IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 3: Linear Layers

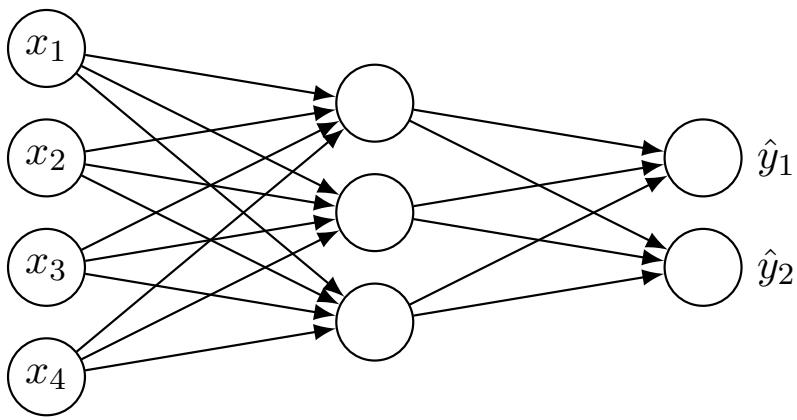
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2024-1

Recall: Supervised learning with FNNs

- Supervised learning
  - Learn a mapping from inputs $\boldsymbol{x}$ to outputs $\boldsymbol{y}$
  - Training set $\mathcal{D} = \{ (\boldsymbol{x}_i, \boldsymbol{y}_i) \}_{i=1}^{N}$ of input-output pairs
  - With FNNs: for input $\boldsymbol{x}_i$, we want output $\hat{\boldsymbol{y}}_i$ "close" to $\boldsymbol{y}_i$
  - Learning means adjusting the weights such that the FNN does this
- FNNs are discriminative
  - Given an input $\boldsymbol{x}$, they compute an output $\hat{\boldsymbol{y}}$
  - But they don't allow going from outputs to inputs
- Hidden layer outputs are inputs of the next layer
  - We may also think of hidden layers as features for the next layer
  - These features are not provided upfront, but learned

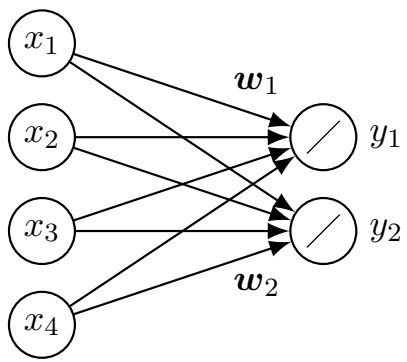


Linear layers

- Layers in which all layer inputs are connected with all layer outputs are called **dense layers** or **fully-connected layers**
- A dense **linear layer** is a layer consisting of only linear neurons
  - n layer inputs ($\boldsymbol{x} \in \mathbb{R}^n$), m layer outputs ($\boldsymbol{y} \in \mathbb{R}^m$)
  - Parameterized by weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m \in \mathbb{R}^n$
  - Optionally: biases $b_1, \ldots, b_m \in \mathbb{R}$
- Outputs given by

$$y_j = \sum_i [\boldsymbol{w}_j]_i x_i + b_j = \langle \boldsymbol{w}_j, \boldsymbol{x} \rangle + b_j$$

- Example: $n = 4$, $m = 2$, no bias



The action of a linear layer

- Without bias, we have: $y_j = \langle \boldsymbol{w}_j, \boldsymbol{x} \rangle$
- Let $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ a **weight matrix** in which the j-th column equals the weights $\boldsymbol{w}_j$ of the j-th layer output

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \dots & \boldsymbol{w}_m \end{pmatrix}$$

- Then: $\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x}$
  - **Linear layers compute a matrix-vector product**
- For our example, $\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 \end{pmatrix} \in \mathbb{R}^{4 \times 2}$ and

$$\boldsymbol{W}^\top \boldsymbol{x} = \begin{pmatrix} \boldsymbol{w}_1^\top \\ \boldsymbol{w}_2^\top \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} \langle \boldsymbol{w}_1, \boldsymbol{x} \rangle \\ \langle \boldsymbol{w}_2, \boldsymbol{x} \rangle \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \boldsymbol{y}$$

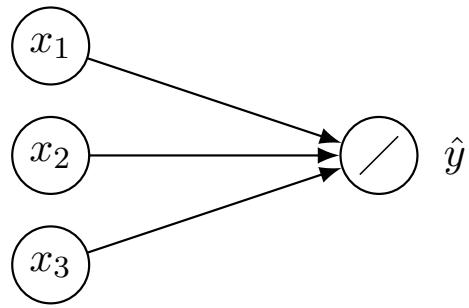
Using linear layers

- Typical uses of linear layers
  - As an output layer for regression tasks
  - As a hidden layer to perform dimensionality reduction ($m < n$) (in ML somewhat confusingly called **linear projection**)
  - Likewise, as a hidden layer to increase dimensionality ($m > n$)
- Number of parameters: nm (without bias), $nm + m$ (with bias)

n	m	# parameters	
64	64	4,096	
128	128	16,384	
256	256	65,536	
512	512	262,144	
1,024	1,024	1,048,576	
768	3,072	2,359,296	(T5-Base dense layer, dim up)
3,072	768	2,359,296	(T5-Base dense layer, dim down)

Linear regression as FNN (1)

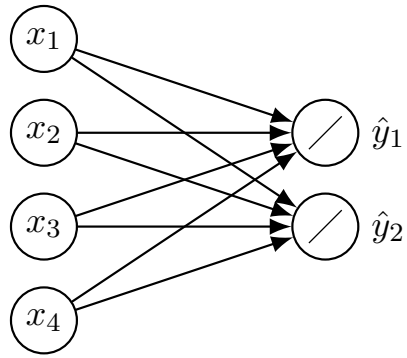
- In a **linear FNN**, all neurons/layers are linear
- Simplest linear FNN: single linear layer with one output



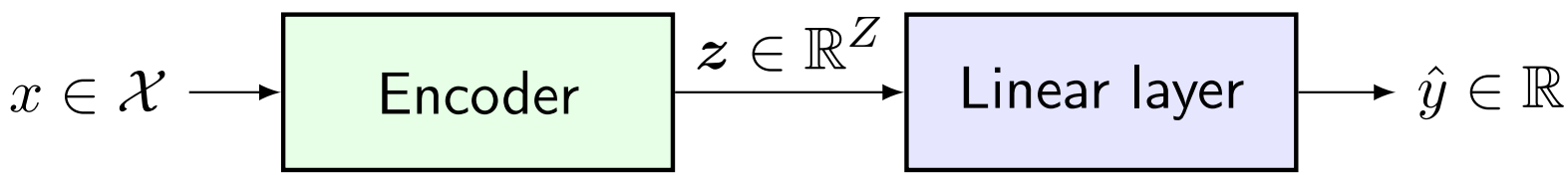
- Output $\hat{y} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is linear in input $\boldsymbol{x} \rightarrow$ linear model
- Suppose we train this network using ERM with squared loss
  - Empirical risk is $\frac{1}{N} \sum_i (y_i - \hat{y}_i)^2 \rightarrow$ minimize
  - We obtain ordinary least squares (OLS) estimate for linear regression
- Suppose we train with MLE assuming i.i.d. normal errors
  - I.e., assuming $y_i = \langle \boldsymbol{w}^*, \boldsymbol{x}_i \rangle + b^* + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
  - Likelihood $\prod_i \mathcal{N}(y_i | \hat{y}_i, \sigma^2) \rightarrow$ maximize
  - Recall: solution is OLS estimator

Linear regression as FNN (2)

- With multiple outputs, we obtain multiple linear regression

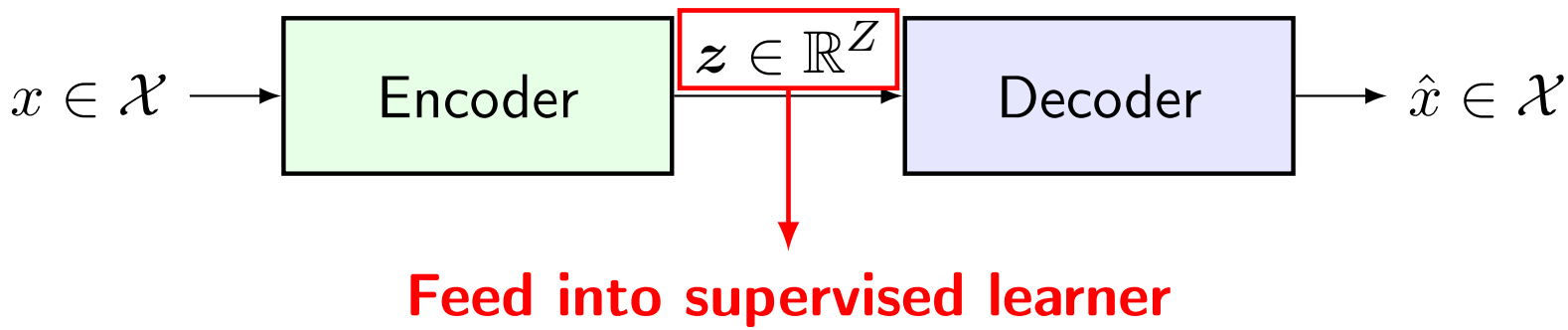


- **Linear FNN (w/o hidden layer) $\equiv$ linear regression**
  - To determine bias and weights, any suitable linear regression library can be used
- Outputs remain linear even with hidden layers ($\rightarrow$ exercise)
  $\rightarrow$ That's why we often want non-linearities
- For regression problems, linear layers often used as output layer



Autoencoders

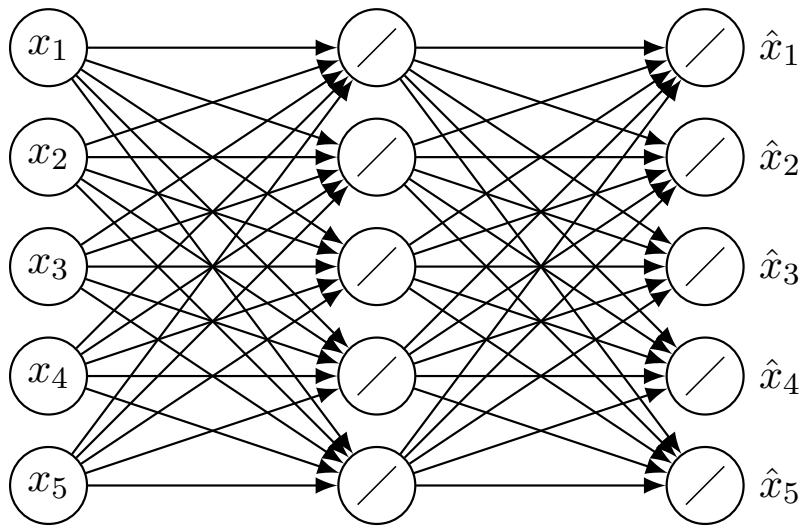
- FNNs are useful for unsupervised learning as well
  - We are given an unlabeled dataset $\mathcal{D} = \{ \boldsymbol{x}_i \}_{i=1}^N$ with $\boldsymbol{x}_i \in \mathbb{R}^D$
  - We don't have outputs
  - We want to find structure, or patterns, or reduce dimensionality
- Idea: train FNN to predict its input, i.e., set $\boldsymbol{y}_i = \boldsymbol{x}_i$
  - The resulting FNN is called an **autoencoder**



- Why autoencoders?
  - Autoencoders are a technique to **learn embeddings** ($\boldsymbol{z}$)
  - E.g., semi-supervised learning: train autoencoder on all inputs (labeled+unlabeled), use embeddings for supervised learner (labeled)
  - E.g., clustering: use embeddings as inputs to, say, K-means
  - E.g., denoising: use $\hat{x}$ instead of x
  - E.g., visualization: visualize $\boldsymbol{z}$ (e.g., using $Z = 2$)

Linear autoencoders

- Linear FNNs can do more than what may be expected at first glance
- A **linear autoencoder** uses only linear layers (in both encoder and decoder)
- A simple (but useless) linear autoencoder

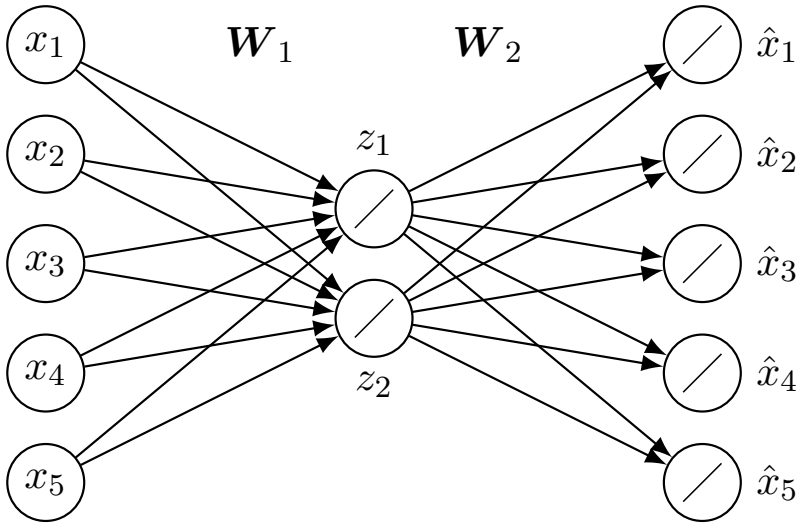


Input layer ($\boldsymbol{x}$) Hidden layer ($\boldsymbol{z}$) Output layer ($\hat{\boldsymbol{x}}$)

- Can you figure out the optimal weight matrices (such that $\hat{x}_j = x_j$)?

Bottlenecks

- Consider a linear autoencoder with $\boldsymbol{x} \in \mathbb{R}^D$ and one hidden layer with $Z < D$ hidden neurons



  - Can you still figure out the optimal weight matrices?
- A layer with few neurons is referred to as a **bottleneck**
  - I.e., fewer neurons than the surrounding layers
  - Forces FNN to "compress" information $\rightarrow$ dimensionality reduction
  - FNNs with bottlenecks *learn* how to compress
- Since autoencoder needs to **reconstruct** all inputs well, the optimal "compression" depends on all training inputs
  - E.g., above: 5D data ($\boldsymbol{x}$) compressed into a 2D representation ($\boldsymbol{z}$)

Obtaining optimal weights

- We have $\boldsymbol{z} = \boldsymbol{W}_1^\top \boldsymbol{x}$ and $\hat{\boldsymbol{x}} = \boldsymbol{W}_2^\top \boldsymbol{z} = \boldsymbol{W}_2^\top \boldsymbol{W}_1^\top \boldsymbol{x}$
- For squared error, solve $\operatorname{argmin}_{\boldsymbol{W}_1, \boldsymbol{W}_2} \left[\sum_i \sum_j (x_{ij} - \hat{x}_{ij})^2 \right]$
- The solution can be read off the **singular value decomposition** (SVD) of $\boldsymbol{X}$ (covered in ML lecture)
  - Let $\boldsymbol{X}$ be the design matrix and $\boldsymbol{U}_Z \boldsymbol{\Sigma}_Z \boldsymbol{V}_Z^\top$ its size-Z truncated SVD
  - $\boldsymbol{U}_Z$ is an $N \times Z$ matrix with the first Z left-singular vectors of $\boldsymbol{X}$
  - $\boldsymbol{V}_Z$ is an $D \times Z$ matrix with the first Z right-singular vectors of $\boldsymbol{X}$
  - $\boldsymbol{\Sigma}_Z$ is an $Z \times Z$ matrix with the first Z singular values of $\boldsymbol{X}$
  - An optimal solution is $\boldsymbol{W}_1 = \boldsymbol{V}_Z$ and $\boldsymbol{W}_2 = \boldsymbol{V}_Z^\top$
  - For this solution, $\boldsymbol{z}_i^\top = [\boldsymbol{U}_Z]_{i:} \boldsymbol{\Sigma}_Z$
- This is closely related to **principal component analysis** (PCA)
  - Main difference: PCA centers the data so that each feature has mean 0 (sometimes: also normalize each feature)
  - Then $\boldsymbol{W}_1$ contains the first Z principal components as its columns
  - And $\boldsymbol{z}_i$ contains the PCA scores for $\boldsymbol{x}_i$

Example: Weather data

$\boldsymbol{X}$	Jan	Apr	Jul	Oct	Year
Stockholm	-0.70	8.60	21.90	9.90	10.00
Minsk	-2.10	12.20	23.60	10.20	10.60
London	7.90	13.30	22.80	15.20	14.80
Budapest	1.20	16.30	26.50	16.10	15.00
Paris	6.90	14.70	24.40	15.80	15.50
Bucharests	1.50	18.00	28.80	18.00	16.50
Barcelona	12.40	17.60	27.50	21.50	20.00
Rome	11.90	17.70	30.30	21.40	20.40
Lisbon	14.80	19.80	27.90	22.50	21.50
Athens	12.90	20.30	32.60	23.10	22.30
Valencia	16.10	20.20	29.10	23.60	22.30
Malta	16.10	20.00	31.50	25.20	23.20

Example: Weights and representation

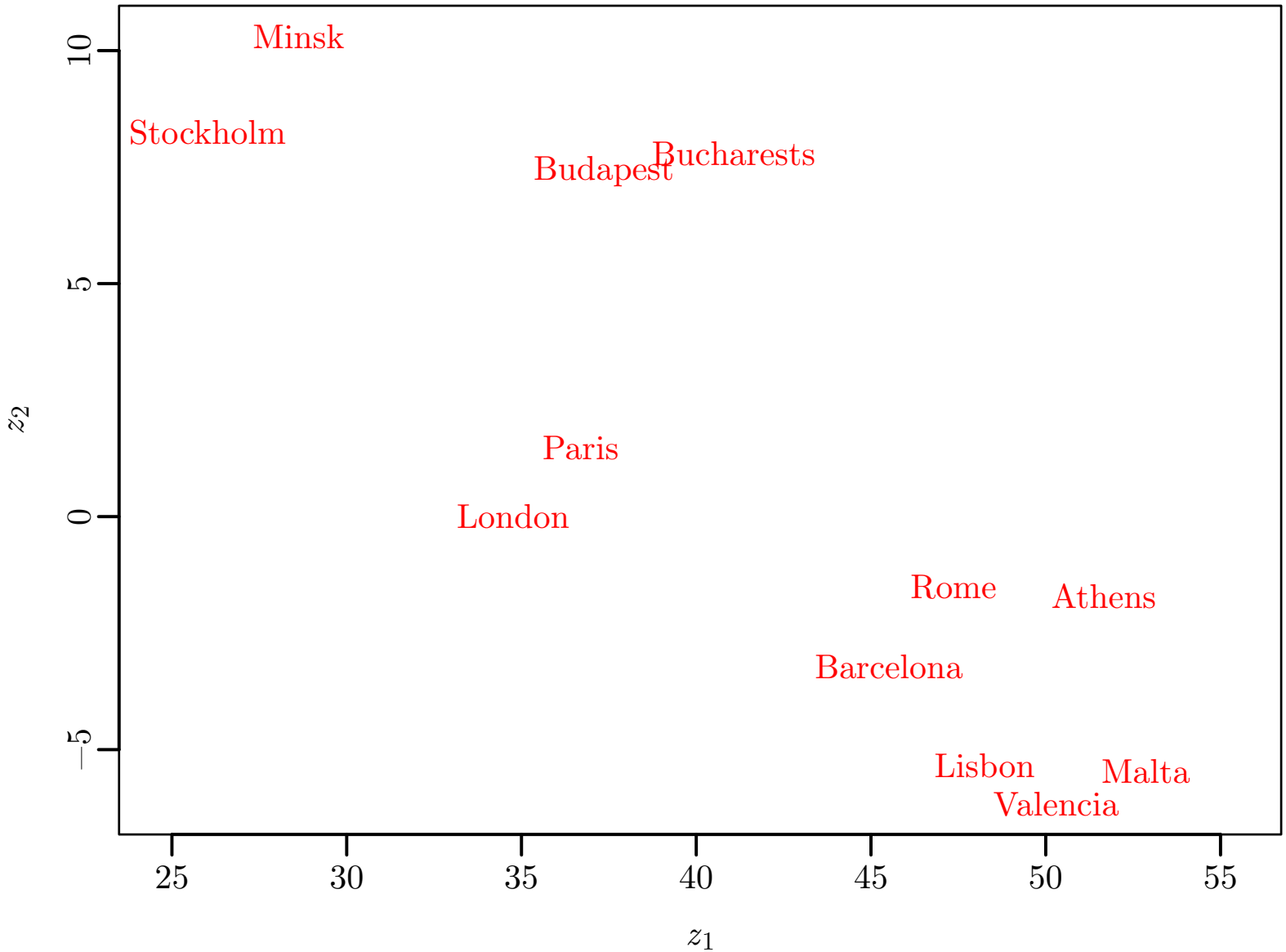
$\boldsymbol{W}_1$	1	2
Jan	0.22	-0.85
Apr	0.40	0.06
Jul	0.64	0.47
Oct	0.45	-0.18
Year	0.43	-0.14

$\boldsymbol{W}_2$	$\widehat{\text{Jan}}$	$\widehat{\text{Apr}}$	$\widehat{\text{Jul}}$	$\widehat{\text{Oct}}$	$\widehat{\text{Year}}$
1	0.22	0.40	0.64	0.45	0.43
2	-0.85	0.06	0.47	-0.18	-0.14

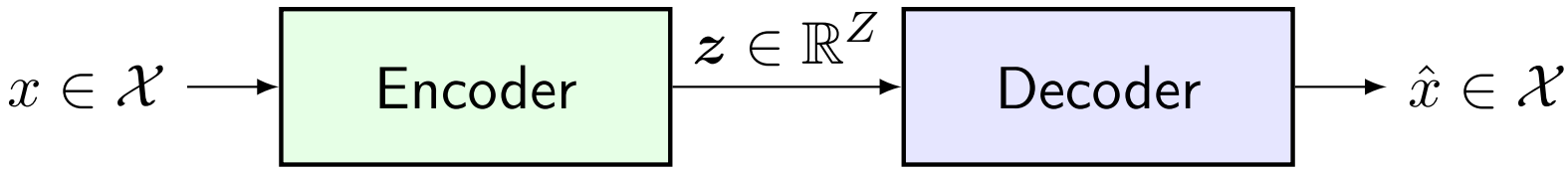
$\boldsymbol{Z}$	1	2
Stockholm	26.02	8.25
Minsk	28.63	10.30
London	34.76	0.00
Budapest	37.36	7.42
Paris	36.69	1.48
Bucharests	41.07	7.79
Barcelona	45.51	-3.22
Rome	47.36	-1.50
Lisbon	48.25	-5.34
Athens	51.66	-1.69
Valencia	50.30	-6.16
Malta	52.86	-5.45

Plot of representation

Bottlenecks of two neurons can be useful for visualization.



General autoencoders



- Encoder is a function (e.g., a FNN) that compresses input $\boldsymbol{x}$ to an embedding $\boldsymbol{z}$ (also called *code* or *distributed representation*)
- Decoder is a function (e.g., a FNN) that decompresses an embedding $\boldsymbol{z}$ to obtain reconstruction $\hat{\boldsymbol{x}}$
  - Think: approximate "inverse" of encoder
  - Decoder may be a "reversed" architecture of the encoder (e.g., layers in reverse order but with different weights)
  - Decoder may be an entirely different network
- Simplest way to train autoencoder is to use data points $\boldsymbol{x}$ as both input and reconstruction target

Example: Representing documents

804414 newswire stories, inputs = per-document rel. frequencies of 2000 most common word stems, autoencoder = logistic hidden units + linear output units

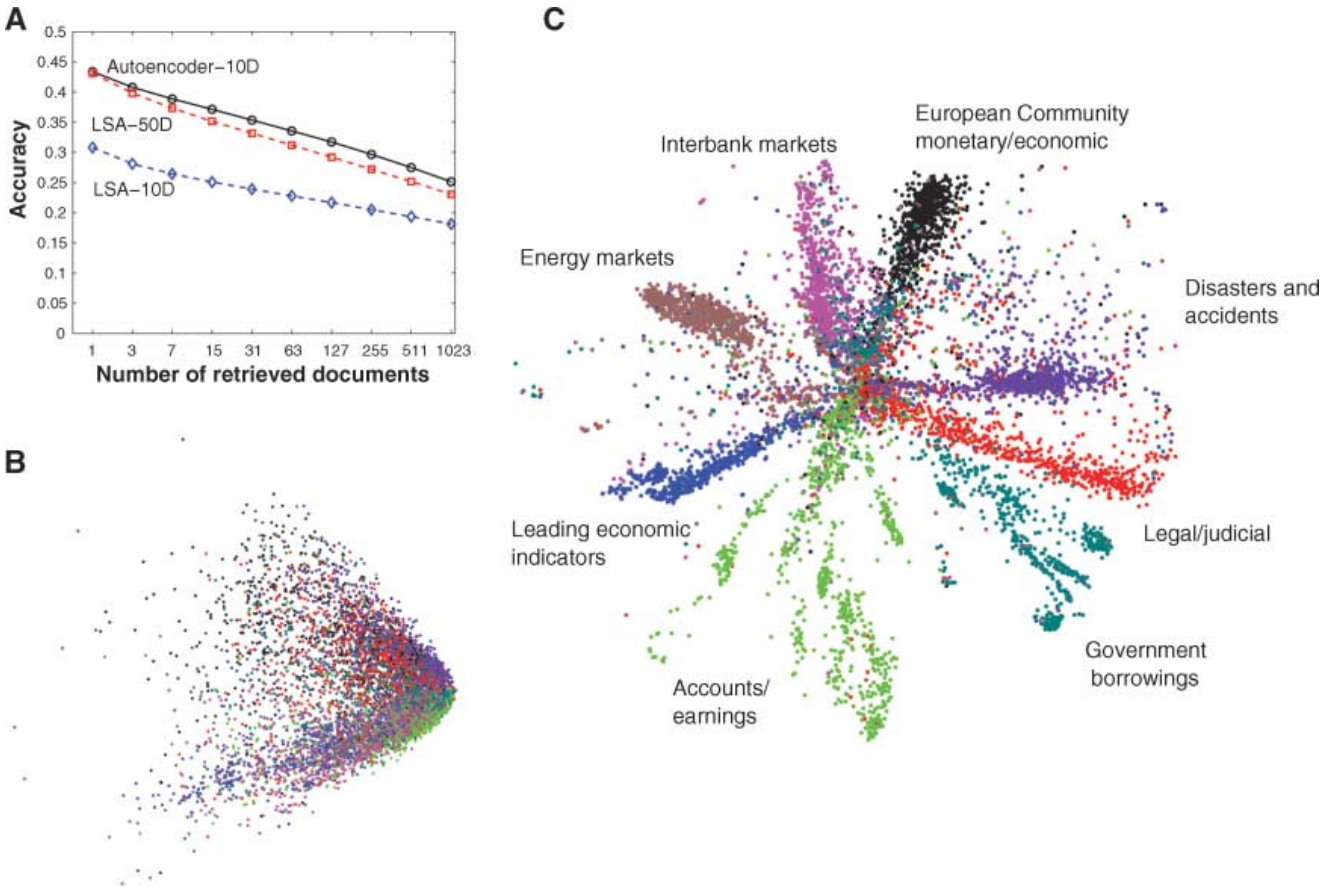


Fig. 4. (**A**) The fraction of retrieved documents in the same class as the query when a query document from the test set is used to retrieve other test set documents, averaged over all 402,207 possible queries. (**B**) The codes produced by two-dimensional LSA. (**C**) The codes produced by a 2000-500-250-125-2 autoencoder.

Hinton and Salakhutdinov, 2006

Discussion (autoencoders)

- Autoencoders are a form of **representation learning**
- Autoencoders are an example of **unsupervised pre-training**
  - I.e., learn (parts of) the weights of a network without supervision
- Many variants exists; e.g.,
  - Architecture of encoder/decoder
  - Choice of cost function
  - Construction of inputs and outputs for learning
  - Constraints on embeddings
- Examples
  - **Denoising autoencoders** perturb the input $\boldsymbol{x}$ with noise to obtain $\tilde{\boldsymbol{x}}$, and then aim to reconstruct the original input $\boldsymbol{x}$ from $\tilde{\boldsymbol{x}}$ $\rightarrow$ noise robustness
  - **Variational autoencoders** force $\boldsymbol{z}$ to follow a specified simple distribution (e.g., diagonal Gaussian) $\rightarrow$ generative model
  - **Sparse autoencoders** force $\boldsymbol{z}$ to be sparse $\rightarrow$ sparse representations