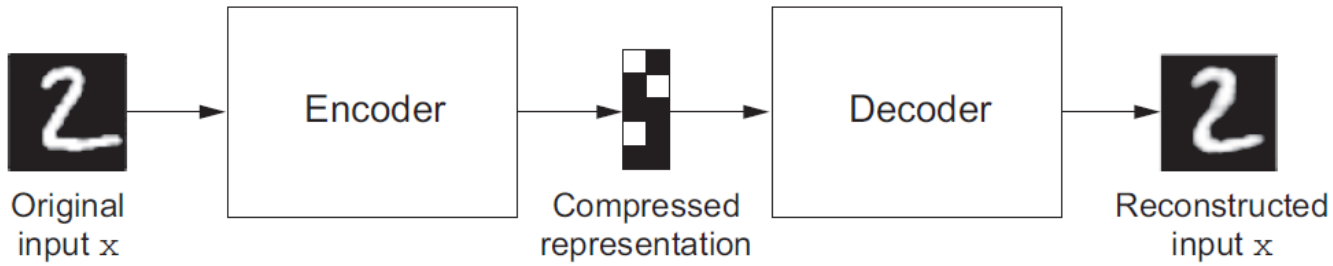
Autoencoders and GANs

Autoencoders

The key idea of image generation is to develop a low-dimensional latent space of representations (which naturally is a vector space) where any point can be mapped to a realistic-looking image. The module capable of realizing this mapping, taking as input a latent point and outputting an image (a grid of pixels), is called a generator (in the case of GANs) or a decoder (in the case of autoencoders). Once such a latent space has been developed, you can sample points from it, either deliberately or at random, and, by mapping them to image space, generate images that have never been seen before.

A classical image autoencoder takes an image, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as the original image, via a decoder module (see figure 8.12). It's then trained by using as target data the same images as the input images, meaning the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more-or-less interesting latent representations of the data. Most commonly, you'll constrain the code to be low-dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.



Simple autoencoder

We'll start with a single fully-connected neural layer as an encoder and as a decoder.

Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from tensorflow.keras.datasets import mnist
import numpy as np

#TODO load MNIST data set
(x_train, _), (x_test, _) =
```

We will normalize all values between 0 and 1 and we will flatten the 28×28 images into vectors of size 784.

```
x_train = x_train.astype('float32') / 255.
```

```
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))

#TODO Prepare the test samples as above
x_test =
x_test =
```

It is time to prepare the model itself. However, before we will do it let's notice that we won't use common models.Sequential() at the beginning. All neural networks introduced before have been implemented using the Sequential model, which makes the assumption that the network has exactly one input and exactly one output, and that it consists of a linear stack of layers. When this is not the case, we can use **Keras functional API** to better personalize it to our needs. In the functional API, you directly manipulate tensors, and you use layers as functions that take tensors and return tensors. Every layer is constructed like this:

output_tensor = Layer(parameters)(input_tensor)

By correct use of input and output variable names, we can connect them to one another. When we chain all layers together we use the *Model* function (with input and output variable names of the whole network as parameters) to create a complete model.

model_name = ***Model****(network_input_name, network_output_name)*

Our first autoencoder can be a good example of this:

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the
input is 784 floats

# this is our input placeholder
input_img = Input(shape=(784,))

# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)

# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
```

Now, let's prepare separate models for encoder and decoder:

```
# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)

# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
```

```
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]

# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Now let's train our autoencoder to reconstruct MNIST digits.

```
#TODO configure the model to use a binary crossentropy loss and the Adam
optimizer:
autoencoder.compile(#TODO)

#TODO complete fit(). As a starting point set epoch to 50, batch size to
256, shuffle data, and use x_test as a validation set.
#Later You can experiment with those values
autoencoder.fit(x_train, x_train, #TODO)

# encode and decode some digits
# note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

We can try to visualize the reconstructed inputs and the encoded representations. We will use Matplotlib.

```
import matplotlib.pyplot as plt

n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(1, n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.

Convolutional autoencoder

Since our inputs are images, it makes sense to use convolutional neural networks (convnets) as encoders and decoders. In practical settings, autoencoders applied to images are always convolutional autoencoders –they simply perform much better.

To train it, we will again use the original MNIST digits but with shape (samples, 28, 28, 1), and we will just normalize pixel values between 0 and 1.

```
from tensorflow.keras.datasets import mnist
import numpy as np

#TODO load MNIST data set and convert it to [0,1] range like before.
(x_train, _), (x_test, _) =

x_train =
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))

#TODO Prepare the test samples as above
x_test =
x_test =
```

Now, let's prepare the model. The encoder will consist of a stack of Conv2D and MaxPooling2D layers (max-pooling being used for spatial down-sampling), while the decoder will consist of a stack of Conv2D and UpSampling2D layers.

```
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,
UpSampling2D
from tensorflow.keras.models import Model
from tensorflow.keras import backend as K

input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first`
image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
```

```
autoencoder = Model(input_img, decoded)
#TODO configure model to use a binary crossentropy loss, and the Adam
optimizer:

autoencoder.compile(#TODO)
#TODO complete fit(). Set epoch to 10, batch size to 128, shuffle data, and
use x_test as a validation set.
#You can experiment with those values. If training takes a lot of time, try
to reduce the number of epochs (but monitor loss value).

autoencoder.fit(x_train, x_train, #TODO)

decoded_imgs = autoencoder.predict(x_test)
```

Let's take a look at the reconstructed digits:

```
#TODO Visualise ten digits like in the first exercise.
```

More examples of how to use autoenkoders can be found in this blog.

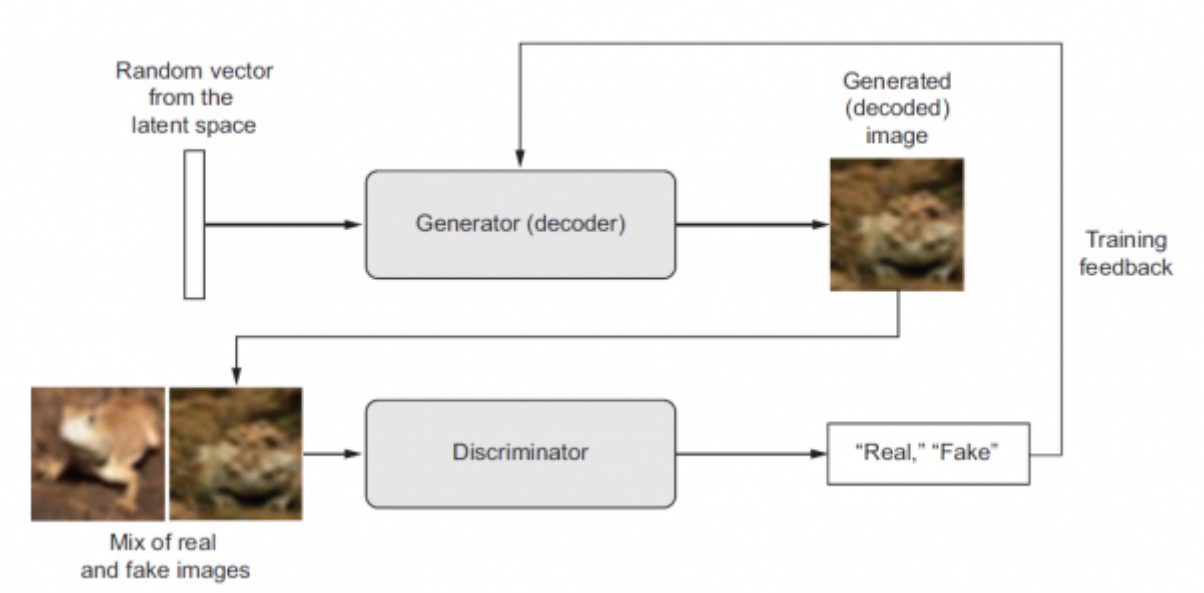
GAN - generative adversarial networks

An intuitive way to understand GANs is to imagine a forger trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos and shows them all to an art dealer. The art dealer makes an authenticity assessment for each painting and gives the forger feedback about what makes a Picasso look like a Picasso. The forger goes back to his studio to prepare some new fakes. As time goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes. In the end, they have on their hands some excellent fake Picassos.

That’s what a GAN is: a forger network and an expert network, each being trained to best the other. As such, a GAN is made of two parts:

- Generator network — Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
- Discriminator network (or adversary) — Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.

The generator network is trained to be able to fool the discriminator network, and thus it evolves toward generating increasingly realistic images as training goes on: artificial images that look indistinguishable from real ones, to the extent that it’s impossible for the discriminator network to tell the two apart. Meanwhile, the discriminator is constantly adapting to the gradually improving capabilities of the generator, setting a high bar of realism for the generated images. Once training is over, the generator is capable of turning any point in its input space into a believable image.



GAN models are quite complex to implement and require more time and tuning to train them correctly. We won't code it during this lab. You can, however, try to run the ready script using Google Colab. In the webpage linked below, You will find a working code that You can analyze, edit, or even run by clicking the *play* button in each section. Training should be fast, as it uses Google's cloud GPUs. The only requirement is having a Google account.

GAN on Google Colab

Sources

Stanford Lecture in PDF

Stanford Lecture on YouTube

Book by F. Chollet: Deep learning with Python

Tutorial by F. Chollet