ECE 1508S2: Applied Deep Learning

Chapter 8: Auto Encoders

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2025

Road-map: *Auto Encoders*

In this chapter, we get to know *Auto Encoders* which are very powerful architectures for feature extraction and data generation: *this chapter is just an introduction, and we will see more details in the Generative AI course*

- *There's going to be an ECE course on Generative AI*
  - ↳ *We start from this point there!*

The best way to understand Auto Encoders is to look at a simple representation problem, but before that let's make an agreement

$$\textit{Auto Encoders} \equiv \textit{Autoencoders} \equiv \textit{AEs}$$

from now on!

Simple Problem: *Representation in Smaller Dimension*

Consider a simple problem: *we have a dataset of two-dimensional data-points*

$$\mathbb{D} = \{\mathbf{x}_b : b = 1, \dots, B\}$$

where $\mathbf{x}_b \in \mathbb{R}^2$, and we want to find two weight vectors

- *A weight vector $\mathbf{w} \in \mathbb{R}^2$ that compresses $\mathbf{x}_b$ into a single number z_b as*

$$z_b = \mathbf{w}^\mathsf{T} \mathbf{x}_b$$

- *A weight vector $\hat{\mathbf{w}}$ that decompresses $\mathbf{x}_b$ from z_b*

$$\hat{\mathbf{x}}_b = \hat{\mathbf{w}} z_b$$

+ *Can we do this?*

– Well! Not always!

Simple Problem: *Representation in Smaller Dimension*

Let's try our ML knowledge: *say we set the the compression vector to* $\mathbf{w}$*; then, the compressed version of the dataset is*

$$\begin{bmatrix} z_1 & \dots & z_B \end{bmatrix} = \begin{bmatrix} \mathbf{w}^\mathsf{T}\mathbf{x}_1 & \dots & \mathbf{w}^\mathsf{T}\mathbf{x}_B \end{bmatrix} = \mathbf{w}^\mathsf{T} \underbrace{\begin{bmatrix} \mathbf{x}_1 & \dots & \mathbf{x}_B \end{bmatrix}}_{\mathbf{X} \in \mathbb{R}^{2\times B}} = \mathbf{w}^\mathsf{T}\mathbf{X}$$

Now, let's find the decompressed versions using $\hat{\mathbf{w}}$

$$\underbrace{\begin{bmatrix} \hat{\mathbf{x}}_1 & \dots & \hat{\mathbf{x}}_B \end{bmatrix}}_{\hat{\mathbf{X}} \in \mathbb{R}^{2\times B}} = \begin{bmatrix} \hat{\mathbf{w}} z_1 & \dots & \hat{\mathbf{w}} z_B \end{bmatrix} = \hat{\mathbf{w}} \begin{bmatrix} z_1 & \dots & z_B \end{bmatrix} = \hat{\mathbf{w}}\mathbf{w}^\mathsf{T}\mathbf{X}$$

After decompression, we get $\hat{\mathbf{X}} = \hat{\mathbf{w}}\mathbf{w}^\mathsf{T}\mathbf{X}$ *which we want to be the same as* $\mathbf{X}$

Simple Problem: *Representation in Smaller Dimension*

Let's try our ML knowledge: *we could define a loss and minimize it via SGD*

$$\hat{R} = \mathcal{L}(\hat{\mathbf{X}}, \mathbf{X}) = \|\hat{\mathbf{w}}\mathbf{w}^\mathsf{T}\mathbf{X} - \mathbf{X}\|^2$$

But, we actually do not need to go that far! We decompress $\hat{\mathbf{X}}$ *simply if*

$$\hat{\mathbf{w}}\mathbf{w}^\mathsf{T}\mathbf{X} \overset{!}{=} \mathbf{X}$$

Or alternatively if we have $\hat{\mathbf{w}}\mathbf{w}^\mathsf{T}$ *to be an identity transform*

$$\hat{\mathbf{w}}\mathbf{w}^\mathsf{T} \overset{!}{=} \mathbf{I}$$

+ *But it's impossible!*
– Yes! $\hat{\mathbf{w}}\mathbf{w}^\mathsf{T}$ can never be $\mathbf{I}$ *since* $\hat{\mathbf{w}}\mathbf{w}^\mathsf{T}$ *is rank one and* $\mathbf{I}$ *is full-rank!*

Simple Problem: *Representation in Smaller Dimension*

Now, let's assume that we are given the following piece of information: *we know that every single point in the dataset is of the form*

$$\mathbf{x}_b = \begin{bmatrix} 2\alpha_b \\ \alpha_b \end{bmatrix}$$

for some real-valued α_b

it turns out we can now do compression and decompression successfully

---

Solution: *we compress by* $\mathbf{w} = [1, -1]^\mathsf{T}$ *and decompress by* $\hat{\mathbf{w}} = [2, 1]^\mathsf{T}$. *We can then write*

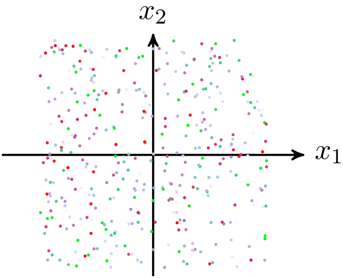
$$z_b = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 2\alpha_b \\ \alpha_b \end{bmatrix} = \alpha_b \rightsquigarrow \hat{\mathbf{x}}_b = \begin{bmatrix} 2 \\ 1 \end{bmatrix} z_b = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \alpha_b = \mathbf{x}_b$$

Simple Problem: *Principle Component*

+ *Wait a moment! Why did it happen? We don't have $\hat{\mathbf{w}}\mathbf{w}^\mathsf{T} = \mathbf{I}$!*
– Yes! It happened because we don't need $\hat{\mathbf{w}}\mathbf{w}^\mathsf{T} = \mathbf{I}$ in this case

---

A general dataset of 2-dimensional vectors look like



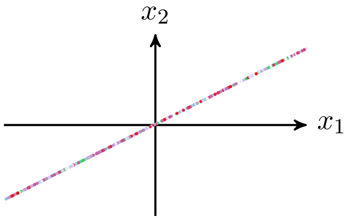
and this dataset cannot be projected on a single axis!

Simple Problem: *Principle Component*

We were successful when *we knew that every data-point is of the form*

$$\mathbf{x}_b = \begin{bmatrix} 2\alpha_b \\ \alpha_b \end{bmatrix}$$

In this case, the dataset is already on a single rotated axis



We just need to find the value of each point on that axis, i.e., α_b

Principle Component Analysis: *Dimensionality Reduction*

This is in fact *a very basic example of*

Principle Component Analysis $\equiv$ *PCA*

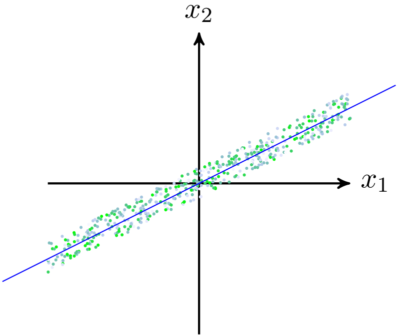
PCA: *Minimum Error Formulation*

In PCA, we have a dataset of N-dimensional data-points and learn a weight matrix $\mathbf{W} \in \mathbb{R}^{L \times N}$ that for each $\mathbf{x}_b$ in the dataset generates a latent variable $\mathbf{z}_b = \mathbf{W}\mathbf{x}_b$ such that by linear reconstruction of $\mathbf{x}_b$ from $\mathbf{z}_b$ has minimum error

+ *Do we always have such property in our dataset?!*
– Perfectly no, but approximately yes!

Principle Component Analysis: *Dimensionality Reduction*

We could in practice have compressible dataset with its main variation being on a reduced dimension, e.g., 2-dimensional points that lie around a single line

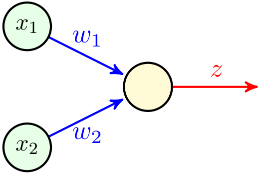


We could have a low-dimensional latent variable for each data-point

↳ *We accept a minor reconstruction error*

PCA as *Neural Network*

Since we like NNs, *let's represent our simple example as an NN architecture and look at its training*

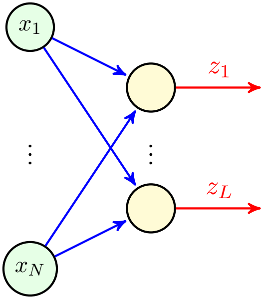


The above NN describe PCA

- *We have two learnable parameters w_1 and w_2*
- *We want to train this NN such that z is the best representation*
  - ↳ *z is a compression for $\mathbf{x}$*

PCA as *Neural Network*

In more general setting, we have N *inputs and* L *latent variables*

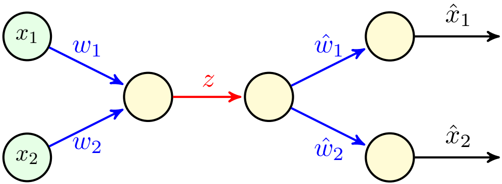


The above NN describes more generic PCA setting

- *We have* $(N+1)\,L$ *learnable parameters:* NL *weights and* L *biases*
- *We want to train this NN such that* $\mathbf{z}$ *is the best representation*
  - ↳ $\mathbf{z}$ *is low dimensional representation of* $\mathbf{x}$

PCA as *Neural Network*

+ *But how can we train this NN? We do not have any label!*
- This is because it's an *unsupervised learning problem*

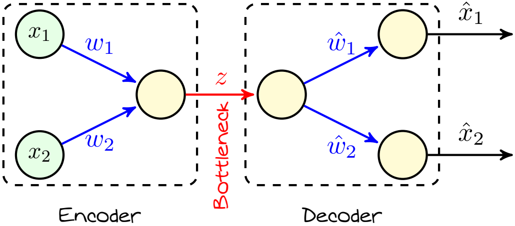


To train this model, we should make some labels

- *We intend to recover* $\mathbf{x}$ *from* z *at the end of the day*
  - ↳ *We can learn further the decompression*
- *We now have some labels*
  - ↳ *We train using the loss* $\hat{R} = \mathcal{L}(\mathbf{x}; \hat{\mathbf{x}})$

PCA as *Neural Network*

+ *This looks like something we had before!*
– Yes! It's an *encoder-decoder architecture whose label is the input*



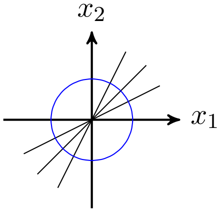
- *Encoder represents input in a lower dimension*
  ↳ *It somehow compresses the input*
- *Bottleneck contains the latent variables*
- *Decoder can return back the data from its low-dimensional representation*
  ↳ *It decompresses the latent variables*

Nonlinear Compression

Encoder and decoder are both *shallow* and *linear* in PCA: *we may however need deep nonlinear encoder and decoder*

Let's consider an example: *consider our initial simple example and now assume that data-points are of the form*

$$\mathbf{x} = \begin{bmatrix} \sin\theta \\ \cos\theta \end{bmatrix}$$

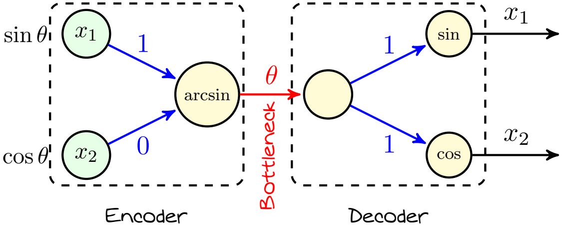


Obviously PCA fails to compress $\mathbf{x}$ *error-less into one dimension*

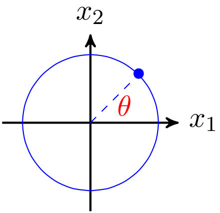
- *PCA searches for a line that represents the dataset*
  - ↳ *In general, it finds the line with minimum error*
- *A line with zero error does not exist in this problem*

Nonlinear Compression

We can however do this by a nonlinear representation

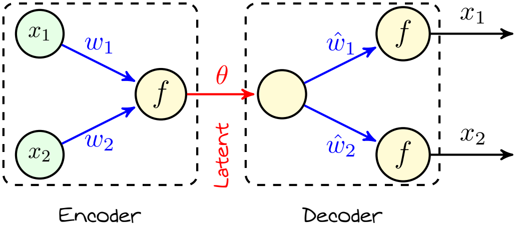


We just need to know the angle θ



Nonlinear PCA

We can use this NN to extract principle components of other datasets



Latent variable θ represents data in lower dimension for minimal recovery error