

Applied Deep Learning

Chapter 8: Representation and Generation

Ali Bereyhi

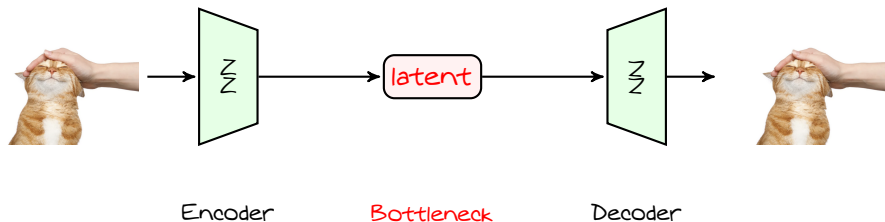
`ali.bereyhi@utoronto.ca`

Department of Electrical and Computer Engineering
University of Toronto

Winter 2026

Auto Encoder: Deep PCA with Encoder-Decoder

AE is in principle a **deep encoder-decoder** architecture used for **nonlinear PCA**



Vanilla AE finds a **latent space** that is **very smaller** in dimension

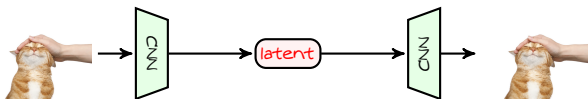
- Each data-point is **encoded** to its low-dimensional **latent representation**
 - ↳ **Latent representation** contains in fact the **principal features of data**
- **Latent representation** can re-generate the data-point via using **decoder**

Vanilla AE

Auto Encoder (AE)

AE is an **encoder-decoder** architecture whose **bottleneck feature**, also called **latent representation**, has **lower dimension** than the input and output of NN

We can implement encoder and decoder by simple NNs, e.g.,



Such an architecture is a **vanilla AE** mainly used for compression

- + Is compression so crucial that AEs become so important?
- Naive answer: **Yes!** Better answer: AEs can do **much more** than compression in fact!

Training AEs: General Approach

We typically use AEs in **unsupervised** settings: it means that we have no labels in the dataset

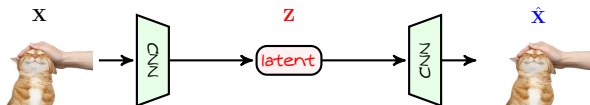
- In AE both **latent representation** and **decoded data** are **outputs**
- For training we need to compute loss between the **outputs** and a **reference**
 - ↳ We cannot compare the latent representation with any reference
 - ↳ We have **no true latent representation**
- We should **extract** some **reference** from our **dataset**
 - ↳ This reference depends on our **target application**
 - ↳ We are going to consider three types in this chapter

To go on with the training of AEs, let's keep the track of their applications

① Compression

- ↳ We intend to **compress** data into a lower-dimensional subspace
- ↳ For loss computation, we compare **decoded data** with **its ground truth**

Training AEs for Compression



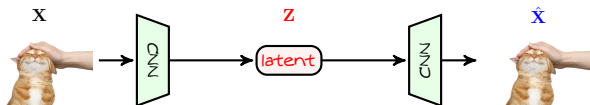
Let's name variables: say the input is \mathbf{X} , e.g., RGB image, *latent representation* is \mathbf{Z} , e.g., multi-channel tensor, and $\hat{\mathbf{X}}$ is *decoded output*, e.g., RGB image

- For compression we wish to recover $\hat{\mathbf{X}} = \mathbf{X}$
 - ↳ Loss is proportional to the difference between \mathbf{X} and $\hat{\mathbf{X}}$
- We are indifferent about the behavior of *latent representation*
 - ↳ We do not need to include \mathbf{Z} directly in loss computation
 - ↳ \mathbf{Z} contributes to loss indirectly through $\hat{\mathbf{X}}$

So, the loss in this case is compute as

$$\hat{R} = \mathcal{L}(\hat{\mathbf{X}}, \mathbf{X})$$

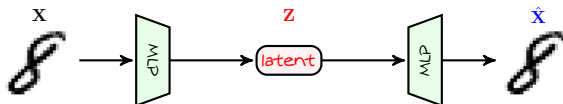
Training AEs for Compression



We know the loss: we can analytically compute $\nabla_{\hat{\mathbf{X}}} \hat{R}$, so training is done by standard forward and backward pass

- 1 Pass \mathbf{X} forward through the **encoder**
 - ↳ Compute output of all layer as well as \mathbf{Z}
- 2 Pass \mathbf{Z} forward through the **decoder**
 - ↳ Compute output of all layer as well as $\hat{\mathbf{X}}$
- 3 Compute $\nabla_{\hat{\mathbf{X}}} \hat{R}$ and backpropagate through decoder
- 4 Compute $\nabla_{\mathbf{Z}} \hat{R}$ from the gradient at the first layer of decoder
- 5 Starting from $\nabla_{\mathbf{Z}} \hat{R}$ backpropagate through encoder
- 6 Update all weights and go for the next round

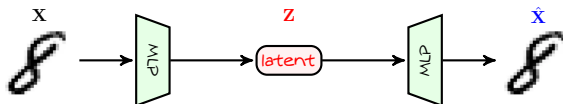
Example: Compressing MNIST



A simple practice can be done on MNIST: we try to represent MNIST images in a **2-dimensional latent space**. For **encoding** we use the following MLP

- 1 It has **four hidden layers**
 - ↳ The widths of layers gradually reduce
 - ↳ The last layer has only two outputs
- 2 All neurons are activated via sigmoid
- 3 We **do not use** any dropout or batch-normalization

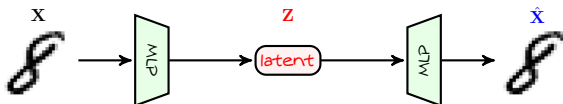
Example: Compressing MNIST



For *decoding* we use another MLP to invert the encoder

- 1 The decoder has **four hidden layer**
 - ↳ The widths of layers gradually increase
 - ↳ The last layer has **784 neurons**
- 2 All neurons are activated via sigmoid
- 3 We finally sort the output into a **28×28 matrix**

Example: Compressing MNIST



Training then follows the *standard approach*: this is in fact an 8-layer MLP

- 1 Pass each training image *forward* through all layers
- 2 Compute *loss* between the output and true image, e.g.,

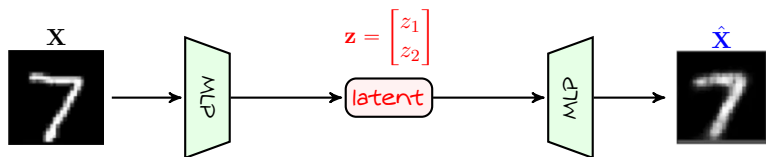
$$\mathcal{L}(\hat{\mathbf{X}}, \mathbf{X}) = \|\hat{\mathbf{X}} - \mathbf{X}\|^2$$

- 3 Compute $\nabla_{\hat{\mathbf{X}}} \hat{R}$ and *backpropagate*
- 4 Update all weights and go for the next round

Example: Compressing MNIST

We can then test our AE

- 1 Pass a test image forward through *encoder*
- 2 Compute *latent representation*
- 3 Pass the latent representation forward through *decoder*
- 4 Compare the images



Obvious Compression via Vanilla AE

For compression it is important that we set

the *latent representation* to be of *smaller* dimension than input

If we set it *larger* or equal to the input size, we end up with an *obvious solution*

$$\text{Decoder}(\text{Encoder}(\cdot)) = \text{Identity}(\cdot)$$

- We want to recover the original data after decoding
 - ↳ Identity is always an obvious solution
- With *larger latent space* we can *always* realize identity
 - ↳ We simply set $\mathbf{Z} = \mathbf{X}$ and $\hat{\mathbf{X}} = \mathbf{Z}$
- This is however *useless* since we do *not compress*

Sparse AEs

Let's keep the track of their applications

① Compression

② Finding a *sparse* representation of *data*

↳ We intend to represent *data* with a *sparse object*

↳ for instance, we intend to represent input $\mathbf{x} \in \mathbb{R}^{100}$ with *another 100-dimensional* vector whose *most of entries* are *zero*

↳ we may want to further *compress*, i.e., represent $\mathbf{x} \in \mathbb{R}^{100}$ with an *80-dimensional sparse* vector

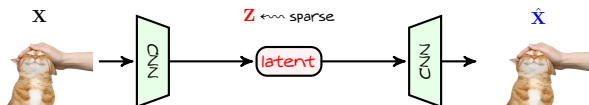
↳ For *loss computation*, we should also take a look at the *latent representation*

↳ we want the latent representation to be *sparse*

↳ in *vanilla AE* there is *no guarantee* that this happens

For such application we use *sparse AEs*

Training AEs for Sparse Representation



Let's formulate the problem: say the input is X , *latent representation* is Z , and \hat{X} is *decoded output*

- We still need to recover from *latent representation*, i.e., we want $\hat{X} = X$
 - ↳ Loss is proportional to the difference between X and \hat{X}
- We also want to have *sparse latent representation*
 - ↳ Z should contribute directly to loss
 - ↳ Loss should also be proportional to the *sparsity* of Z

Training Sparse AEs

Loss is proportional to *difference* between \mathbf{X} and $\hat{\mathbf{X}}$, and *sparsity* of \mathbf{Z}

So, the loss in this case should be

$$\hat{R} = \mathcal{L}(\hat{\mathbf{X}}, \mathbf{X}) + \lambda S(\mathbf{Z})$$

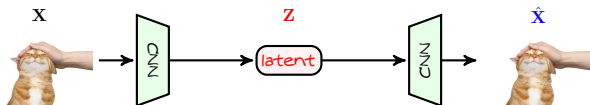
for some function $S(\cdot)$ that is proportional to sparsity, i.e.,

if \mathbf{Z} has less zeros $\rightsquigarrow S(\mathbf{Z})$ should increase

and regularizer λ that is a *hyperparameter*

- $S(\mathbf{Z}) = \|\mathbf{Z}\|_0 \rightsquigarrow$ *non-differentiable ✗*
- $S(\mathbf{Z}) = \|\mathbf{Z}\|_1 \rightsquigarrow$ *convex ✓*
- $S(\mathbf{Z}) = \text{KL}(p_{\mathbf{Z}} \parallel \text{Ber}_{\rho}) \rightsquigarrow$ *convex ✓*
 - ↳ Ber_{ρ} is a Bernoulli distribution with probability of zero being ρ
 - ↳ $p_{\mathbf{Z}}$ is the empirical distribution of the support of \mathbf{Z}

Training Sparse AEs



Let's see how training looks: say we are training with single sample \mathbf{X}

- Pass forward \mathbf{X} through encoder and decoder
- Backpropagate by first computing $\nabla_{\hat{\mathbf{X}}} \hat{R}$
 - ↳ Backpropagate till the **bottleneck**
 - ↳ At the **bottleneck**, we need to compute $\nabla_{\mathbf{Z}} \hat{R}$

$$\nabla_{\mathbf{Z}} \hat{R} = \underbrace{\nabla_{\hat{\mathbf{X}}} \hat{R} \circ \nabla_{\mathbf{Z}} \hat{\mathbf{X}}}_{\text{computed By Backpropagation}} + \lambda \nabla_{\mathbf{Z}} S(\mathbf{Z})$$

- ↳ Start from $\nabla_{\mathbf{Z}} \hat{R}$ and backpropagate till input
- Update weights and go for the next round

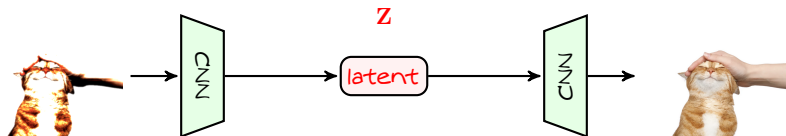
Using AE for Noise Removal

Let's keep the track of their applications

- 1 *Compression*
- 2 *Finding a sparse representation of data*
- 3 *Denoising*
 - ↳ We intend to find a *representation* that can *refine noisy data*
 - ↳ for instance we want to *remove background noise* from an image
 - ↳ for instance we want to *increase the resolution* of an image
 - ↳ for instance we want to *color a gray image*
 - ↳ For loss computation, we should
 - ↳ be able the *recover* the *refined data* at the decoder
 - ↳ unlike *vanilla AE* we can start with *distorted data*

We call such AE architectures *denoising AEs*

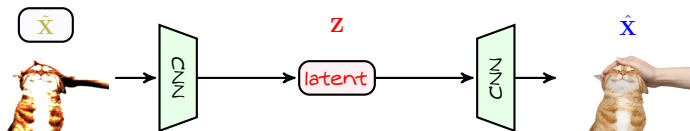
Training Denoising AEs



We can train a **denoising** AE using **degraded** samples

- For each training **sample** we generate its **degraded counterpart**, e.g.,
 - ↳ for each image we also produce a **noisy**, or **low-resolution** or **gray version**
- We give this **noisy version** to the **encoder**
- We set loss to compute difference between **original** samples and **output**
 - ↳ the **decoded image** and the **original RGB image** in dataset

Training Denoising AEs



Let's formulate the problem: say the sample is \mathbf{X} , and its corrupted version is $\tilde{\mathbf{X}}$. Also, denote *latent representation* by \mathbf{Z} and *decoded output* by $\hat{\mathbf{X}}$

- We want to recover original data from *latent representation*, i.e., $\hat{\mathbf{X}} = \mathbf{X}$
 - ↳ Loss is proportional to the difference between \mathbf{X} and $\hat{\mathbf{X}}$
- We may want our representation to be sparse
 - ↳ We could add a penalty proportional to *sparsity* of \mathbf{Z}

So, we set the loss to

$$\hat{R} = \mathcal{L}(\hat{\mathbf{X}}, \mathbf{X}) + \lambda S(\mathbf{Z})$$

Training AEs: Summary

We could have various form of AEs depending on the target application

- **Vanilla** AEs

- ↳ Encoder-decoder with both input and label being data
- ↳ Loss computes difference between input and output \equiv recovery error
- ↳ We can use it for compression

- **Sparse** AEs

- ↳ Encoder-decoder with both input and label being data
- ↳ Loss computes recovery error plus a sparsity penalty
- ↳ We can use it for sparse representation of data

- **Denoising** AEs

- ↳ Encoder-decoder with input being noisy data and label being data
- ↳ Loss computes recovery error
- ↳ We can also regularize with a sparsity penalty if we need sparse latent
- ↳ We can use it for noise removal, resolution increasing and other similar applications