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”
   ✦ Distributed optimisation with MAC: ParMAC
❖ A gallery of nested models trainable with MAC
  1. Learning low-dimensional features for classification
     low-dimensional SVM, low-dimensional logistic regression
  2. Parametric nonlinear embeddings
  3. Binary hashing for fast image search
     binary autoencoder, affinity-based objective function
Nested (hierarchical) systems: examples

Common in computer vision, speech processing, machine learning...

❖ Object recognition pipeline:
  image pixels $\rightarrow$ SIFT/HoG $\rightarrow$ \( k \)-means sparse coding $\rightarrow$ 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: \( x \rightarrow \{ \sigma(w^T_1 x + a_i) \} \rightarrow \{ \sigma(w^T_j \{ \sigma(w^T_i x + a_i) \}) + b_j \} \rightarrow \cdots \rightarrow y \)
Nested systems

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

\[
f(x; W) = f_{K+1}(\ldots f_2(f_1(x; W_1); W_2)\ldots; W_{K+1})
\]

❖ Each layer (processing stage) has its own trainable parameters (weights) \(W_k\).

❖ Each layer performs some (nonlinear, nondifferentiable) processing on its input, extracting ever more sophisticated features from it (ex.: pixels → edges → parts → ⋯)

❖ Often inspired by biological brain processing (e.g. retina → LGN → V1 → ⋯)

❖ 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: $f(x) = \sum_{m=1}^{M} w_m \phi_m(x)$:
    - RBF network: fix nonlinear basis functions $\phi_m$ (e.g. $k$-means), then fix linear weights $w_m$.
    - 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.
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.: \( \frac{\partial}{\partial g} (g(F(\cdot))) = g'(F(\cdot)) \), \( \frac{\partial}{\partial F} (g(F(\cdot))) = g'(F(\cdot)) F'(\cdot) \).

  Then feed to nonlinear optimiser.

  Gradient descent, CG, L-BFGS, Levenberg-Marquardt, Newton, etc.

- **Major breakthrough in the 80s with neural nets.**
  It allowed to train multilayer perceptrons 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.
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 minimum 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
“Filter” vs “wrapper” approaches: consider a nested mapping \( g(F(x)) \) (e.g. \( F \) reduces dimension, \( g \) classifies). How to train \( F \) and \( g \)?

**Filter approach:**

- **Greedy sequential training:**
  1. **Train** \( F \) (the “filter”):
     - **Unsupervised:** use only the input data \( \{x_n\} \)
       - PCA, \( k \)-means, etc.
     - **Supervised:** use the input and output data \( \{(x_n, y_n)\} \)
       - LDA, sliced inverse regression, etc.
  2. **Fix** \( F \), **train** \( g \): fit a classifier with inputs \( \{F(x_n)\} \) and labels \( \{y_n\} \).

- **Very popular:** \( F \) 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** \( F \).

- **... But is still suboptimal:** the preprocessing may not be the best possible for classification.
Wrapper approach:

❖ Train $F$ and $g$ 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.
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.
The method of auxiliary coordinates (MAC)


❖ A general strategy to train all parameters of a nested system. Not an algorithm but a meta-algorithm (like EM).
❖ Enjoys the benefits of layerwise training (fast, easy steps that reuse existing algorithms for shallow systems) but with optimality guarantees.
❖ Embarrassingly parallel iterations.
❖ Basic idea:
  1. Turn the nested problem into a constrained optimisation problem by introducing new parameters to be optimised over (the auxiliary coordinates).
  2. Optimise the constrained problem with a penalty method.
  3. Optimise the penalised objective function with alternating optimisation.

Result: alternate “layerwise training” steps with “coordination” steps.
The nested objective function

Consider for simplicity:

❖ a single hidden layer: \( x \rightarrow F(x) \rightarrow g(F(x)) \)

❖ a least-squares regression for inputs \( \{x_n\}_{n=1}^N \) and outputs \( \{y_n\}_{n=1}^N \):

\[
\min E_{\text{nested}}(F, g) = \frac{1}{2} \sum_{n=1}^N \| y_n - g(F(x_n)) \|^2
\]

F, g have their own parameters (weights). We want to find a local minimum of \( E_{\text{nested}} \).
The MAC-constrained problem

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

\[
\min E(F, g, Z) = \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2
\]

s.t. \( z_n = F(x_n) \quad n = 1, \ldots, N. \)

❖ For each data point, we turn the subexpression \( F(x_n) \) into an equality constraint associated with a new parameter \( z_n \) (the auxiliary coordinates).

Thus, a constrained problem with \( N \) equality constraints and new parameters \( Z = (z_1, \ldots, z_N) \).

❖ We optimise over \( (F, g) \) and \( Z \) 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 $\mu \to \infty$:

$$
\min_{E_Q(F, g, Z; \mu)} = \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \|z_n - F(x_n)\|^2
$$

We can also use the augmented Lagrangian method (ADMM) instead:

$$
\min_{E_L(F, g, Z, \Lambda; \mu)} = \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2 + \sum_{n=1}^{N} \lambda_n^T (z_n - F(x_n)) + \frac{\mu}{2} \sum_{n=1}^{N} \|z_n - F(x_n)\|^2
$$

$$
= \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \|z_n - F(x_n) - \frac{1}{\mu} \lambda_n\|^2
$$

which does an extra update of the Lagrange multipliers: $\lambda_n \leftarrow \lambda_n - \mu(z_n - F(x_n))$, $n = 1, \ldots, N$. For simplicity, we focus on the quadratic-penalty method.
Net effect: unfold the nested objective into shallow additive terms connected by the auxiliary coordinates:

\[
E_{\text{nested}}(F, g) = \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(F(x_n))\|^2 \implies \\
E_Q(F, g, Z; \mu) = \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \|z_n - F(x_n)\|^2
\]

All terms equally scaled, but uncoupled.
Vanishing gradients less problematic. Derivatives required are simpler: no backpropagated gradients, sometimes no gradients at all.

Optimising \(E_{\text{nested}}\) follows a convoluted trajectory in \((F, g)\) space.

Optimising \(E_Q\) can take shortcuts by jumping across \(Z\) space. This corresponds to letting the layers mismatch during the optimisation.
Alternating optimisation of the MAC-penalised function

\((F, g)\) step, for \(Z\) fixed:

\[
\begin{align*}
\min_{F} & \quad \frac{1}{2} \sum_{n=1}^{N} \|z_n - F(x_n)\|^2 \\
\min_{g} & \quad \frac{1}{2} \sum_{n=1}^{N} \|y_n - g(z_n)\|^2
\end{align*}
\]

❖ **Layerwise training**: each layer is trained independently (not sequentially):

❖ **fit** \(F\) to \(\{(x_n, z_n)\}_{n=1}^{N}\) (gradient needed: \(F'(\cdot)\))

❖ **fit** \(g\) to \(\{(z_n, y_n)\}_{n=1}^{N}\) (gradient needed: \(g'(\cdot)\))

This looks like a filter approach with intermediate “features” \(\{z_n\}_{n=1}^{N}\).

❖ 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.)

Z step, for \((F, g)\) fixed:

\[
\min_{z_n} \frac{1}{2} \|y_n - g(z_n)\|^2 + \frac{\mu}{2} \|z_n - F(x_n)\|^2 \quad n = 1, \ldots, N
\]

❖ The auxiliary coordinates are trained independently for each point. 

\(N\) small problems (of size \(|z|\)) instead of one large problem (of size \(N|z|\)).

❖ They “coordinate” the layers.

❖ The Z step has the form of a proximal operator:

\[
\min_z f(z) + \frac{\mu}{2} \|z - u\|^2
\]

The solution has a geometric flavour (“projection”).

❖ Often closed-form (depending on the model).
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:

\[
\min E_{\text{nested}}(F, g) = \frac{1}{2} \sum_{n=1}^{N} \| y_n - g(F(x_n)) \|^2
\]

- **Over g for fixed F**: fit g to \( \{(F(x_n), y_n)\}_{n=1}^{N} \) (needs \( g'(\cdot) \))
- **Over F for fixed g**: needs backprop. gradients over \( g'(F(\cdot))F'(\cdot) \)

Pure alternating optimisation \( \neq \) “layerwise training”.
MAC in general ($K$ layers)

The nested objective function:

$$E_{\text{nested}}(W) = \frac{1}{2} \sum_{n=1}^{N} \| y_n - f(x_n; W) \|^2$$

$$f(x; W) = f_{K+1}(\ldots f_2(f_1(x; W_1); W_2) \ldots; W_{K+1})$$

The MAC-constrained problem:

$$E(W, Z) = \frac{1}{2} \sum_{n=1}^{N} \| y_n - f_{K+1}(z_{K,n}; W_{K+1}) \|^2 \quad \text{s.t.} \quad \begin{cases} z_{K,n} = f_K(z_{K-1,n}; W_K) \\ \ldots \\ z_{1,n} = f_1(x_n; W_1) \end{cases} \quad n = 1, \ldots, N.$$ 

The MAC quadratic-penalty function:

$$E_Q(W, Z; \mu) = \frac{1}{2} \sum_{n=1}^{N} \| y_n - f_{K+1}(z_{K,n}; W_{K+1}) \|^2 + \frac{\mu}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \| z_{k,n} - f_k(z_{k-1,n}; W_k) \|^2.$$ 

Alternating optimisation:

- **W step:** \( \min_{W_k} \sum_{n=1}^{N} \| z_{k,n} - f_k(z_{k-1,n}; W_k) \|^2, \quad k = 1, \ldots, K + 1. \)
- **Z step:** \( \min_{z_n} \frac{1}{2} \| y_n - f_{K+1}(z_K,n) \|^2 + \frac{\mu}{2} \sum_{k=1}^{K} \| z_{k,n} - f_k(z_{k-1,n}) \|^2, \quad n = 1, \ldots, N. \)

MAC also applies with various loss functions, full/sparse layer connectivity, constraints, etc.
MAC in general ($K$ layers): convergence guarantees

Assuming 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.

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

That is, MAC defines a continuous path $(W^*(\mu), Z^*(\mu))$ 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.
How to train your system using auxiliary coordinates:

1. Write your nested objective function $E_{\text{nested}}(W)$.
2. Identify subexpressions and turn them into auxiliary coordinates with equality constraints.
3. Apply quadratic penalty or augmented Lagrangian method.
4. Do alternating optimisation of the penalised function:
   - $W$ step: reuse a single-layer training algorithm, typically
   - $Z$ step: needs to be solved specially for your problem
     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.
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. $W$ and $Z$ steps use Gauss-Newton.

\[
x = W_1 \sigma z_1 = W_2 \sigma z_2 = W_3 \sigma z_3 = W_4 \sigma z_4 = y = x
\]

Objective function

<table>
<thead>
<tr>
<th>Runtime (hours)</th>
<th>MAC (• = 1 it.)</th>
<th>Parallel MAC</th>
<th>MAC (minibatches)</th>
<th>Parallel MAC (minibatches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^1</td>
<td>10^2</td>
<td>10^3</td>
<td>10^4</td>
<td>10^5</td>
</tr>
<tr>
<td>10^6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SGD ($\bullet = 20$ epochs)

CG ($\bullet = 100$ its.)

p. 23
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 $Z$. $W$ step uses $k$-means ($C_k$) + linsys ($W_k$). $Z$ step uses Gauss-Newton.

\[ x = y = x \]
Practicalities

Schedule of the penalty parameter $\mu$:

❖ Theory: $\mu \to \infty$ for convergence.
❖ Practice: stop with finite $\mu$.
❖ Keeping $\mu = 1$ gives quite good results.
❖ How fast to increase $\mu$ depends on the problem.
❖ We increase $\mu$ when the error in a validation set increases.

The postprocessing step:

❖ After the algorithm stops, we satisfy the constraints by:
  ✦ Setting $z_{kn} = f_k(z_{k-1,n}; W_k)$, $k = 1, \ldots, K$, $n = 1, \ldots, N$
  That is, project on the feasible set by forward propagation.
  ✦ Keeping all the weights the same except for the last layer, where
    we set $W_{K+1}$ by fitting $f_{K+1}$ to the dataset $(f_K(\ldots (f_1(X))), Y)$. This provably reduces the error.
Choice of optimisation algorithm for the steps:

- **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).

- **Z** step: often closed-form, otherwise:
  - Small number of parameters in $z_n$: Gauss-Newton
    - The GN matrix is always positive definite because of the $\|z - \cdot\|^2$ terms.
  - Large number of parameters in $z_n$: CG, Newton-CG, L-BFGS.

Standard optimisation and linear algebra techniques apply:

- Inexact steps.
- Warm starts.
- Caching factorisations.

Cleverly used, they can make the W and Z steps very fast.
Practicalities (cont.)

Defining the auxiliary coordinates:

❖ With neural nets, we can introduce them
  ✦ after the nonlinearity: \( z = \sigma(w^T x + b) \)
    logistic regression W step, nonlinear Z step
  ✦ before the nonlinearity: \( z = w^T x + b \):
    linear W step, nonlinear Z step
  ✦ before and after the nonlinearity: \( \zeta = \sigma(z) \) and \( z = w^T x + b \)
    linear W step, linear Z step, scalar \( \zeta \) step
  “Closest point on a sigmoid”: \( \arg \min_\zeta (y - \sigma(\zeta))^2 + (\zeta - x)^2 \) for each element of \( \zeta \).

❖ 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 \( Z \) over the optimisation.

The best strategy will depend on the dataset dimensionality and size, and on the model.
In summary, MAC applies very generally:

- $K$ layers: $f(x; W) = f_{K+1}(\ldots f_2(f_1(x; W_1); W_2)\ldots; W_{K+1})$.
- Various loss functions, full/sparse layer connectivity, constraints…

The resulting optimisation algorithm depends on:

- the type of layers $f_k$ used
  - 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 redefine $Z$ during the optimisation

- how we optimise each step
  - choice of method; inexact steps, warm start, caching factorisations, stochastic updates…
Related work: dimensionality reduction

- Given high-dim data $y_1, \ldots, y_N \in \mathbb{R}^D$, we want to project to latent coordinates $z_1, \ldots, z_N \in \mathbb{R}^L$ with $L \ll D$.

- Optimise reconstruction error over the reconstruction mapping $f: z \rightarrow y$ and the latent coordinates $Z$:

$$\min_{f,Z} \sum_{n=1}^{N} \| y_n - f(z_n) \|^2$$

where $f$ 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 $z$s map to nearby $y$s, but not necessarily vice versa. This can produce a poor representation in latent space.

- This can be solved by introducing the “inverse” mapping $F: y \rightarrow z$. 
“Dimensionality reduction by unsupervised regression”
(Carreira-Perpiñán & Lu, 2008, 2010):

\[
\min_{f,F,Z} \sum_{n=1}^{N} \| y_n - f(z_n) \|^2 + \| z_n - F(y_n) \|^2
\]

❖ Learns both mappings: reconstruction \( f \) and projection \( F \), together with the latent coordinates \( Z \).
❖ Now nearby \( y \)'s also map to nearby \( x \)'s
\( f \) and \( F \) become approximate inverses of each other on the data manifold.
❖ Special case of MAC to solve the autoencoder problem:

\[
\min_{f,F} \sum_{n=1}^{N} \| y_n - f(F(x_n)) \|^2
\]

but with \( \mu = 1 \) (so biased solution).
Related work: learning good internal representations

Updating weights and hidden unit activations in neural nets:


❖ 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:

\[
\min_x \sum_{n=1}^N f_n(x) \iff \min_{x_1, \ldots, x_N, z} \sum_{n=1}^N f_n(x_n) \text{ s.t. } x_n = z, \ n = 1, \ldots, N \iff \\
\min_\mathcal{L}(X, z, \Lambda) = \sum_{n=1}^N \left( f_n(x_n) + \lambda_n^T(x_n - z) + \frac{\mu}{2} \|x_n - z\|^2 \right)
\]

The aug. Lag. \( \mathcal{L} \) is minimised alternatingly over \( X, z \) and \( \Lambda \).
❖ 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 $p_1, \ldots, p_N$, in parallel. Like MAC’s $Z$ 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 $p_1, \ldots, p_N$.

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.
- 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”

- Model selection criteria (AIC, BIC, MDL, etc.) separate over layers:
  \[ \bar{E}(W) = E_{\text{nested}}(W) + C(W) = \text{nested-error} + \text{model-cost} \]
  \[ C(W) \propto \text{total # parameters} = |W_1| + \cdots + |W_K| \]

- Traditionally, a grid search (with \( M \) values per layer) means testing an exponential number \( M^K \) of nested models.

- In MAC, the cost \( C(W) \) separates over layers in the \( W \) step, so each layer can do model selection independently of the others, testing a polynomial number \( M^K \) of shallow models. This still provably minimises the overall objective \( \bar{E}(W) \).

- 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 \( W \) step. The algorithm usually settles in a region of good architectures early during the optimisation, with small and infrequent changes thereafter.

MAC searches over the parameter space of the architecture and over the space of architectures itself, in polynomial time, iteratively.
Experiment: RBF autoencoder (model selection)

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

- **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

- **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.
Shared-memory multiprocessor model using the Matlab Parallel Processing Toolbox: simply change `for` to `parfor` in the W and 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.
Distributed optimisation with MAC: ParMAC


❖ Large dataset distributed over $P$ machines, each storing $\frac{N}{P}$ points.
❖ Must keep communication low (it is much more costly than computation).
❖ ParMAC: a parallel, distributed optimisation framework for MAC:
   ✦ $W$ step: only model parameters are communicated (not data or auxiliary coordinates), following a circular topology, which implicitly implements a stochastic optimisation.
   ✦ $Z$ step: as usual (no communication).
Facilitates data shuffling, load balancing, fault tolerance and streaming data processing.
❖ Convergence guarantees carry over with an SGD-type condition for the $W$ step.
❖ We can predict theoretically the parallel speedup $S(P)$. Typically, $S(P) \approx P$ if $P \leq \# \text{submodels in } W$ step, i.e., the speedup is nearly perfect if using fewer machines than submodels. This affords very large speedups for large datasets and large models.
Experiments with a binary autoencoder for binary hashing using a database of images ($D = 128$ SIFT features):

- **SIFT-1M**: $N = 10^6$ training points, hash code of $L = 16$ bits
- **SIFT-1B**: $N = 10^8$ training points, hash code of $L = 64$ bits

Implemented in C/C++ using MPI (Message Passing Interface).

![Graphs showing speedup $S(P)$ vs. number of machines $P$ for SIFT-1M and SIFT-1B with varying epochs.](attachment:graphs.png)
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

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
- \( Z \) step separable or coupled on \( N \) auxiliary coordinates
1. Learning low-dimensional features for classification

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

- first we reduce the dimensionality of the input \( x \) with a mapping \( F \) 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 \( z_n \) the projection of each \( x_n \).

The resulting MAC algorithm alternates:

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

What if we want to use a different \( F \) or \( g \)?

- \( F \): simply call a different algorithm in \( F \) step black box: no need to compute gradients or code anything
- \( g \): call a different algorithm in \( g \) step, but the \( Z \) step does change.
1a. Low-dimensional SVM


**Binary classifier** $g$: linear SVM $g(z) = w^Tz + b$.

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

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

s.t. \(y_n(w^TF(x_n) + b) \geq 1 - \xi_n, \xi_n \geq 0, n = 1, \ldots, N\).

- If $F =$ identity then $y = g(F(x)) = w^Tx + b$ is a linear SVM and the problem is a convex quadratic program, easy to solve.

- If $F$ is nonlinear then $y = g(F(x)) = w^TF(x) + b$ is a nonlinear classifier and the problem is nonconvex, difficult to solve.
The problem is “nested” because of $g(F(\cdot))$ in the constraints:

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

s.t. $y_n (w^T F(x_n) + b) \geq 1 - \xi_n$, $\xi_n \geq 0$, $n = 1, \ldots, N$.

Break the nesting by introducing an auxiliary coordinate $z_n$ per point:

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

s.t. $y_n (w^T z_n + b) \geq 1 - \xi_n$, $\xi_n \geq 0$, $z_n = F(x_n)$, $n = 1, \ldots, N$.

The auxiliary coordinates correspond to the low-dimensional projections but are separate parameters.
Solve this constrained problem with a quadratic-penalty method: optimise for fixed penalty parameter $\mu > 0$ and drive $\mu \to \infty$:

$$\min_{F, g, \xi, Z} \lambda R(F) + \frac{1}{2} \|w\|^2 + C \sum_{n=1}^{N} \xi_n + \frac{\mu}{2} \sum_{n=1}^{N} \|z_n - F(x_n)\|^2$$

s.t. $y_n(w^T z_n + b) \geq 1 - \xi_n$, $\xi_n \geq 0$, $n = 1, \ldots, N$.

Alternating optimisation of the penalty function for fixed $\mu$:

- **g step**: fit a linear SVM to $\{(z_n, y_n)\}_{n=1}^{N}$
  A classifier using as inputs the auxiliary coordinates.

- **F step**: fit a nonlinear mapping $F$ to $\{(x_n, z_n)\}_{n=1}^{N}$
  A regressor using as outputs the auxiliary coordinates.

- **Z step**: set $z_n = F(x_n) + \gamma_n y_n w$, $n = 1, \ldots, N$
  A closed-form update for each point's auxiliary coordinate $z_n$. 

(p. 46)
1a. Low-dimensional SVM: solution of the Z step

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

$$\min_{z_n \in \mathbb{R}^L, \xi_n \in \mathbb{R}} \|z_n - F(x_n)\|^2 + \frac{2C}{\mu} \xi_n \quad \text{s.t.} \quad y_n (w^Tz_n + b) \geq 1 - \xi_n, \; \xi_n \geq 0$$

with closed-form solution $z_n = F(x_n) + \gamma_n y_n w$, where

$$\gamma_n = \frac{1}{2} \min(\lambda_{1,n}, 2C/\mu) \quad \text{and} \quad \lambda_{1,n} = \frac{2}{\|w\|^2} \max(0, 1 - y_n (w^T F(x_n) + b))$$

This corresponds to three possible cases. Cost: $O(L)$. 

![Diagram showing three cases for $\lambda_{1,n}$: $0$, $0 < \lambda_{1,n} < 2C/\mu$, and $\lambda_{1,n} = 2C/\mu$. The $z_{\text{opt}}$ point is indicated in each case.](image)
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.
- **F step**: a regressor as before.
- **g step**: a classifier as before, but with $K$ binary SVMs trained in parallel (one per class).
- **Z step**: a QP for each point on $L + K$ variables and $2K$ constraints:

$$\min_{\mathbf{z}_n, \{\xi_n^k\}_{k=1}^K} \|\mathbf{z}_n - \mathbf{F}(\mathbf{x}_n)\|^2 + \sum_{k=1}^{K} C^k \xi_n^k$$

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

This has no closed-form solution, so we solve it numerically.
Reducing dimension with a RBF network $F(x) = \sum_{m=1}^{M} \alpha_m \phi_m(x)$ the classifier has the form of a nonlinear SVM $y = \sum_{m=1}^{M} v_m \phi_m(x) + b$.

- 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Test error</th>
<th># BFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest neighbour</td>
<td>5.34</td>
<td>10000</td>
</tr>
<tr>
<td>Linear SVM</td>
<td>9.20</td>
<td>—</td>
</tr>
<tr>
<td>Gaussian SVM</td>
<td>2.93</td>
<td>13827</td>
</tr>
<tr>
<td>low-dim SVM</td>
<td>2.99</td>
<td>2500</td>
</tr>
<tr>
<td>LDA (9) + Gaussian SVM</td>
<td>10.67</td>
<td>8740</td>
</tr>
<tr>
<td>PCA (5) + Gaussian SVM</td>
<td>24.31</td>
<td>13638</td>
</tr>
<tr>
<td>PCA (10) + Gaussian SVM</td>
<td>7.44</td>
<td>5894</td>
</tr>
<tr>
<td>PCA (40) + Gaussian SVM</td>
<td>2.58</td>
<td>12549</td>
</tr>
<tr>
<td>PCA (40) + low-dim SVM</td>
<td>2.60</td>
<td>2500</td>
</tr>
</tbody>
</table>

classification error wrt $L$

$L$-dim space (+ 2D PCA)

parallel processing speedup
1b. Low-dimensional logistic regression

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

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

\[
\min_{F, w} \sum_{n=1}^{N} \left( y_n \log(\sigma(w^T F(x_n))) + (1 - y_n) \log(\sigma(-w^T F(x_n))) \right) + \lambda_F R(F) + \lambda_g R(w)
\]

MAC-constrained problem:

\[
\min_{Z, F, w} \sum_{n=1}^{N} \left( y_n \log(\sigma(w^T z_n)) + (1 - y_n) \log(\sigma(-w^T z_n)) \right) + \lambda_F R(F) + \lambda_g R(w)
\]

s.t. \( z_n = F(x_n), \; n = 1, \ldots, N. \)

MAC-penalised objective (using the augmented Lagrangian, with Lagrange multipliers \( \alpha_n \)):

\[
\min_{Z, F, w} \sum_{n=1}^{N} \left( y_n \log(\sigma(w^T z_n)) + (1 - y_n) \log(\sigma(-w^T z_n)) \right) + \lambda_F R(F) + \lambda_g R(w)
\]

\[
- \sum_{n=1}^{N} \alpha_n^T (z_n - F(x_n)) + \frac{\mu}{2} \sum_{n=1}^{N} \left\| z_n - F(x_n) \right\|^2
\]
Alternating optimisation over \((g, F, Z)\):

- **g step**: fit a logistic regression to \((Z, Y)\).
- **F step**: fit a nonlinear mapping to \((X, Z)\).
- **Z step**: solve for each point \(n = 1, \ldots, N\) in parallel an \(L\)-dim smooth convex problem:

\[
\min_{z_n \in \mathbb{R}^L} - y_n \log(\sigma(w^Tz_n)) - (1 - y_n) \log(\sigma(-w^Tz_n)) \\
+ \frac{\mu}{2} \|z_n - F(x_n)\|^2 - \alpha^T_n (z_n - F(x_n))
\]

- **Solution**: a closed-form update \(z_n = F(x_n) + \frac{1}{\mu} \alpha_n - \frac{t}{\mu} w\)

\(t\) is the single solution of the scalar eq. \(t = 1 - y_n - \sigma \left( \frac{\|w\|^2}{\mu} t - w^T \left( F(x_n) + \frac{1}{\mu} \alpha_n \right) \right)\).

- **Also convex for multi-class case with softmax**

Then, update the Lagrange multipliers:

\[
\alpha_n \leftarrow \alpha_n - \mu(z_n - F(x_n)), \ n = 1, \ldots, N
\]
1b. Low-dimensional logistic regression: experiment

Low-dimensional logistic regression ($F = \text{RBF network with } 2500 \text{ BFIs}$) of MNIST digits ($K = 10$ classes)

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

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

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

Often used to visualise high-dimensional datasets in 2D.
Optimising $E(X)$ is difficult:

- There are $NL$ optimisation variables $x_1, \ldots, x_N$ one vector $x_n \in \mathbb{R}^L$ per data point.
- It is nonlinear and nonconvex.
- It contains $O(N^2)$ terms (one for every pair of points), so computing $E(X)$ and $\nabla E(X)$ is $O(N^2)$.

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.

- Gradient approximated using $N$-body methods, in $O(N \log N)$ (Barnes-Hut tree) or $O(N)$ (fast multipole methods).

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 (\(F\) can be linear, a neural net, etc.). For EE:

\[
P(F) = \sum_{n,m=1}^{N} w_{nm} \|F(y_n) - F(y_m)\|^2 + \lambda \sum_{n,m=1}^{N} \exp \left( - \|F(y_n) - F(y_m)\|^2 \right)
\]

Also called siamese networks.

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

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

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

\[
\min_{F,Z} E(Z) = \sum_{n,m=1}^{N} w_{nm} \|z_n - z_m\|^2 + \lambda \sum_{n,m=1}^{N} \exp \left( - \|z_n - z_m\|^2 \right)
\]

s.t. \(z_n = F(y_n), \ n = 1, \ldots, N\).

MAC-penalised objective: \(\min_{F,Z} E(Z) + \frac{\mu}{2} \|Z - F(Y)\|^2\).
Alternating optimisation over \((F, Z)\):

- **F step**: fit a nonlinear mapping to \((Y, Z)\)
- **Z step**: regularised nonlinear embedding

\[
\min_{F, Z} \left[ E(Z) + \frac{\mu}{2} \left\| Z - F(Y) \right\|^2 \right]
\]

- it does not separate over \(z_1, \ldots, z_N\)
  - 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 \(F\)
  - which does not benefit from the \(N\)-body methods.
- If we want to use a different embedding objective \(E(X)\) or projection mapping \(F\), we simply call a different embedding or regression routine in the \(Z\) or \(F\) step, resp., as a black box.
2. Parametric nonlinear embeddings: experiment

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

\[ P(F) \]

Runtime (seconds)

\[ \begin{array}{c|c|c}
\text{MAC} & \text{PE (minibatch)} & \text{PE (batch)} \\
\hline
-100 & -50 & 0 & 50 & 100 \\
-100 & -50 & 0 & 50 & 100 \\
\end{array} \]

Initial, free embedding $E(X)$

Final parametric embedding $E_p(Y)$
2. Parametric nonlinear embeddings: experiment (cont.)

MNIST digits visualised with EE using a linear mapping \( F(y) = Ay + b \).
2. Parametric nonlinear embeddings (cont.)

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

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

These embeddings appear along the path followed by MAC over \( \mu \): 

\[
\min_{F,Z} E(Z) + \frac{\mu}{2} \|Z - F(Y)\|^2 \quad \left\{ \begin{array}{ll}
\mu = 0, & \text{free embedding} \\
\mu \to 0^+, & \text{direct fit} \\
\mu \to \infty, & \text{parametric embedding}
\end{array} \right.
\]
2. Parametric nonlinear embeddings (cont.)

optimal PE
\( \mu \to \infty \)

direct fit
\( \mu \to 0^+ \)

free embedding
\( \mu = 0 \)

embeddings realizable by mappings \( F \in \mathcal{F} \)

path \( Z^*(\mu) \) for MAC

\( X^* \to F^*(Y) \)
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: $\mathbf{h}(\mathbf{x}) = \lfloor W \mathbf{x} \rfloor$ where $\lfloor \cdot \rfloor$ 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 $\mathcal{E}(\mathbf{h})$ that preserves distances in some way. Many formulations exist.

❖ Difficult optimisation because $\mathcal{E}$ depends on the output of $\mathbf{h}$, which is binary, so $\mathcal{E}$ is piecewise constant, nonsmooth and nonconvex. The problem is usually NP-complete.
We introduce as auxiliary coordinates $z_n \in \{0, 1\}^L$ the binary code of each $x_n$. This confines the difficult part of the optimisation to the $Z$ step. The resulting MAC algorithm alternates:

- training $L$ binary classifiers on $(X, Z)$, in parallel
  done with existing algorithms, e.g. linear SVM
- updating the $N$ auxiliary coordinates $Z$
  this is a binary optimisation that we need to solve
as $\mu \to \infty$ (in fact, we can prove the iterations stop for a finite $\mu$).

If we want to use a different hash function $h$ (linear, neural net. . . ), we simply call a different classifier routine in the $h$ 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


**Original (nested) objective function: reconstruction error:**

\[
E(h, f) = \sum_{n=1}^{N} \| x_n - f(h(x_n)) \|^2
\]

\[
\begin{align*}
\text{encoder } h &: \mathbb{R}^D \to \{0, 1\}^L \text{ (our hash function)} \\
\text{decoder } f &: \{0, 1\}^L \to \mathbb{R}^D
\end{align*}
\]

**MAC-constrained problem:**

\[
\min_{h, f, Z} \sum_{n=1}^{N} \| x_n - f(z_n) \|^2 \quad \text{s.t.} \quad z_n = h(x_n), \ z_n \in \{0, 1\}^L, \ n = 1, \ldots, N.
\]

**MAC-penalised objective:**

\[
\min_{h, f, Z} \sum_{n=1}^{N} \left( \| x_n - f(z_n) \|^2 + \mu \| z_n - h(x_n) \|^2 \right) \quad \text{s.t.} \quad \begin{cases} 
\{ z_n \in \{0, 1\}^L \\
n = 1, \ldots, N. \end{cases}
\]
3a. Binary autoencoder: alternating optimisation

Alternating optimisation over \((h, f, Z)\):

- **h step**: fit \(L\) binary classifiers to \(\{(x_n, z_n)\}_{n=1}^N\).
- **f step**: fit a regression mapping to \(\{(z_n, y_n)\}_{n=1}^N\).
- **Z step**: solve for each point \(n = 1, \ldots, N\) a binary optimisation in \(L\) variables:

\[
\min_{z_n \in \{0,1\}^L} \|x_n - f(z_n)\|^2 + \mu \|z_n - h(x_n)\|^2
\]

This is usually **NP-complete** (for example, if \(f\) 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

\[
\text{input } X_{D \times N} = (x_1, \ldots, x_N), L \in \mathbb{N}
\]

initialise \( Z_{L \times N} = (z_1, \ldots, z_N) \in \{0, 1\}^{LN} \)

\[
\text{for } \mu = 0 < \mu_1 < \cdots < \mu_\infty
\]

\[
\text{for } l = 1, \ldots, L \quad \text{h step}
\]

\[
h_l \leftarrow \text{fit SVM to } (X, Z_l)
\]

\[
f \leftarrow \text{least-squares fit to } (Z, X) \quad \text{f step}
\]

\[
\text{for } n = 1, \ldots, N \quad \text{Z step}
\]

\[
z_n \leftarrow \arg \min_{z_n \in \{0,1\}^L} \|x_n - f(z_n)\|^2 + \mu \|z_n - h(x_n)\|^2
\]

\[
\text{if no change in } Z \text{ and } Z = h(X) \text{ then stop}
\]

\[
\text{return } h, Z = h(X)
\]

Repeatedly solve: classification (h), regression (f), binarisation (Z).
3a. Binary autoencoder: experiment

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

- **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


Original (nested) objective function: \( \mathcal{L}(h) = \sum_{n,m=1}^{N} L(h(x_n), h(x_m); w_{nm}) \)

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

Many formulations exist, e.g. \( \{ w_{nm} = -1/0 \) if similar/dissimilar/don’t know \)

\( L_{KSH}(z_n, z_m; w_{nm}) = (z_n^Tz_m - Lw_{nm})^2. \)

MAC-constrained problem:

\[
\min_{h, Z} \sum_{n,m=1}^{N} L(z_n, z_m; w_{nm}) \quad \text{s.t.} \quad z_n = h(x_n), \quad z_n \in \{0, 1\}^b, \quad n = 1, \ldots, N.
\]

MAC-penalised objective:

\[
\min_{h, Z} \sum_{n,m=1}^{N} L(z_n, z_m; w_{nm}) + \mu \sum_{n=1}^{N} \| z_n - h(x_n) \|^2 \quad \text{s.t.} \quad \left\{ \begin{array}{l}
\quad z_n \in \{0, 1\}^L \\
\quad n = 1, \ldots, N.
\end{array} \right.
\]
3b. Affinity-based objective function: alternating optimisation

Alternating optimisation over \((h, Z)\):

- **h step**: fit \(L\) binary classifiers to \(\{(x_n, z_n)\}_{n=1}^{N}\), in parallel.
- **Z step**: minimise the following regularised binary embedding:

\[
\min_z \sum_{n,m=1}^{N} L(z_n, z_m; w_{nm}) + \mu \sum_{n=1}^{N} \| z_n - h(x_n) \|^2 \quad \text{s.t.} \quad \{ z_n \in \{0, 1\}^L \}
\]

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

Alternating optimisation over the \(l\)th bit of each code for \(l = 1, \ldots, L\), 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.
The free binary codes $Z^*$ are an (approximate) minimiser of the loss $\mathcal{L}(Z)$ without any constraint (i.e., disregarding the hash function). They are usually not realisable by any (linear, say) hash function ($h(X) \neq Z \forall h$). 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.
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:
  - M step: minimise (train) layers
  - C step: coordinate layers
- Advantages:
  - easy to develop, reuses existing algorithms or code for shallow models as a black box
  - convergent
  - embarrassingly parallel
  - 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.
Original reference for MAC:

Tutorials, reviews:
❖ C-P: *A tutorial on nested system optimisation using the method of auxiliary coordinates.*
❖ C-P: *A gallery of proximal operators arising in the method of auxiliary coordinates.*

Extensions or related work:
❖ C-P & Vladymyrov: *A fast, universal algorithm to learn parametric nonlinear embeddings*. NIPS 2015.

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