UNSUPERVISED LEARNING: HEBBIAN LEARNING & AE

INT301 Bio-computation, Week 11, 2021

Introduction

- So far, we mainly studied neural networks that learn from their environment in a **supervised manner**
- Neural networks can also learn in an unsupervised manner as well
- Unsupervised learning **discovers** significant features or patterns in the input data ***through general rules*** that operate locally
- Unsupervised learning networks typically consist of feed-forward connections and elements to facilitate ‘local’ learning

Hebbian Learning

- A simple principle was proposed by Hebb in 1949 in the context of biological neurons
- Hebbian principle

When a neuron repeatedly excites another neuron, then the threshold of the latter neuron is decreased, or the ***synaptic weight between the neurons is increased***, in effect increasing the likelihood of the second neuron to excite

- Hebbian learning rule $\Delta w_{ji} = \eta y_j x_i$
  - There is no desired or target signal required in the Hebb rule, hence it is ***unsupervised learning***

Hebbian Learning

- Consider the update of a single weight **w** (x and y are the pre- and post-synaptic activities)

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta x(n) y(n)$$

- For a linear activation function

$$\mathbf{w}(n+1) = \mathbf{w}(n)[1 + \eta x(n) x^T(n)]$$

  - **Weights increase without bounds.** If initial weight is negative, then it will increase in the negative. If it is positive, then it will increase in the positive range

- Hebbian learning is intrinsically unstable, unlike error-correction learning with BP algorithm

Hebbian Learning

- Consider a single linear neuron with *p* inputs

$$y = \mathbf{w}^T\mathbf{x} = \mathbf{x}^T\mathbf{w}$$

  and

$$\Delta\mathbf{w} = \eta[x_1 y \;\; x_2 y \;\; \ldots \;\; x_p y]^T$$

- The dot product can be written as

$$y = |\mathbf{w}||\mathbf{x}| \cos(\alpha)$$

  - α = angle between vectors **x** and **w**
    - If α approaches 0 (**x** and **w** are ‘close’), y is large
    - If α approaches 90 (**x** and **w** are ‘far’), y is zero

Hebbian Learning

- A network trained with Hebbian learning creates a similarity measure (inner product) in its input space according to the information contained in the weights
  - The weights capture (memorizes) the information in the data during training
- During operation, when the weights are fixed, a large output y signifies that the present input is "similar" to the inputs **x** that created the weights during training

Oja's Rule

- The simple Hebbian rule causes the weights to increase (or decrease) without bounds
- The weights need to be normalized to one as

$$\boxed{w_{ji}(n+1) = \frac{w_{ji}(n) + \eta x_i(n) y_j(n)}{\sqrt{\sum_i [w_{ji}(n) + \eta x_i(n) y_j(n)]^2}}}$$

- Oja proves that, for small η<<1, the above normalization can be approximated as:

$$\boxed{w_{ji}(n+1) = w_{ji}(n) + \eta y_j(n)[x_i(n) - y_j(n) w_{ji}(n)]}$$

  - This is Oja's rule, or the normalized Hebbian rule
  - It involves a '**forgetting term**' that prevents the weights from growing without bounds

Oja's Rule

- It has been proved that /using Lyapunov function analysis, Oja's rule converges asymptotically, unlike Hebbian rule which is unstable

- Oja's rule creates a ***principal component*** in the input space as the weight vector when applied to a single neuron

- How can we find other components in the input space with significant variance?

Recall: Dimensionality Reduction

- One approach to deal with high dimensional data is by reducing their dimensionality.
- Project high dimensional data onto a lower dimensional sub-space using linear or non-linear transformations.

$$\mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix} \xrightarrow[\textbf{?}]{\textbf{Reduce dimensionality}} \mathrm{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_k \end{bmatrix} \quad (K<<N)$$

Oja’s Rule

- How to find the projection onto orthogonal direction?
  - **Deflation method**: subtract the principal component from the input

- Oja’s rule can be extended to extract multiple principal components

Oja's Rule

- Deflation procedure is adopted to compute the other eigenvectors
  - Assume that the first component is already obtained, compute the projection of the first eigenvector on the input

$$y = w_1^T x$$

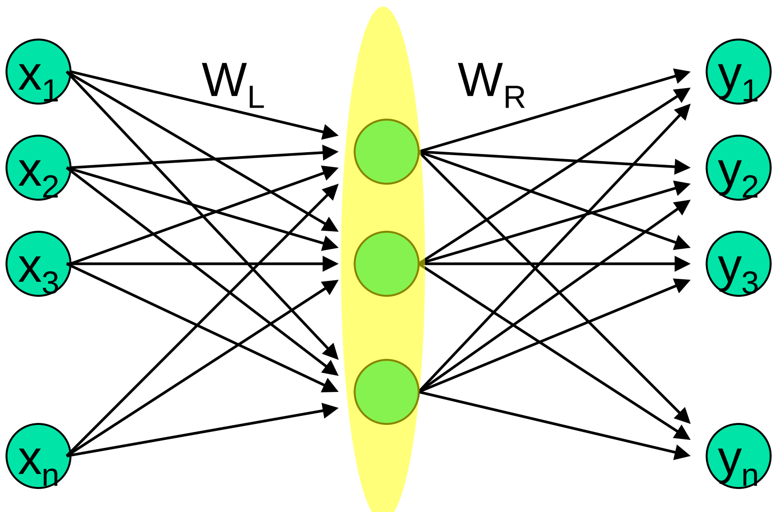
  - Generate the modified input as

$$\hat{x} = x - w_1 y = x - w_1 w_1^T x$$

  - Repeat Oja's rule on the modified data

PCA in Neural Networks

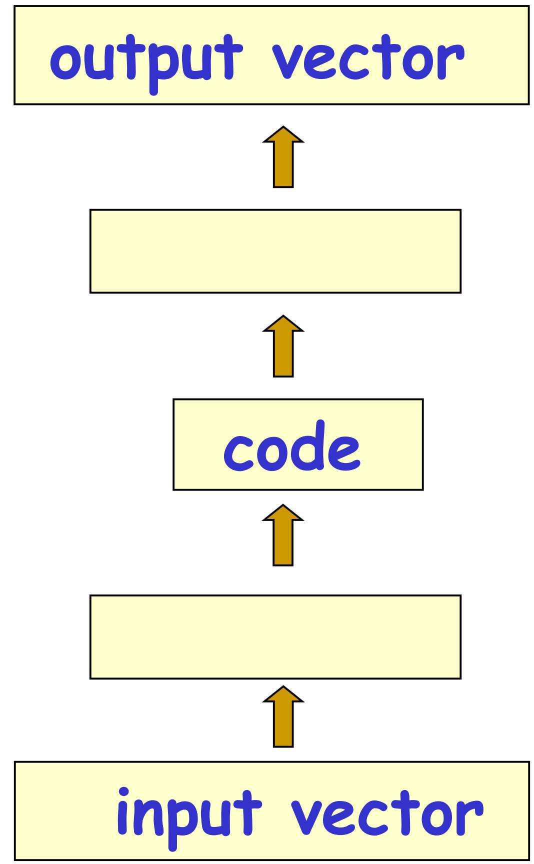
- **Multi-layer networks** with bottleneck layer



- Train using auto-associative output: $e = x – y$
- W_L spans the subspace of the first m principal eigenvectors.

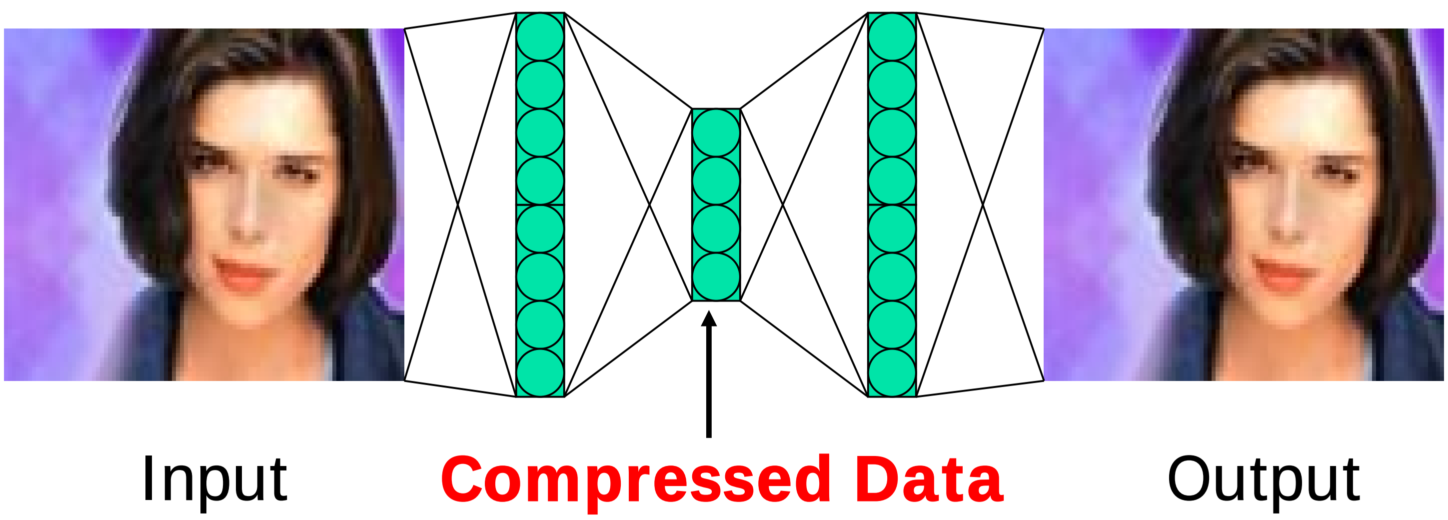
PCA in Neural Networks

- Using back-propagation for unsupervised learning
- Try to make the output be the same as the input in a network with a central bottleneck.
  - The activities of the hidden units in the bottleneck form an **efficient code**.
    - **The bottleneck does not have room for redundant features.**
  - Good for extracting independent features

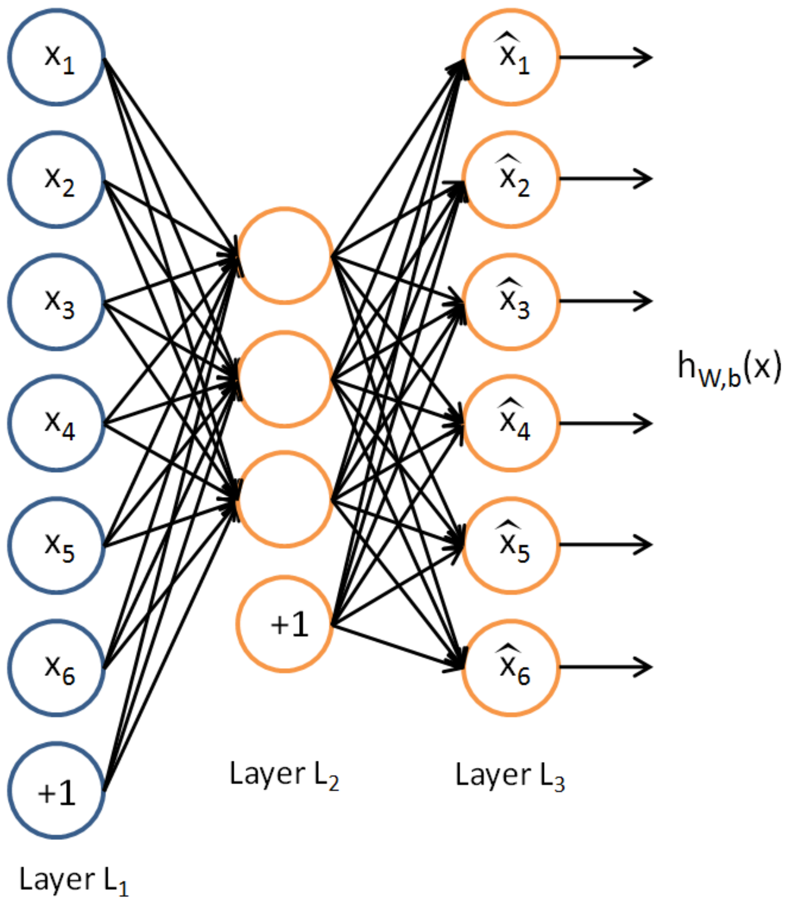


PCA in Neural Networks

- **Back-propagation algorithm can be used for unsupervised learning to discover significant features that characterise input patterns.**
- This can be achieved by learning the identity mapping, passing the data through a bottleneck: **auto-encoders**



Auto-encoders (Rumelhart 86)



An Autoencoder is a feedforward neural network that learns to predict the input itself in the output.

$$y^{(i)} = x^{(i)}$$

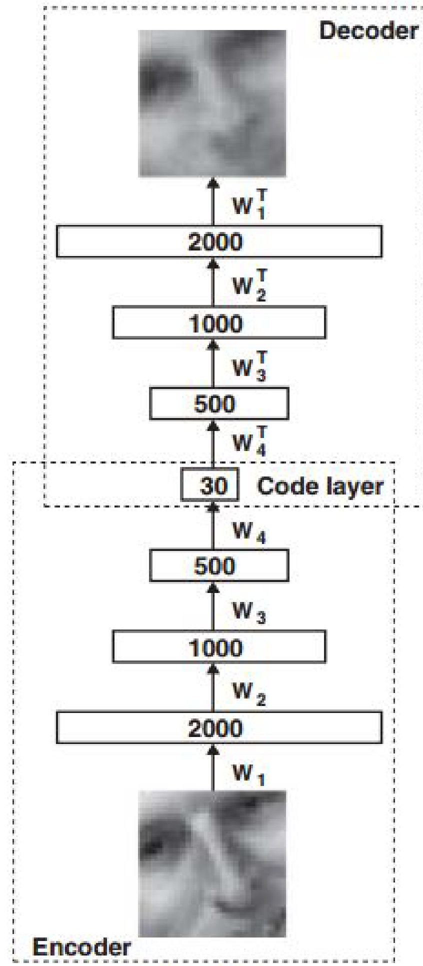
- The input-to-hidden part corresponds to an encoder
- The hidden-to-output part corresponds to a decoder.

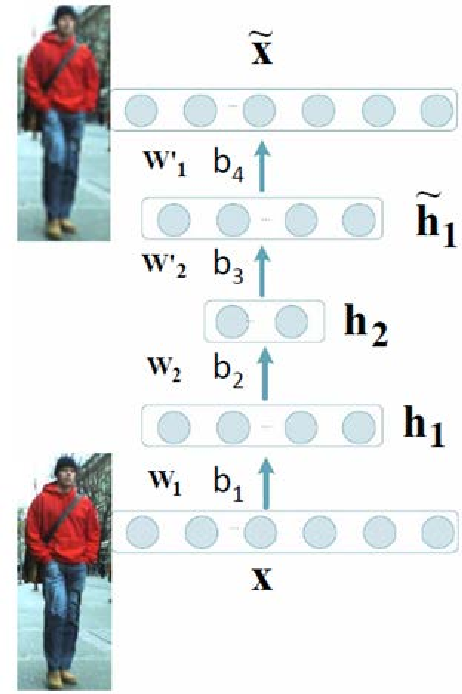
Auto-encoders (Rumelhart 86)

- To reproduce the input patterns at output layer
- Number of hidden layers and the sizes of the layers can vary
- Auto-encoder tends to find a data description which resembles the **PCA**; while small number of neurons in the bottleneck layer of the diabolo network acts as an ***information compressor*** **(code)**

Deep Auto-encoder (Hinton 06)

- A deep auto-encoder is constructed by extending the encoder and decoder of autoencoder with multiple hidden layers.





Encoding: $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x}+b_1)$

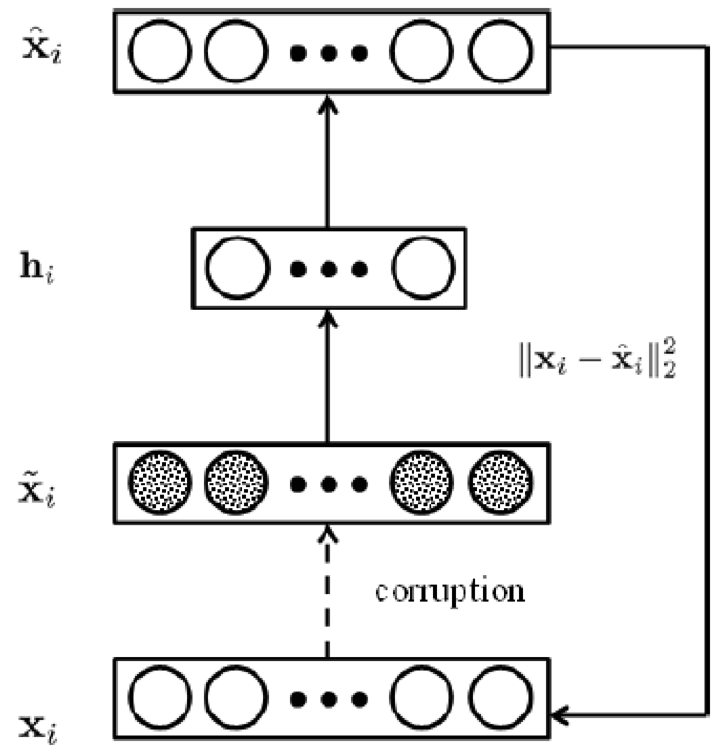
$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1+b_2)$

Decoding: $\tilde{\mathbf{h}}_1 = \sigma(\mathbf{W}'_2\mathbf{h}_2+b_3)$

$\tilde{\mathbf{x}} = \sigma(\mathbf{W}'_1\mathbf{h}_1+b_4)$

Denoising Auto-encoder (Vincent 08)

- By adding stochastic noise, it can force auto-encoder to learn more robust features.
- The loss function



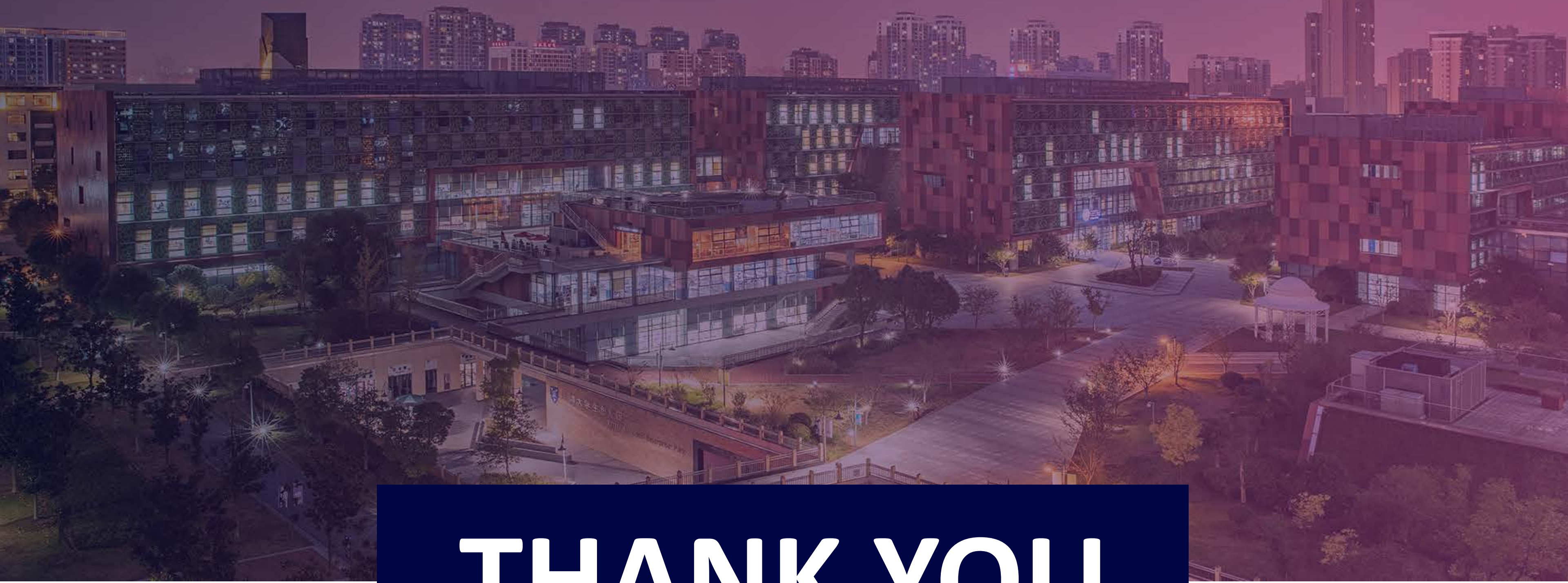
$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}_1 \tilde{\mathbf{x}}^{(\ell)} + \mathbf{b}_1)$$

$$\hat{\mathbf{x}}^{(\ell)} = \sigma(\mathbf{W}_2 \mathbf{h}^{(\ell)} + \mathbf{b}_2)$$

$$\min_{\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2} \sum_{\ell} \|\mathbf{x}^{(\ell)} - \hat{\mathbf{x}}^{(\ell)}\|_2^2 + \lambda \left(\|\mathbf{W}_1\|_F^2 + \|\mathbf{W}_2\|_F^2 \right)$$

Auto-encoders Network

- The network tries to reproduce the input in the output, inducing an short encoding in the hidden layer.
- This encoding retains the maximum amount of information about the input in a smaller dimensional space such that the input can be reconstructed.
- Auto-encoder networks can be used for dimensionality reduction, compression, etc.



THANK YOU

VISIT US

WWW.XJTLU.EDU.CN

FOLLOW US

@XJTLU

Xi'an Jiaotong-Liverpool University
西交利物浦大学