Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

# 10 - Autoencoders

François Pitié

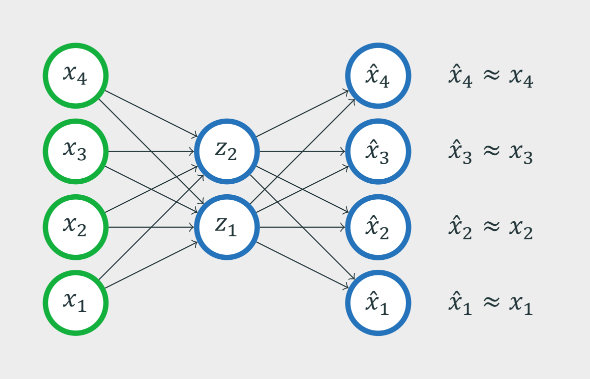
Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

*[4C16/5C16] Deep Learning and its Applications — 2021/2022*

So far, we have looked at **supervised learning applications**, for which the training data $\mathbf{x}$ is associated with ground truth labels $\mathbf{y}$. For most applications, labelling the data is the hard part of the problem.

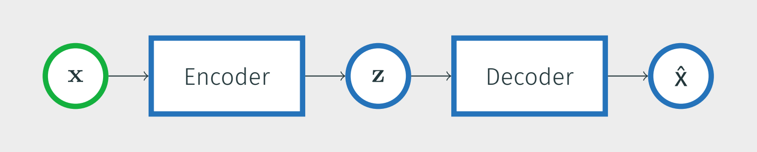
Autoencoders are a form of **unsupervised learning**, whereby a trivial labelling is proposed by setting out the output labels $\mathbf{y}$ to be simply the input $\mathbf{x}$. Thus autoencoders simply try to reconstruct the input as faithfully as possible.

Autoencoders seem to solve a trivial task and the identity function could do the same. However in autoencoders, we also enforce a dimension reduction in some of the layers, hence we try to "compress" the data through a bottleneck.



In this example of autoencoder, the input data $(x_1, x_2, x_3, x_4)$ is mapped into the compressed hidden layer $(z_1, z_2)$ and then re-constructed into $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$.

The idea is to find a lower dimensional representation of the data, where we can explain the whole dataset $\mathbf{x}$ with only two latent variables $(z_1, z_2)$.



$\mathbf{x}$ → Encoder → $\mathbf{z}$ → Decoder → $\hat{\mathrm{x}}$

The autoencoder architecture applies to any kind of neural net, as long as there is a bottleneck layer and that the output tries to reconstruct the input.

Typically, for continuous input data, you could use a $L_2$ loss as follows:

$$\text{Loss}\ \hat{\mathrm{x}} = \frac{1}{2}||\hat{\mathrm{x}} - \mathrm{x}||^2$$

Alternatively you can use cross-entropy if $\mathbf{x}$ is discrete.

The following examples consider the MNIST handwritten digit database and are taken from the link below:

https://blog.keras.io/building-autoencoders-in-keras.html

Below is an example of autoencoder using FC layers:

```
encoding_dim = 32
input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
autoencoder = Model(input_img, decoded)
```

Below is an example of convolutional autoencoder:

```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (7, 7, 32)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

# Transposed Convolution

A note about convolutional autoencoders.

In the convolutions handout, we saw how pooling and strides can be used to reduce the horizontal and vertical dimensions of a tensor.

Similarly we can increase the horizontal and vertical dimensions of a tensor using an upsampling operation. This step is sometimes called **up-convolution**, **deconvolution** or **transposed convolution**.

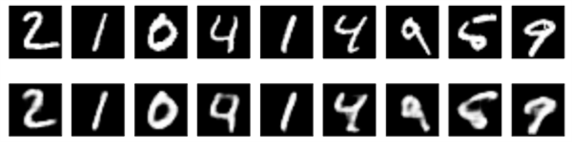
This step is equivalent to first upsampling the tensor by inserting zeros in-between the input samples and then applying a convolution layer. More on this is discussed in the link below.

https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

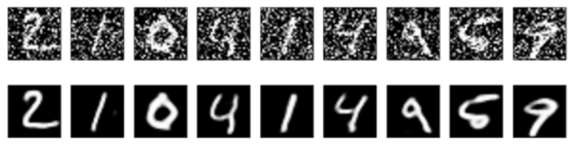
# Transposed Convolution

Just note that Deconvolution is an unfortunate term for this step as deconvolution is already used in signal processing and refers to trying to estimate the input signal/tensor from the output signal. (eg. trying to recover the original image from an blurred image).

Take MNIST as an example. Below are input (top) and reconstructions results (bottom) using the previous slide convolutional autoencoder:



The interest of a compressed representation is illustrated below. The input is noisy but the decoder network is trained to only reconstruct clean images.



This looks interesting but be aware that dimensional reduction does not necessarily imply information compression.

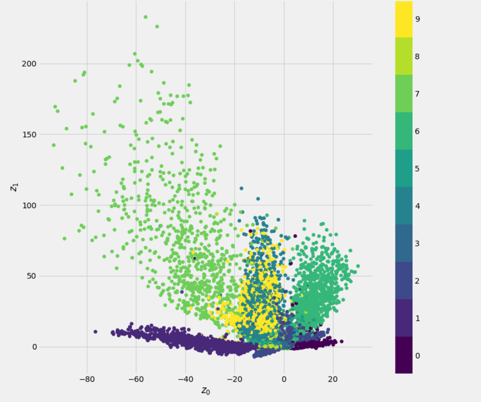
Consider for instance a one-to-one correspondence function (bijection) between $\mathbb{R}^{10}$ and $\mathbb{R}$. Applying this function will not change the entropy of the data.

In practice, the encoder network is not a bijection and information compression does occur, but maybe not as much as hoped for. In the next slider we show this on an AE with only 2 latent variables.

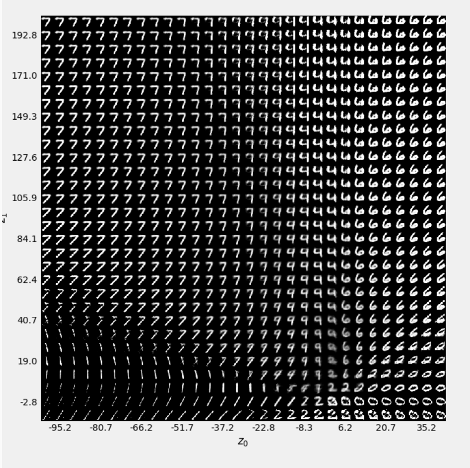
What this means for us, is reducing the dimensions might come at the expense of making the latent space extremely entangled.

Let’s see that on an example for MNIST with a 2D latent space with the following network:

```
inputs = Input(shape=(784,))
x = Dense(512, activation='relu')(inputs)
z = Dense(2, name='latent_variables')(x)
x = Dense(512, activation='relu')(z)
outputs = Dense(784, activation='sigmoid')(x)
```



Here is the associated 2D scatter plot of the latent variable $(z_1, z_2)$, coloured by class id. You can see a complex partition of the space. Classes clusters might be skewed, or broken into different parts, and leaving gaps where values of $(z_1, z_2)$ do not correspond to any digit.



Here we show the decoded images for each value of $(z_1, z_2)$. For values of $(z_1, z_2)$ outside of the main clusters, the reconstructed images become blurred or deformed.

The issue with AEs is that we ask the NN to somehow map 784 dimensions into 2, without enforcing any compactness about the distribution of $\mathbf{z}$. Since the loss is only focused on the reconstruction fidelity, the latent space could end up being messy.

What we are really looking for is an untangled latent space, where each latent variable have its own semantic meaning: eg. $z_1$ controls the size of head, $z_2$ the colour of the hair, $z_3$ the size of the smile, etc.

But with AEs, $z_1, z_2, \cdots, z_n$ could be deeply entangled, making any subsequent analysis difficult.

VAE tries to address this.

# Variational Auto Encoders

In Variational Auto Encoders (VAEs), we impose a prior on the distribution of the latent vectors $\{\mathbf{z}_1, \cdots, \mathbf{z}_n\}$.

In particular, we are going to assume that $p(\mathbf{z})$ should follow a normal distribution:

$$p(\mathbf{z}) = \mathcal{N}(0, Id)$$

Which means that the distribution of $\mathbf{z}$ will be smooth and compact, without any gap.

Since we are looking to constrain the distribution $p(\mathbf{z})$ and not just the actual values of $\mathbf{z}$, we will need to now manipulate distributions rather than data points.

As manipulating distributions is a bit tricky and yield intractable equations, we will make some approximations along the way and resort to a Variational Bayesian framework. In particular, in VAE we will assume that $p(\mathbf{z})$ should follow a normal distribution:

$$p(\mathbf{z}) = \mathcal{N}(0, Id)$$

and also, that the uncertainty $p(\mathbf{z}|\mathbf{x})$ follows a Multivariate Gaussian:

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_{\mathbf{z}|\mathbf{x}}, \Sigma_{\mathbf{z}|\mathbf{x}})$$

Recall that $p(\mathbf{z}|\mathbf{x})$ models the range of values for $\mathbf{z}$ that could have produced $\mathbf{x}$ (eg. variations happen due to unrelated processes such as noise).

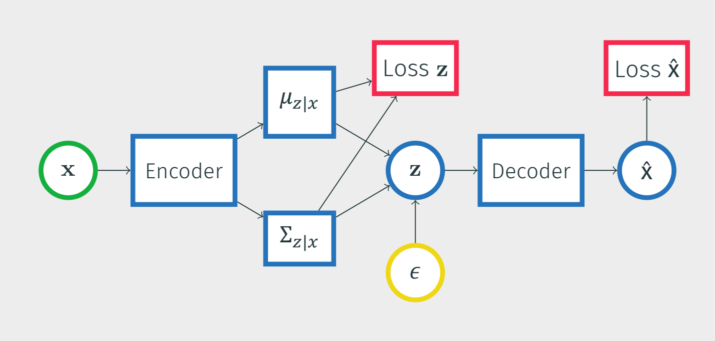
The variational auto-encoder approach is then as follows.

The encoder network predicts the distribution $p(\mathbf{z}|\mathbf{x})$ by directly predicting its mean and variance $\mu_{\mathbf{z}|\mathbf{x}}$ and $\Sigma_{\mathbf{z}|\mathbf{x}}$.

Then, the decoder network samples $z \sim p(\mathbf{z}|\mathbf{x})$ and reconstructs $\hat{\mathbf{x}}$.

We enforce $p(\mathbf{z}) \approx \mathcal{N}(0, Id)$ by adding an additional loss on $p(\mathbf{z}|\mathbf{x})$, which you can think of as a regularisation term for $\mu_{\mathbf{z}|\mathbf{x}}$ and $\Sigma_{\mathbf{z}|\mathbf{x}}$:

$$\text{Loss}\,\mathbf{z} = D_{KL}(\mathcal{N}(\mu_{z|x}, \Sigma_{z|x}), \mathcal{N}(0, Id)) = \frac{1}{2}\sum_k \Sigma_{k,k} + \mu_k^2 - 1 - \log(\Sigma_{k,k})$$

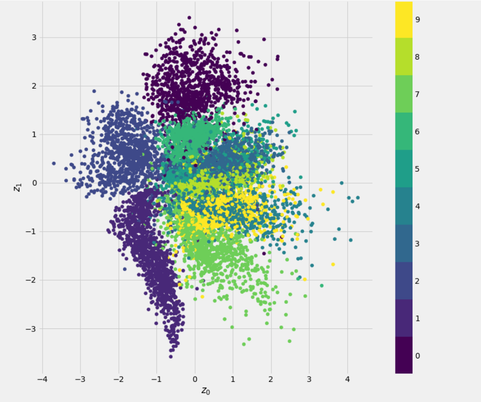


$$\mathbf{z} = \mu_{\mathbf{z}|\mathbf{x}} + \Sigma_{\mathbf{z}|\mathbf{x}}^{\frac{1}{2}} \epsilon$$

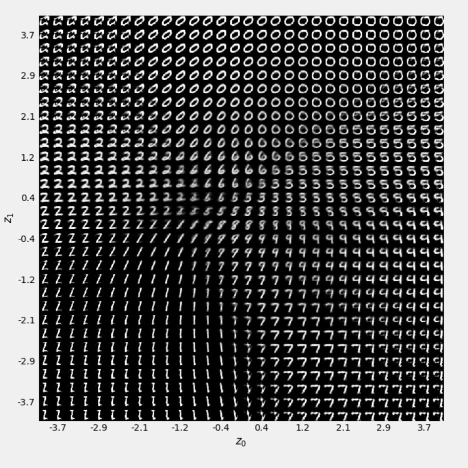
$$\text{Loss } \mathbf{z} = D_{KL}(\mathcal{N}(\mu_{z|x}, \Sigma_{z|x}), \mathcal{N}(0, Id)) = \frac{1}{2} \sum_k \Sigma_{k,k} + \mu_k^2 - 1 - \log(\Sigma_{k,k})$$

$$\text{Loss } \hat{\mathrm{x}} = \frac{1}{2} ||\hat{\mathrm{x}} - \mathrm{x}||^2$$

You can think of the sampling step as a way of working on distributions when in fact we are only defining the decoder network as an operation on data.



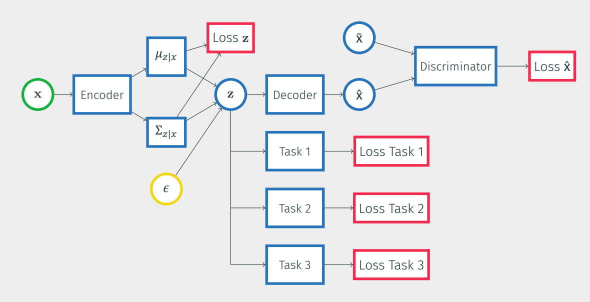
Here is the associated 2D scatter plot of the latent variable $(z_1, z_2)$, coloured by class id. You can see that the distribution is closer to a normal distribution and class clusters are less skewed than compared to AE.



Here we show the decoded images associated with each value of $(z_1, z_2)$. We can see see that ill-formed reconstructions now only arise for extreme values of $(z_1, z_2)$.

# Examples

# Going further: GAN/VAE and multitask ...



A more complete VAE example, including a discriminator loss as in GAN and a few other tasks.

# VAE/GAN

# Vector Attributes

Take away, supervised learning is far better to get good performance. But, what VAE allows you to do is to define a latent space that can then be used for generating data. (Think of the VGG features are very powerful, but you can’t simply recreate images using them alone, VAE can).

Multitask …

https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/