Machine Learning workshop

Prof. Daniele Bonacorsi

ISGC 2021

March 22, 2021

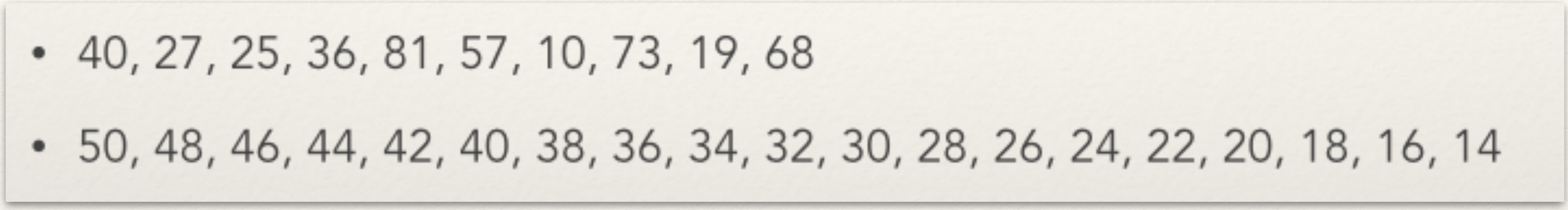
Lab on **Autoencoders** (AE)

Efficient data representation [1/3]

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

Efficient data representation [2/3]



- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance:

- it would seem that the first sequence should be easier, since it is much shorter

However, if you look carefully at the second sequence..

- you will notice that *it is just the list of even numbers from 50 down to 14.*
- Once you notice a **pattern**, the sequence becomes much easier to memorise as you actually do NOT memorise the sequence itself, you just memorise the protocol and the starting/ending numbers (to create any former one from the latter ones)

Why is this important?

Efficient data representation [3/3]

The latter is important because what we did would not be strictly necessary

- if you could quickly and easily memorise very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that.
- The fact that it is hard to memorise long sequences is what makes it **useful to recognise patterns**

This is why we can make a very good use of a NN architecture that I can be **constrained during training**: it pushes it to **discover and exploit patterns in the data**.

Another example: pattern matching in chess

The relationship between memory, perception, and pattern matching was studied in the early 1970's *[Ref-ChaseSimon]*

- observation that expert chess players were able to *memorise the positions of all the pieces in a game by looking at the board for just five seconds*, a task that most people would find impossible
- However, this was the case when the pieces were *placed only in realistic positions (from actual games)*, not when the pieces were placed randomly.
- Ultimately, chess experts don't have a much better memory than others; *they just see (chess) patterns more easily*, thanks to their experience with the game.

Noticing patterns helps them to store information efficiently

The "logic" of the NN architecture we are discussing next is not far from the core of this memory experiment..

Autoencoders (AEs)

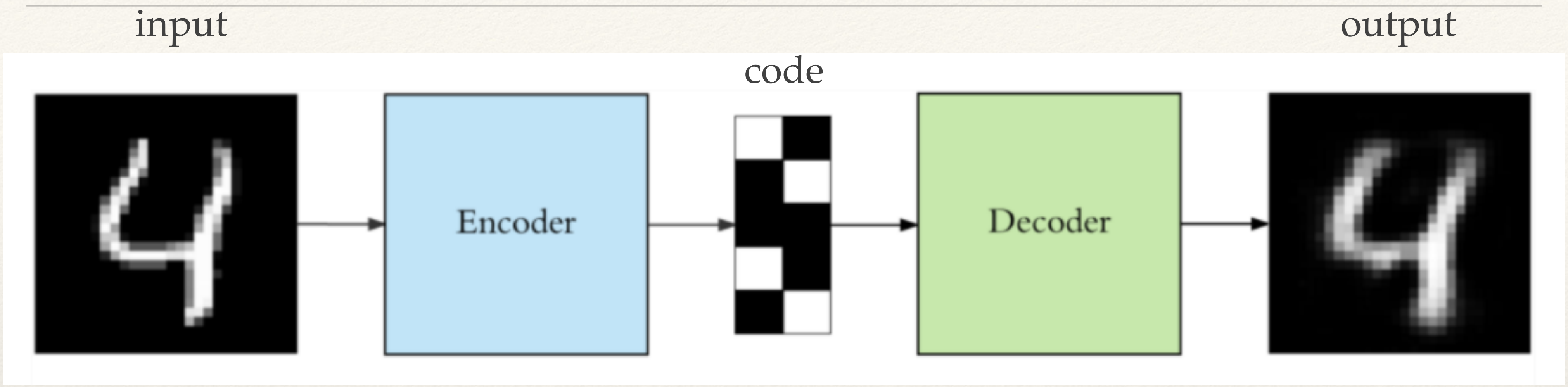
A feed-forward NN with the output that is the same as the input

- said in this way, it sounds useless, right?

The AE:

- compresses the input into a lower-dimensional representation (aka "**latent-space representation**" or "code")
    - The "code" can be thought of a compact "summary", a "compression" of the input
- reconstruct the output from such representation

AE basic concepts



An AE consists of 3 components:

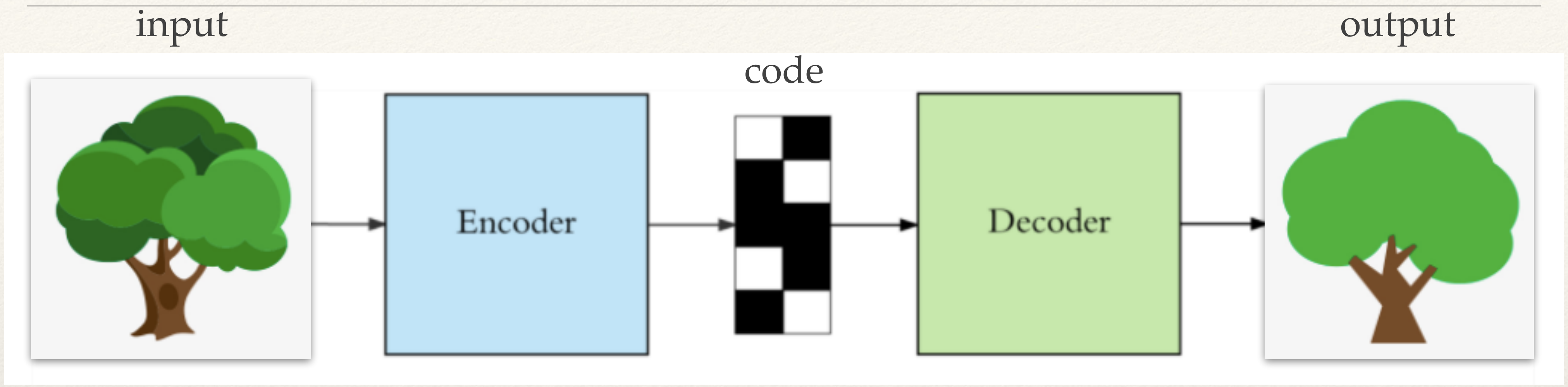
- **encoder**, **code**, **decoder**

Note that encoder and decoder do not talk to each other

- the encoder compresses the input and produces the code
- the decoder uses the code and "reconstructs the input", yielding the output

So, to build an AE we need 1) an encoding method, 2) a decoding method, 3) a loss function to compare the output with the target

AE basic concepts



An AE consists of 3 components:

- **encoder**, **code**, **decoder**

Note that encoder and decoder do not talk to each other

- the encoder compresses the input and produces the code
- the decoder uses the code and "reconstructs the input", yielding the output

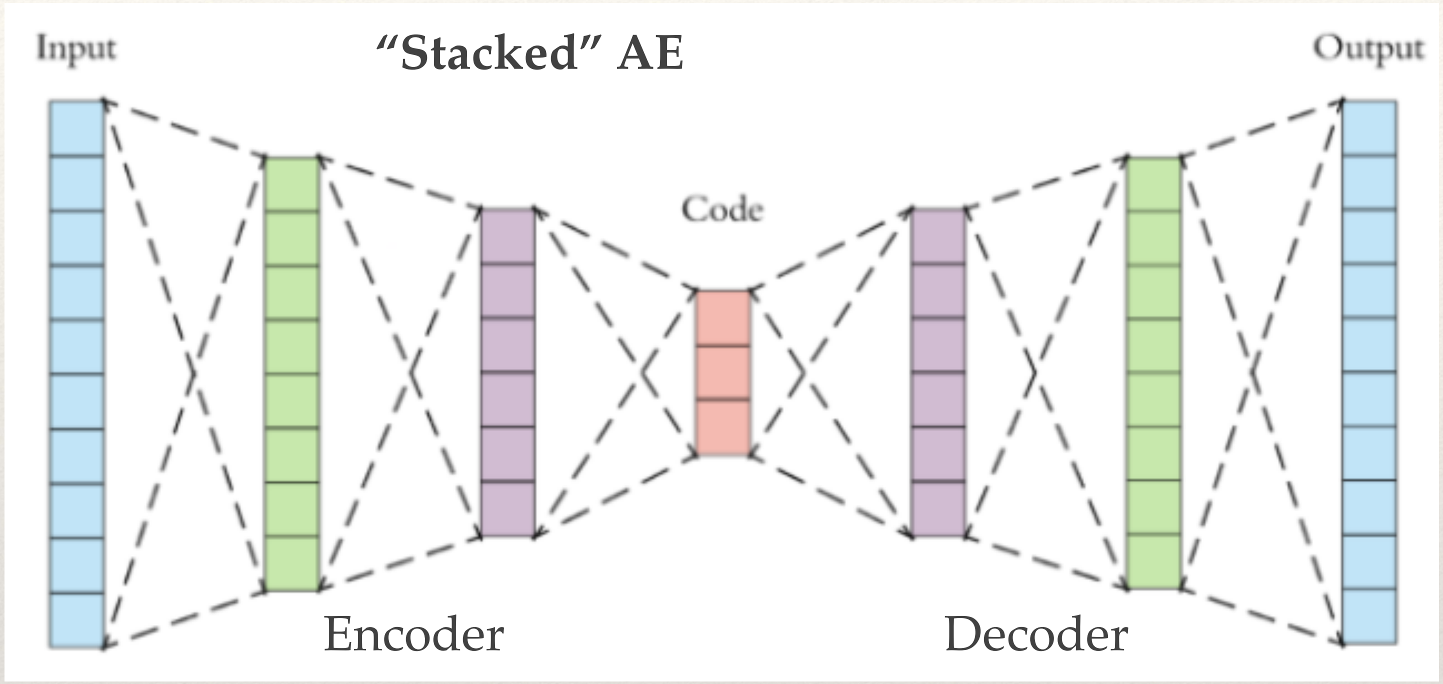
So, to build an AE we need 1) an encoding method, 2) a decoding method, 3) a loss function to compare the output with the target

"AE is not gzip"

AEs can be thought as **dimensionality reduction** (compression) algorithms, but with some peculiar properties

- *data-specific*: w.r.t. standard data compression algorithms, AEs learn features specific for the given training data, so they are only able to meaningfully compress data similar to what they have been trained on
- lossy: the AE's output will be a close but somehow degraded representation of the input
- {un-,self-}supervised: we can throw data to an AE and train it w/o any prior action. No need for labels → un-supervised, but actually imprecise: self-supervised, because they generate their own labels from the training dataset

AE architecture



<u>Requirement</u>: same dimensionality of input and output

<u>**Not** a requirement:</u> encoder/decoder architectures are mirrored

Both **encoder** and **decoder** are FC FF ANN. **Code** is a single layer of an ANN with the dimensionality of our choice

- code size (# nodes in code layer) is a hyperparameter we set before training

AE hyperparameters

Not so many:

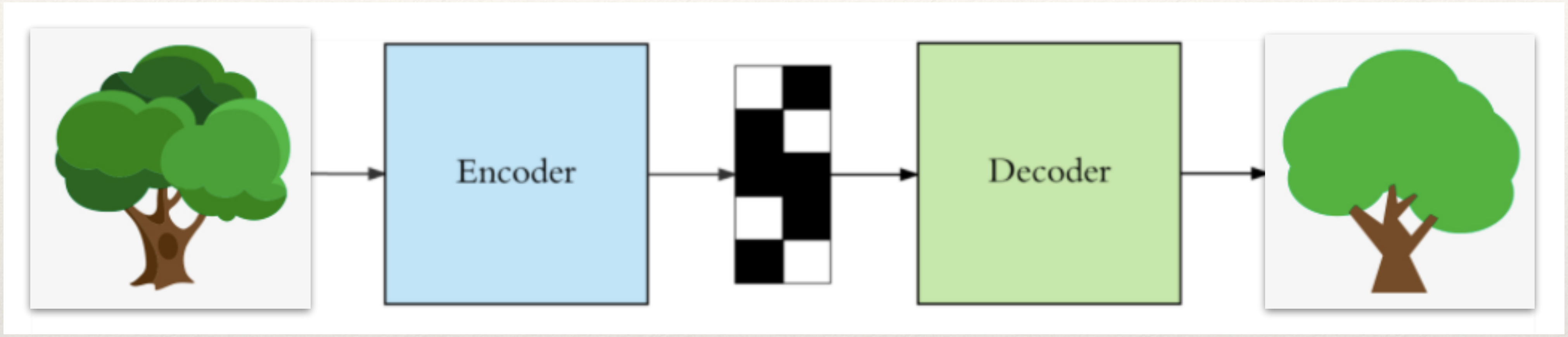
- **Code size**: # of nodes in the code layer
    - Smaller size → more compression
- **# of layers**: AE can be as deep as we like/need
- **# nodes per layer**: Usually, stacked-AE are like a sandwich, i.e. # nodes decreases (increases) with each subsequent layer of the encoder (decoder)
- **loss function**: MSE or binary cross-entropy can be used (typically, the latter if input values are in the range [0, 1])

AEs are trained the same way as ANNs

- via back-propagation and GD

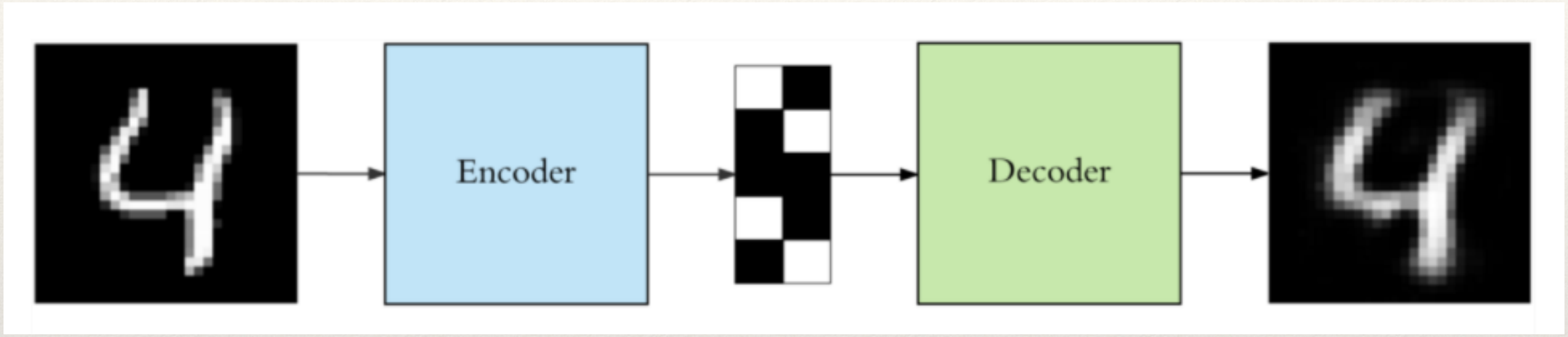
Today: AE implementation in the code

Which input image can we choose for a demo?



Today: AE implementation in the code

Which input image can we choose for a demo?



Let's implement a **very simple AE** to perform **MNIST classification**

- for educational purposes: same example we previously approached:
  1. with basic ML models, and…
  2. … with a CNN

Time to code!

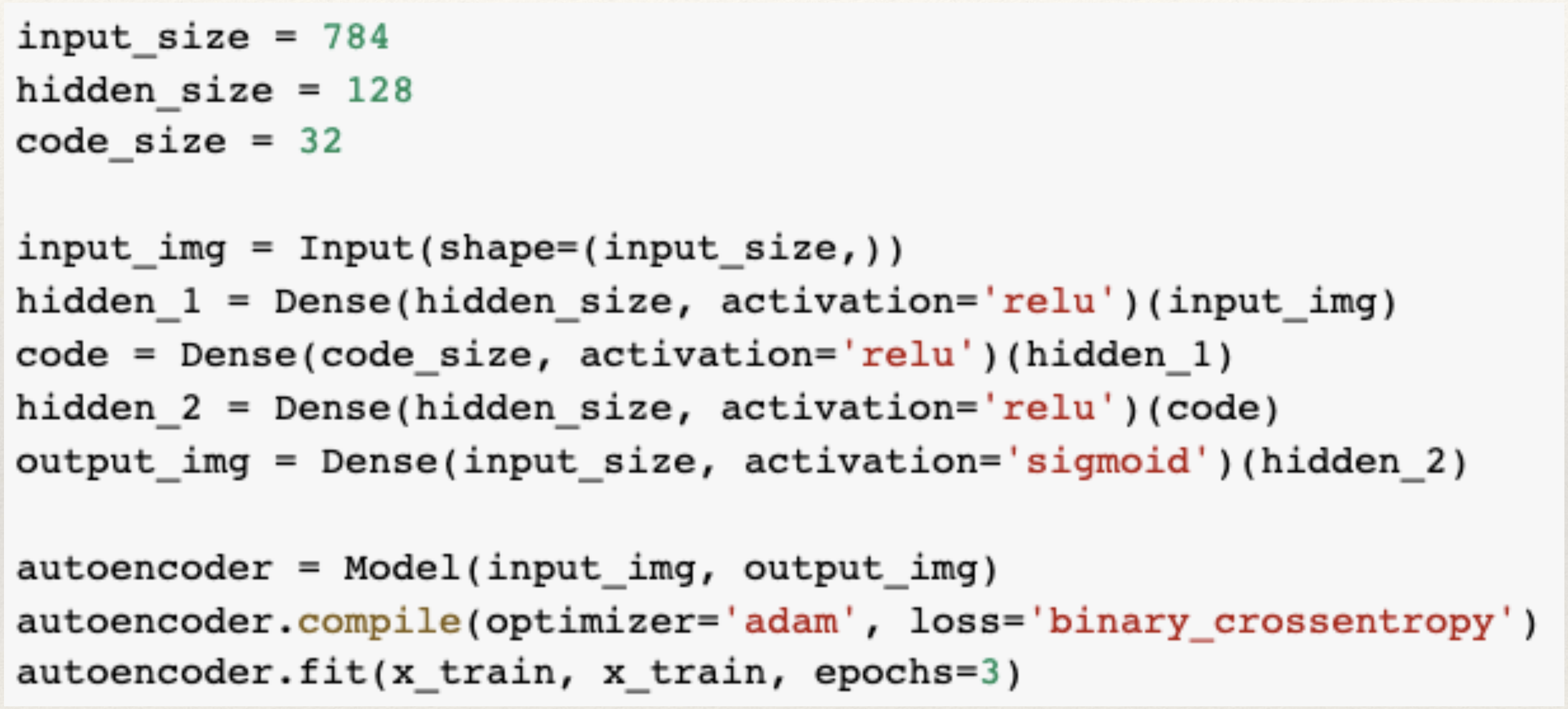
http://bit.ly/Bertinoro1

You can follow these slides, and practice the code later, or you can play on Colab in real time!

*If you are trying all this real-time during the workshop, this means now you should go to **colab** and follow from there*

Time to code!

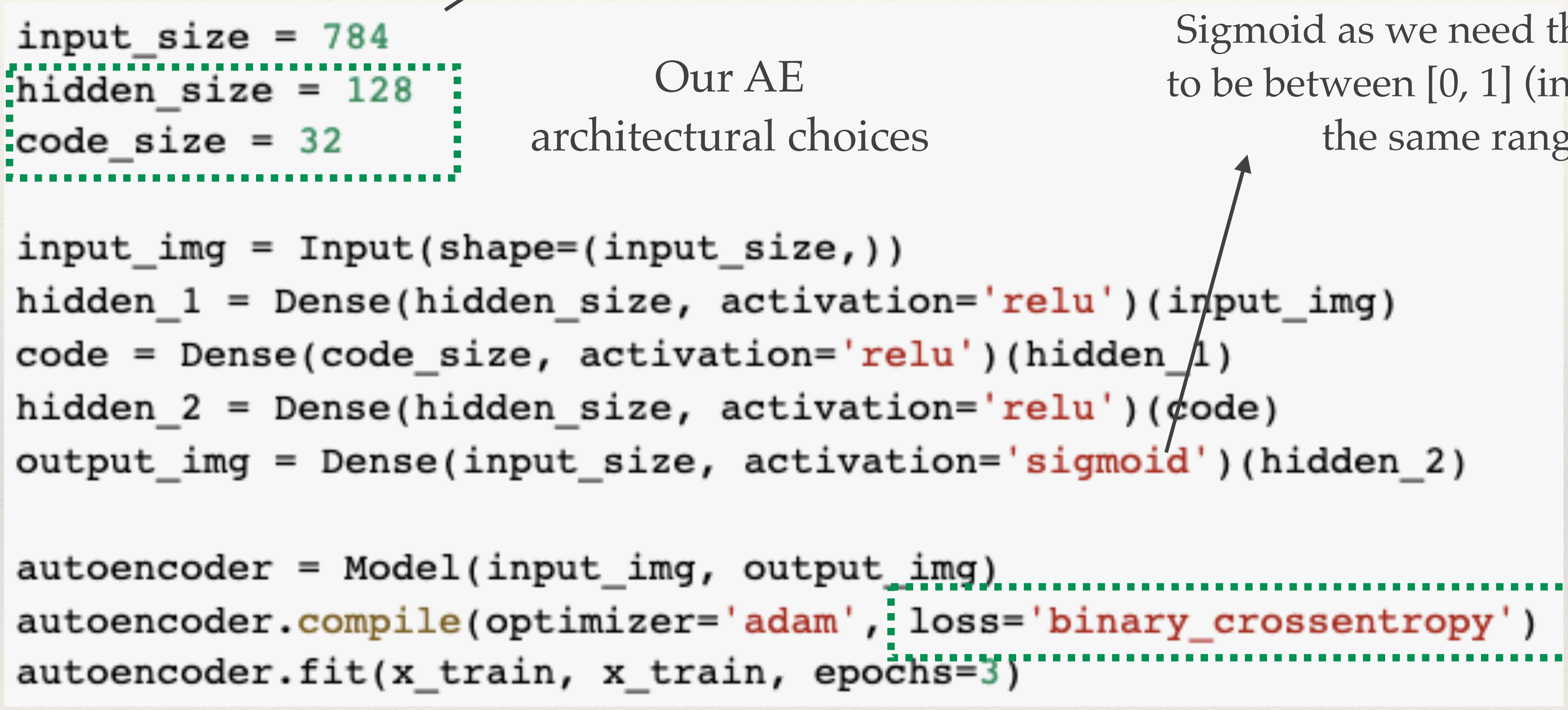
AE implementation: an example



```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

AE implementation: an example

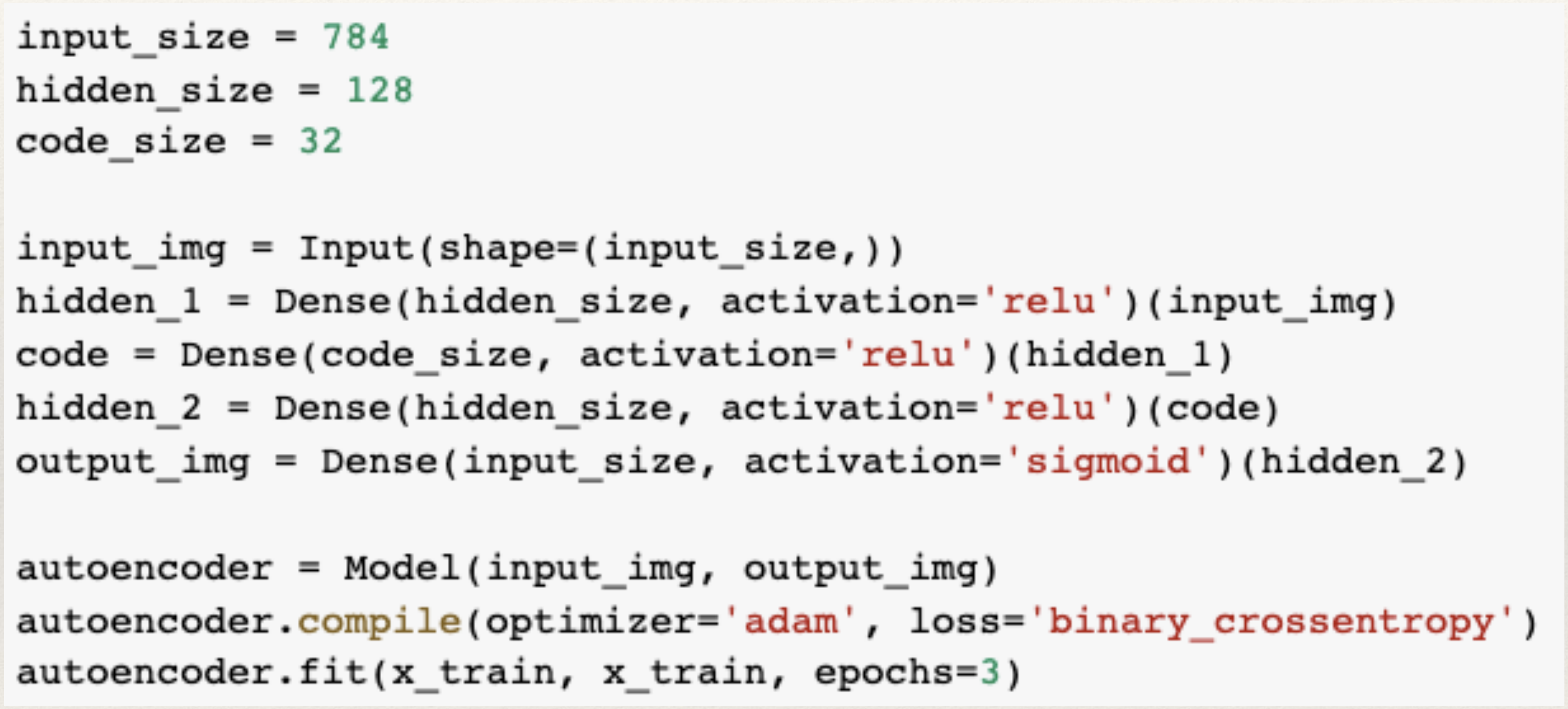
MNIST input is 28x28 images, vectorised into 784 bits long arrays



```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

Our AE architectural choices

Sigmoid as we need the outputs to be between [0, 1] (input also in the same range)

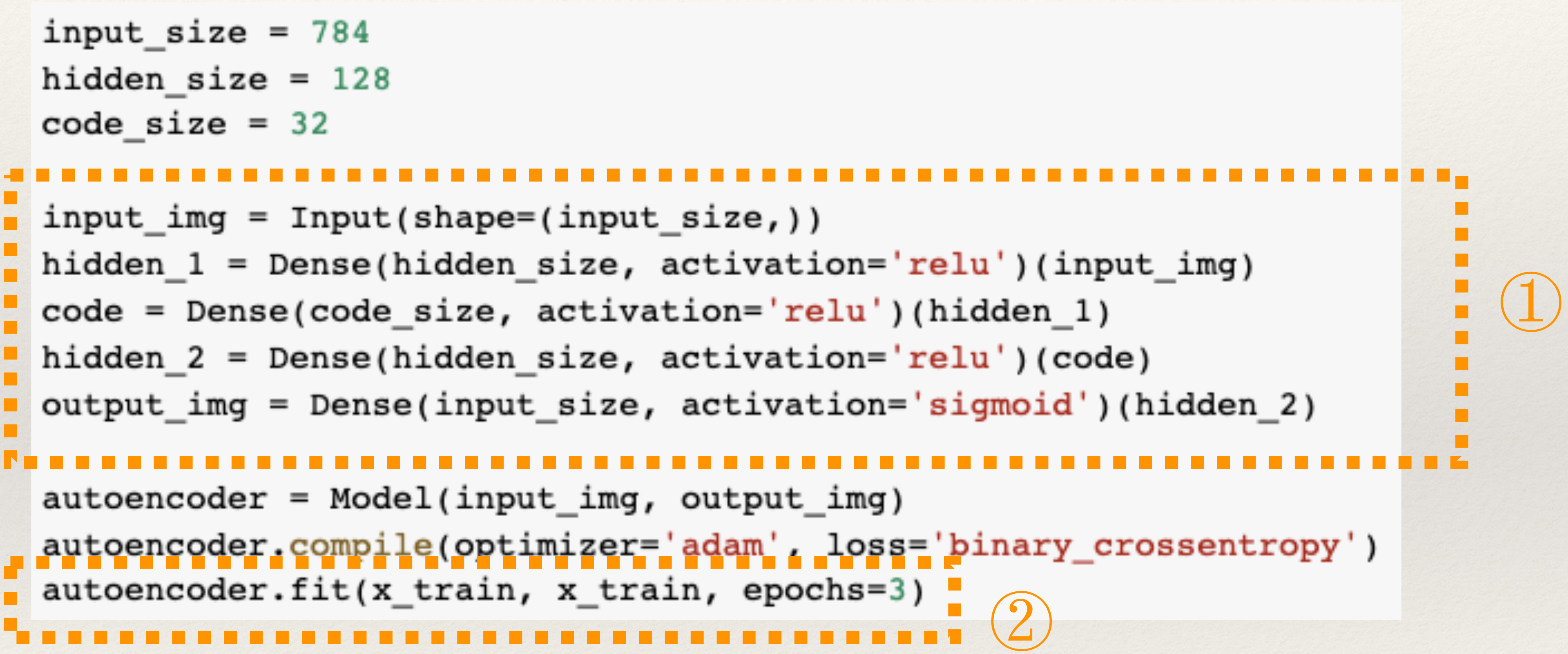


```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

Do you notice anything different w.r.t Keras as we used it before?

```python
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

Do you notice anything different w.r.t Keras as we used it before?



```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

Do you notice anything different w.r.t Keras as we used it before?

Keras **Functional API**

The difference is that we used the Keras **Functional API** ①

- https://keras.io/guides/functional_api/

It is a way to create models in a more flexible way:

- it can handle models with non-linear topology, models with shared layers, models with multiple inputs or outputs, ..

The main idea is that a DL model is usually a directed acyclic graph (DAG) of layers, so the Keras Functional API is a way to build *graphs of layers*.

Keras **Sequential** API

```
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
```

Keras **Functional** API

```
layer_1 = Dense(16, activation='relu')(input)
layer_2 = Dense(8, activation='relu')(layer_1)
```

More verbose but more flexible (better for complex models). The output of *Dense* method is a callable layer, i.e. I can easily grab parts of the model, and flexibly work with those onwards.

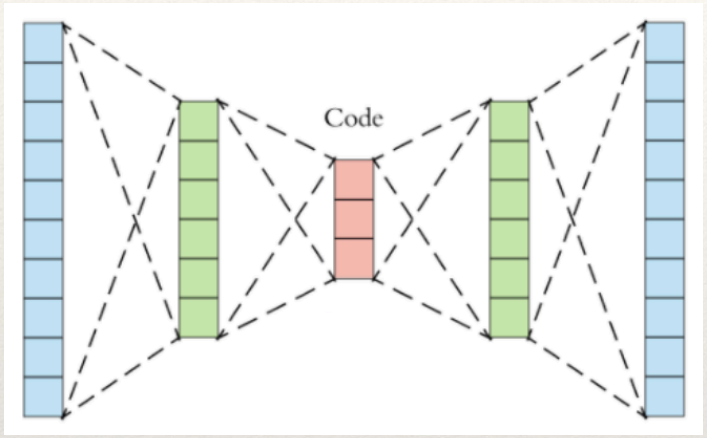
Model fit

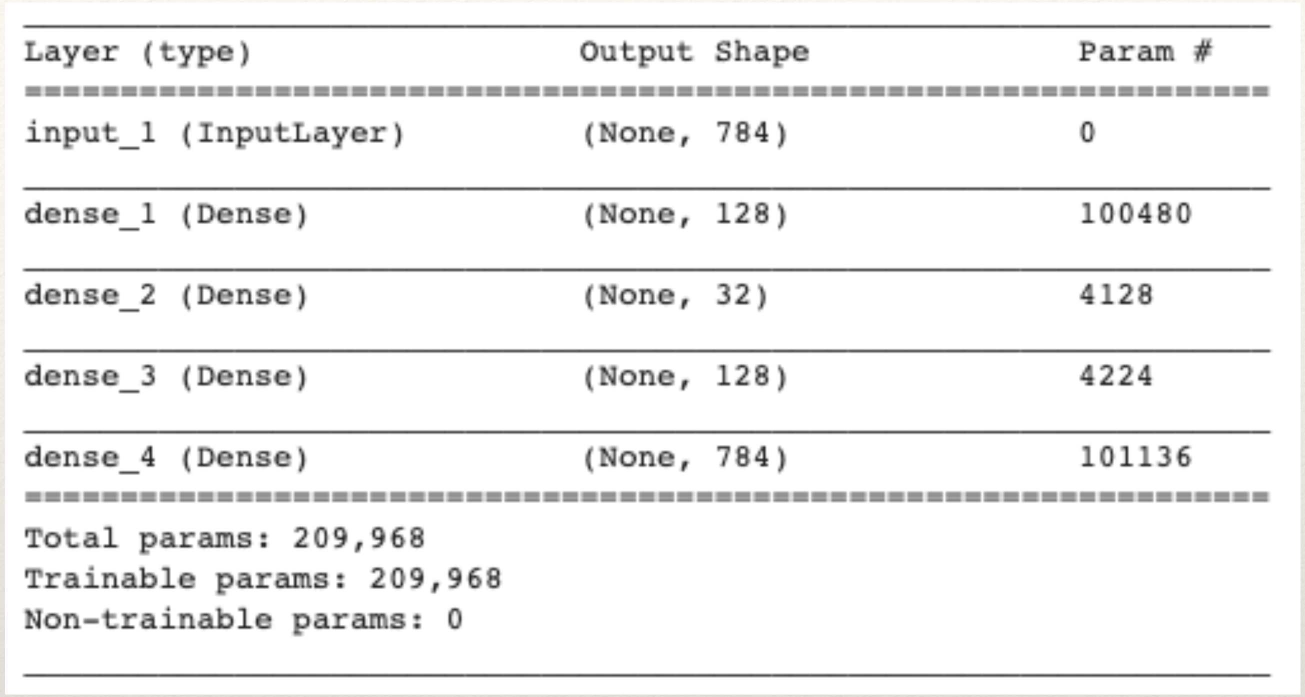
②

```
autoencoder.fit(x_train, x_train, epochs=3)
```

{un-,self-}supervised: the targets of the AE are the same as the input. That's why we supply the training data as the target

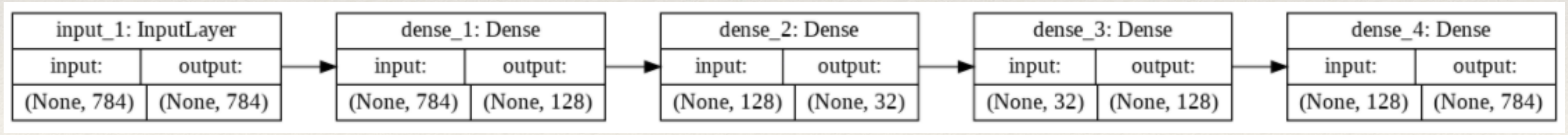
Model summary and viz





```
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 784)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 128)               100480    
_________________________________________________________________
dense_2 (Dense)              (None, 32)                4128      
_________________________________________________________________
dense_3 (Dense)              (None, 128)               4224      
_________________________________________________________________
dense_4 (Dense)              (None, 784)               101136    
=================================================================
Total params: 209,968
Trainable params: 209,968
Non-trainable params: 0
_________________________________________________________________
```

summary()

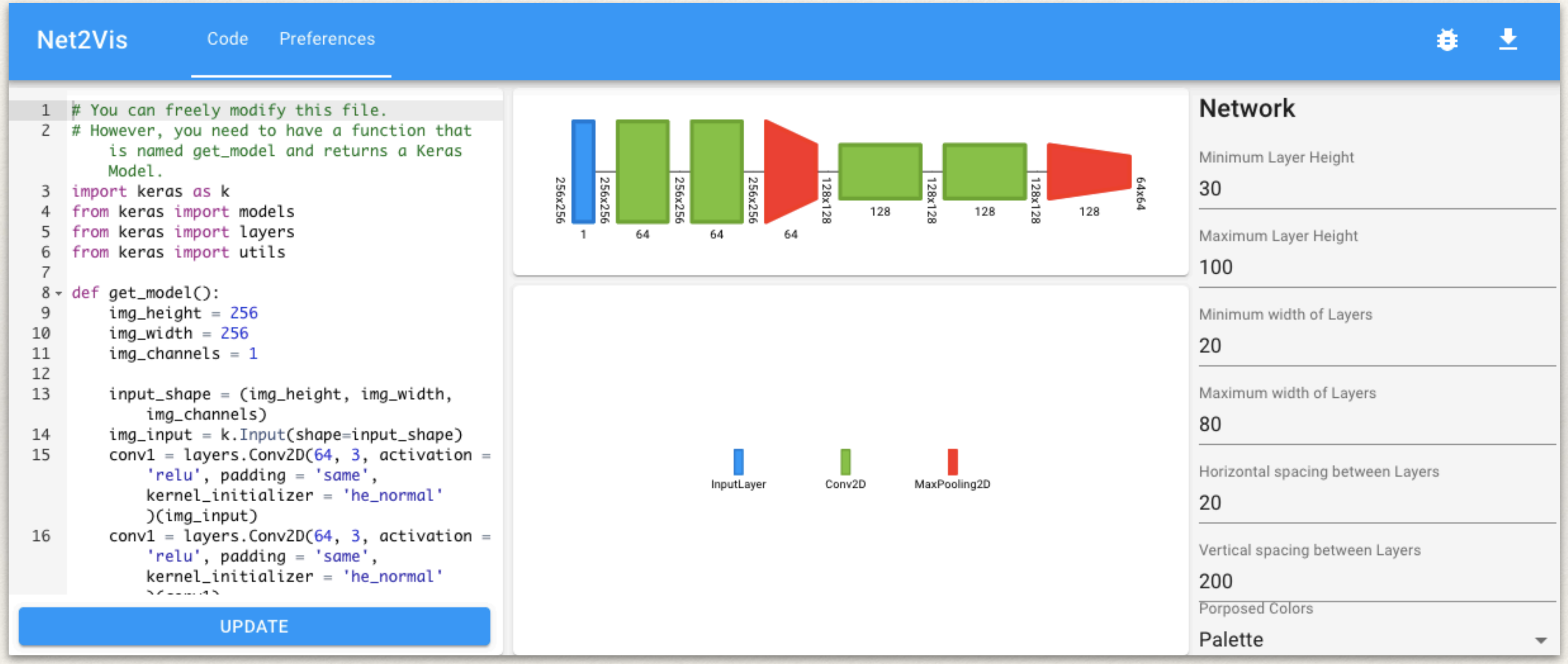


plot_model (in keras.utils)

BTW.. about visualising ANN

Bonus feature!

Net2Vis: https://viscom.net2vis.uni-ulm.de



```
# You can freely modify this file.
# However, you need to have a function that
    is named get_model and returns a Keras
    Model.
import keras as k
from keras import models
from keras import layers
from keras import utils

def get_model():
    img_height = 256
    img_width = 256
    img_channels = 1

    input_shape = (img_height, img_width,
        img_channels)
    img_input = k.Input(shape=input_shape)
    conv1 = layers.Conv2D(64, 3, activation =
        'relu', padding = 'same',
        kernel_initializer = 'he_normal'
        )(img_input)
    conv1 = layers.Conv2D(64, 3, activation =
        'relu', padding = 'same',
        kernel_initializer = 'he_normal'
```

Network

Minimum Layer Height: 30

Maximum Layer Height: 100

Minimum width of Layers: 20

Maximum width of Layers: 80

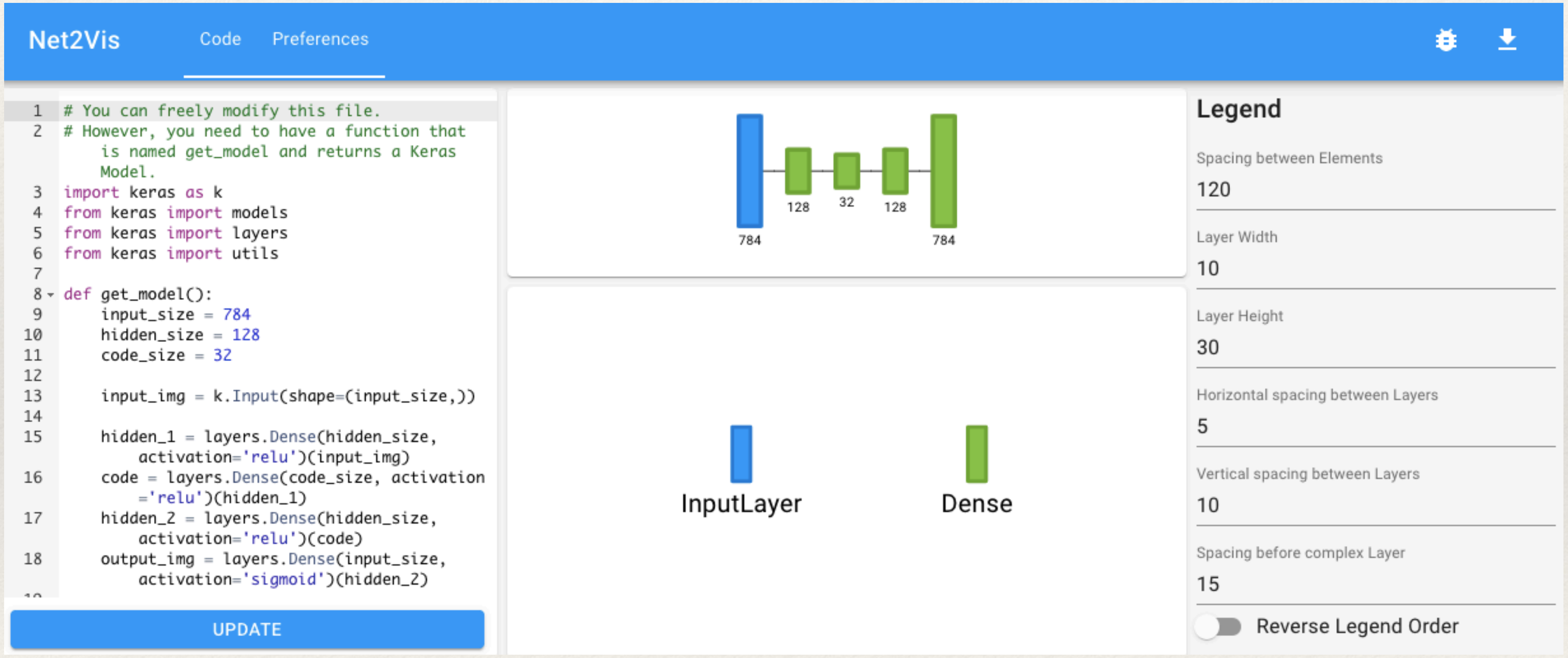
Horizontal spacing between Layers: 20

Vertical spacing between Layers: 200

Porposed Colors: Palette

Bonus feature!

Our AE with Net2Vis



```python
# You can freely modify this file.
# However, you need to have a function that
    is named get_model and returns a Keras
    Model.
import keras as k
from keras import models
from keras import layers
from keras import utils

def get_model():
    input_size = 784
    hidden_size = 128
    code_size = 32

    input_img = k.Input(shape=(input_size,))

    hidden_1 = layers.Dense(hidden_size,
        activation='relu')(input_img)
    code = layers.Dense(code_size, activation
        ='relu')(hidden_1)
    hidden_2 = layers.Dense(hidden_size,
        activation='relu')(code)
    output_img = layers.Dense(input_size,
        activation='sigmoid')(hidden_2)
```

Legend

Spacing between Elements: 120

Layer Width: 10

Layer Height: 30

Horizontal spacing between Layers: 5

Vertical spacing between Layers: 10

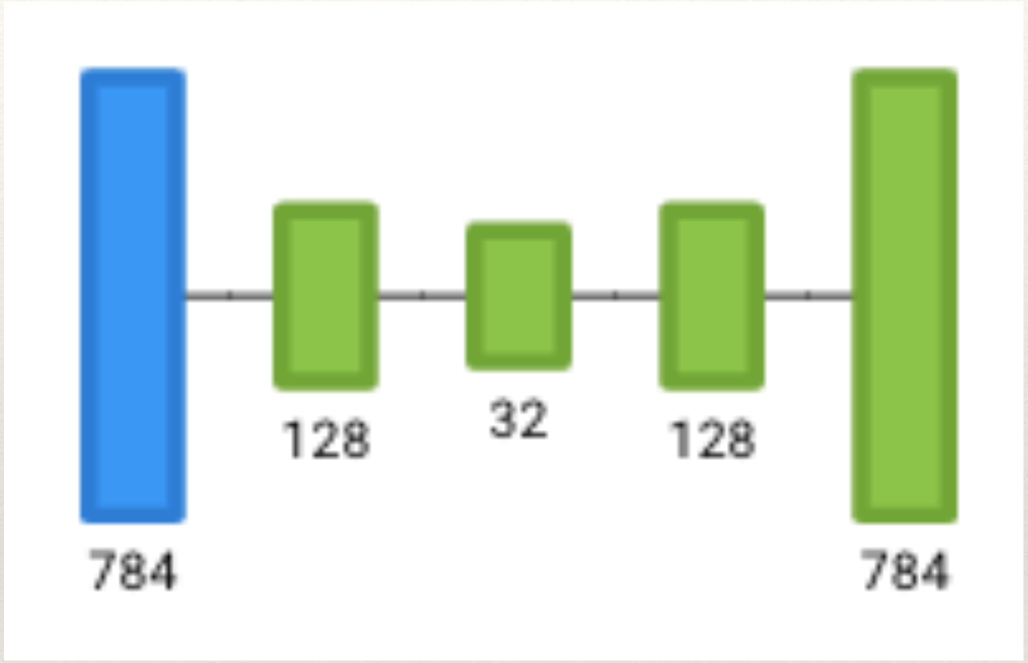
Spacing before complex Layer: 15

Reverse Legend Order

Code to cut&paste for Net2Vis

Bonus feature!

```
# You can freely modify this file.
# However, you need to have a function that is
named get_model and returns a Keras Model.
import keras as k
from keras import models
from keras import layers
from keras import utils

def get_model():
    input_size = 784
    hidden_size = 128
    code_size = 32

    input_img = k.Input(shape=(input_size,))

    hidden_1 = layers.Dense(hidden_size,
activation='relu')(input_img)
    code = layers.Dense(code_size,
activation='relu')(hidden_1)
    hidden_2 = layers.Dense(hidden_size,
activation='relu')(code)
    output_img = layers.Dense(input_size,
activation='sigmoid')(hidden_2)

    model = models.Model(input_img,
output_img)

    return model
```

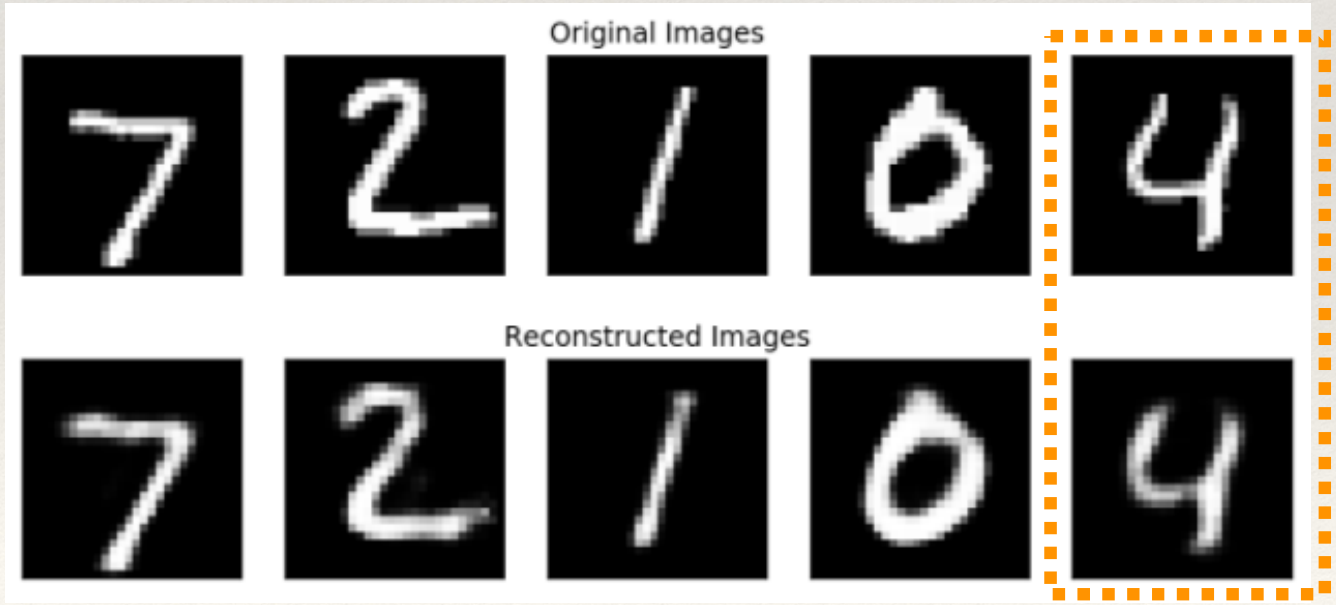


Predict

We can run the AE on the test set simply by using the Keras predict function of Keras.

```
plot_autoencoder_outputs(autoencoder, 5, (28, 28))    # inside it does autoencoder.predict(x_test)
```

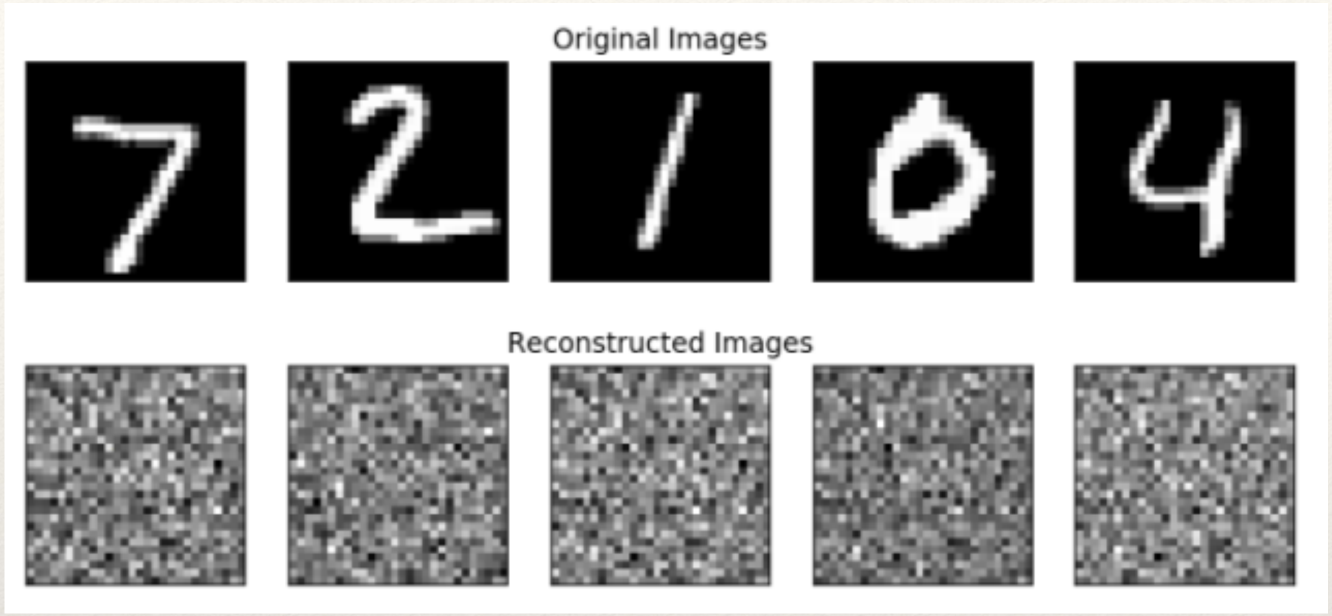
For every image in the test set, we check the AE output. We expect the output to be very similar to the input.



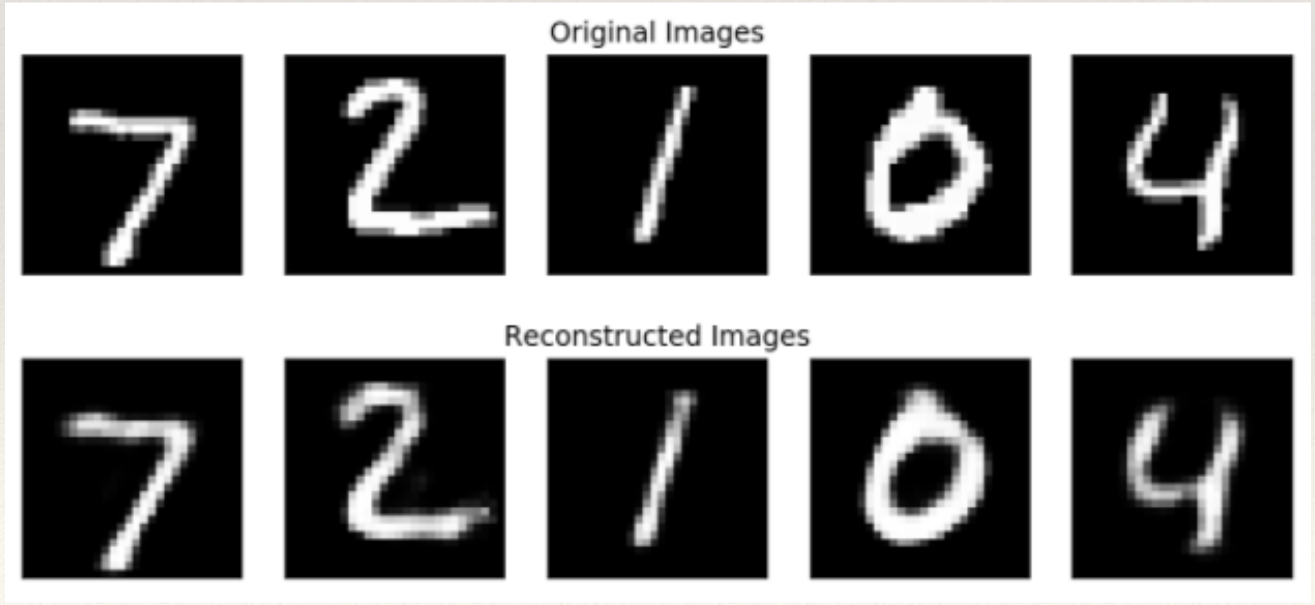
Not perfect, but not bad!
(given the simplicity of the AE architecture)

Note the the handwritten "4" which could be mistakenly taken as a "9"

Sometimes very easy to debug an AE.. ;)

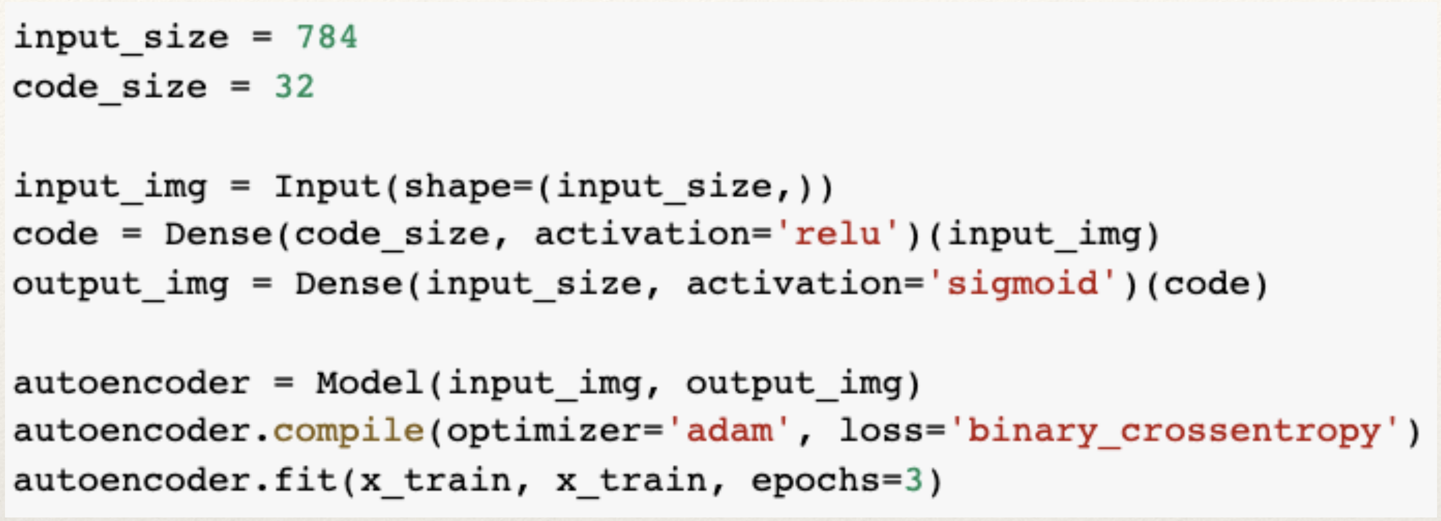


Ehm..



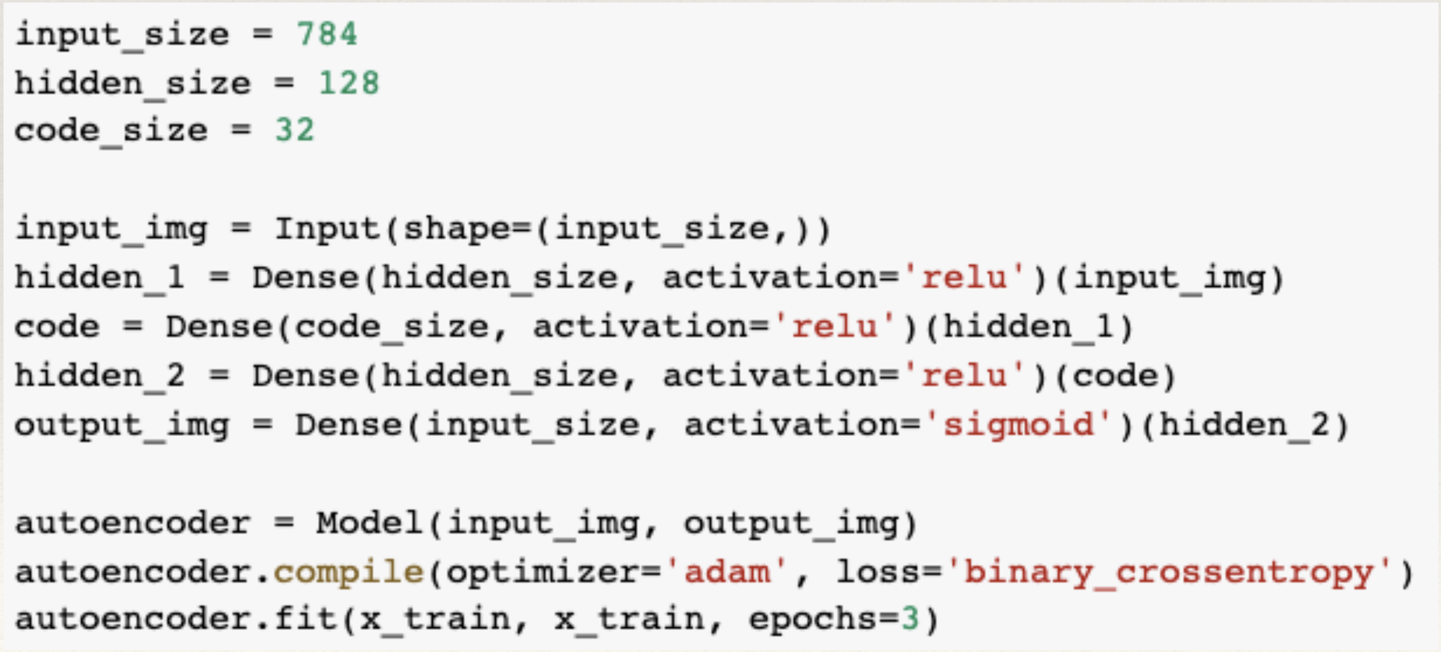
OK..

Shallow vs Deep AE



```
input_size = 784
code_size = 32

input_img = Input(shape=(input_size,))
code = Dense(code_size, activation='relu')(input_img)
output_img = Dense(input_size, activation='sigmoid')(code)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

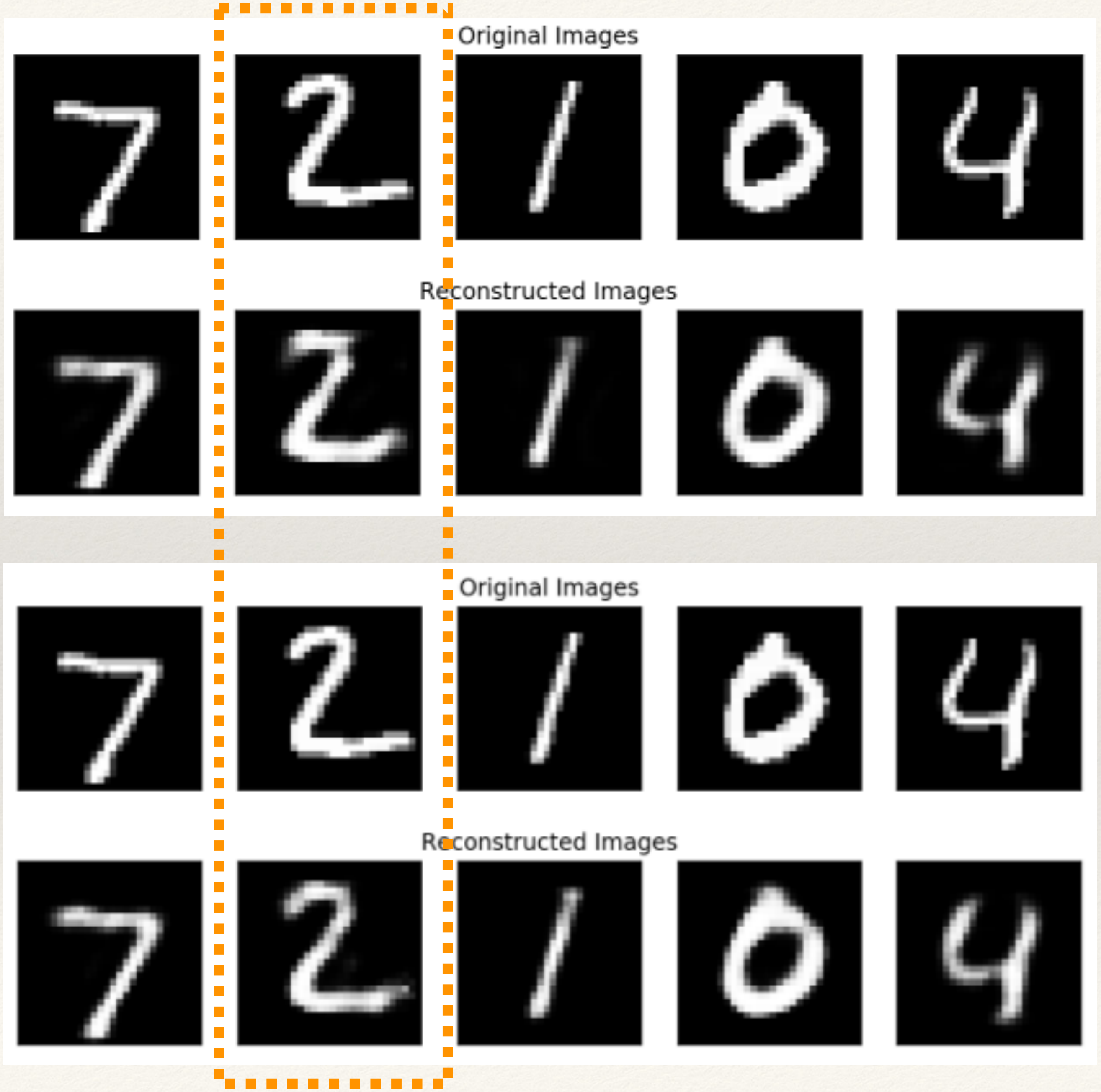
Shallow AE



```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

Deep (well…) AE

Shallow vs Deep AE

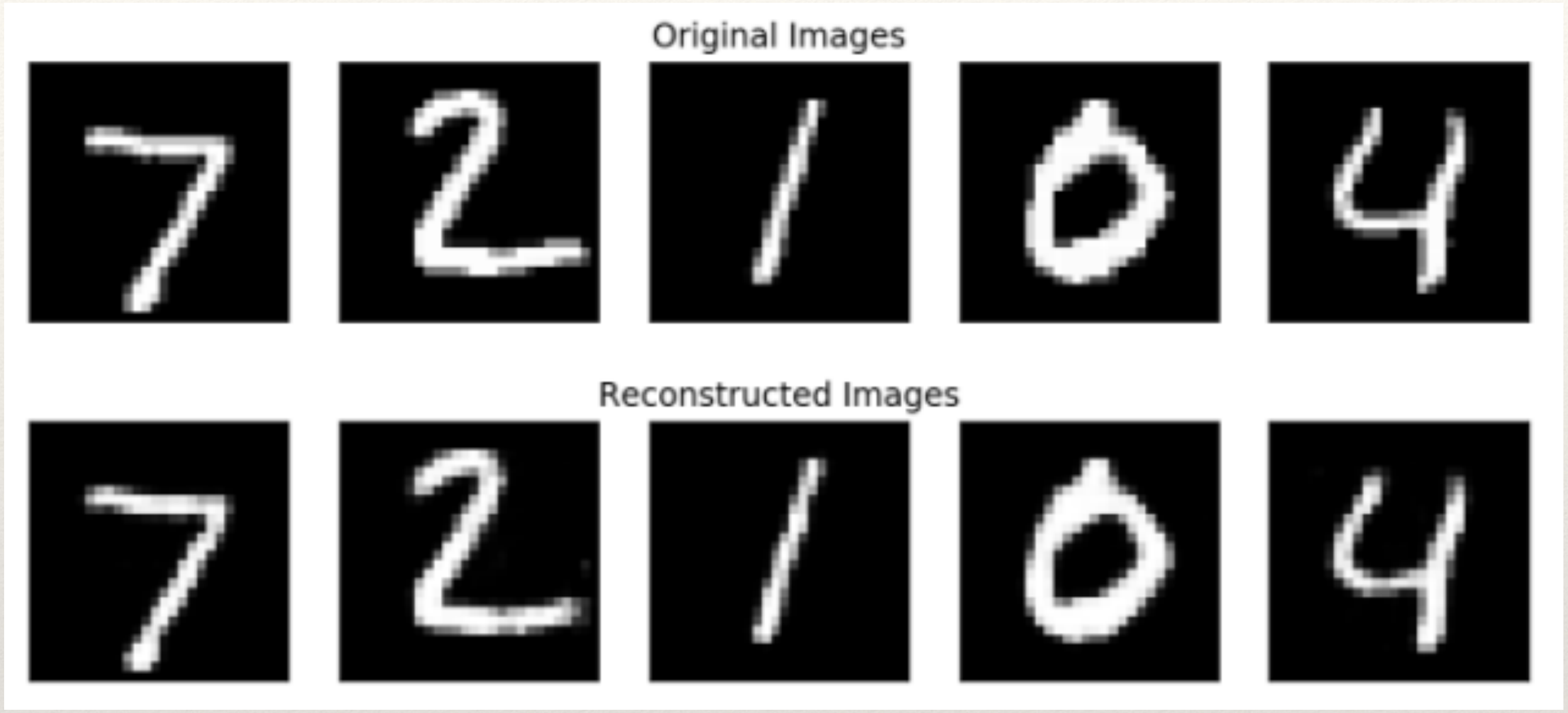


Shallow AE

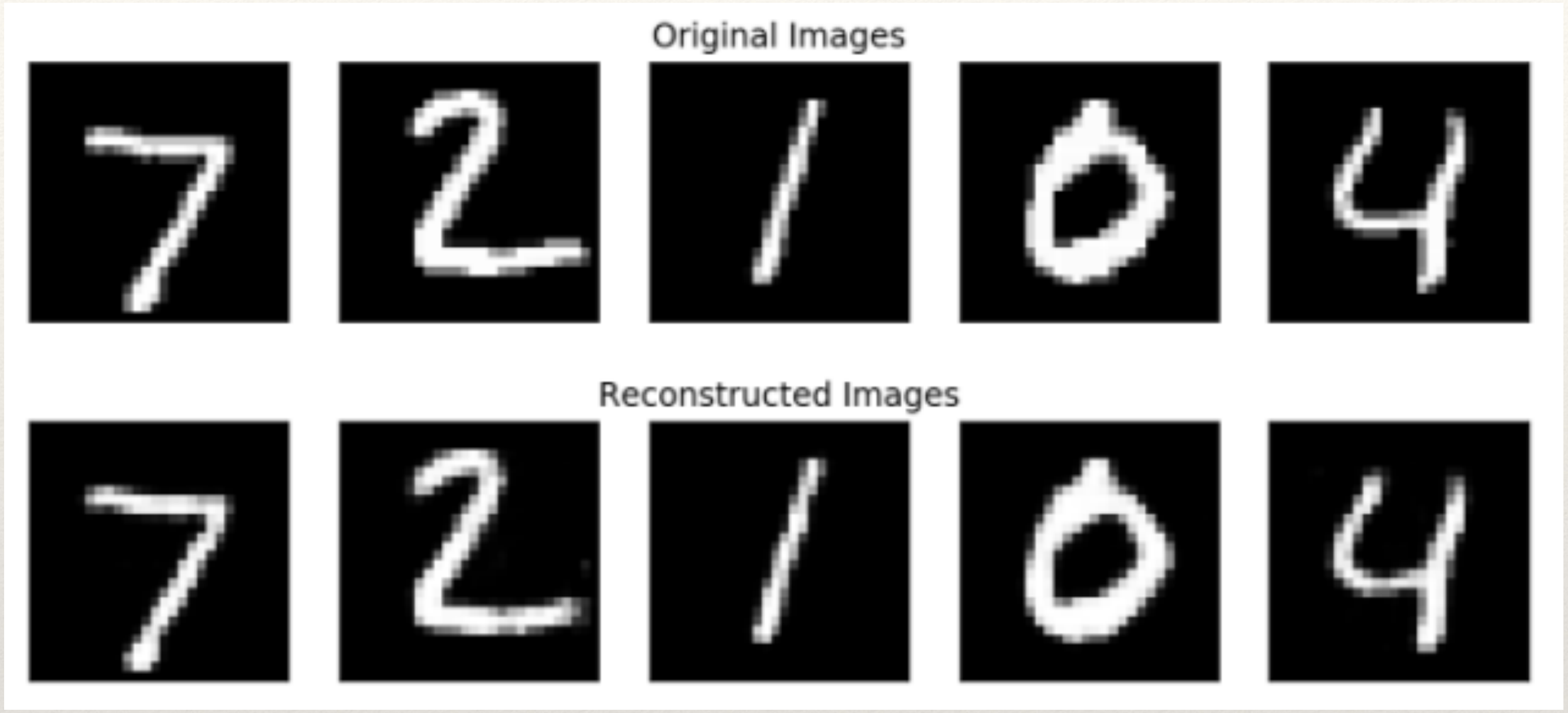
Check details of e.g. the "2" in the two cases

Deep (well…) AE

QUIZ.. can you guess which AE arch gives me this?



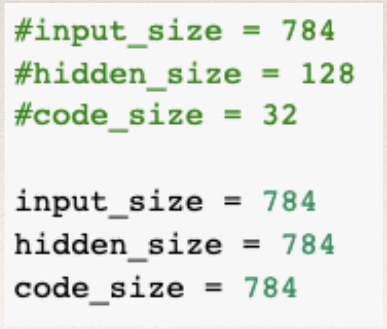
QUIZ.. can you guess which AE arch gives me this?



Basic: trainable params: 209,968, then…

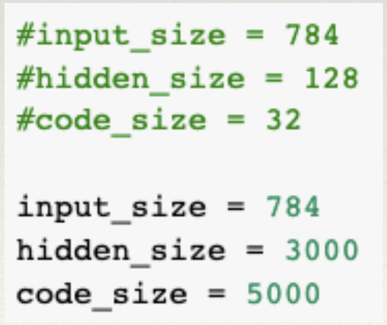
SOLUTION(s):

(for example..)



```
#input_size = 784
#hidden_size = 128
#code_size = 32

input_size = 784
hidden_size = 784
code_size = 784
```

Trainable params: 2,461,760



```
#input_size = 784
#hidden_size = 128
#code_size = 32

input_size = 784
hidden_size = 3000
code_size = 5000
```

Trainable params: 34,715,784

More on AE architectural choice

We can make an AE very powerful by **increasing # of layers, # nodes per layer and (most importantly) the code size**.

- higher values of these hyperparameters → AE will learn more complex codings

Careful about making it too powerful, or it will simply learn to copy its inputs to the output, without learning any meaningful (latent) representation. Not the right way to go, because:

- you do NOT need an ANN to get the identity function :)
- an AE that reconstructs the training data perfectly will be overfitting

→ **"sandwich" AE architecture**, with code size (deliberately) kept small

Code layer with lower dimensionality than the input data → "**undercomplete**" AE: it won't be able to directly copy its inputs to the output, and will be forced to learn intelligent features.

- i.e. learn patterns (e.g. how "0" differs from "7"), and encode it in a compact form. Random images will not be reconstructed well by an undercomplete AE, but in real world images there is luckily a lot of correlations/dependencies, so it just works

A variety of AE techniques

The key is to **force the AE to learn useful features**.

Various ways to do so:

- by keeping the code size small (see previous slides)
- by adding random noise and forcing to recover the noise-free data (next)
- by using regularisation

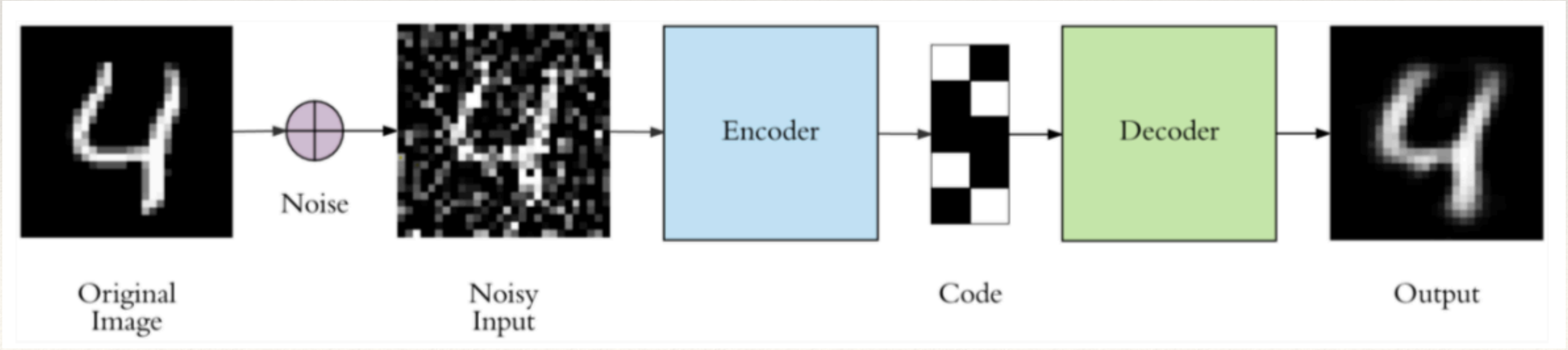
Denoising AE (dAE)

The AE learns an intelligent representation of the input being forced to do so by keeping the code layer small.

An AE can learn useful features by adding random noise to the input and making it recover the original noise-free data

- in this case, a straight copy input-to-output will not work
- the AE is forced to identify the noise and subtract it, thus exposing the underlying meaningful data

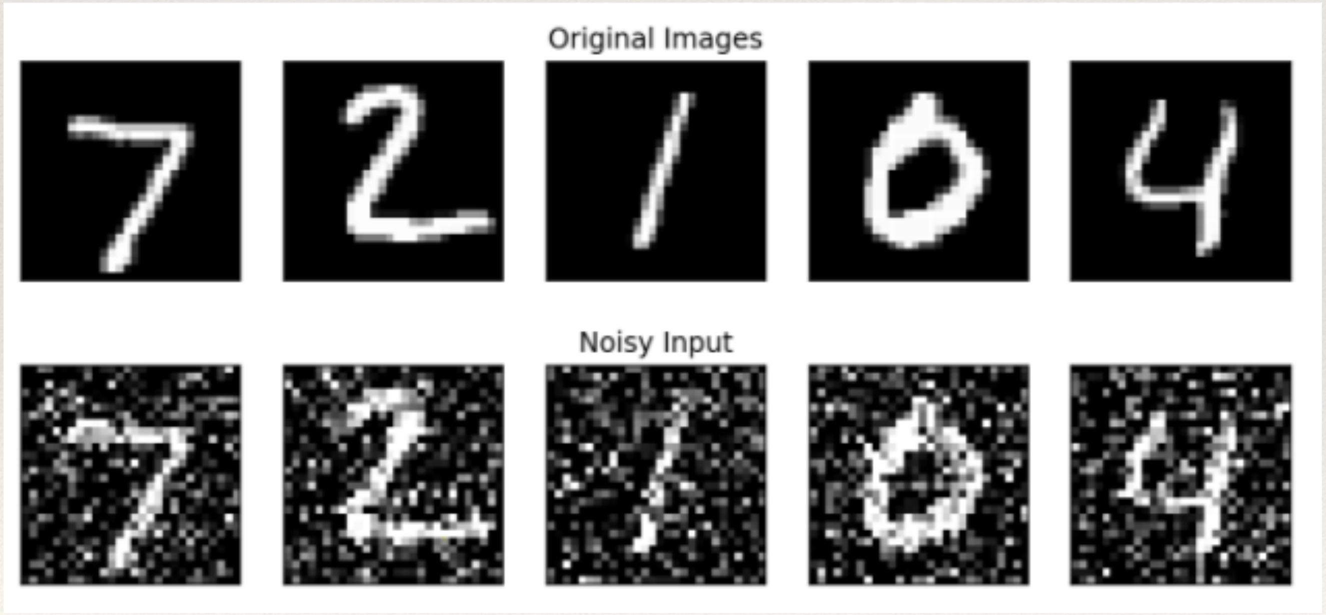
This is a **Denoising AE**.



dAE implementation: an example

We take the original images, add random Gaussian noise, and send these images as input to the dAE

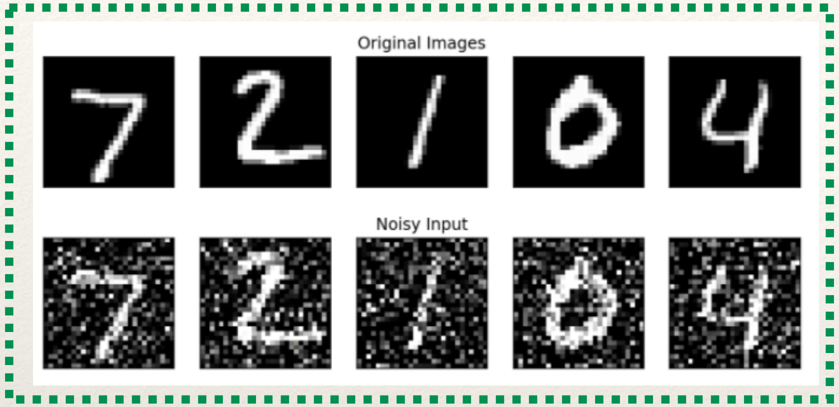
- in this way, the dAE <u>does not see the original images at all</u>
- we expect it to regenerate the noise-free original images, though



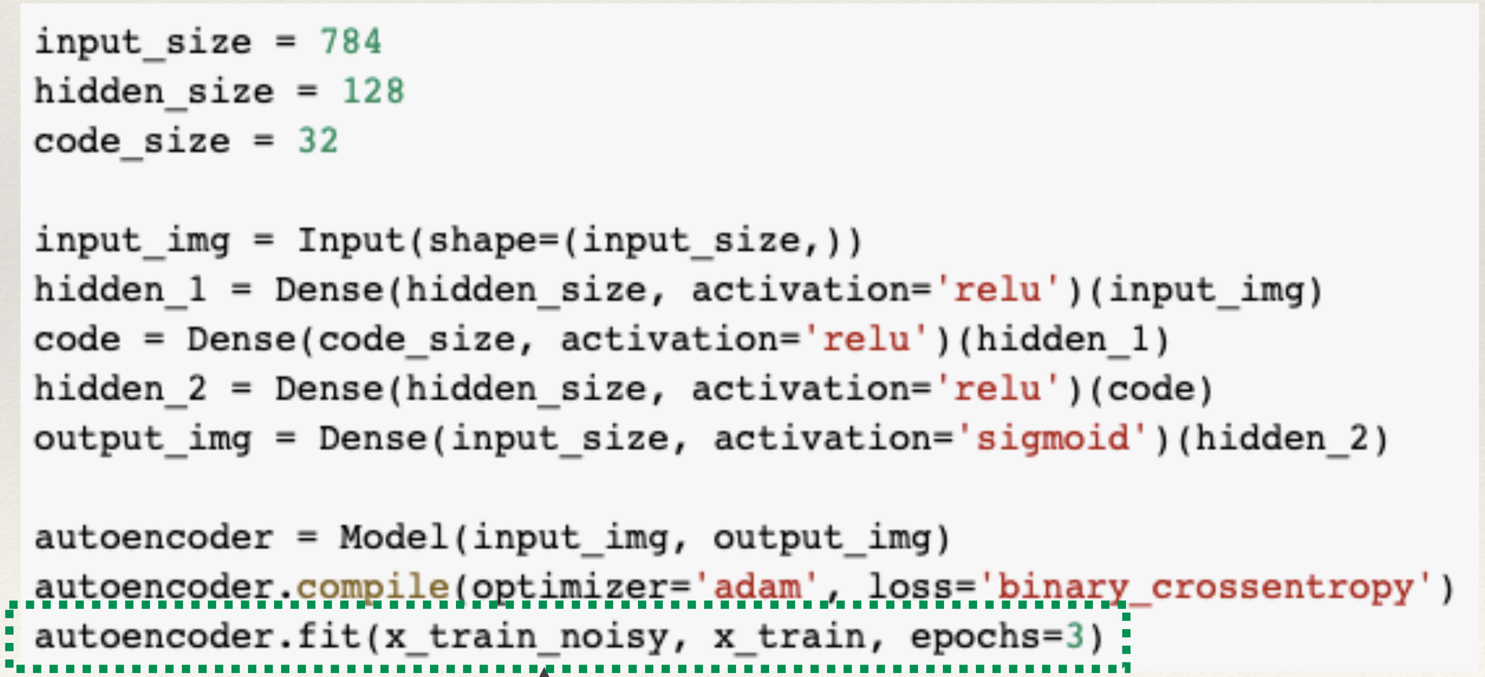
[See the code as of how to generate noise]

dAE implementation: model creation and fit

As before, with only a change:

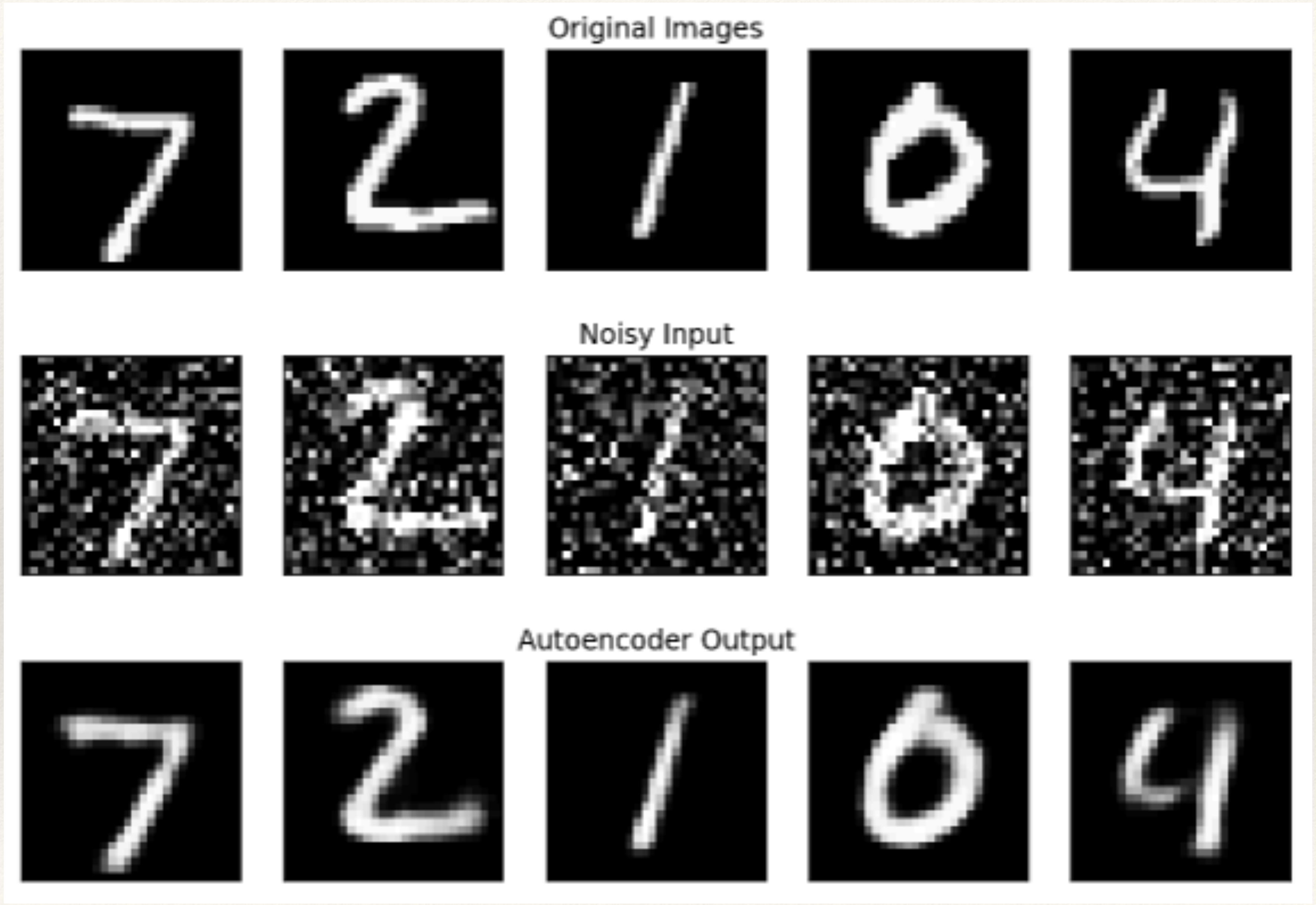


The input to the dAE is the noisy data.
The expected target is the original noise-free data



```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train_noisy, x_train, epochs=3)
```

dAE implementation: results



Sparse AE (sAE)

Method: regularise the AE by using a "sparsity constraint", such that only a fraction of the nodes ("active nodes") would have non-0 values

- how? add a penalty term to the loss function such that only a fraction of the nodes become active.

This simple trick forces the AE to represent each input using a combination of a smaller # of nodes (not all of them), but still demands it to discover interesting structure in the data

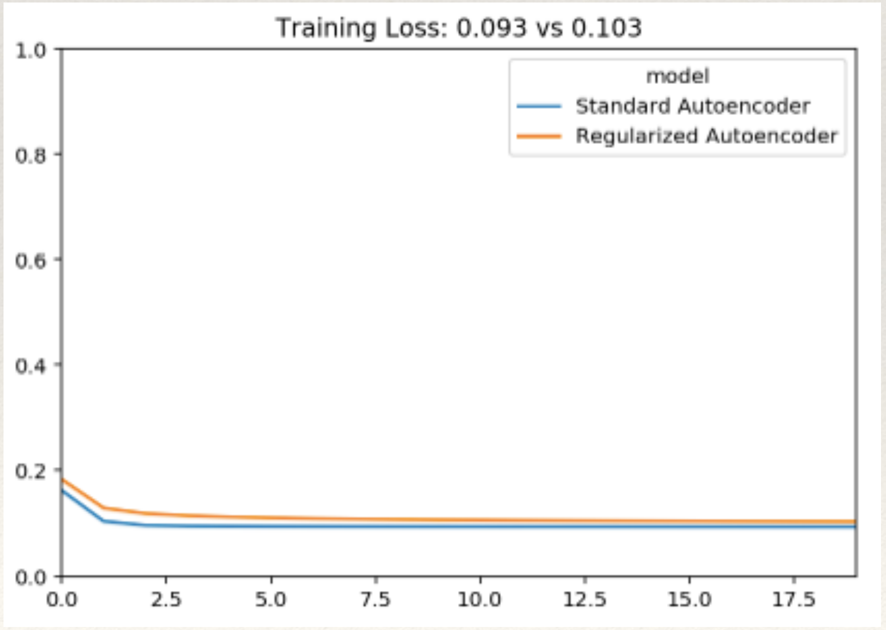
- a plus: it works even if you want to keep the code size large, as only a small subset of the nodes will be active at any time

Sparse AE (sAE): implementation

```
code = Dense(code_size, activation='relu')(input_img)
```

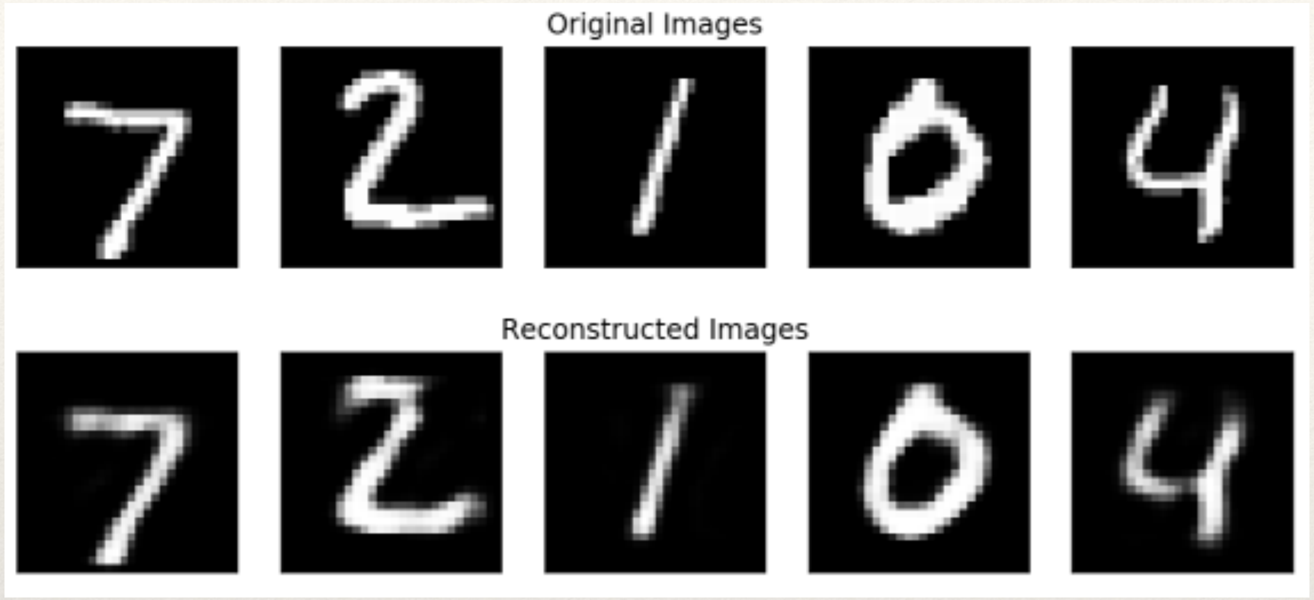
```
code = Dense(code_size, activation='relu', activity_regularizer=l1(10e-6))(input_img)
```

add the activity_regularizer parameter and specifying the regularisation strength (typically [0.001, 0.000001])

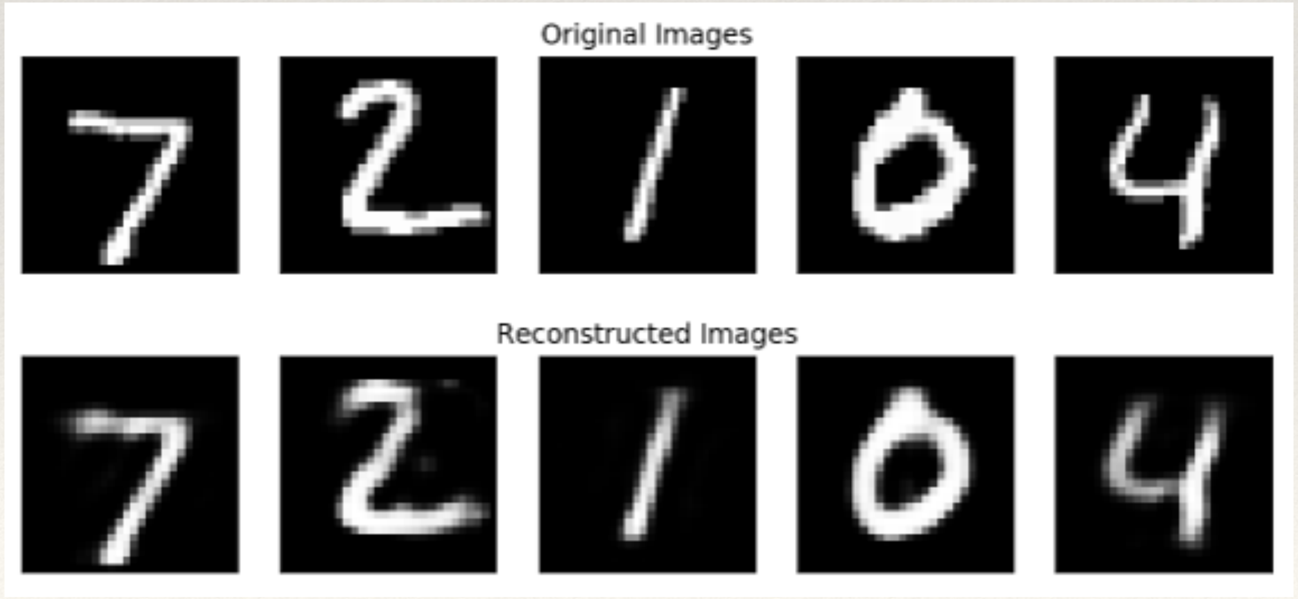


NOTE: the final loss of the sAE model will be higher than the standard AE (due to the added regularisation term), but that’s fine

Sparse AE (sAE): results



(Shallow) AE

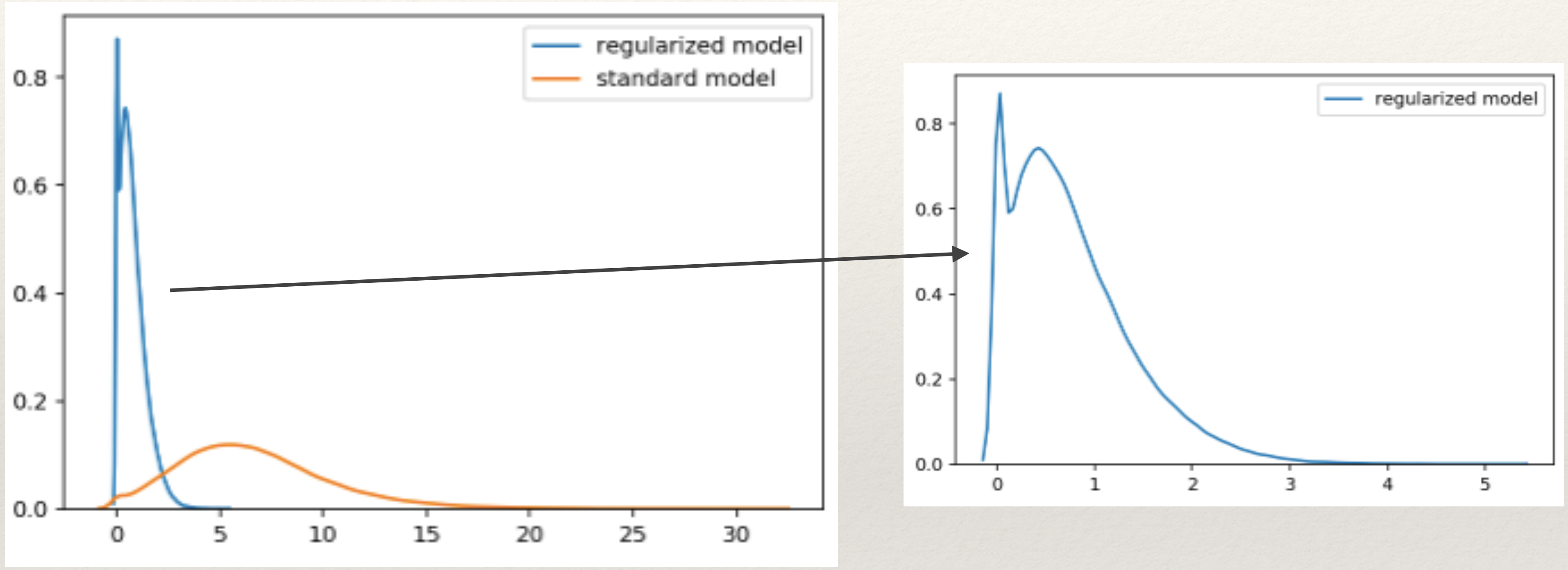


Sparse AE

Visualise the "sparsity" in sAE

Helpful to visualise encodings generated by the sAE to be indeed sparse

- check the distribution of code values in both models, for the images in the test set



Standard AE: mean 6.64, sAE: mean 0.76. A quite big reduction!

- definitely, a large chunk of code values in the sAE model are indeed 0, which is what we wanted
- The variance of the sAE model is also fairly low

Concluding on AEs

Compared to CNNs, not so widely used in real-world applications..

- as a compression method, they don't perform better than its alternatives
- their data-specificity makes them impractical as a general technique

But..

- … first of all you can use the power of CNN and AE together ①
- … and nevertheless, there are extremely interesting use-cases of AE applications ②

Convolutional AE ①

If you are dealing with images, and you might need an AE (e.g. for unsupervised pretraining or dimensionality reduction), but the AEs we have seen so far will not work well (unless the images are very small)

- CNNs are far better suited than dense networks to deal with images

So, if you want to build an AE for images, you will need to build a more complex architecture: a **convolutional AE**, built this way:

- the **encoder** is a **regular CNN** composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing its depth (i.e., the number of feature maps)
- the **decoder** must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use **transpose convolutional layers**; alternatively, you could combine **upsampling layers** with convolutional layers
    - upsampling layers (UpSampling2D) simply double the dimensions of the input
    - transpose convolutional layer (Conv2DTranspose) perform an inverse convolution operation

AEs use-cases ②

- **Data denoising**: we discussed this (for easy images, at least)

- **Dimensionality reduction**: visualising high-dimensional data is challenging

  - t-SNE is one of the most commonly used method but struggles with large number of dimensions (typically above 32). AEs are used as a preprocessing step to reduce the dimensionality, and this compressed representation is used by t-SNE to visualise the data in 2D space

- in the form of **Variational Autoencoders** (**VAE**): a more modern and complex use-case of AEs

  - w.r.t vanilla AEs - that learn an arbitrary function - VAEs learns the parameters of the probability distribution modelling the input data. By sampling points from this distribution we can also use the VAE as a generative model..

That’s it,

for our Lab on **AE**s