50.039 Theory and Practice of Deep Learning

W10-S1 Generative Models in Deep Learning

Matthieu De Mari

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

Outline

In this lecture

- Autoencoders definition
- Autoencoders with fully connected layers
- Fractionally strided convolution layers
- Autoencoders with convolutions and deconvolutions
- Autoencoder uses

In the next lectures

- Variational Autoencoder and noisy latent representations
- Generative Adversarial Networks
- More advanced GANs

What we have done so far

All our models so far had something in common:

→The dimensionality of the inputs was always superior to the dimensionality of the outputs.

E.g. Receiving Tabular data, images, time series, words graphs as inputs and predicting a single value (regression) or a given class in a finite number of classes (much smaller).

Only Seq2Seq models and SkipGram models somehow broke this idea. (But SkipGram was not doing so great anyway).

What we have done so far

All our models so far had something in common:

→ **The dimensionality of the inputs was always superior to the dimensionality of the outputs.**

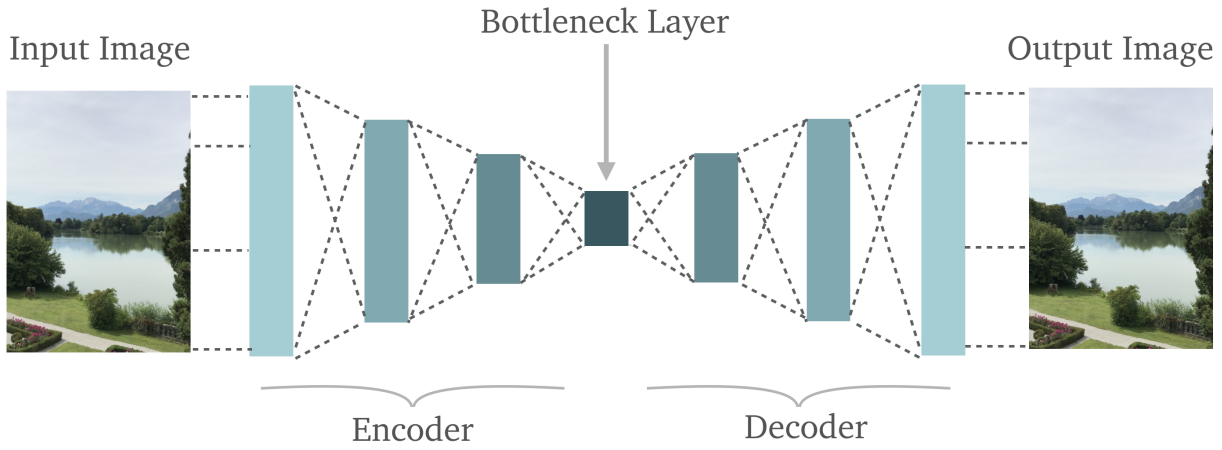
Question:

→ **Does that mean it is impossible for Deep Learning models to have a higher dimensionality for outputs?**

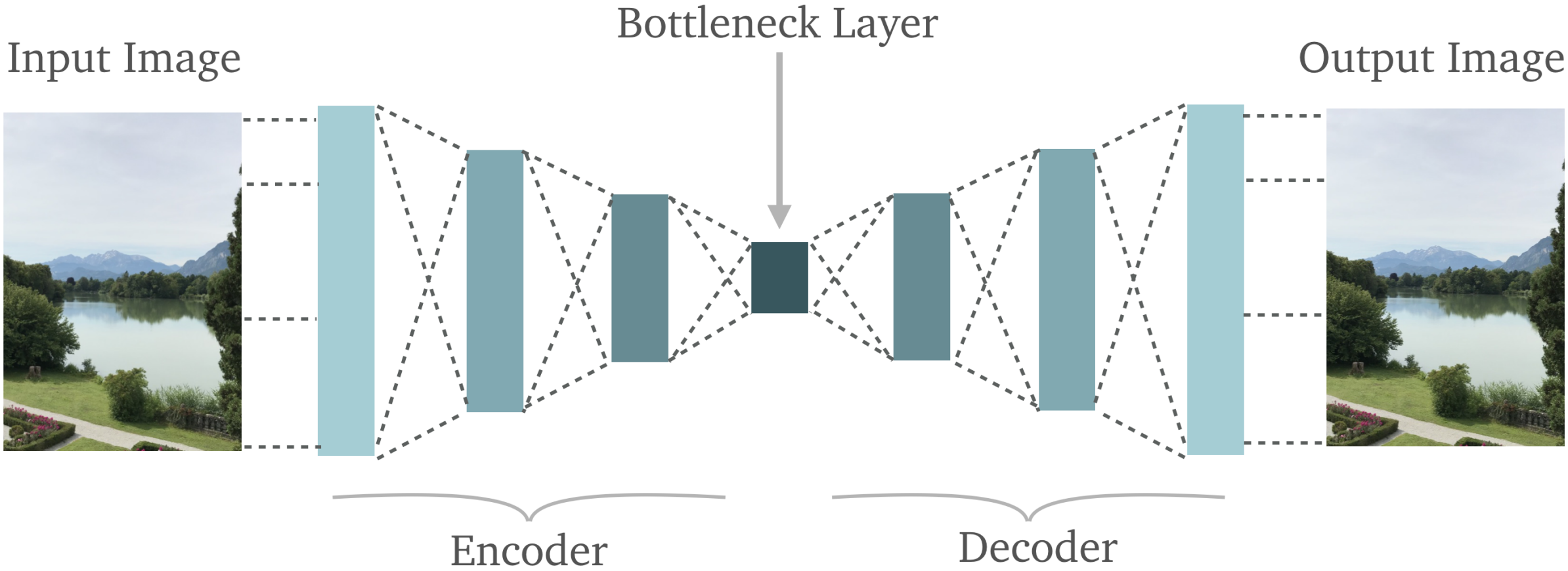
AutoEncoders: a definition

Definition (AutoEncoder):

An **AutoEncoder** is a neural network that learns to **copy its input to its output**. Basically, it attempts to approximate the identity function.



AutoEncoders: a definition

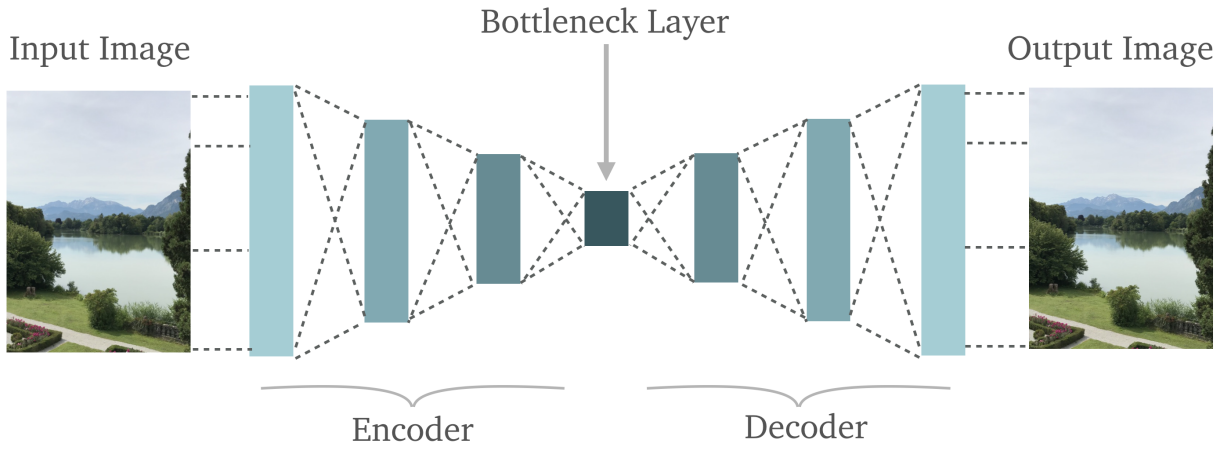


AutoEncoders: a definition

Definition (AutoEncoder):

An **AutoEncoder** is a neural network that learns to **copy its input to its output**. Basically, it attempts to approximate the identity function.

It has an internal (hidden) layer that describes a code used to represent the input, and is called the **feature representation** or **latent representation of the input**.



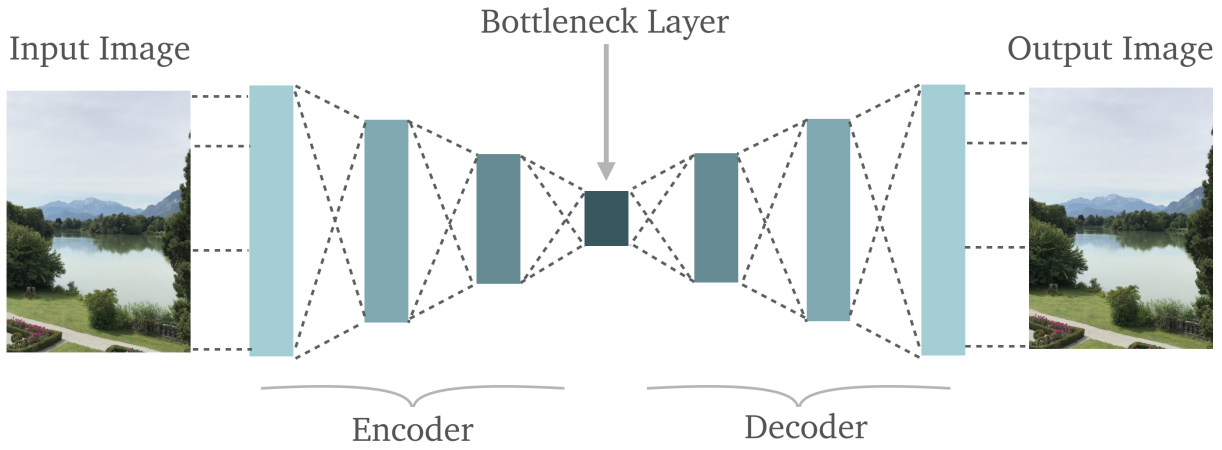
AutoEncoders: a definition

Definition (AutoEncoder):

An **AutoEncoder** is a neural network that learns to **copy its input to its output**. Basically, it attempts to approximate the identity function.

It is constituted of two main parts:

- an **encoder** that maps the input into the code,
- and a **decoder** that maps the code to a reconstruction of the input.

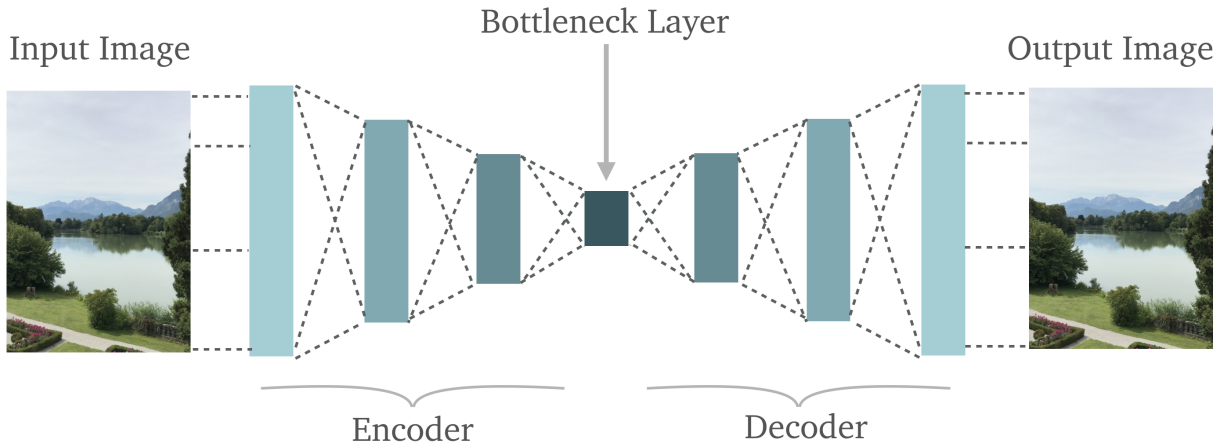


AutoEncoders: a definition

Definition (AutoEncoder):

Autoencoders are usually restricted in their **latent representation dimensionality**.

This forces them to reconstruct the input approximately, **preserving only the most relevant aspects of the data in the copy**.

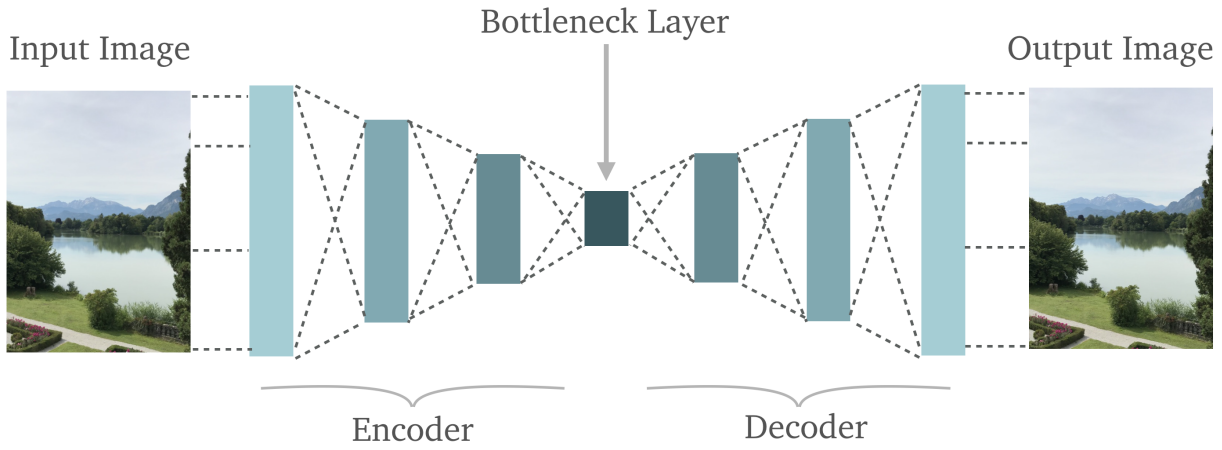


AutoEncoders: a definition

Definition (AutoEncoder):

AutoEncoders are (un)supervised artificial neural networks, which

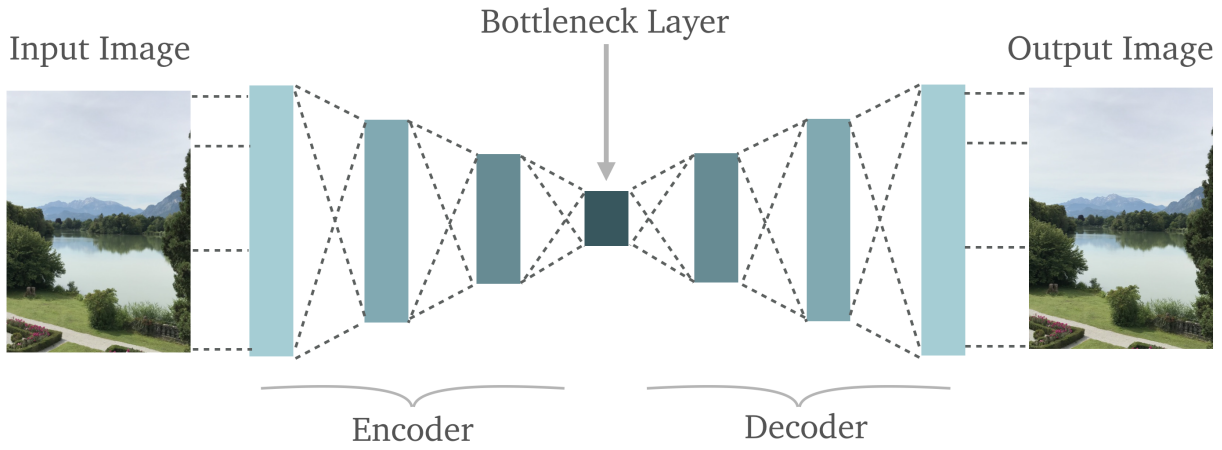
- learn how to efficiently compress and encode data,
- and then learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible.



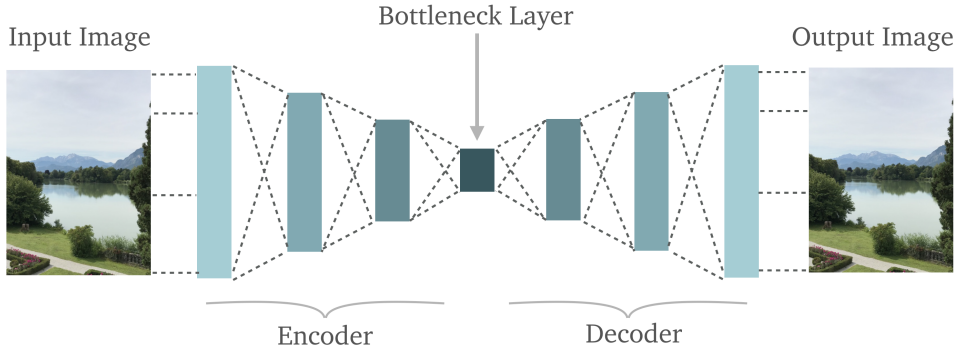
AutoEncoders structure

Autoencoders consists of 4 main parts, to be discussed in the next few slides:

1. **Encoder**
2. **Bottleneck**
3. **Decoder**
4. **Reconstruction Loss**

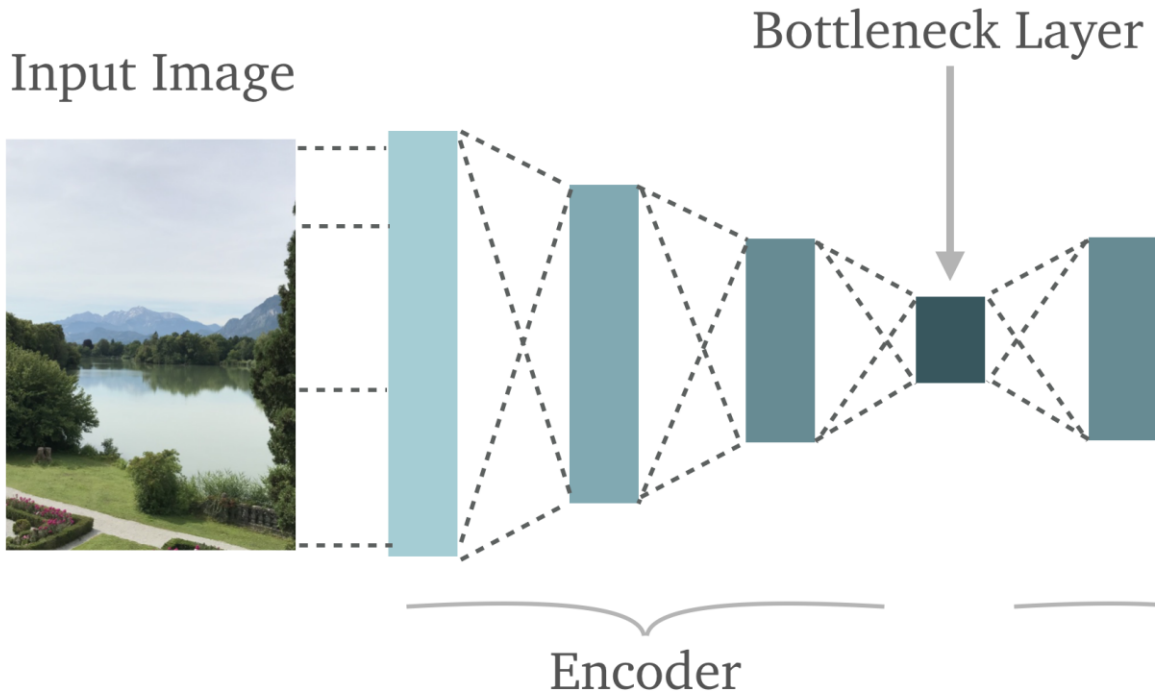


AutoEncoders structure

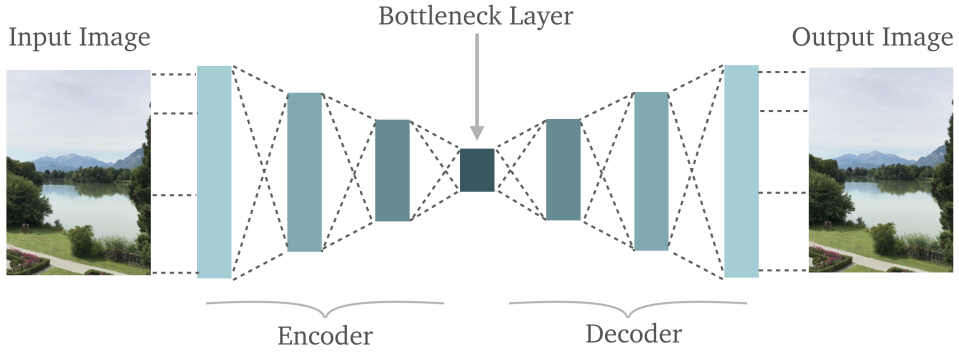


1. **Encoder**: In which the model learns how to reduce the input dimensions and compress the input data into an encoded representation.

   An encoder will, for instance, consist of **successive fully connected layers, decreasing in size.**

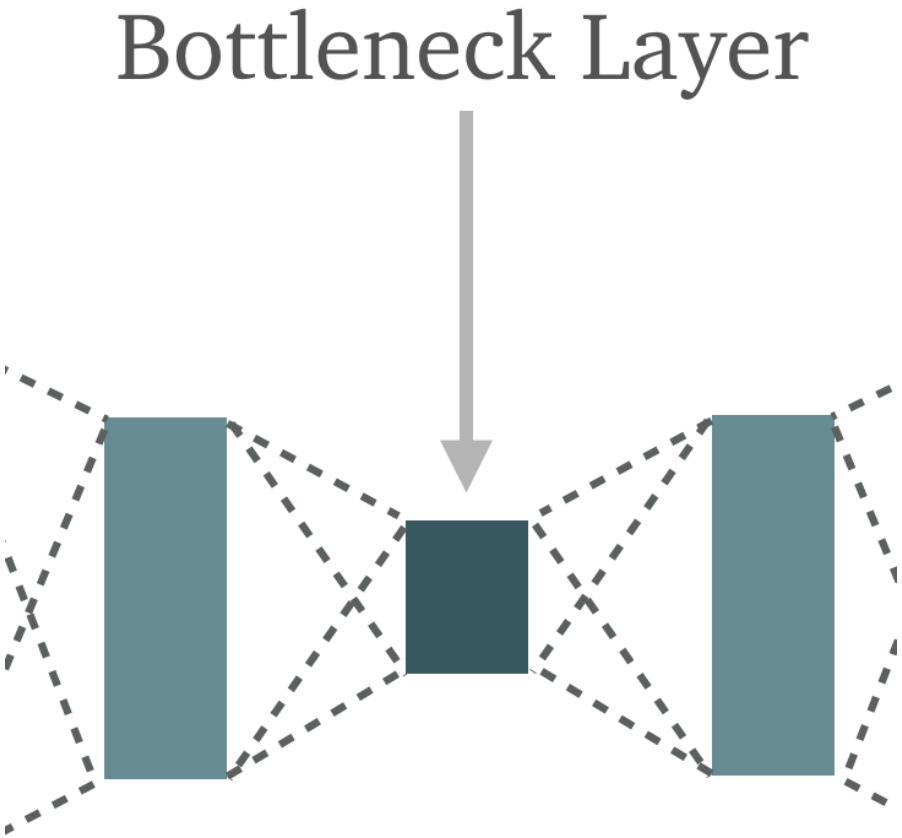


AutoEncoders structure

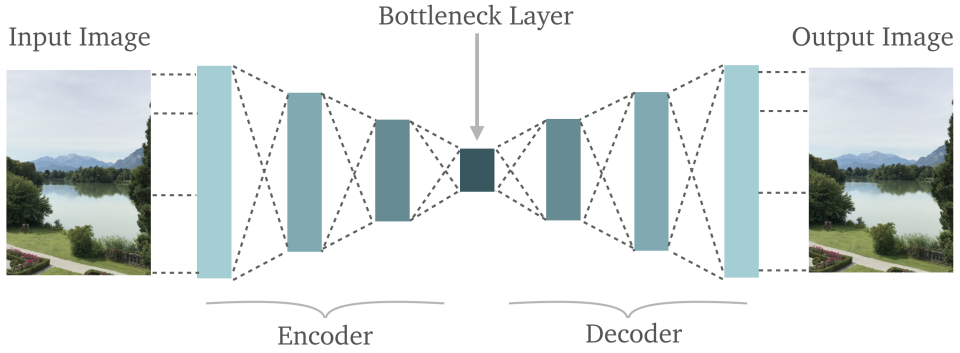


2. **Bottleneck**: which is the layer that contains the compressed representation of the input data.

   Technically, we want the dimensionality to be the lowest possible dimension of the input data (more on this later).

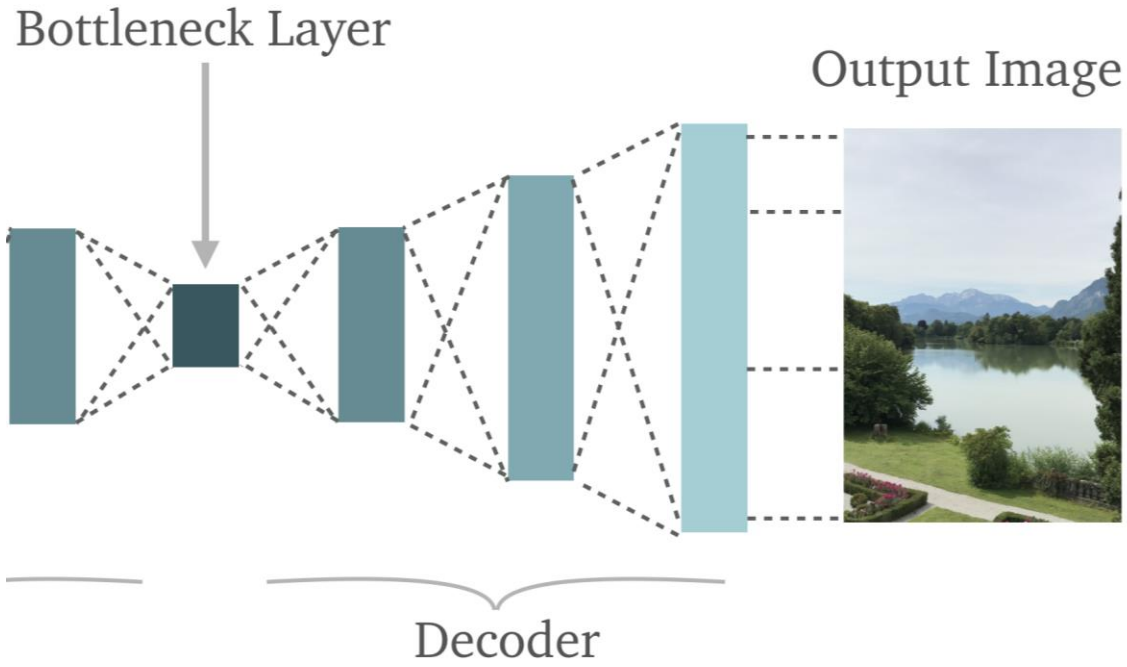


AutoEncoders structure



3. **Decoder**: In which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible.

   Our decoder will, for instance, consist of **successive fully connected layers, increasing in size, back to the original input size.**



AutoEncoders structure

Autoencoders consists of 4 main parts:

4. **Reconstruction Loss**: This is the method that measures measure how well the decoder is performing and how close the output is to the original input.

   In the case of images, we can typically use an MSE on all pixels, or variations of it.

Example: input x on the right, output y on the left.

$$MSE(x, y) = \sqrt{\sum_{i,j} (x_{i,j} - y_{i,j})^2}$$

On reversibility of layers

What would a perfect autoencoder consist of?

- Let us call f, the encoder function and g the decoder one. The autoencoder $h = fog$ attempts to reproduce the identity function. A perfect autoencoder would then require $f = g^{-1}$
- In practice however, it is difficult to train an autoencoder to replicate the identity function exactly. This is because of **different weights** and **activation functions**, as well as the **low-dimensionality latent representation**.

```
# Encoder part will be a simple FC + ReLU.
self.encoder = nn.Sequential(nn.Linear(28*28, hidden_layer), nn.ReLU(True))

# Decoder part will be a simple FC + Tanh
self.decoder = nn.Sequential(nn.Linear(hidden_layer, 28*28), nn.Tanh())
```

On reversibility of layers

- At best, we can simply ensure that the layers preserve dimensionality.
- That is, the encoder compresses an input into a low-dimensionality latent representation in the same way that the decoder decompresses the low-dimensionality latent representation into an output of the same dimensions.
- **This is simply done, for instance, with fully connected layers, with mirrored sizes.**

```
# Encoder part will be a simple FC + ReLU.
self.encoder = nn.Sequential(nn.Linear(28*28, hidden_layer), nn.ReLU(True))

# Decoder part will be a simple FC + Tanh
self.decoder = nn.Sequential(nn.Linear(hidden_layer, 28*28), nn.Tanh())
```

A first toy example: MNIST autoencoder with fully connected layers

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code).

The dataset is simply MNIST, input dimensions are 28 by 28 pixel images, greyscale.

```
# Data Preprocessing
# - ToTensor
# - Image Normalization
transform = transforms.Compose([transforms.ToTensor(), \
                                transforms.Normalize((0.1307,), (0.3081,))])
```

```
# Train datasets/dataloaders
train_set = torchvision.datasets.MNIST(root='./data', \
                                       train = True, \
                                       download = True, \
                                       transform = transform)
train_loader = torch.utils.data.DataLoader(train_set, \
                                           batch_size = 32, \
                                           shuffle = False)
```

A first toy example: MNIST autoencoder with fully connected layers

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code).

Our model consists of

- An **encoder**, simply modelled as a **fully connected layer** from **28*28** into a **hidden layer size**, to be decided.
- A **decoder**, simply modelled as a **fully connected layer** from the **hidden layer size**, back into **28*28**.

A first toy example: MNIST autoencoder with fully connected layers

```python
# Define AutoEncoder Model for MNIST
class MNIST_Autoencoder(nn.Module):

    def __init__(self, hidden_layer = 3):

        # Init from nn.Module
        super().__init__()

        # Encoder part will be a simple FC + ReLU.
        self.encoder = nn.Sequential(nn.Linear(28*28, hidden_layer), nn.ReLU(True))

        # Decoder part will be a simple FC + Tanh
        self.decoder = nn.Sequential(nn.Linear(hidden_layer, 28*28), nn.Tanh())


    def forward(self,x):

        # Forward is encoder into decoder
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

A first toy example: MNIST autoencoder with fully connected layers

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code).

Our model will be trained on the MNIST dataset, with the following parameters:

- Start with **hidden_size = 5**, that is the **dimensionality of the latent representation in the bottleneck layer**.
- **MSE** as the **reconstruction loss**.
- Optimizer is Adam, with default parameters.
- Mini-batch with size 128 and 25 epochs.

A first toy example: MNIST autoencoder with fully connected layers

```
# Initialize MNIST Autoencoder
torch.manual_seed(0)
model = MNIST_Autoencoder(hidden_layer = 5).to(device)
```

```
# Defining Parameters
# - MSE Loss, which will be our reconstruction loss for now
# - Adam as optimizer
# - 25 Epochs
# - 128 as batch size
num_epochs = 25
batch_size = 128
distance = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), weight_decay = 1e-5)
```

A first toy example: MNIST autoencoder with fully connected layers

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code).

Our model will be trained on the MNIST dataset, with the steps:

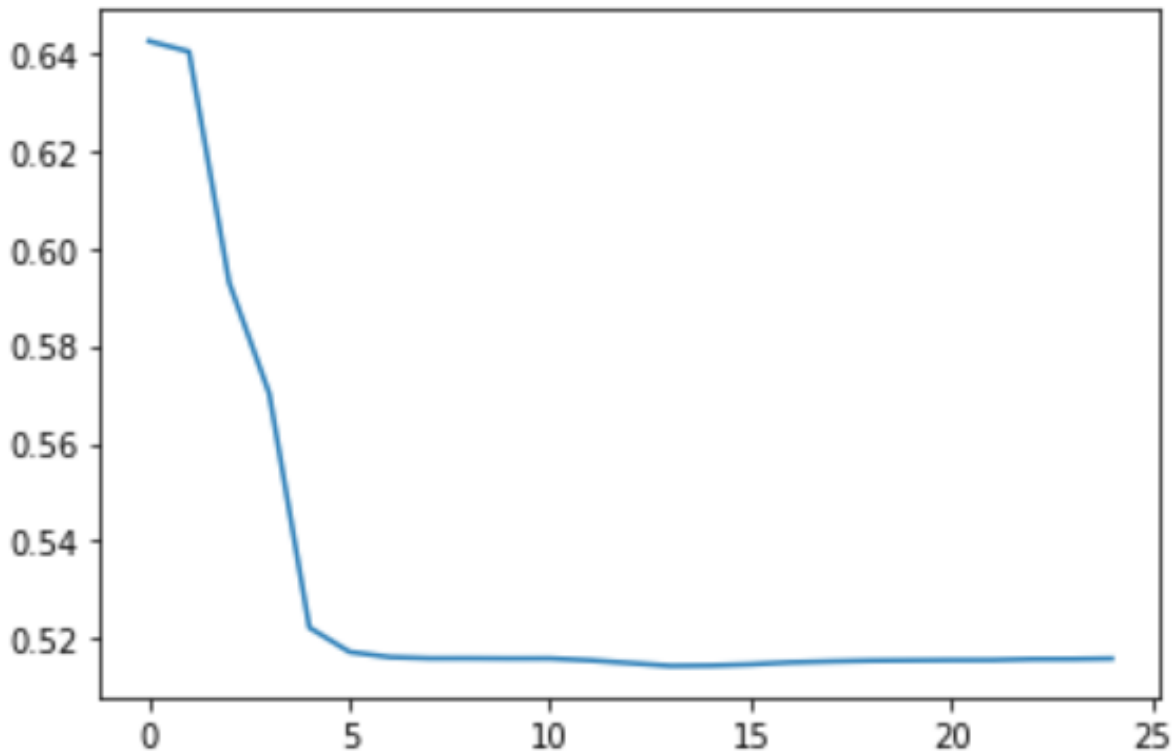
- Flatten the image, so it can be fed into FC layers.
- Forward pass.
- Backward pass and backpropagation.
- Loss and Outputs kept for visualization later on.

A first toy example: MNIST autoencoder with fully connected layers

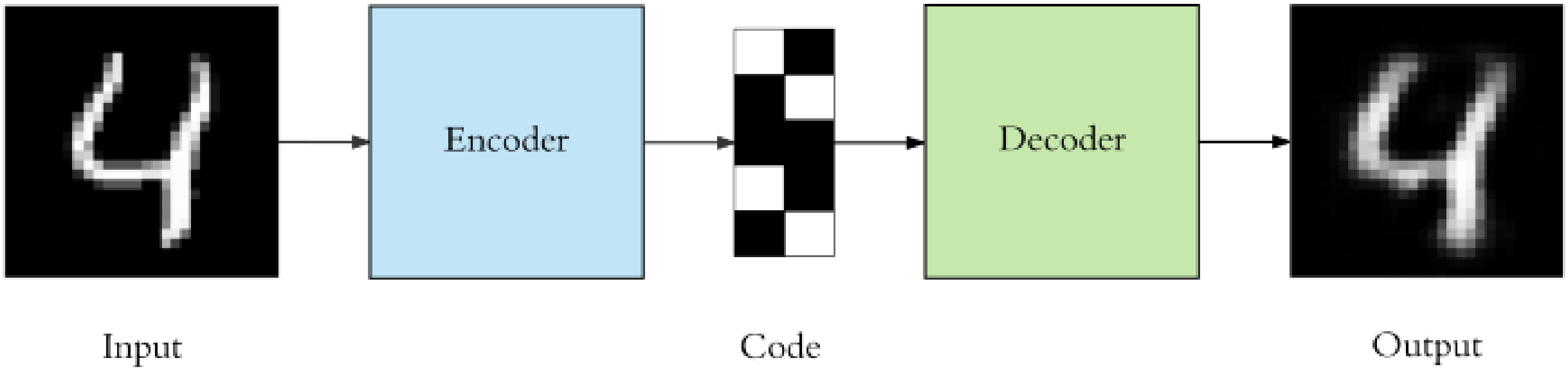
```
outputs_list = []
loss_list = []
for epoch in range(num_epochs):
    for data in train_loader:

        # Flatten image and send data to device
        img, _ = data
        img = img.view(img.size(0), -1)
        img = Variable(img).to(device)

        # Forward pass
        output = model(img)
        loss = distance(output, img)

        # Backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Display
    print('epoch {}/{}, loss {:.4f}'.format(epoch + 1, num_epochs, loss.item()))
    outputs_list.append((epoch, img, output),)
    loss_list.append(loss.item())
```

A first toy example: MNIST autoencoder with fully connected layers

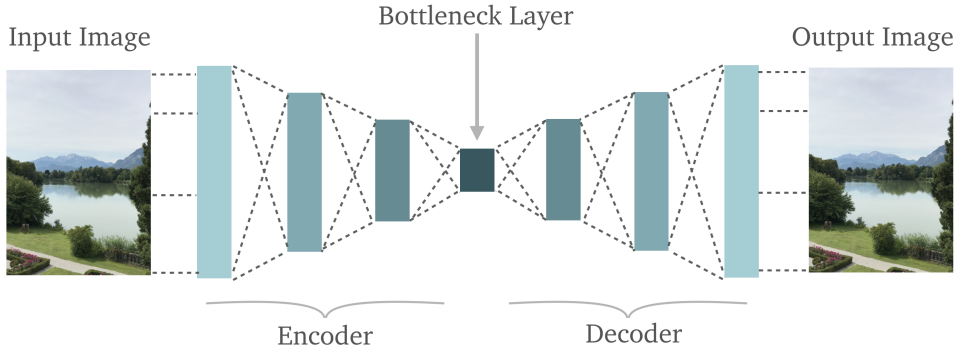
```
1  plt.figure()
2  plt.plot(loss_list)
3  plt.show()
```



A first toy example: MNIST autoencoder with fully connected layers

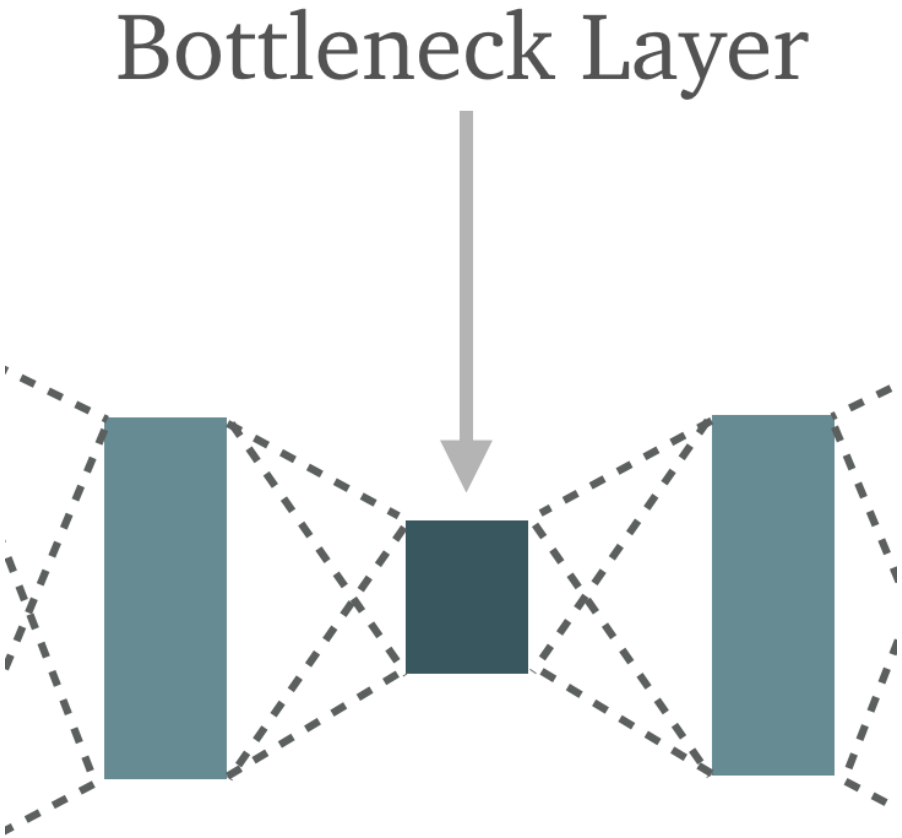


AutoEncoders structure (Reminder)



2. **Bottleneck**: which is the layer that contains the compressed representation of the input data.
   Technically, we want the dimensionality to be the lowest possible dimension of the input data.

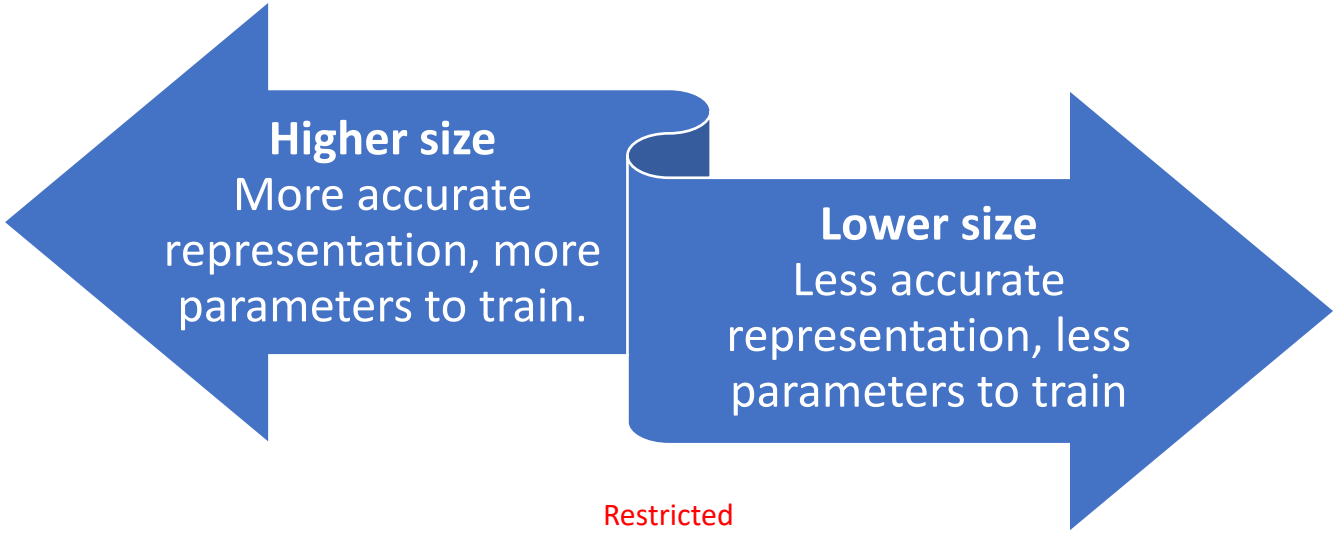
   How to decide on the best dimensionality for the hidden_size?



On latent dimensionality

How to decide on the best dimensionality for the hidden_size?

- Technically, we want the dimensionality to be the lowest possible dimension of the input data.
- It is often impossible to define it mathematically.
- More of a trial and error process, with a tradeoff at play.



On latent dimensionality

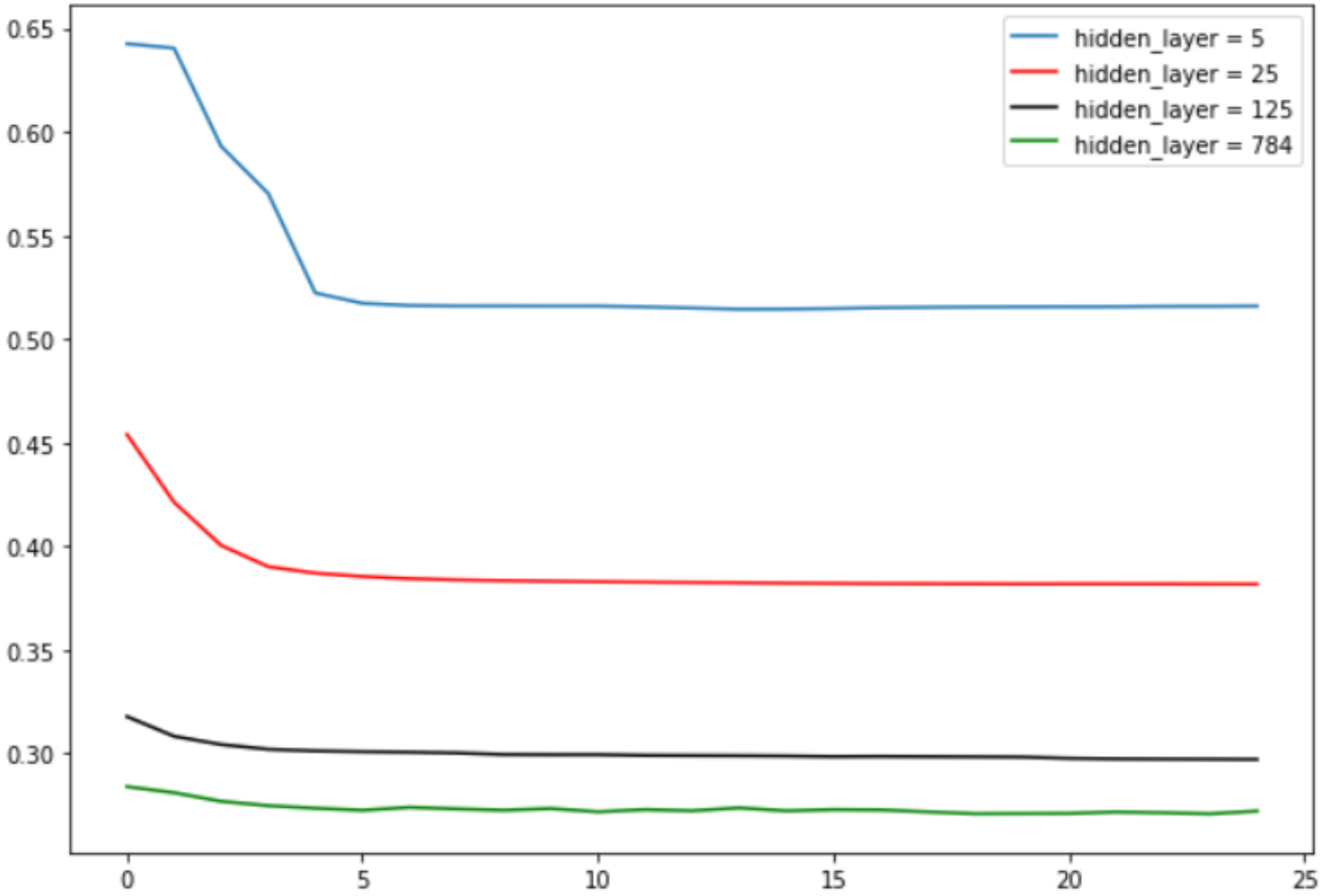
In Notebook 1, we have trained 4 models with different hidden layer sizes for comparison purposes.

- **Model #1:** hidden_size = 5
- **Model #2:** hidden_size = 25
- **Model #3:** hidden_size = 125
- **Model #4:** hidden_size = 784 = 28*28

We then visualize the performance of each model, in terms of loss and in terms of outputs visualization.

On latent dimensionality vs. loss effect

```
1 plt.figure(figsize = (10, 7))
2 plt.plot(loss_list, label = "hidden_layer = 5")
3 plt.plot(loss_list2, 'r', label = "hidden_layer = 25")
4 plt.plot(loss_list3, 'k', label = "hidden_layer = 125")
5 plt.plot(loss_list4, 'g', label = "hidden_layer = 784")
6 plt.legend(loc = 'best')
7 plt.show()
```

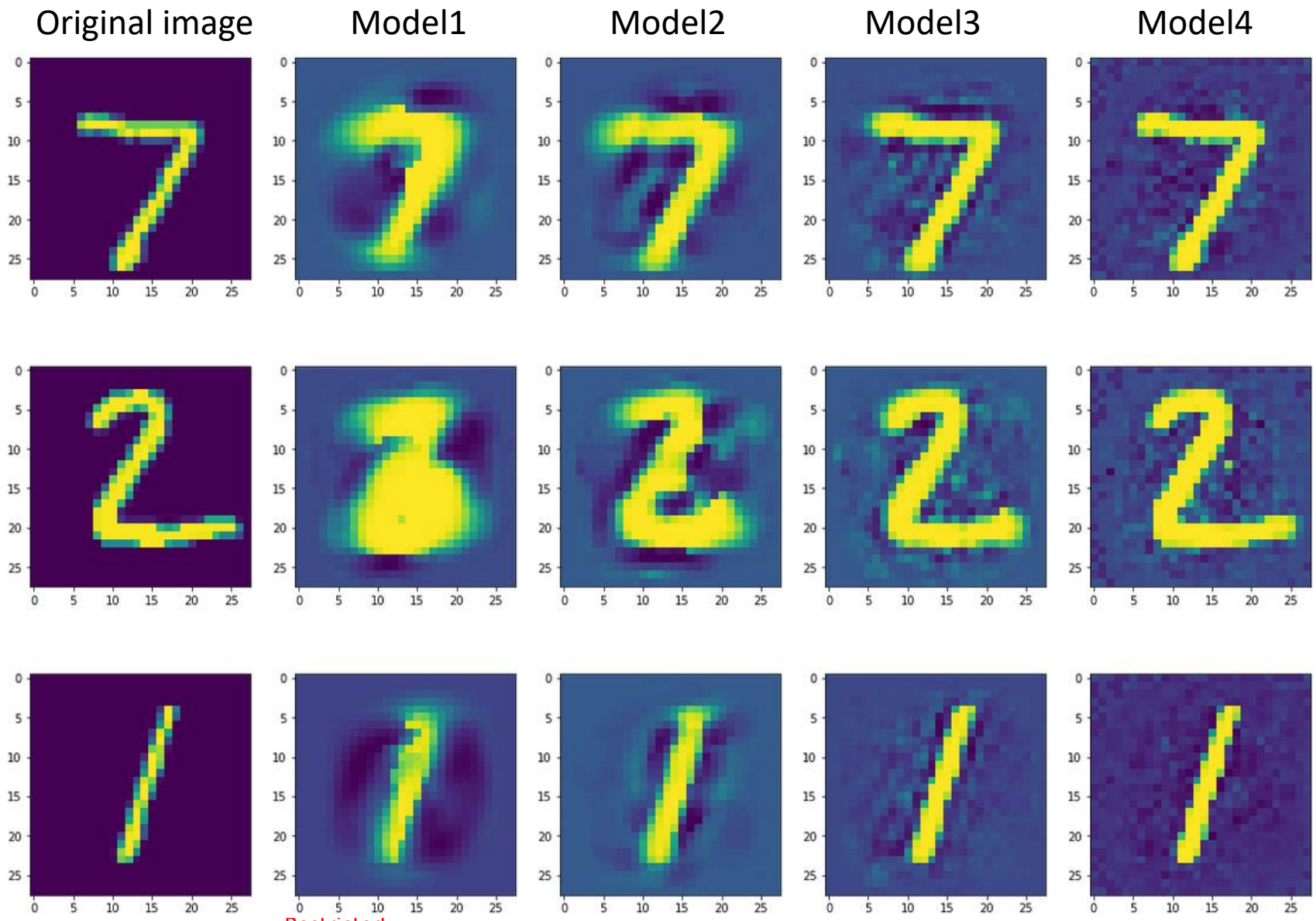


On latent dimensionality vs. reconstruction quality

```python
for data in test_loader:
    break
img, _ = data
img = img.view(img.size(0), -1)
img = Variable(img).to(device)
out1 = model(img).cpu().detach().numpy().reshape(5, 28, 28)
out2 = model2(img).cpu().detach().numpy().reshape(5, 28, 28)
out3 = model3(img).cpu().detach().numpy().reshape(5, 28, 28)
out4 = model4(img).cpu().detach().numpy().reshape(5, 28, 28)
img = img.cpu().detach().numpy().reshape(5, 28, 28)
```

```python
plt.figure(figsize = (20, 14))
n = 3
for i in range(n):
    plt.subplot(n, 5, 5*i + 1)
    plt.imshow(img[i])
    plt.subplot(n, 5, 5*i + 2)
    plt.imshow(out1[i])
    plt.subplot(n, 5, 5*i + 3)
    plt.imshow(out2[i])
    plt.subplot(n, 5, 5*i + 4)
    plt.imshow(out3[i])
    plt.subplot(n, 5, 5*i + 5)
    plt.imshow(out4[i])
```

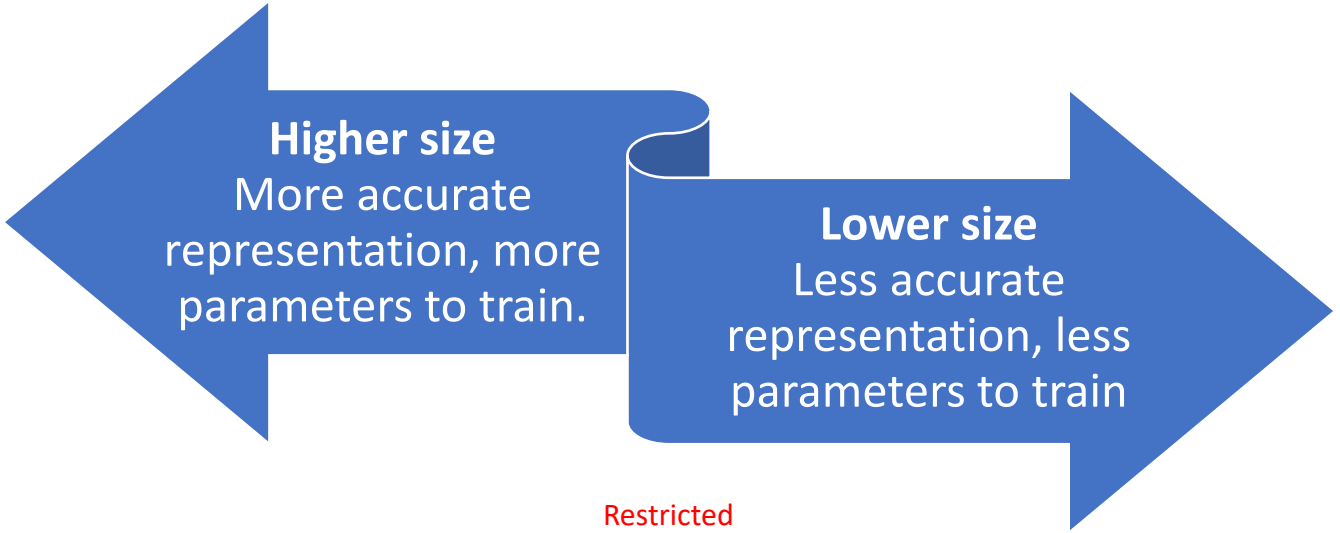
On latent dimensionality vs. reconstruction quality



On latent dimensionality (recap)

How to decide on the best dimensionality for the hidden_size?

- Technically, we want the dimensionality to be the lowest possible dimension of the input data.
- It is often impossible to define it mathematically.
- More of a trial and error process, with a tradeoff at play.



A first toy example: MNIST autoencoder with fully connected layers (Reminder)

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code). Our model consists of

- An **encoder**, simply modelled as a **fully connected layer** from **28*28** into a **hidden layer size**, to be decided.
- A **decoder**, simply modelled as a **fully connected layer** from the **hidden layer size**, back into **28*28**.

A first toy example: MNIST autoencoder with fully connected layers

```python
# Define AutoEncoder Model for MNIST
class MNIST_Autoencoder(nn.Module):

    def __init__(self, hidden_layer = 3):

        # Init from nn.Module
        super().__init__()

        # Encoder part will be a simple FC + ReLU.
        self.encoder = nn.Sequential(nn.Linear(28*28, hidden_layer), nn.ReLU(True))

        # Decoder part will be a simple FC + Tanh
        self.decoder = nn.Sequential(nn.Linear(hidden_layer, 28*28), nn.Tanh())


    def forward(self,x):

        # Forward is encoder into decoder
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

A first toy example: MNIST autoencoder with fully connected layers (Reminder)

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code). Our model consists of

- An **encoder**, simply modelled as a **fully connected layer** from **28*28** into a **hidden layer size**, to be decided.
- A **decoder**, simply modelled as a **fully connected layer** from the **hidden layer size**, back into **28*28**.

The two FC layers were performing “inverse” operations, in terms of dimensionality.

A first toy example: MNIST autoencoder with fully connected layers (Reminder)

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code). Our model consists of

- An **encoder**, simply modelled as a **fully connected layer** from **28*28** into a **hidden layer size**, to be decided.
- A **decoder**, simply modelled as a **fully connected layer** from the **hidden layer size**, back into **28*28**.

The two FC layers were performing “inverse” operations, in terms of dimensionality.

However, we never use FC layers on images, but CNNs instead.

A first toy example: MNIST autoencoder with fully connected layers (Reminder)

Let us start with a simple autoencoder on MNIST (refer to Notebook 1. for demo code). Our model consists of

- An **encoder**, simply modelled as a **fully connected layer** from **28*28** into a **hidden layer size**, to be decided.
- A **decoder**, simply modelled as a **fully connected layer** from the **hidden layer size**, back into **28*28**.

The two FC layers were performing “inverse” operations, in terms of dimensionality.

However, we never use FC layers on images, but CNNs instead.

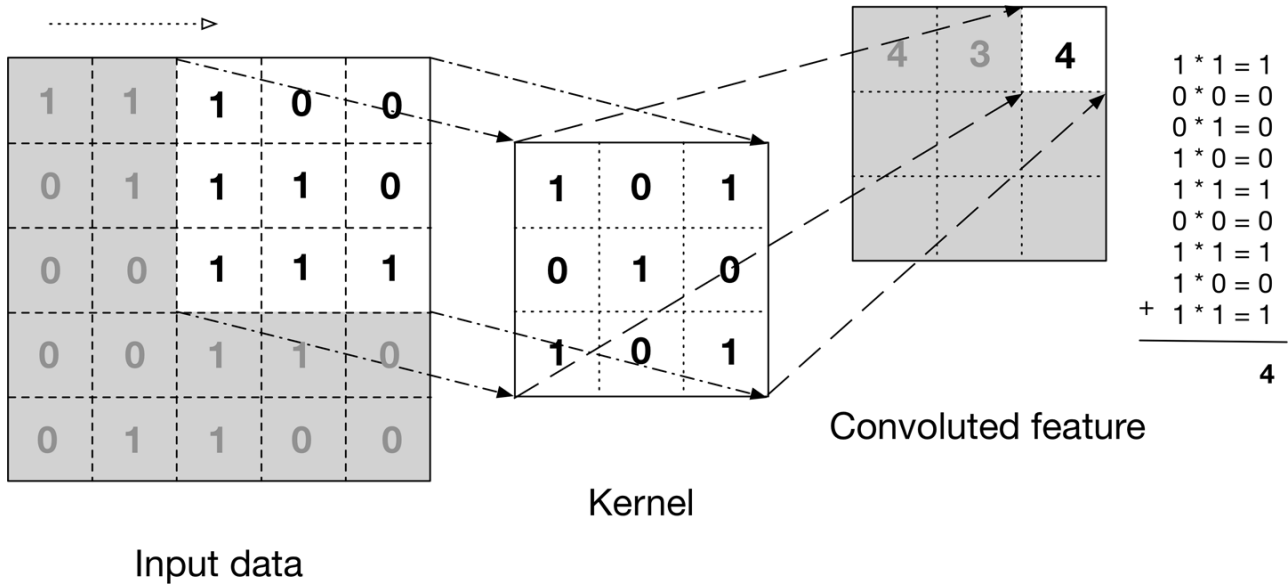
What would then be the “inverse” operation of the convolution?

Convolution layers (reminder)

- Convolution layers performed transformations of input and output by convolutions.
- **Magic formula:**

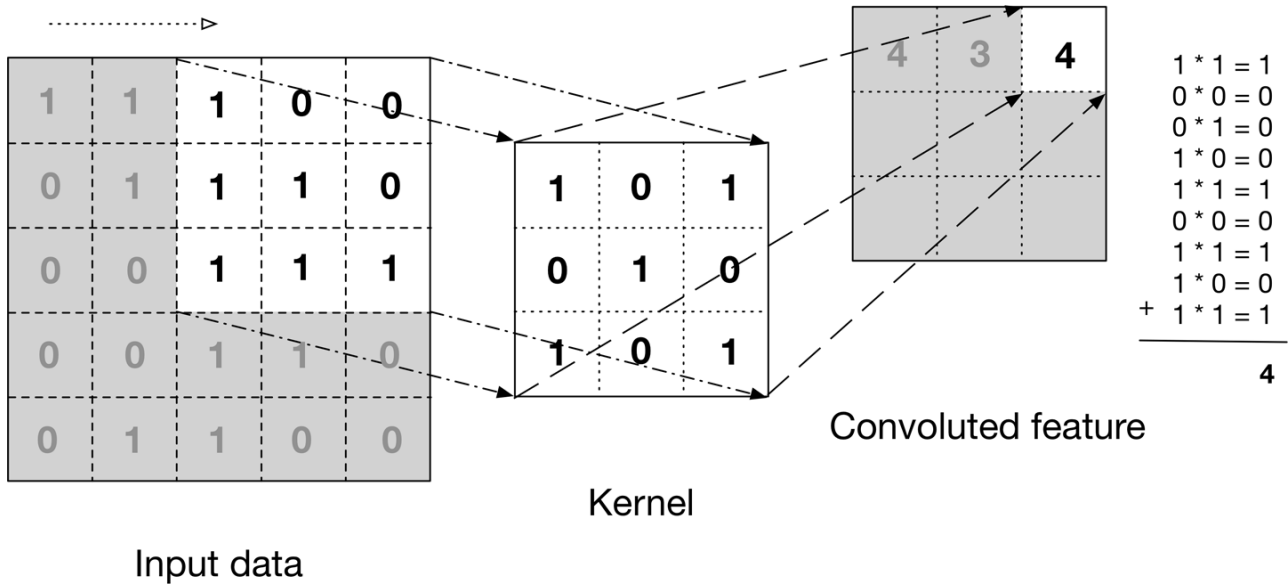
  output_size = (input_size + 2*padding – kernel_size + stride)/stride

- Holds per dimension, i.e. 1D, 2D and N-D convolutions.



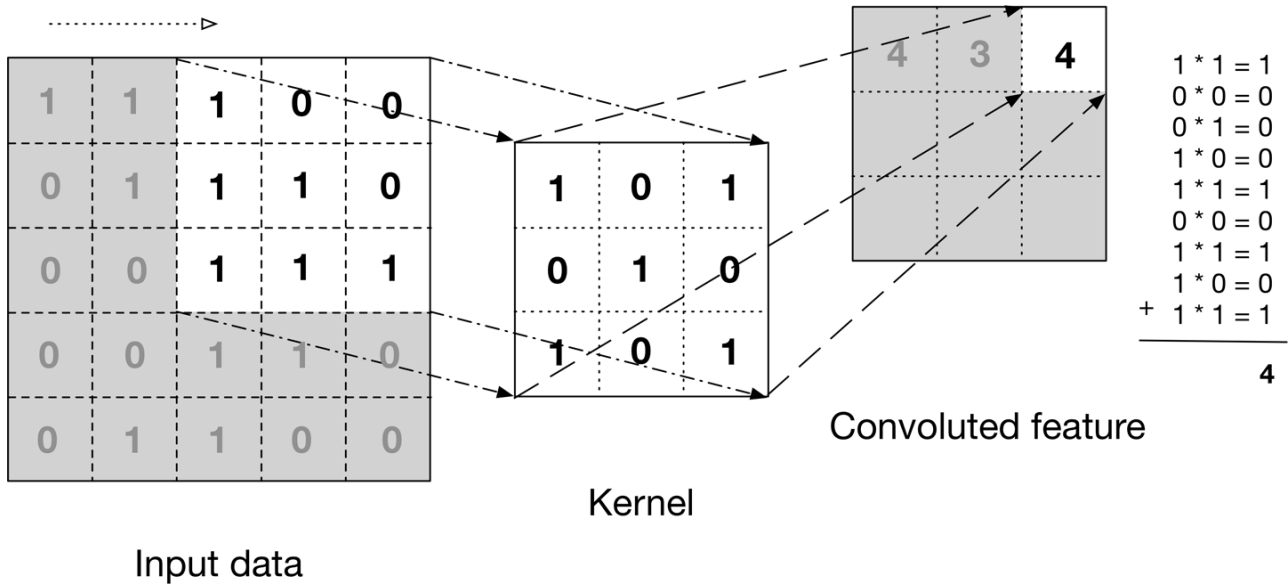
Convolution layers (reminder)

- Convolution layers performed transformations of input and output by convolutions.
- Convolutions were typically used in CNNs to progressively reduce the size of the input images into a fully-connected layer with expected dimension.



Convolution layers (reminder)

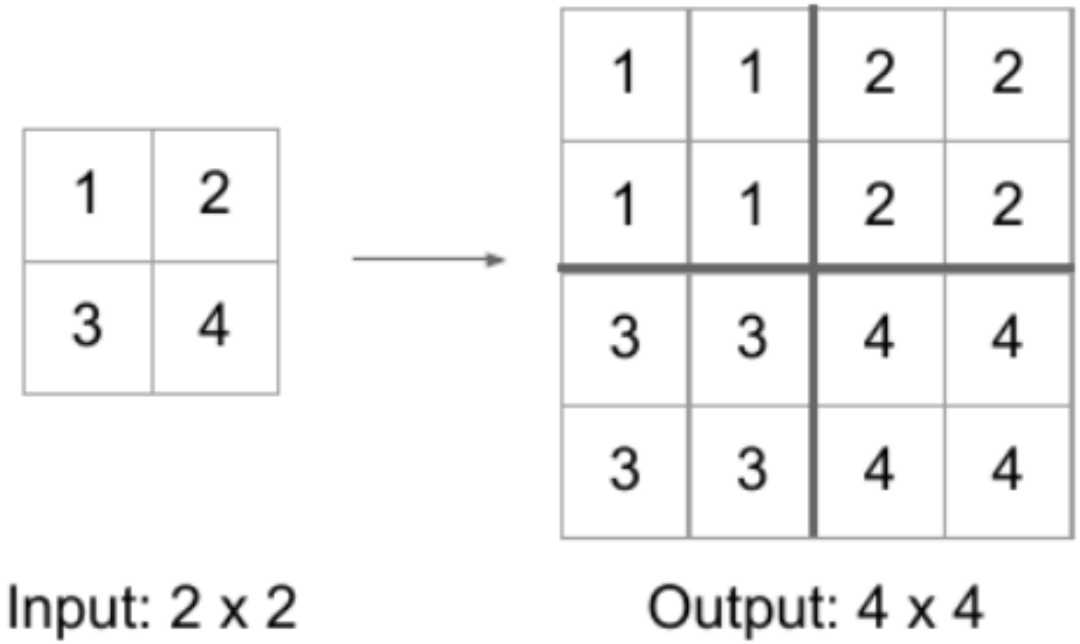
- Convolution layers performed transformations of input and output by convolutions.
- Convolutions were typically used in CNNs to progressively reduce the size of the input images into a fully-connected layer with expected dimension.
- **What would then be the "inverse" operation of convolution layers?**



Option 1: Nearest Neighbors upsampling

Definition (Nearest Neighbors upsampling technique)

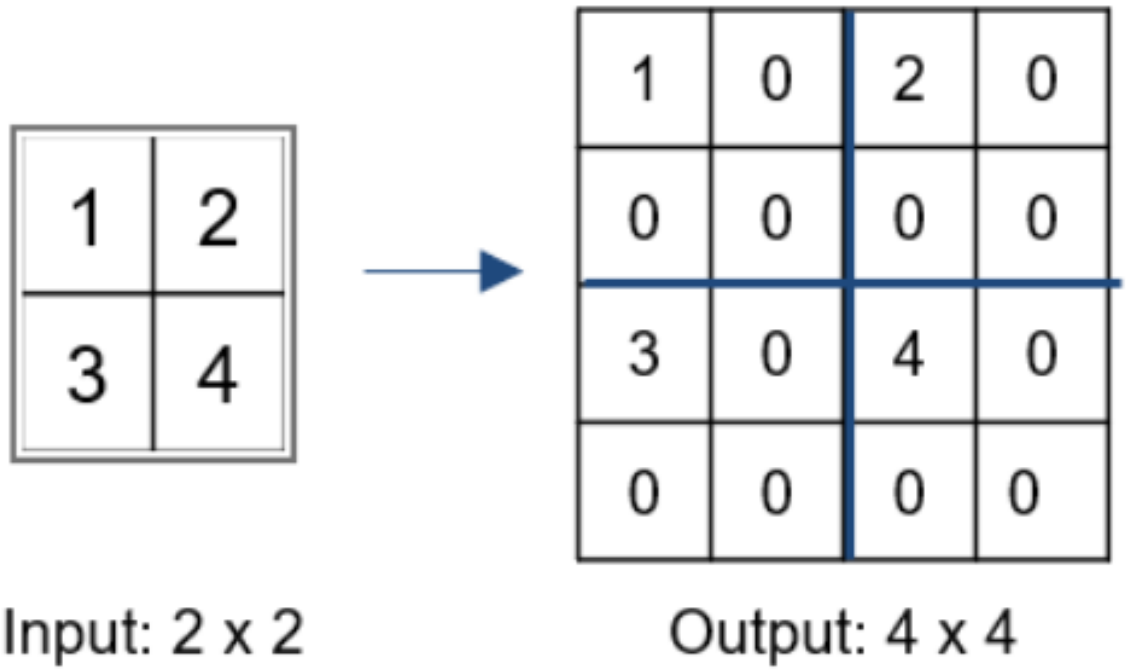
In the **Nearest Neighbors upsampling technique**, we simply take an input pixel value and copy it to the K-Nearest Neighbors where K depends on the expected output.



Option 2: Bed of Nails upsampling

Definition (Bed of Nails upsampling technique)

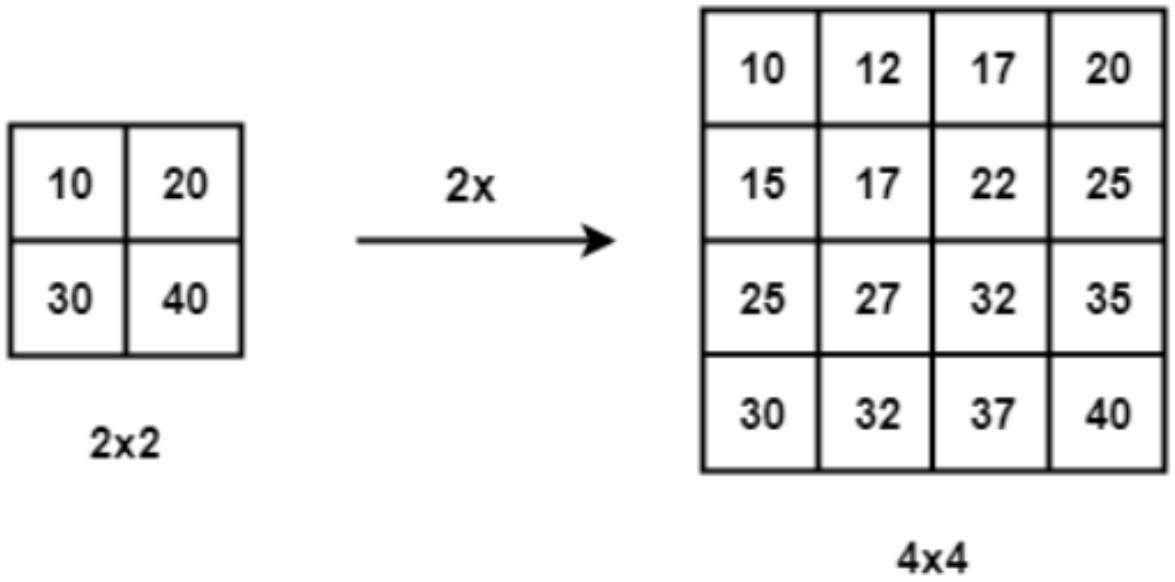
In the **Bed of Nails upsampling technique**, we simply take the value of the input pixel at the corresponding position in the output image and filling zeros in the remaining positions.



Option 3: Bi-Linear Interpolation upsampling

Definition (Bi-Linear Interpolation upsampling technique)

In the **Bi-Linear Interpolation upsampling technique**, we take the 4 nearest pixel value of the input pixel and perform a weighted average based on the distance of the four nearest cells, therefore smoothing the output.



2x2

4x4

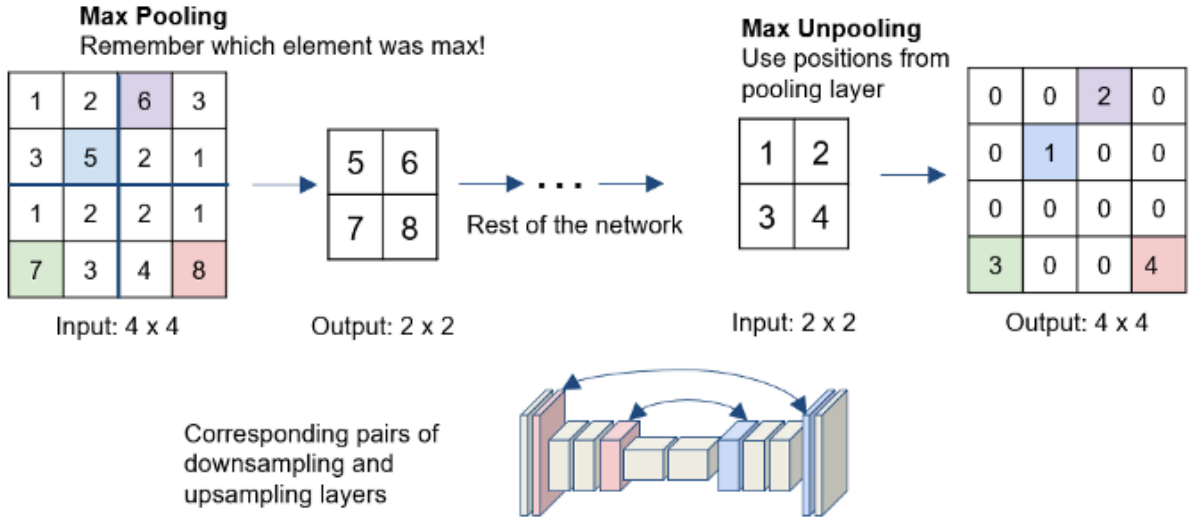
Option 4: Reversed MaxPooling upsampling

Definition (Reversed MaxPooling upsampling technique)

The Max-Pooling layer in CNN takes the maximum among all the values in the kernel.

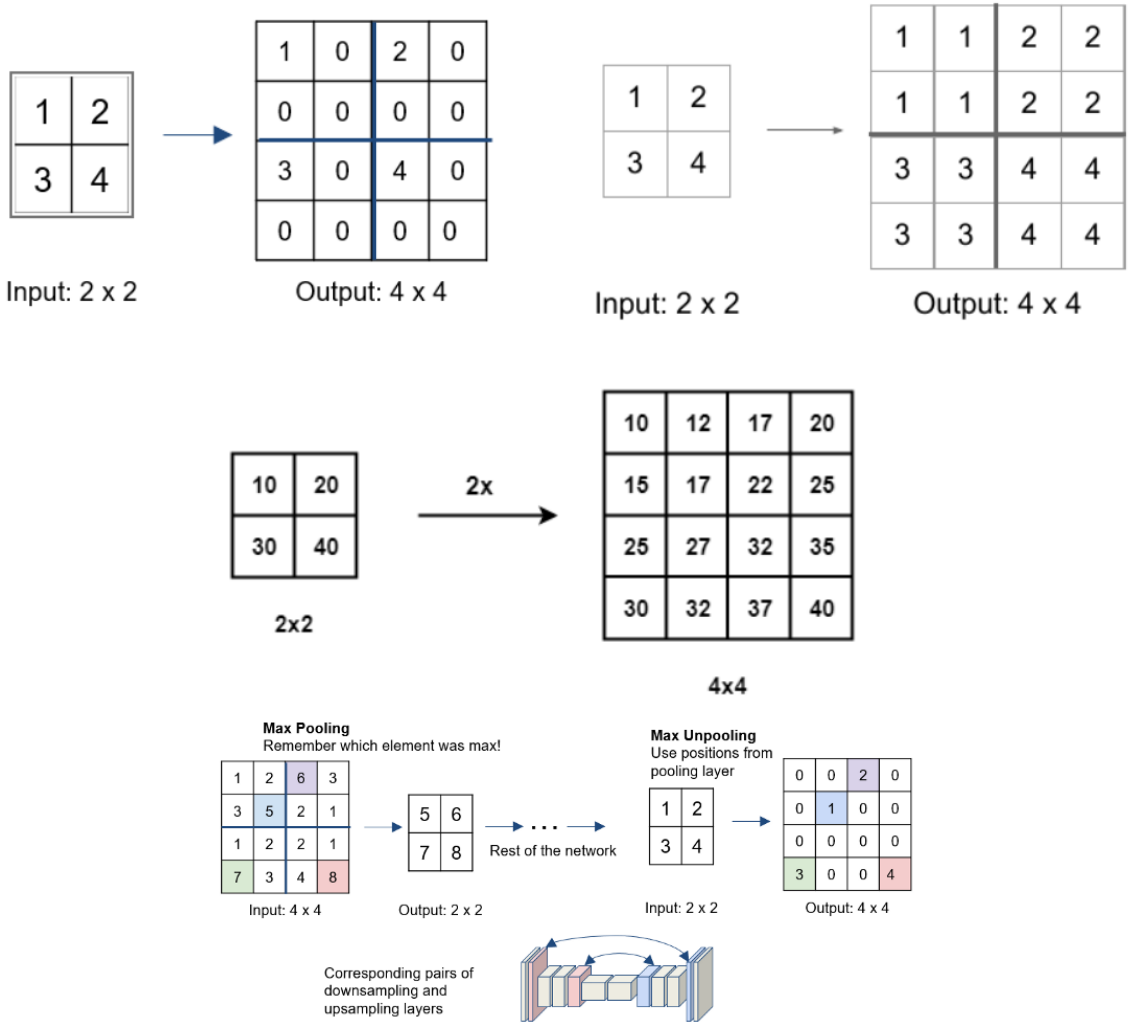
To perform **Reversed MaxPooling** , first, the index of the maximum value is saved for every max-pooling layer during the encoding step.

The saved index is then used during the decoding step where the input pixel is mapped to the saved index, filling zeros everywhere else.



On the need for deconvolution layers

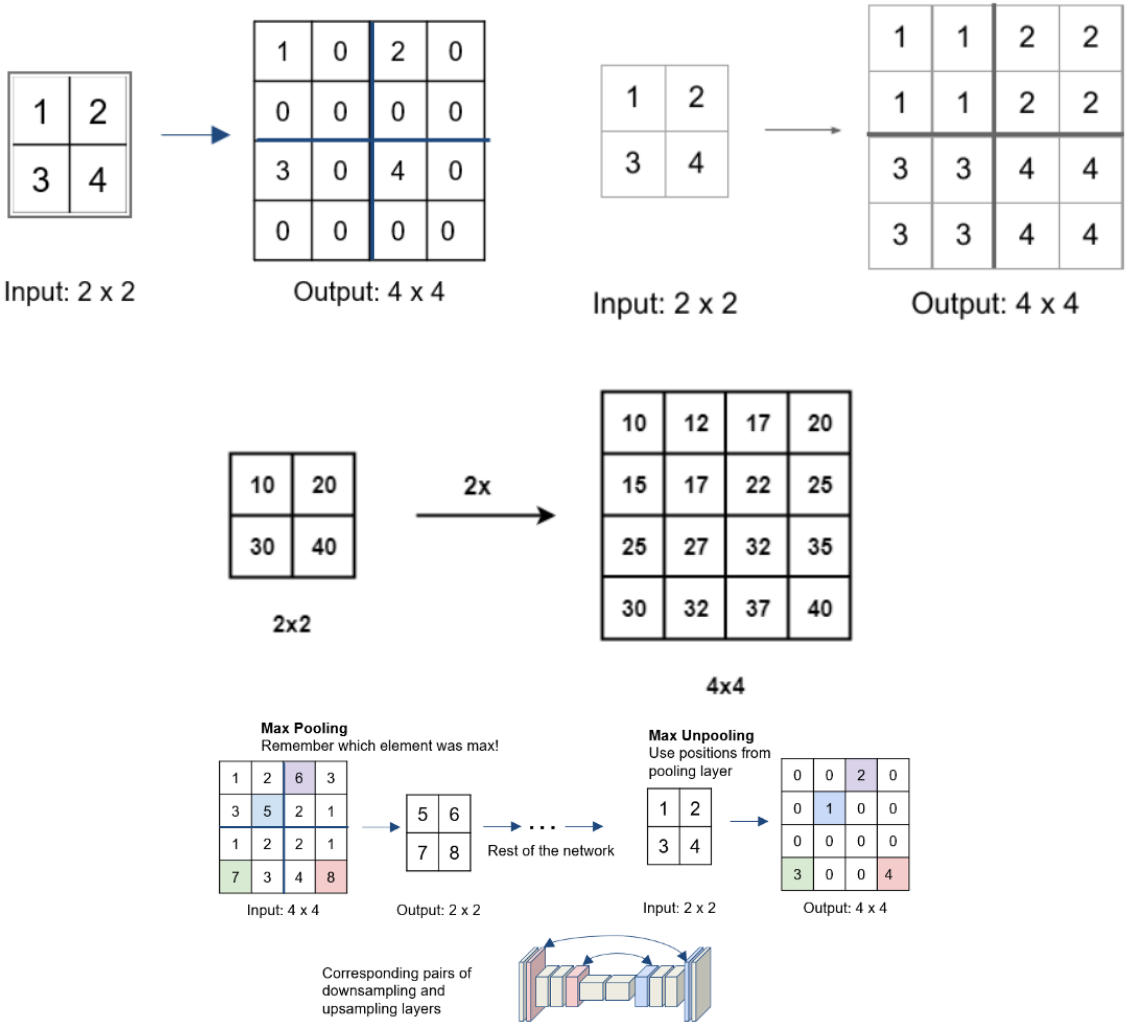
- **Problem #1:** the four options mentioned earlier are a bit too simplistic and might **not be able to "reconstruct" the missing information of an image properly.**



On the need for deconvolution layers

- **Problem #1:** the four options mentioned earlier are a bit too simplistic and might **not be able to "reconstruct" the missing information of an image properly.**

- **Problem #2:** more importantly, these layers are not **trainable and we have no control over them**.

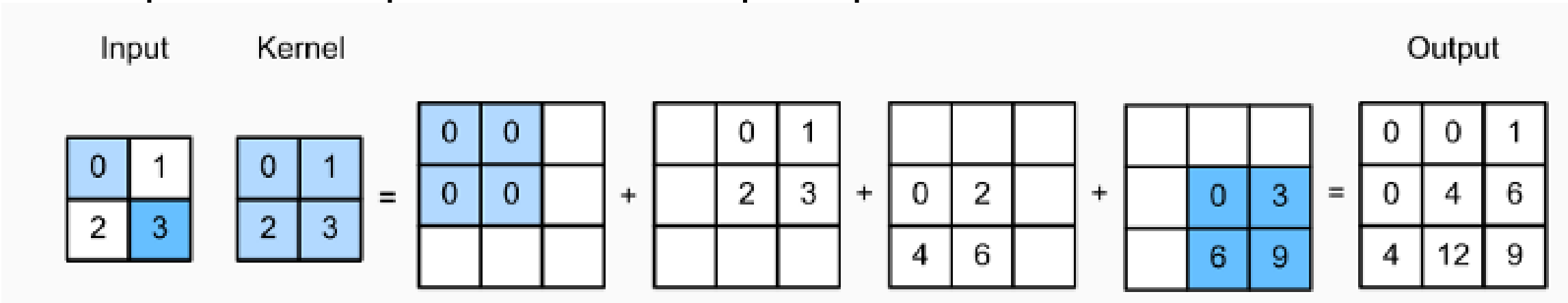


Deconvolution operation

Definition (Deconvolution layer):

The **Deconvolution** (also commonly referred to as **Transposed Convolution**, or **Fractionally Strided Convolution**) **layer** is used to **upsample the input feature map** to a **desired output feature** map using **some learnable parameters**.

Works as a convolution, multiply input with kernel elements but copy output in multiple locations to upsample.



Deconvolution operation

The **Deconvolution layer** consists of a simple matrix multiplication.

$$\begin{bmatrix} 0, 1, 0, 2, 3, 0, 0, 0, 0 \\ 0, 0, 1, 0, 2, 3, 0, 0, 0 \\ 0, 0, 0, 0, 1, 0, 2, 3, 0 \\ 0, 0, 0, 0, 0, 1, 0, 2, 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 5 \\ 0 \cdot 2 + 1 \cdot 3 + 2 \cdot 5 + 3 \cdot 6 \\ 0 \cdot 4 + 1 \cdot 5 + 2 \cdot 7 + 3 \cdot 8 \\ 0 \cdot 5 + 1 \cdot 6 + 2 \cdot 8 + 3 \cdot 9 \end{bmatrix} = \begin{bmatrix} 25 \\ 31 \\ 43 \\ 49 \end{bmatrix}$$

Convolution with 2x2 kernel as a linear mapping

Flattened 3x3 input image

$$\underset{\text{Input}}{\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}} * \underset{\text{Kernel}}{\begin{bmatrix} 0, 1 \\ 2, 3 \end{bmatrix}} = \underset{\text{Output}}{\begin{bmatrix} 25, 31 \\ 43, 49 \end{bmatrix}}$$

Convolution operator

Deconvolution operation

Deconvolution layers are trainable layers, which allow to upsample an input (as opposed to downsampling Convolution layers).

They are easily implemented on PyTorch, using the **ConvTranspose** types of layers (exists in 1D, 2D, and n-D if needed).

```
# A standard 2D Convolution
conv = nn.Conv2d(in_channels = 8, \
                 out_channels = 8, \
                 kernel_size = 5)
```

```
x = torch.randn(2, 8, 64, 64)
print(x.shape)
```

```
torch.Size([2, 8, 64, 64])
```

```
y = conv(x)
print(y.shape)
```

```
torch.Size([2, 8, 60, 60])
```

```
# A deconvolution layer
convt = nn.ConvTranspose2d(in_channels = 8, \
                           out_channels = 8, \
                           kernel_size = 5)
z = convt(y)
print(z.shape)
```

```
torch.Size([2, 8, 64, 64])
```

Deconvolution operation

Deconvolution layers are trainable layers, which allow to upsample an input (as opposed to downsampling Convolution layers).

Important note: Deconvolution is NOT the inverse operation of Convolution, mathematically speaking!

```
1 # Comparing the first line of x and z
2 print(x[0, 0, 0])
3 print(z[0, 0, 0])
```

```
tensor([-1.2188,  0.2657, -1.3329, -0.4168, -0.6341,  1.1183, -0.5112,  0.7523,
         0.6789,  1.0619, -1.1984,  1.2694, -0.4343,  1.3192, -1.7979, -1.0675,
        -1.1264, -0.1616, -0.7532,  0.1439,  0.3421,  0.2679, -0.7575, -0.1454,
        -0.6408,  1.6836,  0.4977, -0.0429,  0.8867, -0.7273,  0.0034, -1.8623,
         0.9508,  0.2095,  2.2252,  1.4001, -1.1226,  0.2137,  0.9925, -0.5685,
         1.2946,  0.0808, -0.3045,  1.0974, -1.4405, -0.4726,  1.5628, -0.4525,
         0.1548, -0.9152,  0.6577, -0.0264, -0.8594,  0.8746,  0.5634, -0.2501,
         0.8192,  0.4786, -0.4876, -0.8358, -1.4187,  0.7590, -0.6741, -0.5103])
tensor([ 0.1368, -0.0531, -0.0926, -0.1074, -0.0481, -0.1166, -0.2710,  0.0608,
         0.2051, -0.2144,  0.0303, -0.0265,  0.2279, -0.0527,  0.0277, -0.0830,
        -0.0043,  0.0161,  0.0875, -0.1546,  0.1444, -0.1395,  0.1548,  0.1265,
        -0.1966, -0.0479,  0.0851, -0.1487, -0.3349,  0.2991, -0.1975,  0.0677,
         0.1445,  0.1438, -0.3004, -0.1749, -0.0758,  0.1199, -0.1008, -0.0129,
        -0.0192,  0.0156, -0.1973, -0.2705, -0.1303,  0.0219, -0.0135, -0.2100,
         0.2152,  0.0296,  0.0122, -0.3307, -0.0893, -0.4357,  0.0294,  0.0351,
         0.0635, -0.0181,  0.0719, -0.0090,  0.1576,  0.0166,  0.0040,  0.0209],
       grad_fn=<SelectBackward>)
```

Deconvolution operation

- Deconvolution also works with additional parameters, such as **stride** and **padding**.
- They might also take an additional parameter, known as **output padding:** while standard padding is applied to the input elements, **output padding** is simply an **additional size added to one side of each dimension in the output shape**.

```
# A deconvolution layer with padding
convt = nn.ConvTranspose2d(in_channels = 16, \
                           out_channels = 8, \
                           kernel_size = 5, \
                           padding = 2)
x = torch.randn(32, 16, 64, 64)
y = convt(x)
print(y.shape)
```

```
torch.Size([32, 8, 64, 64])
```

```
# A deconvolution layer with stride and padding
convt = nn.ConvTranspose2d(in_channels = 16, \
                           out_channels = 8, \
                           kernel_size = 5, \
                           stride = 2, \
                           output_padding = 1, \
                           padding = 2)
x = torch.randn(32, 16, 64, 64)
y = convt(x)
print(y.shape)
```

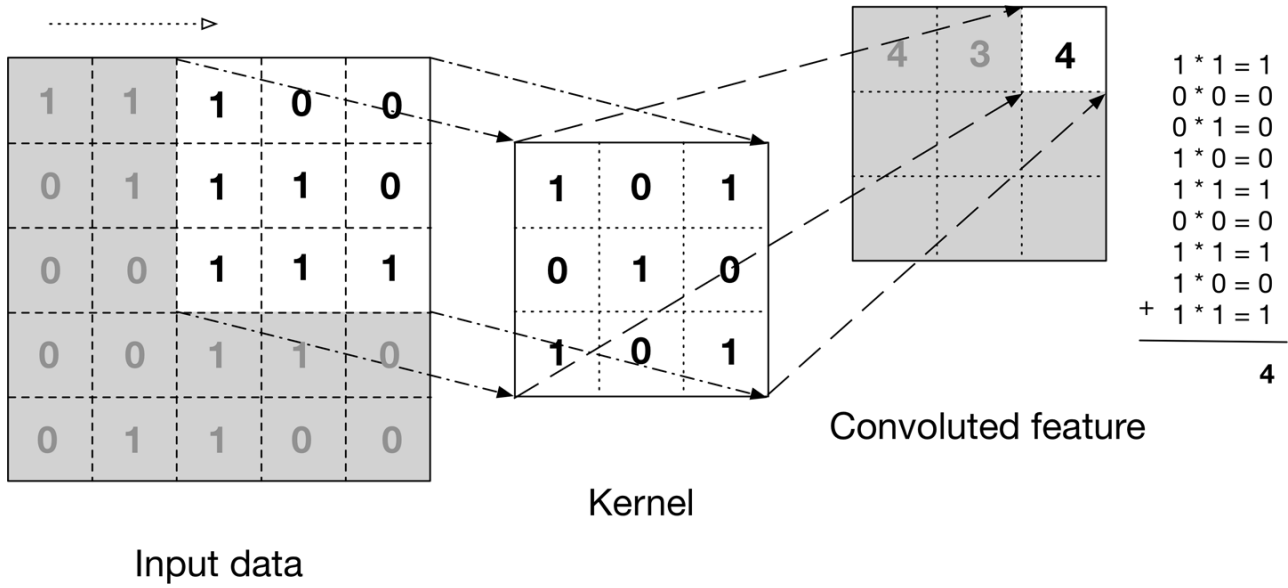
```
torch.Size([32, 8, 128, 128])
```

Convolution layers (reminder)

- Convolution layers performed transformations of input and output by convolutions.
- **Magic formula:**

  output_size = (input_size + 2*padding – kernel_size + stride)/stride

- Holds per dimension, i.e. 1D, 2D and N-D convolutions.



Convolution vs. Deconvolution layers

- Convolution layers performed transformations of input and output by convolutions.
- **Magic formula:**

  output_size = (input_size + 2*padding – kernel_size + stride)/stride

- Holds per dimension, i.e. 1D, 2D and N-D convolutions.

- Deconvolution layers performed transformations of input and output by upsampling convolutions.
- **Magic formula:**

  output_size = (input_size – 1)*stride - 2*padding + kernel_size + **output_padding**

- Holds per dimension, i.e. 1D, 2D and N-D convolutions.

A note on dilation in Deconvolution layers

- Deconvolution layers can also take one additional parameter called dilation.
- **This slightly changes the magic formula into:**

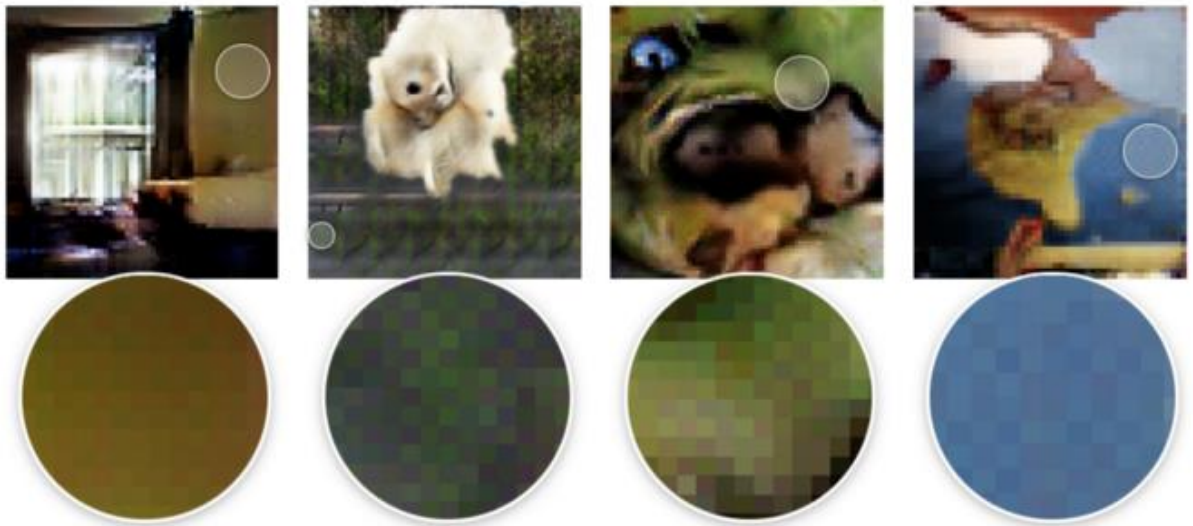
$$\text{output\_size} = (\text{input\_size} - 1)*\text{stride} - 2*\text{padding} + \textbf{output\_padding} + 1 + \textbf{dilation}*(\text{kernel\_size} - 1)$$

- Holds per dimension, i.e. 1D, 2D and N-D convolutions.

Note: **dilation** is hard to explain but easily visualized.
(https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md)

A note on Deconvolution layers

- Transposed convolutions suffer from “chequered” board effects.
- The main cause of this is uneven overlap at some parts of the image causing artifacts.
- This can be fixed, or at least partially reduced, by using a kernel size divisible by the stride, e.g taking a kernel-size of 2x2 or 4x4 when having a stride of 2.



AutoEncoder with Conv and DeConv layers

- Let us put our Deconvolution layers to use, by replacing the FC layers in our original autoencoder, with Convolutions and Deconvolution ones!

AutoEncoder with Conv and DeConv layers

```python
# Define AutoEncoder Model for MNIST
class MNIST_Autoencoder(nn.Module):

    def __init__(self, hidden_layer = 7):

        # Init from nn.Module
        super().__init__()

        # Encoder part will be a simple sequence of Conv2d+ReLU.
        self.encoder = nn.Sequential(nn.Conv2d(1, 16, 3, stride = 2, padding = 1),
                                     nn.ReLU(),
                                     nn.Conv2d(16, 32, 3, stride = 2, padding = 1),
                                     nn.ReLU(),
                                     nn.Conv2d(32, 64, hidden_layer))


        # Decoder part will be a simple sequence of TransposeConv2d+ReLU.
        # Finish with Sigmoid
        self.decoder = nn.Sequential(nn.ConvTranspose2d(64, 32, hidden_layer),
                                     nn.ReLU(),
                                     nn.ConvTranspose2d(32, 16, 3, stride = 2, padding = 1, output_padding = 1),
                                     nn.ReLU(),
                                     nn.ConvTranspose2d(16, 1, 3, stride = 2, padding = 1, output_padding = 1),
                                     nn.ReLU())


    def forward(self,x):

        # Forward is encoder into decoder
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

AutoEncoder with Conv and DeConv layers

- Let us put our Deconvolution layers to use, by replacing the FC layers in our original autoencoder, with Convolutions and Deconvolution ones!
- Deconvolutions in the decoder phase follow the same logic (dimensionality variations) as the convolutions ones in the encoder phase.

```python
# Define AutoEncoder Model for MNIST
class MNIST_Autoencoder(nn.Module):

    def __init__(self, hidden_layer = 7):

        # Init from nn.Module
        super().__init__()

        # Encoder part will be a simple sequence of Conv2d+ReLU.
        self.encoder = nn.Sequential(nn.Conv2d(1, 16, 3, stride = 2, padding = 1),
                                     nn.ReLU(),
                                     nn.Conv2d(16, 32, 3, stride = 2, padding = 1),
                                     nn.ReLU(),
                                     nn.Conv2d(32, 64, hidden_layer))


        # Decoder part will be a simple sequence of TransposeConv2d+ReLU.
        # Finish with Sigmoid
        self.decoder = nn.Sequential(nn.ConvTranspose2d(64, 32, hidden_layer),
                                     nn.ReLU(),
                                     nn.ConvTranspose2d(32, 16, 3, stride = 2, padding = 1, output_padding = 1),
                                     nn.ReLU(),
                                     nn.ConvTranspose2d(16, 1, 3, stride = 2, padding = 1, output_padding = 1),
                                     nn.ReLU())


    def forward(self,x):

        # Forward is encoder into decoder
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

AutoEncoder with Conv and DeConv layers

- Our trainer function does not change, except for the part where we were flattening the input image (no longer necessary, convolutions).

```
# Train
outputs_list = []
loss_list = []
for epoch in range(num_epochs):
    for data in train_loader:

        # Send data to device
        img, _ = data
        img = Variable(img).to(device)

        # Forward pass
        output = model(img)
        loss = distance(output, img)

        # Backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Display
    print('epoch {}/{}, loss {:.4f}'.format(epoch + 1, num_epochs, loss.item()))
    outputs_list.append((epoch, img, output),)
    loss_list.append(loss.item())
```

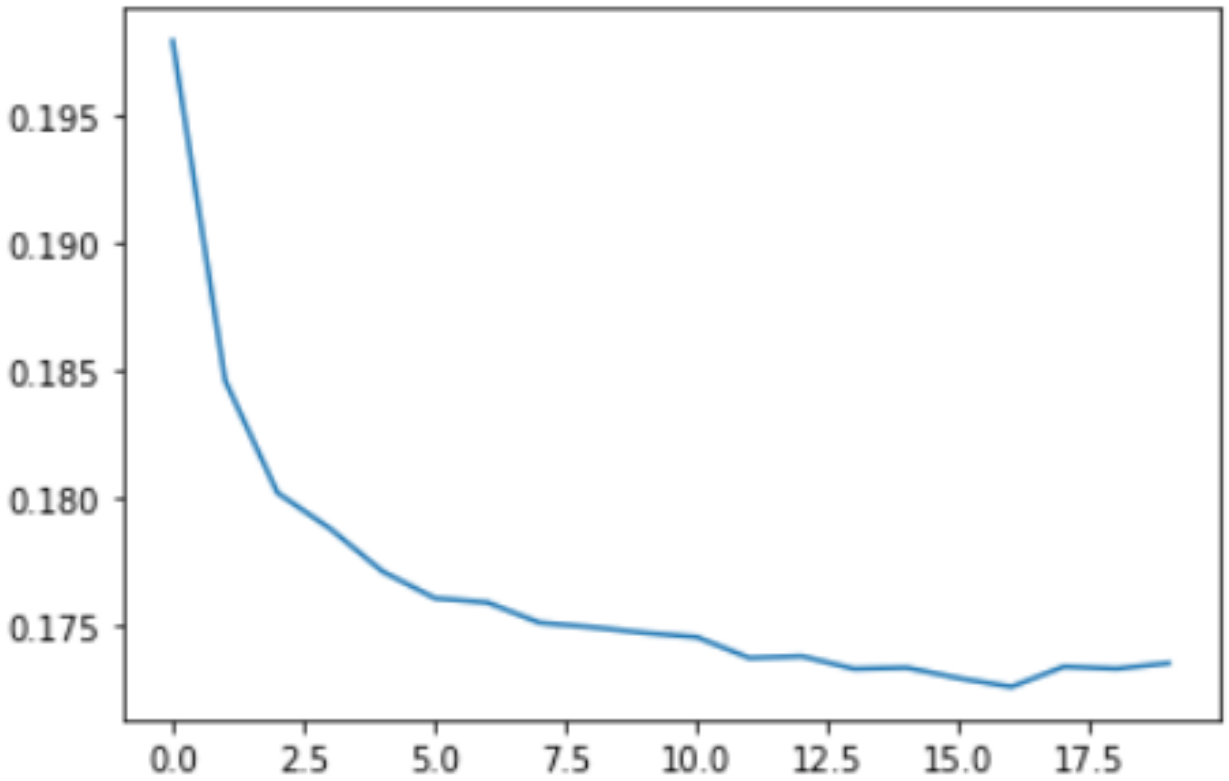
AutoEncoder with Conv and DeConv layers

- Our trainer function does not change, except for the part where we were flattening the input image (no longer necessary, convolutions).

```
# Train
outputs_list = []
loss_list = []
for epoch in range(num_epochs):
    for data in train_loader:

        # Send data to device
        img, _ = data
        img = Variable(img).to(device)

        # Forward pass
        output = model(img)
        loss = distance(output, img)

        # Backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Display
    print('epoch {}/{}, loss {:.4f}'.format(epoch + 1, num_epochs, loss.item()))
    outputs_list.append((epoch, img, output),)
    loss_list.append(loss.item())
```

AutoEncoder with Conv and DeConv layers

```
1  # Display loss
2  plt.figure()
3  plt.plot(loss_list)
4  plt.show()
```

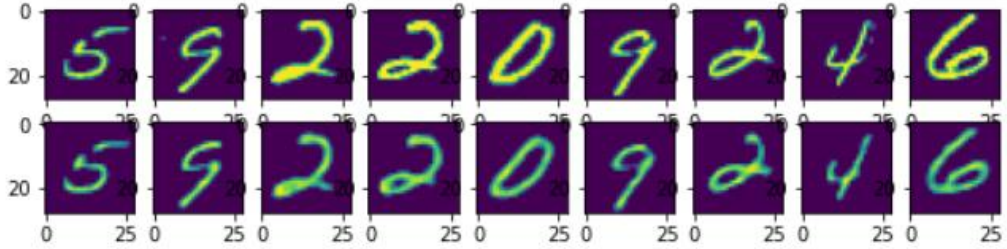


AutoEncoder with Conv and DeConv layers

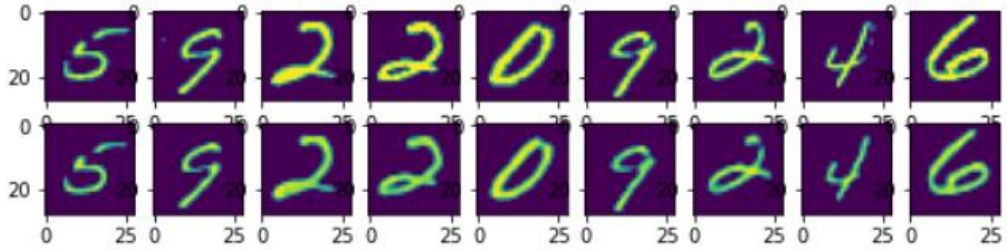
- We can also see the evolution on the outputs being generated during the different iterations of training.

```
# Display some outputs
for k in range(0, num_epochs, 4):
    plt.figure(figsize = (9, 2))
    imgs = outputs_list[k][1].cpu().detach().numpy()
    recon = outputs_list[k][2].cpu().detach().numpy()
    for i, item in enumerate(imgs):
        if i >= 9: break
        plt.subplot(2, 9, i+1)
        plt.imshow(item[0])

    for i, item in enumerate(recon):
        if i >= 9: break
        plt.subplot(2, 9, 9+i+1)
        plt.imshow(item[0])
```

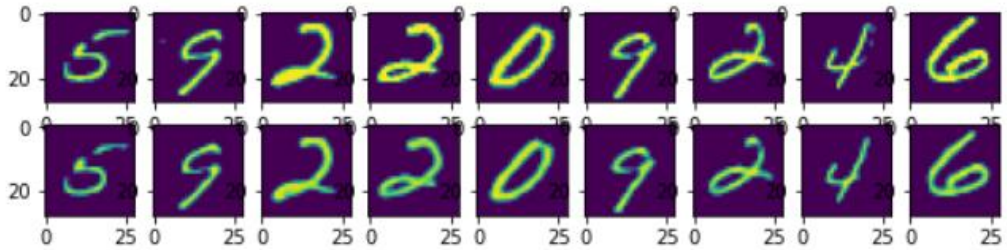
Iteration 5



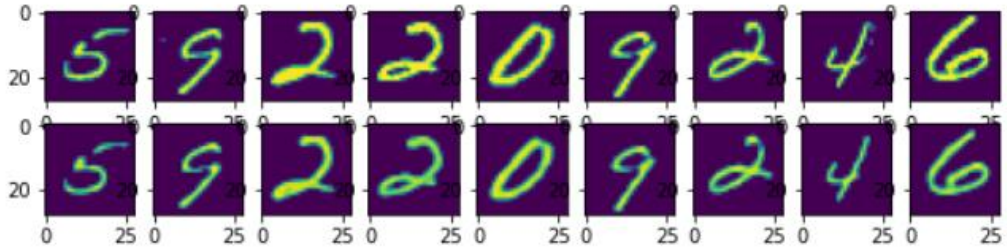
Iteration 10



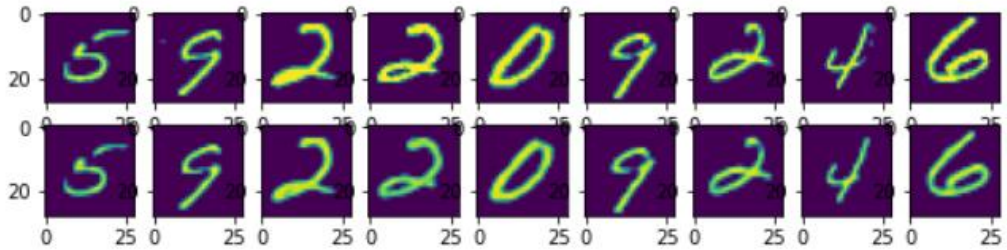
Iteration 15



Iteration 20

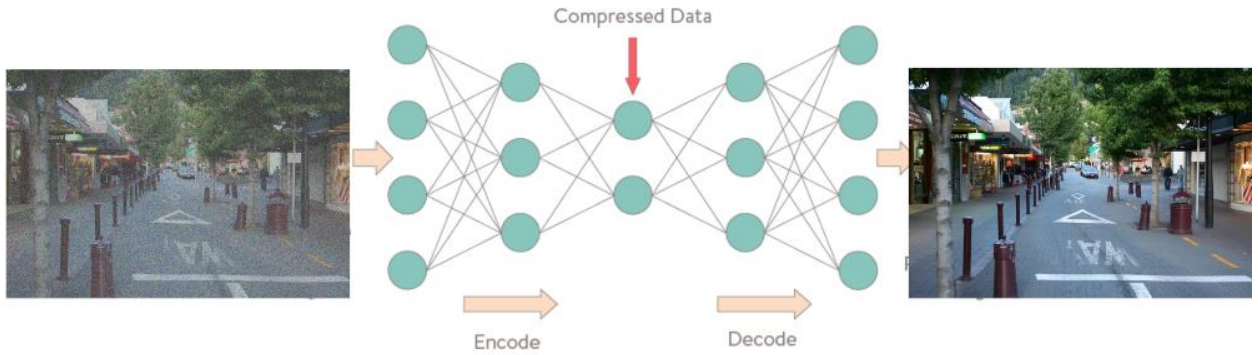


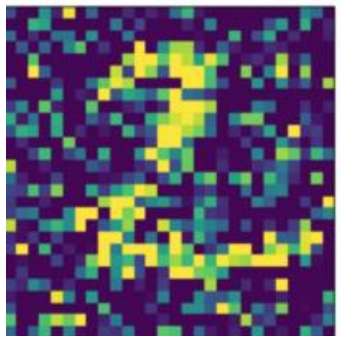
Iteration 25



AutoEncoders for Signal Denoising

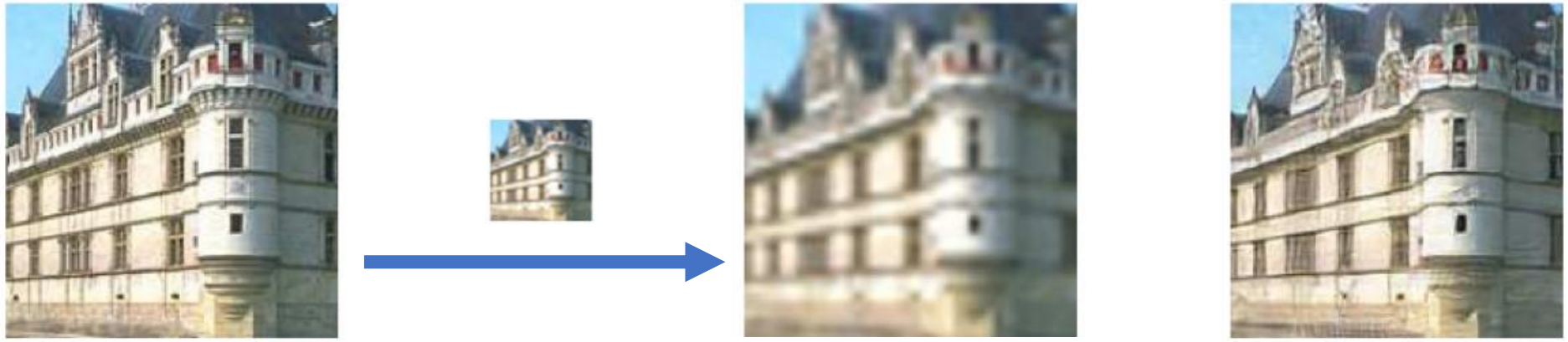
- On top of their ability to encode an input into a low-dimension latent vectors, the AutoEncoders can be used for…
- **Denoising images:** the autoencoder will simply focus on keeping the most important features of the image and will likely drop the noise present in the image.
- **This can be used to protect against Noising Attacks!**





AutoEncoders for Supersizing Images

- On top of their ability to encode an input into a low-dimension latent vectors, the AutoEncoders can be used for…
- **Supersizing images:** the autoencoder will simply attempt to reconstruct the missing pixels in the image to increase its resolution.



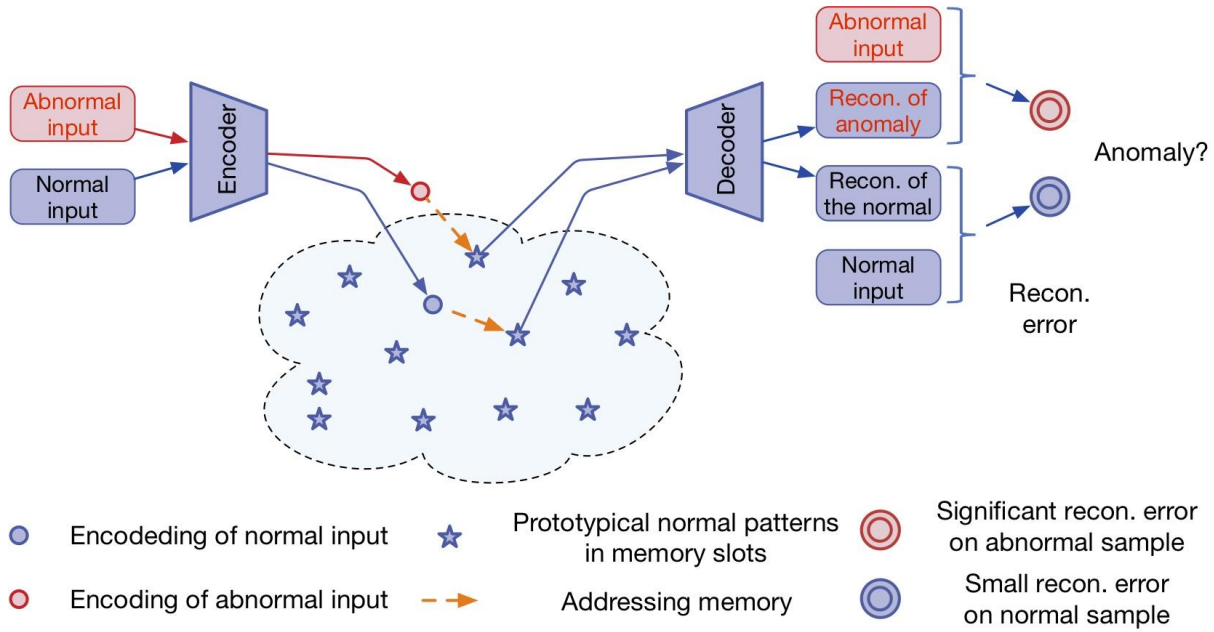
Ground Truth

Downsample, and then pixelate with Nearest Neighbors

Output (super resolution)

AutoEncoders for Anomaly Detection

- On top of their ability to encode an input into a low-dimension latent vectors, the AutoEncoders can be used for…
- **Anomaly detection:** There are many ways and techniques to detect anomalies and outliers. However, if you have correlated input data, the autoencoder method will work very well because the encoding operation relies on the correlated features to compress the data.



Conclusion

In this lecture

- Autoencoders definition
- Autoencoders with fully connected layers
- Fractionally strided convolution layers
- Autoencoders with convolutions and deconvolutions
- Autoencoder uses

In the next lectures

- Variational Autoencoder and noisy latent representations
- Generative Adversarial Networks
- More advanced GANs

Learn more about these topics

Out of class, for those of you who are curious

- None for now!