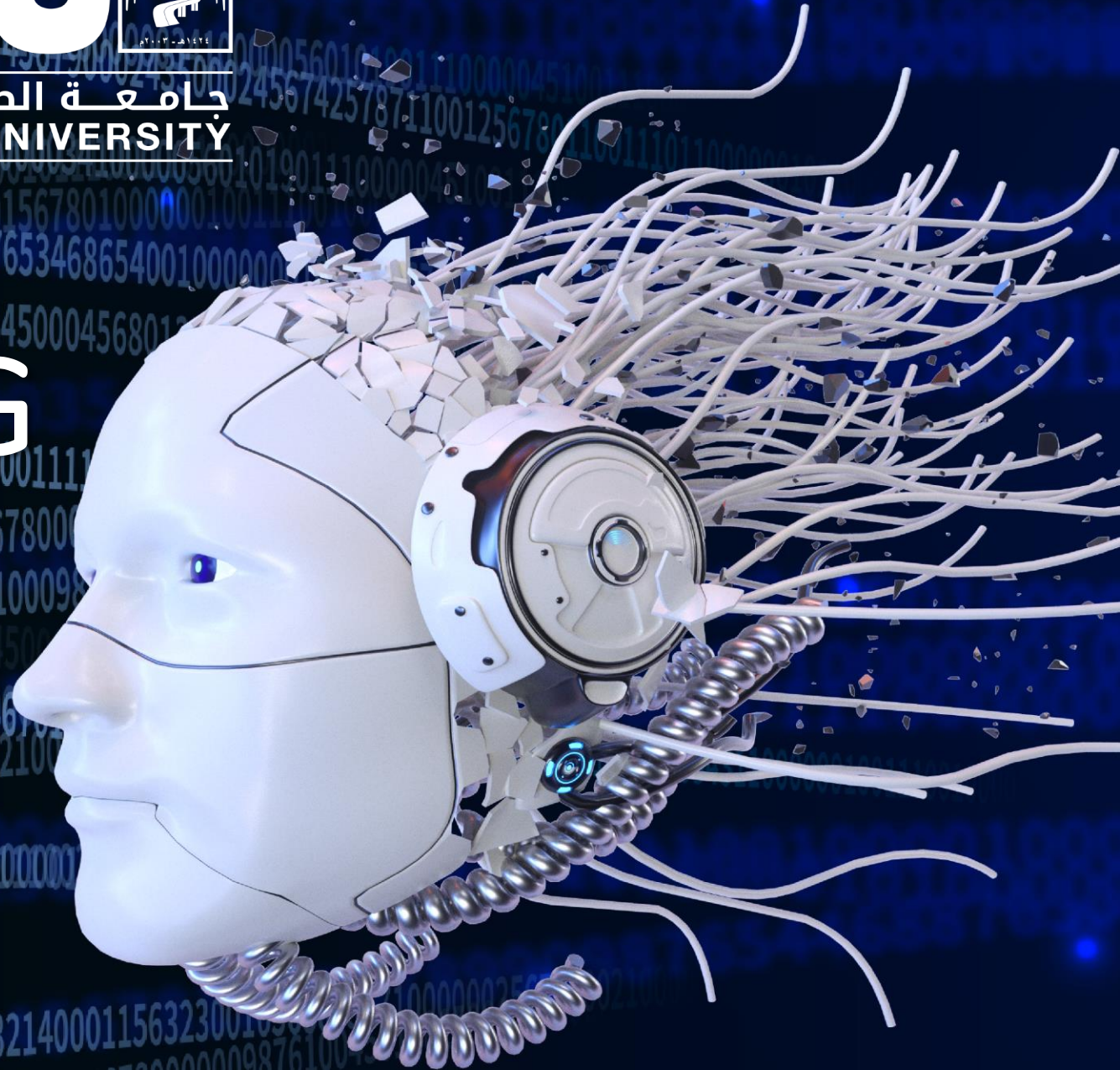




جامعة الطائف
TAIF UNIVERSITY

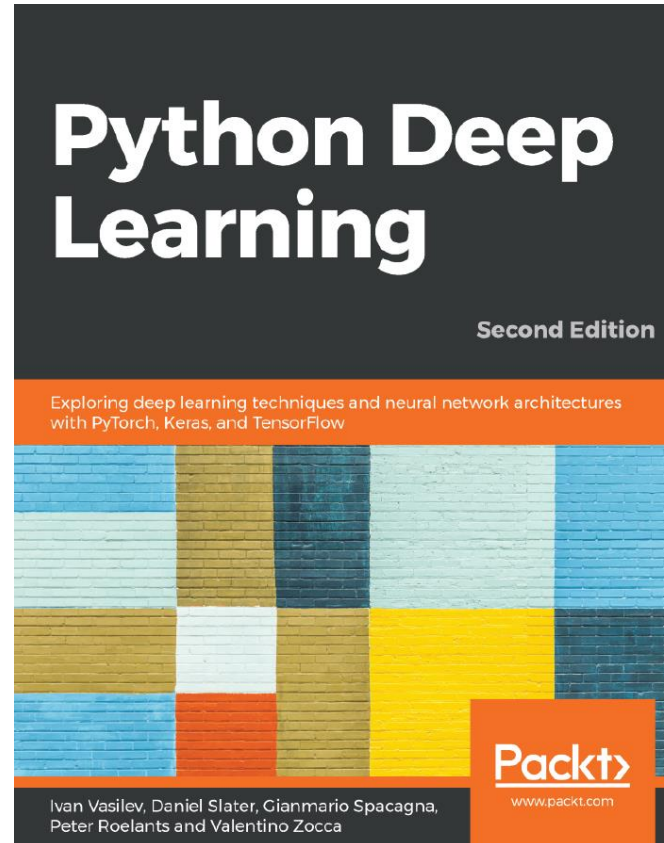
DEEP LEARNING

Assoc. Prof. Dr. Salha Alzahrani
Lecture Notes for MSc. in Data Science



In this chapter, we will cover the following topics:

- Intuition and justification of generative models
- Autoencoders
- Variational Autoencoders (VAEs)
- Generative Adversarial networks (GANs)
- Generating new MNIST digits with VAEs
- Generating new MNIST images with GANs



Intuition and justification of generative models

- So far, we've used neural networks as **discriminative models**. This simply means that given input data, a discriminative model will map it to a certain label (in other words, a **classification**).
- A typical example is the classification of MNIST images in 1 of 10-digit classes, where the neural network maps the input data features (pixel intensities) to the digit label.
- We can also say this in another way, a discriminative model gives us the probability of y (class), given x (input) :

$$P(Y|X = x)$$

- On the other hand, a **generative model learns the distribution of the classes**. You can think of it as the opposite of what the discriminative model does. Instead of predicting the class probability, y , given certain input features, it tries to predict the probability of the input features, given a class:

$$y \rightarrow P(X|Y = y)$$

- For example, a generative model will be able to create an image of a handwritten digit, given the digit class. Since we only have 10 classes, it will be able to generate just 10 images. But we used this example just to better illustrate the concept.

Intuition and justification of generative models

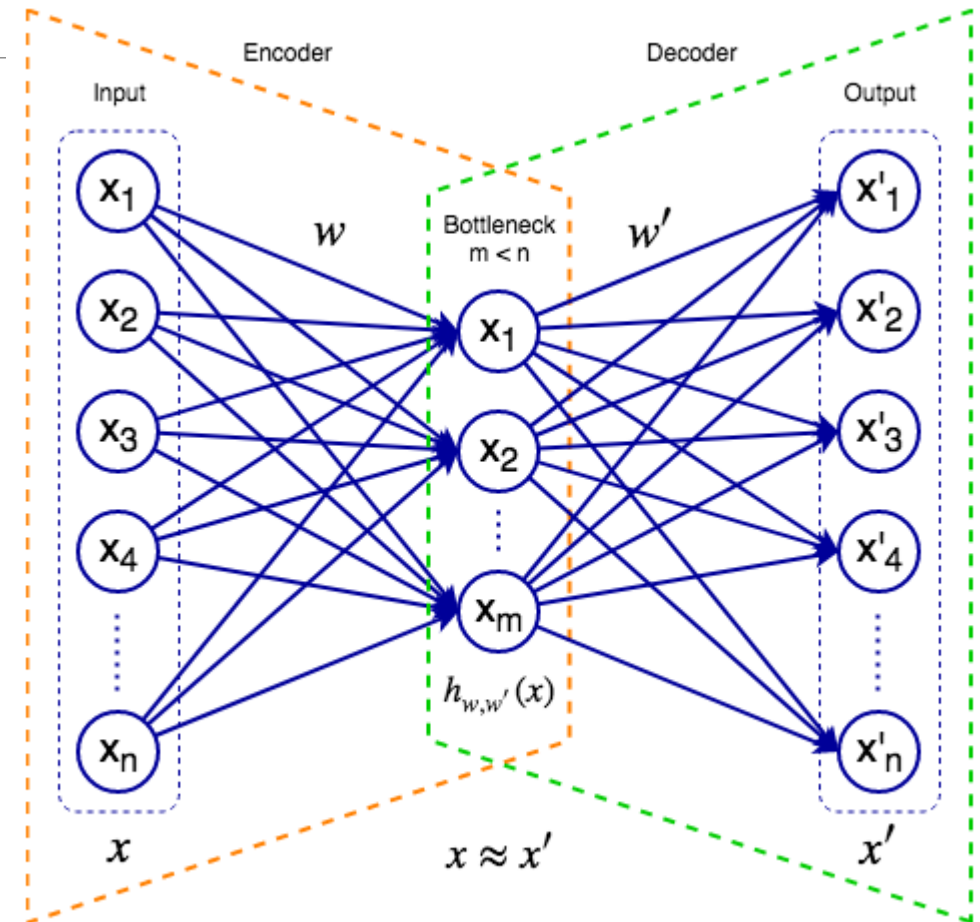
- In reality, the y "class" could be an arbitrary tensor of values, and the model would be able to generate an unlimited number of images with different features.
- Two of the most popular ways to use neural networks in a generative way are:
 - Variational Autoencoders (VAEs)
 - Generative Adversarial networks (GANs).

Autoencoders

- An autoencoder is a feed-forward neural network that tries to reproduce its input.
- In other words, the target value (label) of an autoencoder is equal to the input data, $y^i = x^i$, where i is the sample index.
- We can formally say that it tries to learn an **identity function**, (a function that repeats its input).

$$h_{w,w'}(x) = x$$

- Since our "labels" are just the input data, the autoencoder is an **unsupervised** algorithm.



Autoencoders

- An autoencoder consists of an **input**, hidden (or **bottleneck**), and **output** layers. Although it's a single network, we can think of it as a virtual composition of two components:
 - **Encoder:** Maps the input data to the network's internal representation. For the sake of simplicity, in this example the encoder is a single, fully-connected hidden bottleneck layer. The internal state is just its activation vector. In general, the encoder can have multiple hidden layers, including convolutional layers.
 - **Decoder:** Tries to reconstruct the input from the network's internal data representation. The decoder can also have a complex structure, which typically mirrors the encoder.

Autoencoders

- We can train the autoencoder by minimizing a loss function, which is known as the **reconstruction error** which measures the distance between the original input and its reconstruction :

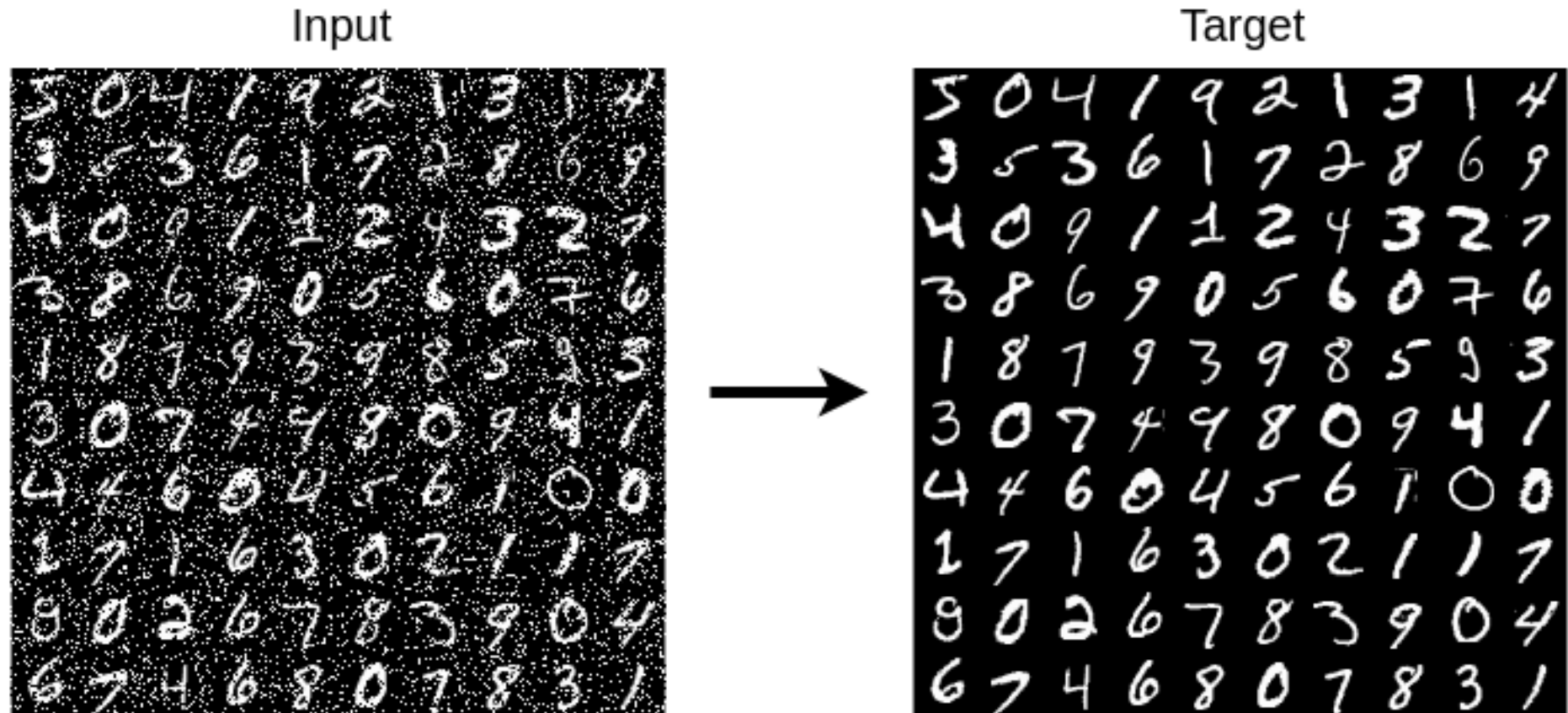
$$\mathcal{L} = (x, x')$$

- We can minimize it in the usual way with gradient descent and backpropagation. Depending on the approach, we can use either **mean square error (MSE)** or **binary cross-entropy** (such as cross-entropy, but with two classes) as reconstruction errors.
- At this point, you might wonder what the point of the autoencoder is, since it just repeats its input.
- However, we are not interested in the network output, but in its **internal data representation (which is also known as representation in the latent space)**.
- The latent space contains hidden data features, which are not directly observed, but are inferred by the algorithm instead.

Autoencoders

- The key is that **the bottleneck layer has fewer neurons than the input/output ones**. There are two main reasons for this:
 - Because the network tries to reconstruct its input from a smaller feature space, it learns a compact representation of the data. You can think of it as a compression (but not lossless).
 - By using fewer neurons, the network is forced to learn only the most important features of the data. To illustrate this concept, let's look at denoising autoencoders, where we intentionally use corrupted input data, but non-corrupted target data during training. For example, if we train a denoising autoencoder to reconstruct MNIST images, we can introduce noise by setting max intensity (white) to random pixels of the image (the following screenshot). To minimize the loss with the noiseless target, the autoencoder is forced to look beyond the noise in the input and learn only the important features of the data. However, if the network had more hidden neurons than input, it could overfit on the noise. With the additional constraint of fewer hidden neurons, it has nowhere to go but to try to ignore the noise. Once trained, we can use a denoising autoencoder to remove the noise from real images.

Autoencoders



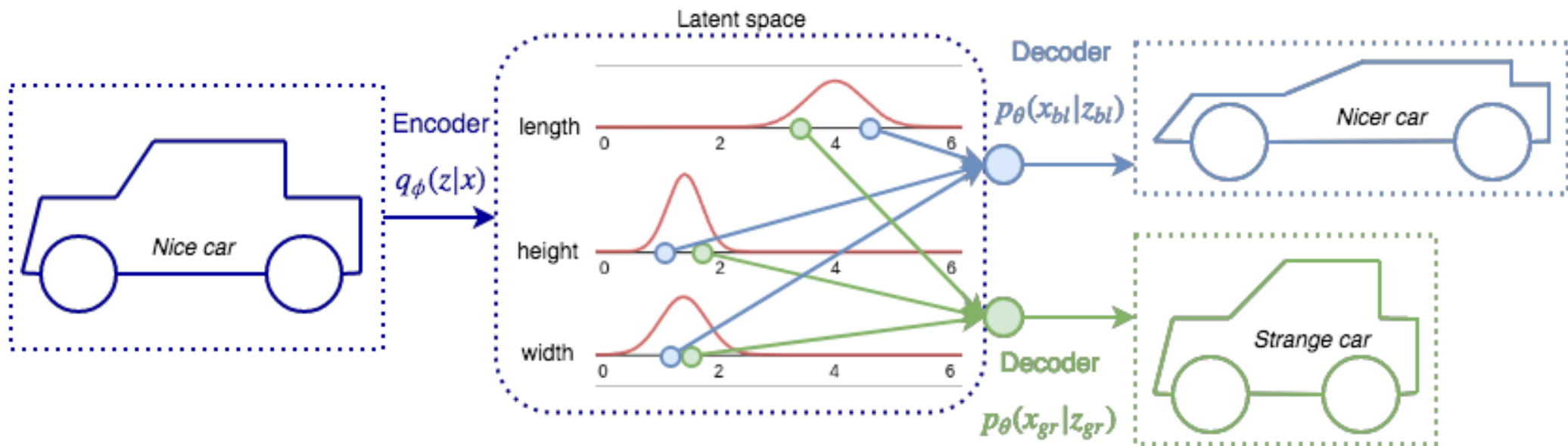
Denoising autoencoder input and target

Variational autoencoders

- The encoder maps each input sample to the latent space and each attribute of the latent representation has a discrete value. That means that an input sample can have only one latent representation. Therefore, the decoder can reconstruct the input in only one possible way. In other words, we can generate a single reconstruction of one input sample. But we don't want this. Instead, we want to generate new images that are different from the original. Enter VAEs.
- A VAE can describe the latent representation in probabilistic terms. That is, instead of discrete values, we'll have a probability distribution for each latent attribute, making the latent space continuous. This makes it easier for random sampling and interpolation.
- Let's illustrate this with an example. Imagine that we try to encode an image of a vehicle and our latent representation has n attributes (n neurons in the bottleneck layer). Each attribute represents one vehicle property, such as length, height, and width. Say that the average vehicle length is four meters. Instead of the fixed value, the VAE can decode this property as a normal distribution with a mean of 4 (the same applies for the others). Then, the decoder can choose to sample a latent variable from the range of its distribution.

Variational autoencoders

- For example, it can reconstruct a longer and lower vehicle, compared to the input. In this way, the VAE can generate an unlimited number of modified versions of the input:



An example of a variational encoder, sampling different values from the distribution ranges of the latent variables

Variational autoencoders

Let's formalize this:

- We'll denote the encoder with $q_{\phi}(z|x)$, where ϕ are the weights and biases of the network, x is the input, and z is the latent space representation. The encoder output is a distribution (for example, Gaussian) over the possible values of z , which could have generated x .
- We'll denote the decoder with $p_{\theta}(x|z)$, where θ are the decoder weights and biases. First, z is sampled stochastically (randomly) from the distribution. Then, it's sent through the decoder, whose output is a distribution over the possible corresponding values of x .

Variational autoencoders

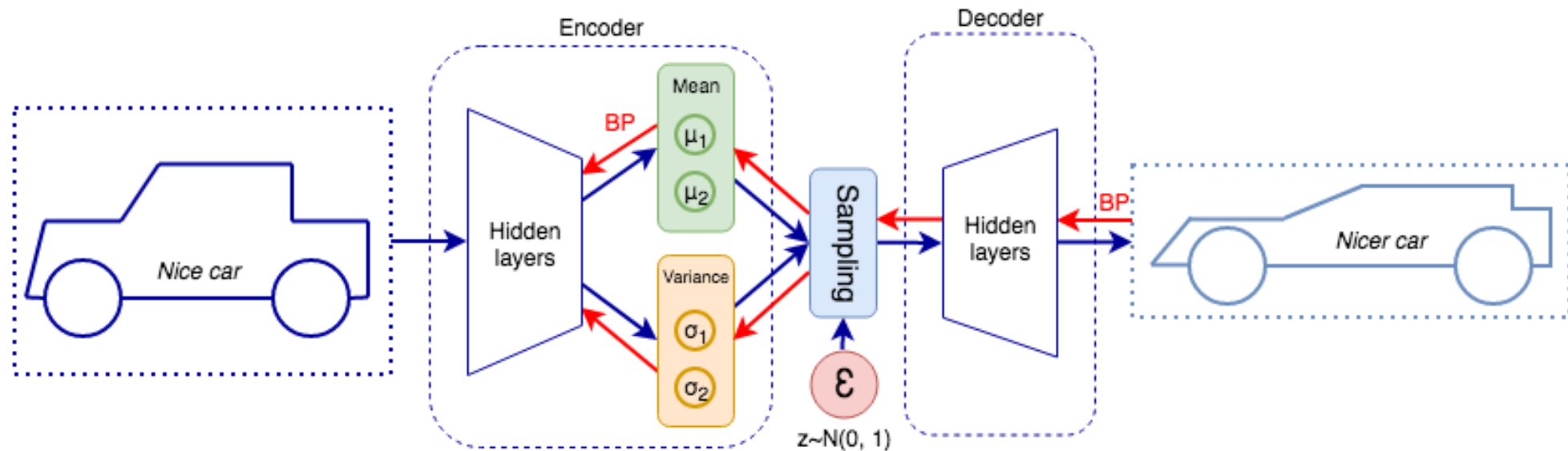
- The VAE uses a special type of loss function with two terms:

$$L(\theta, \varphi; x) = -D_{KL}(q_{\varphi}(z|x)||p_{\theta}(z)) + E_{q_{\varphi}(z|x)}[\log(p_{\theta}(x|z))]$$

The first is the Kullback-Leibler divergence between the probability distribution $q_{\phi}(z|x)$ and the expected probability distribution, $p(z)$. It measures how much information is lost, when we use $q_{\phi}(z|x)$ to represent $p(z)$ (in other words, how close the two distributions are). It encourages the autoencoder to explore different reconstructions. The second is the reconstruction loss, which measures the difference between the original input and its reconstruction. The more they differ, the more it increases. Therefore, it encourages the autoencoder to better reconstruct the data.

Variational autoencoders

- To implement this, the bottleneck layer won't directly output the latent state variables. Instead, it will output two vectors, which describe the mean and variance of the distribution of each latent variable:



Variational encoder sampling

Variational autoencoders

Once we have the mean and variance distributions, we can sample a state, z , from the latent variable distributions and pass it through the decoder for reconstruction. But we cannot celebrate yet, because this presents us with another problem: backpropagation doesn't work over random processes such as the one we have here. Fortunately, we can solve this with the so-called **reparameterization trick**.

First, we'll sample a random vector, ϵ , with the same dimensions as z from a Gaussian distribution (the ϵ circle in the preceding figure).

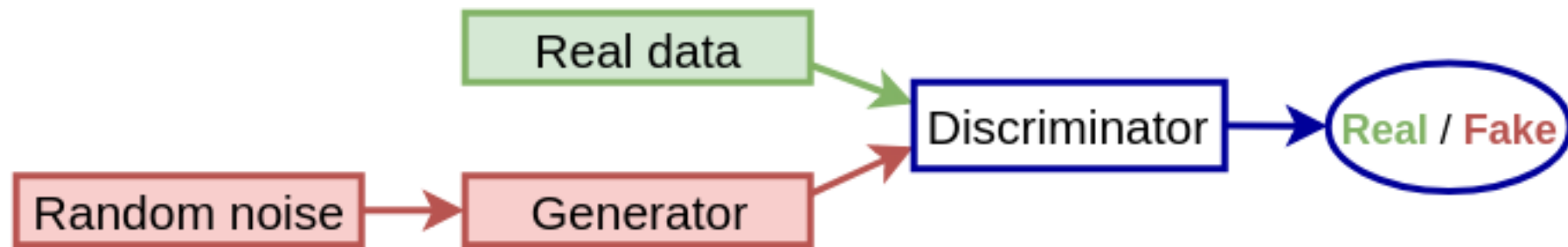
Then, we'll shift it by the latent distribution's mean, μ , and scale it by the latent distribution's variance, σ :

$$z = \mu + \sigma \odot \epsilon$$

In this way, we'll be able to only optimize the mean and variance (red arrows) and we'll omit the random generator from the backward pass. At the same time, the sampled data will have the properties of the original distribution.

Generative Adversarial networks

- In this section, we'll talk about the most popular generative model today: the GANs framework.
- It was first introduced in 2014 in the landmark paper Generative Adversarial Nets (<http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>) by Ian J. Goodfellow, et al.
- The GANs framework can work with any type of data, but its most popular application by far is to generate images.



A GAN system

Generative Adversarial networks

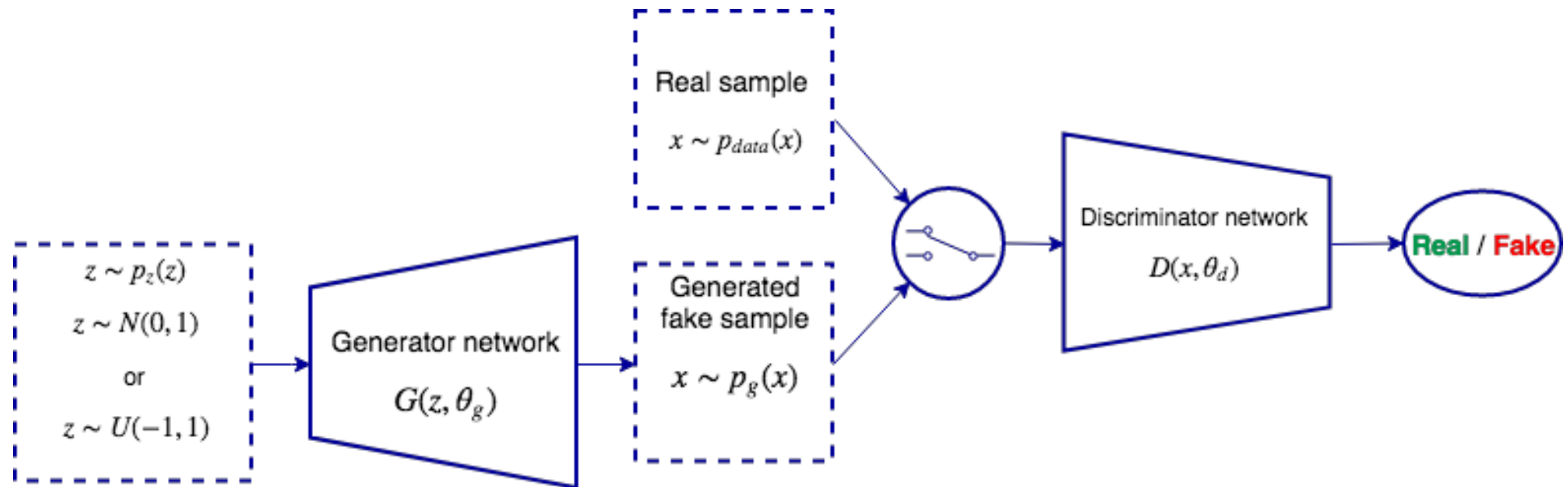
- A GAN is a system of **two components** (neural networks):
 - **Generator:** This is the generative model itself. It takes a probability distribution (random noise) as input and tries to generate a realistic output image. Its purpose is similar to the decoder part of the VAE.
 - **Discriminator:** This takes two alternating inputs: the real images of the training dataset or the generated fake samples from the generator. It tries to determine whether the input image comes from the real images or the generated ones.

Generative Adversarial networks

- The two networks are trained together as a system :
 - On the one hand, **the discriminator** tries to get better at distinguishing between the real and fake images.
 - On the other hand, **the generator** tries to output more realistic images, so it could "deceive" the discriminator into thinking that the generated image is real. To use the analogy in the original paper, you can think of the generator as a team of counterfeiters, trying to produce fake currency. Conversely, the discriminator acts as a police officer, trying to capture the fake money, and the two are constantly trying to deceive each other (hence the name adversarial). The ultimate goal of the system is to make the generator so good that the discriminator wouldn't be able to distinguish between the real and fake images. Even though the discriminator does classification, a GAN is still unsupervised, since we don't need labels for the images.

Generative Adversarial networks

- Our main goal is for the generator to produce realistic images and the GAN framework is a vehicle for that goal. We'll train the generator and the discriminator **separately** and **sequentially** (one after the other), and **alternate between the two phases multiple times**.



Detailed example of a Generative Adversarial network

Generative Adversarial networks

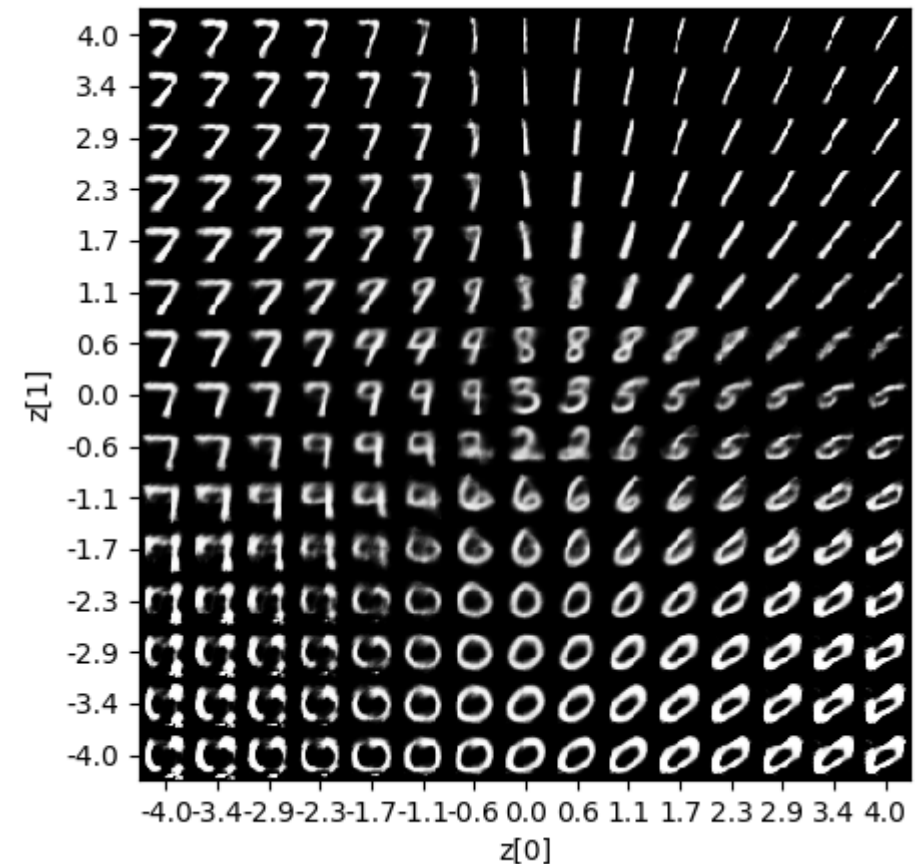
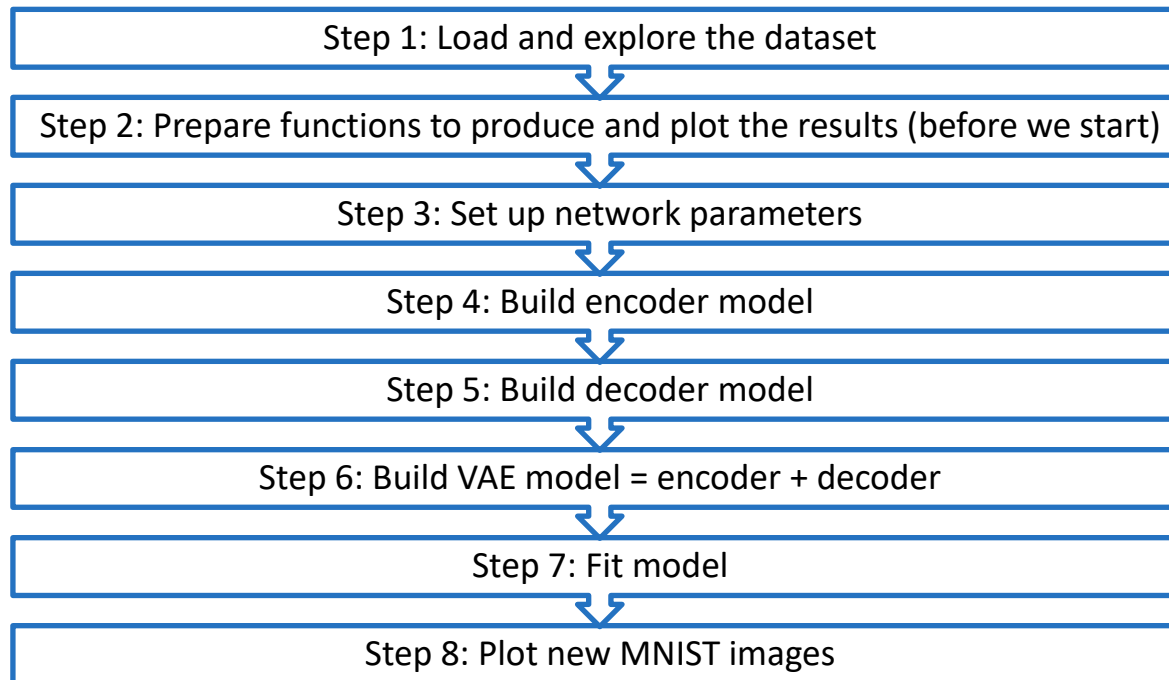
- We'll denote the generator with $G(z, \theta_g)$, where θ_g are the network weights, and z is the latent vector, which serves as an input to the generator. Think of it as a random seed value to kickstart the image-generation process. It is similar to the latent vector in the VAEs. z has a probability distribution, $p_z(z)$, which is usually random normal or random uniform. The generator outputs fake samples, x , with a probability distribution of $p_g(x)$. You can think of $p_g(x)$ as the probability distribution of the real data according to the generator.
- We'll denote the discriminator with $D(x, \theta_d)$, where θ_d are the network weights. It takes as input either the real data with the $x \sim p_{data}(x)$ distribution, or the generated samples, $x \sim p_g(x)$. The discriminator is a binary classifier, which outputs whether the input image is part of the real (network output 1) or the generated data (network output 0).
- During training, we'll denote the discriminator and generator loss functions with $J^{(D)}$ and $J^{(G)}$, respectively.

Let's have fun 😊



Generating new MNIST digits with VAE

- How to use VAE (Variational Auto-Encoder) to generate different MNIST images?
- VAE use convolution layers in encoder and decoder.



Generating new MNIST digits with VAE

Step 1: Load and explore the dataset

```
➤ (x_train, y_train), (x_test, y_test) = mnist.load_data()

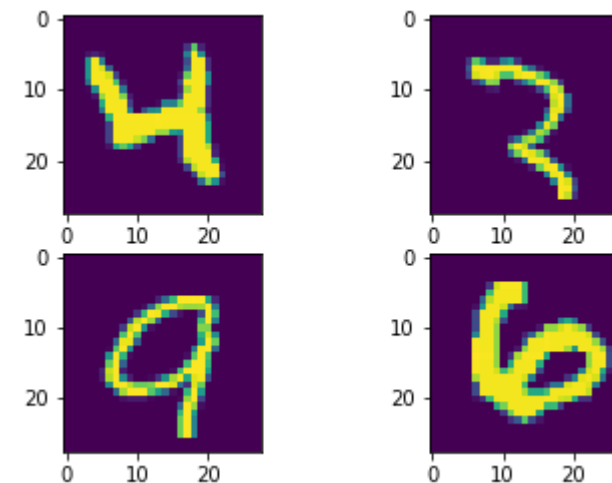
image_size = x_train.shape[1]
# Only get some data to train and test
train_len = 10000
test_len = 1000
x_train = np.reshape(x_train[:train_len], [-1, image_size, image_size, 1])
x_test = np.reshape(x_test[:test_len], [-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_test = y_test[:test_len]
```

```
➤ x_train.shape
```

```
0]: (10000, 28, 28, 1)
```

```
➤ x_test.shape
```

```
1]: (1000, 28, 28, 1)
```



Generating new MNIST digits with VAE

Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to
encoder_input (InputLayer)	[(None, 28, 28, 1)]	0	
conv2d (Conv2D)	(None, 14, 14, 32)	320	encoder_input[0][0]
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18496	conv2d[0][0]
flatten (Flatten)	(None, 3136)	0	conv2d_1[0][0]
dense (Dense)	(None, 16)	50192	flatten[0][0]
z_mean (Dense)	(None, 2)	34	dense[0][0]
z_log_var (Dense)	(None, 2)	34	dense[0][0]
z (Lambda)	(None, 2)	0	z_mean[0][0]



Generating new MNIST digits with VAE

Model: "decoder"

Layer (type)	Output Shape	Param #
=====		
z_sampling (InputLayer)	[(None, 2)]	0
dense_1 (Dense)	(None, 3136)	9408
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose (Conv2DTran	(None, 14, 14, 64)	36928
conv2d_transpose_1 (Conv2DTr	(None, 28, 28, 32)	18464
decoder_output (Conv2DTransp	(None, 28, 28, 1)	289
=====		
Total params: 65,089		
Trainable params: 65,089		
Non-trainable params: 0		



Generating new MNIST digits with VAE

```
outputs = decoder(encoder(inputs)[2])
vae = Model(inputs, outputs, name='vae')
```



```
vae.compile(optimizer='rmsprop')
vae.summary()
```

Model: "vae"

Layer (type)	Output Shape	Param #	Connected to
=====			
encoder_input (InputLayer)	[(None, 28, 28, 1)]	0	
encoder (Functional)	[(None, 2), (None, 2)]	69076	encoder_input[0][0]
decoder (Functional)	(None, 28, 28, 1)	65089	encoder[0][2]
conv2d (Conv2D)	(None, 14, 14, 32)	320	encoder_input[0][0]
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18496	conv2d[0][0]
flatten (Flatten)	(None, 3136)	0	conv2d_1[0][0]
dense (Dense)	(None, 16)	50192	flatten[0][0]
z_log_var (Dense)	(None, 2)	34	dense[0][0]
z_mean (Dense)	(None, 2)	34	dense[0][0]
tf.reshape_1 (TFOpLambda)	(None,)	0	decoder[0][0]
tf.reshape (TFOpLambda)	(None,)	0	encoder_input[0][0]



Generating new MNIST digits with VAE

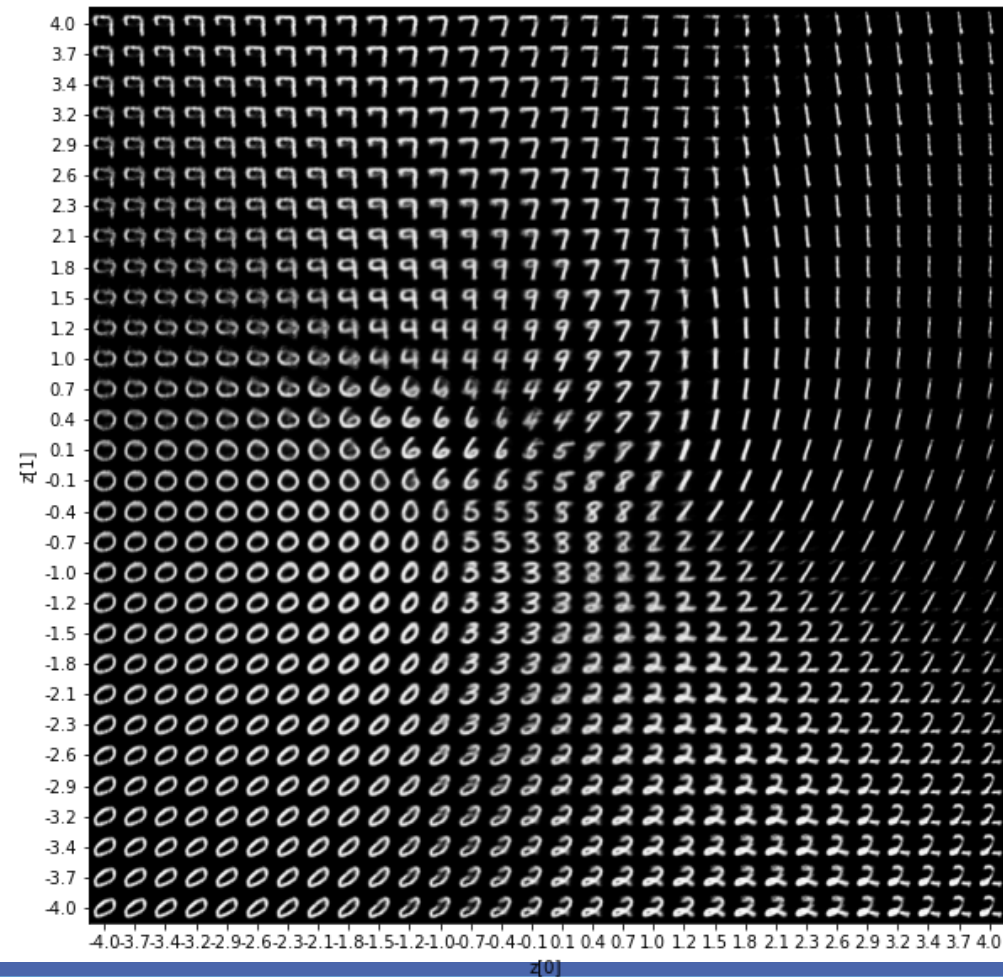
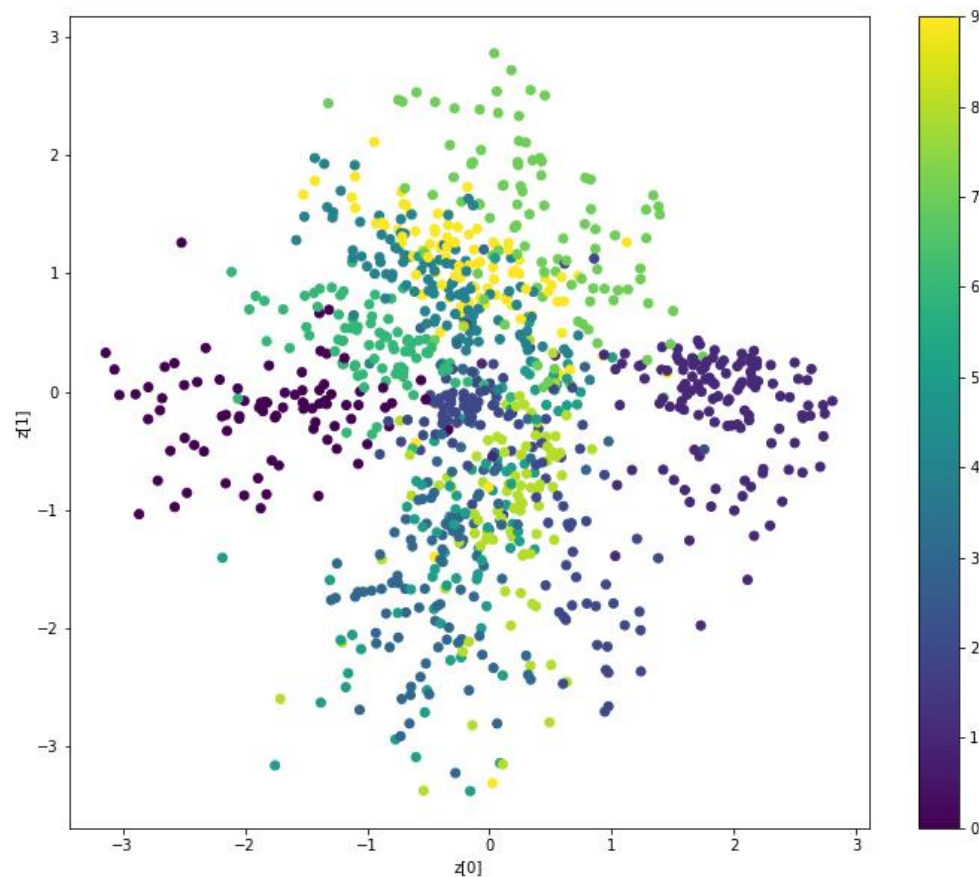
Step 7: Fit model

```
▶ if load_weights:
    vae = vae.load_weights(args.weights)
else:
    # train the autoencoder
    vae.fit(x_train,
           epochs=epochs,
           batch_size=batch_size,
           validation_data=(x_test, None))
    vae.save_weights('vae_cnn_mnist.h5')
```

```
Epoch 1/30
313/313 [=====] - 14s 28ms/step - loss: 64.5804 - val_loss: 45.2958
Epoch 2/30
313/313 [=====] - 7s 22ms/step - loss: 45.1016 - val_loss: 42.8691
Epoch 3/30
313/313 [=====] - 7s 22ms/step - loss: 43.6458 - val_loss: 42.0320
```

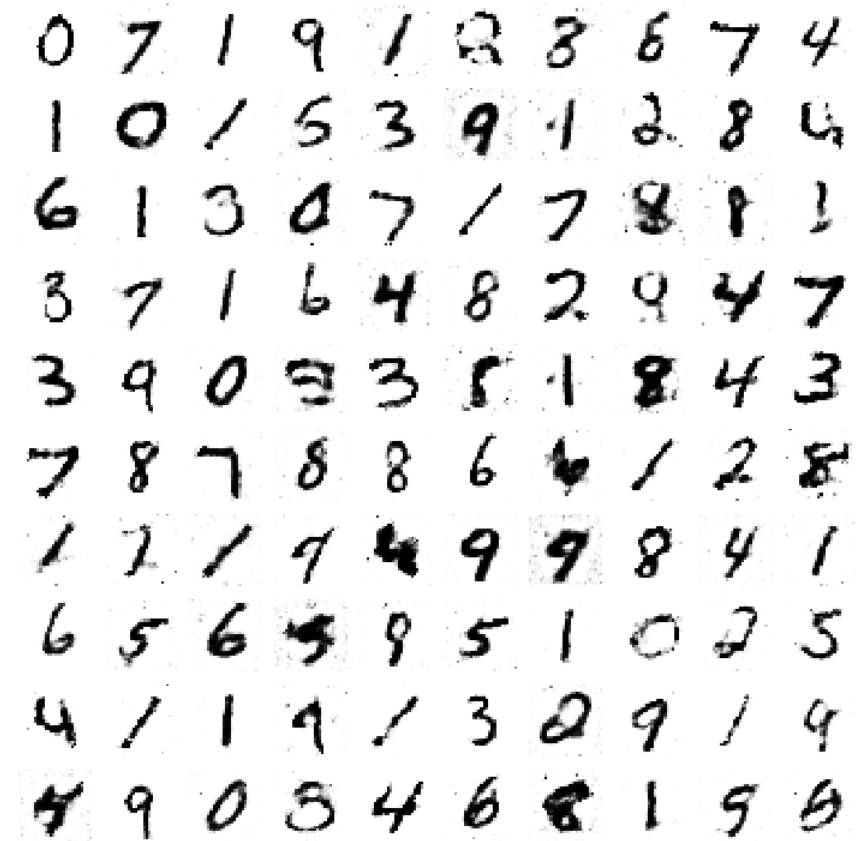
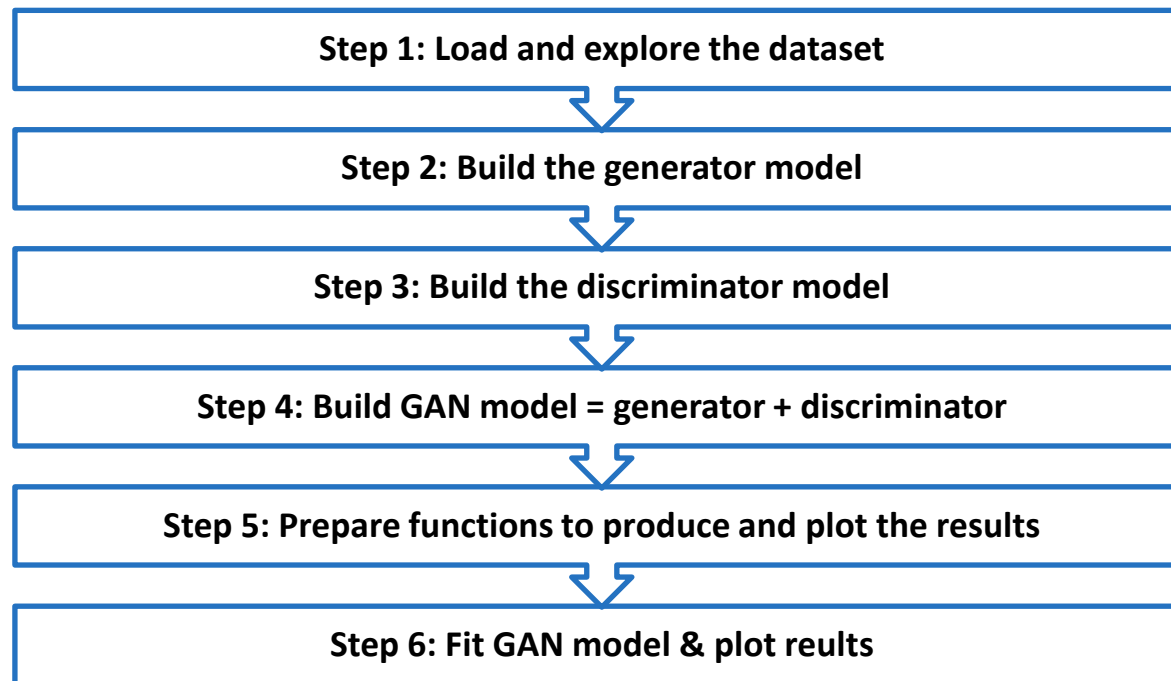
Generating new MNIST digits with VAE

```
plot_results(encoder, decoder, x_test, y_test, batch_size=batch_size)
```



Generating new MNIST images with GANs

- How to use GAN (Generative Adversarial Network) to generate different MNIST images?
- GAN use dense layers in generator and discriminator.





Generating new MNIST images with GANs

Model: "generator"

Layer (type)	Output Shape	Param #
=====		
dense_8 (Dense)	(None, 256)	25856
leaky_re_lu_6 (LeakyReLU)	(None, 256)	0
dense_9 (Dense)	(None, 512)	131584
leaky_re_lu_7 (LeakyReLU)	(None, 512)	0
dense_10 (Dense)	(None, 1024)	525312
leaky_re_lu_8 (LeakyReLU)	(None, 1024)	0
dense_11 (Dense)	(None, 784)	803600
=====		
Total params: 1,486,352		
Trainable params: 1,486,352		
Non-trainable params: 0		



Generating new MNIST images with GANs

Model: "discriminator"

Layer (type)	Output Shape	Param #
=====		
dense_12 (Dense)	(None, 1024)	803840
leaky_re_lu_9 (LeakyReLU)	(None, 1024)	0
dropout_3 (Dropout)	(None, 1024)	0
dense_13 (Dense)	(None, 512)	524800
leaky_re_lu_10 (LeakyReLU)	(None, 512)	0
dropout_4 (Dropout)	(None, 512)	0
dense_14 (Dense)	(None, 256)	131328
leaky_re_lu_11 (LeakyReLU)	(None, 256)	0
dropout_5 (Dropout)	(None, 256)	0
dense_15 (Dense)	(None, 1)	257



Generating new MNIST images with GANs

Step 4: Build GAN model = generator + discriminator

```
# Combined network
discriminator.trainable = False
ganInput = Input(shape=(randomDim,))
x = generator(ganInput)
ganOutput = discriminator(x)
gan = Model(inputs=ganInput, outputs=ganOutput)
gan.compile(loss='binary_crossentropy', optimizer=adam)
gan.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 100)]	0

generator (Sequential)	(None, 784)	1486352

discriminator (Sequential)	(None, 1)	1460225
=====		
Total params: 2,946,577		
Trainable params: 1,486,352		
Non-trainable params: 1,460,225		

Generating new MNIST images with GANs

Step 6: Fit GAN model & plot results

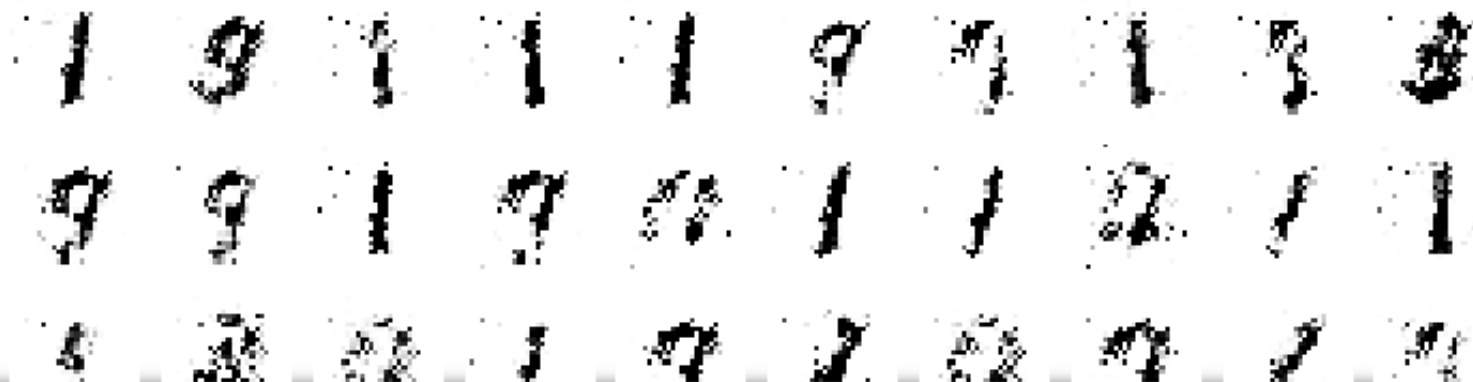
In [27]: `train(100, 128)`

----- Epoch 99 -----

100%|██████████| 468/468 [00:39<00:00, 11.76it/s]
0%| | 2/468 [00:00<00:38, 12.24it/s]

----- Epoch 100 -----

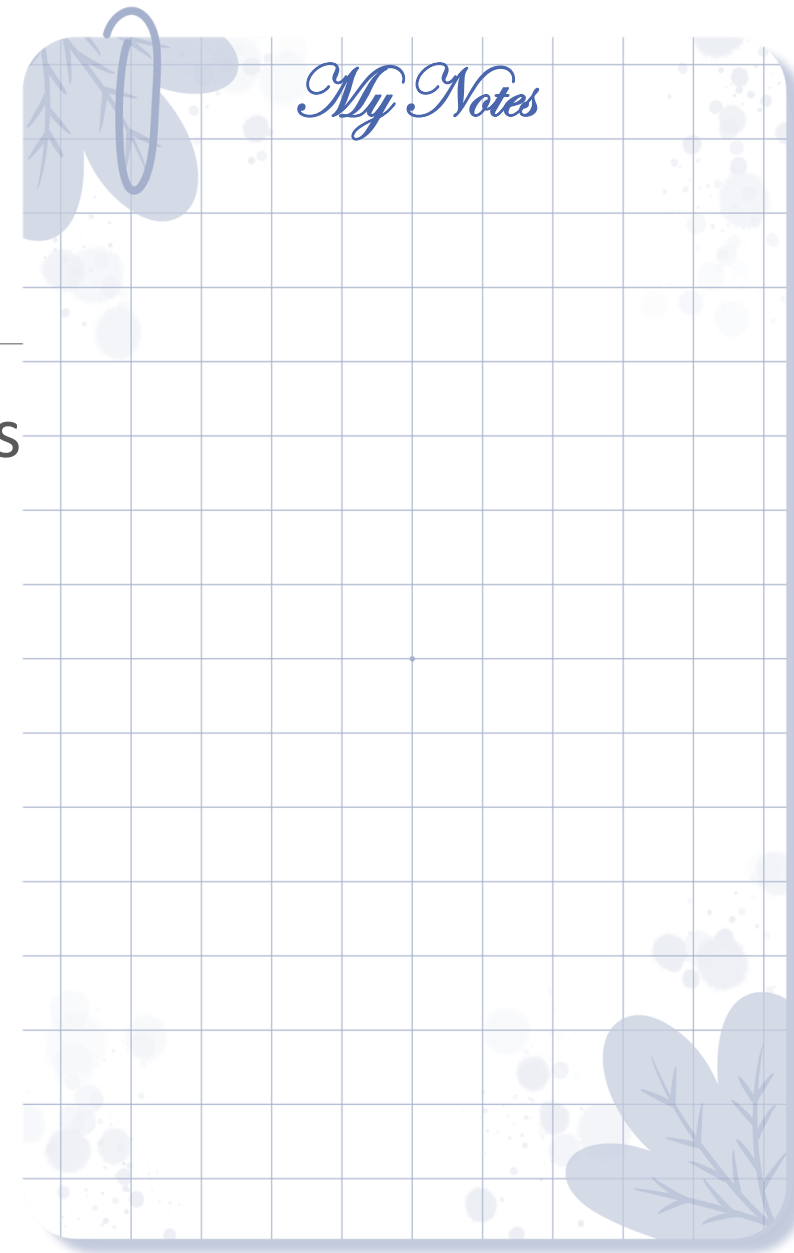
100%|██████████| 468/468 [00:39<00:00, 11.99it/s]






Recap!

- ✓ Intuition and justification of generative models
- ✓ Autoencoders
- ✓ Variational Autoencoders (VAEs)
- ✓ Generative Adversarial networks (GANs)
- ✓ Generating new MNIST digits with VAE
- ✓ Generating new MNIST images with GANs





Success is not final, failure
is not fatal: it is the courage
to continue that counts.

Winston Churchill

quote fancy

@SalhaAlzahrani