Deep learning

7.1. Transposed convolutions

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Constructing deep generative architectures requires layers to increase the signal dimension, the contrary of what we have done so far with feed-forward networks.

Some generative processes optimize the input, and as such rely on back-propagation to expend the signal from a low-dimension representation to the high-dimension signal space (e.g. lecture 9.4. "Optimizing inputs")

The same can be done in the forward pass with **transposed convolution layers** whose forward operation corresponds to a convolution layer's backward pass.

Consider a 1d convolution with a kernel κ

$$\begin{aligned} y_i &= (x \circledast \kappa)_i \\ &= \sum_a x_{i+a-1}\, \kappa_a \\ &= \sum_u x_u\, \kappa_{u-i+1}. \end{aligned}$$

We get

$$\begin{aligned} \left[\frac{\partial \ell}{\partial x}\right]_u &= \frac{\partial \ell}{\partial x_u} \\ &= \sum_i \frac{\partial \ell}{\partial y_i} \frac{\partial y_i}{\partial x_u} \\ &= \sum_i \frac{\partial \ell}{\partial y_i} \kappa_{u-i+1}. \end{aligned}$$

which looks a lot like a standard convolution layer, except that the kernel coefficients are visited in reverse order.

This is actually the standard convolution operator from signal processing. If $*$ denotes this operation, we have

$$(x * \kappa)_i = \sum_a x_a \, \kappa_{i-a+1}.$$

Coming back to the backward pass of the convolution layer, if

$$y = x \circledast \kappa$$

then

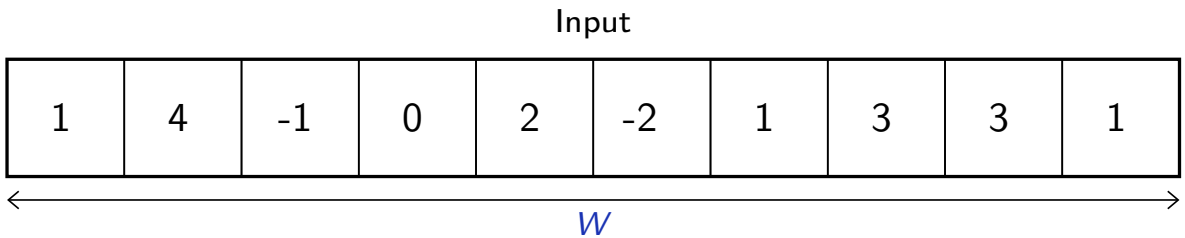
$$\left[\frac{\partial \ell}{\partial x}\right] = \left[\frac{\partial \ell}{\partial y}\right] * \kappa.$$

In the deep-learning field, since it corresponds to transposing the weight matrix of the equivalent fully-connected layer, it is called a **transposed convolution**.

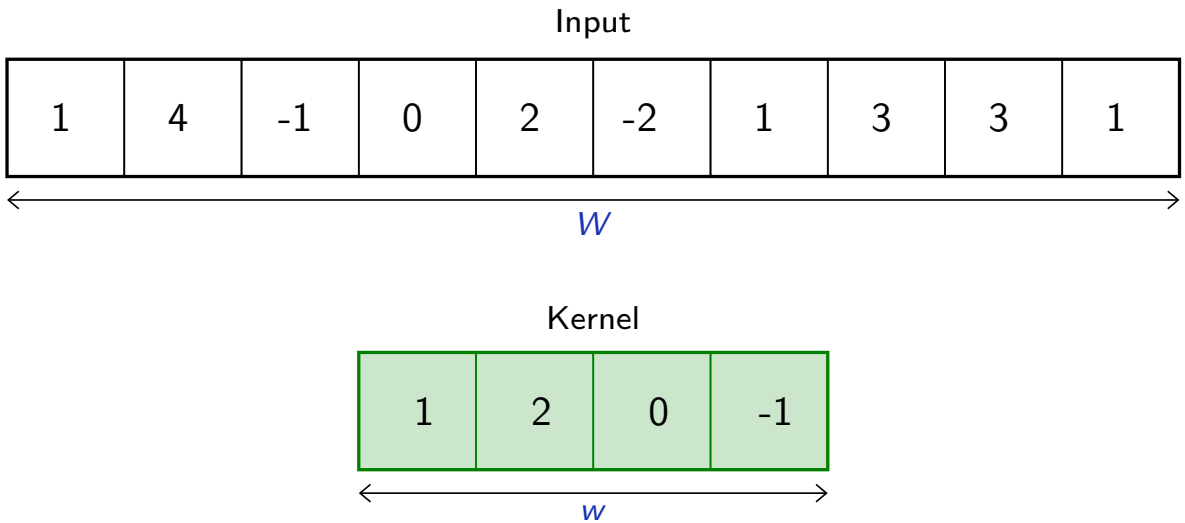
$$\begin{pmatrix} \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\ 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\ 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\ 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\ 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \end{pmatrix}^\top = \begin{pmatrix} \kappa_1 & 0 & 0 & 0 & 0 \\ \kappa_2 & \kappa_1 & 0 & 0 & 0 \\ \kappa_3 & \kappa_2 & \kappa_1 & 0 & 0 \\ 0 & \kappa_3 & \kappa_2 & \kappa_1 & 0 \\ 0 & 0 & \kappa_3 & \kappa_2 & \kappa_1 \\ 0 & 0 & 0 & \kappa_3 & \kappa_2 \\ 0 & 0 & 0 & 0 & \kappa_3 \end{pmatrix}$$

A convolution can be seen as a series of inner products, a transposed convolution can be seen as a weighted sum of translated kernels.

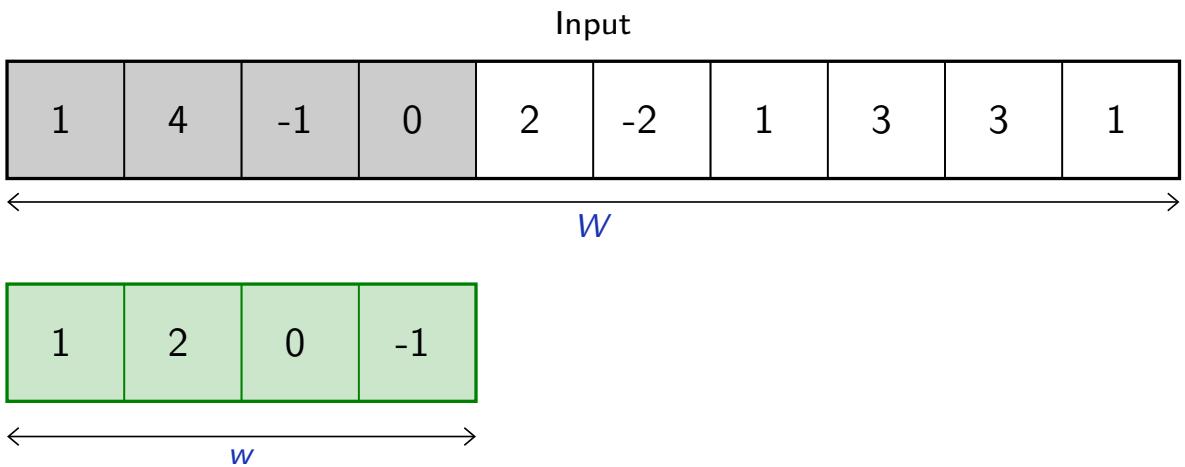
Convolution layer



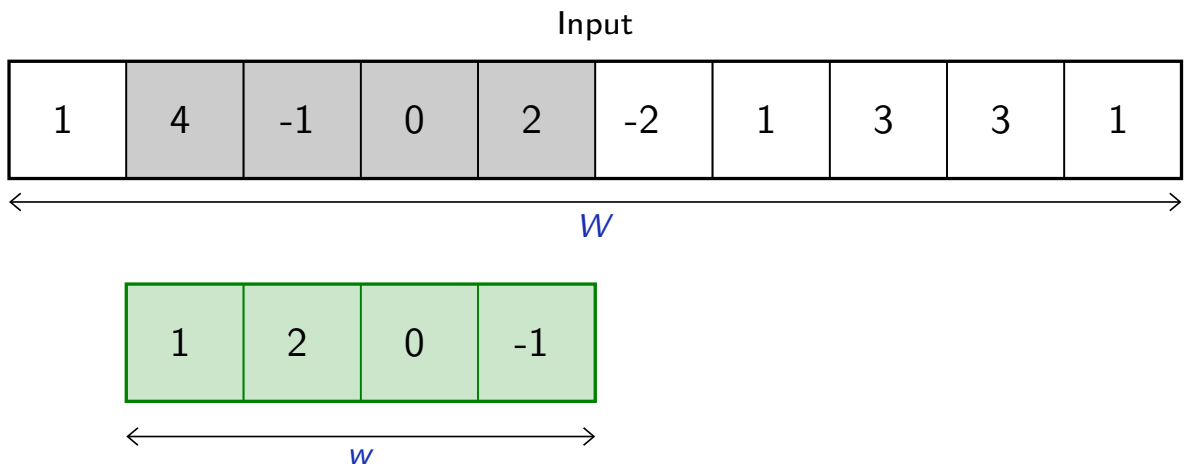
Convolution layer



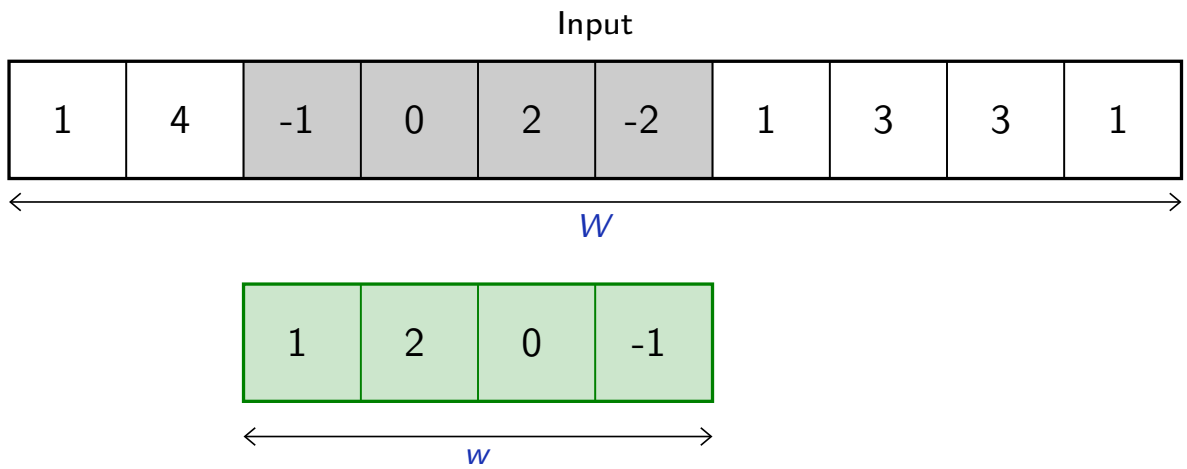
Convolution layer



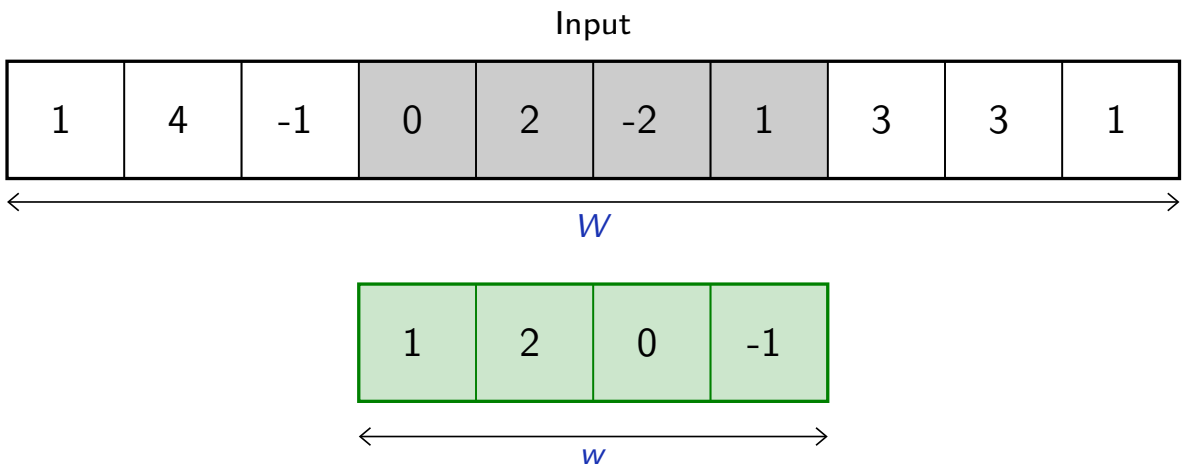
Convolution layer



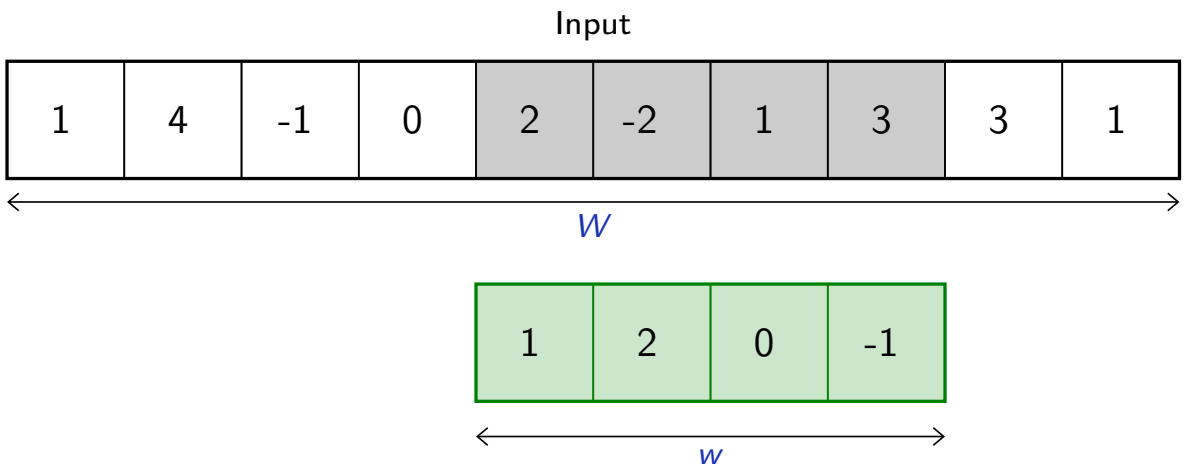
Convolution layer



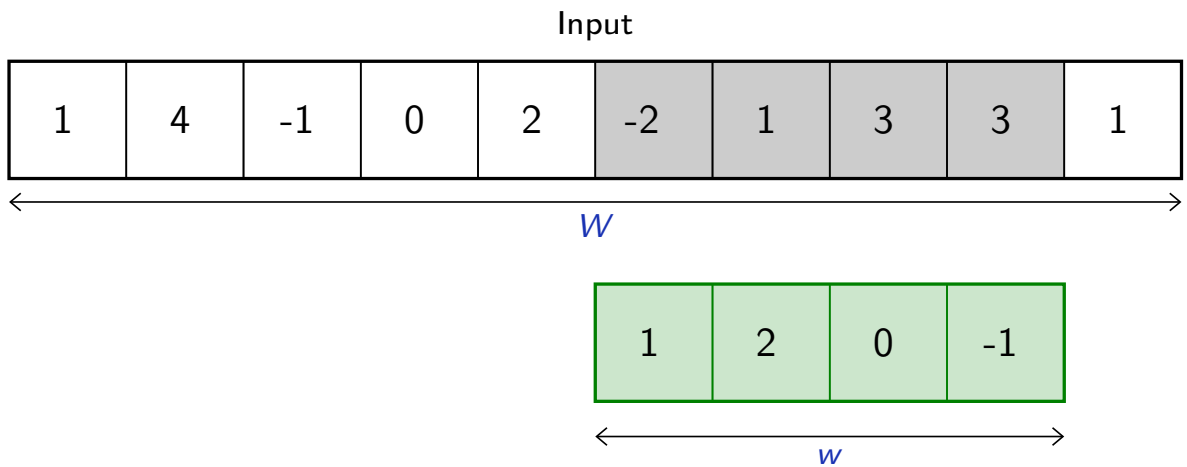
Convolution layer



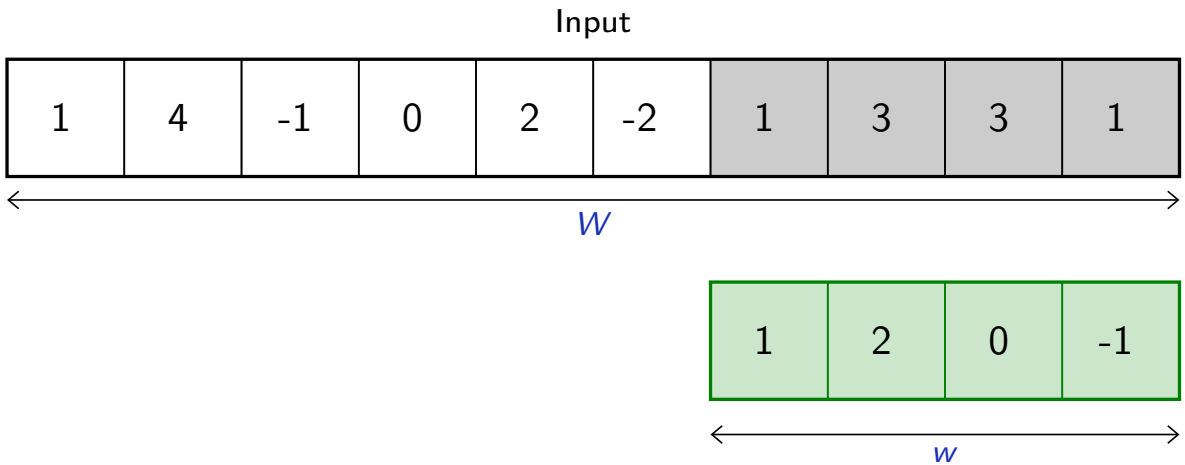
Convolution layer



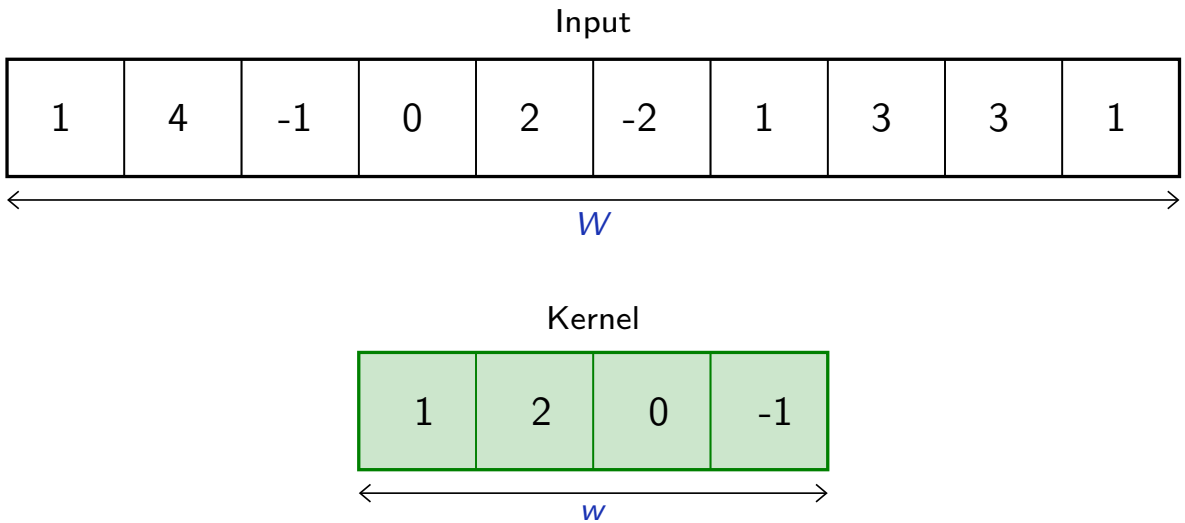
Convolution layer



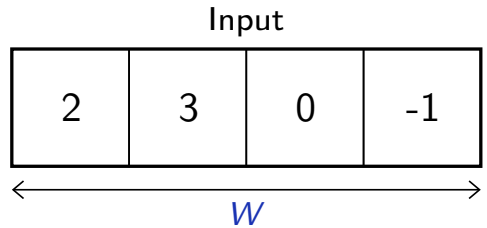
Convolution layer

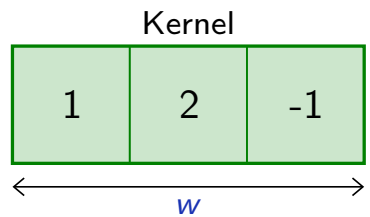


Convolution layer

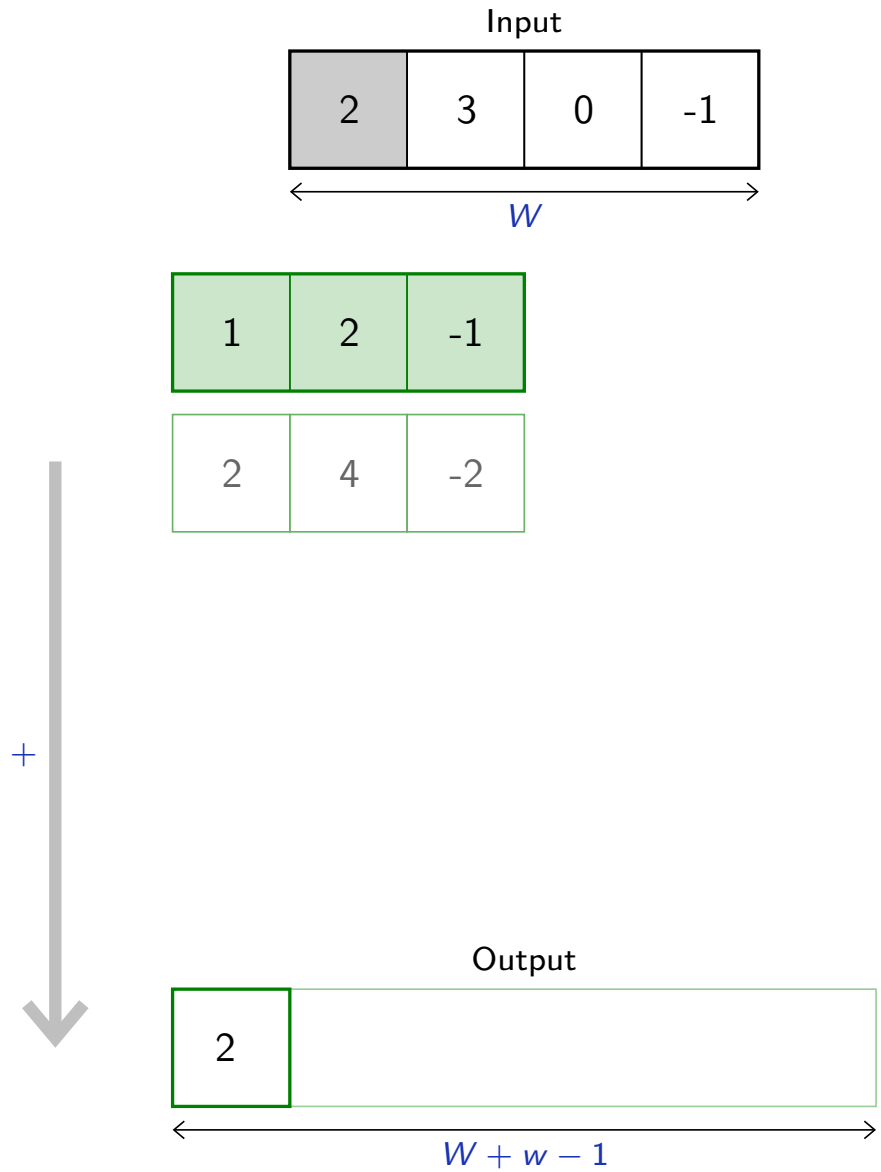


Transposed convolution layer

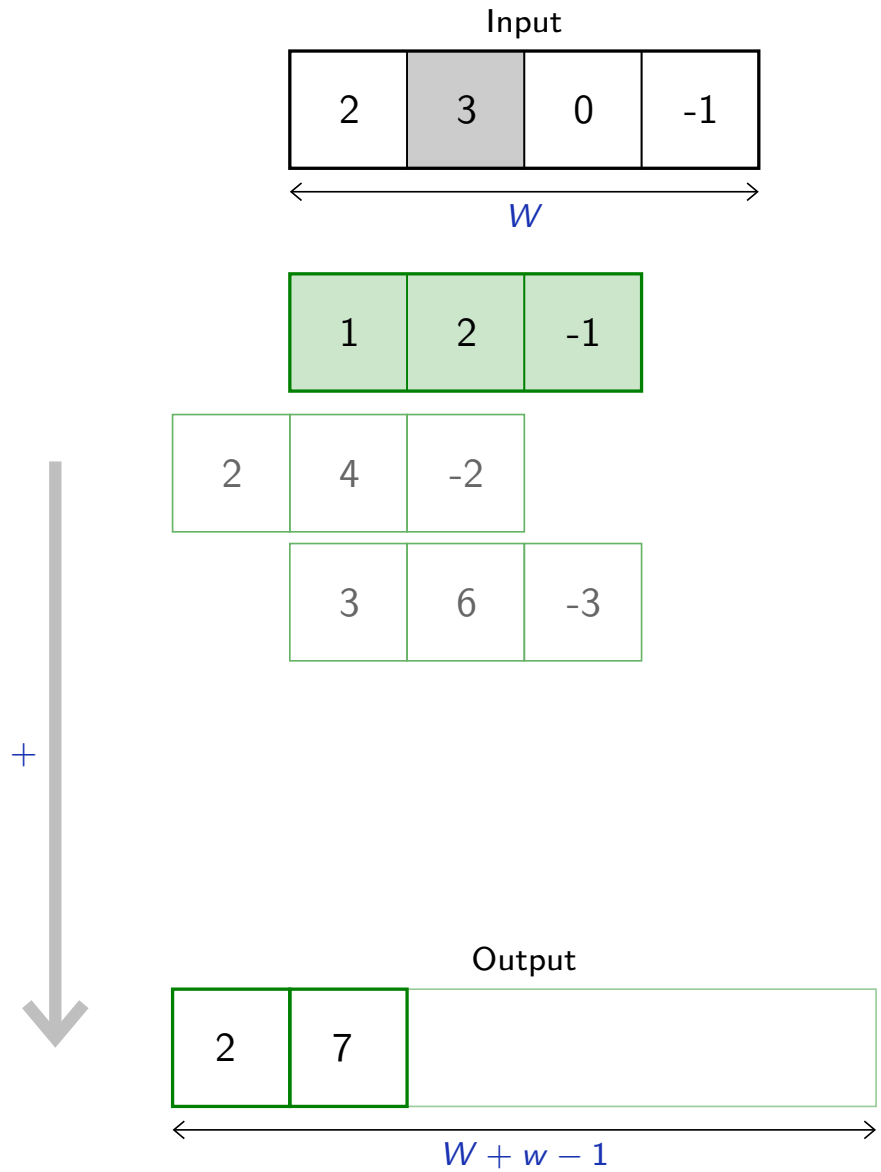




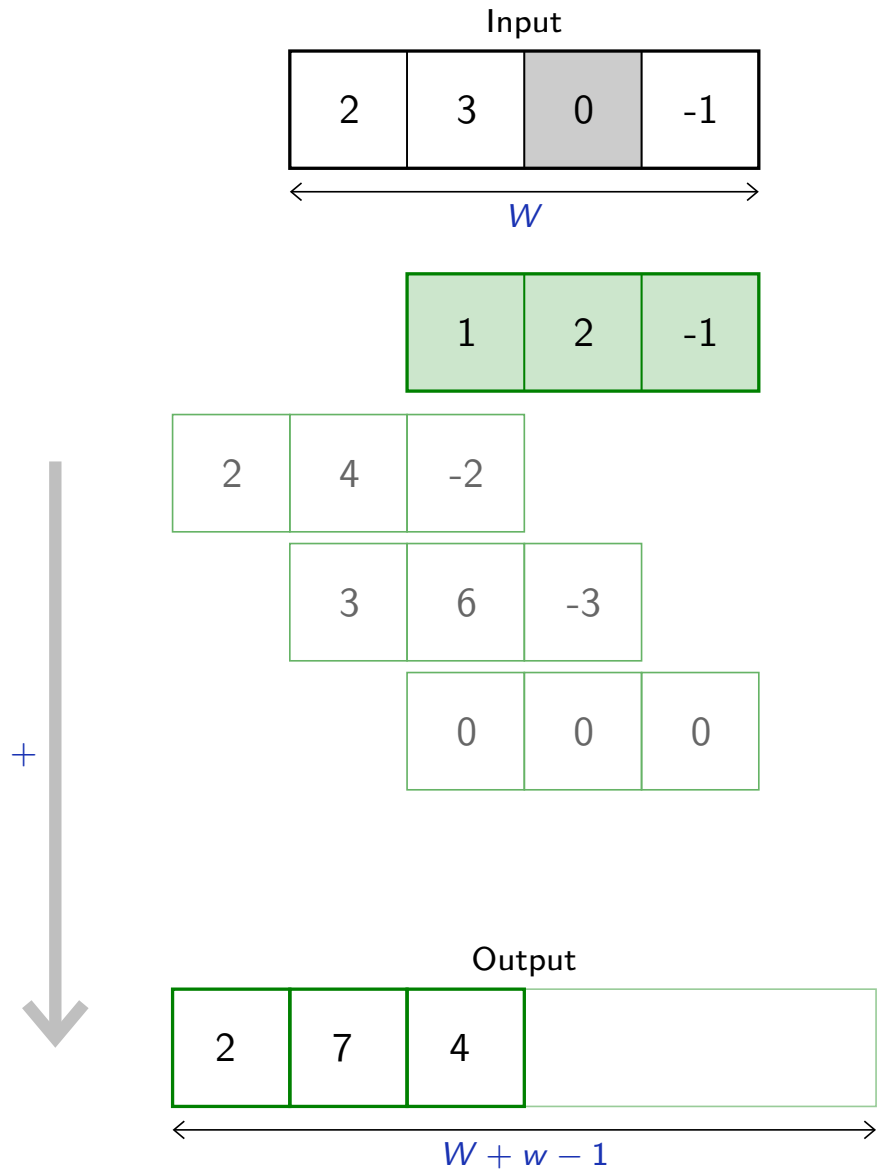
Transposed convolution layer



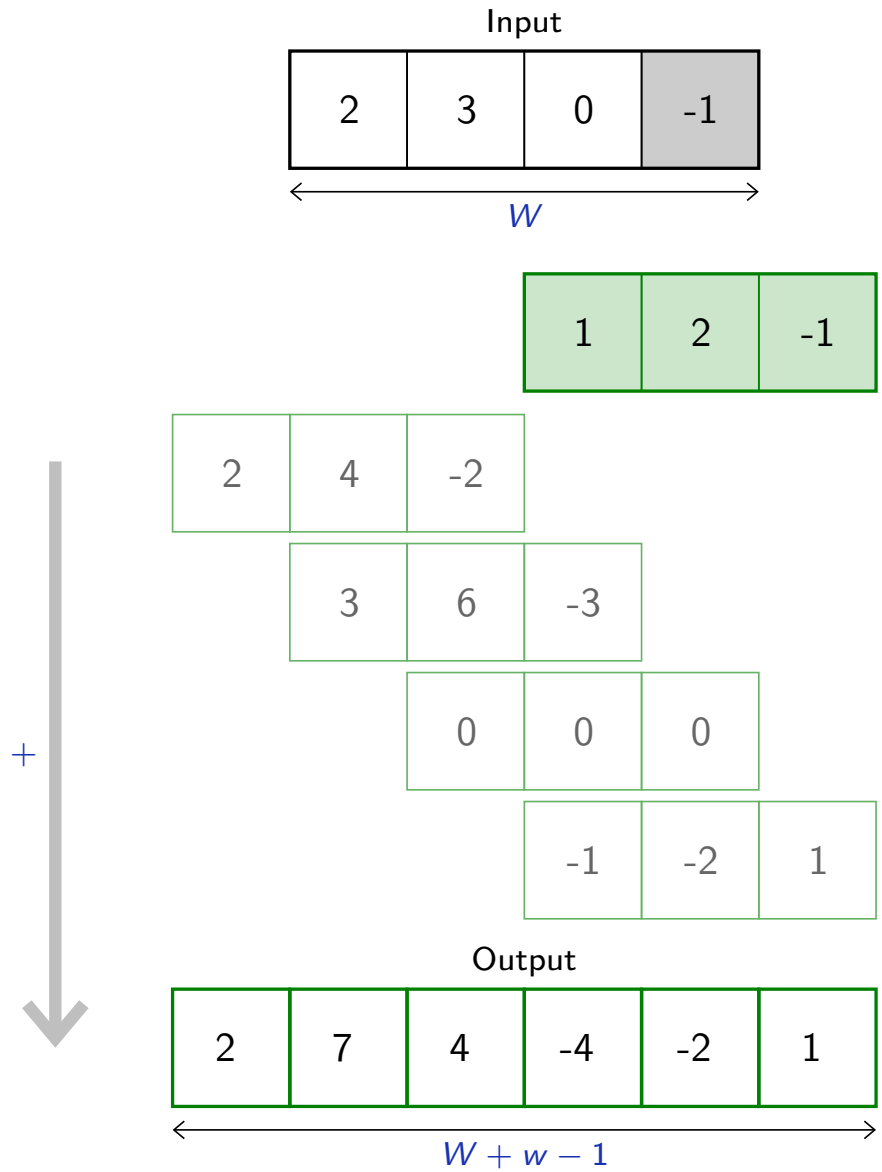
Transposed convolution layer



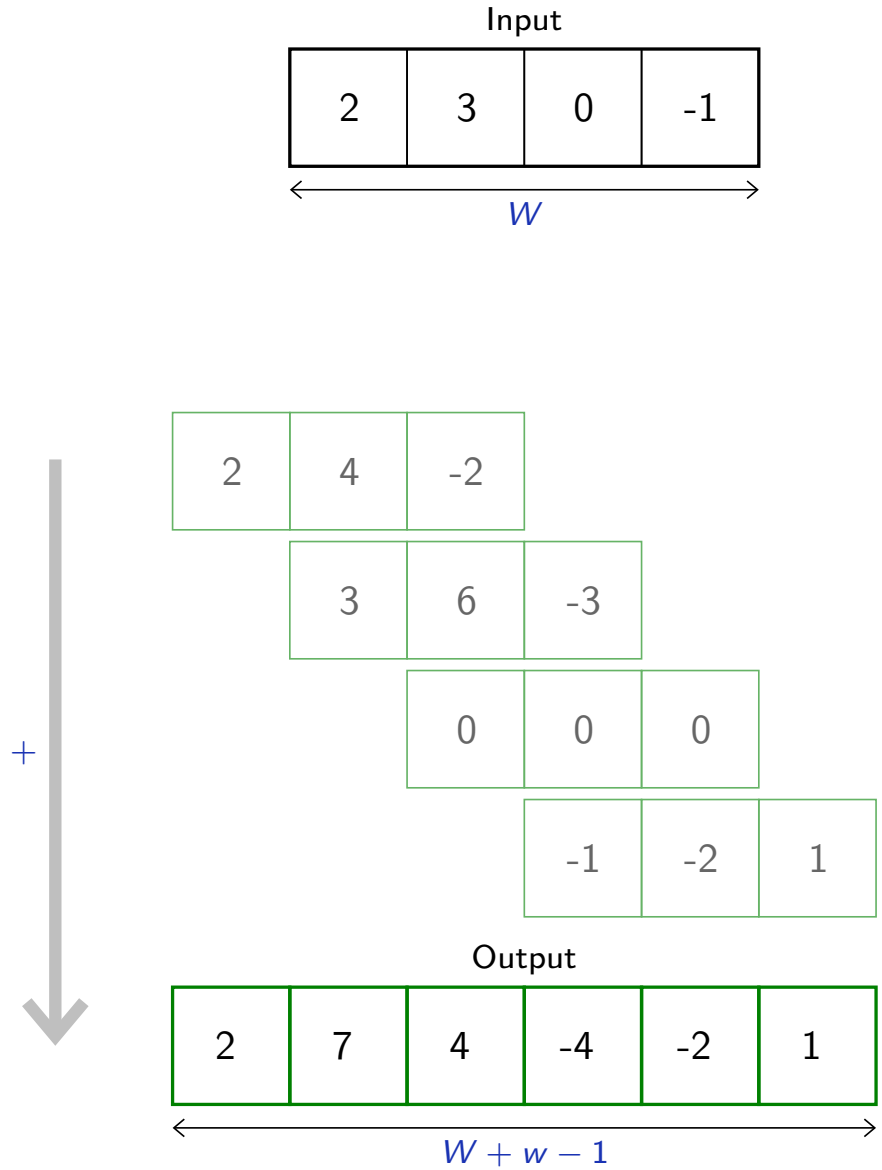
Transposed convolution layer



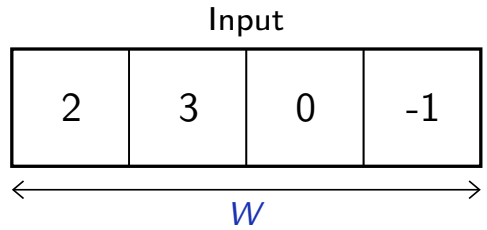
Transposed convolution layer

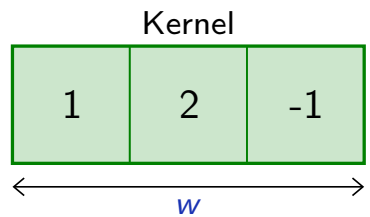


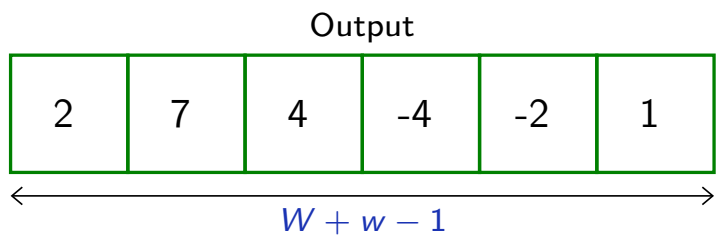
Transposed convolution layer



Transposed convolution layer



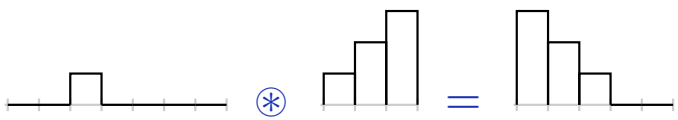




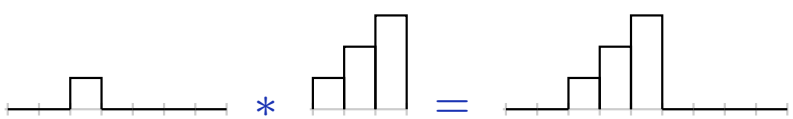
`F.conv_transpose1d` implements the operation we just described. It takes as input a batch of multi-channel samples, and produces a batch of multi-channel samples.

We can compare on a simple 1d example the results of a standard and a transposed convolution:

```
>>> x = torch.tensor([[[0., 0., 1., 0., 0., 0., 0.]]])
>>> k = torch.tensor([[[1., 2., 3.]]])
>>> F.conv1d(x, k)
tensor([[[ 3.,  2.,  1.,  0.,  0.]]])
```



```
>>> F.conv_transpose1d(x, k)
tensor([[[ 0.,  0.,  1.,  2.,  3.,  0.,  0.,  0.,  0.]]])
```



The class `nn.ConvTranspose1d` embeds that operation into a `nn.Module`.

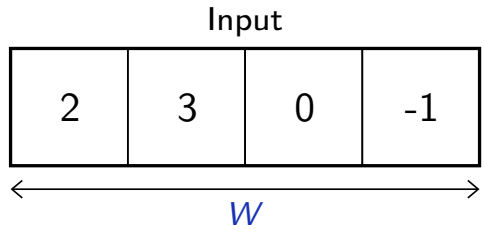
```
>>> x = torch.tensor([[[ 1., 0., 0., 0., -1.]]])
>>> m = nn.ConvTranspose1d(1, 1, kernel_size=3)
>>> with torch.autograd.no_grad():
...   m.bias.zero_()
...   m.weight.copy_(torch.tensor([ 1, 2, 1 ]))
...
Parameter containing:
tensor([0.], requires_grad=True)
Parameter containing:
tensor([[[1., 2., 1.]]], requires_grad=True)
>>> y = m(x)
>>> y
tensor([[[ 1.,  2.,  1.,  0., -1., -2., -1.]]], grad_fn=<SqueezeBackward1>)
```

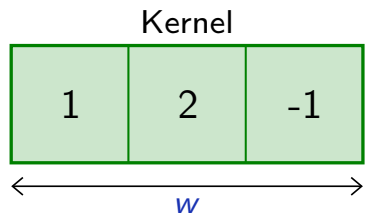
Transposed convolutions also have a `dilation` parameter that behaves as for convolution and expends the kernel size without increasing the number of parameters by making it sparse.

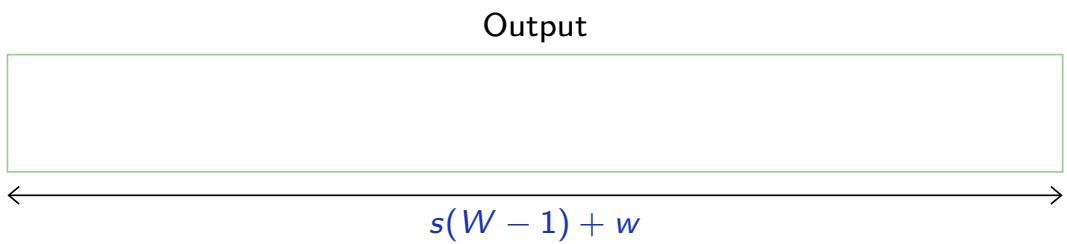
They also have a `stride` and `padding` parameters, however, due to the relation between convolutions and transposed convolutions:

While for convolutions `stride` and `padding` are defined in the input map, for transposed convolutions these parameters are defined in the output map, and the latter modulates a cropping operation.

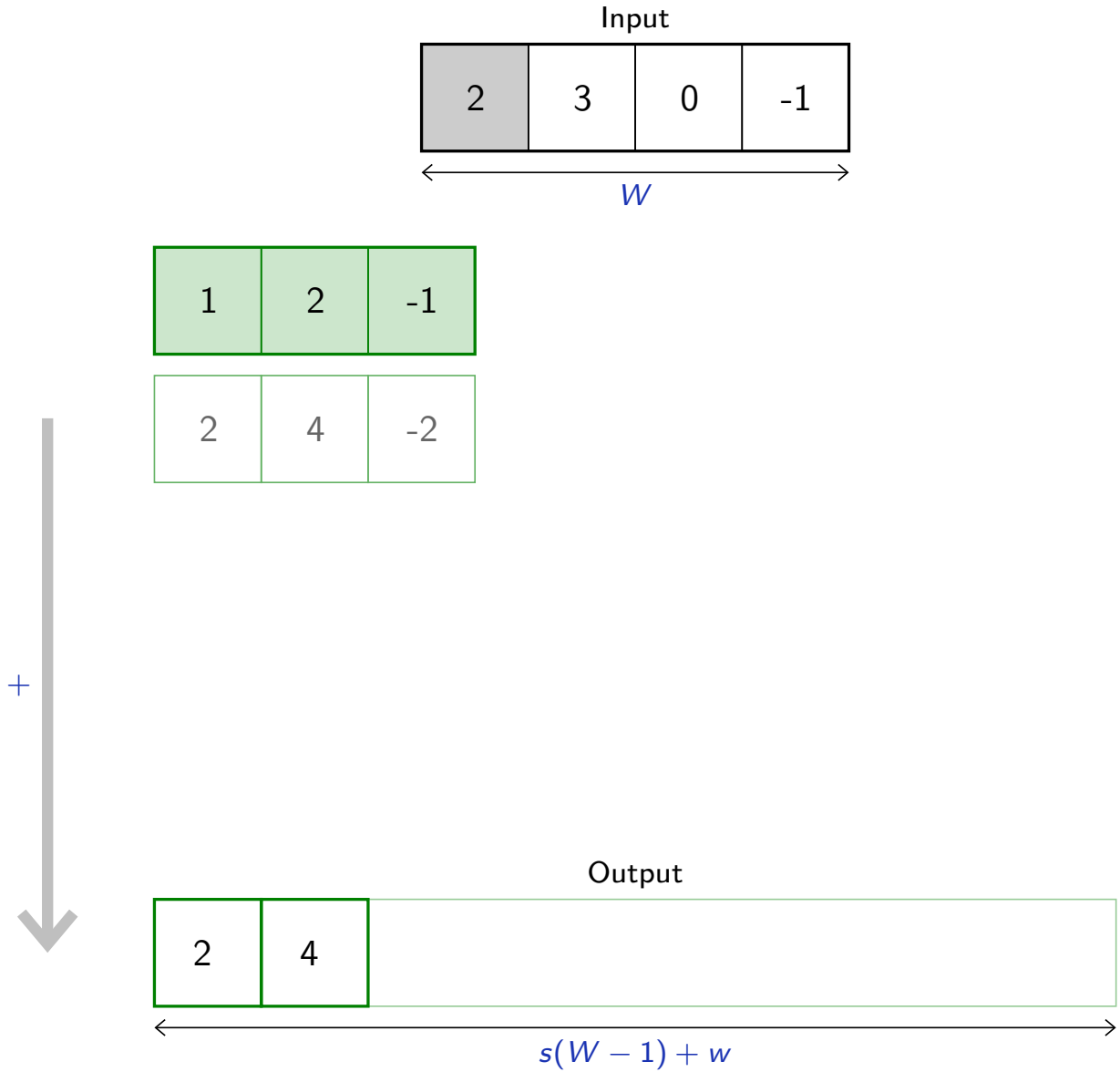
Transposed convolution layer (stride $= 2$)



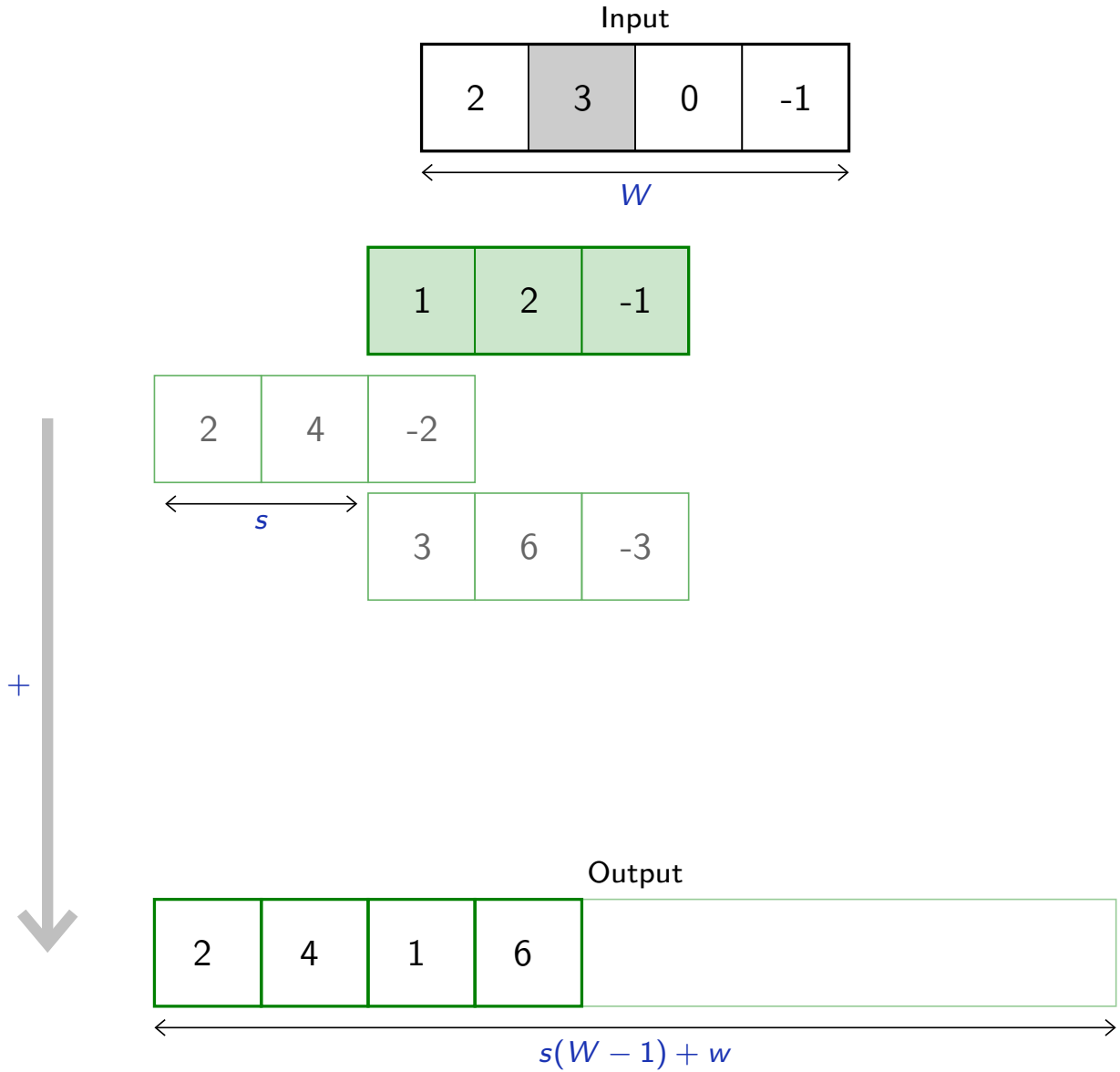




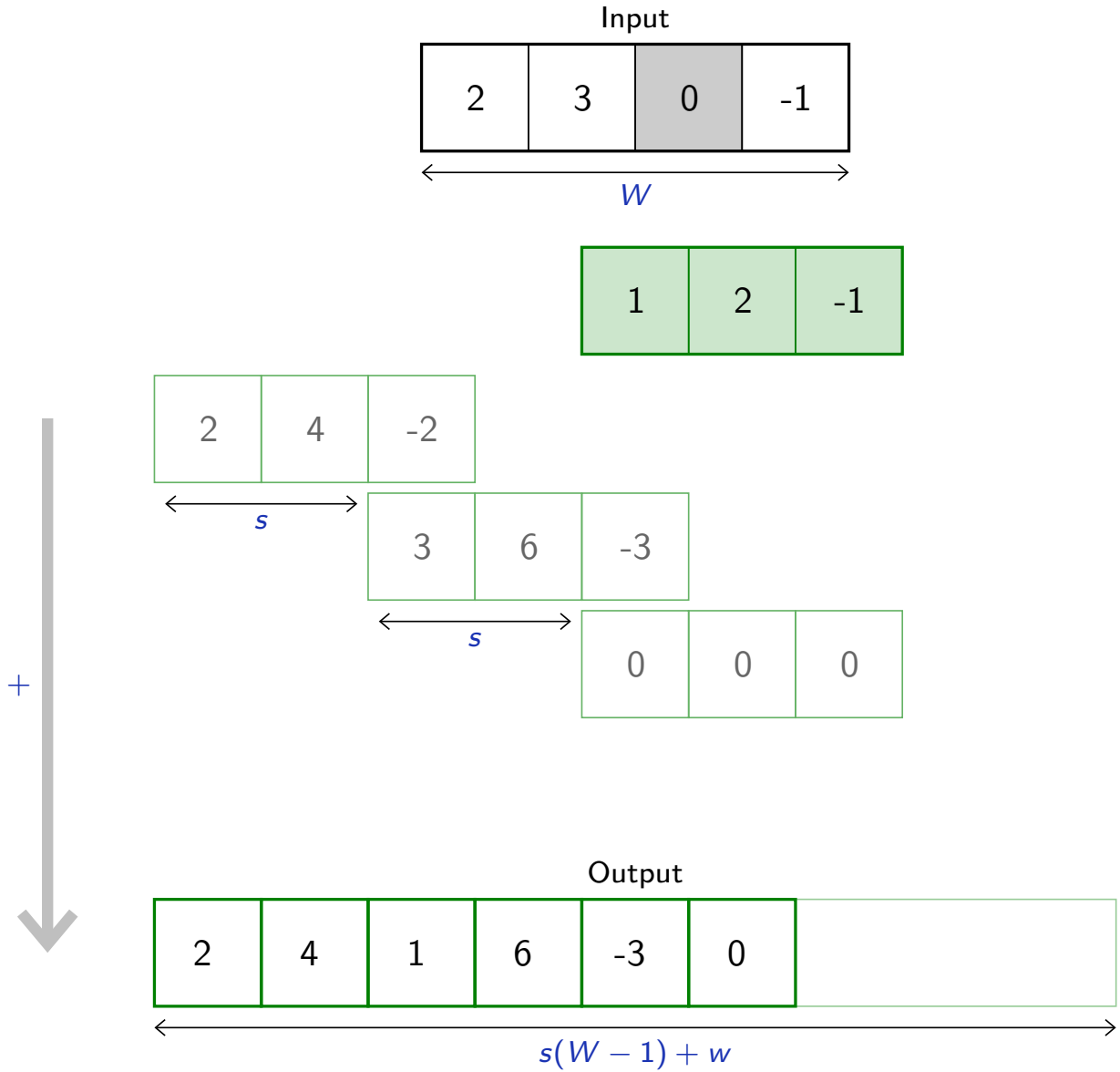
Transposed convolution layer (stride $= 2$)



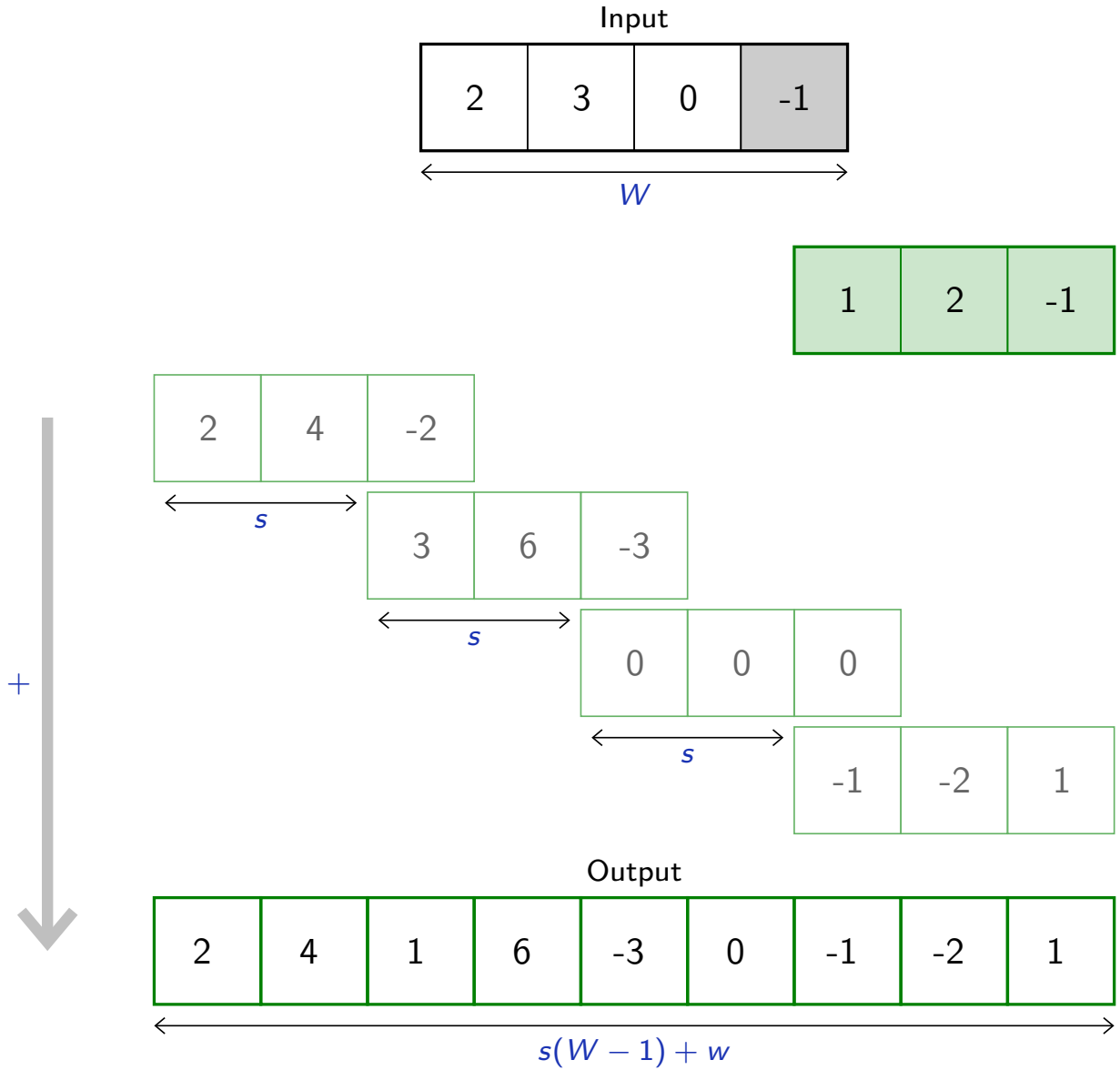
Transposed convolution layer (stride $= 2$)



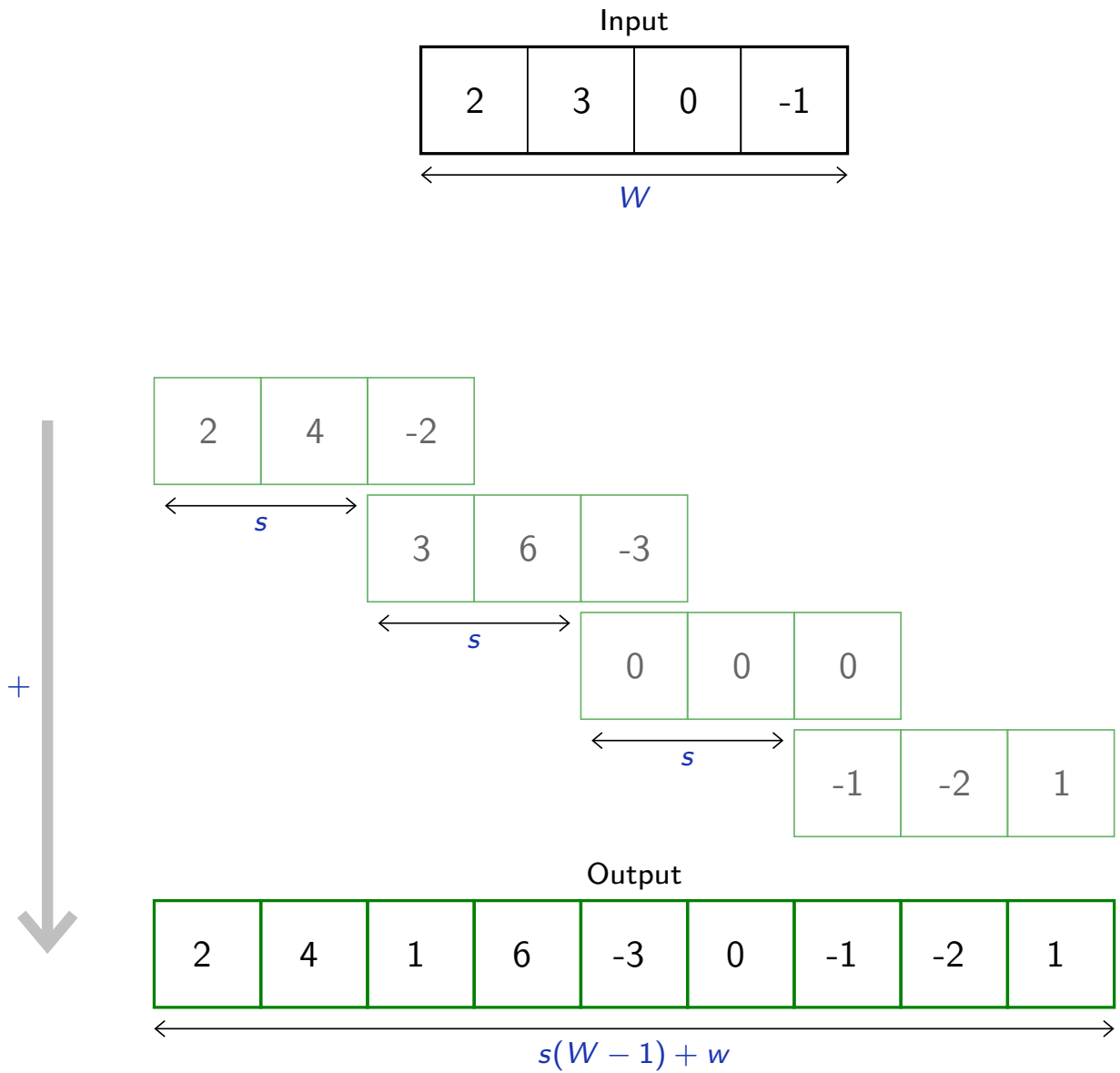
Transposed convolution layer (stride $= 2$)



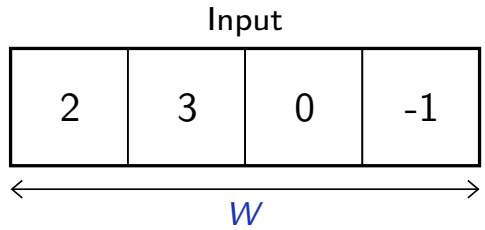
Transposed convolution layer (stride $= 2$)

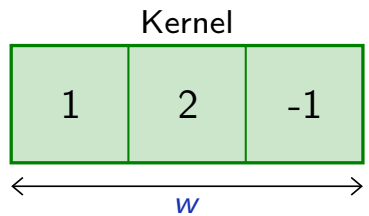


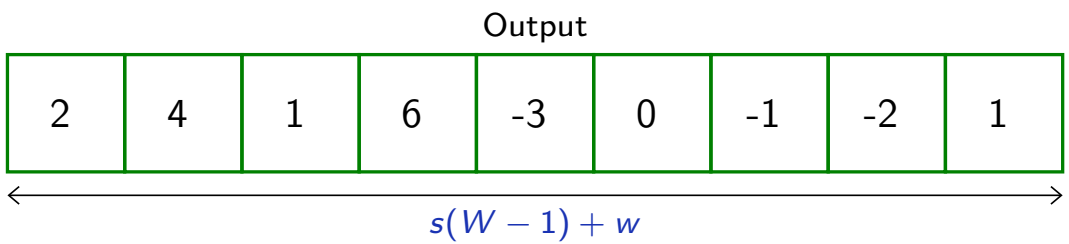
Transposed convolution layer (stride $= 2$)



Transposed convolution layer (stride $= 2$)





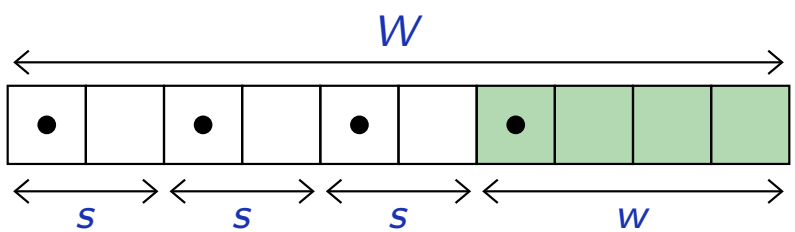


The composition of a convolution and a transposed convolution of same parameters keep the signal size [roughly] unchanged.

A convolution with a stride greater than one may ignore parts of the signal. Its composition with the corresponding transposed convolution generates a map **of the size of the observed area**.

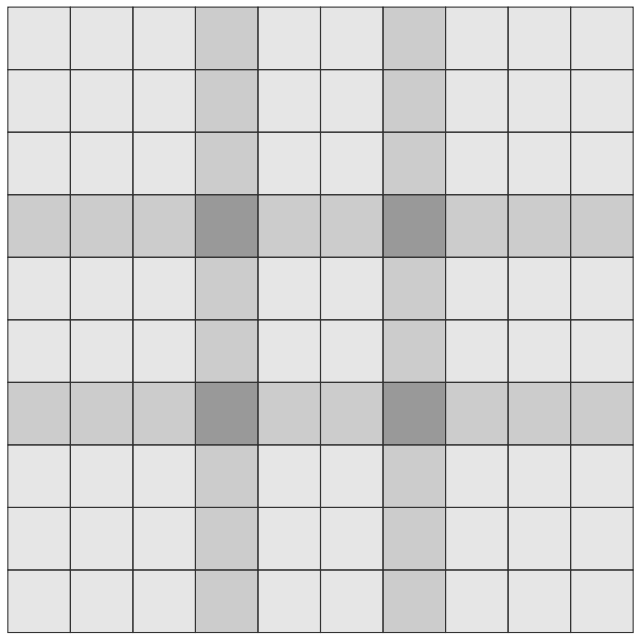
For instance, a 1d convolution of kernel size w and stride s composed with the transposed convolution of same parameters maintains the signal size W, only if

$$\exists q \in \mathbb{N},\ W = w + s\,q.$$



It has been observed that transposed convolutions may create some grid-structure artifacts, since generated pixels are not all covered similarly.

For instance with a 4×4 kernel and stride 3



An alternative is to use an analytic up-scaling, implemented in the PyTorch functional `F.interpolate`.

```
>>> x = torch.tensor([[[[ 1., 2. ], [ 3., 4. ]]]])
>>> F.interpolate(x, scale_factor = 3, mode = 'bilinear')
tensor([[[[1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
          [1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
          [1.6667, 1.6667, 2.0000, 2.3333, 2.6667, 2.6667],
          [2.3333, 2.3333, 2.6667, 3.0000, 3.3333, 3.3333],
          [3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000],
          [3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000]]]])

>>> F.interpolate(x, scale_factor = 3, mode = 'nearest')
tensor([[[[1., 1., 1., 2., 2., 2.],
          [1., 1., 1., 2., 2., 2.],
          [1., 1., 1., 2., 2., 2.],
          [3., 3., 3., 4., 4., 4.],
          [3., 3., 3., 4., 4., 4.],
          [3., 3., 3., 4., 4., 4.]]]])
```

Such module is usually combined with a convolution to learn local corrections to undesirable artifacts of the up-scaling.

In practice, a transposed convolution such as

```
tconv = nn.ConvTranspose2d(nic, noc,
                           kernel_size = 3, stride = 2,
                           padding = 1, output_padding = 1),

y = tconv(x)
```

can be replaced by

```
conv = nn.Conv2d(nic, noc, kernel_size = 3, padding = 1)

u = F.interpolate(x, scale_factor = 2, mode = 'bilinear')
y = conv(u)
```

Deep learning

7.2. Deep Autoencoders

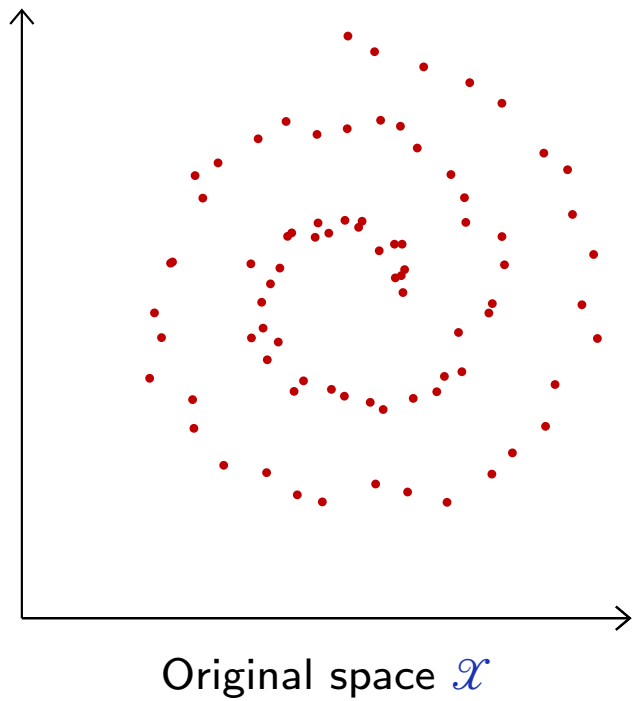
François Fleuret

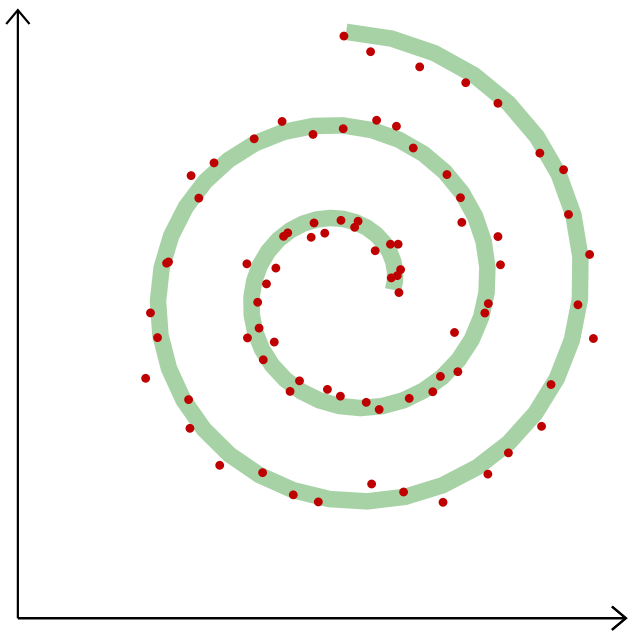
https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

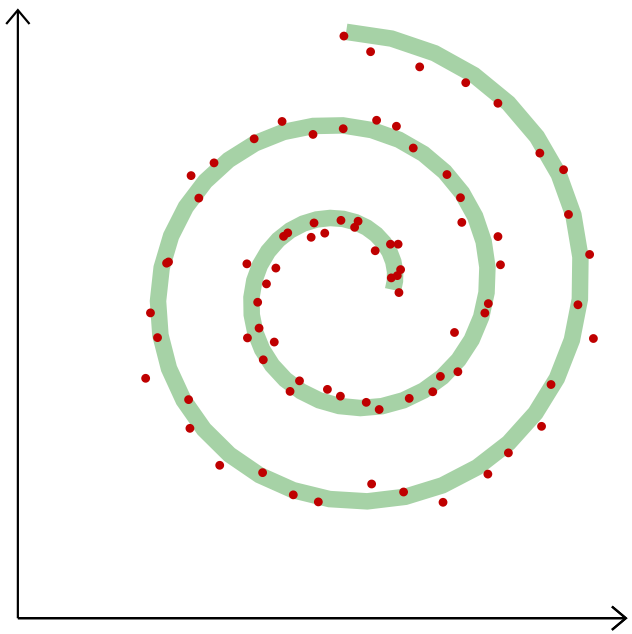
Many applications such as image synthesis, denoising, super-resolution, speech synthesis, compression, etc. require to go beyond classification and regression, and model explicitly a high dimension signal.

This modeling consists of finding "meaningful degrees of freedom" that describe the signal, and are of lesser dimension.

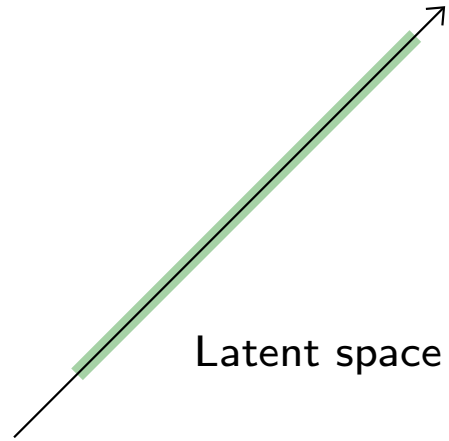

Original space $\mathcal{X}$



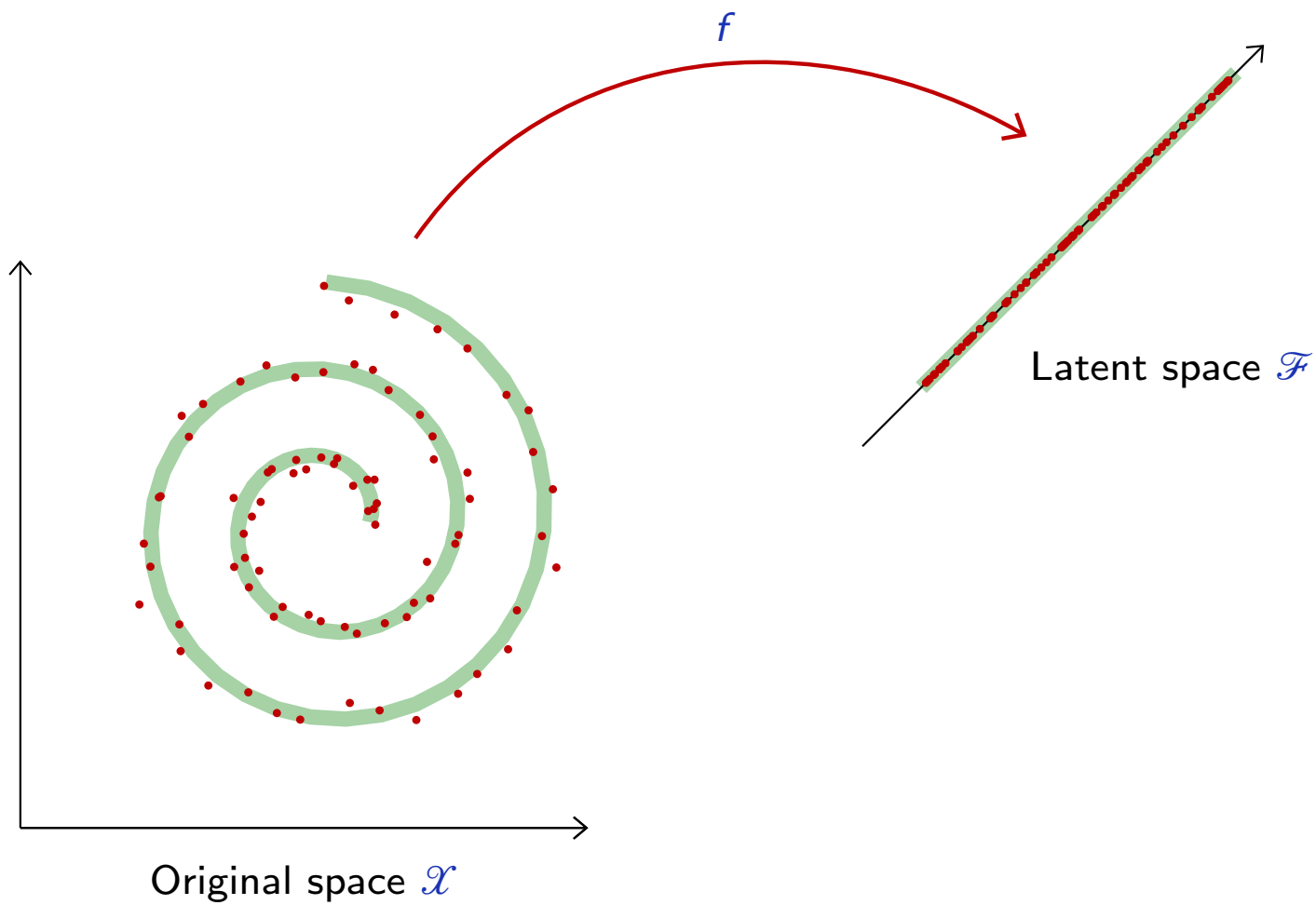
Original space $\mathcal{X}$

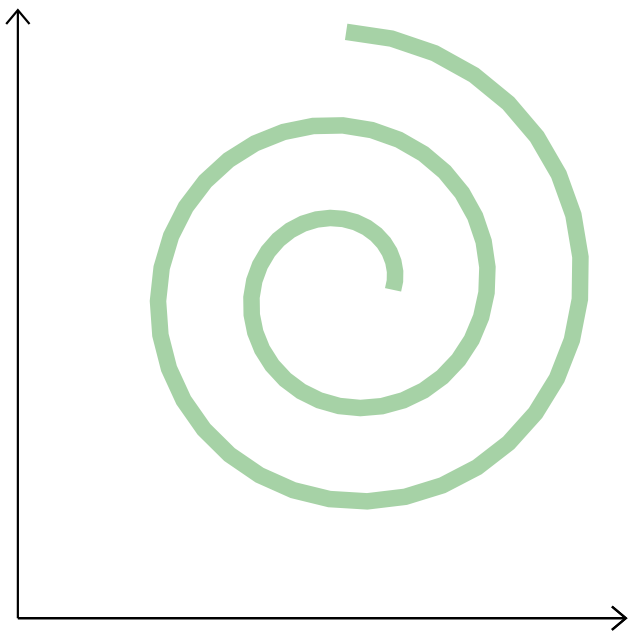


Original space $\mathcal{X}$

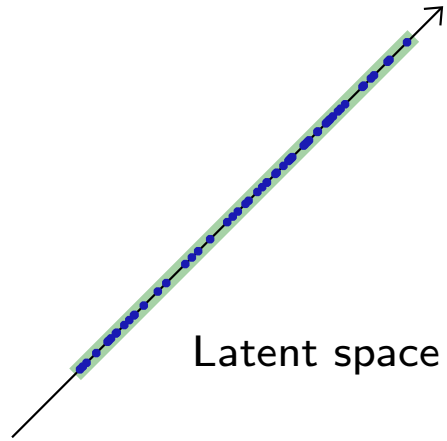


Latent space $\mathcal{F}$

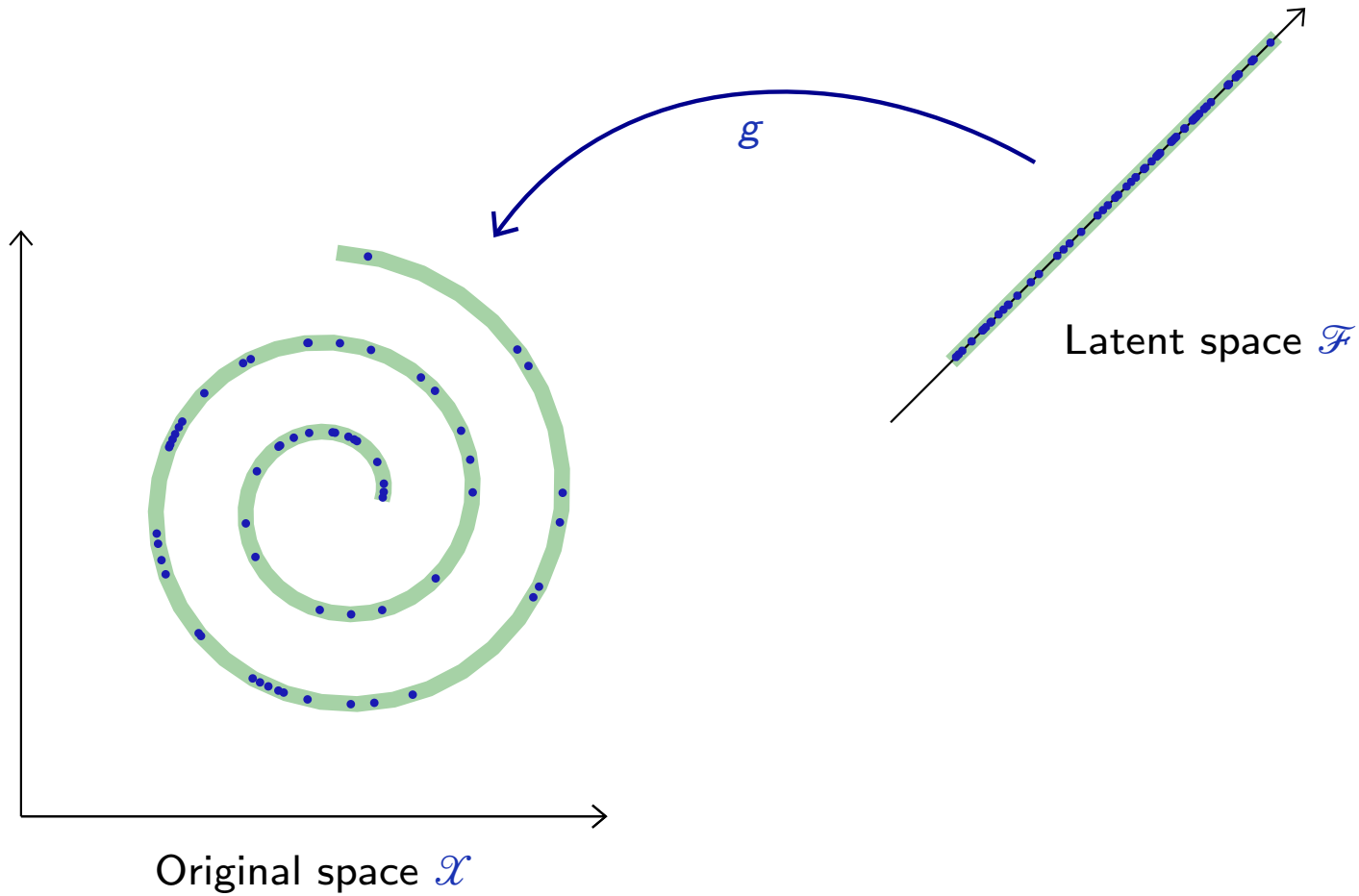

f
Latent space $\mathcal{F}$
Original space $\mathcal{X}$

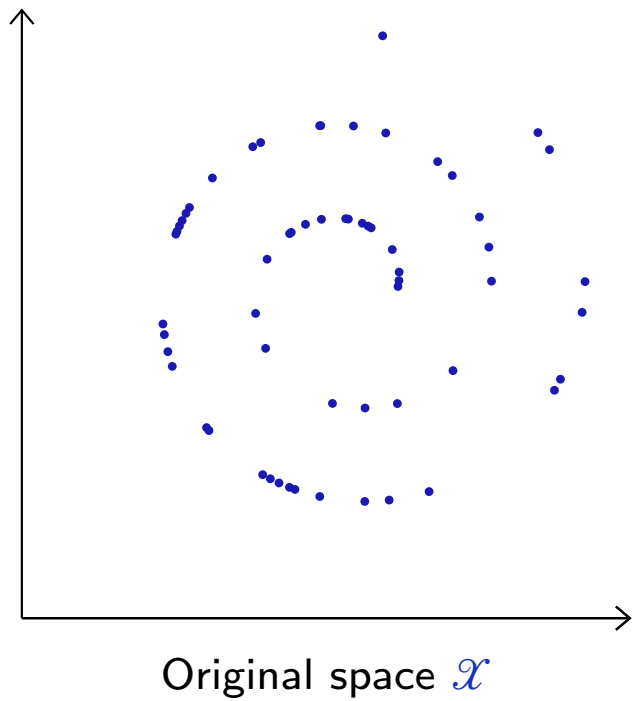


Original space $\mathcal{X}$



Latent space $\mathcal{F}$


g
Latent space $\mathcal{F}$
Original space $\mathcal{X}$


Original space $\mathcal{X}$

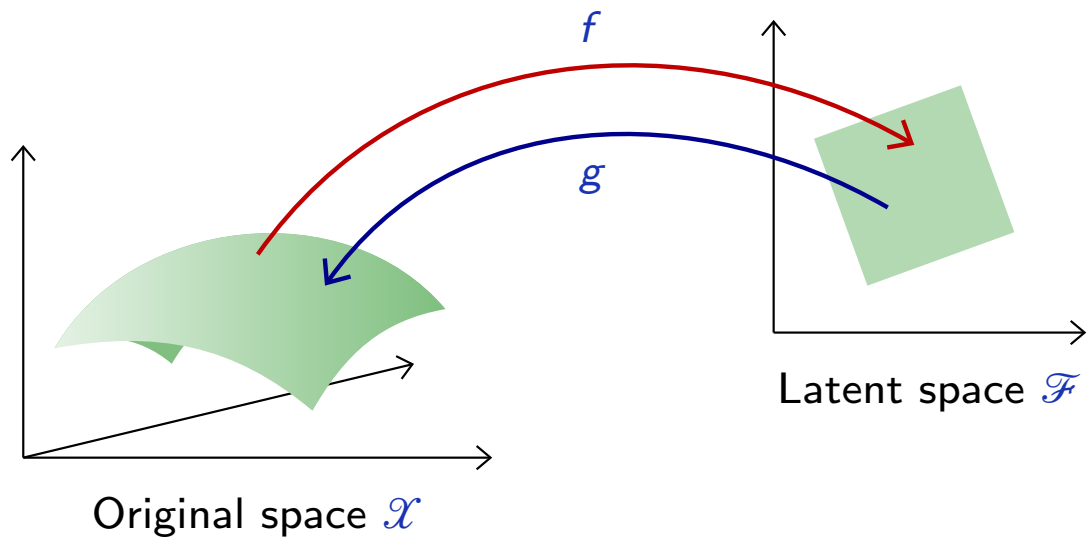
When dealing with real-world signals, this objective involves the same theoretical and practical issues as for classification or regression: defining the right class of high-dimension models, and optimizing them.

Regarding synthesis, we saw that deep feed-forward architectures exhibit good generative properties, which motivates their use explicitly for that purpose.

Autoencoders

An autoencoder maps a space to itself and is [close to] the identity on the data.

Dimension reduction can be achieved with an autoencoder composed of an **encoder** f from the original space $\mathcal{X}$ to a **latent** space $\mathcal{F}$, and a **decoder** g to map back to $\mathcal{X}$ (Bourlard and Kamp, 1988; Hinton and Zemel, 1994).



If the latent space is of lower dimension, the autoencoder has to capture a "good" parametrization, and in particular dependencies between components.

Let q be the data distribution over $\mathcal{X}$. A good autoencoder could be characterized with the quadratic loss

$$\mathbb{E}_{X \sim q}\left[\|X - g \circ f(X)\|^2\right] \simeq 0.$$

Given two parametrized mappings $f(\cdot\,; w_f)$ and $g(\cdot\,; w_g)$, training consists of minimizing an empirical estimate of that loss

$$\hat{w}_f, \hat{w}_g = \operatorname*{argmin}_{w_f, w_g} \frac{1}{N} \sum_{n=1}^{N} \|x_n - g(f(x_n; w_f); w_g)\|^2 .$$

A simple example of such an autoencoder would be with both f and g linear, in which case the optimal solution is given by PCA. Better results can be achieved with more sophisticated classes of mappings, in particular deep architectures.

Deep Autoencoders

A deep autoencoder combines an encoder composed of convolutional layers, with a decoder composed of transposed convolutions or other interpolating layers. E.g. for MNIST:

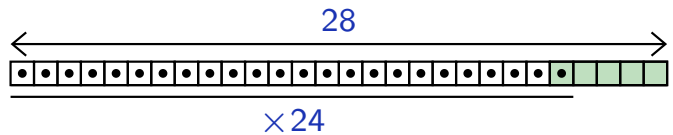
```
AutoEncoder (
  (encoder): Sequential (
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
    (3): ReLU (inplace)
    (4): Conv2d(32, 32, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU (inplace)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2))
    (7): ReLU (inplace)
    (8): Conv2d(32, 8, kernel_size=(4, 4), stride=(1, 1))
  )
  (decoder): Sequential (
    (0): ConvTranspose2d(8, 32, kernel_size=(4, 4), stride=(1, 1))
    (1): ReLU (inplace)
    (2): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(2, 2))
    (3): ReLU (inplace)
    (4): ConvTranspose2d(32, 32, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU (inplace)
    (6): ConvTranspose2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
    (7): ReLU (inplace)
    (8): ConvTranspose2d(32, 1, kernel_size=(5, 5), stride=(1, 1))
  )
)
```

Encoder

Tensor sizes / operations

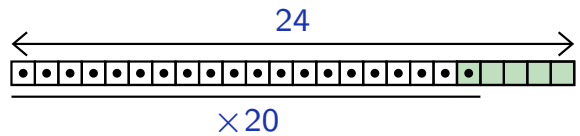
$1 \times 28 \times 28$

`nn.Conv2d(1, 32, kernel_size=5, stride=1)`



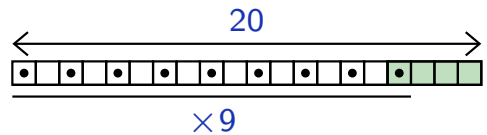
$32 \times 24 \times 24$

`nn.Conv2d(32, 32, kernel_size=5, stride=1)`



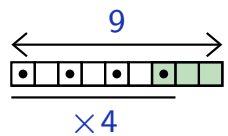
$32 \times 20 \times 20$

`nn.Conv2d(32, 32, kernel_size=4, stride=2)`



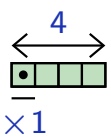
$32 \times 9 \times 9$

`nn.Conv2d(32, 32, kernel_size=3, stride=2)`



$32 \times 4 \times 4$

`nn.Conv2d(32, 8, kernel_size=4, stride=1)`



$8 \times 1 \times 1$

Decoder

Tensor sizes / operations

$8 \times 1 \times 1$

`nn.ConvTranspose2d(8, 32, kernel_size=4, stride=1)`

$32 \times 4 \times 4$

`nn.ConvTranspose2d(32, 32, kernel_size=3, stride=2)`

$32 \times 9 \times 9$

`nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2)`

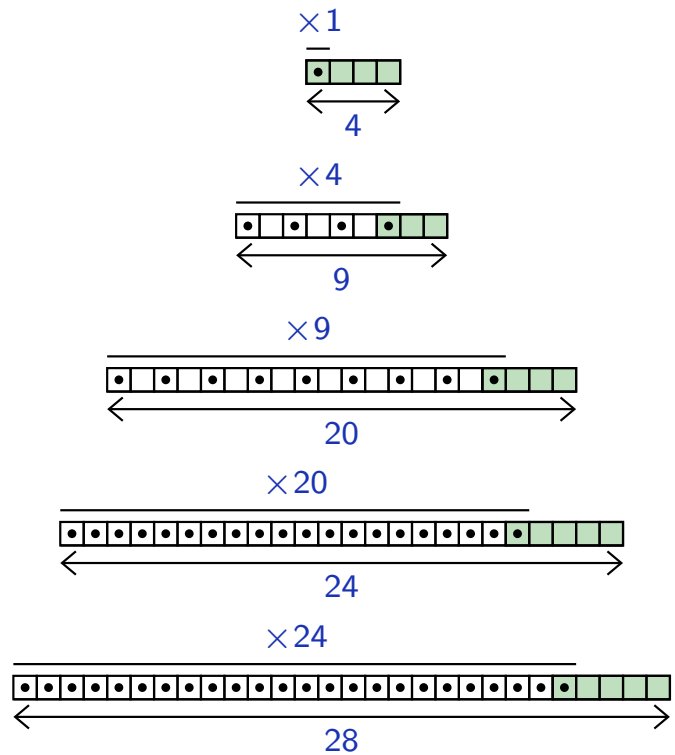
$32 \times 20 \times 20$

`nn.ConvTranspose2d(32, 32, kernel_size=5, stride=1)`

$32 \times 24 \times 24$

`nn.ConvTranspose2d(32, 1, kernel_size=5, stride=1)`

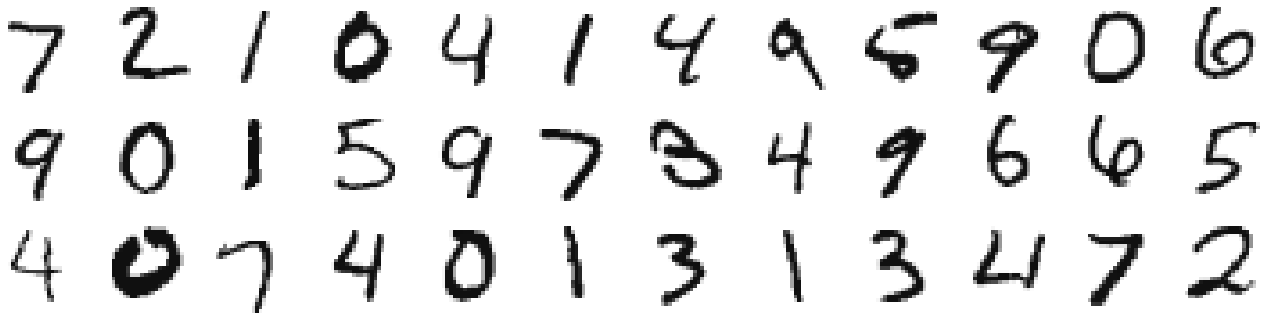
$1 \times 28 \times 28$



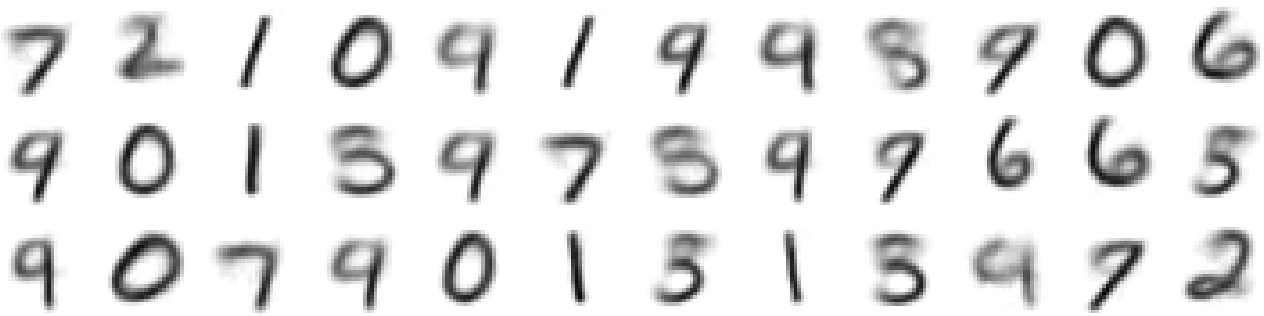
Training is achieved with quadratic loss and Adam

```
model = AutoEncoder(nb_channels, embedding_dim)

optimizer = optim.Adam(model.parameters(), lr = 1e-3)

for epoch in range(args.nb_epochs):
    for input in train_input.split(batch_size):
        z = model.encode(input)
        output = model.decode(z)
        loss = 0.5 * (output - input).pow(2).sum() / input.size(0)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

X (original samples)

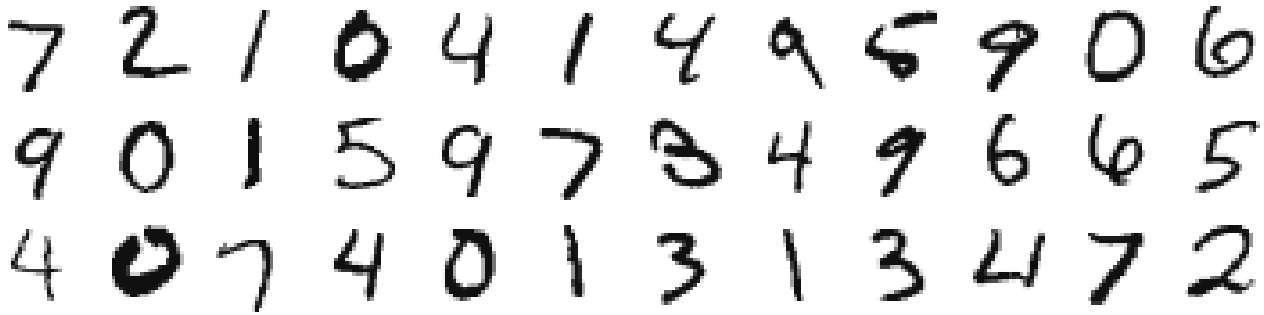


$g \circ f(X)$ (CNN, $d = 2$)

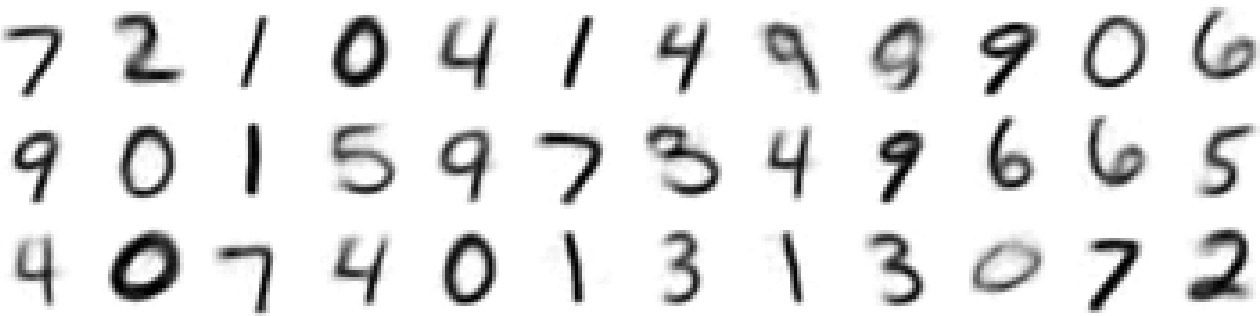


$g \circ f(X)$ (PCA, $d = 2$)

X (original samples)

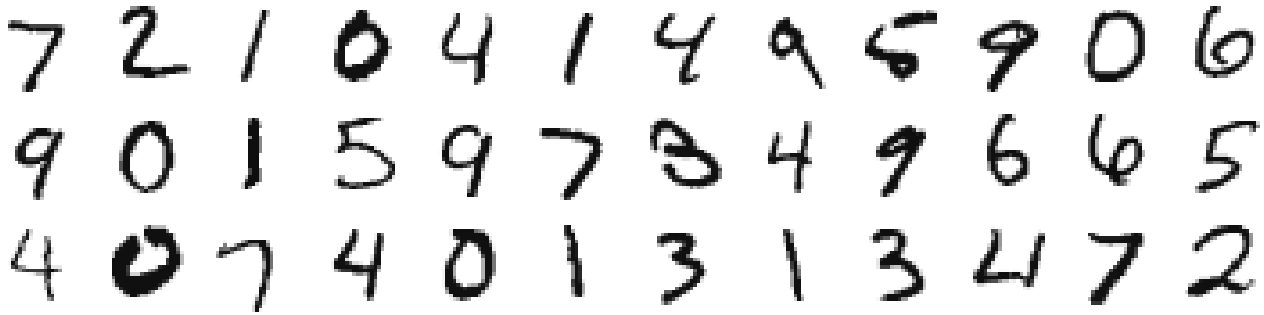


$g \circ f(X)$ (CNN, $d = 4$)

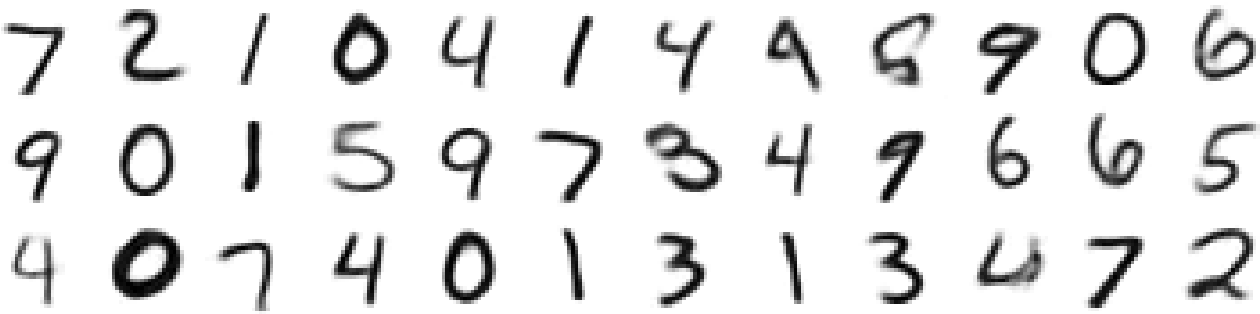


$g \circ f(X)$ (PCA, $d = 4$)

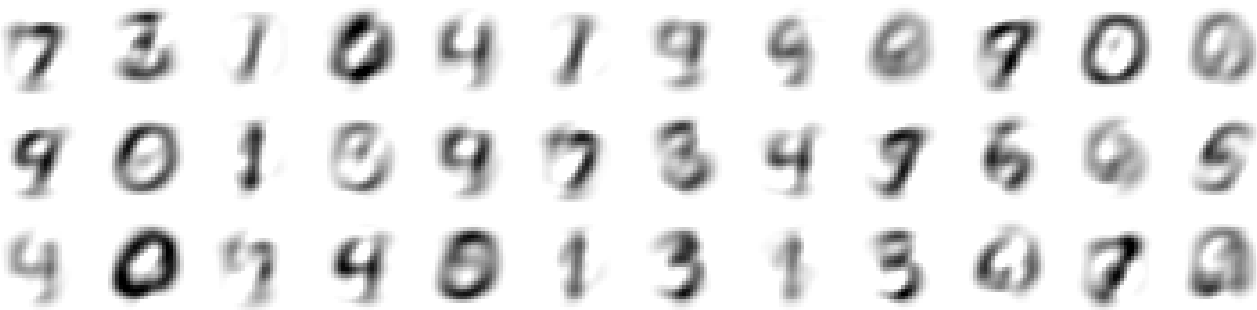
X (original samples)



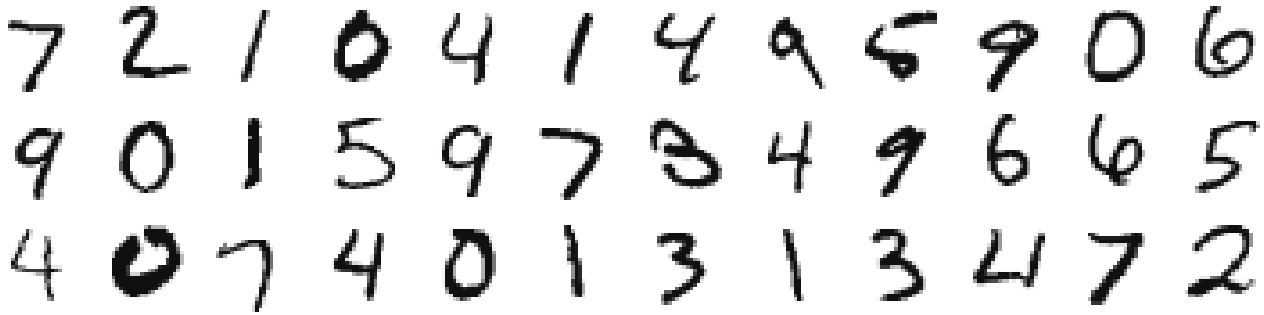
$g \circ f(X)$ (CNN, $d = 8$)



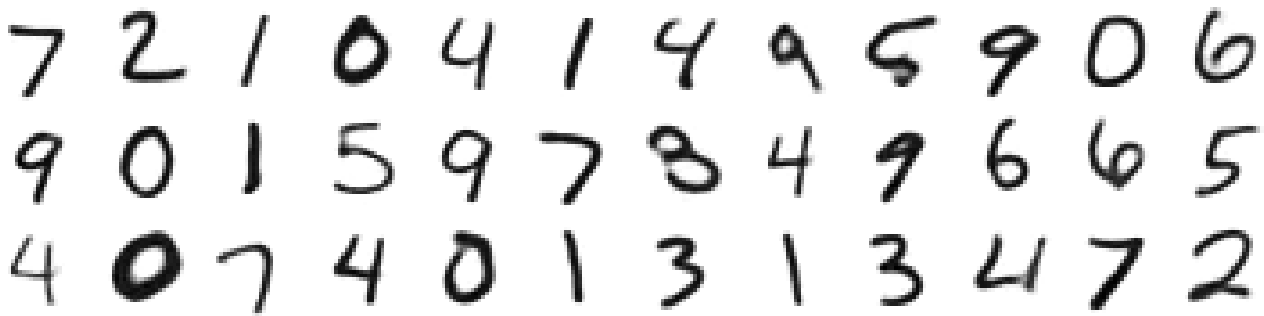
$g \circ f(X)$ (PCA, $d = 8$)



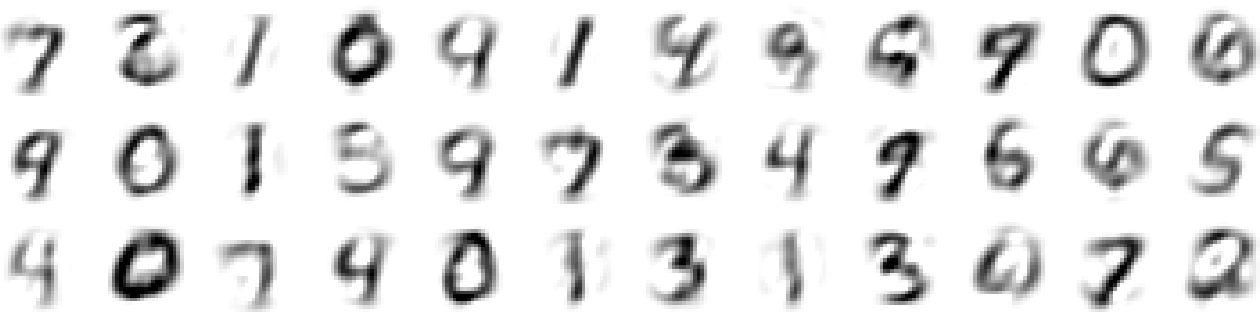
X (original samples)



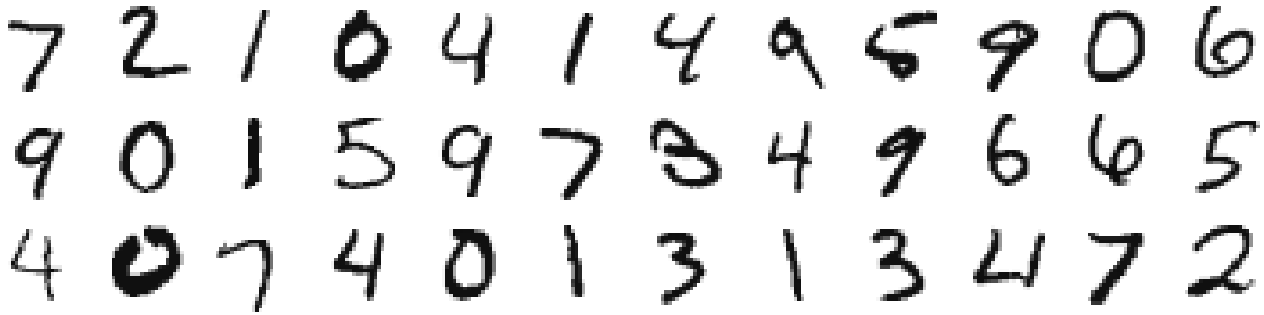
$g \circ f(X)$ (CNN, $d = 16$)



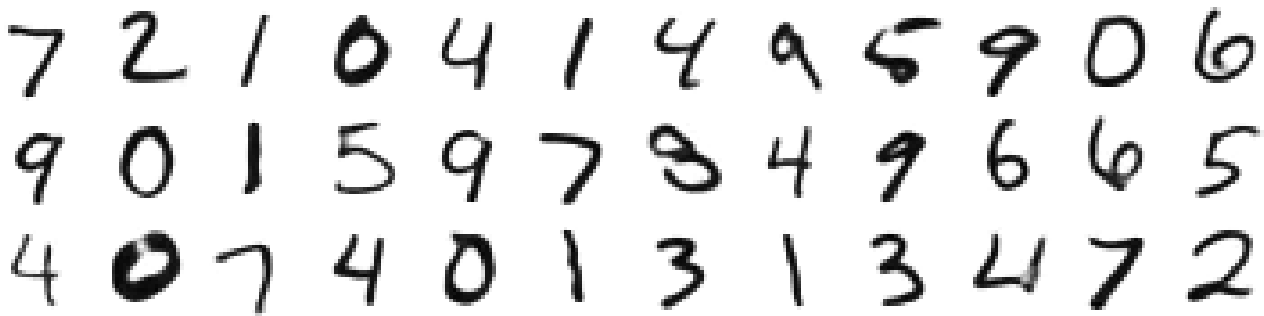
$g \circ f(X)$ (PCA, $d = 16$)



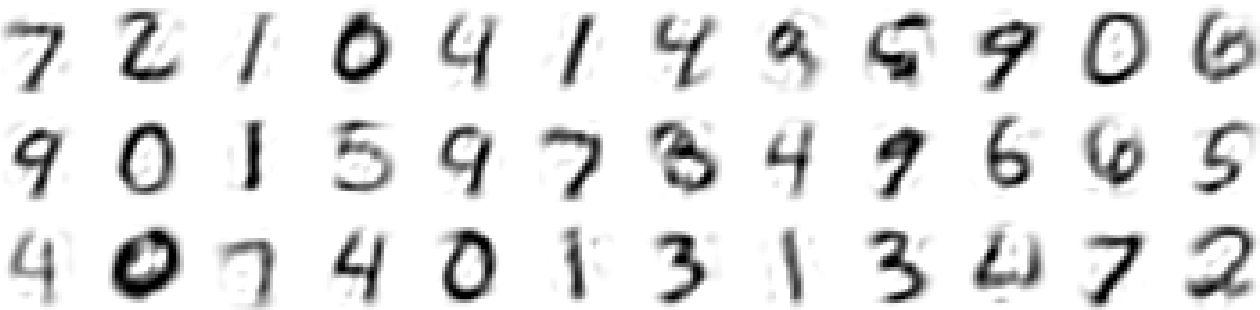
X (original samples)



$g \circ f(X)$ (CNN, $d = 32$)

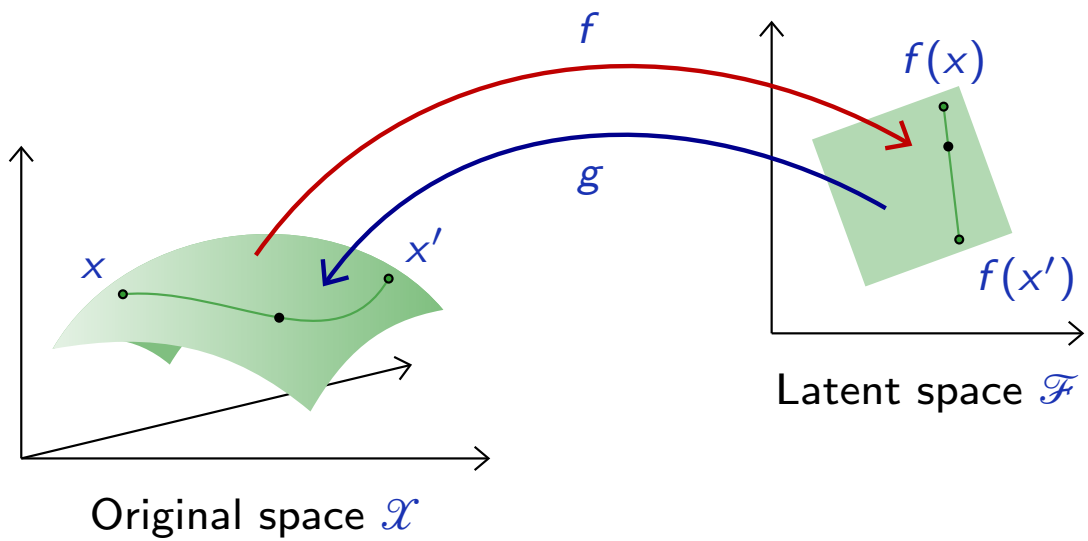


$g \circ f(X)$ (PCA, $d = 32$)

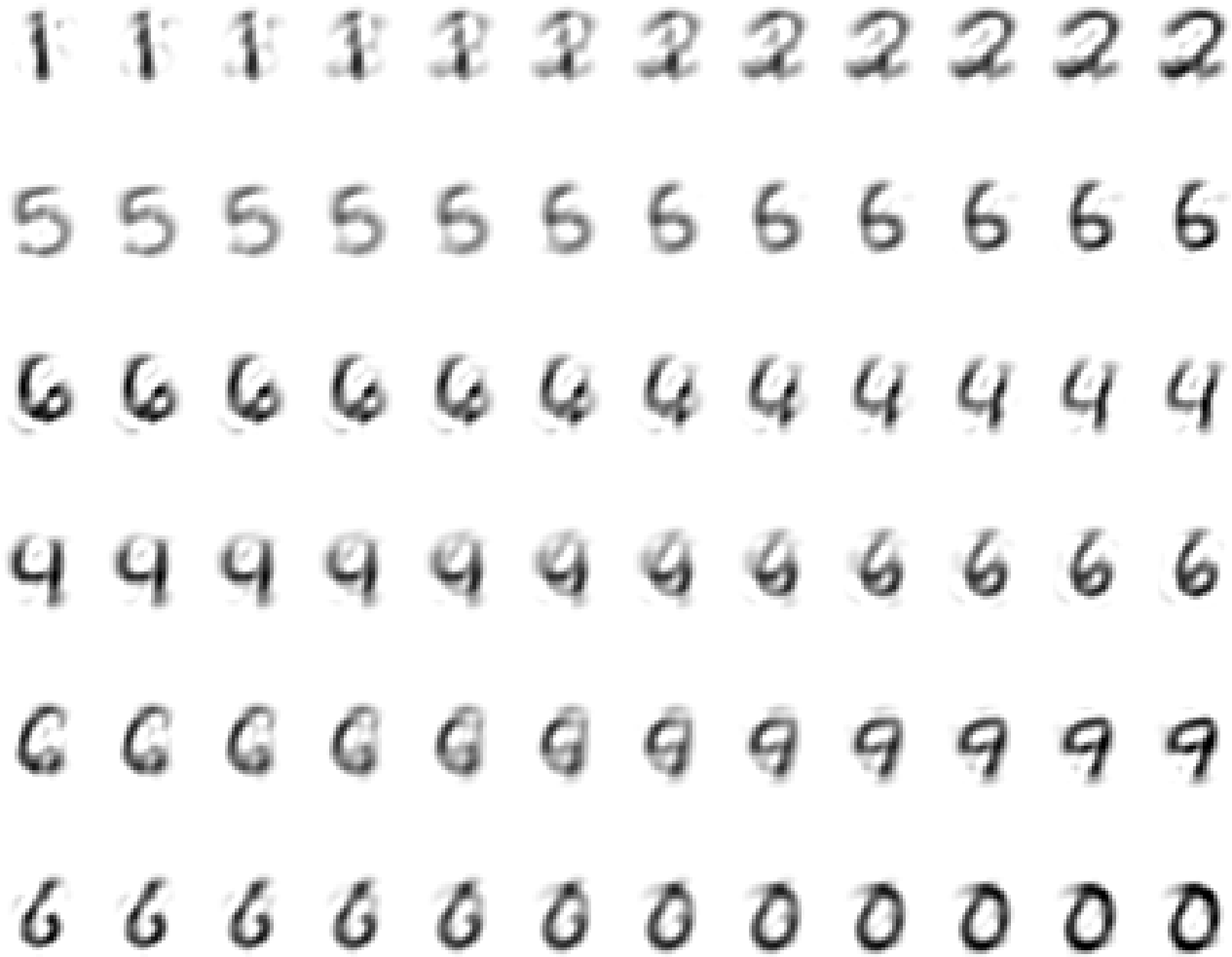


To get an intuition of the latent representation, we can pick two samples x and x' at random and interpolate samples along the line in the latent space

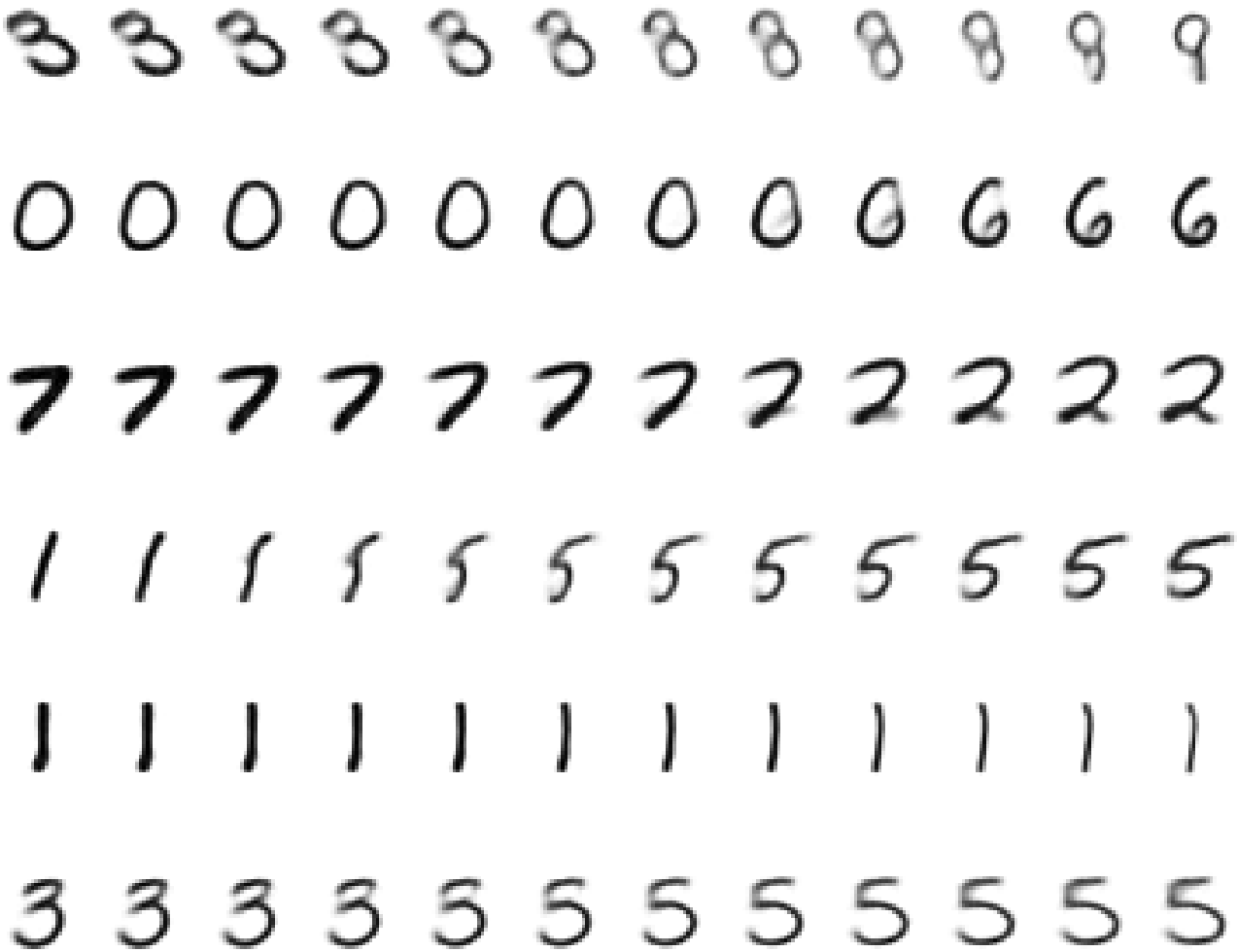
$$\forall x, x' \in \mathcal{X}^2,\ \alpha \in [0,1],\quad \xi(x, x', \alpha) = g((1-\alpha)f(x) + \alpha f(x')).$$



PCA interpolation ($d = 32$)



Autoencoder interpolation ($d = 8$)



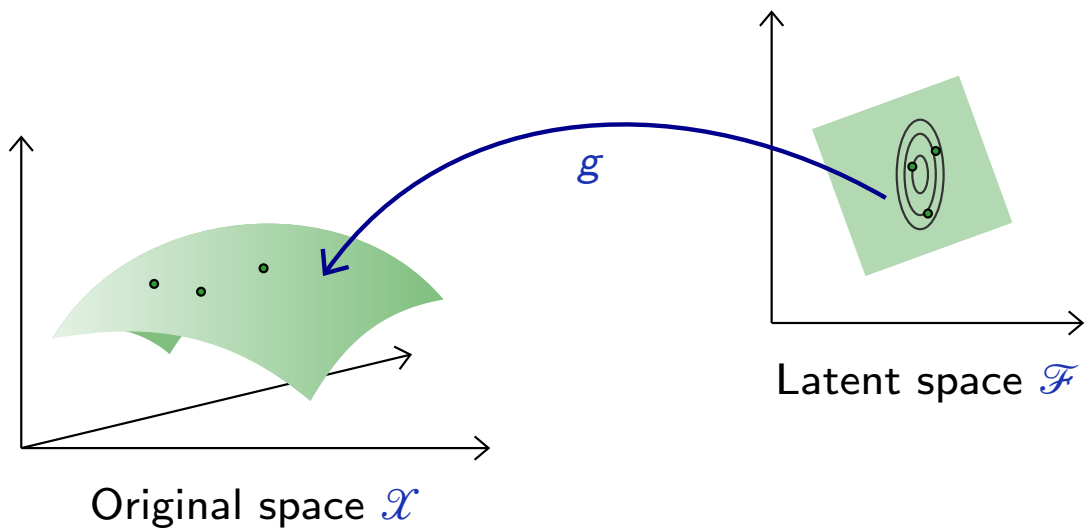
Autoencoder interpolation ($d = 32$)

And we can assess the generative capabilities of the decoder g by introducing a [simple] density model q^Z over the latent space $\mathcal{F}$, sample there, and map the samples into the image space $\mathcal{X}$ with g.

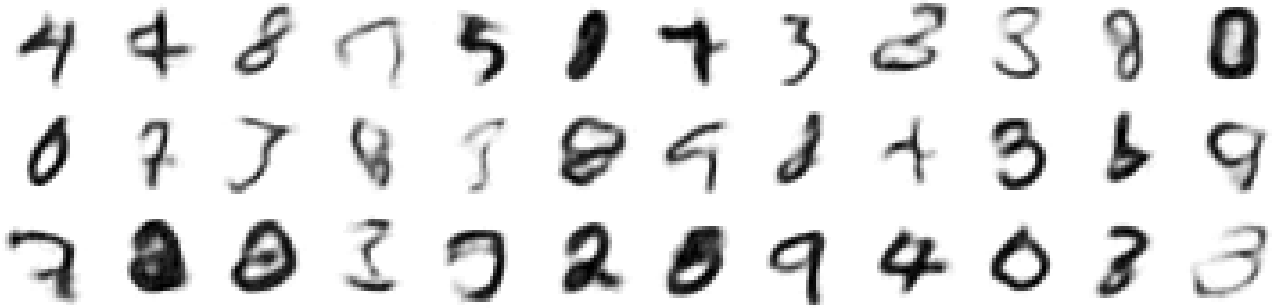
We can for instance use a Gaussian model with diagonal covariance matrix.

$$f(X) \sim \mathcal{N}(\hat{m}, \hat{\Delta})$$

where $\hat{m}$ is a vector and $\hat{\Delta}$ a diagonal matrix, both estimated on training data.



Autoencoder sampling ($d = 8$)



Autoencoder sampling ($d = 16$)

Autoencoder sampling ($d = 32$)

These results are unsatisfying, because the density model used on the latent space $\mathscr{F}$ is too simple and inadequate.

Building a "good" model amounts to our original problem of modeling an empirical distribution, although it may now be in a lower dimension space.

Deep learning

7.3. Denoising autoencoders

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Beside dimension reduction, autoencoders can capture dependencies between signal components to restore a degraded input.

In that case, we can ignore the encoder/decoder structure, and such a model

$$\phi : \mathcal{X} \to \mathcal{X}.$$

is referred to as a **denoising** autoencoder.

The goal is not anymore to optimize ϕ so that

$$\phi(X) \simeq X$$

but, given a perturbation $\tilde{X}$ of the signal X, to restore the signal, hence

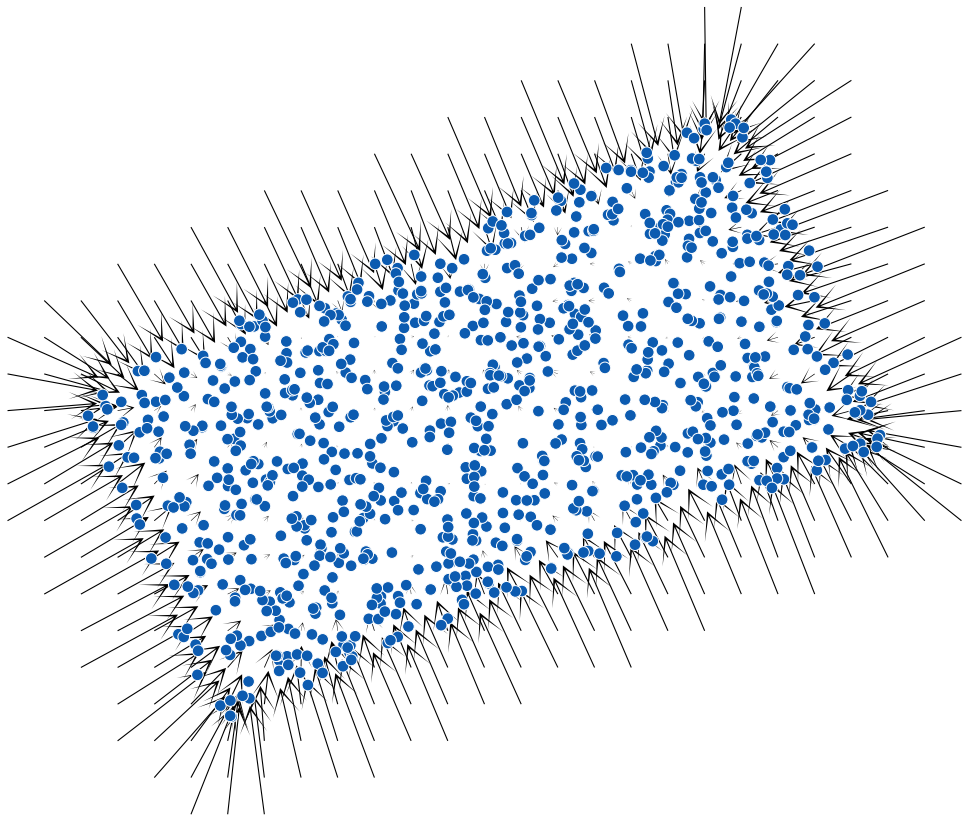
$$\phi(\tilde{X}) \simeq X.$$

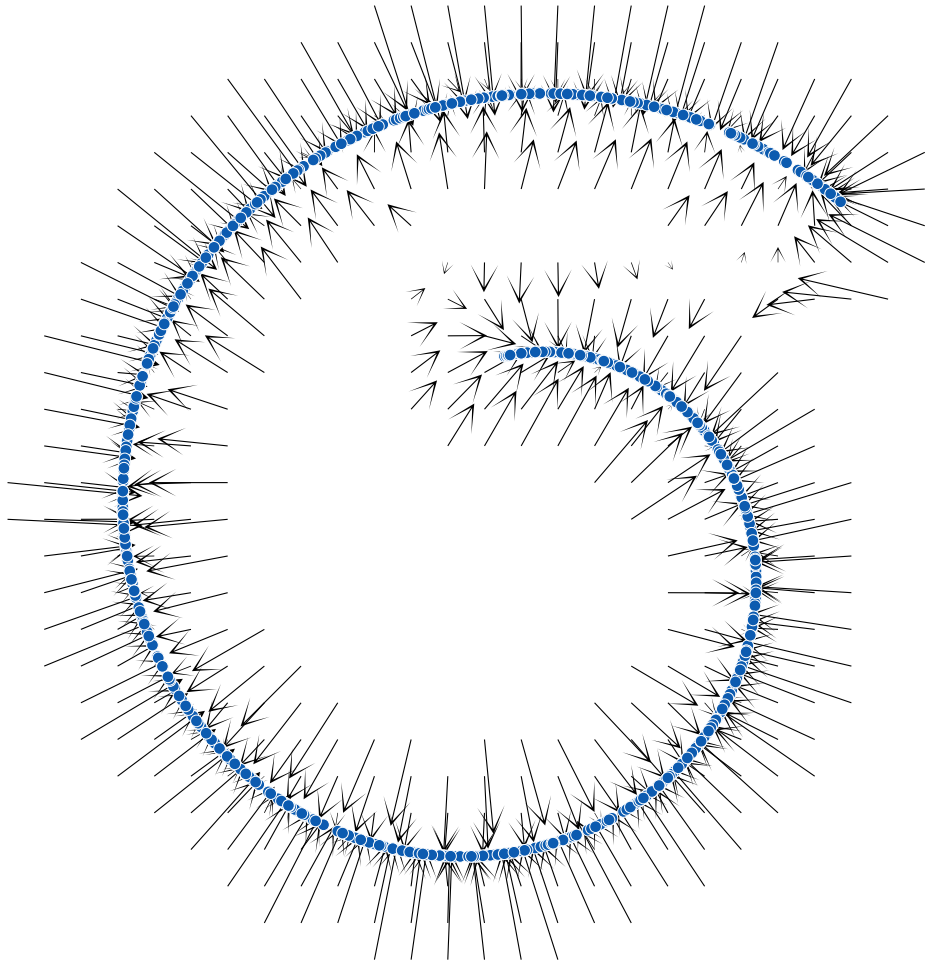
We can illustrate this notion in $2d$ with an additive Gaussian noise, and the quadratic loss, hence

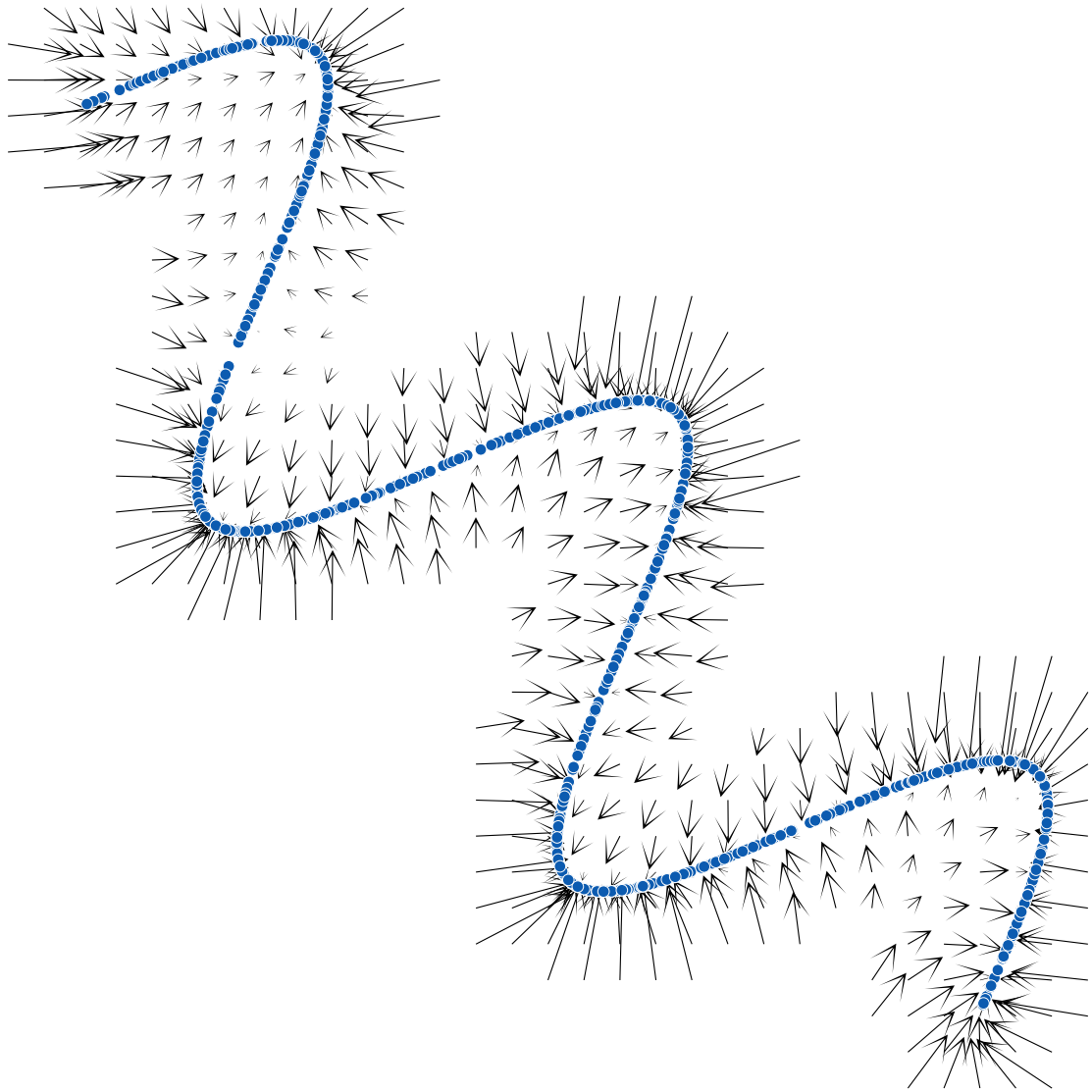
$$\hat{w} = \operatorname*{argmin}_{w} \frac{1}{N} \sum_{n=1}^{N} \|x_n - \phi(x_n + \epsilon_n; w)\|^2 ,$$

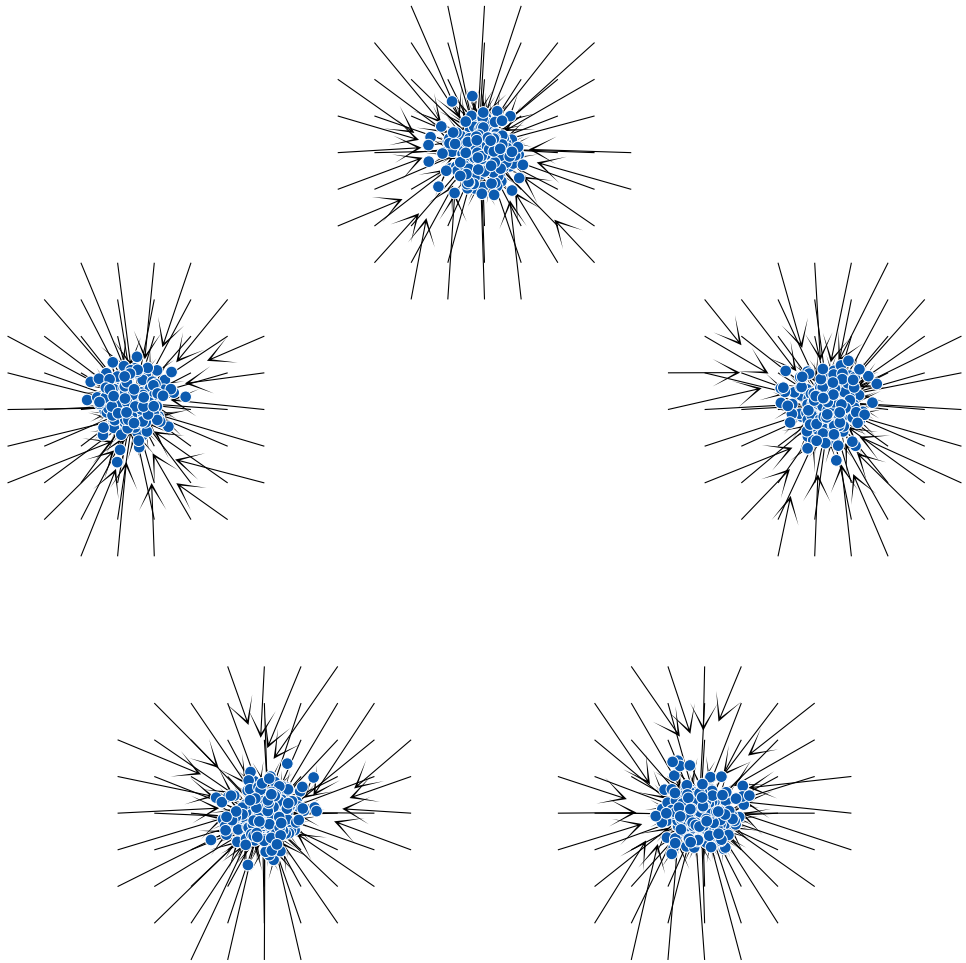
where x_n are the data samples, and ϵ_n are Gaussian random noise vectors.

```
model = nn.Sequential(
    nn.Linear(2, 100),
    nn.ReLU(),
    nn.Linear(100, 2)
)

batch_size, nb_epochs = 100, 1000
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
mse = nn.MSELoss()

for e in range(nb_epochs):
    for input in data.split(batch_size):
        noise = input.new(input.size()).normal_(0, 0.1)
        output = model(input + noise)
        loss = mse(output, input)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```









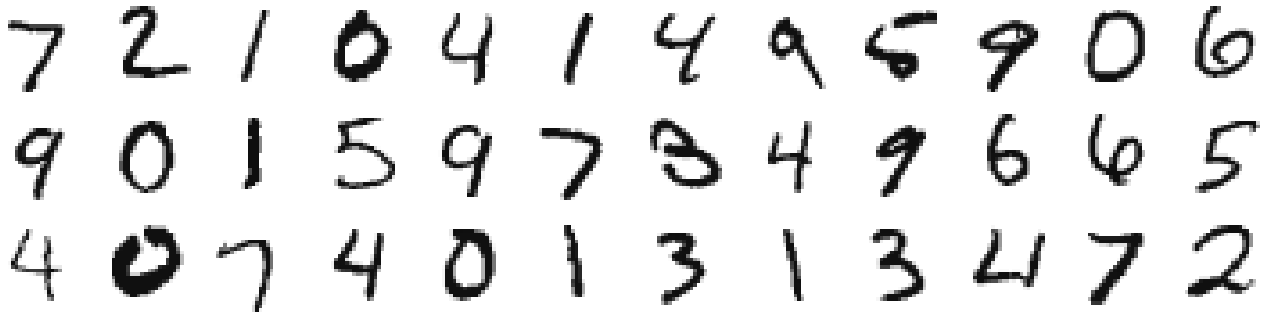
We can do the same on MNIST, for which we keep our deep autoencoder, and ignore its encoder/decoder structure.

```
corrupted_input = corruptor.corrupt(input)

output = model(corrupted_input)
loss = 0.5 * (output - input).pow(2).sum() / input.size(0)

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

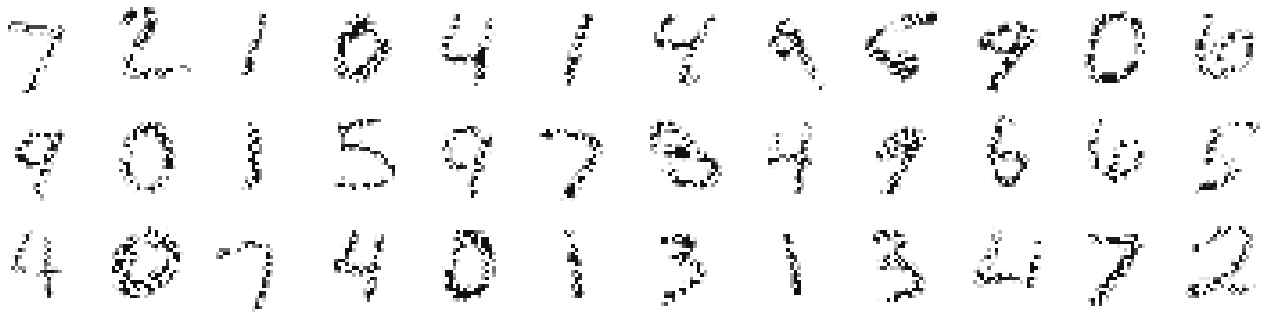
We consider three types of corruptions, that go beyond additive noise:

Original Pixel erasure Blurring Block masking

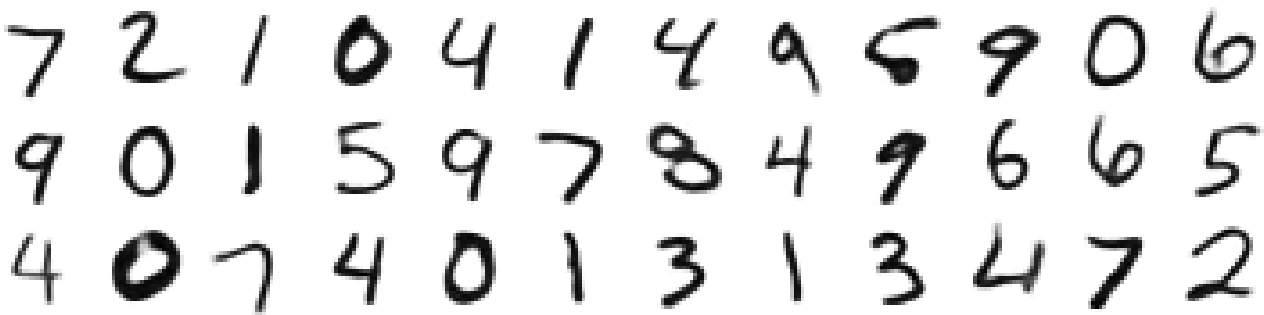
Original



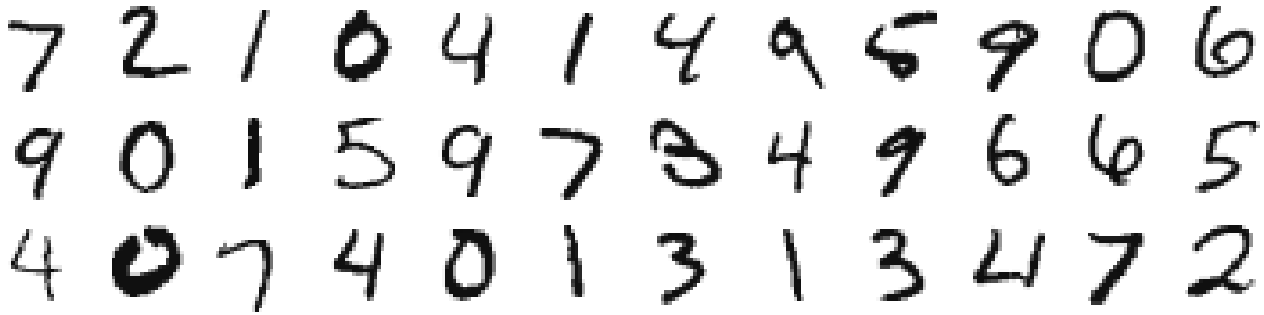
Corrupted ($p = 0.5$)



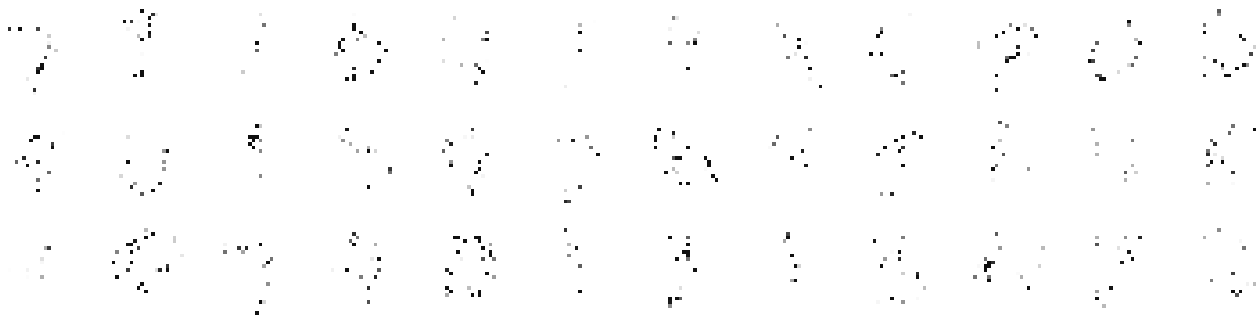
Reconstructed



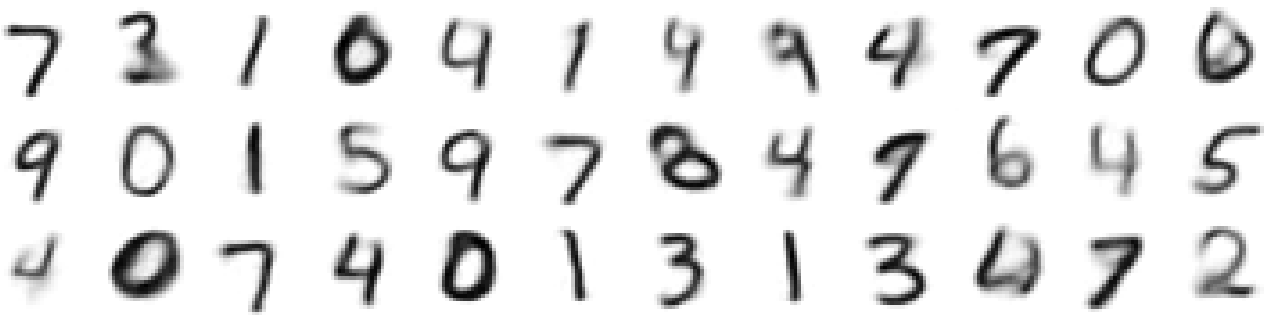
Original



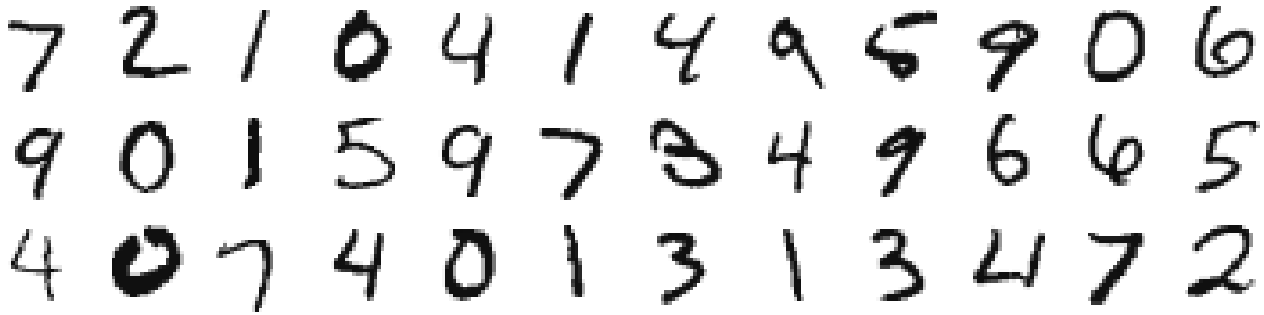
Corrupted ($p = 0.9$)



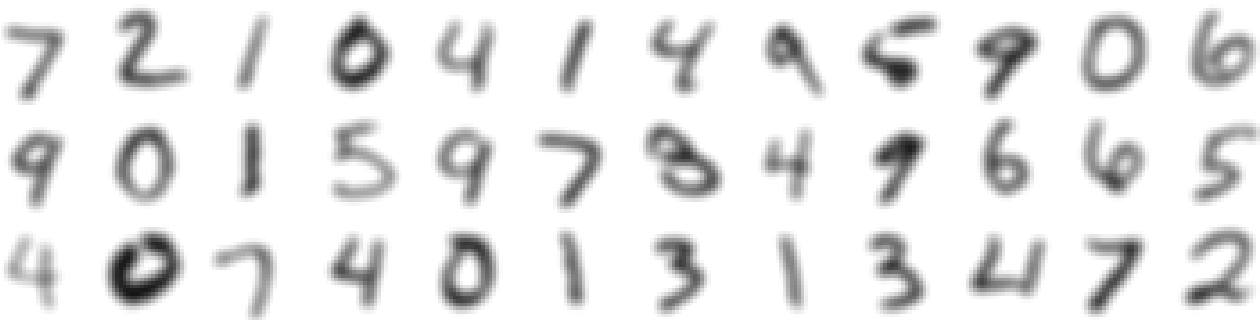
Reconstructed



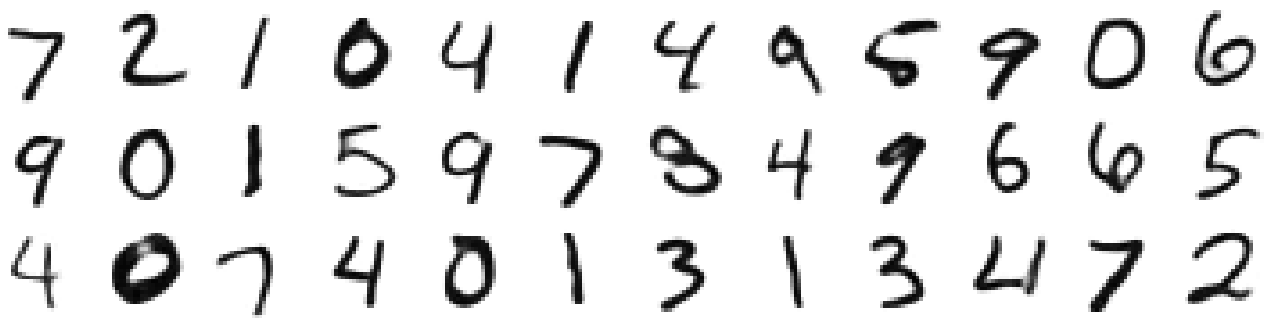
Original



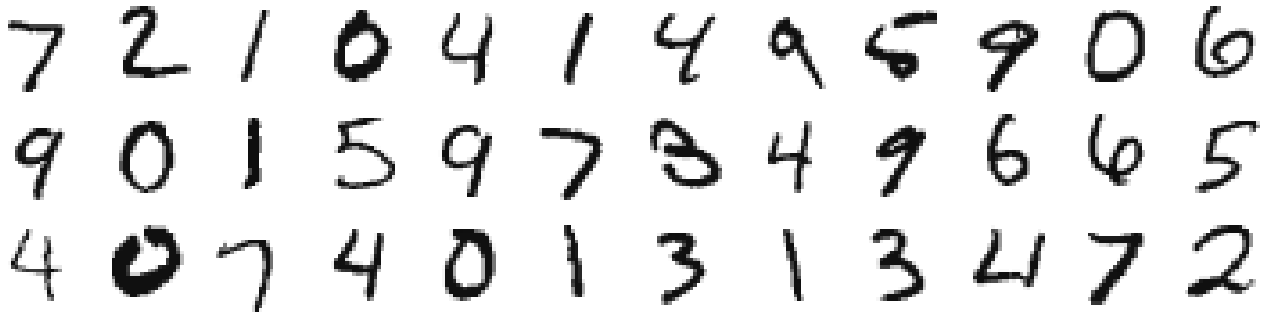
Corrupted ($\sigma = 2$)



Reconstructed

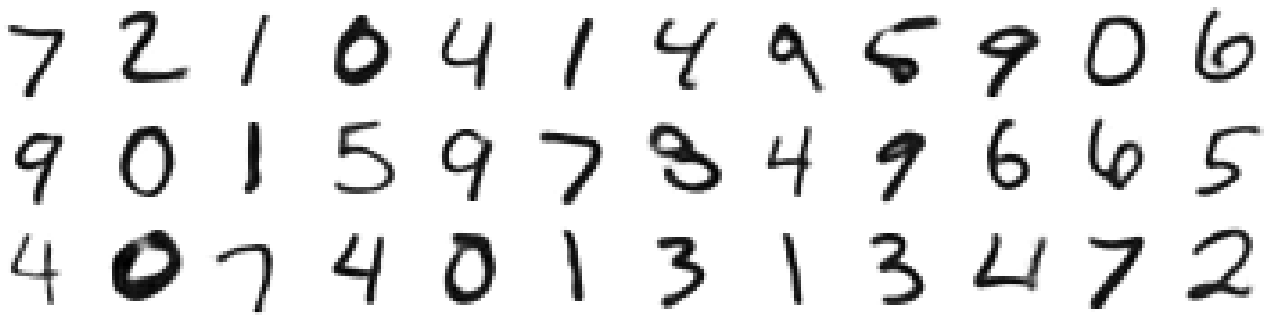


Original

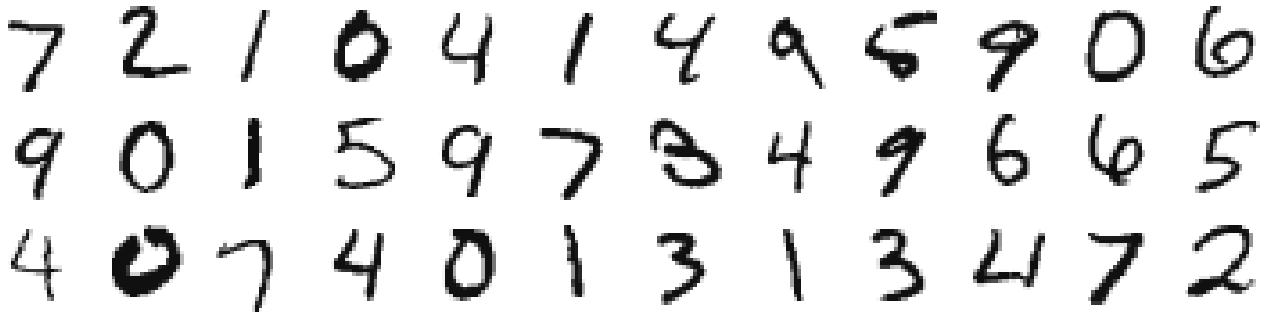


Corrupted ($\sigma = 4$)

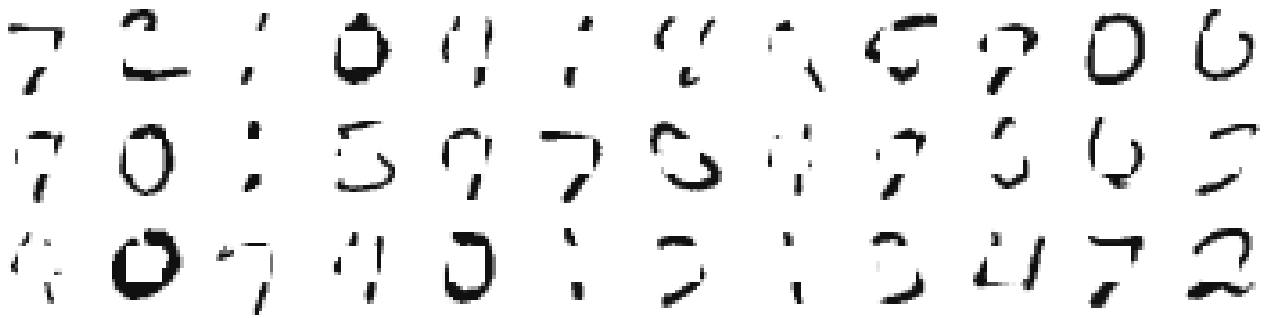
Reconstructed



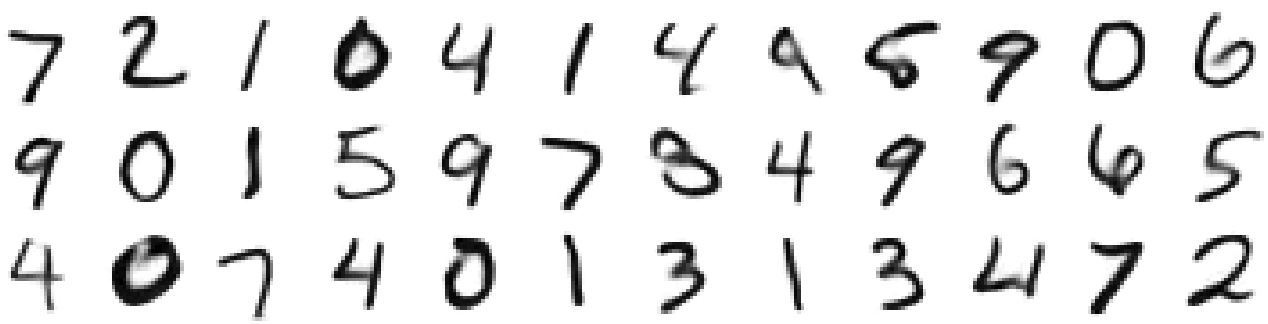
Original



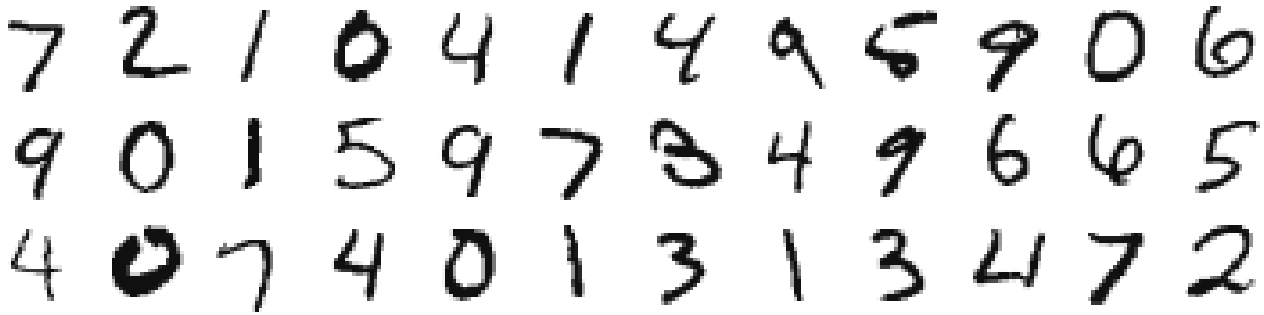
Corrupted (10×10)



Reconstructed

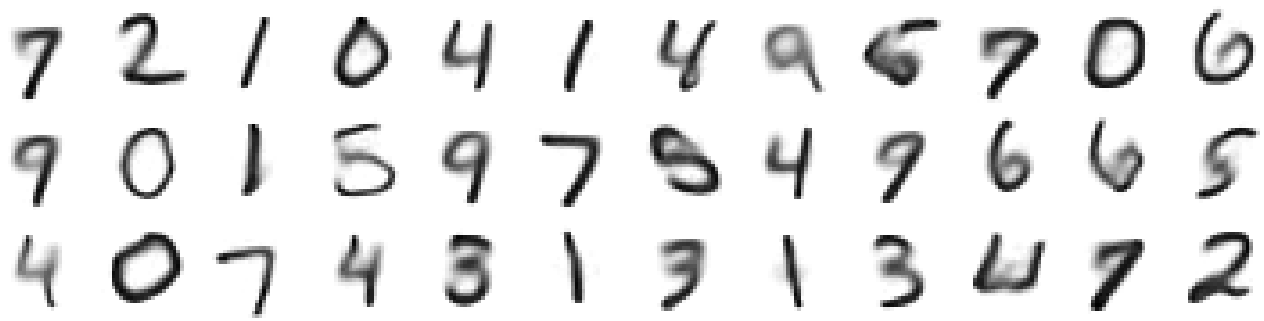


Original



Corrupted (16×16)

Reconstructed



A key weakness of this type of denoising is that the posterior

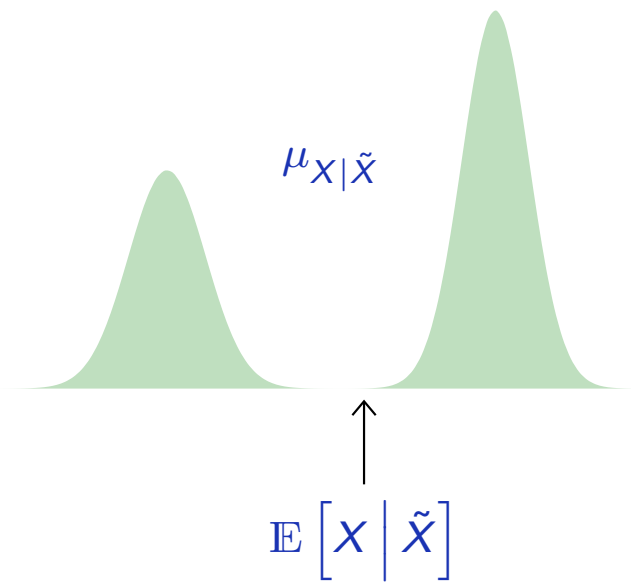
$$\mu_{X|\tilde{x}}$$

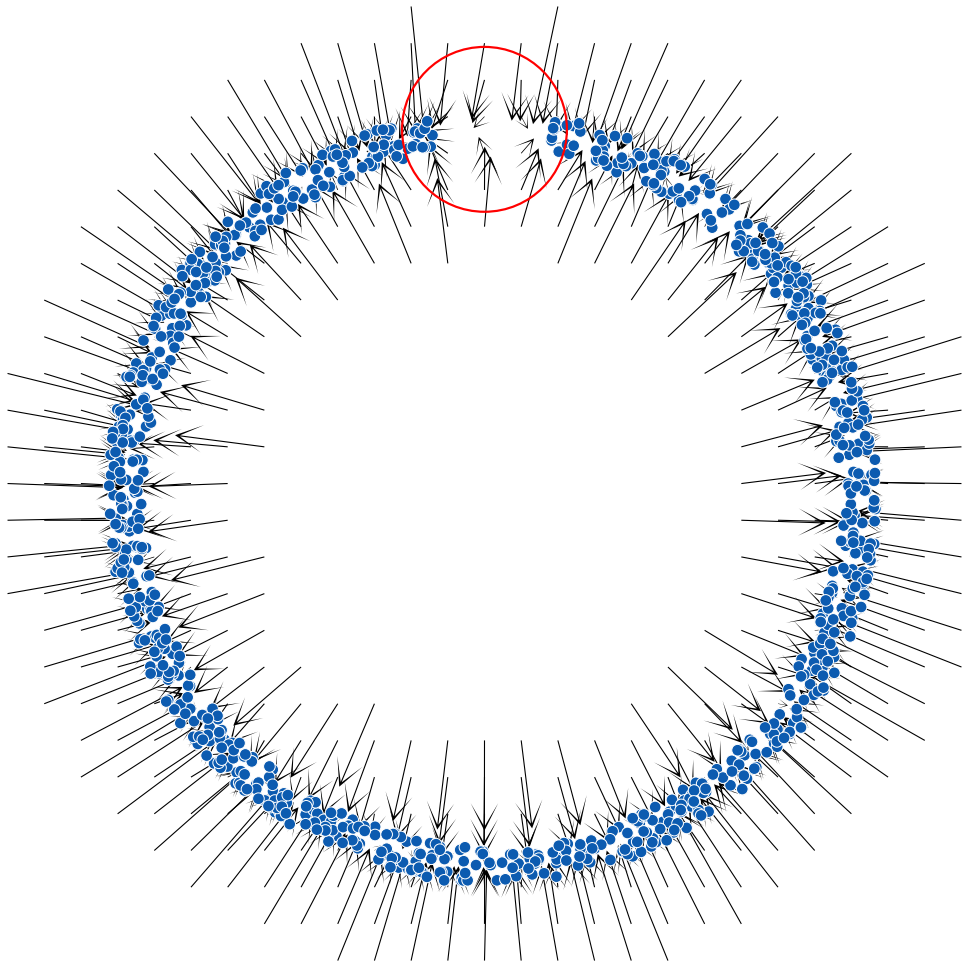
may be non-deterministic, possibly multi-modal.

If we train an autoencoder with the quadratic loss, the best reconstruction is

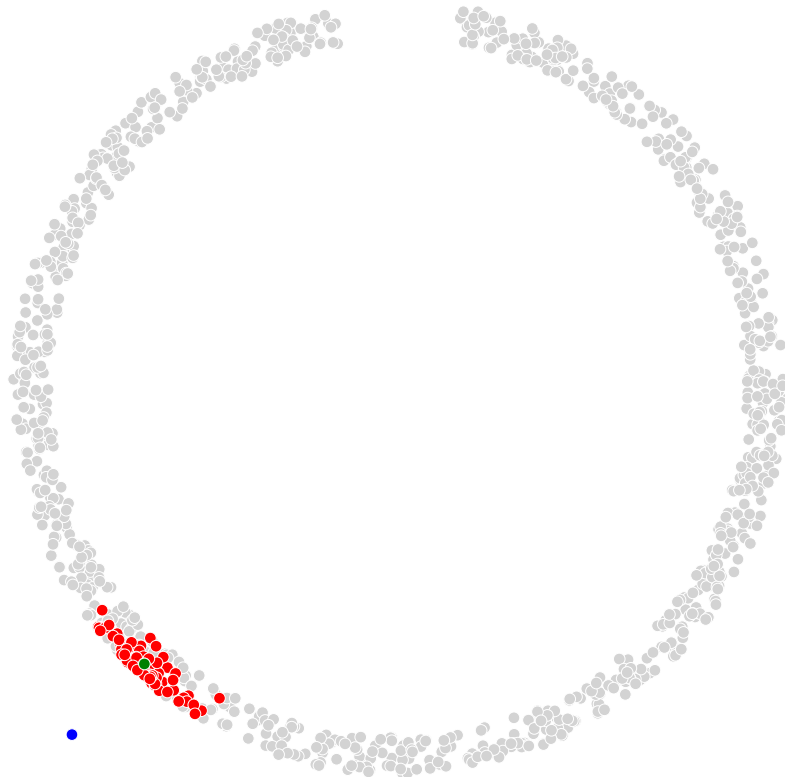
$$\phi(\tilde{X}) = \mathbb{E}\left[X \,\middle|\, \tilde{X}\right],$$

which may be very unlikely under $\mu_{X|\tilde{x}}$.

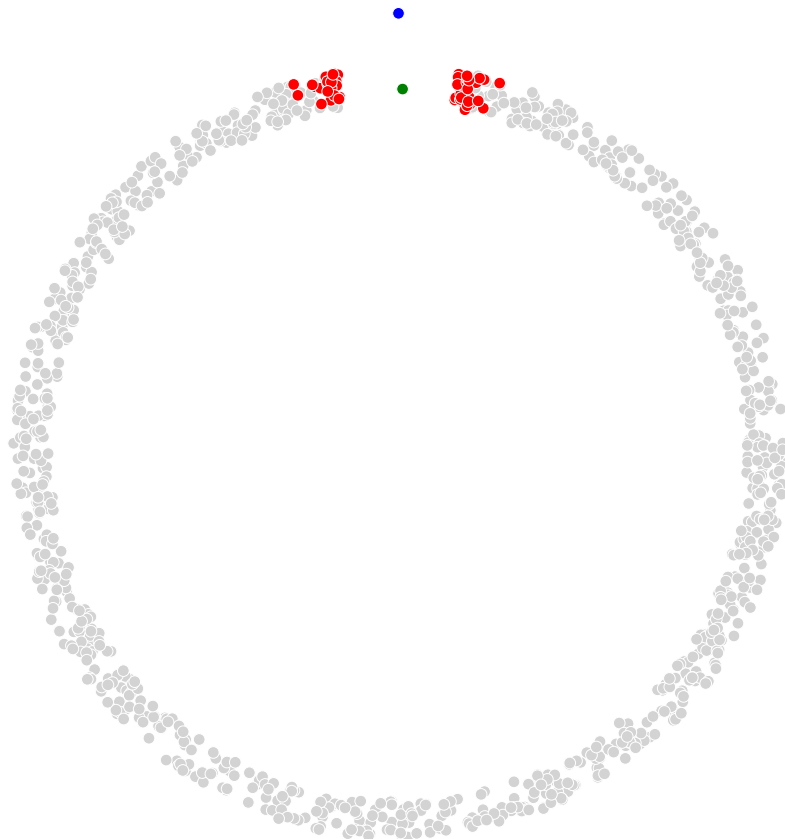




This phenomenon happens here in the area marked with the red circle. Points there can be noisy versions of points originally in either of the two extremities of the open circle, hence minimizing the MSE puts the denoised result in the middle of the opening, even though the density has no mass there.

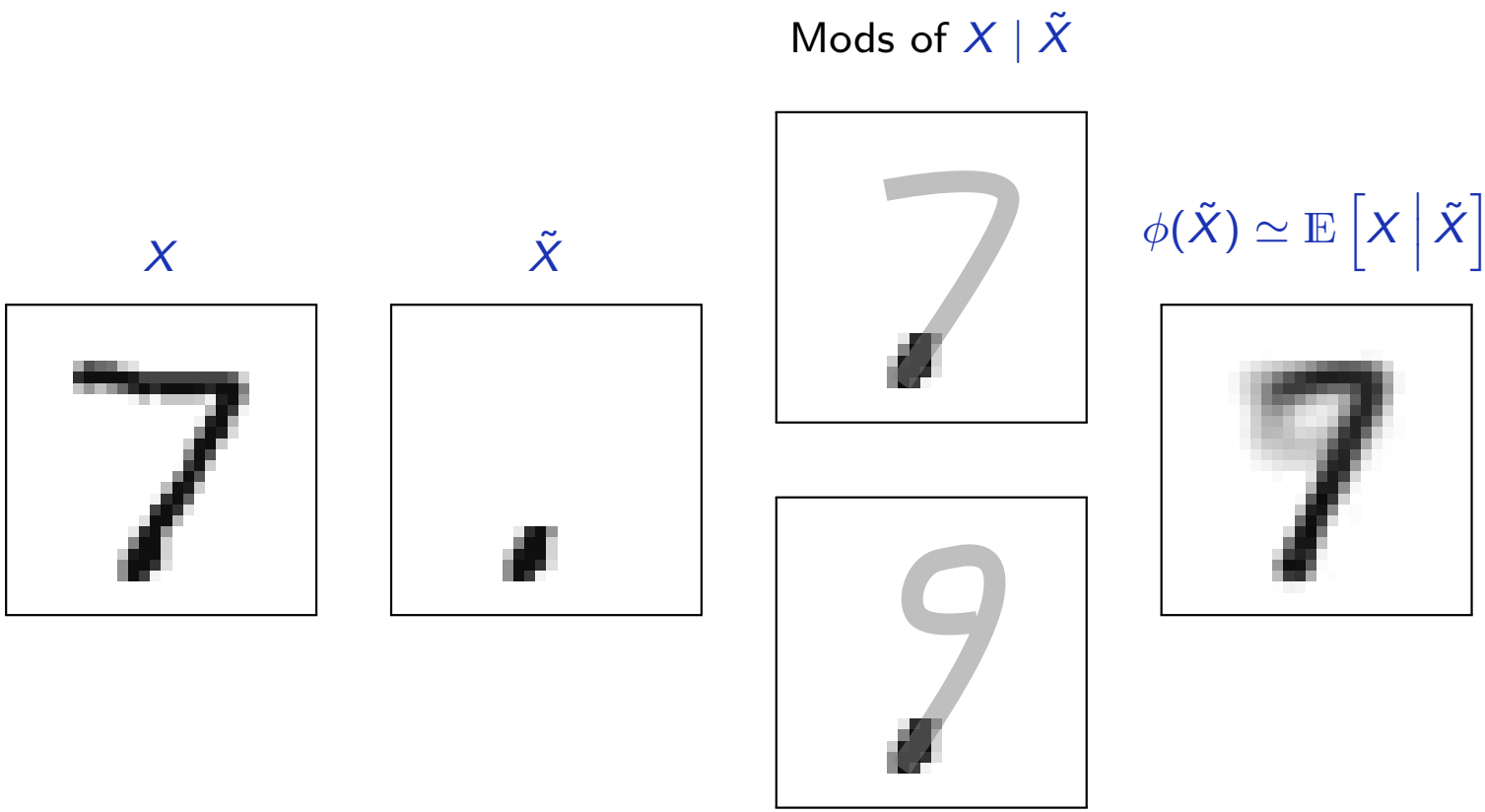


We can clarify this phenomenon given an $\tilde{x}$ (in blue) by sampling a large number of pairs $(X, \tilde{X})$, keeping only the Xs whose $\tilde{X}$ is very close to $\tilde{x}$, resulting in a sampling of $X|\tilde{X} = \tilde{x}$ (in red), whose mean $\mathbb{E}[X|\tilde{X} = \tilde{x}]$ (in green) minimizes the MSE.



We can clarify this phenomenon given an $\tilde{x}$ (in blue) by sampling a large number of pairs $(X, \tilde{X})$, keeping only the Xs whose $\tilde{X}$ is very close to $\tilde{x}$, resulting in a sampling of $X|\tilde{X} = \tilde{x}$ (in red), whose mean $\mathbb{E}[X|\tilde{X} = \tilde{x}]$ (in green) minimizes the MSE.

We observe the same phenomenon with very corrupted MNIST digits.



This can be mitigated by using in place of loss a second network that assesses if the output is realistic.

Such methods are called **adversarial** since the second network aims at spotting the mistakes of the first, and the first aims at fooling the second.

It can be combined with a stochastic denoiser that samples an X according to $X \mid \tilde{X}$ instead of computing a deterministic reconstruction.

We will come back to that in lecture 11.1. "Generative Adversarial Networks".

Noise2Noise

Denoising can be achieved without clean samples, if the noise is additive and unbiased. Consider ϵ and δ two unbiased and independent noises. We have

$$\begin{aligned}
&\mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-(X+\delta)\|^2\Big] \\
&\quad=\mathbb{E}\Big[\|(\phi(X+\epsilon;\theta)-X)-\delta\|^2\Big] \\
&\quad=\mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-X\|^2\Big]-2\mathbb{E}\Big[\delta^\top(\phi(X+\epsilon;\theta)-X)\Big]+\mathbb{E}\Big[\|\delta\|^2\Big] \\
&\quad=\mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-X\|^2\Big]-2\underbrace{\mathbb{E}\big[\delta\big]^\top}_{=0}\mathbb{E}\big[\phi(X+\epsilon;\theta)-X\big]+\mathbb{E}\Big[\|\delta\|^2\Big] \\
&\quad=\mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-X\|^2\Big]+\mathbb{E}\Big[\|\delta\|^2\Big].
\end{aligned}$$

Hence

$$\operatorname*{argmin}_\theta \mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-(X+\delta)\|^2\Big]=\operatorname*{argmin}_\theta \mathbb{E}\Big[\|\phi(X+\epsilon;\theta)-X\|^2\Big].$$

Using L_1 instead of L_2 estimates the median instead of the mean, and similarly is stable to noise that keeps the median unchanged.

Lehtinen et al. (2018)'s Noise2Noise approach uses this for image restoration, as many existing image generative processes induce an unbiased noise.

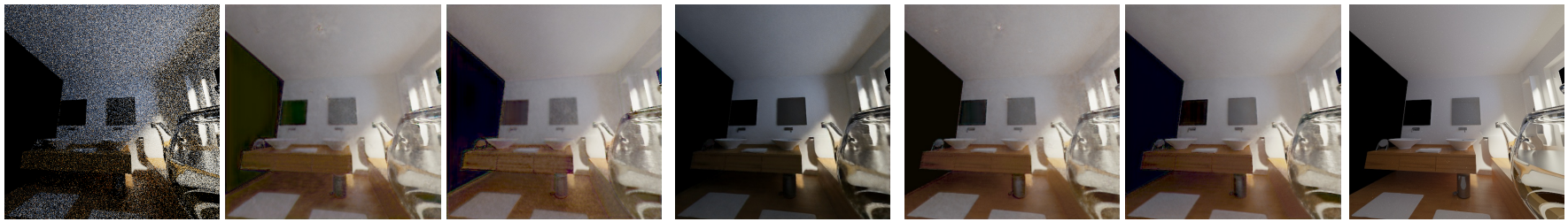
> *In many image restoration tasks, the expectation of the corrupted input data is the clean target that we seek to restore. Low-light photography is an example: a long, noise-free exposure is the average of short, independent, noisy exposures.*

> *Physically accurate renderings of virtual environments are most often generated through a process known as Monte Carlo path tracing. /.../ The Monte Carlo integrator is constructed such that the intensity of each pixel is the expectation of the random path sampling process, i.e., the sampling noise is zero-mean.*

(Lehtinen et al., 2018)

NAME	N_{out}	FUNCTION
INPUT	n	
ENC_CONV0	48	Convolution 3×3
ENC_CONV1	48	Convolution 3×3
POOL1	48	Maxpool 2×2
ENC_CONV2	48	Convolution 3×3
POOL2	48	Maxpool 2×2
ENC_CONV3	48	Convolution 3×3
POOL3	48	Maxpool 2×2
ENC_CONV4	48	Convolution 3×3
POOL4	48	Maxpool 2×2
ENC_CONV5	48	Convolution 3×3
POOL5	48	Maxpool 2×2
ENC_CONV6	48	Convolution 3×3
UPSAMPLE5	48	Upsample 2×2
CONCAT5	96	Concatenate output of POOL4
DEC_CONV5A	96	Convolution 3×3
DEC_CONV5B	96	Convolution 3×3
UPSAMPLE4	96	Upsample 2×2
CONCAT4	144	Concatenate output of POOL3
DEC_CONV4A	96	Convolution 3×3
DEC_CONV4B	96	Convolution 3×3
UPSAMPLE3	96	Upsample 2×2
CONCAT3	144	Concatenate output of POOL2
DEC_CONV3A	96	Convolution 3×3
DEC_CONV3B	96	Convolution 3×3
UPSAMPLE2	96	Upsample 2×2
CONCAT2	144	Concatenate output of POOL1
DEC_CONV2A	96	Convolution 3×3
DEC_CONV2B	96	Convolution 3×3
UPSAMPLE1	96	Upsample 2×2
CONCAT1	$96+n$	Concatenate INPUT
DEC_CONV1A	64	Convolution 3×3
DEC_CONV1B	32	Convolution 3×3
DEV_CONV1C	m	Convolution 3×3, linear act.

(Lehtinen et al., 2018)



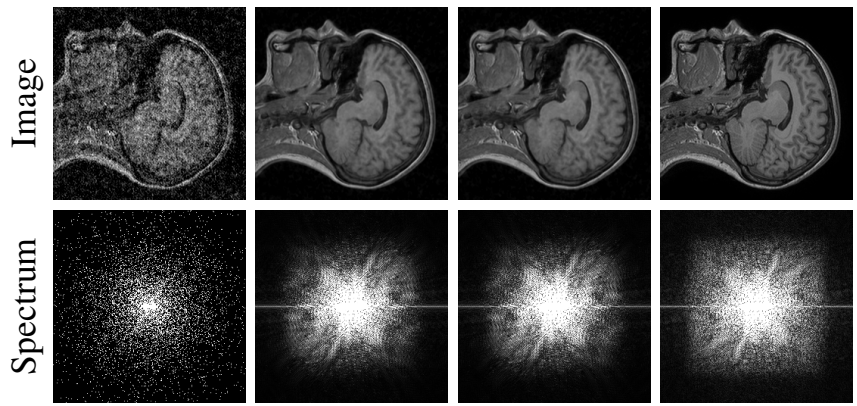
Input, 8 spp	L_2 with $\hat{x}, \hat{y}$	L_2 with $T(\hat{x}), \hat{y}$	L_2 with $T(\hat{x}), T(\hat{y})$	L_{HDR} with $\hat{x}, \hat{y}$	L_{HDR} with $T(\hat{x}), \hat{y}$	Reference, 32k spp
11.32 dB	25.46 dB	25.39 dB	15.50 dB	29.05 dB	30.09 dB	PSNR

Figure 6. Comparison of various loss functions for training a Monte Carlo denoiser with noisy target images rendered at 8 samples per pixel (spp). In this high-dynamic range setting, our custom relative loss L_{HDR} is clearly superior to L_2. Applying a non-linear tone map to the inputs is beneficial, while applying it to the target images skews the distribution of noise and leads to wrong, visibly too dark results.

(a) Input (64 spp), 23.93 dB (b) Noisy targets, 32.42 dB (c) Clean targets, 32.95 dB (d) Reference (131k spp)

Figure 7. Denoising a Monte Carlo rendered image. (a) Image rendered with 64 samples per pixel. (b) Denoised 64 spp input, trained using 64 spp targets. (c) Same as previous, but trained on clean targets. (d) Reference image rendered with 131 072 samples per pixel. PSNR values refer to the images shown here, see text for averages over the entire validation set.

(Lehtinen et al., 2018)



(a) Input 18.93 dB (b) Noisy trg. 29.77 dB (c) Clean trg. 29.81 dB (d) Reference

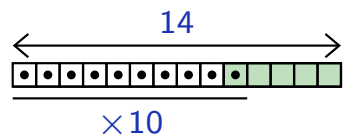
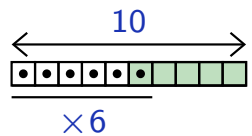
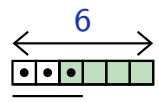
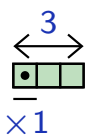
Figure 9. MRI reconstruction example. (a) Input image with only 10% of spectrum samples retained and scaled by $1/p$. (b) Reconstruction by a network trained with noisy target images similar to the input image. (c) Same as previous, but training done with clean target images similar to the reference image. (d) Original, uncorrupted image. PSNR values refer to the images shown here, see text for averages over the entire validation set.

(Lehtinen et al., 2018)

Super-resolution

A special case of denoising is to increase an image resolution. We use an encoder/decoder whose encoder's input is smaller than the decoder's output.

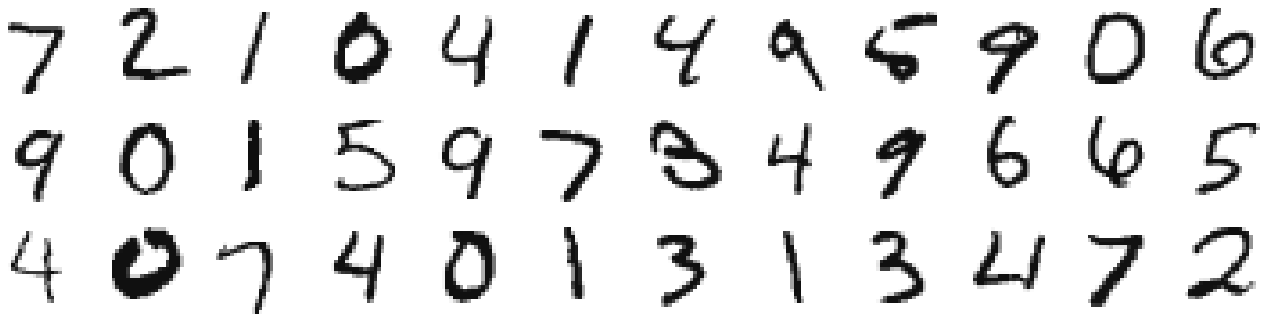
Encoder

Tensor sizes / operations	
$1 \times 14 \times 14$	
`nn.Conv2d(1, 32, kernel_size=5, stride=1)`	
$32 \times 10 \times 10$	
`nn.Conv2d(32, 32, kernel_size=5, stride=1)`	
$32 \times 6 \times 6$	
`nn.Conv2d(32, 32, kernel_size=4, stride=1)`	
$32 \times 3 \times 3$	
`nn.Conv2d(32, 32, kernel_size=3, stride=1)`	
$32 \times 1 \times 1$	

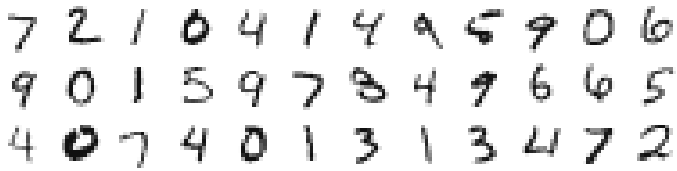
```
MNISTUpscaler(
  (encoder): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
    (3): ReLU(inplace=True)
    (4): Conv2d(32, 32, kernel_size=(4, 4), stride=(1, 1))
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
  )
  (decoder): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(4, 4), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(2, 2))
    (3): ReLU(inplace=True)
    (4): ConvTranspose2d(32, 32, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
    (7): ReLU(inplace=True)
    (8): ConvTranspose2d(32, 1, kernel_size=(5, 5), stride=(1, 1))
  )
)
```

```
for original in train_input.split(batch_size):
    input = F.avg_pool2d(original, kernel_size = 2)
    output = model(input)
    loss = (output - original).pow(2).sum() / output.size(0)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

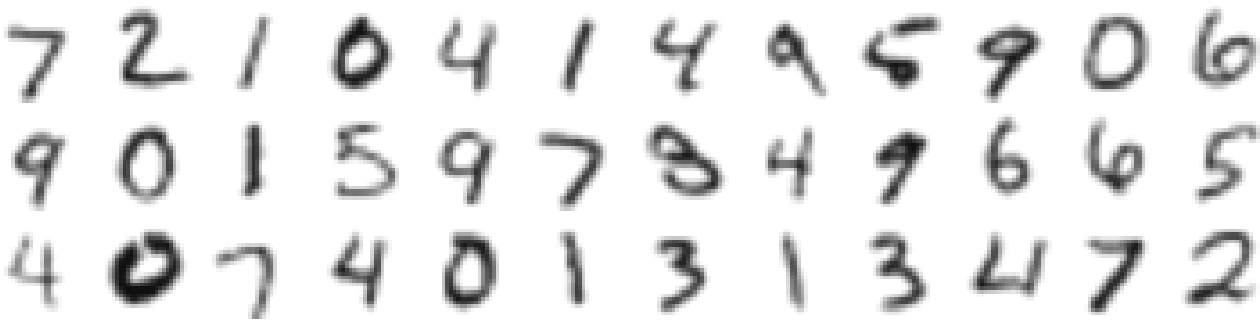
Original



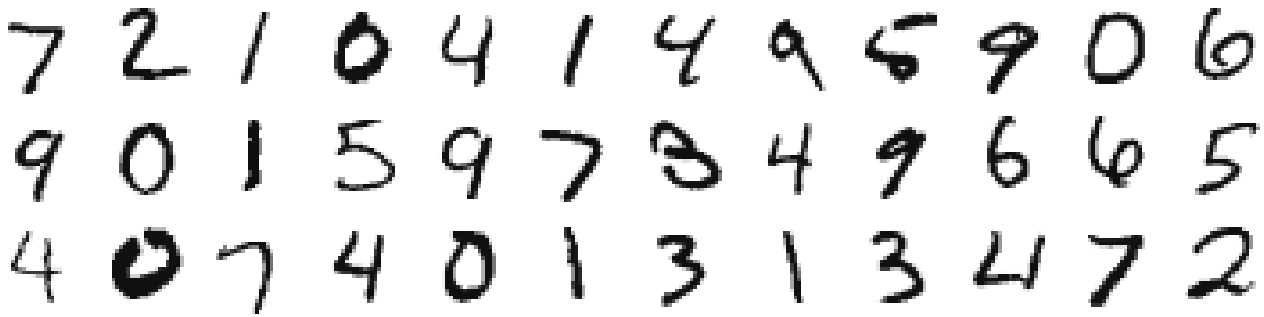
Input



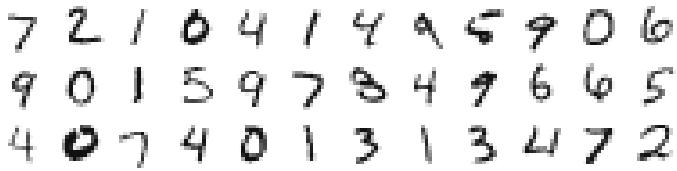
Bilinear interpolation



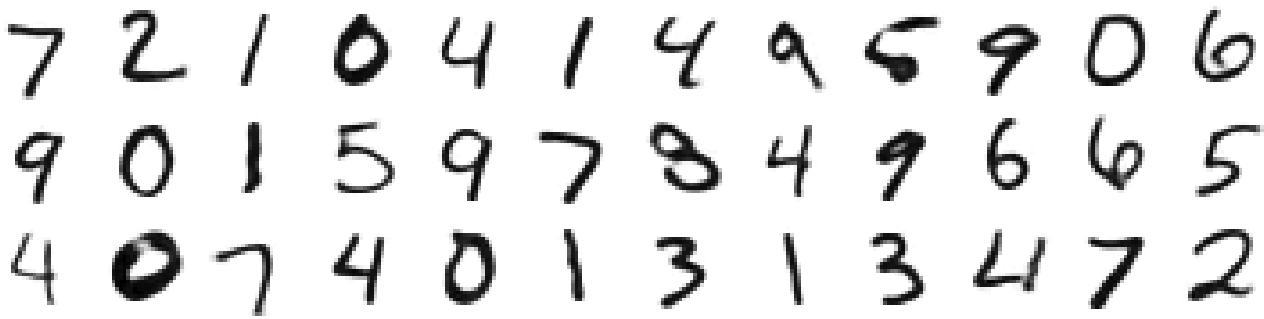
Original



Input

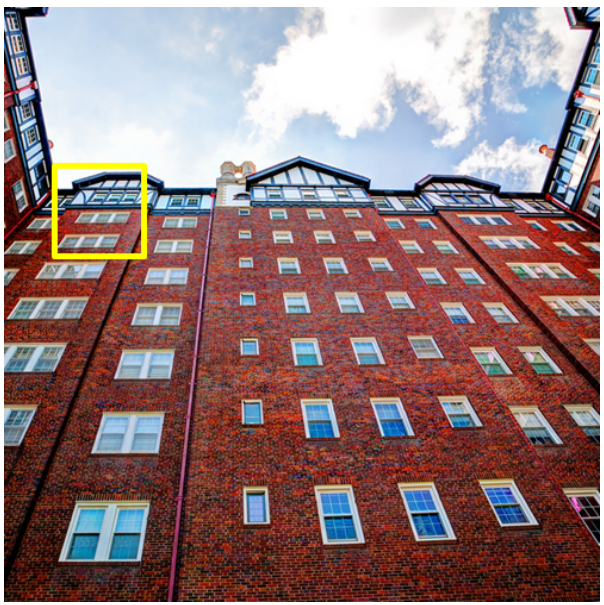


Autoencoder output



Lim et al. (2017) use two different resnets.

	EDSR	MDSR
Nb blocks	32	80
Channels	256	64
Nb parameters	43M	8M



img034 from Urban100 [10]

HR
(PSNR / SSIM)

Bicubic
(21.41 dB / 0.4810)

A+ [27]
(22.21 dB / 0.5408)

SRCNN [4]
(22.33 dB / 0.5461)

VDSR [11]
(22.62 dB / 0.5657)

SRResNet [14]
(23.14 dB / 0.5891)

EDSR+ (Ours)
(23.48 dB / 0.6048)

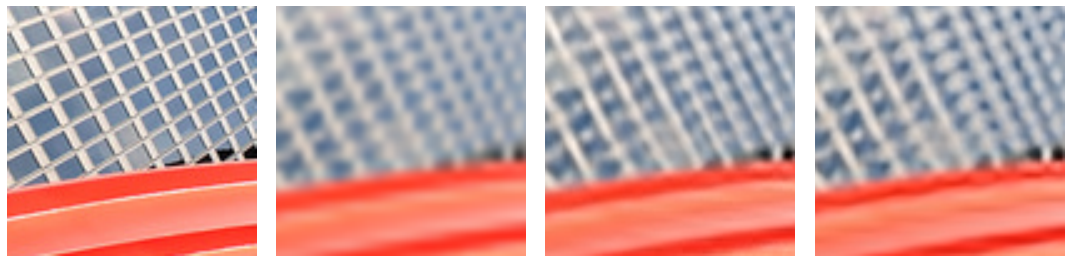
MDSR+ (Ours)
(23.46 dB / 0.6039)

(Lim et al., 2017)

Lim et al. (2017) use two different resnets.

	EDSR	MDSR
Nb blocks	32	80
Channels	256	64
Nb parameters	43M	8M

img062 from Urban100 [10]

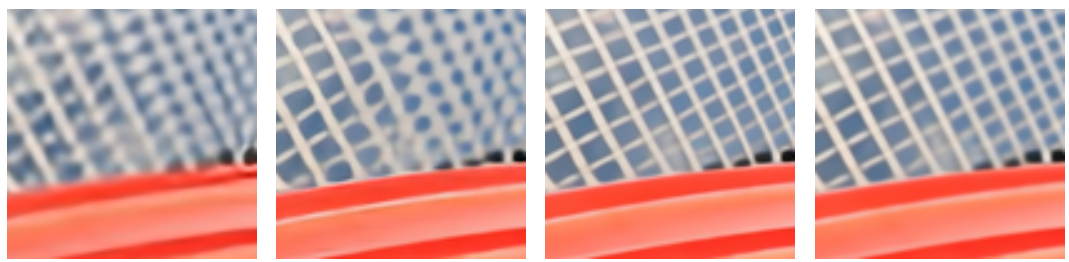


HR
(PSNR / SSIM)

Bicubic
(19.82 dB / 0.6471)

A+ [27]
(20.43 dB 0.7145)

SRCNN [4]
(20.61 dB / 0.7218)



VDSR [11]
(20.75 dB / 0.7504)

SRResNet [14]
(21.70 dB / 0.8054)

EDSR+ (Ours)
(22.70 dB / 0.8537)

MDSR+ (Ours)
(22.66 dB / 0.8508)

(Lim et al., 2017)

Lim et al. (2017) use two different resnets.

	EDSR	MDSR
Nb blocks	32	80
Channels	256	64
Nb parameters	43M	8M

0869 from DIV2K [26]

HR
(PSNR / SSIM)

Bicubic
(22.66 dB / 0.8025)

A+ [27]
(23.10 dB / 0.8251)

SRCNN [4]
(23.14 dB / 0.8280)

VDSR [11]
(23.36 dB / 0.8365)

SRResNet [14]
(23.71 dB / 0.8485)

EDSR+ (Ours)
(23.89 dB / 0.8563)

MDSR+ (Ours)
(23.90 dB / 0.8558)

(Lim et al., 2017)

Autoencoders as self-training

Vincent et al. (2010) interpret training the autoencoder as maximizing the mutual information between the input and the latent states.

Let X be a sample, $Z = f(X; \theta)$ its latent representation, and $q(x, z)$ the distribution of (X, Z).

We have

$$\operatorname*{argmax}_{\theta} \ \mathbb{I}(X; Z) = \operatorname*{argmax}_{\theta} \ \mathbb{E}_{q(X,Z)} \Big[\log q(X \mid Z) \Big].$$

However, there is no expression of $q(X \mid Z)$ in any reasonable setup.

For any distribution p we have

$$\mathbb{E}_{q(X,Z)}\Big[\log q(X \mid Z)\Big] \geq \mathbb{E}_{q(X,Z)}\Big[\log p(X \mid Z)\Big].$$

So we can in particular try to find a "good p", so that the left term is a good approximation of the right one.

If we consider the following model for p

$$p(\cdot \mid Z = z) = \mathcal{N}(g(z;\eta), \sigma)$$

where g is deterministic and σ fixed, we get

$$\mathbb{E}_{q(X,Z)}\Big[\log p(X \mid Z)\Big] = -\frac{1}{2\sigma^2}\mathbb{E}_{q(X,Z)}\left[\|X - g(f(X;\theta);\eta)\|^2\right] + k.$$

If optimizing η makes the bound tight, the final loss is the reconstruction error

$$\operatorname*{argmax}_{\theta} \mathbb{I}(X;Z) \simeq \operatorname*{argmin}_{\theta}\left(\min_{\eta} \frac{1}{N}\sum_{n=1}^{N}\|x_n - g(f(x_n;\theta);\eta)\|^2\right).$$

This abstract view of the encoder as "maximizing information" justifies its use to build generic encoding layers.

In the perspective of building a good feature representation, just retaining information is not enough, otherwise the identity would be a good choice.

In their work, Vincent et al. consider a denoising auto-encoder, which makes the model retain information about structures beyond local noise.

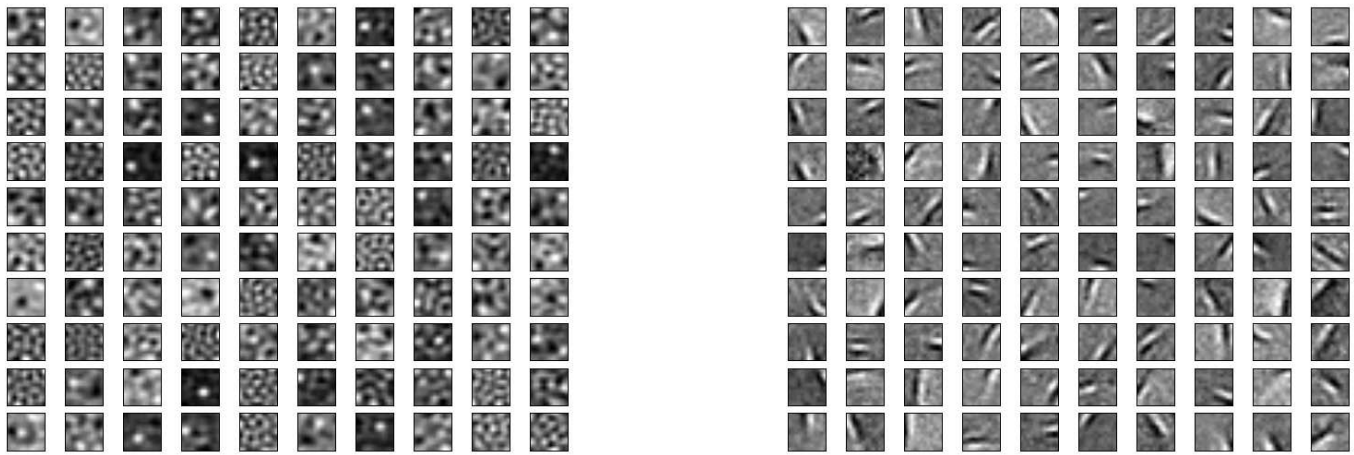


Figure 6: Weight decay vs. Gaussian noise. We show typical filters learnt from natural image patches in the over-complete case (200 hidden units). *Left:* regular autoencoder with weight decay. We tried a wide range of weight-decay values and learning rates: filters never appeared to capture a more interesting structure than what is shown here. Note that some local blob detectors are recovered compared to using no weight decay at all (Figure 5 right). *Right:* a denoising autoencoder with additive Gaussian noise ($\sigma = 0.5$) learns Gabor-like local oriented edge detectors. Clearly the filters learnt are qualitatively very different in the two cases.

(Vincent et al., 2010)

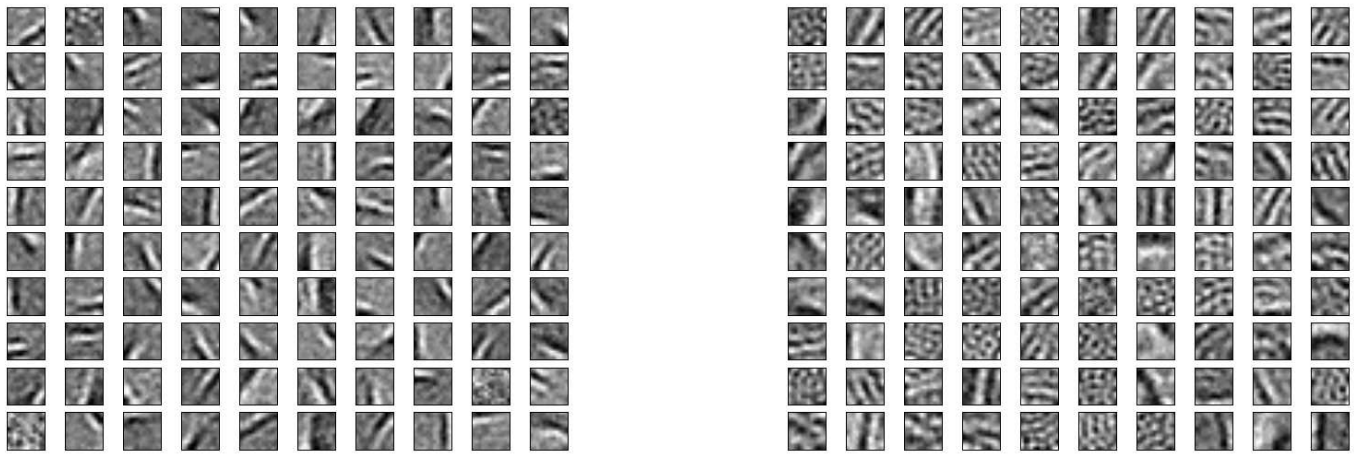
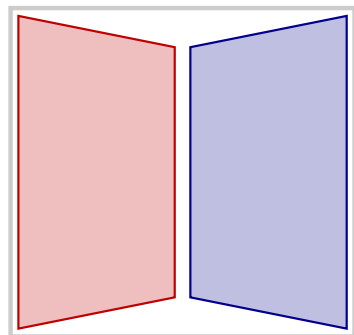


Figure 7: Filters obtained on natural image patches by denoising autoencoders using other noise types. *Left:* with 10% salt-and-pepper noise, we obtain oriented Gabor-like filters. They appear slightly less localized than when using Gaussian noise (contrast with Figure 6 right). *Right:* with 55% zero-masking noise we obtain filters that look like oriented gratings. For the three considered noise types, denoising training appears to learn filters that capture meaningful natural image statistics structure.

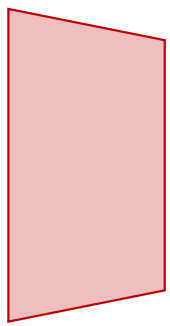
(Vincent et al., 2010)

Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.

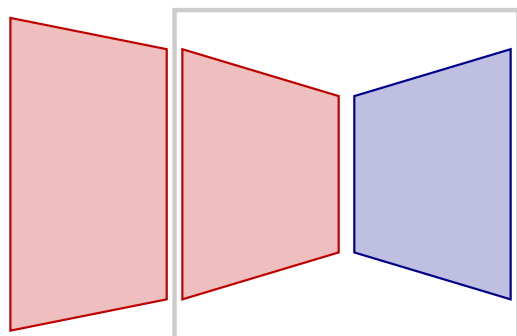


Autoencoder

Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.

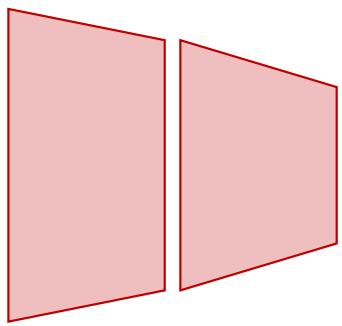


Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.

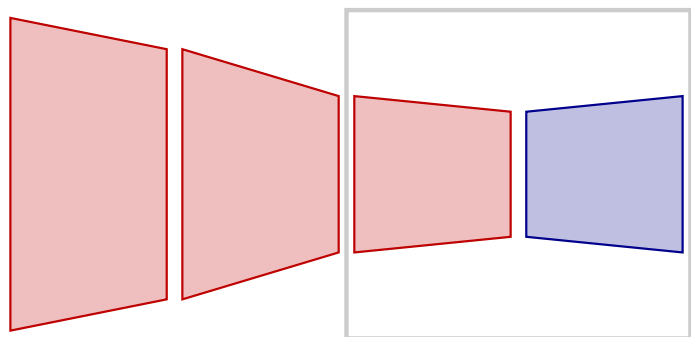


Autoencoder

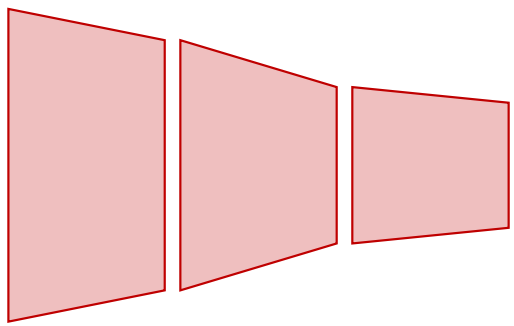
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



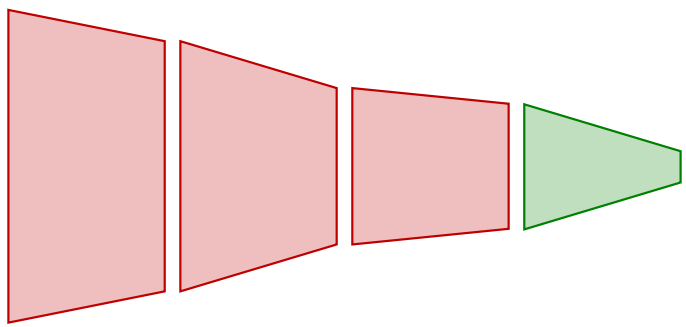
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



A final classifying layer is added and the full structure can be fine-tuned.

This approach, and others in the same spirit (Hinton et al., 2006), were seen as strategies to complement gradient-descent for building deep nets.

Data Set	SVM_{rbf}	DBN-1	SAE-3	DBN-3	SDAE-3 (ν)
MNIST	**1.40**±0.23	**1.21**±0.21	**1.40**±0.23	**1.24**±0.22	**1.28**±0.22 (25%)
basic	**3.03**±0.15	3.94±0.17	3.46±0.16	3.11±0.15	**2.84**±0.15 (10%)
rot	11.11±0.28	14.69±0.31	10.30±0.27	10.30±0.27	**9.53**±0.26 (25%)
bg-rand	14.58±0.31	9.80±0.26	11.28±0.28	**6.73**±0.22	10.30±0.27 (40%)
bg-img	22.61±0.37	**16.15**±0.32	23.00±0.37	**16.31**±0.32	**16.68**±0.33 (25%)
bg-img-rot	55.18±0.44	52.21±0.44	51.93±0.44	47.39±0.44	**43.76**±0.43 (25%)
rect	**2.15**±0.13	4.71±0.19	2.41±0.13	2.60±0.14	**1.99**±0.12 (10%)
rect-img	24.04±0.37	23.69±0.37	24.05±0.37	22.50±0.37	**21.59**±0.36 (25%)
convex	19.13±0.34	19.92±0.35	**18.41**±0.34	**18.63**±0.34	**19.06**±0.34 (10%)
tzanetakis	**14.41**±2.18	18.07±1.31	**16.15**±1.95	18.38±1.64	**16.02**±1.04(0.05)

(Vincent et al., 2010)

Deep learning

7.4. Variational autoencoders

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Coming back to generating a signal, instead of training an autoencoder and modeling the distribution of Z, we can try an alternative approach:

Impose a distribution for Z and then train a decoder g so that $g(Z)$ matches the training data.

We consider two distributions:

- p is the distribution on $\mathscr{X} \times \mathbb{R}^d$ of a pair (X, Z) composed of an encoding state $Z \sim \mathcal{N}(0, I)$ and the output of the decoder g on it.

- q is the distribution on $\mathscr{X} \times \mathbb{R}^d$ of a pair (X, Z) composed of a sample X taken from the data distribution and the output of the encoder on it,

Our goal is that $p(X)$ mimics the data-distribution $q(X)$, that is to find g that maximizes the log-likelihood

$$\frac{1}{N} \sum_n \log p(x_n) = \hat{\mathbb{E}}_{q(X)} \Big[\log p(X) \Big].$$

However, while we can sample z and compute $g(z)$ for complicated gs, **we cannot compute $p(x)$ for a given x**, and even less compute its derivatives.

The **Variational Autoencoder** proposed by Kingma and Welling (2013) relies on a tractable approximation of this log-likelihood.

Note that their framework involves **stochastic** encoder f, and decoder g, whose outputs depend on both their inputs and additional randomness.

Remember that $q(X)$ is the data distribution, and $f(x) \sim q(Z \mid X = x)$.

We want to maximize

$$\mathbb{E}_{q(X)}\Big[\log p(X)\Big],$$

and it can be shown that

$$\log p(X = x) \geq \underbrace{\mathbb{E}_{q(Z|X=x)}\Big[\log p(X = x \mid Z)\Big] - \mathbb{D}_{\mathsf{KL}}(q(Z \mid X = x) \,\|\, p(Z))}_{\text{“Evidence lower bound” (ELBO)}}.$$

So it makes sense to maximize

$$\mathbb{E}_{q(X,Z)}\Big[\log p(X \mid Z)\Big] - \mathbb{E}_{q(X)}\Big[\mathbb{D}_{\mathsf{KL}}(q(Z \mid X) \,\|\, p(Z))\Big].$$

So the final loss is

$$\mathscr{L} = \mathbb{E}_{q(X)}\Big[\mathbb{D}_{\mathsf{KL}}(q(Z \mid X) \,\|\, p(Z))\Big] - \mathbb{E}_{q(X,Z)}\Big[\log p(X \mid Z)\Big],$$

with

- $q(X)$ is the data distribution
- $p(Z) = \mathcal{N}(0, I)$.

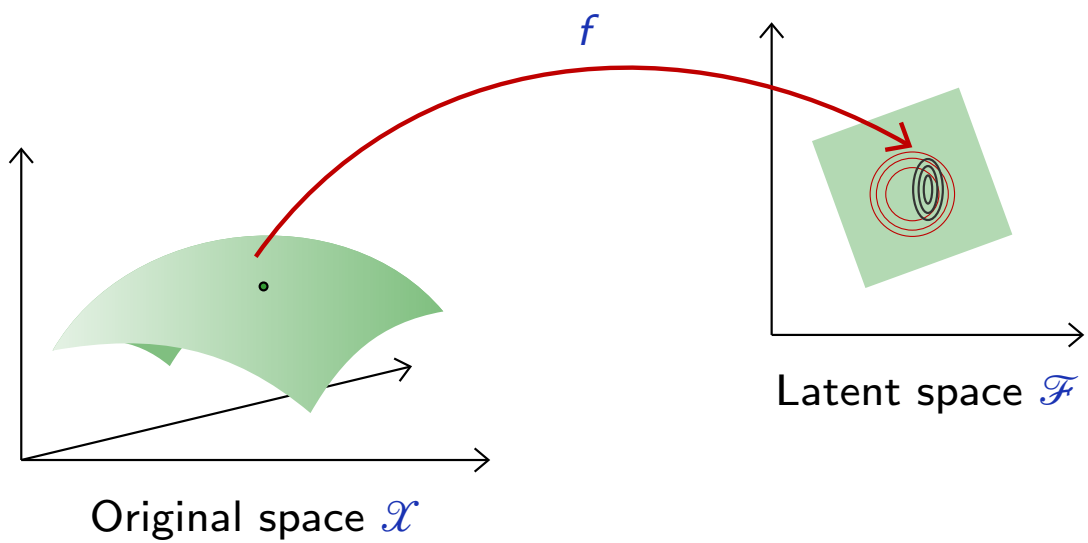
Kingma and Welling propose that both the encoder f and decoder g map to a Gaussian with diagonal covariance. Hence they map to twice the dimension (e.g. $f(x) = (\mu^f(x), \sigma^f(x))$) and

- $q(Z \mid X = x) \sim \mathcal{N}(\mu^f(x), \mathrm{diag}(\sigma^f(x)))$
- $p(X \mid Z = z) \sim \mathcal{N}(\mu^g(z), \mathrm{diag}(\sigma^g(z)))$.

The first term of $\mathscr{L}$ is the average of

$$\mathbb{D}_{\mathrm{KL}}\big(\underbrace{q(Z \mid X = x)}_{\mathcal{N}(\mu^f(x),\sigma^f(x))} \,\|\, \underbrace{p(Z)}_{\mathcal{N}(0,I)}\big) = -\frac{1}{2}\sum_d \left(1 + 2\log\sigma_d^f(x) - \left(\mu_d^f(x)\right)^2 - \left(\sigma_d^f(x)\right)^2\right).$$

over the x_ns.



The first term of $\mathscr{L}$ is the average of

$$\mathbb{D}_{\mathrm{KL}}\big(\underbrace{q(Z \mid X = x)}_{\mathcal{N}(\mu^f(x),\sigma^f(x))} \,\|\, \underbrace{p(Z)}_{\mathcal{N}(0,I)}\big) = -\frac{1}{2}\sum_d \Big(1 + 2\log\sigma_d^f(x) - \big(\mu_d^f(x)\big)^2 - \big(\sigma_d^f(x)\big)^2\Big).$$

over the x_ns.

This can be implemented as

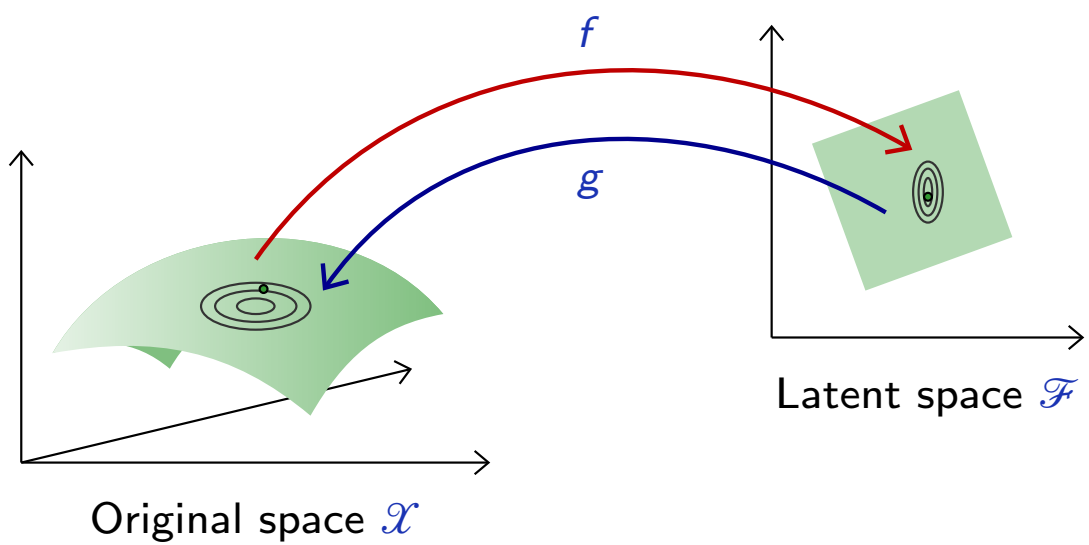
```
param_f = model.encode(input)
mu_f, logvar_f = param_f.split(param_f.size(1)//2, 1)

kl = - 0.5 * (1 + logvar_f - mu_f.pow(2) - logvar_f.exp())
kl_loss = kl.sum() / input.size(0)
```

As Kingma and Welling (2013), we use a constant variance of 1 for the decoder, so the second term of $\mathscr{L}$ becomes the average of

$$-\log p(X = x \mid Z = z) = \frac{1}{2}\sum_d (x_d - \mu_d^g(z))^2 + \text{cst}$$

over the x_n, with one z_n sampled for each, i.e.

$$z_n \sim \mathcal{N}\left(\mu^f(x_n), \sigma^f(x_n)\right), \quad n = 1, \ldots, N.$$



As Kingma and Welling (2013), we use a constant variance of 1 for the decoder, so the second term of $\mathscr{L}$ becomes the average of

$$-\log p(X = x \mid Z = z) = \frac{1}{2}\sum_d (x_d - \mu_d^g(z))^2 + \text{cst}$$

over the x_n, with one z_n sampled for each, i.e.

$$z_n \sim \mathcal{N}\left(\mu^f(x_n), \sigma^f(x_n)\right), \quad n = 1, \ldots, N.$$

This can be implemented as

```
std_f = torch.exp(0.5 * logvar_f)
z = torch.randn_like(mu_f) * std_f + mu_f
output = model.decode(z)

fit = 0.5 * (output - input).pow(2)
fit_loss = fit.sum() / input.size(0)
```

We had for the standard autoencoder

```
z = model.encode(input)
output = model.decode(z)
loss = 0.5 * (output - input).pow(2).sum() / input.size(0)
```

and putting everything together we get for the VAE

```
param_f = model.encode(input)
mu_f, logvar_f = param_f.split(param_f.size(1)//2, 1)

kl = - 0.5 * (1 + logvar_f - mu_f.pow(2) - logvar_f.exp())
kl_loss = kl.sum() / input.size(0)

std_f = torch.exp(0.5 * logvar_f)
z = torch.randn_like(mu_f) * std_f + mu_f
output = model.decode(z)

fit = 0.5 * (output - input).pow(2)
fit_loss = fit.sum() / input.size(0)

loss = kl_loss + fit_loss
```

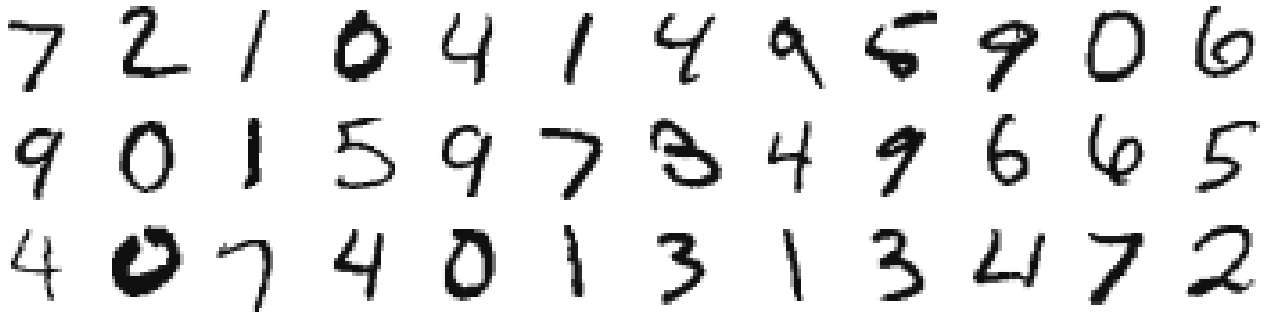
During inference we do not sample, and instead use μ^f and μ^g as prediction.

Note in particular the **re-parameterization trick**:

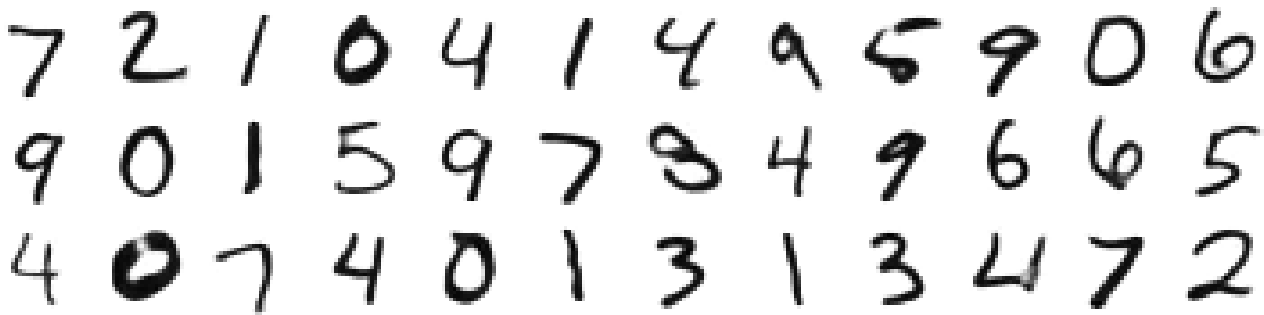
```
z = torch.randn_like(mu_f) * std_f + mu_f
output = model.decode(z)
```

Implementing the sampling of z that way allows to compute the gradient w.r.t f's parameters without any particular property of `normal_()`.

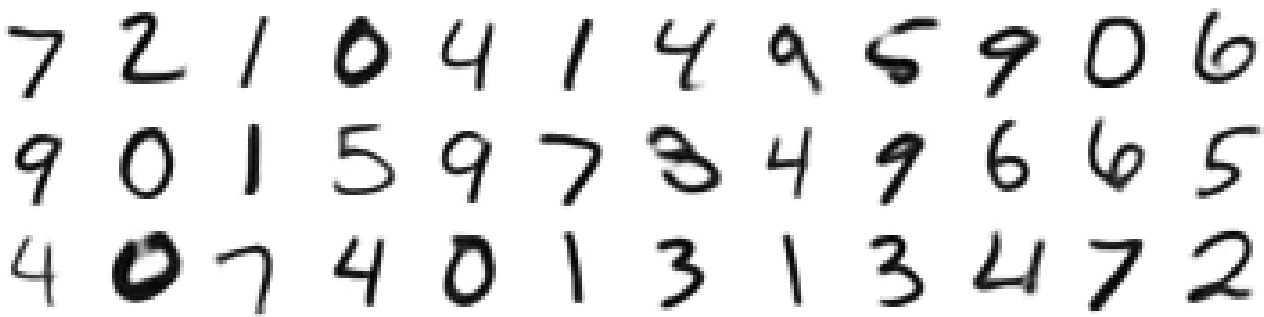
Original



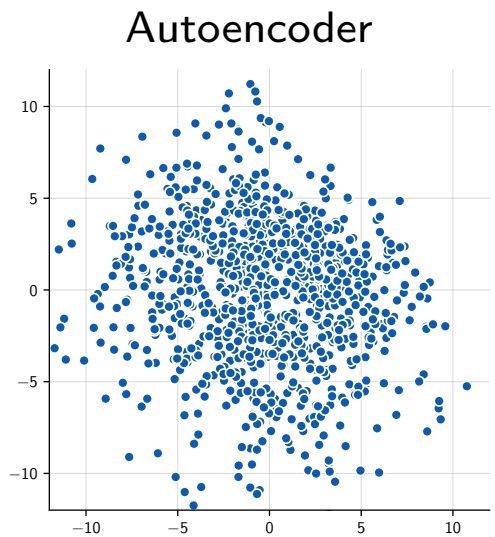
Autoencoder reconstruction ($d = 32$)



Variational Autoencoder reconstruction ($d = 32$)

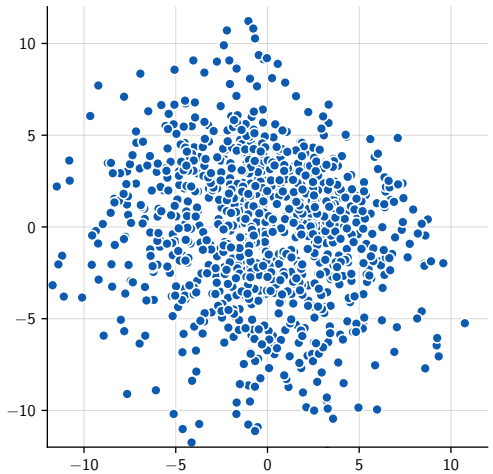


We can look at two latent features to check that they are Normal for the VAE.

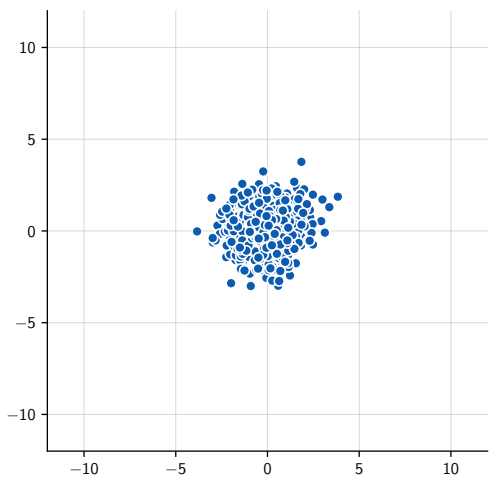


We can look at two latent features to check that they are Normal for the VAE.

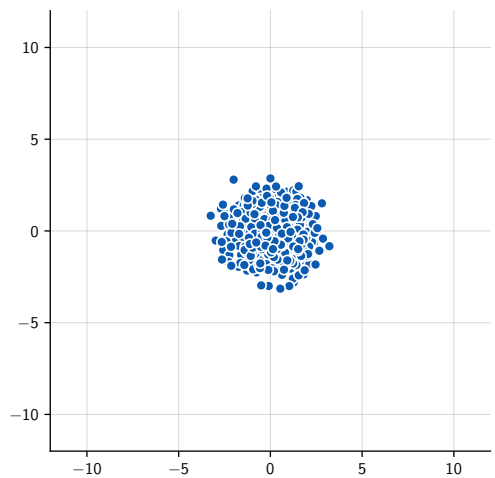
Autoencoder



Variational autoencoder

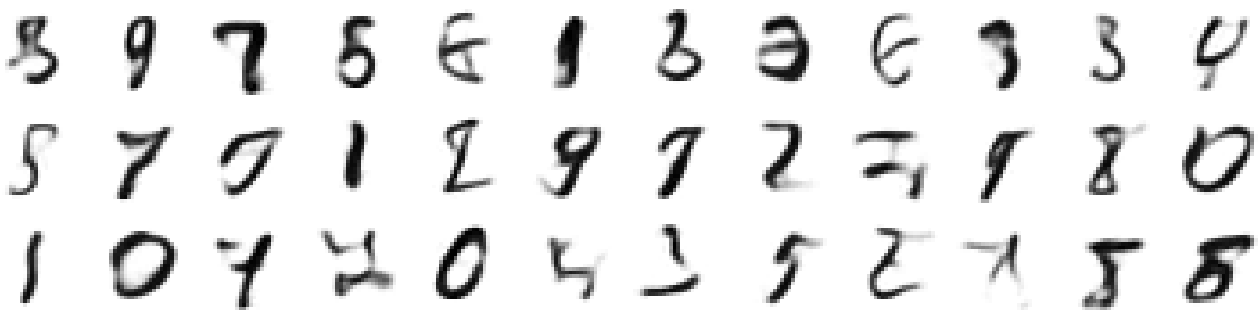


$\mathcal{N}(0, 1)$



Autoencoder sampling ($d = 32$)

Variational Autoencoder sampling ($d = 32$)

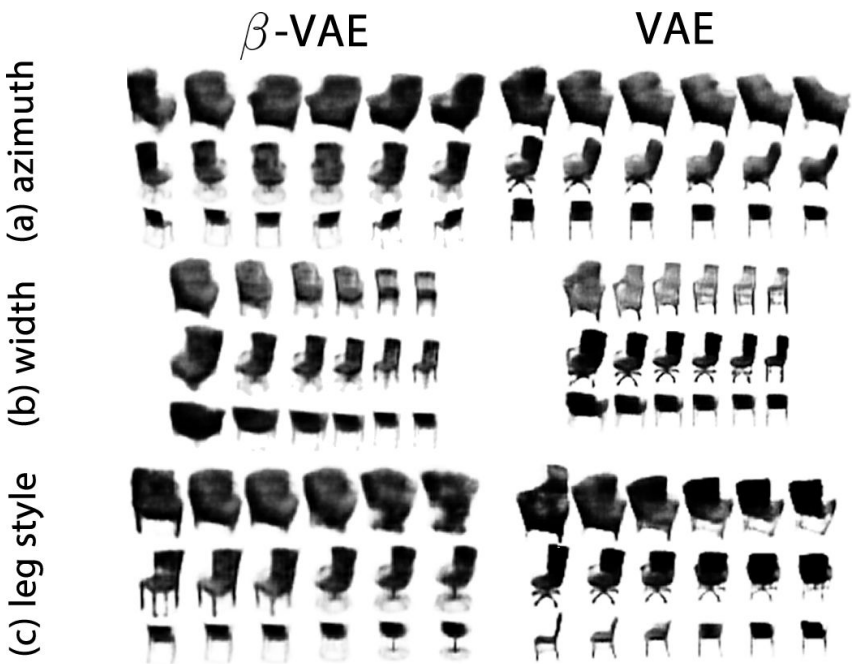


Making the embedding $\sim \mathcal{N}(0, 1)$, often results in "disentangled" representations.

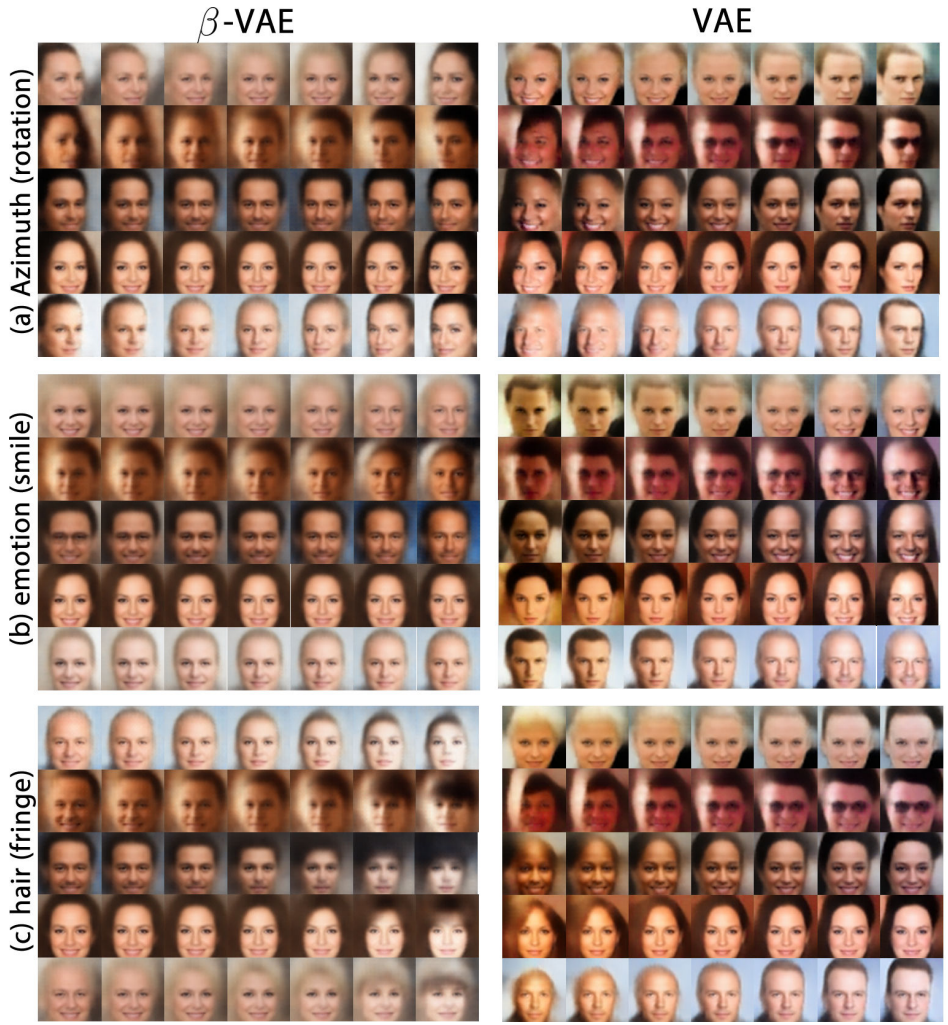
This effect can be reinforced with a greater weight of the KL term

$$\mathscr{L} = \beta \, \mathbb{E}_{q(X)}\Big[\mathbb{D}_{\mathsf{KL}}(q(Z \mid X) \,\|\, p(Z))\Big] - \mathbb{E}_{q(X,Z)}\Big[\log p(X \mid Z)\Big],$$

resulting in the β-VAE proposed by Higgins et al. (2017).



(Higgins et al., 2017)



(Higgins et al., 2017)

The end