel. System-dependent; can also use several parameter servers.

So the computation time (minibatch updates within each machine) is the same, but the parameter-server topology has more communication, and some additional disadvantages:

- parallel SGD converges more slowly than true SGD
- difficult to apply if the $\mathbf{W}$ step is nonconvex
- needs an extra machine to act as parameter server.

Both topologies differ in how they employ the available parallelism:

- circular: update different, independent submodels
- parameter-server: update replicas of the same submodels

Theoretical model of the parallel speedup

Useful to estimate the optimal number of machines P to use with a given problem, or to explore the effect on the speedup of different parameter settings (e.g. the number of submodels M).

- Runtime $T(P)$ using P machines.
- Speedup $S(P) = T(1)/T(P) =$ (serial runtime) / (parallel runtime).

Consider a ParMAC algorithm:

- on a dataset with N training points, distributed over P identical machines (each with N/P points)
- with M independent submodels of the same size in the $\mathbf{W}$ step
- using a circular topology in the $\mathbf{W}$ step

  parameter server: same model if using $t_c^{\mathbf{W}} \frac{P}{C}$
- operating synchronously.

We ignore small overheads (setup and termination) and estimate the total runtime as proportional to the number of iterations.

Theoretical model of the parallel speedup (cont.)

Parameters of the theoretical model of the speedup:

- $P \geq 1$: number of machines.
- $N \geq 1$: number of training points. Assume $N > P$ is divisible by P.
- $M \geq 1$: number of submodels in the $\mathbf{W}$ step. May be $<$, $=$, $>$ P.
- $e \geq 1$: number of epochs in the $\mathbf{W}$ step.
- $t_r^{\mathbf{W}} > 0$: computation time per submodel & data point in the $\mathbf{W}$ step.

  Time to process (within the current epoch) one data point by a submodel, i.e., the time do an SGD update to a weight vector, per data point.
- $t_c^{\mathbf{W}} > 0$: communication time per submodel in the $\mathbf{W}$ step.

  Time to send one submodel from one machine to another, including overheads (buffering, partitioning into messages or waiting time). We assume communication does not overlap with computation.
- $t_r^{\mathbf{Z}} > 0$: computation time per data point in the $\mathbf{Z}$ step.

  Time to finish one data point entirely, using whatever optimisation algorithm performs the $\mathbf{Z}$ step.

We assume $t_r^{\mathbf{W}}$, $t_c^{\mathbf{W}}$ and $t_r^{\mathbf{Z}}$ are constant and equal for every submodel and data point. Not true in reality…

Theoretical model of the parallel speedup (cont.)

Runtime in each step:

- $\mathbf{Z}$ step: $T^{\mathbf{Z}}(P) = M\frac{N}{P}t_r^{\mathbf{Z}}$.

  The time for any one machine to process its N/P points on all M submodels.

- $\mathbf{W}$ step: $T^{\mathbf{W}}(P) = \lceil M/P \rceil \left(t_r^{\mathbf{W}} \frac{N}{P} + t_c^{\mathbf{W}} \right) Pe + \lceil M/P \rceil t_c^{\mathbf{W}} P$.

  Make M divisible by P by adding fictitious submodels that do work but are discarded at the end.
  Runtime in each tick (there are Pe ticks):

  - $\lceil M/P \rceil \frac{N}{P} t_r^{\mathbf{W}}$: time for any machine to process its $\frac{N}{P}$ points on its portion $\lceil \frac{M}{P} \rceil$ of submodels.
  - $\lceil M/P \rceil t_c^{\mathbf{W}}$: time for any machine to send its portion of submodels.

  $\lceil M/P \rceil t_c^{\mathbf{W}} P$: time of the final round of communication.

Total runtime $T^{\mathbf{W}}(P) + T^{\mathbf{Z}}(P)$ per ParMAC iteration with P machines:

$$T(P) = M\frac{N}{P}t_r^{\mathbf{Z}} + P\lceil M/P \rceil \left(e \left(t_r^{\mathbf{W}} \frac{N}{P} + t_c^{\mathbf{W}} \right) + t_c^{\mathbf{W}} \right), \quad P > 1$$

$$T(1) = MNt_r^{\mathbf{Z}} + MNet_r^{\mathbf{W}}.$$

Theoretical model of the parallel speedup (cont.)

Parallel speedup:

$$S(P) = \frac{\rho \frac{1}{\lceil M/P \rceil} MP}{\frac{1}{N} P^2 + \rho_2 P + \rho_1 \frac{1}{\lceil M/P \rceil} M}$$

by defining the following ratios of computation vs communication:

$$\rho_1 = \frac{t_r^{\mathbf{Z}}}{(e+1)t_c^{\mathbf{W}}} \qquad \rho_2 = \frac{e t_r^{\mathbf{W}}}{(e+1)t_c^{\mathbf{W}}} \qquad \rho = \rho_1 + \rho_2 = \frac{e t_r^{\mathbf{W}} + t_r^{\mathbf{Z}}}{(e+1)t_c^{\mathbf{W}}}$$

- independent of training set size, # submodels, # machines
- depend on the actual computation in the $\mathbf{W}$ and $\mathbf{Z}$ step (optimisation), and on the performance of the distributed system (communication)

  computation power of each machine, communication speed over the network or shared memory, efficiency of the parallel processing library that handles the communication between machines.
- In practice $\rho, \rho_1, \rho_2 \ll 1$

  The communication time dominates the computation time in current architectures.
  The actual values of ρ, ρ_1, ρ_2 can vary considerably depending on the problem.

Theoretical model of the parallel speedup (cont.)

Analysis of the speedup model: various situations possible depending on the parameters. The most practical one is the

large dataset case: $P \ll \rho_2 N, \;\; M < \rho_1 N$.

Then the speedup simplifies to:

$$S(P) \approx \begin{cases} P, & P \leq M \text{ (perfect speedup)} \\ \rho / \left(\frac{\rho_1}{P} + \frac{\rho_2}{M} \right), & P > M \text{ (weighted harmonic mean)} \end{cases}$$

and the maximum speedup S^* (over P) is achieved for $P = P^* > M$:

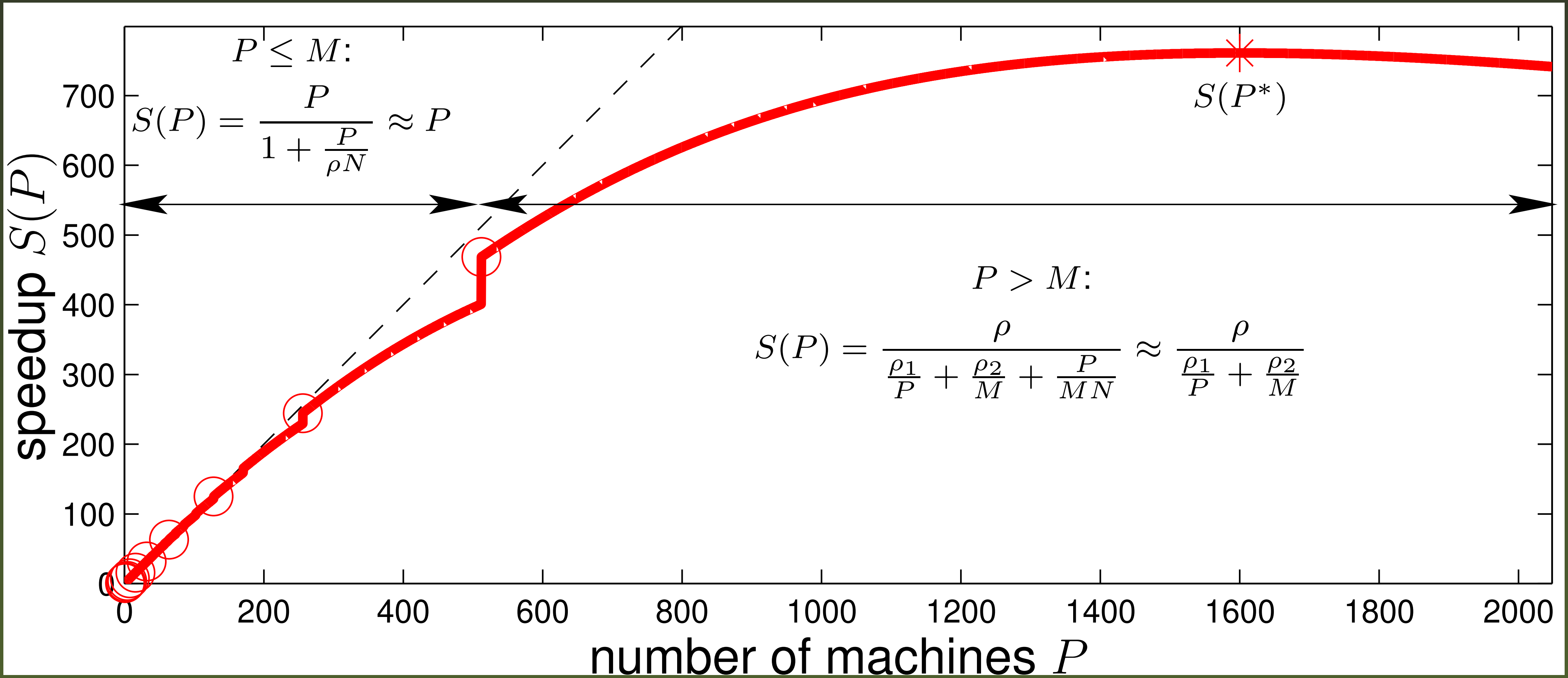
$$S^* = \frac{\rho M}{\rho_2 + 2\sqrt{\rho_1 M/N}} > M \quad \text{achieved at} \quad P^* = \sqrt{\rho_1 M N} > M.$$

So we expect very high parallel speedups with large datasets if $M \lesssim P$.

Theoretical model of the parallel speedup (cont.)

Typical theoretical speedup curve for realistic parameter settings.

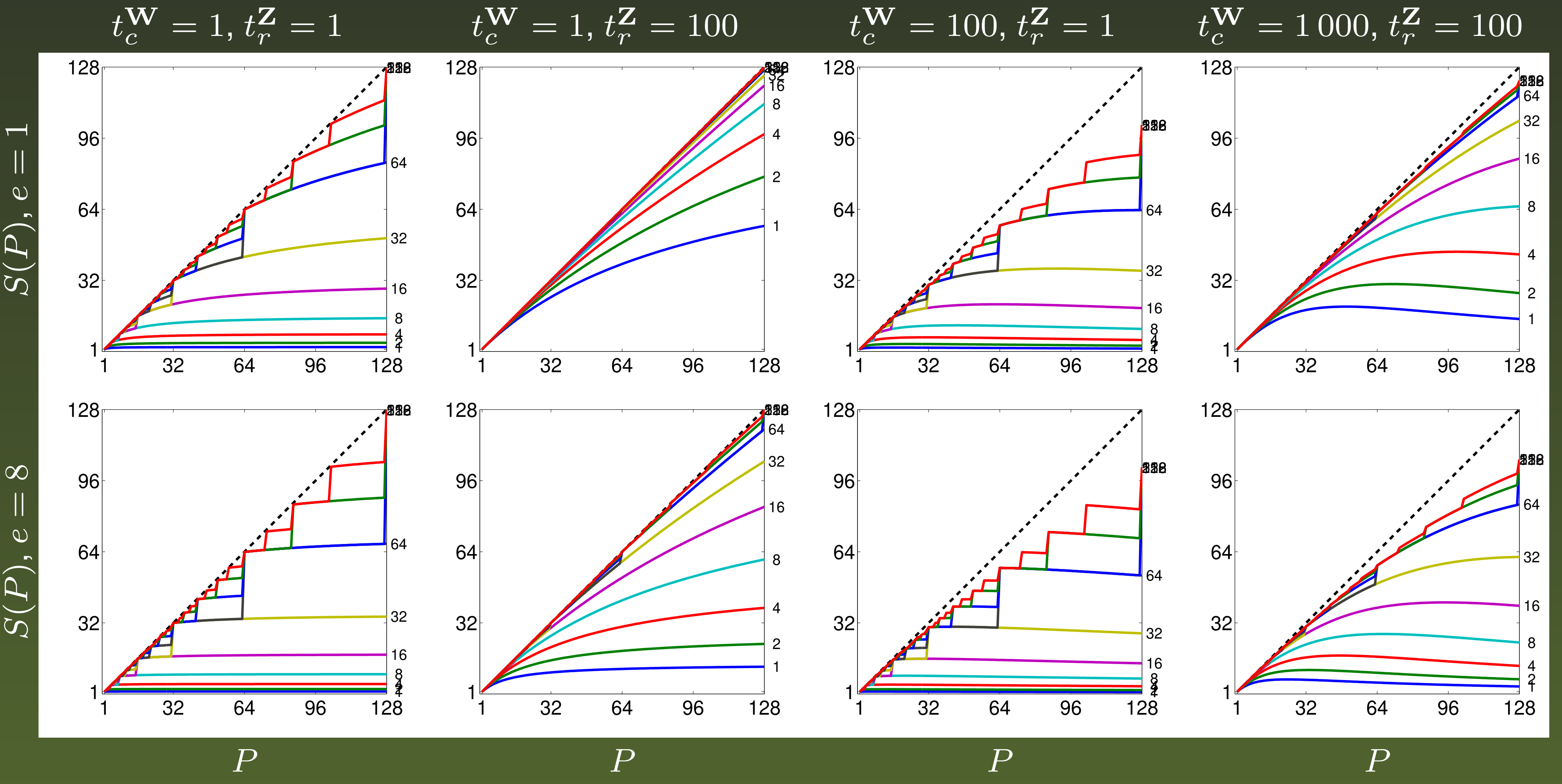
$N = 1$M data points, $M = 512$ submodels, $e = 1$ epoch in the $\mathbf{W}$ step, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} = 5$, $t_c^{\mathbf{W}} = 10^3$ (so $\rho_1 = 0.0025$, $\rho_2 = 0.0005$ and $\rho = 0.003$)



Theoretical model of the parallel speedup (cont.)

Theoretical speedup curves for other parameter settings.

$N = 50\text{K}$, $M \in \{2^0, \ldots, 2^9\}$, $e \in \{1, 8\}$, $t_r^{\mathbf{W}} = 1$, $t_r^{\mathbf{Z}} \in \{1, 100\}$, $t_c^{\mathbf{W}} \in \{1, 100, 1\,000\}$



Theoretical model of the parallel speedup (cont.)

Practical considerations:

- The speedup is unchanged by trading off dataset size (N) and computation/communication times ($t_r^{\mathbf{W}}$, $t_r^{\mathbf{Z}}$, $t_c^{\mathbf{W}}$) in various ways.

  Scaling by $\alpha > 0$:

$N, t_r^{\mathbf{W}}, t_r^{\mathbf{Z}}$	$\rightarrow$	$\alpha N, \frac{1}{\alpha} t_r^{\mathbf{W}}, \frac{1}{\alpha} t_r^{\mathbf{Z}}$	*larger dataset, faster computation*
$N, t_c^{\mathbf{W}}$	$\rightarrow$	$\alpha N, \alpha t_c^{\mathbf{W}}$	*larger dataset, slower communication*
$t_r^{\mathbf{W}}, t_r^{\mathbf{Z}}, t_c^{\mathbf{W}}$	$\rightarrow$	$\alpha t_r^{\mathbf{W}}, \alpha t_r^{\mathbf{Z}}, \alpha t_c^{\mathbf{W}}$	*faster computation, faster communication.*

- Pick M divisible by P if $P < M$ (more efficient parallelism, no machines ever idle).
- Machines of different power: data load proportional to power.
- Models of different sizes: group models to equalise them.

  Binary autoencoder with L encoders and D decoders: group the D decoders into L groups of D/L decoders each $\rightarrow$ total $M = 2L$ same-size submodels.
- Other considerations: cost of machines, etc.

Parameter-server topology: replace $t_c^{\mathbf{W}}$ with $t_c^{\mathbf{W}} \frac{P}{C}$ in $S(P)$.

ParMAC: implementation for binary autoencoders

- We implemented the circular topology.
- C/C++ using the GSL and BLAS libraries for numerical operations.
- Message Passing Interface (MPI) for interprocess communication:
  - Different processes cannot directly access each other's memory space. Data is transferred by sending messages from one process to another, or collectively among multiple processes.
  - Widely used for high-performance parallel computing.
  - Several implementations available, such as MPICH or OpenMPI.
- Basic code structure:
  - Initialise/finalise MPI environment at start/end of code.
  - To receive data: synchronous blocking `MPI_Recv`.

    Process calling `MPI_Recv` blocks until data arrives.
  - To send data: buffered blocking `MPI_Bsend`.

    Allocate memory and attach it to the system. Process calling `MPI_Bsend` blocks until the buffer is copied to the MPI internal memory. After that, the MPI library takes care of sending the data.

ParMAC: implementation for binary autoencoders (cont.

```
MPI_Init(&argc, &argv);       // init. MPI execution environment
MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
loadsettings();                   // $\mu$, epochs, dataset path, etc.
loaddatasets();       // datasets and initial auxiliary coordinates
initializelayers();                    // initialise $\mathbf{f}$, $\mathbf{h}$ and $\mathbf{Z}$ steps
// allocate big enough buffer for MPI_Bsend
MPI_Pack_size(commbuffsize, MPI_CHAR, MPI_COMM_WORLD,
  &mpi_attach_buff_size);
mpi_attach_buff = malloc(totalsubmodelcount*
  (mpi_attach_buff_size+MPI_BSEND_OVERHEAD));
MPI_Buffer_attach(mpi_attach_buff, mpi_attach_buff_size);

for (iter=1 to length($\mu$)) {  // ————————————————————

  // begin $\mathbf{W}$-step
  visitedsubmodels = 0;
  // each process visits all the submodels, epochs + 1 times
  while (visitedsubmodels <= totalsubmodelcount*epochs) {
    // stepcounter indicates how far trained each submodel is
    if (stepcounter > 0) {  // not 1st submodel? wait to receive
      // MPI_Recv blocks until requested data is available
      MPI_Recv(receivebuffer, commbuffsize,
        MPI_CHAR, MPI_ANY_SOURCE, MODEL_MSG_TAG,
        MPI_COMM_WORLD, &recvStatus);
      savesubmodel(receivebuffer);
    }
    if (stepcounter < epochs*mpisize) {    // not in last round
      switch(submodeltype)    // train submodel according to type
      case 'SVM': HtrainSGD();
      case 'linlayer':  FtrainSGD();
    }
    if (stepcounter < (ringepochs+1)*mpisize) {
      // pick the successor process from the lookup table
      successor = next_in_lookuptable();
      loadsubmodel(sendbuffer);
      MPI_Bsend(sendbuffer, taskbufsize*sizeof(double),
        MPI_CHAR, successor, MODEL_MSG_TAG, MPI_COMM_WORLD);
    }
    visitedsubmodels++;
  }
  // end $\mathbf{W}$-step

  // begin $\mathbf{Z}$-step
  updateZ();                         // optimise auxiliary coordinates
  // end $\mathbf{Z}$-step

}    // ————————————————————————————

// detach the allocated buffer
MPI_Buffer_detach(&mpi_attach_buff, &mpi_attach_buff_size);
free(mpi_attach_buff);                // free the allocated memory
MPI_Finalize();             // terminate MPI execution environment
```

Experiments

ParMAC with circular topology, e communication epochs (no within-machine epochs).

Computing systems:

- Distributed-memory: UCSD Triton Shared Computing Cluster (TSCC). Each node contains 2 8-core Intel Xeon E5-2670 processors (16 cores in total), 64GB DRAM (4GB/core) and a 500GB hard drive. Nodes connected through a 10GbE network. We used up to $P = 128$ processors.
- Shared-memory: 72-processor machine with 256GB RAM at UC Merced. Processors communicate through shared memory. We used up to $P = 64$ proc.

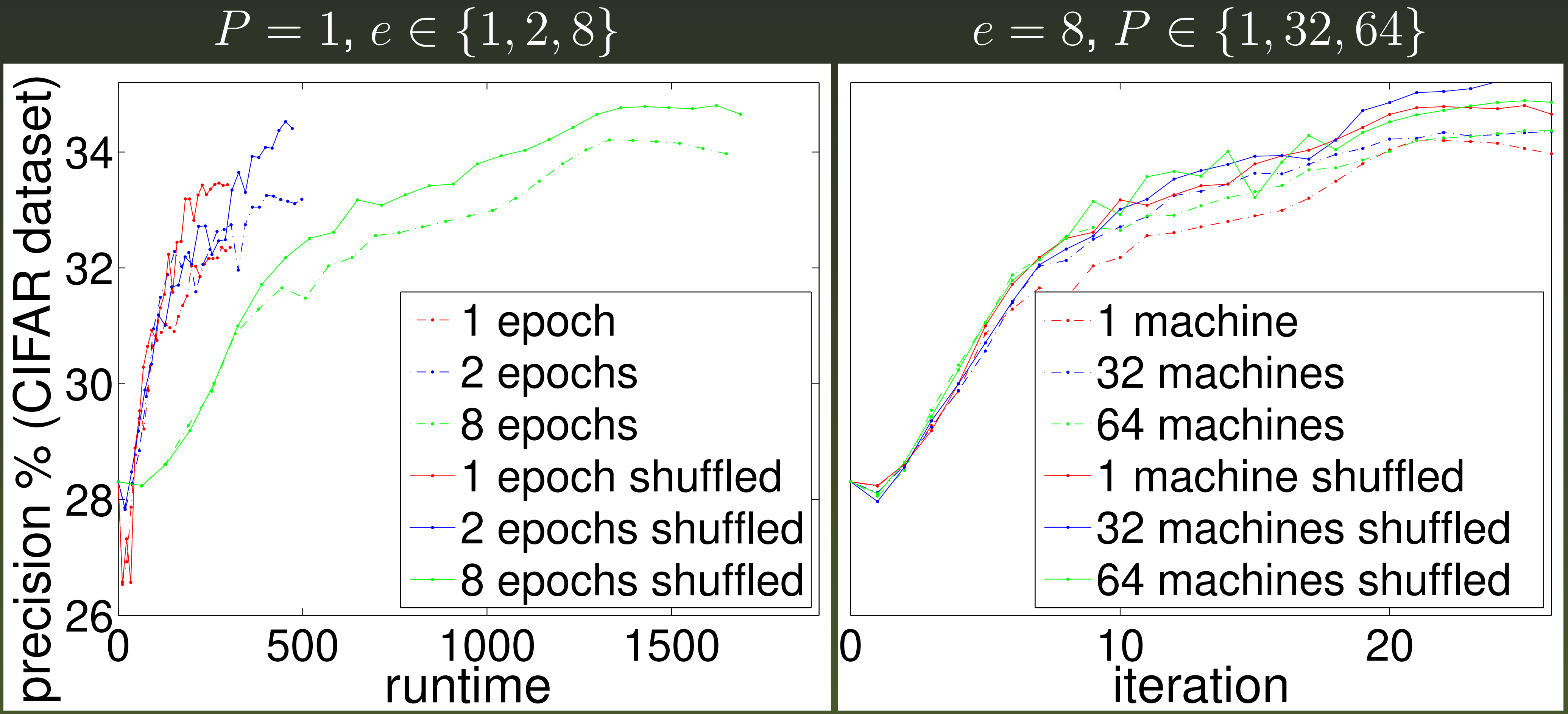
Image retrieval datasets:

- CIFAR: $D = 320$ GIST features; $N =$ 50k.
- SIFT: $D = 128$ SIFT features; $N =$ 10k, 1M (SIFT-1M), 100M (SIFT-1B).
  SIFT-1B: largest(?) public benchmark for nearest-neighbour search with known ground-truth.

Binary autoencoder:

- Encoder: linear/RBF network (L SVMs), $L = \begin{cases} 16 \text{ (CIFAR, SIFT-1M)} \\ 64 \text{ (SIFT-1B).} \end{cases}$
  Decoder: linear (D linear mappings).
- $\mathbf{W}$ step: SGD code from `http://leon.bottou.org/projects/sgd`.
- $\mathbf{Z}$ step: alternating optimisation over the bits of $\mathbf{z}_n \in \{0,1\}^L$.

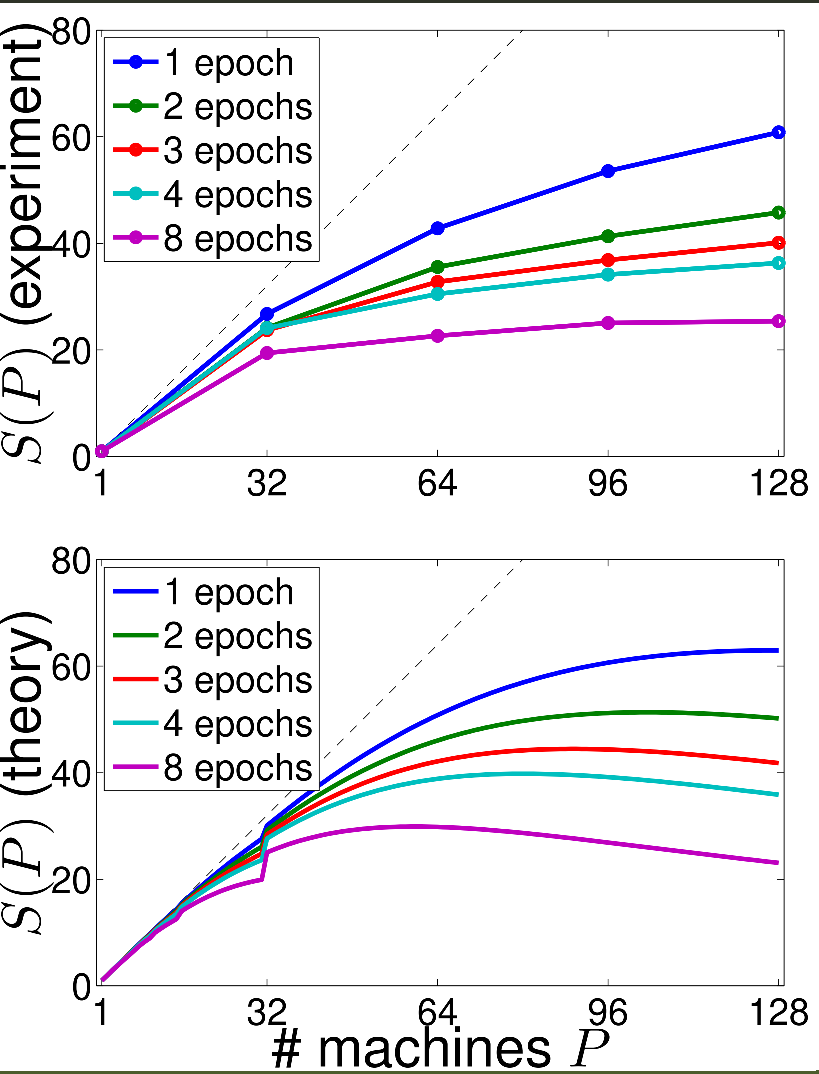
Experiments: effect of stochastic steps in the $\mathbf{W}$ step

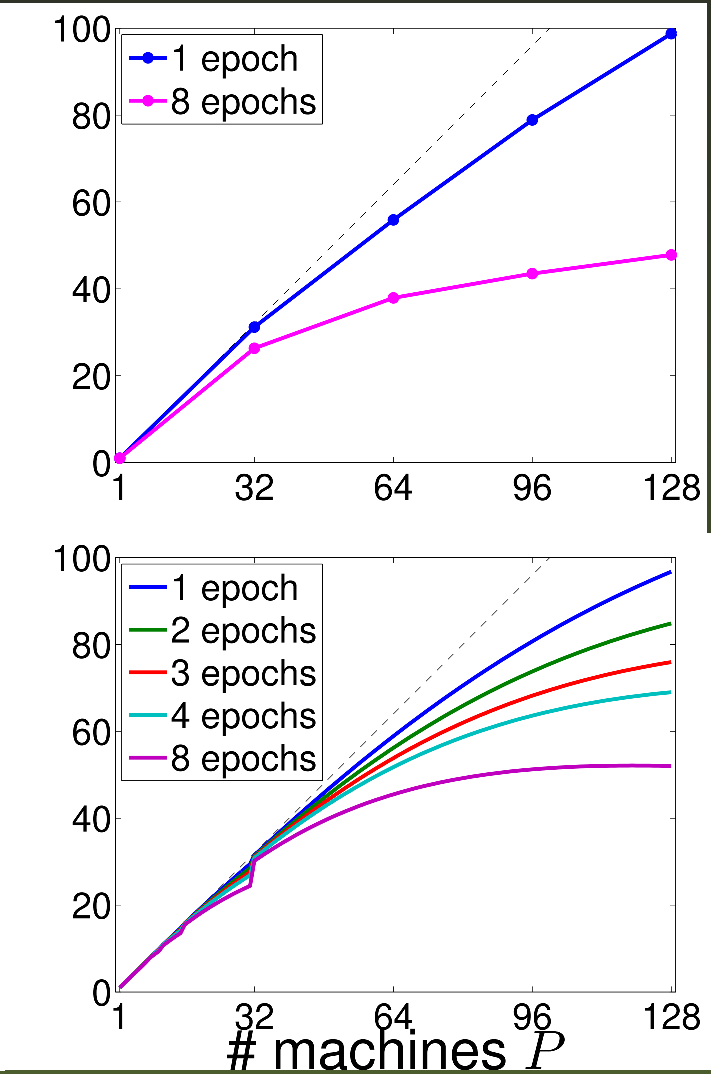


- More epochs $\rightarrow$ more exact $\mathbf{W}$ step.
  Little degradation with $e = 1$ epoch even though the dataset is small.
- Shuffling generally reduces the error and increases the precision with no increase in runtime.

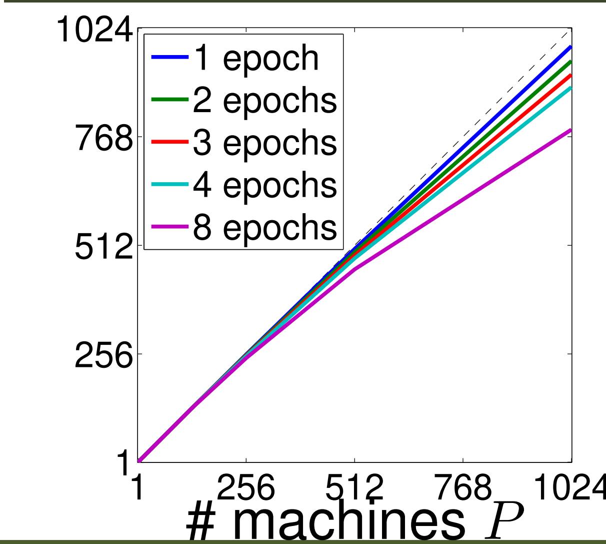
Experiments: speedup $S(P)$

CIFAR $N = 50\text{K}, M = 32$ SIFT-1M $N = 10\text{M}, M = 32$ SIFT-1B $N=100\text{M}, M=128$





too long to run
in a single machine
(would take months)



Runtime on $P = 128$ machines: SIFT-1M ($e = 1$): 12', speedup 100×
SIFT-1B ($e = 2$): 29h, speedup 128× (estimated).

Experiments: speedup $S(P)$ (cont.)

As a function of the # machines P for fixed problem size (dataset and model) (strong scaling), in the distributed memory system. In general:

- Theoretical speedups match experimental ones reasonably well.
  For BAs: $M = 2L$ effective submodels of the same size.
- The speedup is nearly perfect for $P \leq M$ and holds very well for $P > M$ up to the maximum # machines we used ($P = 128$ in the distributed system). Eventually it does decrease, of course.

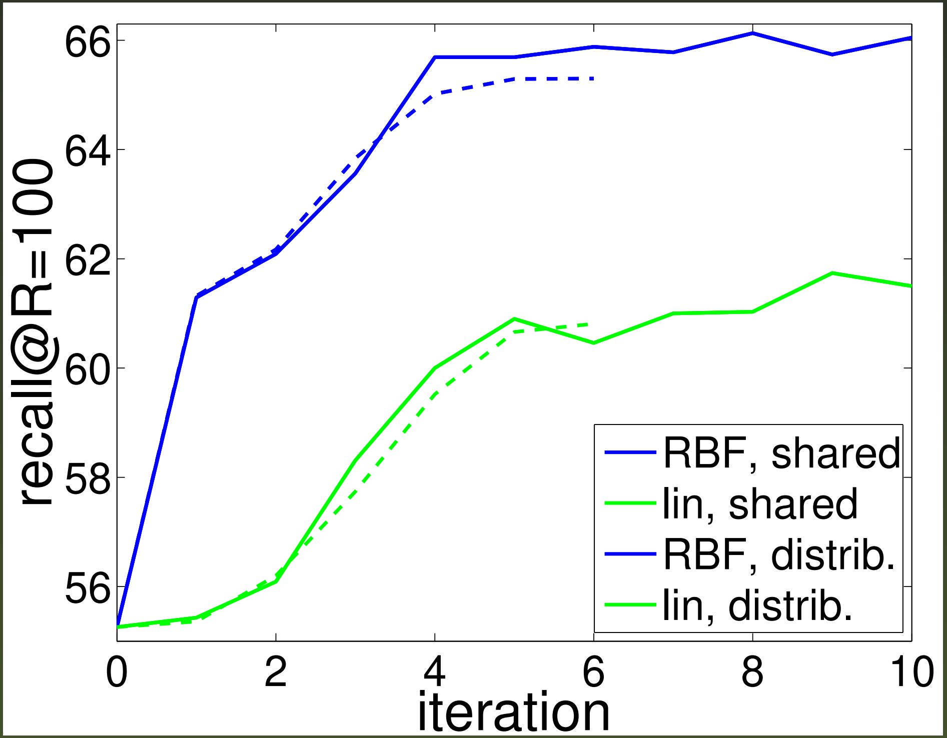
Specifically for each dataset:

- CIFAR, SIFT-1M: the speedups flatten as the # epochs e (hence communication) increases, because for this experiment the bottleneck is the $\mathbf{W}$ step, whose parallelisation ability (# concurrent processes) is limited by M, which is small.
  Note small N and small M for CIFAR, so worse speedup than for SIFT-1M.
- SIFT-1B theoretical speedup nearly perfect in whole range of P.

Experiments: large-scale dataset, SIFT-1B

Distributed system: $P = 128$ processors.
Shared-memory system: $P = 64$ proc.

- Single-machine runtime: months.
- $N = 100$M training points, $D = 128$ (linear SVMs) or $2\,000$ (kernel SVMs), $L = 64$ bits (# hash functions).
- $e = 2$ epochs with shuffling.
- # MAC iterations: shared-memory 10, distributed 6.
- Learning curves essentially identical over distributed & shared-memory.
  The total runtime in the shared-memory system is shorter because it has faster processors and faster interprocessor communication.
- Speedup predicted to be nearly perfect for $P = 128$.



Hash function (encoder)	Recall @R=100	Time (hours)	
		distrib.	shared
linear SVM	61.5%	29.30	11.04
kernel SVM	66.1%	83.44	32.19

Conclusion: ParMAC

A distributed model for MAC for training nested, nonconvex models:

- MAC creates parallelism by introducing auxiliary coordinates for each data point to decouple nested terms in the objective function.
- ParMAC translates this parallelism to a distributed system by using:
  1. Data parallelism: each machine keeps a portion of the original data and its corresponding auxiliary coordinates.
  2. Model parallelism: independent submodels visit every machine in a circular topology, effectively doing SGD.
- Parallel speedup with large datasets:
  - nearly perfect when # submodels $\gtrsim$ # machines
  - eventually saturates as we continue to increase the # machines.
- Data shuffling, load balancing, streaming & fault tolerance.

References and code http://faculty.ucmerced.edu/mcarreira-perpinan/MAC

Original reference for MAC:

- Miguel Carreira-Perpiñán and Weiran Wang: *Distributed optimization of deeply nested systems*. AISTATS 2014, arXiv:1212.5921.

Extensions or related work:

- C-P & Alizadeh: *ParMAC: distributed optimisation of nested functions, with application to learning binary autoencoders*. SysML 2019, arXiv:1605.09114.
- C-P & Raziperchikolaei: *Optimizing affinity-based binary hashing using auxiliary coordinates*. NIPS 2016, arXiv:1501.05352.
- C-P & Vladymyrov: *A fast, universal algorithm to learn parametric nonlinear embeddings*. NIPS 2015.
- C-P & Raziperchikolaei: *Hashing with binary autoencoders*. CVPR 2015, arXiv:1501.00756.
- Wang and C-P: *The role of dimensionality reduction in classification*. AAAI 2014, arXiv:1405.6444.
- Wang and C-P: *Nonlinear low-dimensional regression using auxiliary coordinates*. AISTATS 2012.
- C-P and Lu: *Manifold learning and missing data recovery through unsupervised regression*. ICDM 2011.
- C-P and Lu: *Parametric dimensionality reduction by unsupervised regression*. CVPR 2010.
- C-P and Lu: *Dimensionality reduction by unsupervised regression*. CVPR 2008.

Work partially supported by NSF awards CAREER IIS–0754089, IIS–1423515 and a Google Faculty Research Award.