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

# 09 - Generative Models

François Pitié

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

*[4C16/5C16] Deep Learning and its Applications — 2023/2024*

# Discriminative Models

So far, we have mainly looked as Discriminative Models.

The models try to estimate the likelihood $P(Y|X)$.

For instance, logistic regression would be a discriminative classifier.

These where all **supervised learning** models (the training data $\mathbf{X}$ is associated with given labels $\mathbf{y}$).

### Example

Given a dataset of labelled images, detect whether the picture is of a face or not.

# Generative Models

Generative Models try to model the probability distributions of the data itself, *ie.* $P(X)$ or $P(X|Y)$.

The objective is to learn how to synthesise new data, that is similar to the training data.

As we don't have ground truth data about what that data should be, it is a form of **unsupervised learning** (we're just trying to learn from the data itself).

### Example

Given a dataset of faces, and you would like to generate new realistic looking faces:

$$X \sim P(X|Y = '\texttt{cat}')$$

# Generative Models

Popular generative models that use deep learning include:

- Generative Adversarial Networks
- Autoencoders
- Auto-Regressive Models (*eg.* GTP-3,4, see later handout)
- Diffusion Models (*eg.* image generation apps like Midjourney, DallE, *etc.*, not covered in this module)

# Generative Adversarial Networks (GANs)

# GANs

It is not immediately obvious how solve the generative problem with the kind of approach we've adopted so far. We have a number of issues to solve:

**A.** How do you generate the new data?

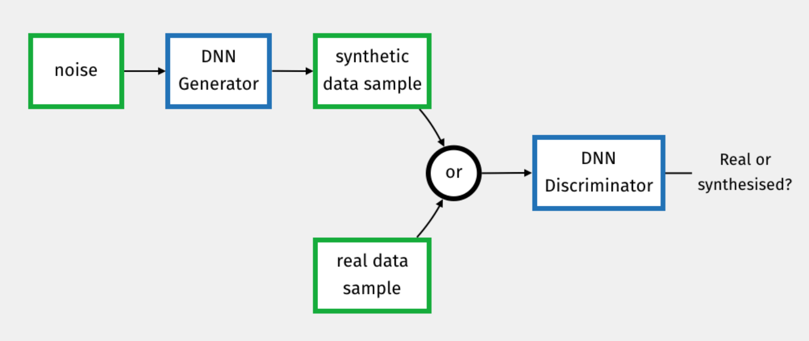
**B.** Then, what would be the loss function? If we generate data that is different from any sample from our existing dataset, how do we measure how realistic this is, *ie.* how do we know whether we are indeed sampling from $P(X|Y)$?

GANs tries to solve this by addressing both issues at the same time.

> Ian Goodfellow et al. "Generative Adversarial Network" (2014). [https://arxiv.org/abs/1406.2661]

# GANs

GAN's architecture (see below) is made of two DNNs: a **Generator Network**, that is responsible for generating fake data, and a **Discriminator Network** that is responsible for detecting whether data is fake or real.



# GANs

Typically the **generator** network takes an input noise and transforms that noise into a sample of a target distribution. This is our photo portrait generator: it transforms a random seed into a photo portrait of a random person.

The loss function for that generator is the **discriminator** network that in our case classifies between fake and real photos.

It is thus an arms race where each network is trying to outdo the other.

Both problems taken independently are hard because the generator is missing a loss function and the discriminator is missing data but, by addressing both problem at the same time in a single architecture, we can solve for both problems.

# GANs

A GAN can be trained by alternating the phases of training:

1. Freeze the Generator and train the Discriminator (eg. 1+ epochs)
2. Freeze the Discriminator and train the Generator (eg. $> 1$ epochs)
3. Repeat steps 1. and 2.

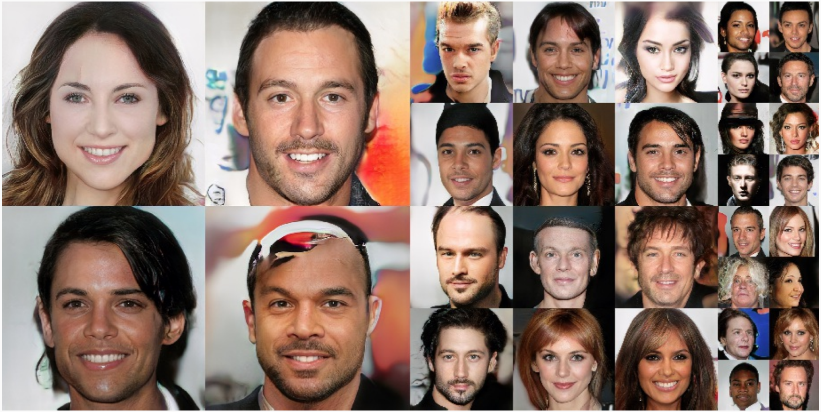
The issue is that the convergence is a bit complicated...

For instance, if the Generator is perfect, there is no flaw to be learned for the discriminator, and the discriminator will be a bit random, and at the next step the Generator will not be put in check anymore by the discriminator.

Thus training a GAN network is particularly difficult.

# GANs

but the benefits can be spectacular:



Example of GAN generating pictures of fake celebrities (see 2018 ICLR NVidia paper here [https://goo.gl/AgxRhp])

# Autoencoders

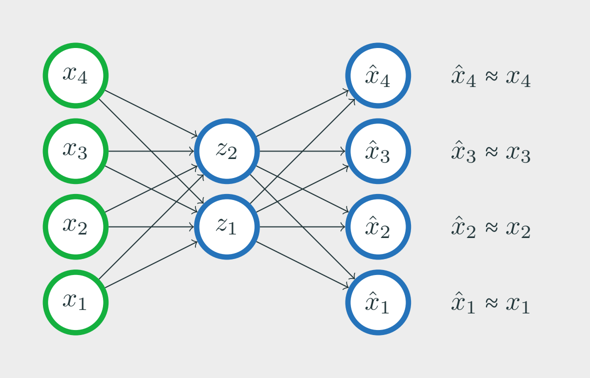
# Autoencoders

For most supervised learning applications, labelling the data is the hard part of the problem.

In Autoencoders we propose to solve the generative model problem by establishing a trivial labelling. The idea is to set out the output labels $\mathbf{y}$ to be simply the input $\mathbf{x}$. In another words, 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.

# Autoencoders

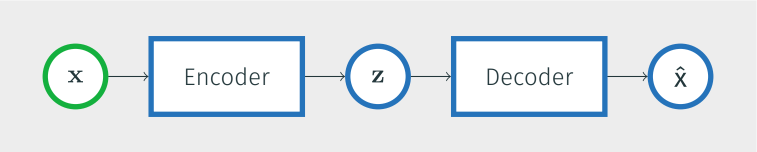


In this example of autoencoder, the input data $(x_1, x_2, x_3, x_4)$ is mapped into the compressed hidden layer $(z_1, z_2)$ and then re-constructed into $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$.

# Autoencoders

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

# Autoencoders



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:

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

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

# Autoencoders

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

# Autoencoders

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)
```

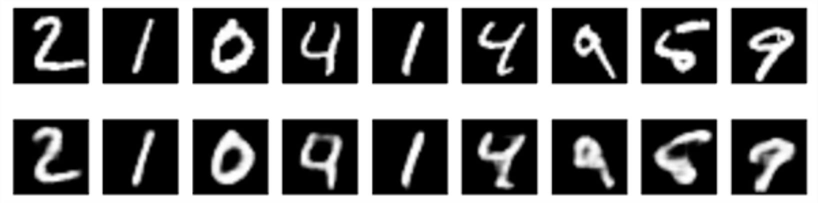
# Autoencoders

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')
```

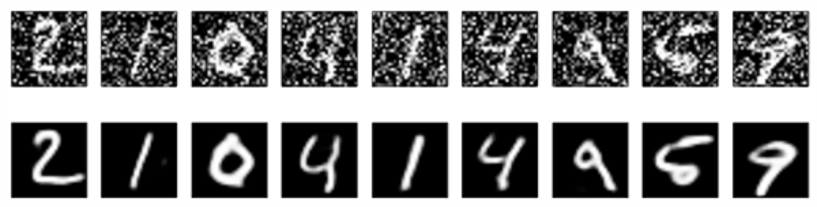
# Autoencoders

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



# Autoencoders

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



# Autoencoders

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 there is no clear guarantee.

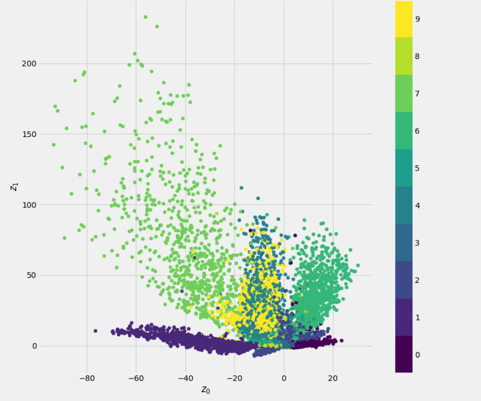
What this means for us, is dimension reduction might come at the expense of making the latent space extremely entangled.

# Autoencoders

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

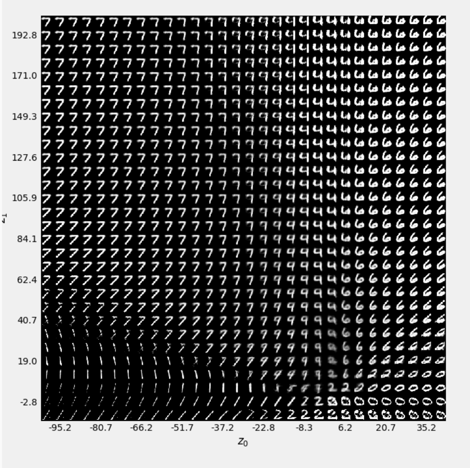
```
inputs = Input(shape=(784,))
x = Dense(512, activation=’relu’)(inputs)
z = Dense(2, name=’latent_variables’)(x)
x = Dense(512, activation=’relu’)(z)
outputs = Dense(784, activation=’sigmoid’)(x)
```

# Autoencoders



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.

# Autoencoders



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.

# Autoencoders

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.

# Variational Auto Encoders

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).

# Variational Auto Encoders

The variational auto-encoder approach is then as follows.

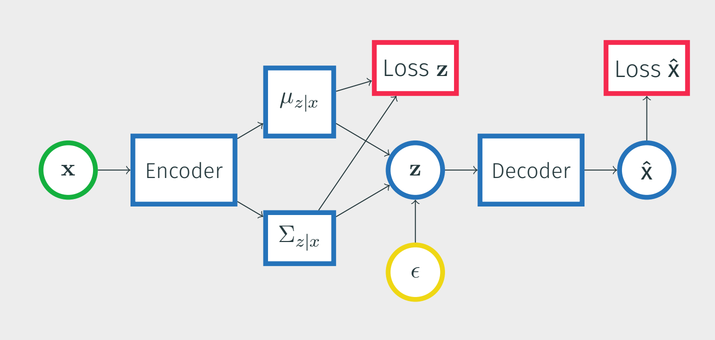
The encoder network predicts the distribution $p(\mathbf{z}|\mathbf{x})$ by directly predicting its mean and variance $\mu_{\mathbf{z}|\mathbf{x}}$ and $\Sigma_{\mathbf{z}|\mathbf{x}}$.

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 $\Sigma_{\mathbf{z}|\mathbf{x}}$:

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

# Variational Auto Encoders



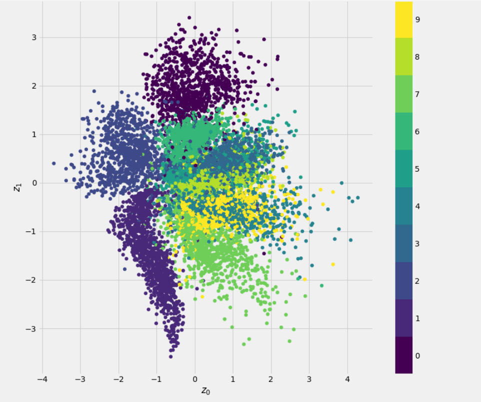
$$\mathbf{z} = \mu_{\mathbf{z}|\mathbf{x}} + \Sigma_{\mathbf{z}|\mathbf{x}}^{\frac{1}{2}}\epsilon$$

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

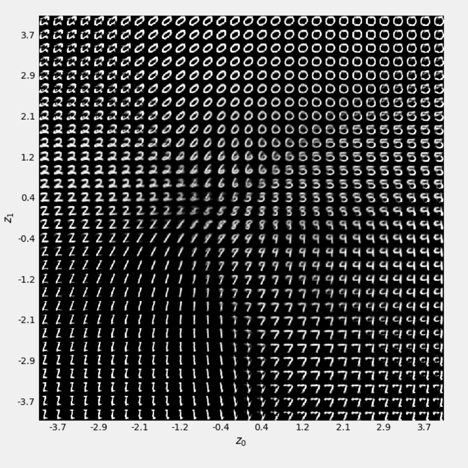
$$\text{Loss } \hat{\mathsf{x}} = \frac{1}{2}\|\hat{\mathsf{x}} - \mathsf{x}\|^2$$

# Variational Auto Encoders

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.

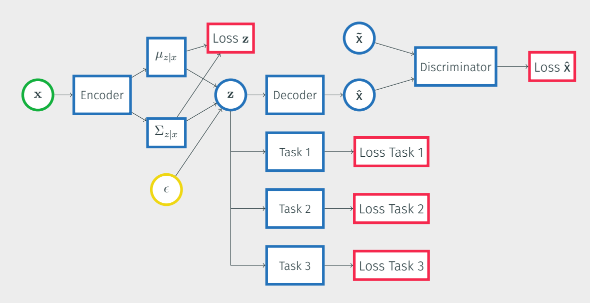


Here is the associated 2D scatter plot of the latent variable $(z_1, z_2)$, coloured by class id. You can see that the distribution is closer to a normal distribution and class clusters are less skewed than compared to AE.



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 $(z_1, z_2)$.

# Variational Auto Encoders



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

# Deep Auto-Regressive Models

This is something we already looked at in RNNs. The idea is to forecast future behavior based on past behavior data:

$$p(x_i|x_1,\ldots,x_{i-1})$$

Generally we restrict ourselves to a context window and end up with something like a Markov chain:

$$p(x_i|x_1,\ldots,x_{i-1}) = p(x_i|x_{i-k},\ldots,x_{i-1})$$

See handout on RNNs how to use it for text generation. This is also the basis for large language models (see later handout).

# Take Away

As always, the strength of Deep Learning comes with scale.

If supervised learning is still the most efficient way to get good performance and still represents the vast majority of Deep Learning applications (eg. VGG still represents, to that day, a very good basis of visual features), it is not always easy to find large datasets for training.

The solution found in unsupervised learning is to adopt trivial labelling schemes. For instance, in autoencoders you try to predict yourself, in autoregressive models you try to predict your present self from your own past, in GAN you try to establish a trivial classification problem where you attempt to detect whether you are yourself or not.

The key is to find data where it is plentiful, and if there is no good labels to use, then assign trivial labels to it.