Neural Computation VS 265

Contents

Neural Computation **Fall 2016**

Supervised Learning: September 15

Table of Contents Local *Written by Brandon McKinzie*

- Perceptron review for the plebs.
- XOR problem not linearly separable.
- Learning rules for two-layer network:
  - $E^{(\alpha)} = \frac{1}{2}\sum_i [T_i^\alpha - z_i(x^\alpha)]^2$.
  - $\Delta V_{ij} = \Big[[T_i - z_i(x)]\Big]\frac{\partial z_i}{\partial V_{ij}} = \delta_{ij} y_j$.
  - That was outer layer. For hidden layer:

$$\Delta W_{kl} = \eta \sum_i [T_i - z_i(x)] \frac{\partial z_i}{\partial W_{kl}}$$

  - Chain rule:
- **Second-order Methods**:
  - $E(w_0 + \Delta w) \approx E(w_0) + \Delta w^T \nabla E + \frac{1}{2}\Delta w^T H \Delta w$.
  - Minimized when $\nabla E + H\Delta w = 0$, thus $\Delta w* = -H^{-1}\nabla E$. Hessian, rather than approximating function as a line (like the gradient), approximates as a quadratic function.
  - **Momentum** is kinda second order.

$$\Delta w_{kl}(t+1) = -\eta \frac{\partial E}{\partial w_{kl}} + \alpha \Delta w_{kl}(t) \tag{1}$$

Neural Computation **Fall 2016**

Competitive Learning & Sparse Coding: September 27

Table of Contents Local

Written by Brandon McKinzie

- Competitive Learning (what follows is unrelated)
    - Most real data non-Gaussian. (bad for PCA/Hebbian).
    - Try **non-linear Hebbian learning**.

$$\begin{aligned} \Delta w_i &\propto y x_i \qquad (2) \\ &= f\left(\sum_j w_j x_j \right) x_i \qquad (3) \end{aligned}$$

    where we can expand f in a taylor series.

- Winner-take-all learning.
    - output neurons are connected to each other. fighting it out.

$$y_i = \begin{cases} 1 & u_i > u_j \forall j \neq i \\ 0 & \text{otherwise} \end{cases} \qquad (4)$$

    where u are standard weighted sum of inputs.

    - Learning rule:

$$\Delta w_{ij} = \eta y_i (x_j - w_{ij}) \qquad (5)$$

    where, if y_i wins, moves towards vector x_j.

    - Visually, weight vectors shift to align with clusters in data.
    - weight vector with highest inner product wins competition, and is therefore the one that moves toward the given data point (where each output has an associated weight vector).
    - Algorithm $\equiv$ K-means.
    - energy function:

$$\begin{aligned} E(\{w_i\}) &= \frac{1}{2} \sum_{i,\mu} M_i^\mu |x^{(\mu)} - w_i|^2 \qquad (6) \\ \Delta w_i &= \eta \sum_\mu M_i^\mu (x^\mu - w_i) \qquad (7) \end{aligned}$$

- **Sparse Coding:**
  - Allow multiple units to be active.
  - Barlow paper in 1972 is genesis of the idea.
  - Sensory system is organized to achieve as complete a representation of the sensory stimulus as possible w/min number of active neurons. i.e. minimize number of neurons k constrained by wanting to preserve max amount of information as possible about the input.
  - Adapt the coding strategy to the data structure.
  - is in the middle of *local codes* (grandmother cells) and *dense codes* (e.g. ascii). i.e. in the middle of easy-to-read-out and maximum-combinatorial capacity.
- Autoencoder networks.

  $$\min_{W,M} |x - \hat{x}|^2 \tag{8}$$

  Want to train output $\hat{x}$ to be same as input x, where data is passed through some bottleneck y bridged by W and M from input-¿middle-¿out. Idea is to exploit correlations in the input so that can pass through smaller space while preserving most of the information, and then passed back to the original (larger) space with a more sparse encoding. Basically is a compression algorithm. Also, see 'retinal bottleneck.'
- Bottleneck may have same number of units but with lower capacity (e.g. less bits per neuron).
- **sparse code bottleneck** limits the number of active units. i.e. middle space may in fact be much larger, but only allowed to use a subset of it when mapping $x \rightarrow \hat{x}$.

Neural Computation **Fall 2016**

Sparse Distributed Coding: September 29

Table of Contents Local *Written by Brandon McKinzie*

- VI simple-cell receptive fields are localized, oriented, and bandpass.
- PCA is really bad for such situations.
- To detect sharp edges in images, need high frequency and in-phase combinations.
- Higher-order image statistics:
  - phase alignment
  - orientation
  - motion
- want to move beyond pairwise correlations.
- WTA is too greedy, want more distributed strategy.
- Idea: **Projection pursuit**.
  - Look for low-dimensional projections that are as non-Gaussian as possible.
  - Projections tend to result in Gaussian distributions by the C.L.T.
  - Want to explore projections onto a weight vector until find something Non-Gaussian. Why? Because such a distribution could not have happened by accident.
- **Gabor-filter** response histograms are highly non-Gaussian.
- (Lab-related) Paper on *Forming sparse representations by local anti-Hebbain learning.*
  - Each neuron takes weighted input sum, as well as getting lateral inhibitaion by neighbors, but where the lateral weights are all negative. Put all through f, some sigmoidal non-linearity. ”leaky integrator”
  - Want population-sparcity, so need neurons decorrelated. Have three learning rules: anti-Hebbian, Hebbian, and threshold modification.

  - Threshold modification resembles homeostasis.

    $$\Delta t_i = \gamma(y_i - p) \tag{9}$$

    which is essentially SGD. Think about average behavior, as it relates to y_i output and p. p is a constant to be determined. Feedback loop. Adjusts spiking threshold.

  - Anti-hebb guarantees neurons are decorrelated.

    $$\Delta w_{ij} = \alpha(y_i y_j - p^2) \tag{10}$$

    where p^2 because this is what we would expected if i and j were decorrelated. There more coactive two neurons are, the more this drives them to repulse one another.

  - Standard hebbian rule

    $$\Delta q_{ij} = \beta y_i(x_j - q_{ij}) \tag{11}$$

    relates to sparsity fraction of neurons.

- Problems:
  - Don't know how to deal with graded (i.e. non-binary) input signals. Non-discrete stuff.
  - No objective function. Would like way to characterize how well system is performing.

- Led to Bruno's work: **sparse coding for graded signals**
  - Data described by

    $$I(x,y) = \sum_i a_i \; \phi_i(x,y) + \epsilon(x,y) \tag{12}$$

  - basis decomposition of input. neuron i with activity a_i means that need feature functions ϕ to describe model. Want the **neural activities a_i to be sparse**.
  - Constrain sparseness of a_i by imposing cost function on the activity:

    $$E = \frac{1}{2}|I - \Phi a|^2 + \lambda \sum_i C(a_i) \tag{13}$$

    where first term: preserve information and second term: I want to be sparse.

  - Penalty function C shaped like really steep parabola on zero. Or could do $C = |a_i|$, v-shaped thing.
  - Energy function determines dynamics of system. Want neuron activity to be expressible as a function of the input I.

- Compute coefficients a_i by gradient descent.

$$\tau \dot{a}_i = -\frac{dE}{da_i} \tag{14}$$

- Neuron i inhibited by neuron j proportioanl to their functions *phi* inner products.
- self-inhibition of neuron back on itself makes it sparse.
- Learning rule:

$$\begin{aligned} \Delta\phi_i &= -\eta\frac{\partial E}{\partial \phi_i} \qquad (15) \\ &= \Big[I - \Phi\hat{a}\Big]\hat{a}_i \qquad (16) \end{aligned}$$

Neural Computation **Fall 2016**

Foldiak Paper - Sparse Coding:

Table of Contents Local *Written by Brandon McKinzie*

- Abstract: A layer of simple Hebbian units connected by modifiable anti-Hebbian feedback connections can learn to code a set of patterns in such a way that statistical dependency between the elements of the representation is reduced, while information is preserved.
- ***Introduction***
  - Input-space that is our surrounding is enormous, but most inputs are highly correlated, which the brain may exploit to transform the high-dimensional pattern inputs to symbolic representations. Objects may be defined as conjunctions of highly correlated sets of components that are relatively independent of other such conjunctions[1]
- ***Unsupervised Learning***
  - The complexity of the mapping to be learnt $\Leftarrow$ complexity of the input.
  - Unsupervised learning exploits statistical regularities in input to learn a more meaningful symbolic representation.
- ***The Hebb Unit***
  - Simple model of cell (basically perceptron)

$$y = \begin{cases} 1 & \sum_j w_j x_j > \text{thresh} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

  - Can be thought of as pattern matching; y is maximal when weight vector = input vector pattern.
  - Hebb proposed: connection should become stronger if the two units being connected are active simultaneously: $\Delta w_j = x_j y$.
- ***Competitive Learning***

[1] Translation: objects are clumps of stuff that are usually found clumped together, and such that these clumps tend not to clump with other clumps.

- Out of the units receiving weighted sums of the input, only activate the unit with the largest weighted sum; suppress the output of all others.
- Results in a local, ”**grandmother-cell**” representation.
- Limited in number of different inputs it can discriminate, and in ability to generalize.

- ***Sparse Coding***
  - Distributed coding: instead, code each input state by a set of active units (rather than just one).
  - Pros: combinatorics of input states increases representational capacity. Cons: situations where many units are active per input pattern, and fact that learning can be extremely slow.
  - **Sparse Coding** is a compromise between distributed and local representations.
- ***Decorrelation***
  - Units *within* a layer are connected by modifiable inhibitory weights, governed by an **Anti-Hebbian learning rule**: if two units in same layer are active, connection becomes more inhibitory[2].

[2] which discourages their joint activity

Unsupervised Learning

- ***Bruno:PCA***
  - First, let's get this straight. Difference between **covariance** and **correlation**:

$$\textbf{COV}[X, Y] \triangleq \mathbb{E}\left[(X - \mu_X)(Y - \mu_Y)\right] \tag{18}$$

$$\textbf{CORR}[X, Y] \equiv \rho_{XY} \triangleq \frac{Cov[X, Y]}{\sigma_X \; \sigma_Y} \tag{corr}$$

  - Consider input stream $\mathbf{x}$ that has linear pairwise correlations[3] among its elements. Mathematically, correlation between elements x_i and x_j would imply that

$$\langle x_i x_j \rangle = \frac{\mathbb{E}\left[x_i x_j\right]}{\sqrt{\mathbb{E}\left[x_i\right] \mathbb{E}\left[x_j\right]}} \neq 0 \tag{19}$$

    or, equivalently, that $\mathbb{E}\left[x_i x_j\right] \neq \mathbb{E}\left[x_i\right] \mathbb{E}\left[x_j\right] = 0$. Bruno is correct that linear pairwise correlations imply that $c_{ij} \neq 0$, he is *absolutely incorrect* to say that c_{ij} is an "average over many examples." That is nothing more than academic sloppiness at its finest.

- ***HKP:PCA***
  - Goal: Find a set of M orthogonal vectors in data space that account for as much as possible of the data's variance. Projecting the data from original N-dimensional space onto the M-dimensional subspace spanned by these vectors then performs a **dimensionality reduction**.
  - HKP actually states accurately what Bruno meant to state: The kth principal component direction is along an eigenvector direction belonging to the kth largest eigenvalue of the full **covariance matrix**

$$\langle (\xi_i - \mu_i)(\xi_j - \mu_j) \rangle \tag{20}$$

[3] This is exactly what is meant by eq corr, Pearson's correlation coefficient. *Linear* because "it is a measure of the linear dependence between two variables X and Y."

  - **For zero-mean data this reduces to the corresponding EIGENVECTORS of the correlation matrix[4] C**
  - Note: I am now going to start from beginning of CH8 of HKP since I'm not understanding the stuff they are referencing FML

- ***HKP Ch8: Unsupervised Hebbian Learning***
  - Units need to learn patterns/correlations/categories in inputs and code the output. Units and connections display some degree of **self-organization**.
  - Redundancy provides knowledge: w/o redundancy there would be no patters to learn.

$$\text{MaxInfoPossible} - \text{InputContent} = \text{DegreeOfRedundancy} \tag{21}$$

  - **PLAIN HEBBIAN LEARNING.** Context: output will be continuous-valued and DO NOT have a winner-take-all character[5], and so the **purpose** is to measure familiarity or projecting onto principal components of input data.
  - Setup: Draw at each time step an input vector $\boldsymbol{\xi}$ from (multivariate) probability distribution $P(\boldsymbol{\xi})$ that has N components[6]. Network will learn to tell us - as output - how well an input conforms to the distribution[7]
  - (One linear output unit): Let V be a scalar-valued continuous output with a bunch of inputs pointing to it, with

$$V = \sum_j w_j \xi_j = \boldsymbol{w}^T \boldsymbol{\xi} = \boldsymbol{\xi}^T \boldsymbol{w} \tag{22}$$

  - Want large (on average) $V \leftrightarrow$ more probable $\boldsymbol{\xi}$. Why? Because then we can use the relative size of the output as a way of characterizing the sort of input it just received (see footnote 26 below). The weight update to do this is **plain Hebbian learning update:**

$$\Delta w_i = \eta V \xi_i \tag{23}$$

[4] NOTICE HOW I DIDN'T SAY REDUCES TO THE CORRELATION MATRIX.

[5] TODO: Come back and explain why this is true, because current Brandon thought otherwise.

[6] Confusingly, here N refers to the dimension of space that each input vector lives in (usually denoted by d.)

[7] **Q:** Come back and explain why we would want a network to do this. Biological relevance/analog? **A**: You need to view it in the context of the grandmother-cell. That's what this is all about. If a given neuron has a large linear output, then we have a good idea of what type of input went in; it was an input really similar to the weight vector. This begs the question, though: how does one determine a reasonable initialization for a given connected layer of weights to a single output? I suppose the answer is that this is the wrong question. Rather, we should interpret the outcome as resulting from a stream of particular inputs and, based on its future responses to inputs, we can determine what type of input went in. With the brain, this is like the jennifer aniston neuron: if that neuron fires, we can assume the person just saw something that resembled Jennifer Aniston.

where it is perhaps easier to think about the situation where $\Delta w_i = 0$ when analyzing, i.e. If $\xi_i = 0$ (which means it had nothing to do with the output), then don't increase it's weight[8].

- Problem: $\boldsymbol{w}$ grows without bound. However, suppose stable equilib exists for $\boldsymbol{w}$. This could happen for example, when considering that the update just performs $\boldsymbol{w} = \eta V \boldsymbol{\xi}$, where eventually $||\boldsymbol{w}|| >> ||\boldsymbol{\xi}||$ in addition to the fact that ξ is quite likely to be along $\boldsymbol{w}$. So at equilib, expect the updates to average to 0:

$$\begin{aligned} 0 &= \langle \Delta w_i \rangle && (24) \\ &= \langle \sum_j w_j \xi_j \xi_i \rangle && (25) \\ &= \sum_j \mathbf{C}_{ij} w_j && (26) \end{aligned}$$

where the brackets are *expectation values* in the sense that

$$\langle \xi_i \xi_j \rangle = \iint_{-\infty}^{\infty} \xi_i \xi_j f_{\xi_i \xi_j}(\xi_i, \xi_j) d\xi_i d\xi_j \tag{27}$$

where f is the PDF for the two random variables in question. I suppose that, since strictly speaking $\boldsymbol{w}$ isn't a random variable, that it can be pulled out along with the summation. That satisfies me for now.

- Given that $\boldsymbol{\xi}$ can be interpreted as a column vector, we have

$$\begin{aligned} \mathbf{C}_{ij} &\equiv \langle \xi_i \xi_j \rangle && (28) \\ \mathbf{C} &\equiv \langle \boldsymbol{\xi} \boldsymbol{\xi}^T \rangle && (29) \end{aligned}$$

Now, to be perfectly clear, this is NOT the correlation, but I am so sick and tired of caring that I'm just going to accept their absolutely incorrect definition and move on.

- Since I've read ahead, I know that the following property will be important to remember:

$$\begin{aligned} \forall \boldsymbol{x}, \; \boldsymbol{x}^T \mathbf{C} \boldsymbol{x} &= \boldsymbol{x}^T \langle \boldsymbol{\xi} \boldsymbol{\xi}^T \rangle \boldsymbol{x} && (30) \\ &= \langle \boldsymbol{x}^T \boldsymbol{\xi} \boldsymbol{\xi}^T \boldsymbol{x} \rangle && (31) \\ &= \langle (\boldsymbol{\xi}^T \boldsymbol{x})^2 \rangle && (32) \end{aligned}$$

- There are *only* unstable fixed points (unstable equilib) for the plain Hebbian learning procedure.
- **OJA'S RULE**. Goal: Modify plain Hebb rule such that $|\boldsymbol{w}| = 1$.
- Solution: Add a **weight decay** proportional to V^2:

[8] Minor TODO: Analyze case of non-binary (i.e. continuous both pos/neg) inputs/outputs.

$$\Delta w_i = \eta V(\xi_i - V w_i) \tag{33}$$

and we see that Δw depends on the difference between the input and the back-propagated output[9]

- Informal analysis for zero-mean data: The average component of ξ along w will be zero, but since this is an algorithm depending on an unstable equilibrium, it will tend to fall along the maximal eigenvector of $\mathbf{C}$.
- Oja's rule chooses the direction of $\boldsymbol{w}$ to maximize $\langle V^2 \rangle$.
- **Sanger's Learning Rule**. Setup: Now, instead of 1 output, have M output neurons with the hopes that they gives us the first M principal components of the input data. Architecture is ONE LAYER fully connected.
- The ith output is a linear neuron as usual given by

$$V_i = \sum_j w_{ij}\xi_j = \boldsymbol{w}_i^T \boldsymbol{\xi} = \boldsymbol{\xi}^T \boldsymbol{w}_i \tag{34}$$

- The Sanger's learning rule update for the connection *from* the jth input component *to* the ith output neuron (so we are only updating a single edge/line in the following) is

$$\Delta w_{ij} = \eta V_i \left(\xi_j - \sum_{k=1}^{i} V_k w_{kj} \right) \tag{35}$$

where the (converged) weight vectors to the output neurons are orthonormal and converge to the normalized eigenvectors in order of largest to smallest eigvals:

$$\boldsymbol{w}_i^T \boldsymbol{w}_j = \delta_{ij} \tag{36}$$

$$\boldsymbol{w}_i \to \pm \boldsymbol{c}^i \tag{37}$$

[9] Say 'back-propagated output' because we are subtracting what was put into the network by the resultant output *times* the connection (weight) between the input and said output. Dwelling on this *would* be overly pedantic, so move on.

Neural Computation **Fall 2016**

Lab 4 & LCA Handout:

Table of Contents Local *Written by Brandon McKinzie*

- Want to learn a “dictionary” from data
- Encode input data such that it can be reconstructed from that code, where dim(encoding) ¿ dim(input).
- Given N-dimensional input, build $N \times M$ dictionary[10] (matrix) $\mathbf{\Phi}$ where each column ϕ_i is a dictionary element with corresponding coefficient[11] a_i. Want to assemble $a_i\phi_i$ into a vector of **activations**.
- **GOAL:** Minimize energy function E, defined as

$$E = \frac{1}{2}||S - \hat{S}||_2^2 + \lambda \sum_i^M C(a_i) \tag{38}$$

  where $\hat{S} = \sum_i^M a_i\phi_i$ is for some reason called the image reconstruction. View this like a regularization procedure where the terms mean: (1) smallest difference between true image and reconstructed image (**reconstruction quality**); and (2) limit the number of **active elements**[12] a_i.

- Want to minimize E such that reconstructs data with fewest number of active elements, expressed as

$$\underset{a,\ \mathbf{\Phi}}{\arg\min}\Big(E\Big) \tag{argminE}$$

  where I guess the double argmin means ”minimize E by changing a and $\mathbf{\Phi}$ only and then give me the values of a and $\mathbf{\Phi}$.

- Popular cost function is the ℓ_1 penalty:

$$\sum_i^M C(a_i) = \sum_i^M |a_i| \tag{39}$$

- We compute coeff vector a using a “dynamic process”[13] that minimizes argminE.

[10] M ¿ N.

[11] Looks like $a_i \notin \mathbf{\Phi}$

[12] A.k.a sparsity constraints a.k.a limit activations.

[13] Okay well what the fuck is it?

- Method for computing the sparse code from a given input signal S and dictionary element ϕ_i is the **Locally Competitive Algorithm**.

  > The model describes an activation coefficient, a_k, as the thresholded output of some model neuron's **internal state**, u_k, which is analogous to the neuron's membrane potential.

- Here we compute the equation for state transitions (updates) from the energy function. First, for grad descent on an individual neuron's activity, $a_k(t)$:

  $$-\frac{\partial E(t)}{\partial a_k(t)} = \sum_i^N \left[S_i \mathbf{\Phi}_{ik} - \sum_{j \neq k}^M \mathbf{\Phi}_{ik} \mathbf{\Phi}_{ij} a_j \right] - a_k - \lambda \frac{\partial C(a_k)}{\partial a_k} \tag{40}$$

  where the constants are S and $\mathbf{\Phi}$. Want system to evolve over time to produce optimal set of activations $a(t)$.

- Meaning of ϕ_k. Associated with kth (output?) neuron. Indicates the connection strength [between that neuron and] each pixel in the input.

  > In this model, we are going to
  >
  > nd a sparse code for one patch of an image at a time, so that all M neurons are connected to the same image patch, S.

HKP 9.4 - Feature Mapping:

Nearby (similar) outputs corresponding to nearby (similar) input patterns. Such a map (similar inputs → similar outputs) is a **feature map**. The conventional case: 2 continuous-valued inputs x and y map (fully-connected) to a two-dimensional x,y grid. Want nearby input values (in the actual euclidean sense) (x, y) to be mapped closely in the output 2D grid.

Kohonen's Algorithm implements the self-organizing (feature) map by using competitive learning, where now we update weights going to the *neighbors* of the winning unit as well as those of the winning unit itself.

- Setup: N continuous-valued inputs ξ_1 to ξ_N, defining a point $\boldsymbol{\xi}$ in N-dimensional space. Outputs O_i are arranged in (typically) a 1-D or 2-D array fully connected via w_{ij} to the inputs.
- A competitive learning rule is used, choosing output O_i^* as winner, determined by

$$|\boldsymbol{w_i^*} - \boldsymbol{\xi}| \leq |\boldsymbol{w}_i - \boldsymbol{\xi}| \quad \textbf{(for all i)} \tag{41}$$

- The **Kohonen Learning Rule** is

$$\Delta w_{ij} = \eta \Lambda(i, i^*)(\xi_j - w_{ij}) \tag{42}$$

  where $\Lambda(i, i^*)$ is the **neighborhood function**, equal to 1 for $i = i^*$ and falls off with distance $|\boldsymbol{r} - \boldsymbol{r_i^*}|$.
- A typical choice for $\Lambda(i, i^*)$ is

$$\Lambda(i, i^*) = \exp\left(- \frac{|\boldsymbol{r} - \boldsymbol{r_i^*}|^2}{2\sigma^2} \right) \tag{43}$$

  where σ is width parameter that *is gradually decreased.* Apparently $\eta(t) \propto t^{-\alpha}$ where $0 < \alpha \leq 1$ is a good choice.

Neural Computation **Fall 2016**

Locally Linear Embedding: October 19

Table of Contents Local *Written by Brandon McKinzie*

LLE is an unsupervised learning algorithm for dimensionality reduction. Similar to PCA and MDS[14], LLE is called an *eigenvector method*. The basic idea is illustrated below in figure 1.

The **LLE algorithm**:

1. Compute the neighbors of each data point, X_i.
2. Compute the weights W_{ij} that best reconstruct each X_i from its neighbors, minimizing the cost in

$$ReconErr(W) = \sum_i |X_i - \sum_j W_{ij} X_j|^2 \tag{44}$$

   by constrained linear fits.

3. Compute the Y_i reconstructed by the weights W_{ij}, minimizing the quadratic form in

$$\Phi(Y) = \sum_i |Y_i - \sum_j W_{ij} Y_j|^2 \tag{45}$$

   by its bottom nonzero eigenvectors.

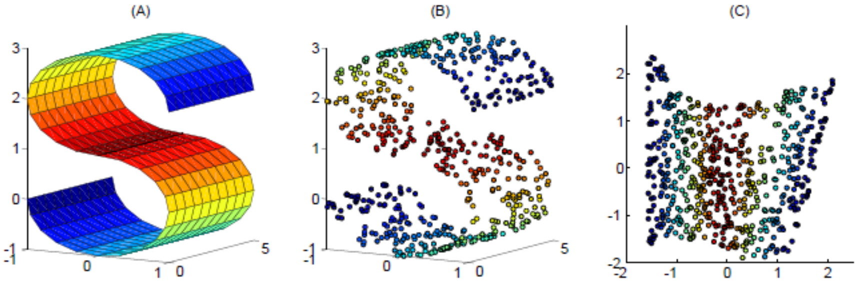


Figure 1: (A) Multidimensional sampling distribution with clear underlying manifold representation. (B) Points that were sampled. (C) The neighborhood-preserving mapping discovered by LLE.

[14] Multidimensional scaling

Some intuition/overview of the algorithm. We expect each X_i and its neighbors to lie on or close to a **locally linear** patch of the manifold. We characterize these patches by linear coefficients W_{ij} that reconstruct each X_i from its neighbors. As seen in eq. 44, the reconstructed point X_i is given by $\sum_j W_{ij} X_j$.

Computing/analyzing the weights W_{ij}. Minimize eq 44 subject to

$$\forall X_j \notin \text{Neighbors}(X_i) \; : \; W_{ij} = 0 \tag{46}$$

$$\sum_j W_{ij} = 1 \tag{47}$$

where the optimal weights are found by solving a least squares problem. Note that for a given data point, *the weights are invariant to rotations, rescalings, and translations of that data point and its neighbors.*[15] If the data lie on some nonlinear manifold of $d << D$, then there exists a *linear mapping* (approx) from the high-D coordinates of each neighborhood to global ('internal') coordinates on the manifold. Lucky for us, W can also do this![16]

Explanation of eqs. 44 45. Note that eq. 44 is minimized over the W_{ij}, while equation 45 is minimized over the Y_i. In English: We first want the weights W that reconstruct each X_i by its neighbors in the high-D space. Then, we want the low-d coordinates Y_i, representing the global coordinates on the manifold, that correspond to each X_i from the original space. **How it is minimized:**

> it can be minimized by solving a sparse $N \times N$ eigenvector problem, whose bottom d non-zero eigenvectors provide an ordered set of orthogonal coordinates centered on the origin.

Implementation of algorithm. Only one free parameter: number of neighbors per data point K. W_{ij} and Y_i are computed by 'standard linear algebra'.

[15] In other words, since the weights just characterize the local patch of the given data point, that patch shouldn't change if we shift the data, rotate it, or scale it. The neighboring points should remain the same.

[16] In particular, the same weights W_{ij} that reconstruct the ith data point in D dimensions should also reconstruct its embedded manifold coordinates in d dimensions

Neural Computation **Fall 2016**

Recurrent Neural Networks: October 20

Table of Contents Local *Written by Brandon McKinzie*

Lab 6 Overview. Briefly goes over how we can corrupt some number of bits and reconstruct a desired image [with hopfield nets]. Unfortunately, can get "spurious basins of attractions." Pushing down on some region of landscape causes pushing up of some other region. Want to carve energy landscape so that we push down only where we want.

Bump circuits and ring attractors. Want family of solutions (e.g. a line) that solutions drawn to (called line attractors). Head-direction neurons[17] look like an internal compass for animals; encode direction of head in *world coordinate system*. Different dots represent a single neuron's firing rate at different relative head directions. **Ring attractors**: population of neurons that with bumps that are stable (?). Convergence/stability because T_{ij} matrix is symmetric. Symmetric = fixed stable; Asymmetric =

Bruno shows simulation:

- 32 neurons where bar is activity of neuron.
- Start with random symmetric weight hopfield net.
- Eventually weights converge to gaussian-like bump; an equipotential pattern.
- If we add small asymmetry (gamma) to weights, then population (bump) would shift. Bump change is shifting position, and when the asymmetry stops (we stop moving our head) the population stays fixed. In English: moving head causes bump to move but when we stop moving, they stay put.
- **For more**: Read "catcher and zong" paper. I misspelled that.

[*Enter guest lecturer Alex Anderson*] ***Recurrent Neural Networks***:

→ Starts with handwriting network.
→ RNNs good for sequence prediction tasks with "long-term dependencies."

[17] Literally referring to direction of [e.g. some animal's] head

Backprop Review. Blobs do activation computation and transformers do propagations. Note: A^t is target output values.

Problem to Solve. Feed net a bunch of sentences and have it fill in the blank somewhere, based on the previous info it was fed. Mad libs. Have network understand particular frame of movie by exploiting context; just showing it a bunch of frames isn't enough/good approach.

RNN loops/Notation. Feed *time sequence* x_t to block A. Two figures in this slide are different reps of same thing; instructor prefers the right fig. H_k is hidden state we want to predict[18]. f can be some nonlinearity like *tanh*. In RNNs, cost function typically broken up over time; so C_k is cost at timestep k. Usually want hidden state to *summarize* the past. Hidden state traces out a trajectory over time [wut].

Unroll a RNN. Can basically turn RNN into a linearized hidden markov chain, where time proceeds to the right. Total cost is given by cost at each time step.

Long-term Dependencies. Shows toy model. Imagine ur an ant walking along graph. Given string of nodes, predict next letter each timestep [solve the question mark in slide]. Don't necessarily want/need whole past as input. Want to remember past [hidden] states, but they usually get overwritten; want to save it more efficiently. Key: want to make function simple, give the network parameterization.

Exploding/vanishing gradients. Local dependencies easy to learn.

→ Once we get to B, want network to output a U.
→ To learn, errors need to propagate back [in time], so we can change the weights that started the error: gradient of cost at timestep k with respect to initial weights using chain rule. Basically a product of k matrices.
→ If k large and matrices have eigvals less than 1, gradients *vanish*. If eigvals above 1, gradients *explode*. So what we want is for eigvals to be very near 1.
→ **Todo**: lookup relationship between eigval magnitudes and determinant.

[18] Analogy to hopfield: H is like hopfield B. X is like external I in hopfield.

Solution: Multiplicative Gating. Helps protect hidden state. MultGate can be either 0 or 1, and we multiply the hidden state by that value; if we 0 lose the hidden state; if 1 we keep the hidden state. Since binary functions not smooth/differentiable, continuous gating is better. [slide note: top row is w/o multgate, lower row is with multgate]. Key equation:

$$c_t = f_t \odot c_{t-1} + i_t \odot j_t$$

where $\odot$ is elementwise product.

Note: This is in TensorFlow now.

Neural Computation **Fall 2016**

Hopfield Networks Handout: October 26

Table of Contents Local *Written by Brandon McKinzie*

Energy Function. The following governs the dynamic of pairwise recurrently connected networks.

$$E = \frac{1}{2} \sum_i \sum_{j \neq i} T_{ij} V_i V_j \tag{48}$$

For symmetric weights $T_{ij} = T_{ji}$, consider the change in energy ΔE resulting from making a positive change to V_k[19]

$$\Delta E = -\Delta V_k \sum_{i \neq k} T_{ki} V_i \tag{49}$$

which will be *negative* if both ΔV_k and the sum are positive, thus decreasing the overall energy (good). Conversely, if sum is negative, we should decrease value of V_k. **Critical assumption**: Symmetric $T_{ij} = T_{ji}$. Without this assumption, impossible to show the system will have fixed points.

> For a network with symmetric connections though, the dynamics will converge to so-called **basins of attraction**.

Setting the Weights. Goal: store pattern $\mathbf{V}^\alpha$ as basin of attraction in network. One approach: the **Hebbian prescription** $T_{ij} = V_i^\alpha V_j^\alpha$.

→ **Single memory storage**. Now, the summed input sent to, say, the ith unit in response to some $\mathbf{V}^\beta$ will be given by

$$U_i = V_i^\alpha \sum_{j \neq i} V_j^\alpha V_j^\beta \tag{50}$$

and thus if $\mathbf{V}^\alpha = \mathbf{V}^\beta$, U_i won't flip sign and the networks stays put.

→ **Multiple memories**. Now, need to form as many basins of attractions as memories we want stored. Set weights with a superposition over each desired *memory* $\mathbf{V}^\alpha$: $T_{ij} = \sum_\alpha V_i^\alpha V_j^\alpha$, and the corresponding response of the ith neuron is

$$U_i = \sum_\alpha V_i^\alpha \sum_{j \neq i} V_j^\alpha V_j^\beta \tag{51}$$

[19] If it is -1, change to +1, else just keep where it is.

Capacity for a Hopfield Network. If the patterns to store (memories) have *few elements in common*, then cross terms $\sum_{j \neq i} V_j^\alpha V_j^\beta$ tend to zero for $\alpha \neq \beta$ (since each V_j^α is ±1 and a random average over ±1 is zero) and U_i won't change. As we store more patterns which are *similar*, memories degrade and basins gone from desired locations. This **capacity** for Hopfield is $\approx 15\%$ of the number of neurons in network[20].

[20] Assuming the stored patterns are relatively dissimilar.

Neural Computation **Fall 2016**

Mixture of Gaussians and EM Algorithm: November 3

Table of Contents Local *Written by Brandon McKinzie*

(MOG) Model Assumptions. Assume each $x^{(i)}$ generated by sampling $z^(i) \sim Multinom(\phi)$[21] and then positing that $x^{(i)}$ was drawn from the Gaussian associated with $z^{(i)}$ (so k possible Gaussians since z could be one of k classes), i.e.

$$x^{(i)}|z^{(i)} = j \sim \mathcal{N}(\mu_j, \Sigma_j) \tag{52}$$

Goal. Estimate the parameters of our model, ϕ, μ, Σ. We can do this by writing the likelihood of our data (m data points):

$$\ell(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \log \left[\sum_{z^{(i)}=1}^{k} p\left(x^{(i)}|z^{(i)};\ \mu, \Sigma\right)\ p(z^{(i)}; \phi) \right] \tag{53}$$

where the inner sum comes from using Bayes rule on $p(x^{(i)})$. Note that *we don't know the $z(i)$ for each data point*, that information is hidden. This means we can't get closed-form solutions by doing MLE/taking derivatives as usual.

EM Algorithm. Purpose: allow us to move forward given we can't do the standard MLE approach (since we don't know the values of each $z^{(i)}$.)

1. **E-step**. Estimate the values of the $z^{(i)}$s by evaluating

$$\text{For each } i, j, \text{ set } w_j^{(i)} := p(z^{(i)} = j \mid x^{(i)};\ \phi, \mu, \Sigma) \tag{54}$$

   with Bayes' rule, using whatever the current values of the estimated parameters are.

[21] where k-vector ϕ elements $\phi_j = P(z^{(i)} = j)$

2. **M-step**. Update parameters via standard MLE approach.

$$\phi_j := \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)} \tag{55}$$

$$\mu_j := \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}} \tag{56}$$

$$\mathbf{\Sigma}_j := \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}} \tag{57}$$

Some properties of the EM-algorithm. It is similar to the K-means algorithm, and like K-means, it is also prone to local minima, so reinitializing at several different initial parameters may be a good idea.

Neural Computation **Fall 2016**

Boltzmann Machines: November 6

Table of Contents Local *Written by Brandon McKinzie*

Lecture Slides

[Here, I use my own notation that actually remains consistent/intuitive...]

The energy, taken as a sort of average over all pairwise connections and the corresponding network-wide probability is

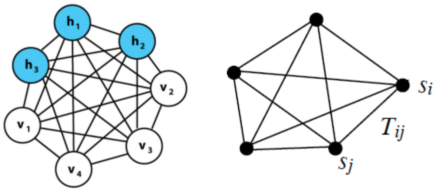
$$E(\boldsymbol{s}) = -\frac{1}{2}\sum_{ij} w_{ij} s_i s_j \tag{58}$$

$$P(\boldsymbol{s}) = \frac{1}{Z} e^{-\beta E(\boldsymbol{s})} \tag{59}$$

HKP Chapter 7.1

Structure. The units S_i are divided into

1. **Visible units**. These may be further divided into, e.g., input and output units.
2. **Hidden units**. No connection to the outside world.



The units are **stochastic**, meaning

$$S_i = \begin{cases} +1 & \text{with probability } g(h_i) \\ -1 & \text{with probability } 1 - g(h_i) \end{cases} \tag{60}$$

$$h_i = \sum_j w_{ij} S_j \tag{61}$$

$$g(h) = \frac{1}{1 + \exp(-2\beta h)} \tag{62}$$

where $\beta = 1/T$ and T is the "temperature." **Goal:** adjust the weights w_{ij} to give the states of the *visible* units a particular desired probability distribution. This differs from a Hopfield network in that now we have hidden units. With hidden units, we can specify *higher-order correlations* between units[22].

Boltzmann Machines - AIFH

A Boltzmann machine is essentially a fully connected, two-layer neural network; one visual layer and one hidden layer. **Restricted** Boltzmann machines are not fully connected; all hidden neurons are connected to each visible neuron and vice versa, but there are no connections between neurons of the same layer.

- Boltzmann machines are a generative model.
- The values presented to the visible neurons of a Boltzmann machines, when considered with the weights, specify a probability that the hidden neurons will assume a value of 1, as opposed to 0.

Differences with Hopfield networks.

→ Hopfield networks suffer from recognizing false patterns.
→ BM can store a greater capacity of patterns than HN.
→ HN require the input patterns to be uncorrelated.
→ BM can be stacked to form layers.

[22] Whereas, in hopfield networks, we can do no more than specify all the $\langle S_i \rangle$ and $\langle S_i S_j \rangle$

Neural Computation **Fall 2016**

Independent Component Analysis: November 12

Table of Contents Local *Written by Brandon McKinzie*

Sparse Coding Review. The goal is to represent input x as vector of sparse coefficients s, where their relationship is given in the form

$$\boldsymbol{x} = \mathbf{A}\boldsymbol{s} + \boldsymbol{n} \tag{63}$$

→ $\boldsymbol{x}$: Data matrix with shape n by d.
→ $\mathbf{A}$: Feature matrix (contains the basis functions) of shape n by $k > n$. Here, k is the number of sparse coefficents (shape of $\boldsymbol{s}$).
→ $\boldsymbol{s}$: The sparse coefficient matrix. Recall that the equation above is sometimes written as a sum over the basis functions ϕ.

$$\boldsymbol{x} = \left(\sum_k \phi_k(\boldsymbol{x}) s_k \right) + \boldsymbol{n} \tag{64}$$

We say "sparse" because $s_i = 0$ a lot, and shape($\boldsymbol{s}$) > shape($\boldsymbol{x}$).
→ n: Gaussian noise. Entries are typically much smaller than entries of $\mathbf{A}\boldsymbol{s}$.

- **Model distribution.**

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{s}) p_s(\boldsymbol{s}) d\boldsymbol{s} \qquad \text{where} \tag{65}$$

$$p(\boldsymbol{x}|\boldsymbol{s}) \propto e^{-\frac{|\boldsymbol{x}-\mathbf{A}\boldsymbol{s}|^2}{2\sigma_n^2}} \qquad \text{and} \tag{66}$$

$$p_s(\boldsymbol{s}) \propto e^{-\sum_i C(s_i)} \quad \leftrightarrow \quad \mathbf{C} = -\log(p(\boldsymbol{s})) \tag{67}$$

- **Learning Rule**.

$$\Delta\mathbf{A} \propto \frac{\partial}{\partial \mathbf{A}} \langle \log p(\boldsymbol{x}) \rangle \tag{68}$$

$$\Delta\mathbf{A} \propto \left\langle \int [\boldsymbol{x} - \mathbf{A}\boldsymbol{s}]\boldsymbol{s}^T p(s|x) d\boldsymbol{s} \right\rangle \qquad \text{(Analytic)} \tag{69}$$

$$\Delta\mathbf{A} \propto \left\langle [\boldsymbol{x} - \mathbf{A}\hat{\boldsymbol{s}}]\hat{\boldsymbol{s}}^T \right\rangle \qquad \text{(In Practice)} \tag{70}$$

where, $\hat{\boldsymbol{s}}$ represents a single sample at the posterior maximum:

$$\begin{aligned}
\hat{\boldsymbol{s}} &= \arg\max_{s} p(\boldsymbol{s}|\boldsymbol{x}) \quad (71)\\
&= \arg\min_{s} \left[-\log(p(\boldsymbol{s}|\boldsymbol{x}))\right] \quad (72)\\
&= \arg\min_{s} \left[\frac{\lambda_n}{2}|\boldsymbol{x}-\mathbf{A}\boldsymbol{s}|^2 + \sum_i C(s_i)\right] \quad (73)\\
\nabla_s \hat{\boldsymbol{s}} &\propto \lambda_n \mathbf{A}^T[\boldsymbol{x}-\mathbf{A}\boldsymbol{s}] - C'(\boldsymbol{s}) \quad (74)\\
&\triangleq \lambda_n[\boldsymbol{b}-\mathbf{G}\boldsymbol{s}] - \boldsymbol{z}(\boldsymbol{s}) \quad (75)
\end{aligned}$$

Independent Component Analysis

Independent Component Analysis. Special case where (1) $\mathbf{A}$ is square and full rank, and (2) the noise $\boldsymbol{n} = 0$. Now we have model

$$\boldsymbol{x} = \mathbf{A}\boldsymbol{s} \quad \rightarrow \quad \boldsymbol{s} = \mathbf{A}^{-1}\boldsymbol{x} \quad (76)$$

Learning rule.

$$\Delta\mathbf{A} \propto \left\langle [\boldsymbol{x} - \mathbf{A}\hat{\boldsymbol{s}}]\hat{\boldsymbol{s}}^T \right\rangle \quad (77)$$

$$\Delta\mathbf{A} \propto \mathbf{A}\left\langle \boldsymbol{z}(\boldsymbol{s})\boldsymbol{s}^T \right\rangle - \mathbf{A} \quad (78)$$

Equations for Different Priors

$$\text{[LAPLACE]} \quad P(s_i) \propto e^{-|s_i|} \quad \leftrightarrow \quad z_i = \operatorname{sign}(s_i) \quad (79)$$

$$\text{[CAUCHY]} \quad P(s_i) \propto \frac{1}{1+s_i^2} \quad \leftrightarrow \quad z_i = \frac{2|s_i|}{1+s_i^2} \quad (80)$$

$$\text{[GAUSS]} \quad P(s_i) \propto e^{-s_i^2}2 \quad \leftrightarrow \quad z_i = |s_i| \quad (81)$$

Algorithm summary/procedure.

1. Initialize square matrix A.
2. Until A converges, do:
   - Compute source vector via $\boldsymbol{s} = \mathbf{A}^{-1}\boldsymbol{x}$.
   - Compute

$$\begin{aligned} \boldsymbol{z} &= \nabla_s C(\boldsymbol{s}) && (82) \\ &= \nabla_s \left[-\log(p(\boldsymbol{s})) \right] && (83) \\ &= -\sum_i \nabla_s \log(p_s(s_i)) && (84) \end{aligned}$$

   - Update

$$\Delta A \propto A \left\langle \boldsymbol{z}(\boldsymbol{s})\boldsymbol{s}^T \right\rangle - A \qquad (85)$$

ICA - Andrew Ng - CS 229

Cocktail Party. There are n people talking at a cocktail party, and we've placed n microphones in the room to see if we can separate out the original n speakers' speech signals. We observe

$$x = As \qquad (86)$$

→ $x^{(i)}$ denotes the n-dimensional vector of our microphone recordings at time i.
→ $s^{(i)}$ denotes the n-dimensional vector of each speakers' output at time i.
→ A be the unknown square **mixing matrix**. **Goal**: Find the *unmixing matrix* $W = A^{-1}$. Then we can recover the generated sources via $s^{(i)} = Wx^{(i)}$.

$$W = \begin{bmatrix} - & w_1^T & - \\ - & w_2^T & - \\ & \vdots & \\ - & w_n^T & - \end{bmatrix} \qquad \longrightarrow \qquad s_j^{(i)} = w_j^T x^{(i)} \qquad (87)$$

[ELI5] "People say stuff $s^{(i)}$, but it gets all mixed up so we hear stuff $x^{(i)}$. We want to know how they related. Luckily, we can just unmix the mixed stuff."

Ambiguities. The (1) order and (2) scaling of the n "w_i" vectors in W is ambiguous. The sign of $s_j^{(}i)$ is irrelevant (sounds the same on a speaker). **Key point**: The aforementioned ambiguities are the *only* ambiguities, so long as the sources s_i are *non-Gaussian*. Basically, this is due to things like rotational invariance that can be present in Gaussians.

ICA Algorithm.

1. Let each source s_i have density $p_s(s_i)$. Then the joint distribution $p(s)$ is the probability of hearing all independent sources s_i. From previous results, we know that this implies a density for $x = As = W^{-1}s$ (below).

$$p(s) = \prod_{i=1}^{n} p_s(s_i) \qquad \Longrightarrow \qquad p(x) = \prod_{i=1}^{n} p_s(w_i^T x) \cdot |W| \tag{88}$$

Final Project

Contents

Final Project **Fall 2016**

WYGIWYS - A Visual Markup Decompiler: December 3

Table of Contents Local *Written by Brandon McKinzie*

Inputs/Outputs.

- **Input:** $\boldsymbol{x} \in \mathcal{X}$, consists of an image e.g. $\mathbb{R}^{H \times W}$ for grayscale images.
- **Output:** $\boldsymbol{y} \in \mathcal{Y}$, where $\boldsymbol{y} = \langle y_1, \ldots, y_C \rangle$ contains C tokens in the markup language.
- **Example**: Below, should we feed the input as the image of the equation on the left, the output would be the vector on the right.

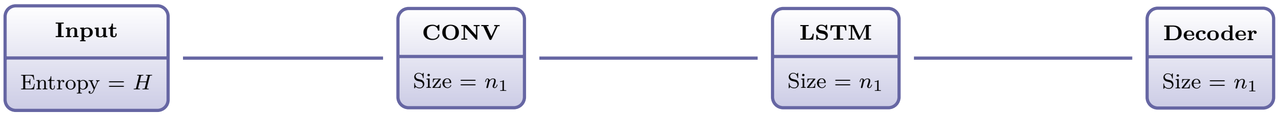
$$\rho = \textstyle\sum_{\alpha > 0} \alpha \qquad \langle rho, \ \ =, \ sum, \ alpha, \ \ >, \ 0, \ alpha \rangle$$

At training time, we assume we are given a sequence of input images $\boldsymbol{x}$ and ground-truth LaTeXlabels $\boldsymbol{y}$

$$\left((\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \ldots, (\boldsymbol{x}^{(J)}, \boldsymbol{y}^{(J)}) \right)$$

and at test time, given raw input image $\boldsymbol{x}$, we predict its corresponding LaTeXsource code, and output the $\hat{x}$ after compiling our prediction, then comparing $\hat{x}$ with x.

The Model.


Input
Entropy = H
CONV
Size = n_1
LSTM
Size = n_1
Decoder
Size = n_1