Machine Learning

Variational Autoencoders and Generative models

Dr Guillermo Hamity

University of Edinburgh

October 25, 2021

European Research Council

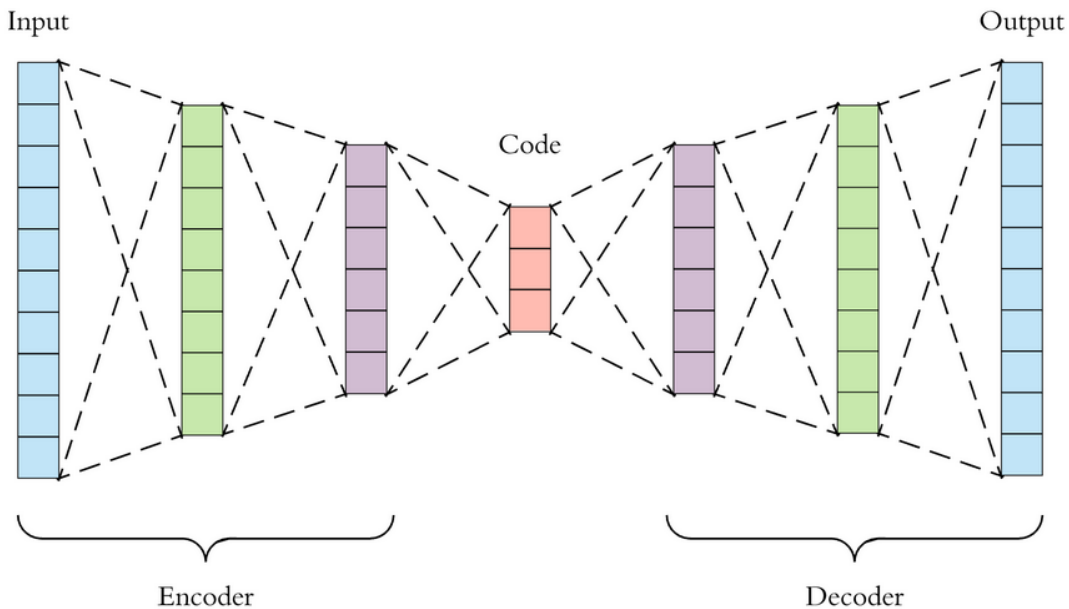
Established by the European Commission

Auto encoders

Recap on checkpoint on autoencoders

- **Encoder** maps to a laten space
  - `28x28` to lower dimensional order 10 features
- **Decoder** reconstructs image from laten space

> We did this with convolutional neural networks



Lets discuss some potential solutions

Checkpoint 1 (Solution 1)

- Encoder 16 (28x28)->8 (14x14)->4 (7x7)->2 (7x7)->1 (7x7)
- Decoder 16 (7x7)->8 (14x14)->4 (28x28)->2 ->1 (28x28)

```
i = Input(shape=shape, name='input')
x = i
x = Conv2D(16, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 8, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 4, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 2, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 1, kernel_size=(3,3), padding='same', activation='relu')(x)
e = Flatten()(x)

encoder = Model(i, e, name='encoder')

# Define convolutional decoder model
t = Input(shape=(np.prod(encoding_shape),))
x = Reshape(encoding_shape)(t)
x = Conv2D(16, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2D(8, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2D(4, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D(2, kernel_size=(3,3), padding='same', activation='relu')(x)
o = Conv2D(1, kernel_size=(3,3), padding='same', activation='sigmoid')(x)
decoder = Model(t, o, name='decoder')
decoder.summary()
```

Checkpoint 1 (Solution 2) — Better

An convolution (encoder) and a deconvolution (decoder) with symmetrically inverted structures

- Encoder 1 (28x28)-> 2 (14x14)-> 4 (7x7)->8 (7x7)->16 (7x7)
- Decoder 16 (7x7)->8 (14x14)->4 (28x28)->2 ->1 (28x28)

```
i = Input(shape=shape, name='input')
x = i
x = Conv2D(1, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 2, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 4, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 8, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 16, kernel_size=(3,3), padding='same', activation='relu')(x)
e = Flatten()(x)

encoder = Model(i, e, name='encoder')

# Define convolutional decoder model
t = Input(shape=(np.prod(encoding_shape),))
x = Reshape(encoding_shape)(t)
x = Conv2DTranspose(16, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2DTranspose(8, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2DTranspose(4, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2DTranspose(2, kernel_size=(3,3), padding='same', activation='relu')(x)
o = Conv2DTranspose(1, kernel_size=(3,3), padding='same', activation='sigmoid')(x)
decoder = Model(t, o, name='decoder')
```

Note on Convolution Transpose

In previous lecture we saw that convolution:

- 10 trainable parameters
- NxN figure goes to (N-2)x(N-2)

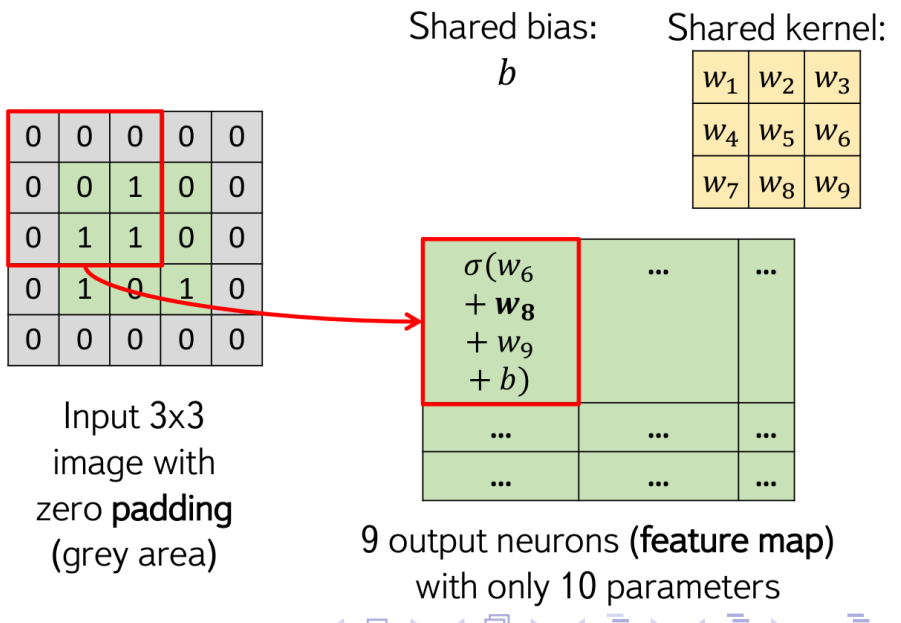
```
x = Conv2D( 16, kernel_size=(3,3))(x)
```

```
x = Conv2DTranspose(16,
    kernel_size=(3,3))(x)
```

Transpose operation is also available:

- 10 trainable parameters
- (N-2)x(N-2) figure goes to NxN

With **same padding** prectically identical



Checkpoint 1 (Solution 3) — Best I found

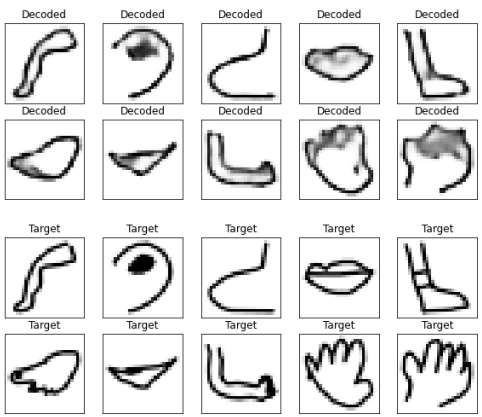
Same as solution 2, but **more kernels** (approx 10 min training)

```
i = Input(shape=shape, name='input')
x = i
x = Conv2D(1, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 5, kernel_size=(3,3), padding='same', activation='relu')(x)
x = MaxPooling2D(pool_size=(2,2))(x)
x = Conv2D( 10, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 20, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D( 40, kernel_size=(3,3), padding='same', activation='relu')(x)
e = Flatten()(x)

encoder = Model(i, e, name='encoder')

# Define convolutional decoder model
t = Input(shape=(np.prod(encoding_shape),))
x = Reshape(encoding_shape)(t)
x = Conv2D(40, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2D(20, kernel_size=(3,3), padding='same', activation='relu')(x)
x = UpSampling2D(size=(2,2))(x)
x = Conv2D(10, kernel_size=(3,3), padding='same', activation='relu')(x)
x = Conv2D(5, kernel_size=(3,3), padding='same', activation='relu')(x)
o = Conv2D(1, kernel_size=(3,3), padding='same', activation='sigmoid')(x)
decoder = Model(t, o, name='decoder')
decoder.summary()
```

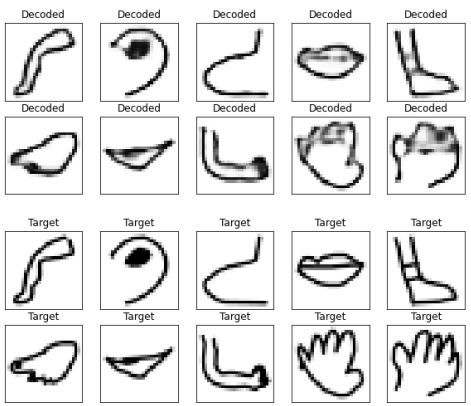
Results

Resolution improves to the right (visible fingers!)

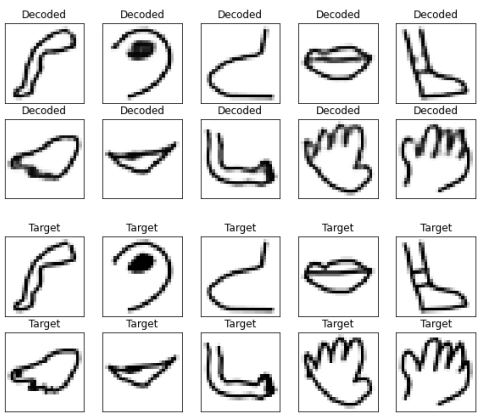
Solution1



Solution2



Solution3



Generative algorithms

- Generative algorithms are part of unsupervised learning techniques
- A generative model can learn to mimic any distribution of it.
- They are powerful because can learn to reproduce similar models in any domain
- some of these domains are:
  - Images
  - Music
  - Speech
  - Text
  - Videos

Discriminative vs generative algorithms

- **Discriminative models** aim to predict the label to which the data belongs.
  - it aims to map features to label
- **Generative models** on the other hand do the opposite: they aim to predict features given a certain label.

Discriminative vs generative: spam emails

Example (Is an email spam)

- Let's consider a practical example in the context of whether an email is spam or not
- We can consider **x** the **model feature**
  - for example all the words in an email
- We can consider **y** the **target variable**
  - whether the email is actually spam
- The discriminative and generative models will aim to answer the following questions:

Discriminative

p(y|x): Given the input feature **x** what is the probability of the email being spam?

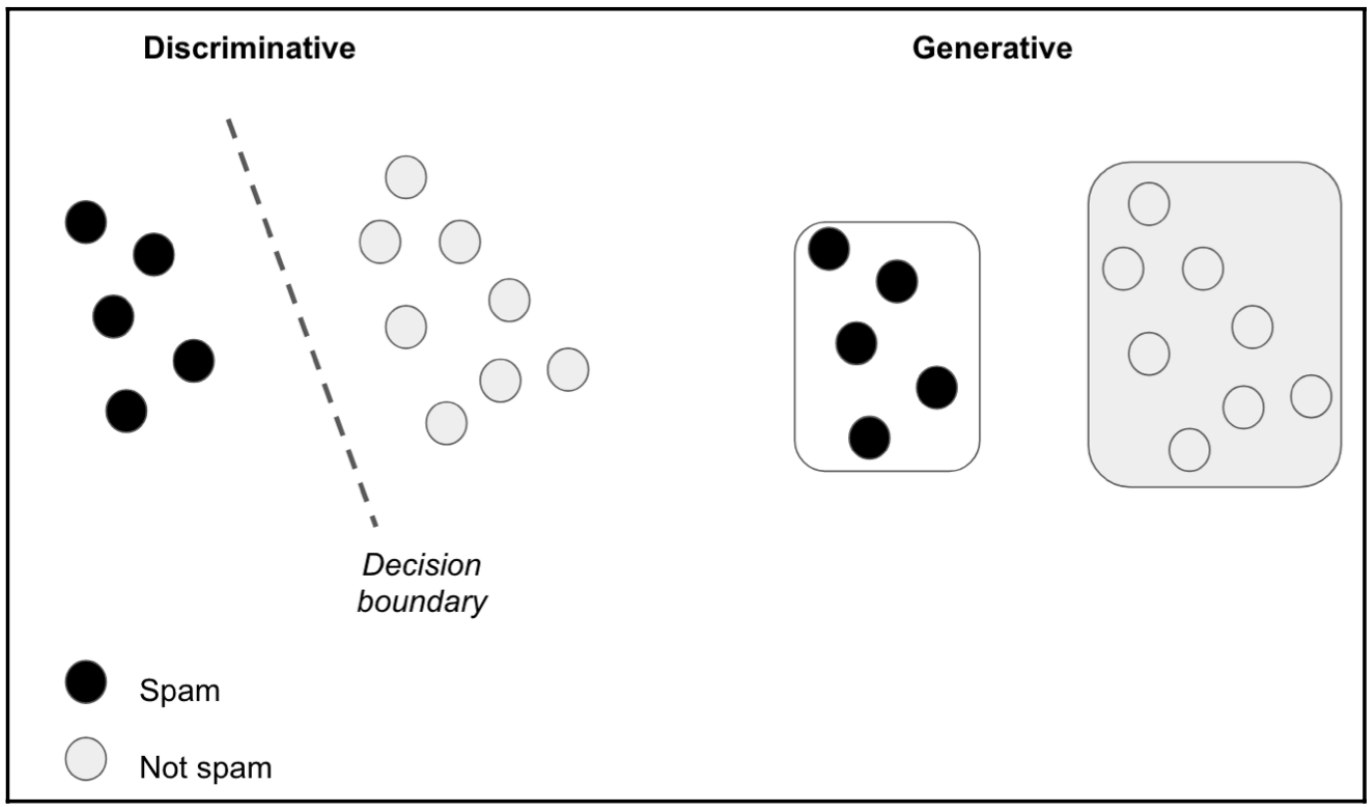
Generative

p(x|y): Given that the email is spam how likely are the input features to be x?

Discriminative vs generative: spam emails

In other words

- **Discriminative models** learns the boundary between classes
- **Generative models** learn to model the distribution of individual classes



Example Generative model

Foreshadowing generative model you will use in the course **Monter Carlo (MC) sampling**

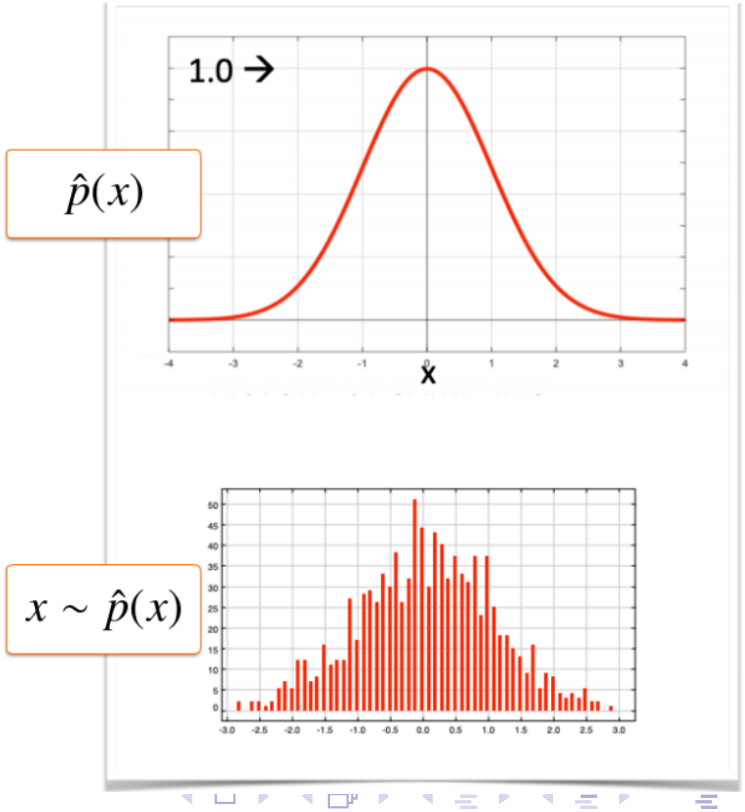
- Generative model

$$\hat{p}(x) = N(x|\mu = 0, \sigma = 1)$$

- Sample using e.g. random number generation
- Method is **transparent** and works well for small number of dimensions:

$$n \approx O(1)$$

- Quickly becomes expensive with multiple dimensions
- How to generate e.g. images that are $O(100'000)$ dimensional?



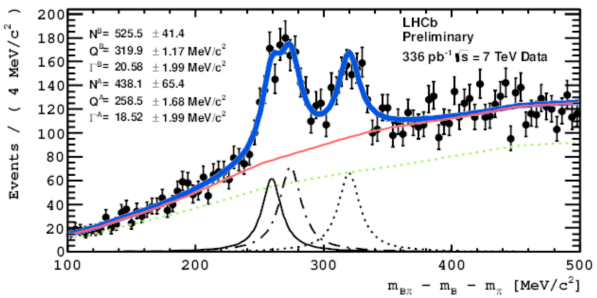
Building generative model

Build model from **explicit** probability distribution, i.e. we choose p.d.f. structure

Fitting

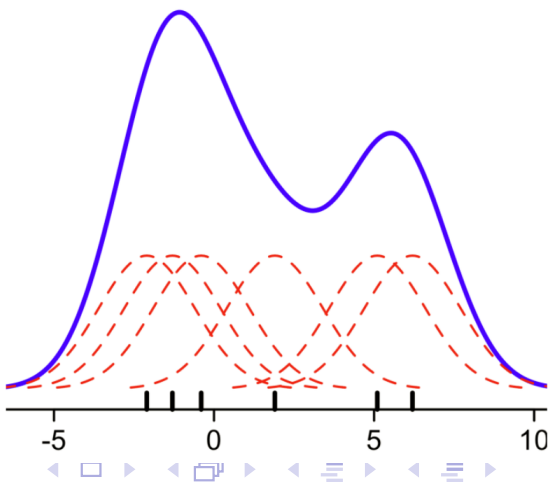
- Given data points
- choose parameterised distribution $P(x|\theta)$
- Find θ which maximises the likelihood:

$$\max L(\theta|x) = \max \Pi P(x|\theta)$$



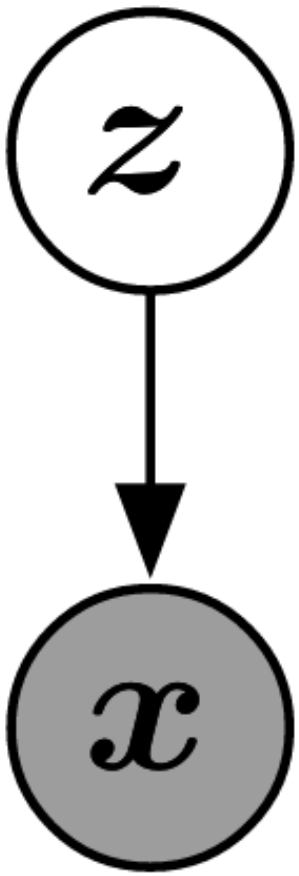
Kernel density

- Place many Gaussians on data point, call sum a PDF



Different idea for generative model structure

- **Given**: $P(x)$ is the distribution of the data
  - High dimensionality (**x** images)
- **Choose**: $P(z)$ is some easily samplable distribution (e.g. Gaussian)
  - **z** is laten variables in with low dimensionality



Generative model!

What if we could find some mapping $F(z)$ so that $P(F(z)) = P(x)$?

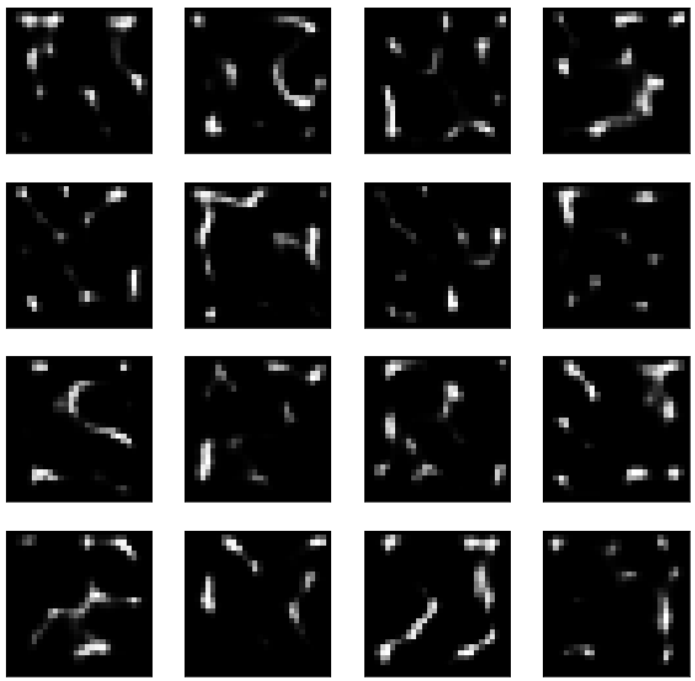
Autoencoder

Doesn't autoencoder give us exactly this?

Generating images with autoencoder

From checkpoint 1, sample from random values in 7x7 laten space

```
z_random = np.random.rand(16, np.prod(encoding_shape))
gen= decoder.predict(z_random)
```



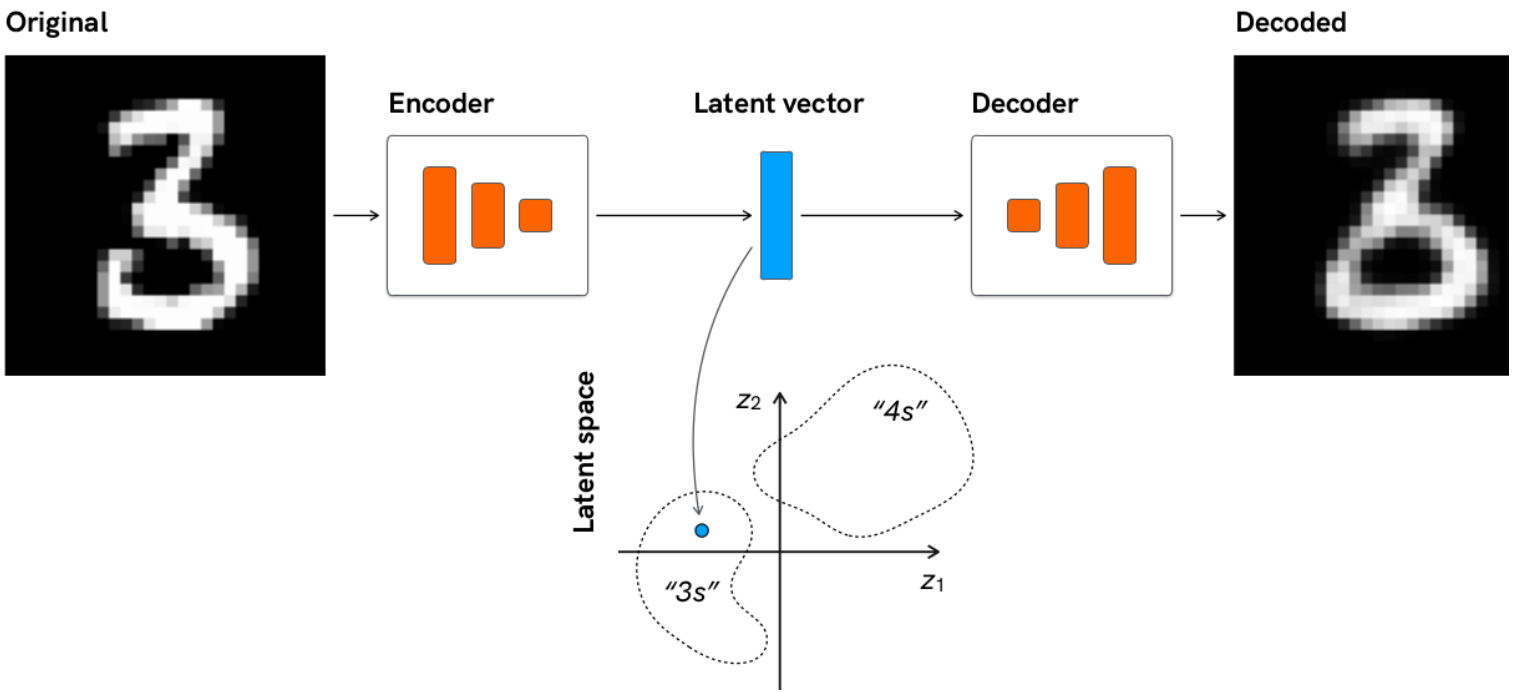
Generated images from decoder

Looks like random noise, but why?

- AE is deterministic not probabilisitic
  - Encoding exact data
  - Not a probability distribution
- Variables in 7x7 grid should belong to image domain

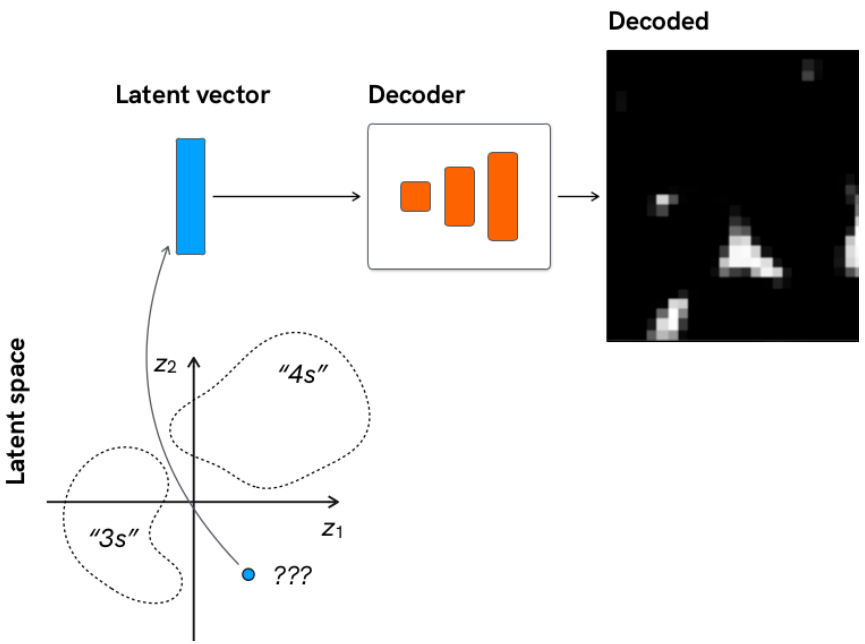
Generating images with autoencoder

Auto encoder is good at memorising



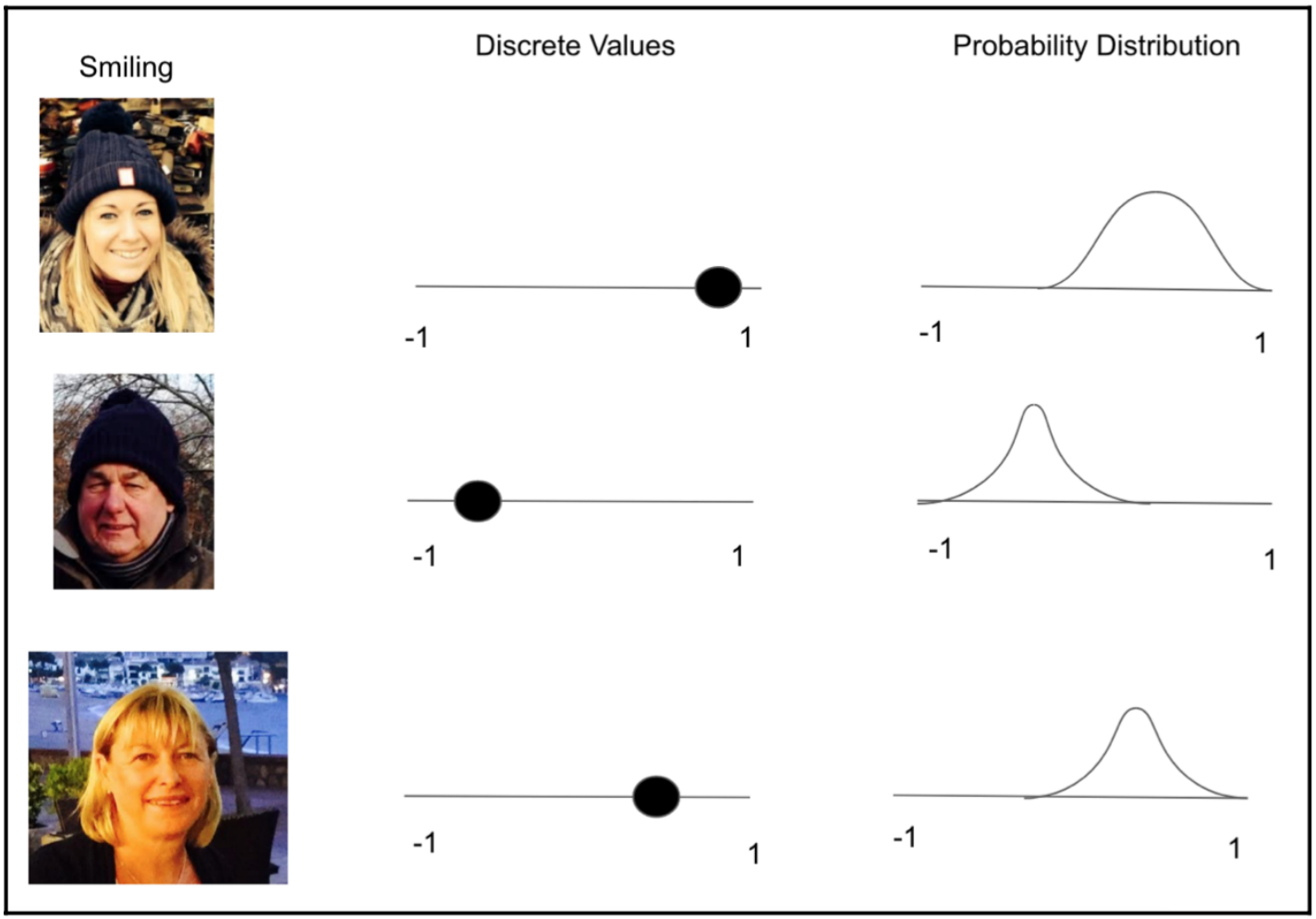
Generating images with autoencoder

Laten space is not continuous, so can't interpolate!



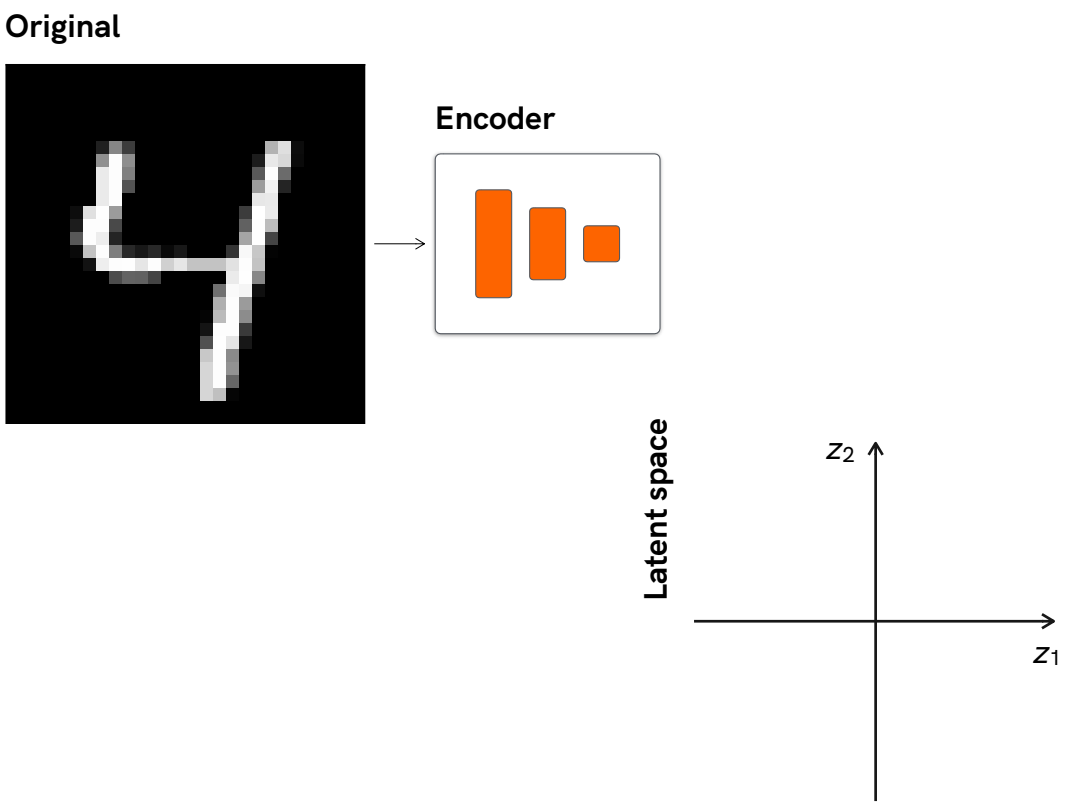
Proposed Solution

We need to transition to describing attributes in a probabilistic way



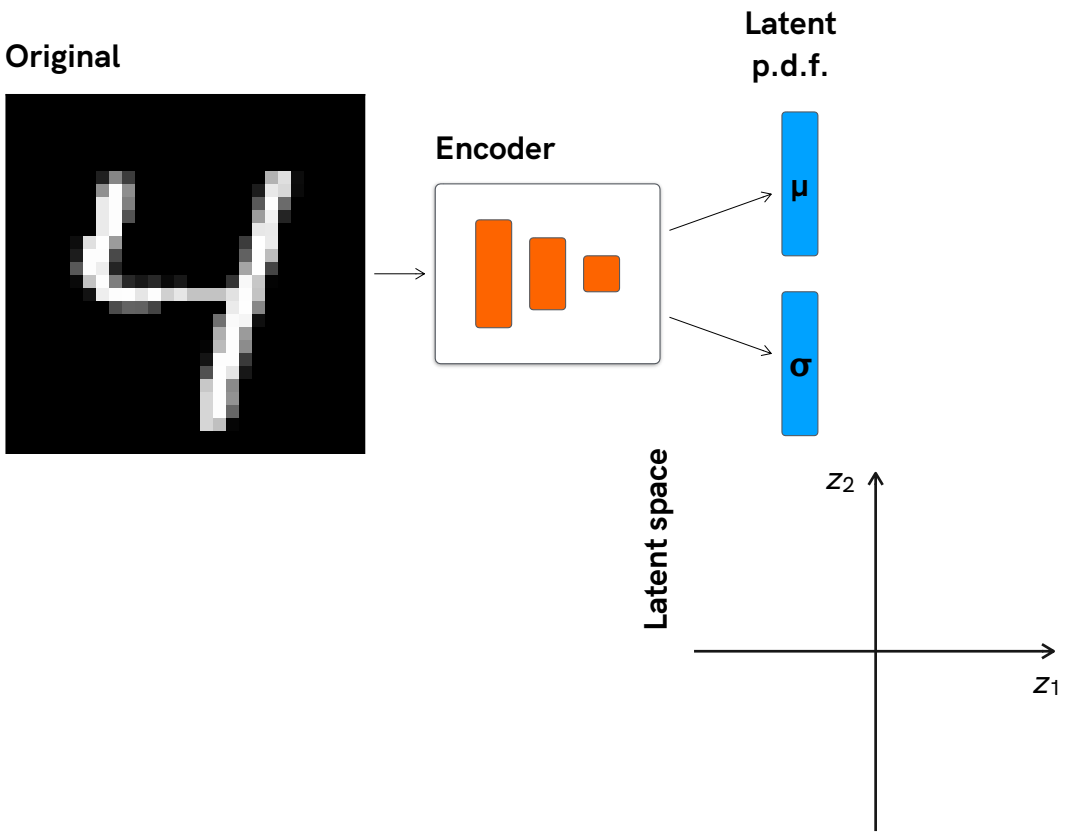
Variational autoencoders

- Variational auto-encoders (VAEs) introduces **random sampling** as a way to ensure continuity of latent space



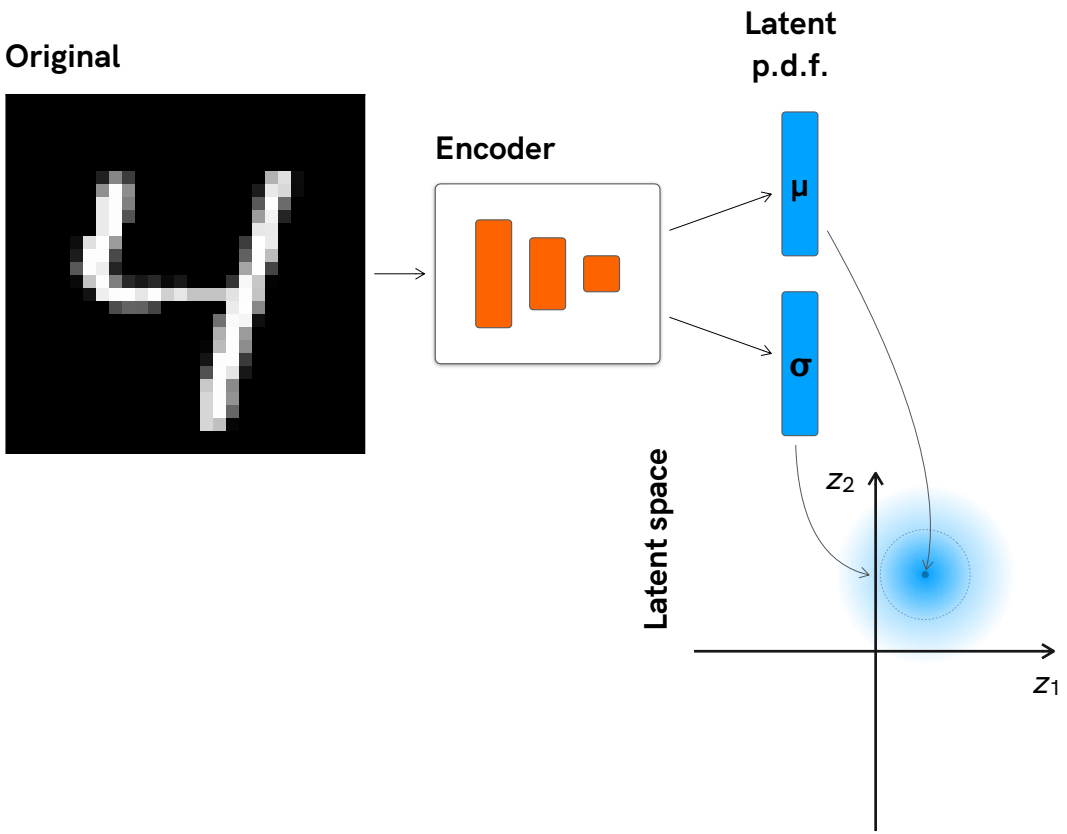
Variational autoencoders

- Instead of specifying latent vector, encoder parametrises a Gaussian **probability density function (p.d.f.)** in the latent space — $N(\mu, \sigma)$



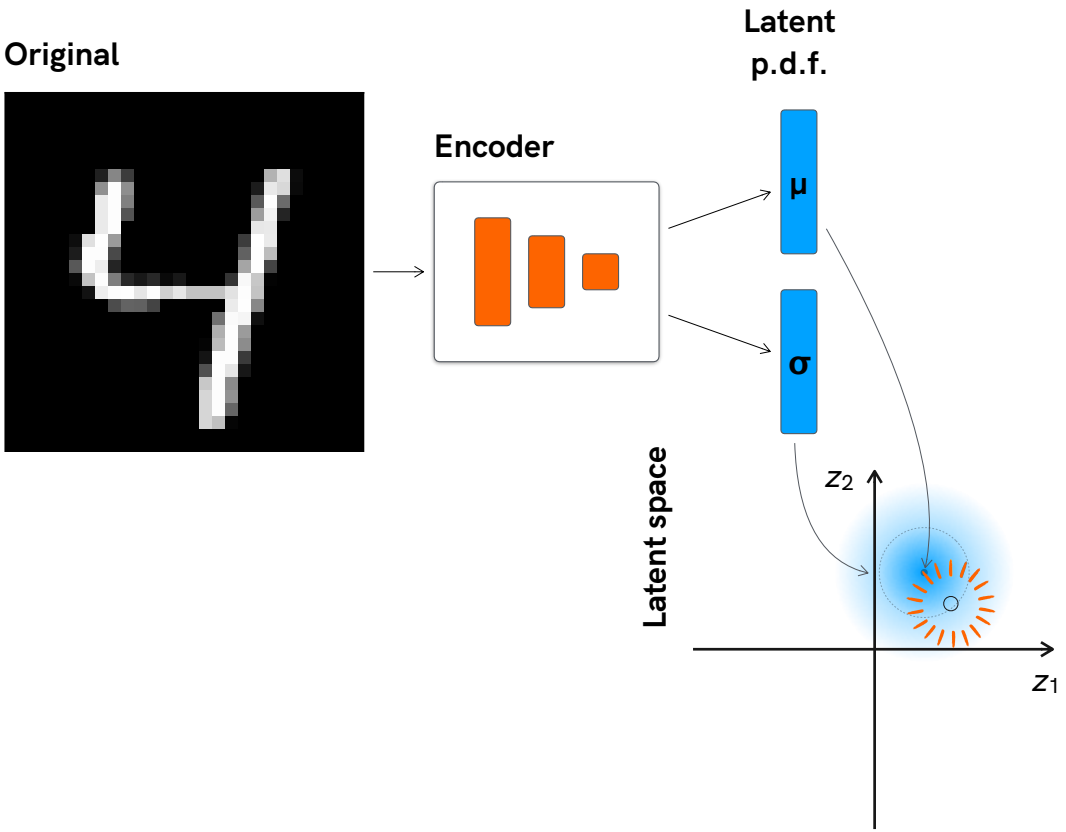
Variational autoencoders

- Instead of specifying latent vector, encoder parametrises a Gaussian **probability density function (p.d.f.)** in the latent space — $N(\mu, \sigma)$



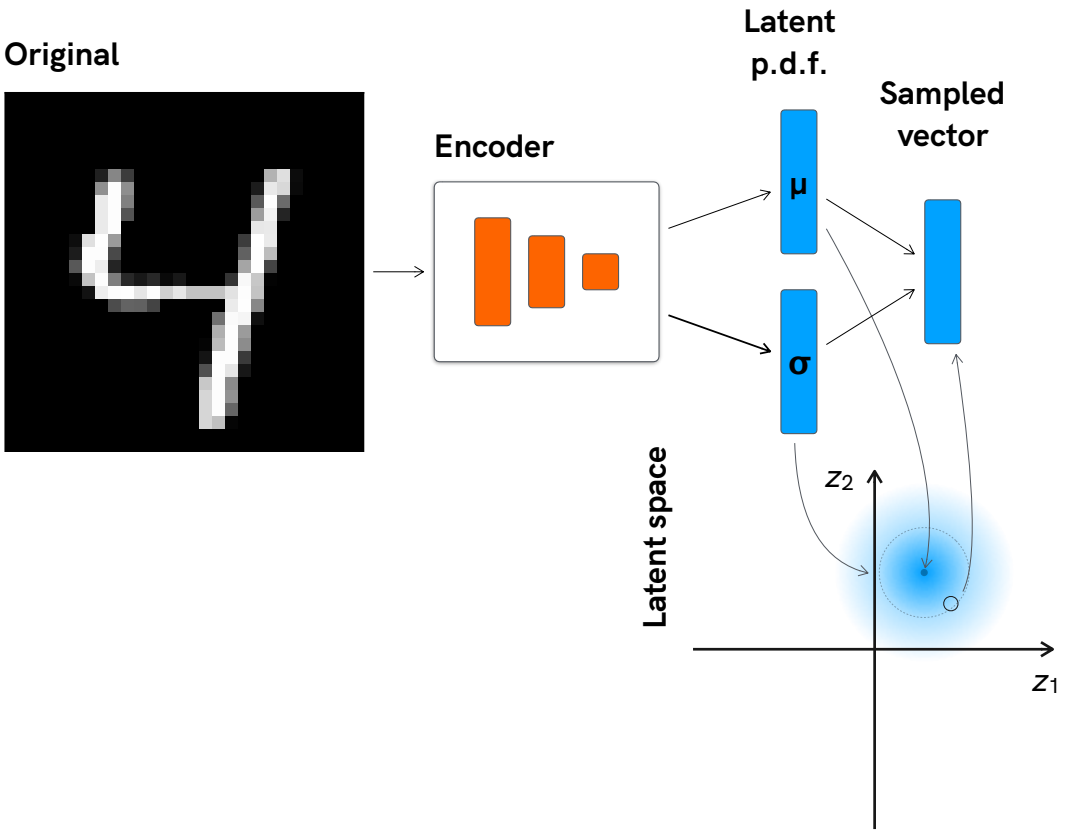
Variational autoencoders

- Instead of specifying latent vector, encoder parametrises a Gaussian **probability density function (p.d.f.)** in the latent space — $N(\mu, \sigma)$ — from which latent vectors are sampled



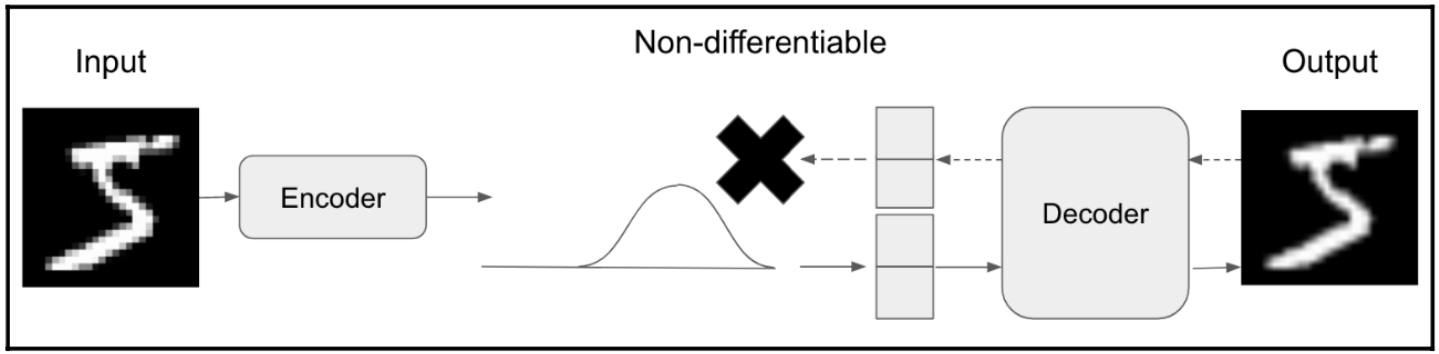
Variational autoencoders

- Instead of specifying latent vector, encoder parametrises a Gaussian **probability density function (p.d.f.)** in the latent space — $N(\mu, \sigma)$ — from which latent vectors are sampled



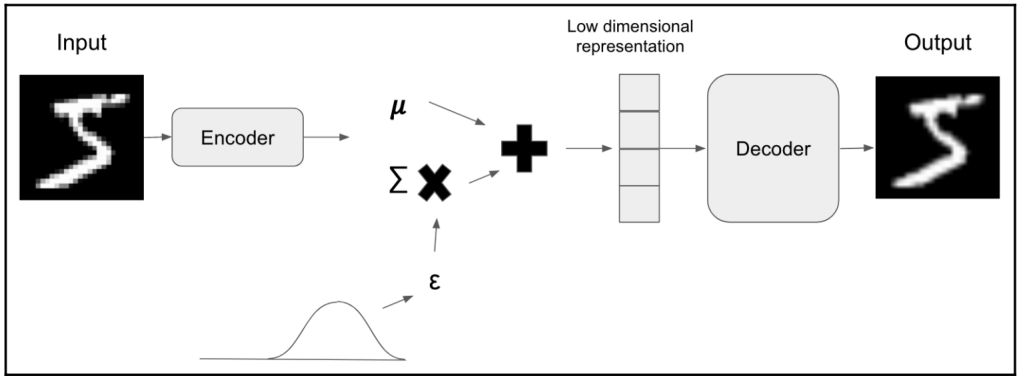
Reparameterization trick

Doing backpropagation not possible if sampling was central in VAE



Reparameterization trick: $z = \mu + \sigma\epsilon$

- Sample ϵ from a unit normal distribution
- and mean and std are differentiable, we can now train the model via simple backpropagation



Reparameterization trick with Keras

- Define sampling with reparameterization trick

```
def sample_z(args):
  mu, sigma = args
  batch     = K.shape(mu)[0]
  dim       = K.int_shape(mu)[1]
  eps       = K.random_normal(shape=(batch, dim))
  return mu + K.exp(sigma / 2) * eps
```

- We then use this with a Lambda to ensure that correct gradients are computed during the backwards pass based on our values for mu and sigma

```
z       = Lambda(sample_z, output_shape=(latent_dim, ), name='z')([mu,
    sigma])
```

VAE Loss function

VAE is trained to minimise sum of two losses

1. **Reconstruction loss:** Pixel-wise BCE or MSE (same as AEs)

$$L_R = \text{BCE or MSE}$$

2. **Kullback-Leibler loss:** Regularisation loss making p.d.f closer to unit normal
   - i.e. $N(\mu, \sigma)(0, 1)$

$$L_{KL} = \sum_{i}^{d_{\text{laten}}} \mu_i^2 + \sigma_i^2 - \log \sigma_i^2 - 1$$

Total Loss: $L = L_R + C \times L_{KL}$

- Higher C: prioritizes the quality of sampled images for lower sample diversity

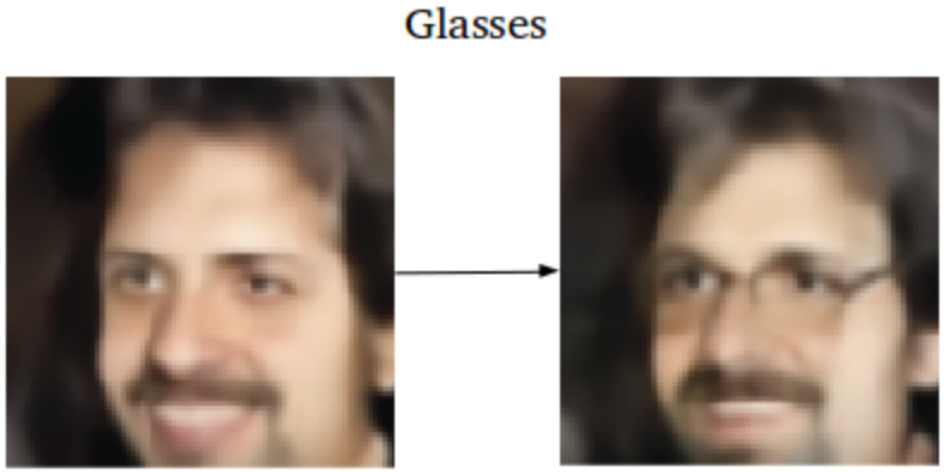
AE vs VAE

Model structure

- AE is deterministic (memorising the data)
- VAE is **probabilistic** building a posterior probability
- The output is therefore a probability distribution rather than a single value

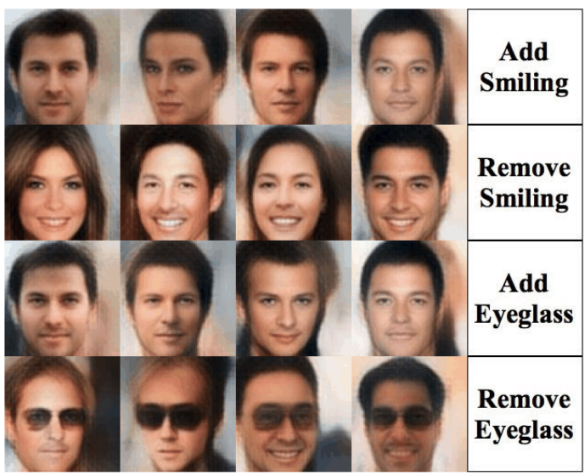
Model features

- Standard autoencoders useful to **replicate** data, so limited applications in real world
- VAEs are powerful **generative models**: `output data` != `input data`
  - For example are useful for exploring a specific variation of input data

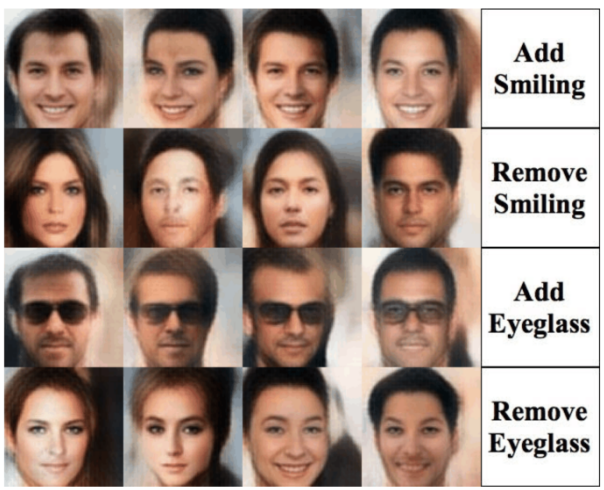


Variational autoencoders

Example interpolation between features (link)



→



Can we do better?

VAE (Explicit PDF)

- With KL loss we are actually maximising a lower bound likelihood (not the likelihood itself)
- This makes the posterior PDF an **approximation**
  - If weak prior or posterior PDF, even with perfect training and infinite data, gap to true likelihood can result in bad model.

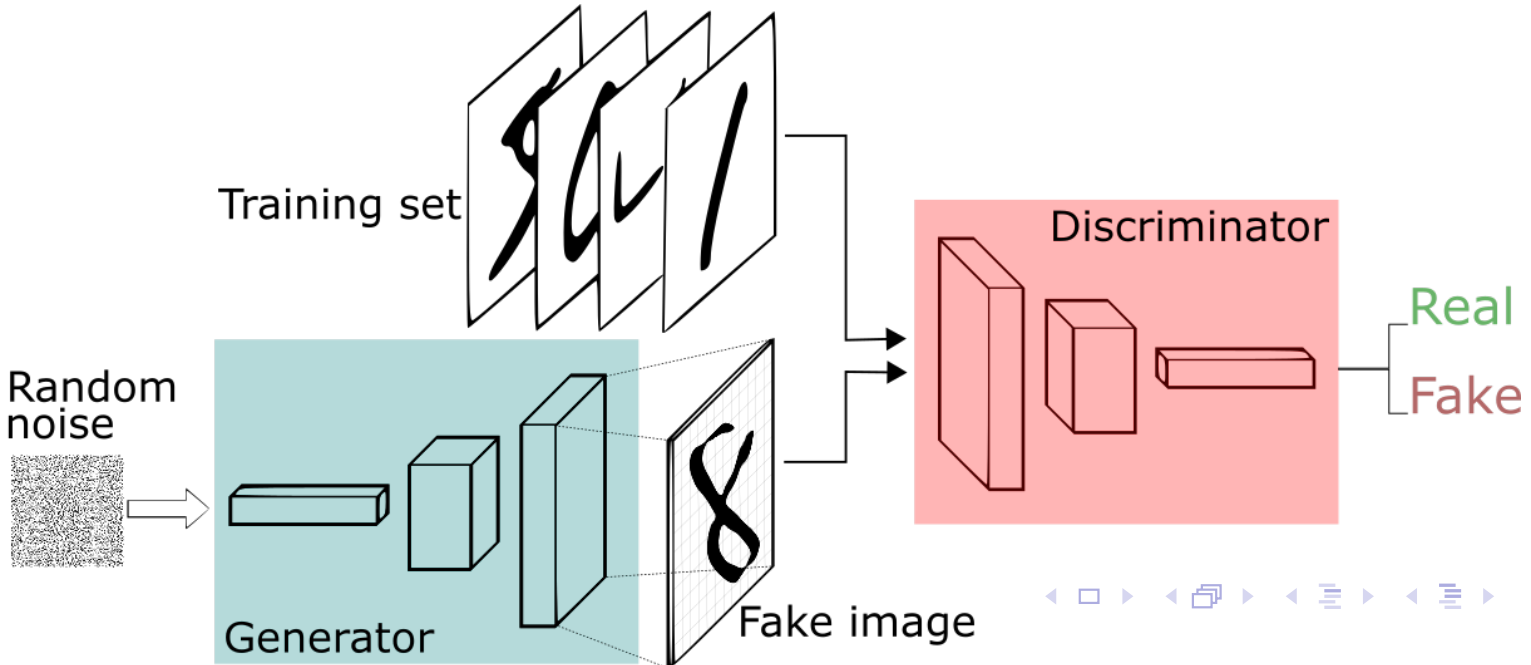
GAN (Implicit PDF)

- Generative adversarial network trained **without** need to **explicitly define PDF**.
- Usually considered to provide better quality reconstruction than VAEs

Generative Adversial Networks (GANs)

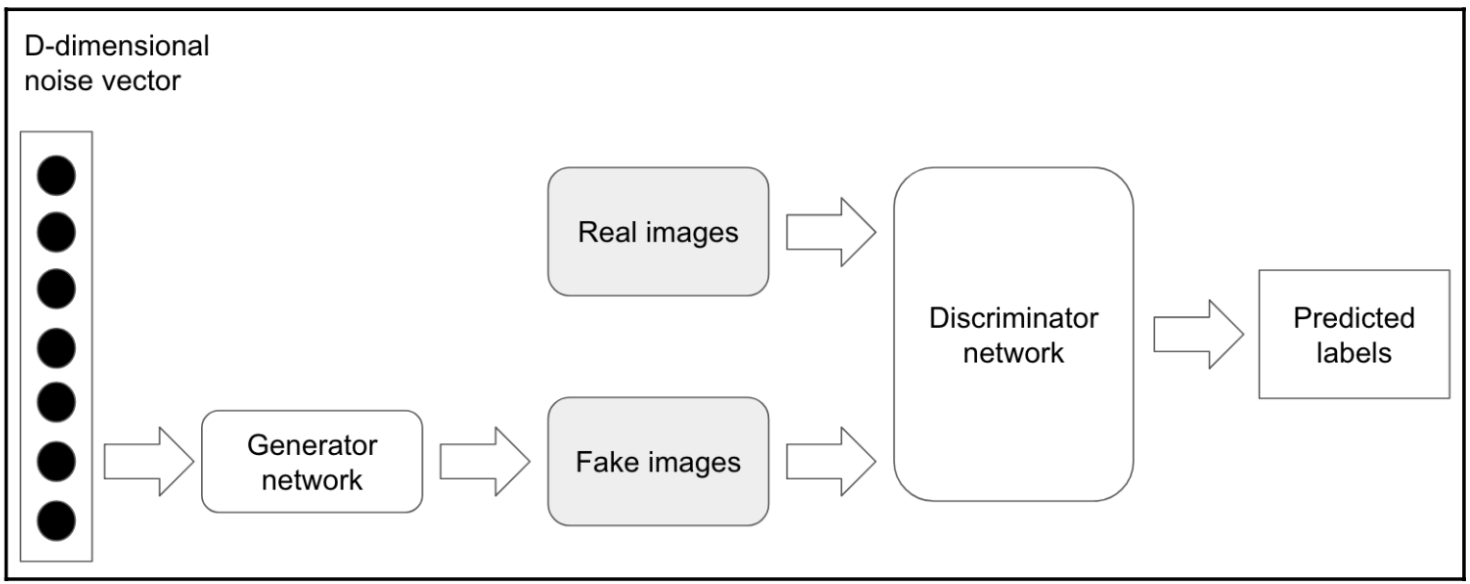
GANs are generative models that try to learn the model to generate the input distribution as realistically as possible.

- A GAN is an unsupervised learning technique and consists of two neural networks:
  - A generative model **Generator** (G) generates new data points from some random uniform distribution.
  - A discriminative model **Discriminator** (D) identifies fake data produced by Generator from the real data.



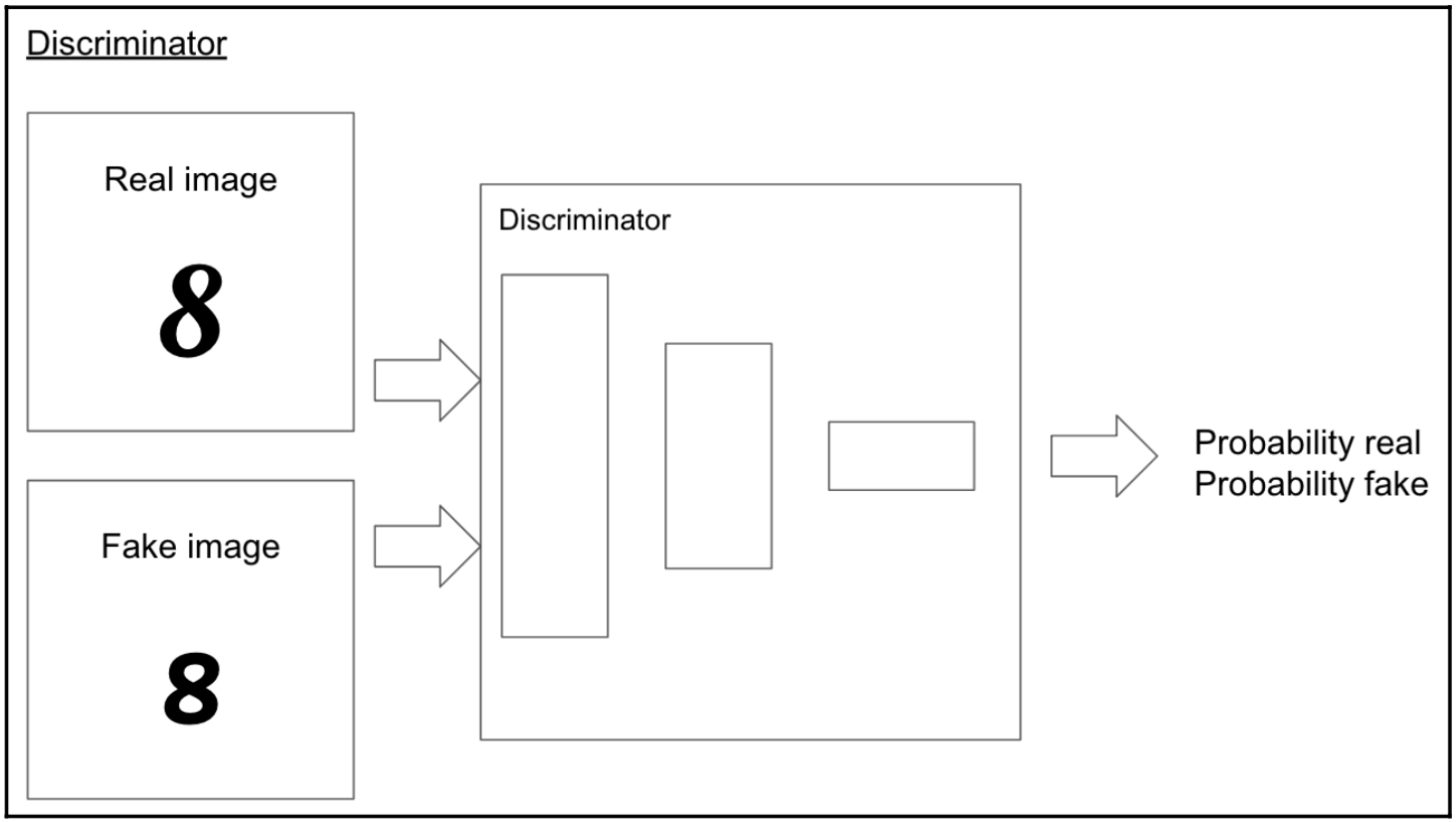
Steps that a GAN takes

1. Random numbers are fed into generator and an image is generated
2. The generated image is fed into the discriminator along with other images taken from the real dataset
3. The discriminator considers all of the images fed into it and returns a probability as to wheter it thinks the image is real or fake



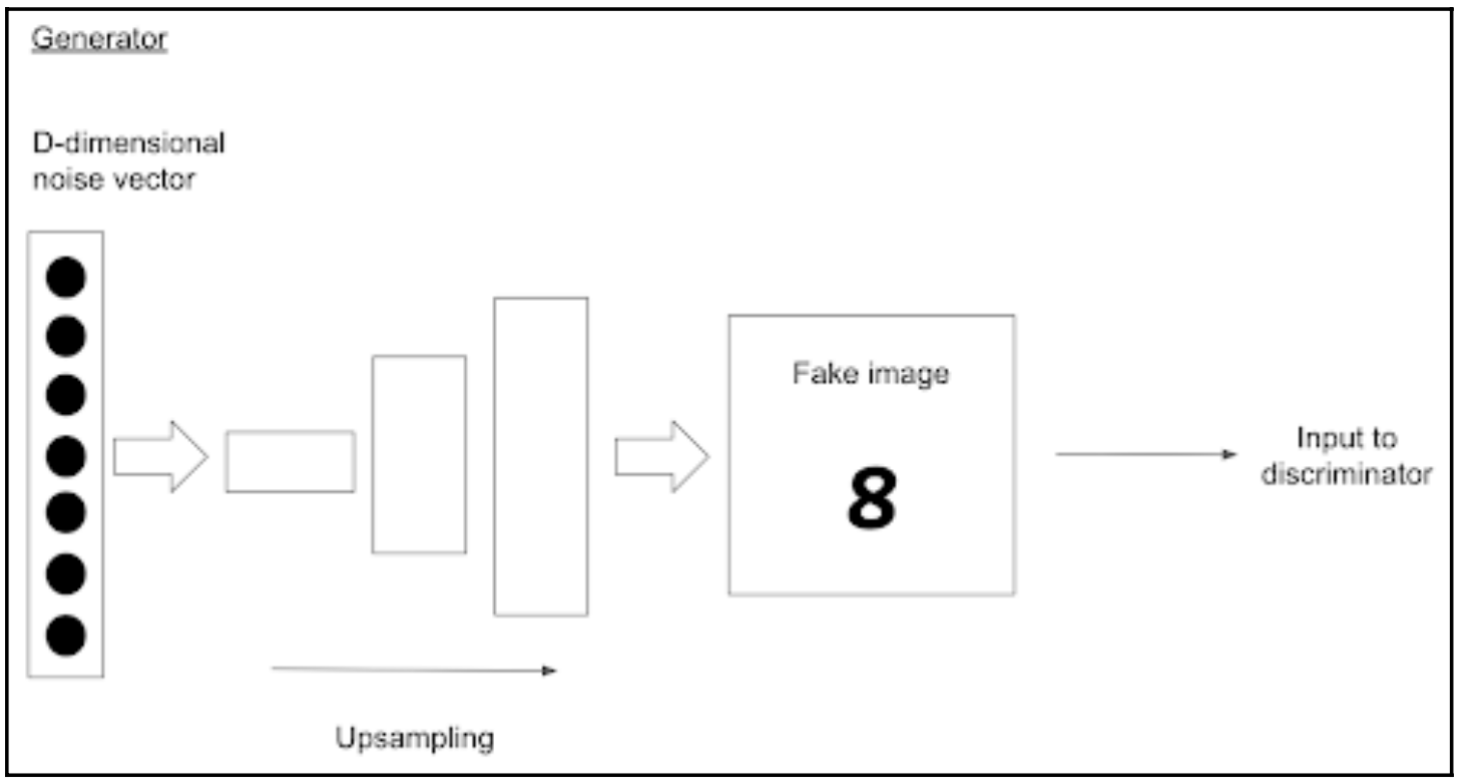
Discriminator network

- The discriminator network is simply a standard convolutional network that categorises images being feed to it.
- It performs downsampling and classifies the images in a binary way, labeling each image as real or fake



Generator network

- The generator network is essentially the reverse of a convolutional network.
- It takes the random noise and performs upsampling in order to output the image



MinMax Game on GANs

- **Generator network:** try to fool the discriminator by generating real-looking images
- **Discriminator network:** try to distinguish between real and fake images

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

- $D(\theta_d)$ wants to maximize objective such that D(x) is close to 1 (real) and $D(G(z))$ is close to 0 (fake)
- $G(\theta_d)$ wants to minimize objective sush that $D(G(z))$ is close to 1
  - discriminator is fooled into thinking generated G(z) is real

Training GANs

Because a GAN contains two separetely trained networks, its training algorithm must address 2 complications:

- GANs must juggle two different kinds of training (generator and discriminator)
- GAN convergence is hard to identify

GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs
2. The generator trains for one or more epochs
3. Repeat the 2 steps to continue to train the generator and discriminator networks

Training GANs

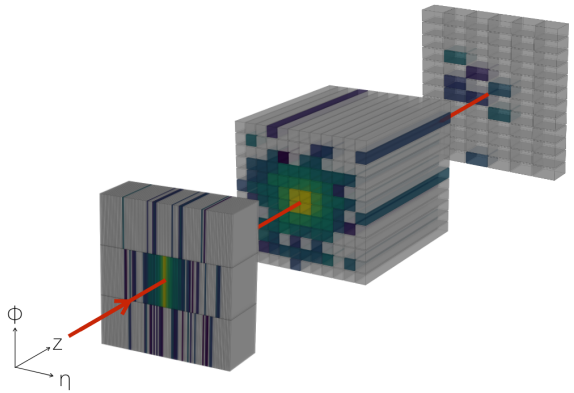
- K keep the **generator constant** during the **discriminator training** phase. This because the discriminator has to learn how to recognize the generator's flaws.
- We keep the **discriminator constant** during the **generator training** phase. Otherwise the generator would be trying to hit a moving target and might never converge.

Convergence is difficult

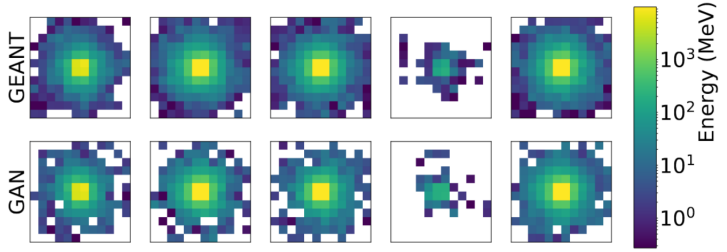
- As the generator improves with training, the discriminator performance get worse because it cannot easily tell the difference between real and fake
- If the generator succeeds perfectly, then the discriminator has a 50\
- This poses a problem for convergence of the GAN: the discriminator feedback gets less meaningful over time
- If the GAN continues training past the point where the discriminator is giving completely random feedback, then the generator starts to train on junk feedbacks and its own quality may **collapse**

Use in particle physics

- Physics-based simulation of particles interactions with matter using **Geant4 is *very computationally expensive** (as you will see in DAML)
- Possible to use GANS to generate particle showers (https://arxiv.org/pdf/1712.10321.pdf)



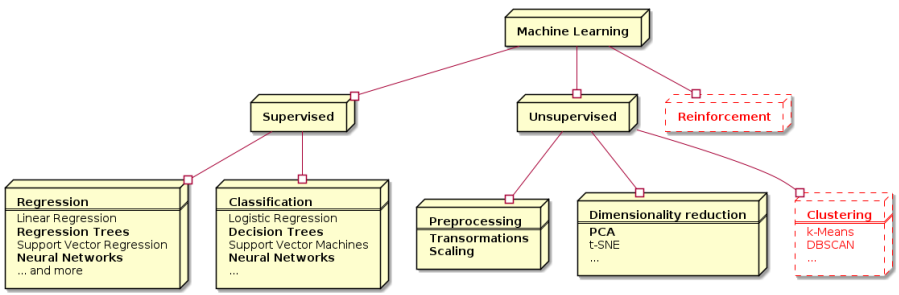
Simulator	Hardware	Batch Size	ms/shower
GEANT4	CPU	N/A	1772
CALOGAN	CPU	1	13.1
		10	5.11
		128	2.19
		1024	2.03
	GPU	1	14.5
		4	3.68
		128	0.021



- Speed up simulation by $O(10^5)$
- Paradigm shift through ML

Thanks for attention

- Discussed in 1st lecture that ML topic is HUGE!
- We covered
  - Decision trees
  - DNNs
  - CNNs
  - AEs, VAEs
  - and GANs
- Hopefully learned something new!



End

- Reminder on feedback:
  - Upload the MS form for Mid semester feedback into the "have your say" section in Learn for the semester 1 and year courses.
  - Deadline is 29th October noon
- Project
  - Will try finalise this in Wednesdays CP

The deadline for the responses is 29th October at noon