

# EE-559 – Deep learning

## 9.1. Transposed convolutions

François Fleuret

<https://fleuret.org/ee559/>

Mon Feb 18 13:36:16 UTC 2019

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

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

Generative processes that consist of optimizing the input rely on back-propagation to expand the signal from a low-dimension representation to the high-dimension signal space.

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

Generative processes that consist of optimizing the input rely on back-propagation to expand the signal from a low-dimension representation to the high-dimension signal space.

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  $\kappa$

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

Consider a 1d convolution with a kernel  $\kappa$

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

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

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

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

$$y = x * \kappa$$

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**.

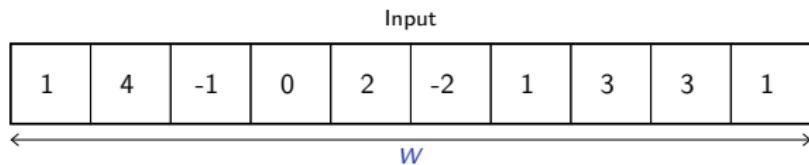
$$\begin{pmatrix} \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\ 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\ 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\ 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\ 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \end{pmatrix}^T = \begin{pmatrix} \kappa_1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

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**.

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

While 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

Input

1	4	-1	0	2	-2	1	3	3	1
---	---	----	---	---	----	---	---	---	---

$\xleftarrow{\quad} \textcolor{blue}{w} \xrightarrow{\quad}$

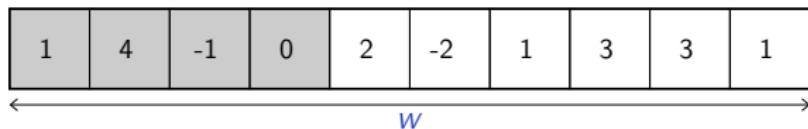
Kernel

1	2	0	-1
---	---	---	----

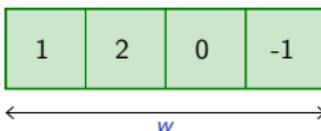
$\xleftarrow{\quad} \textcolor{blue}{w} \xrightarrow{\quad}$

## Convolution layer

Input

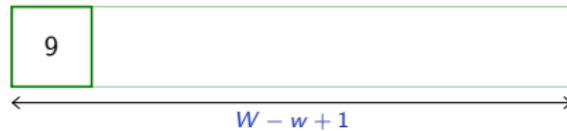


$w$



$w$

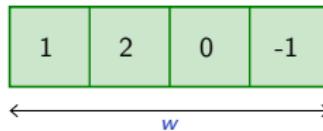
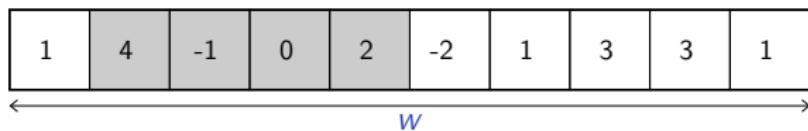
Output



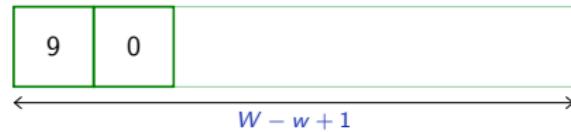
$W - w + 1$

## Convolution layer

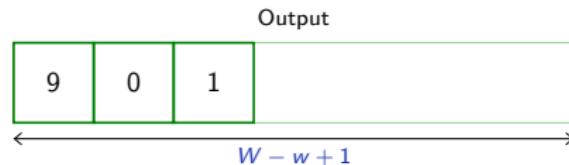
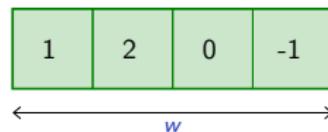
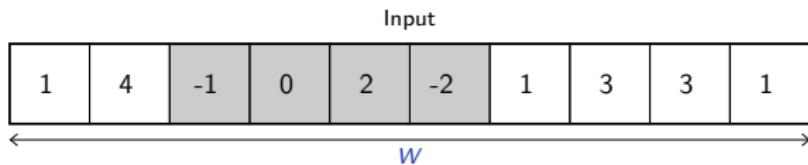
Input



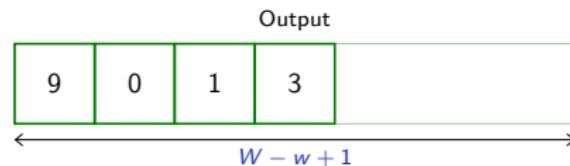
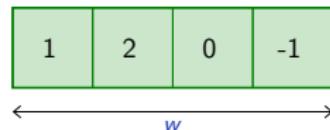
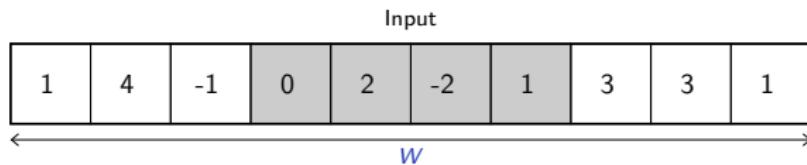
Output



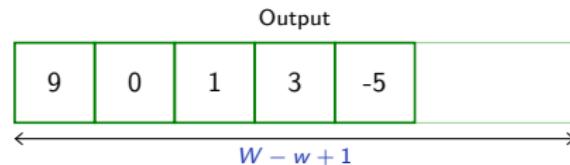
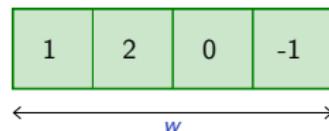
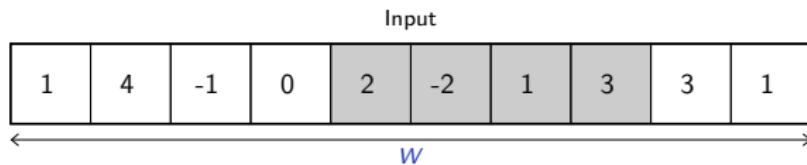
## Convolution layer



## Convolution layer

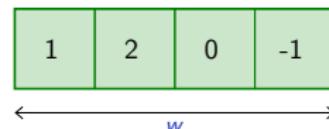
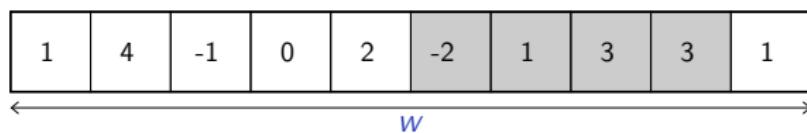


## Convolution layer

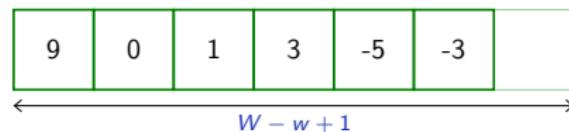


## Convolution layer

Input



Output



## Convolution layer

Input

1	4	-1	0	2	-2	1	3	3	1
---	---	----	---	---	----	---	---	---	---

$\xleftarrow{\quad} \textcolor{blue}{W} \xrightarrow{\quad}$

1	2	0	-1
---	---	---	----

$\xleftarrow{\quad} \textcolor{blue}{w} \xrightarrow{\quad}$

Output

9	0	1	3	-5	-3	6
---	---	---	---	----	----	---

$\xleftarrow{\quad} \textcolor{blue}{W - w + 1} \xrightarrow{\quad}$

## Convolution layer

Input

1	4	-1	0	2	-2	1	3	3	1
---	---	----	---	---	----	---	---	---	---

$\xleftarrow{\quad}$   $w$   $\xrightarrow{\quad}$

Kernel

1	2	0	-1
---	---	---	----

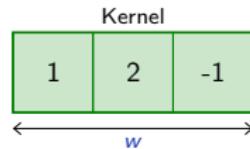
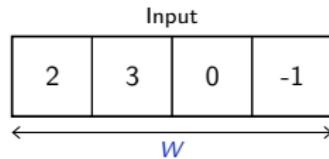
$\xleftarrow{\quad}$   $w$   $\xrightarrow{\quad}$

Output

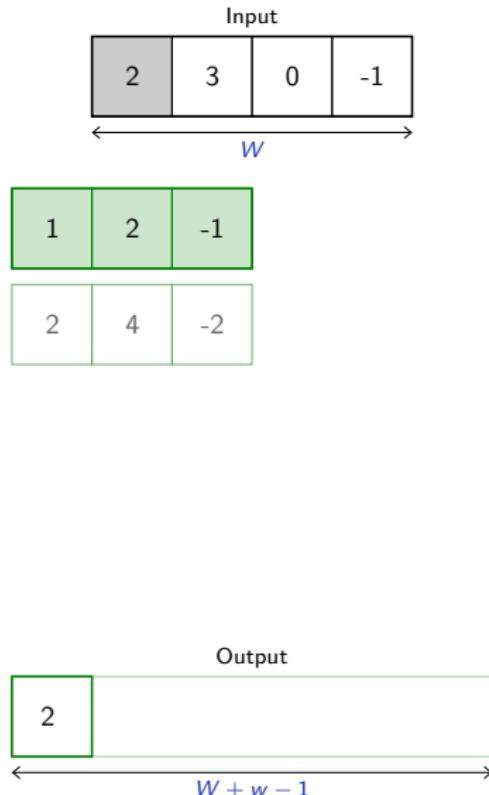
9	0	1	3	-5	-3	6
---	---	---	---	----	----	---

$\xleftarrow{\quad}$   $w - w + 1$   $\xrightarrow{\quad}$

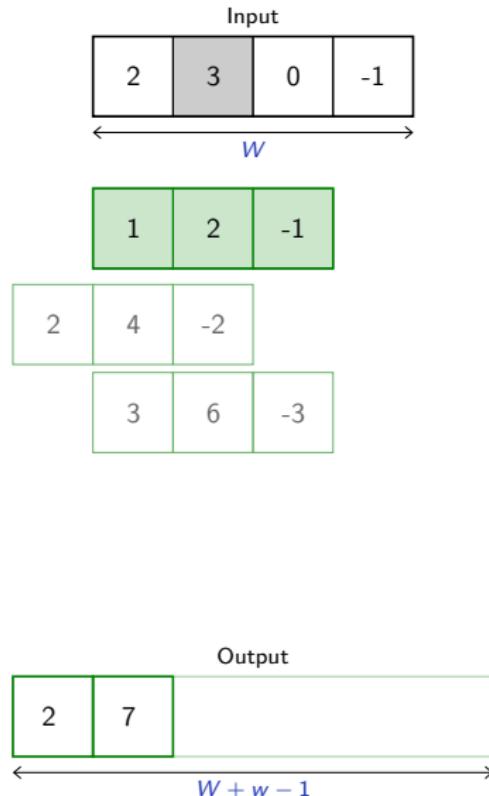
## Transposed convolution layer



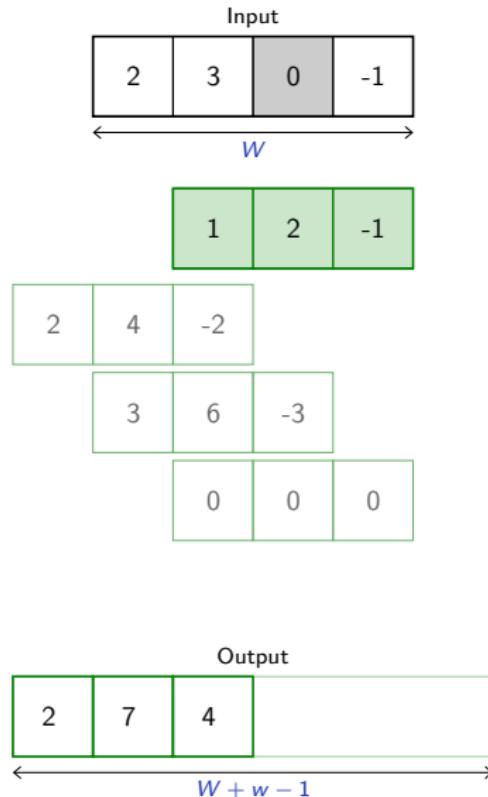
## Transposed convolution layer



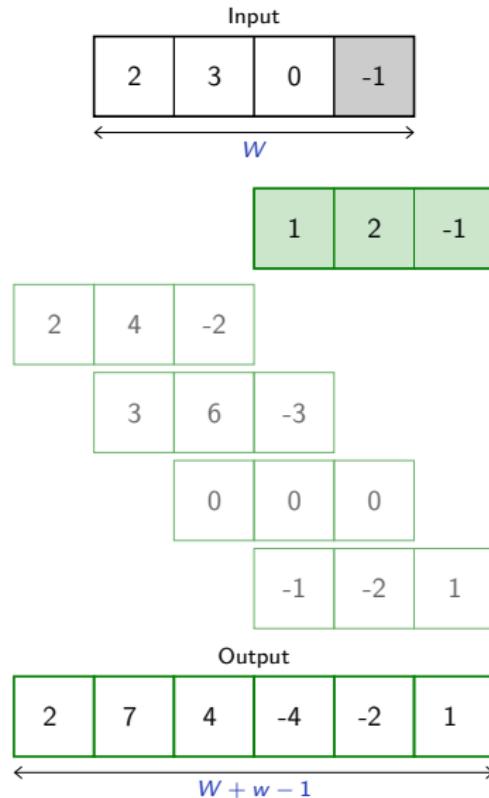
## Transposed convolution layer



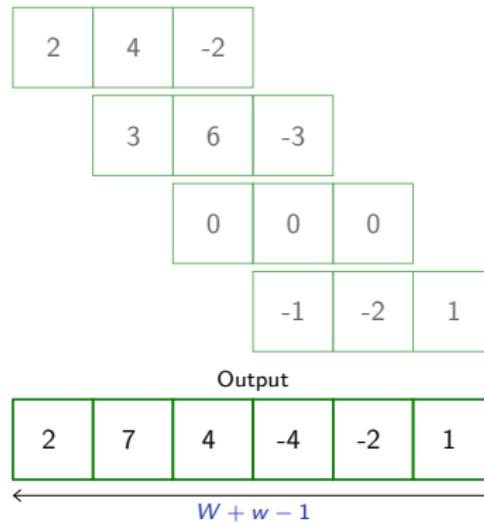
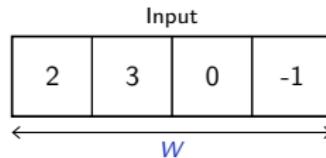
## Transposed convolution layer



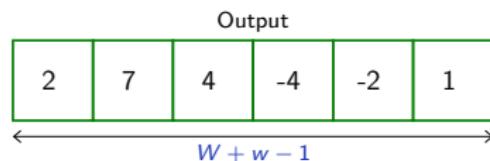
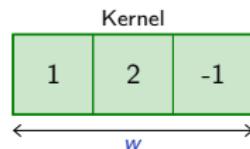
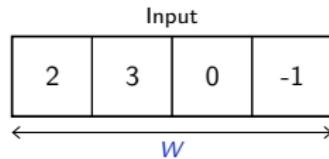
## Transposed convolution layer



## Transposed convolution layer

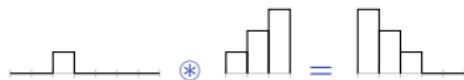


## Transposed convolution layer



`torch.nn.functional.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.

```
>>> 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.]]])
```



`torch.nn.functional.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.

```
>>> 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.]]])
```



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

```
>>> x = torch.tensor([[[ 2., 3., 0., -1.]]])
>>> m = nn.ConvTranspose1d(1, 1, kernel_size=3)
>>> m.bias.data.zero_()
tensor([0.])
>>> m.weight.data.copy_(torch.tensor([ 1, 2, -1 ]))
tensor([[[ 1., 2., -1.]]])
>>> y = m(x)
>>> y
tensor([[[ 2., 7., 4., -4., -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.

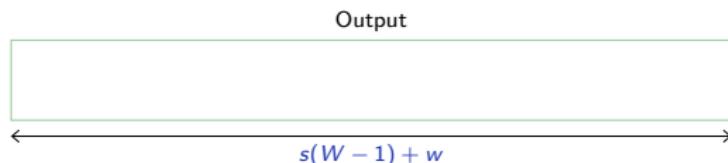
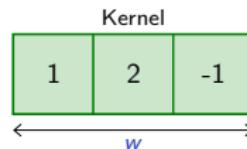
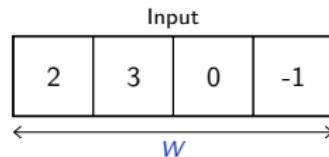
Transposed convolutions also have a `dilation` parameter that behaves as for convolution and expends the kernel size without increasing the number of parameters by making it sparse.

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

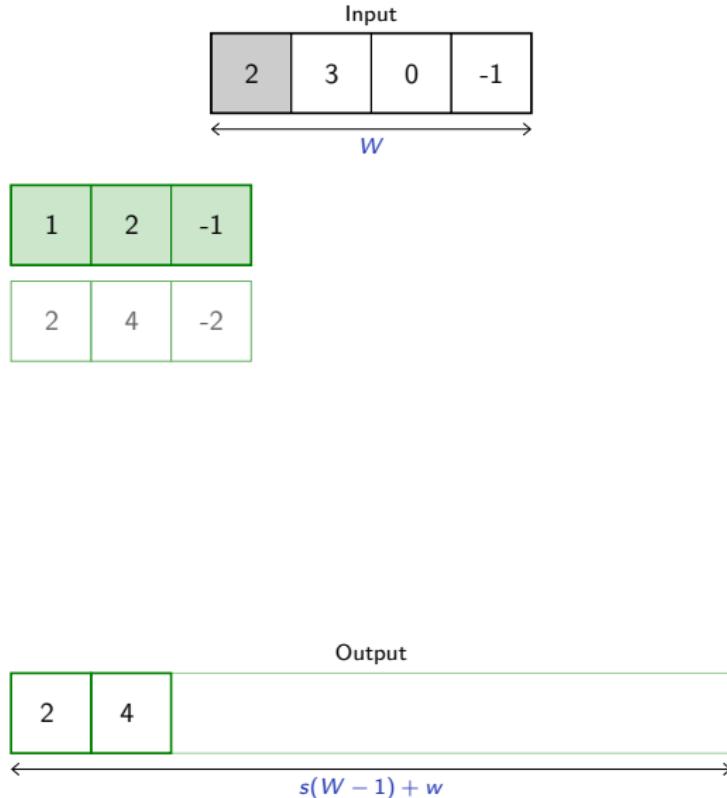


While for convolutions `stride` and `padding` are defined in the input map, for transposed convolutions these parameters are defined in the output map, and the latter modulates a cropping operation.

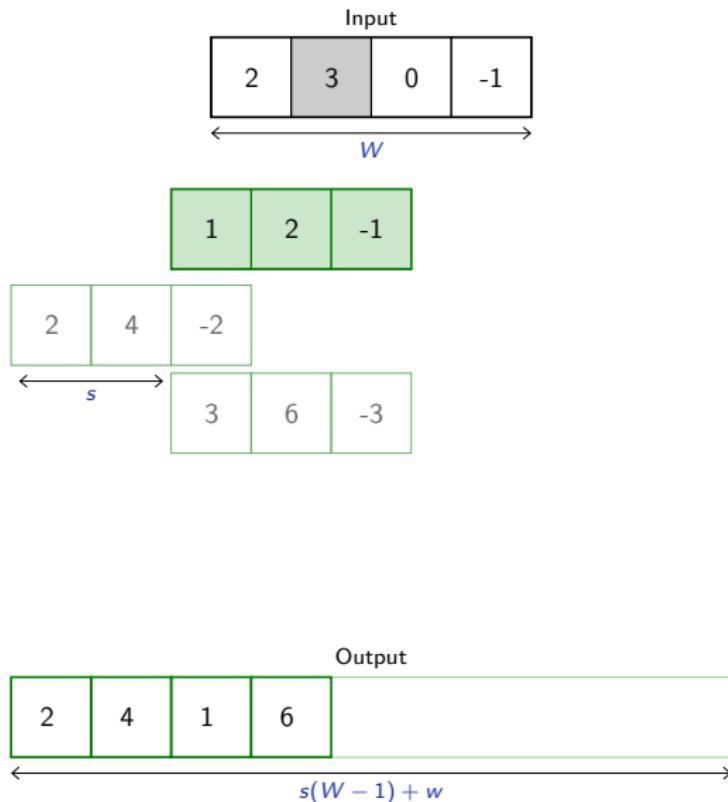
## Transposed convolution layer (stride = 2)



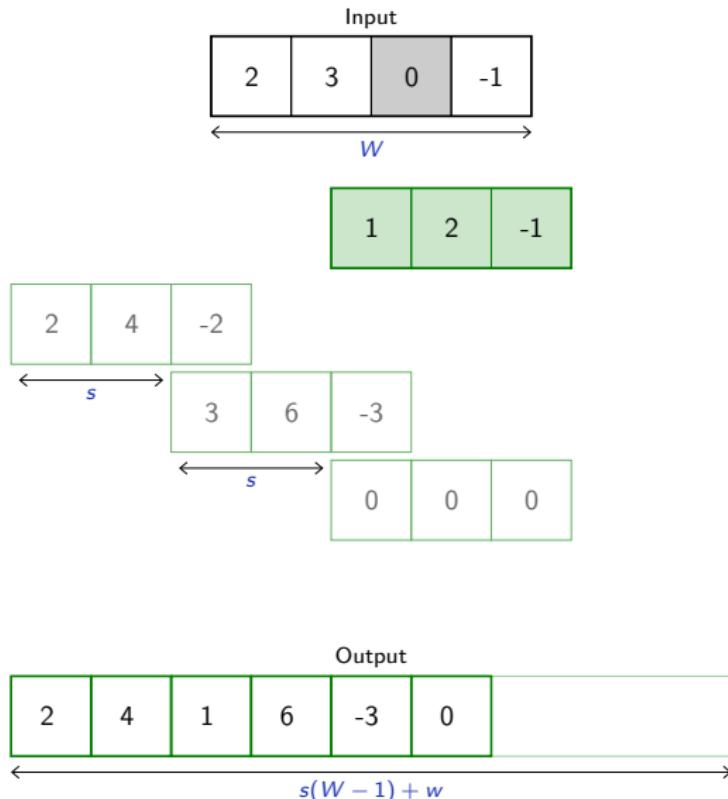
### Transposed convolution layer (stride = 2)



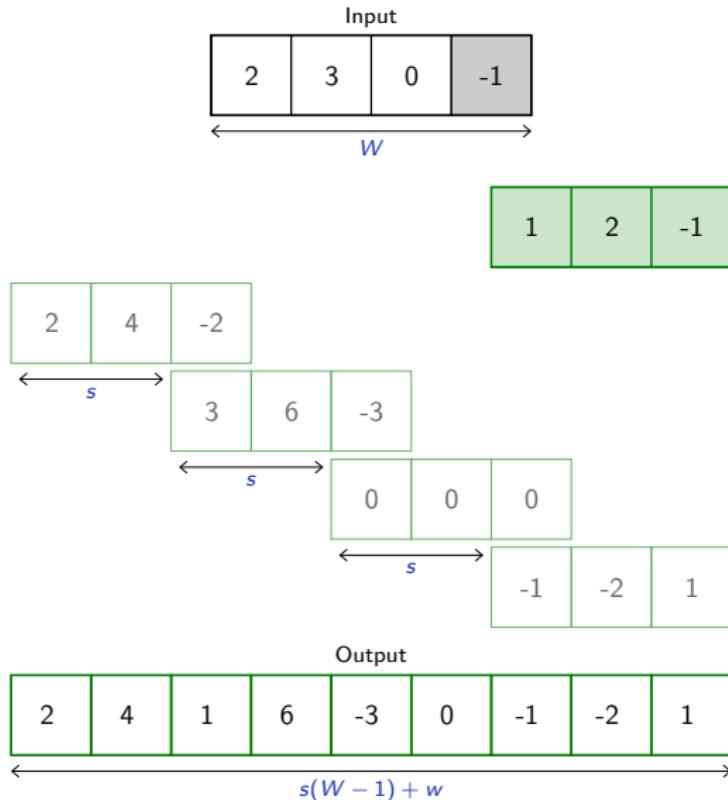
## Transposed convolution layer (stride = 2)



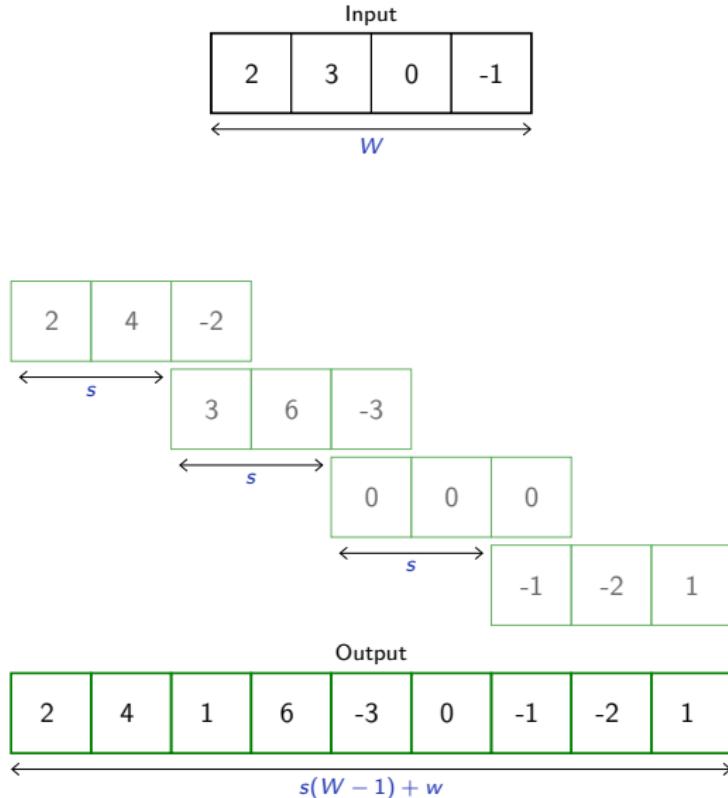
## Transposed convolution layer (stride = 2)



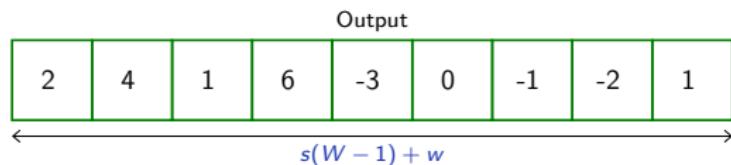
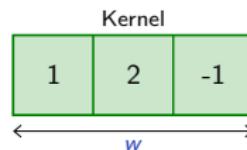
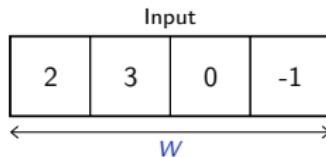
## Transposed convolution layer (stride = 2)



## 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**.

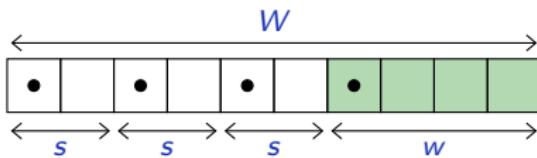
The composition of a convolution and a transposed convolution of same parameters keep the signal size [roughly] unchanged.



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**.

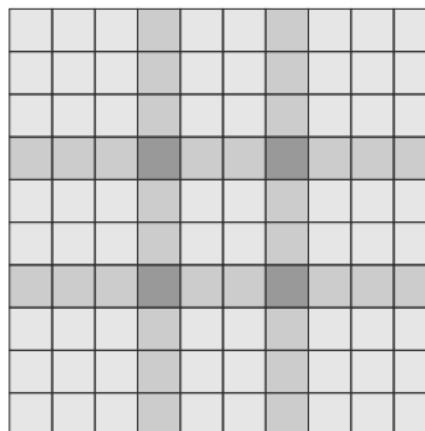
For instance, a 1d convolution of kernel size  $w$  and stride  $s$  composed with the transposed convolution of same parameters maintains the signal size  $W$ , only if

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



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  $4 \times 4$  kernel and stride 3



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

```
>>> x = torch.tensor([[[[ 1., 2. ], [ 3., 4. ]]]])
>>> nn.functional.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]]]])
```

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

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

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

The end

# EE-559 – Deep learning

## 9.2. Autoencoders

François Fleuret

<https://fleuret.org/ee559/>

Mon Feb 18 13:36:19 UTC 2019

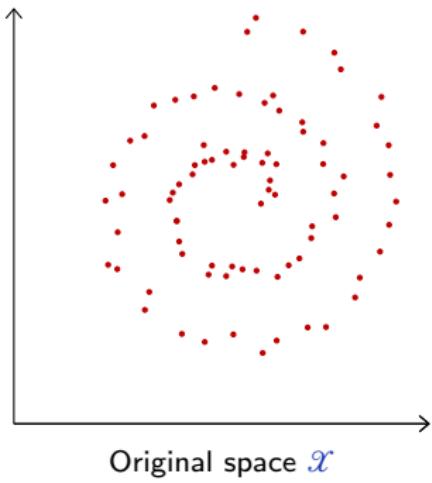


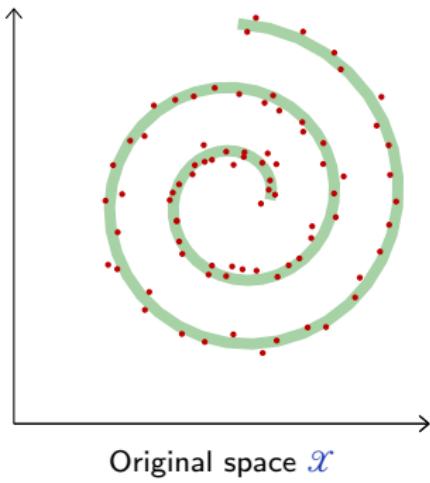
## Embeddings and generative models

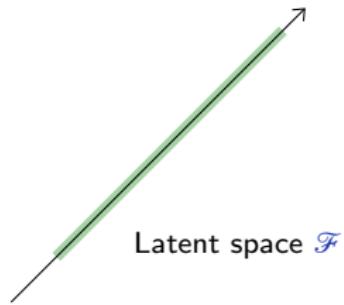
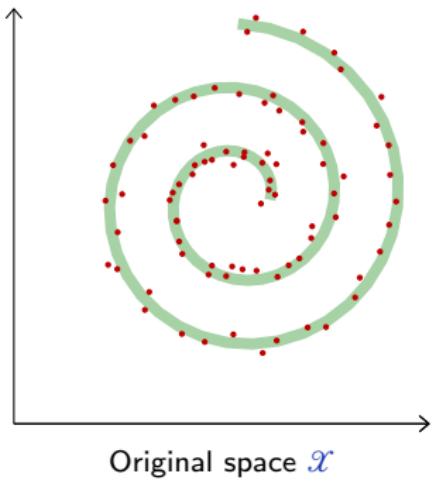
Many applications such as image synthesis, denoising, super-resolution, speech synthesis, compression, etc. require to go beyond classification and regression, and model explicitly a high dimension signal.

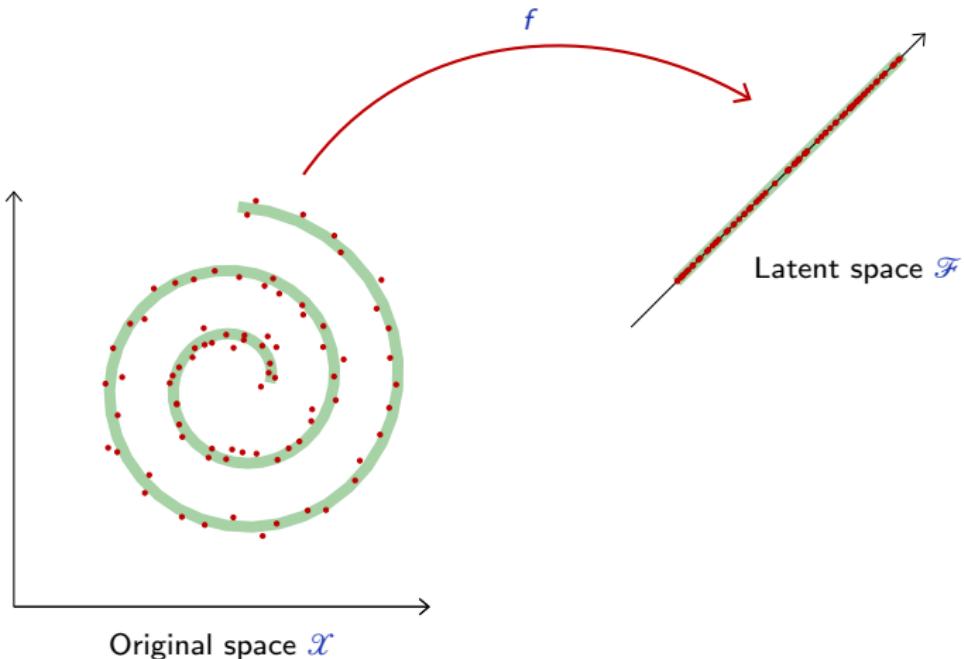
Many applications such as image synthesis, denoising, super-resolution, speech synthesis, compression, etc. require to go beyond classification and regression, and model explicitly a high dimension signal.

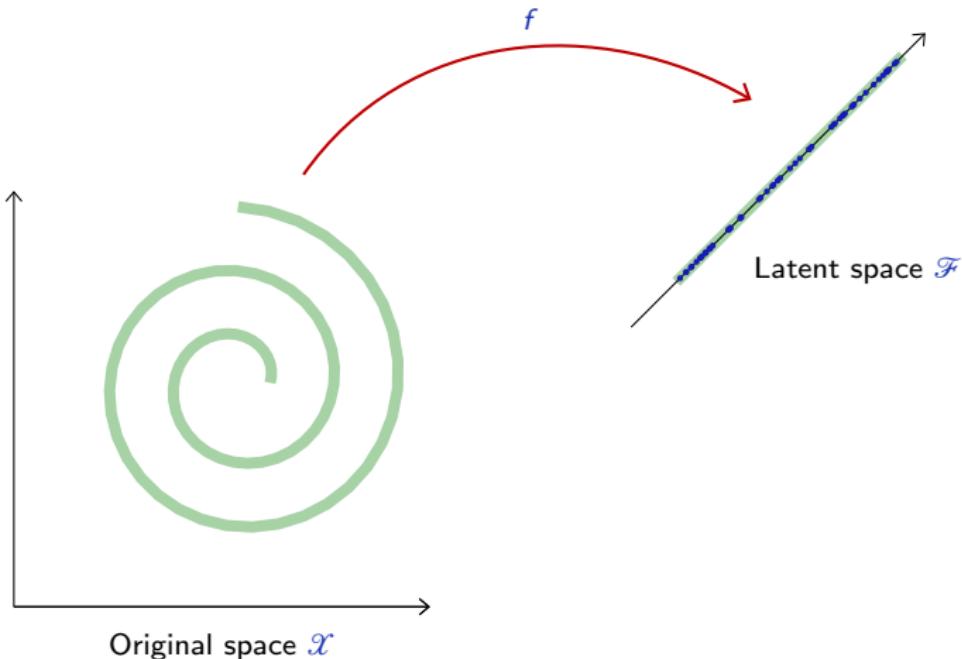
This modeling consists of finding “meaningful degrees of freedom” that describe the signal, and are of lesser dimension.

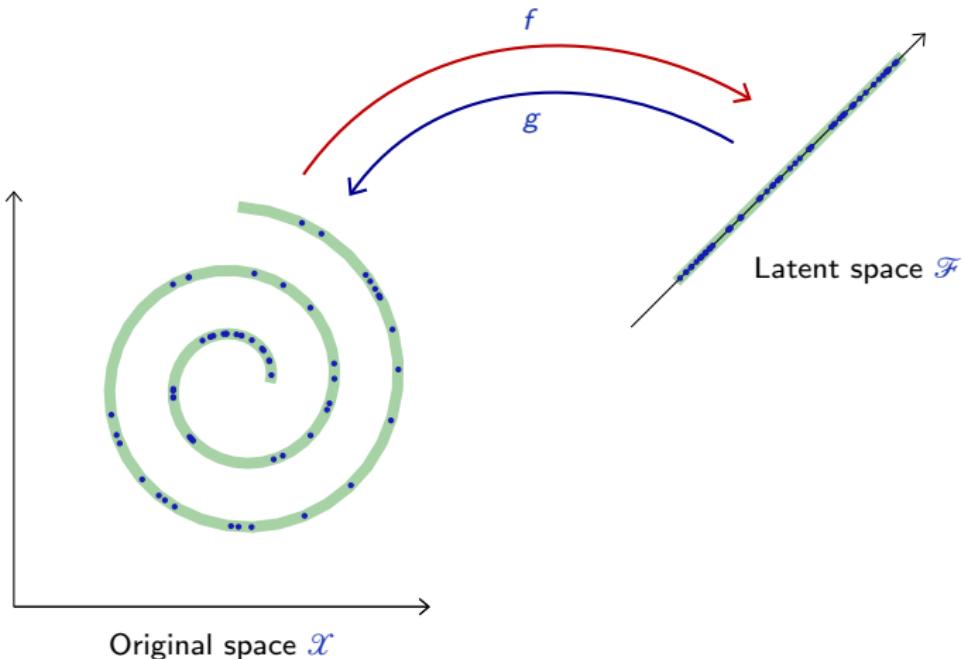


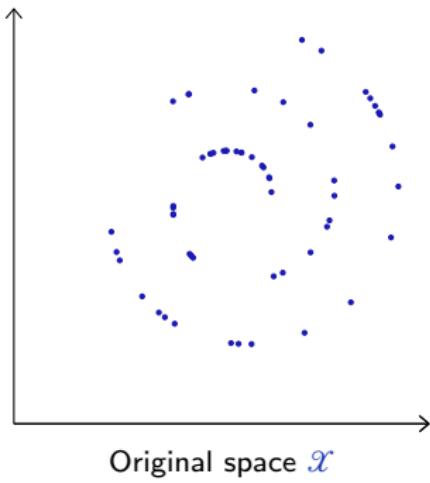












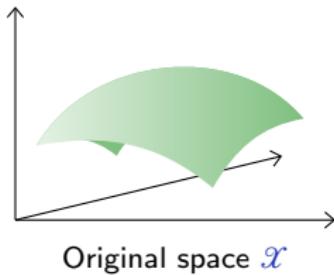
When dealing with real-world signals, this objective involves the same theoretical and practical issues as for classification or regression: defining the right class of high-dimension models, and optimizing them.

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

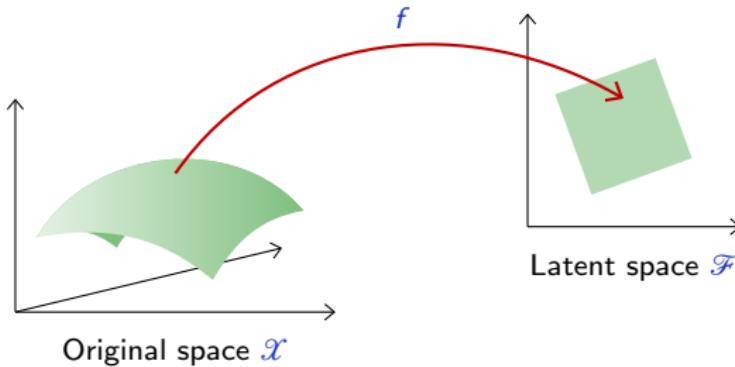
## Autoencoders

An autoencoder (Bourlard and Kamp, 1988; Hinton and Zemel, 1994) combines an **encoder**  $f$  from the original space  $\mathcal{X}$  to a **latent** space  $\mathcal{F}$ , and a **decoder**  $g$  to map back to  $\mathcal{X}$ , such that  $g \circ f$  is [close to] the identity on the data.

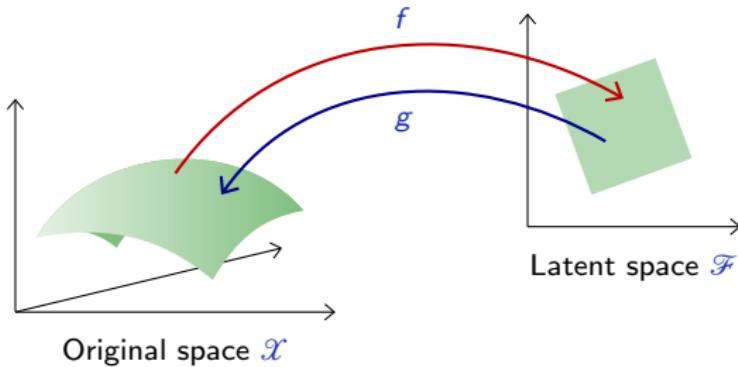
An autoencoder (Bourlard and Kamp, 1988; Hinton and Zemel, 1994) combines an **encoder**  $f$  from the original space  $\mathcal{X}$  to a **latent** space  $\mathcal{F}$ , and a **decoder**  $g$  to map back to  $\mathcal{X}$ , such that  $g \circ f$  is [close to] the identity on the data.



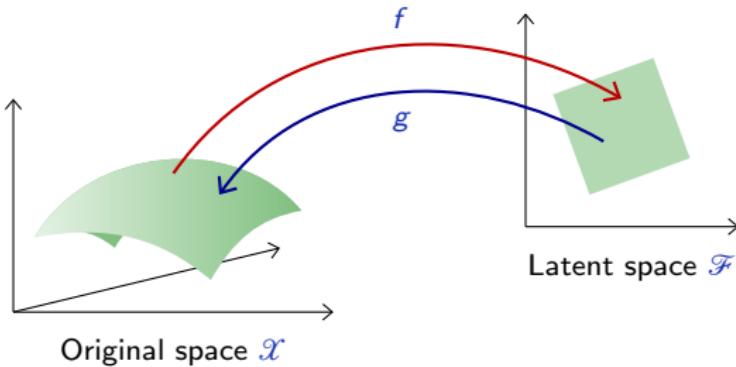
An autoencoder (Bourlard and Kamp, 1988; Hinton and Zemel, 1994) combines an **encoder**  $f$  from the original space  $\mathcal{X}$  to a **latent** space  $\mathcal{F}$ , and a **decoder**  $g$  to map back to  $\mathcal{X}$ , such that  $g \circ f$  is [close to] the identity on the data.



An autoencoder (Bourlard and Kamp, 1988; Hinton and Zemel, 1994) combines an **encoder**  $f$  from the original space  $\mathcal{X}$  to a **latent** space  $\mathcal{F}$ , and a **decoder**  $g$  to map back to  $\mathcal{X}$ , such that  $g \circ f$  is [close to] the identity on the data.



An autoencoder (Bourlard and Kamp, 1988; Hinton and Zemel, 1994) combines an **encoder**  $f$  from the original space  $\mathcal{X}$  to a **latent** space  $\mathcal{F}$ , and a **decoder**  $g$  to map back to  $\mathcal{X}$ , such that  $g \circ f$  is [close to] the identity on the data.



A proper autoencoder has to capture a “good” parametrization of the signal, and in particular the statistical dependencies between the signal components.

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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 the reciprocal transposed convolution 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)
```

$32 \times 9 \times 9$

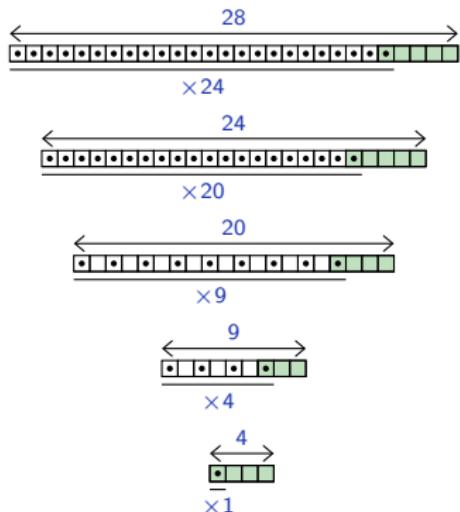
```
nn.Conv2d(32, 32, kernel_size=3, stride=2)
```

$32 \times 4 \times 4$

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

$8 \times 1 \times 1$

---



## Decoder

### Tensor sizes / operations

$8 \times 1 \times 1$

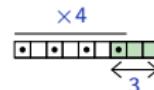
`nn.ConvTranspose2d(8, 32, kernel_size=4, stride=1)`

$32 \times 4 \times 4$



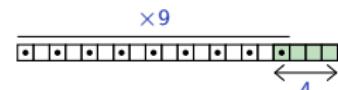
`nn.ConvTranspose2d(32, 32, kernel_size=3, stride=2)`

$32 \times 9 \times 9$



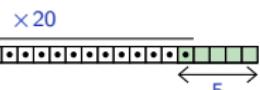
`nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2)`

$32 \times 20 \times 20$



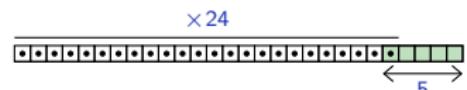
`nn.ConvTranspose2d(32, 32, kernel_size=5, stride=1)`

$32 \times 24 \times 24$



`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)

model.to(device)

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

for epoch in range(args.nb_epochs):
    for input, _ in iter(train_loader):
        input = input.to(device)

        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()
```

$\textcolor{blue}{X}$  (original samples)

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{blue}{X})$  (CNN,  $d = 2$ )

7 2 1 0 9 1 9 9 6 9 0 6  
9 0 1 5 9 7 5 9 9 6 6 5  
9 0 7 9 0 1 5 1 3 6 7 2

$g \circ f(\textcolor{blue}{X})$  (PCA,  $d = 2$ )

9 3 1 0 9 1 9 9 0 9 0 0  
9 0 1 3 9 9 3 9 9 0 9 0  
9 0 9 9 0 1 3 1 3 0 9 0

$\textcolor{blue}{X}$  (original samples)

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{blue}{X})$  (CNN,  $d = 4$ )

7 2 1 0 4 1 9 8 9 9 0 6  
9 0 1 5 4 7 5 9 9 6 6 5  
9 0 7 4 0 1 3 1 3 0 7 2

$g \circ f(\textcolor{blue}{X})$  (PCA,  $d = 4$ )

9 2 1 0 9 1 9 9 0 9 0 0  
9 0 1 3 9 9 8 9 9 0 6 5  
9 0 9 9 0 1 3 1 3 0 9 0

$\textcolor{blue}{X}$  (original samples)

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{blue}{X})$  (CNN,  $d = 8$ )

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{blue}{X})$  (PCA,  $d = 8$ )

7 3 1 0 4 1 9 9 0 9 0 0  
9 0 1 0 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 0 7 0

$\textcolor{blue}{X}$  (original samples)

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 4 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{teal}{X})$  (CNN,  $d = 16$ )

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 4 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{teal}{X})$  (PCA,  $d = 16$ )

7 2 1 0 9 1 4 9 6 9 0 6  
9 0 1 5 9 7 3 4 9 6 4 5  
4 0 7 4 0 1 3 1 3 0 7 2

$\textcolor{blue}{X}$  (original samples)

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{teal}{X})$  (CNN,  $d = 32$ )

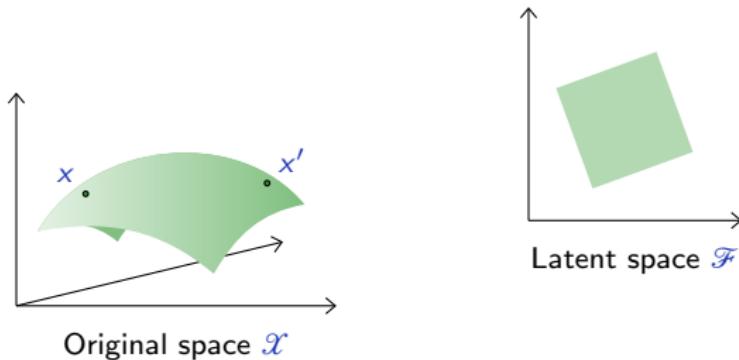
7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(\textcolor{teal}{X})$  (PCA,  $d = 32$ )

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 3 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

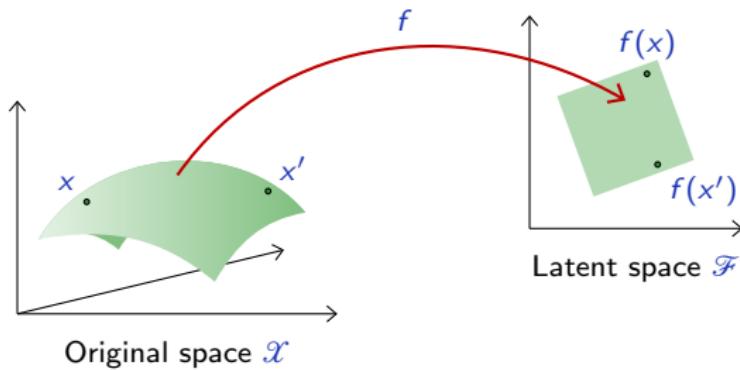
To get an intuition of the latent representation, we can pick two samples  $x$  and  $x'$  at random and interpolate samples along the line in the latent space

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



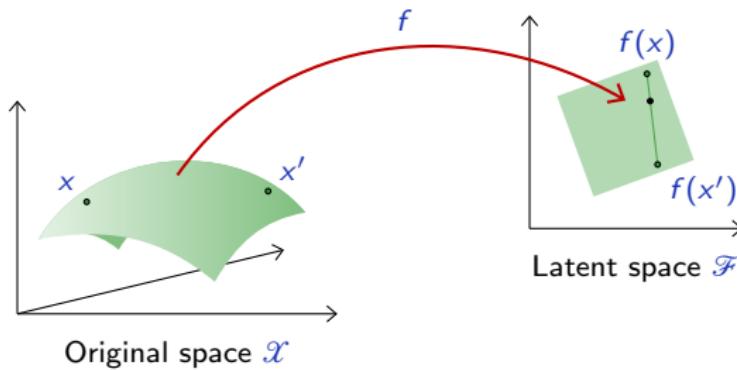
To get an intuition of the latent representation, we can pick two samples  $x$  and  $x'$  at random and interpolate samples along the line in the latent space

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



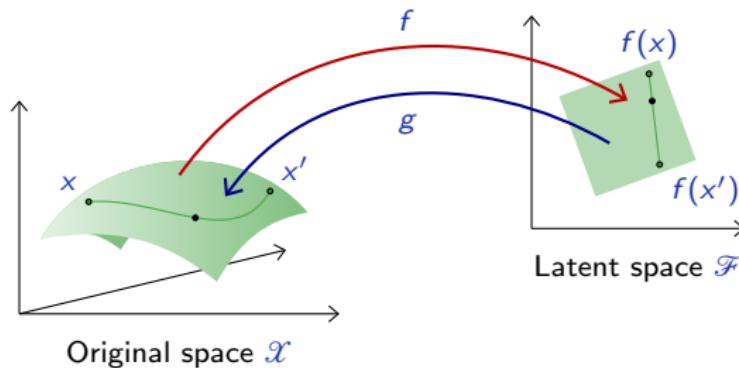
To get an intuition of the latent representation, we can pick two samples  $x$  and  $x'$  at random and interpolate samples along the line in the latent space

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

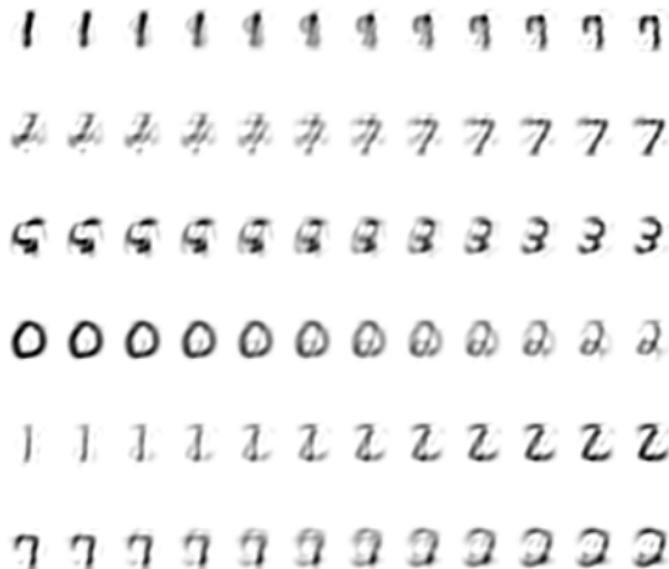


To get an intuition of the latent representation, we can pick two samples  $x$  and  $x'$  at random and interpolate samples along the line in the latent space

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



PCA interpolation ( $d = 32$ )



Autoencoder interpolation ( $d = 8$ )

3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6

9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6

6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6

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  $g$ .

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.

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

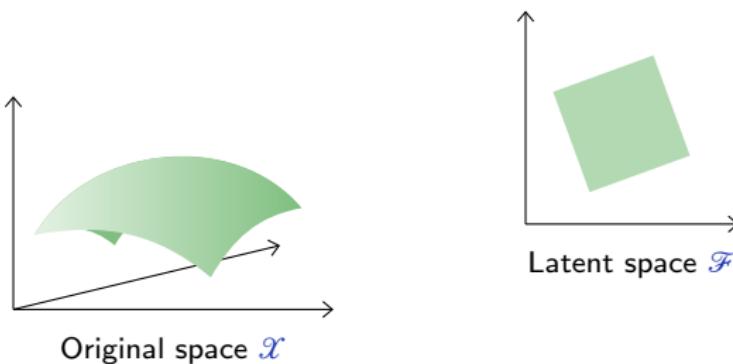
where  $\hat{m}$  is a vector and  $\hat{\Delta}$  a diagonal matrix, both estimated on training data.

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.

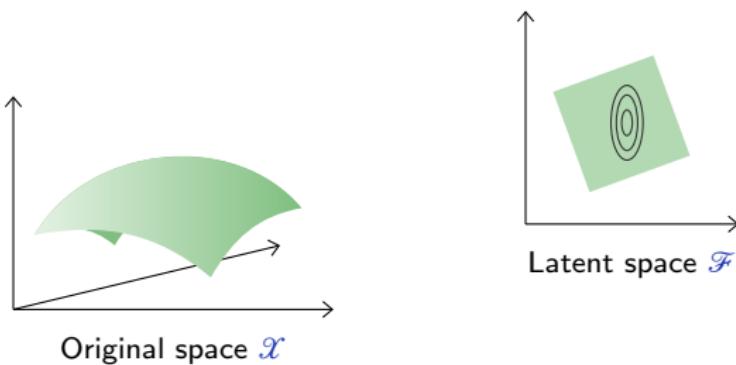


And we can assess the generative capabilities of the decoder  $g$  by introducing a [simple] density model  $q^Z$  over the latent space  $\mathcal{F}$ , sample there, and map the samples into the image space  $\mathcal{X}$  with  $g$ .

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.

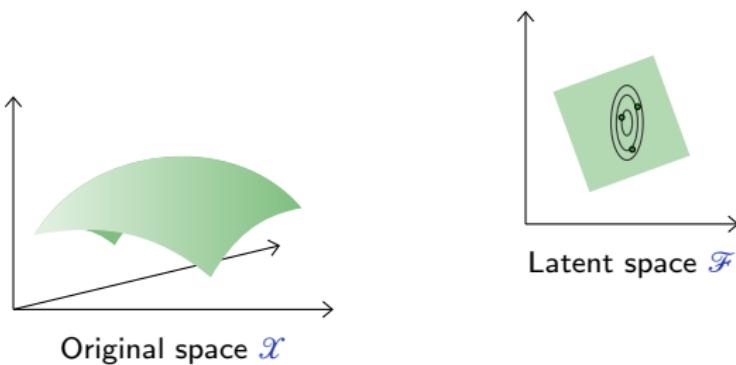


And we can assess the generative capabilities of the decoder  $g$  by introducing a [simple] density model  $q^Z$  over the latent space  $\mathcal{F}$ , sample there, and map the samples into the image space  $\mathcal{X}$  with  $g$ .

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.

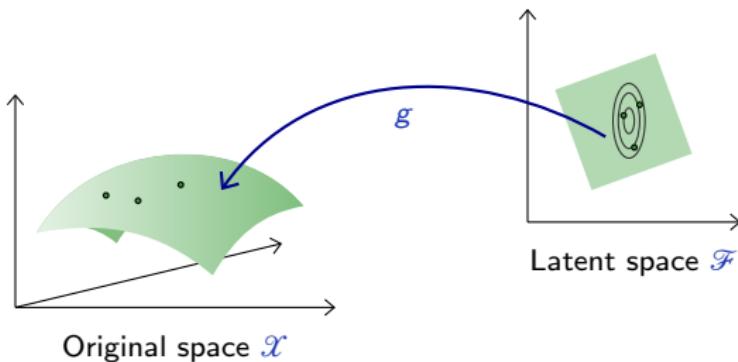


And we can assess the generative capabilities of the decoder  $g$  by introducing a [simple] density model  $q^Z$  over the latent space  $\mathcal{F}$ , sample there, and map the samples into the image space  $\mathcal{X}$  with  $g$ .

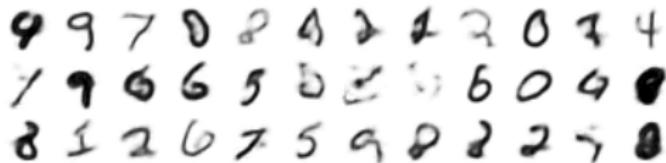
We can for instance use a Gaussian model with diagonal covariance matrix.

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

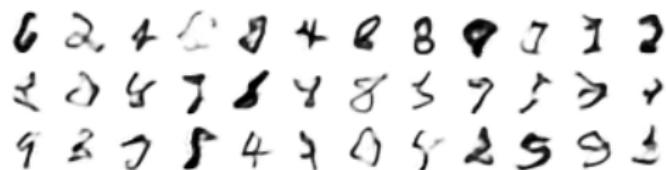
where  $\hat{m}$  is a vector and  $\hat{\Delta}$  a diagonal matrix, both estimated on training data.



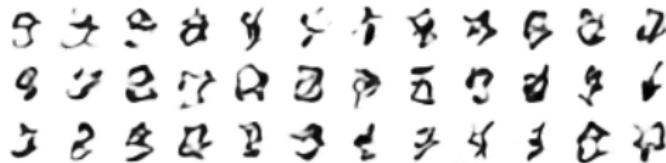
Autoencoder sampling ( $d = 8$ )



Autoencoder sampling ( $d = 16$ )



Autoencoder sampling ( $d = 32$ )



These results are unsatisfying, because the density model used on the latent space  $\mathcal{F}$  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.

The end

## References

- H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4):291–294, 1988.
- G. E. Hinton and R. S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Neural Information Processing Systems (NIPS)*, pages 3–10, 1994.

## EE-559 – Deep learning

### 9.3. Denoising and variational autoencoders

François Fleuret

<https://fleuret.org/ee559/>

Mon Feb 18 13:36:23 UTC 2019

## Denoising Autoencoders

Vincent et al. (2010) interpret the autoencoder in a probabilistic framework as a way of building an encoder that maximizes the mutual information between the input and the latent state.

Vincent et al. (2010) interpret the autoencoder in a probabilistic framework as a way of building an encoder that maximizes the mutual information between the input and the latent state.

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

Vincent et al. (2010) interpret the autoencoder in a probabilistic framework as a way of building an encoder that maximizes the mutual information between the input and the latent state.

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

We have

$$\begin{aligned}\operatorname{argmax}_{\theta} \mathbb{I}(X, Z) &= \operatorname{argmax}_{\theta} \mathbb{H}(X) - \mathbb{H}(X | Z) \\ &= \operatorname{argmax}_{\theta} -\mathbb{H}(X | Z) \\ &= \operatorname{argmax}_{\theta} \mathbb{E}_{q(X, Z)} [\log q(X | Z)].\end{aligned}$$

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

Given  $(X, Z) \sim q_{\theta}$ , for any distribution  $p$  we have

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

Given  $(X, Z) \sim q_{\theta}$ , for any distribution  $p$  we have

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

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  $p$

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

where  $g$  is deterministic

If we consider the following model for  $p$

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

where  $g$  is deterministic, we get

$$\begin{aligned}\mathbb{E}_{q(X, Z)} [\log p(X | Z)] &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right] \\ &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(f(X); \eta)\|^2}{2\sigma^2} \right].\end{aligned}$$

If we consider the following model for  $p$

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

where  $g$  is deterministic, we get

$$\begin{aligned}\mathbb{E}_{q(X, Z)} [\log p(X | Z)] &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right] \\ &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(f(X); \eta)\|^2}{2\sigma^2} \right].\end{aligned}$$

If optimizing  $\eta$  makes the bound tight, the final loss is the reconstruction error

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

If we consider the following model for  $p$

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

where  $g$  is deterministic, we get

$$\begin{aligned}\mathbb{E}_{q(X, Z)} [\log p(X | Z)] &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right] \\ &= -\mathbb{E}_{q(X, Z)} \left[ \frac{\|X - g(f(X); \eta)\|^2}{2\sigma^2} \right].\end{aligned}$$

If optimizing  $\eta$  makes the bound tight, the final loss is the reconstruction error

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

**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 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. propose to degrade the signal with noise before feeding it to the encoder, but to keep the MSE to the original signal.

This makes the encoder retain information about structures beyond local noise.

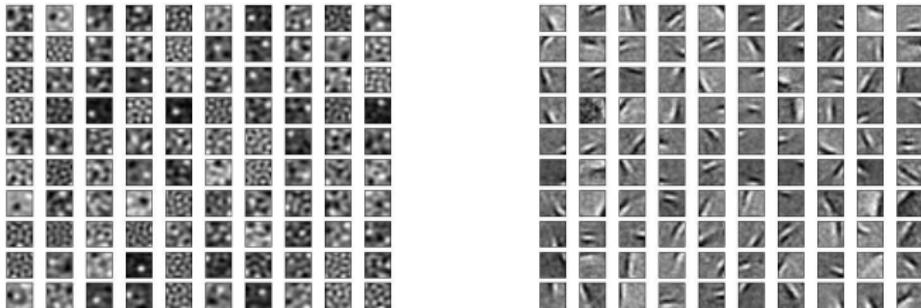


Figure 6: Weight decay vs. Gaussian noise. We show typical filters learnt from natural image patches in the over-complete case (200 hidden units). *Left:* regular autoencoder with weight decay. We tried a wide range of weight-decay values and learning rates: filters never appeared to capture a more interesting 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)

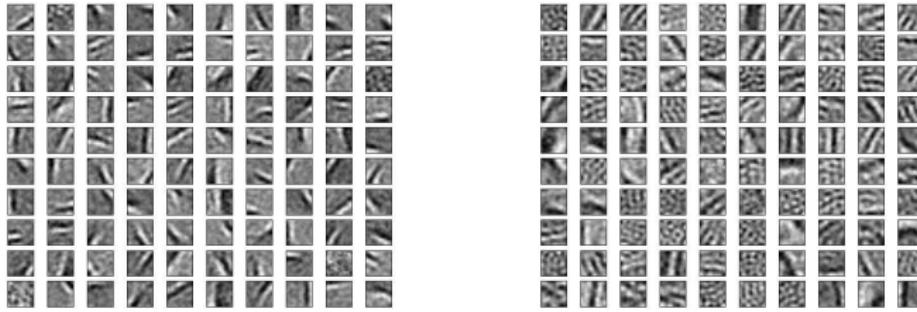
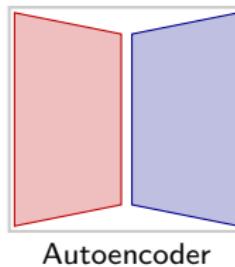


Figure 7: Filters obtained on natural image patches by denoising autoencoders using other noise types. *Left*: with 10% salt-and-pepper noise, we obtain oriented Gabor-like filters. They appear slightly less localized than when using Gaussian noise (contrast with Figure 6 right). *Right*: with 55% zero-masking 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.

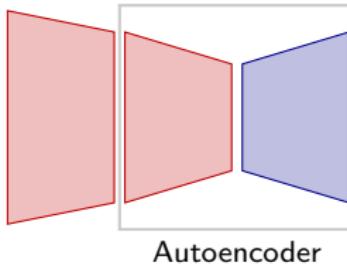
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



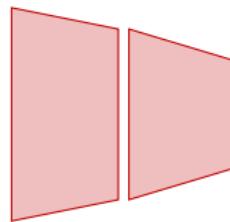
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



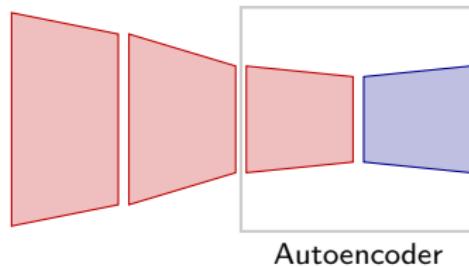
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



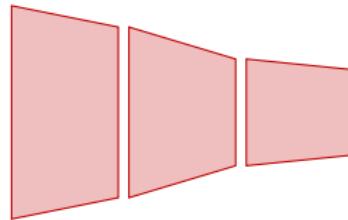
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



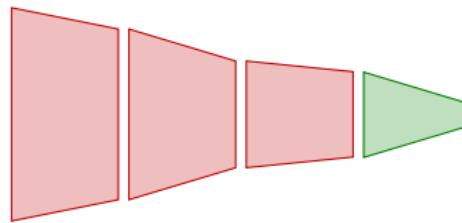
Vincent et al. build deep MLPs whose layers are initialized successively as encoders trained within a noisy autoencoder.



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.

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

(Vincent et al., 2010)

## Variational Autoencoders

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 the following two distributions:

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

We consider the following two distributions:

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

What matters is that  $p(X)$  mimics the data-distribution, that is find  $g$  that maximizes the log-likelihood

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

We consider the following two distributions:

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

What matters is that  $p(X)$  mimics the data-distribution, that is find  $g$  that maximizes the log-likelihood

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

However, while we can sample  $z$  and compute  $g(z)$ , 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  $q(Z | X = x)$  is the distribution of the latent encoding  $f(x)$ . We want to maximize

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

or equivalently minimize

$$\begin{aligned} \mathbb{E}_{q(X)} \left[ \log q(X) - \log p(X) \right] &= \mathbb{D}_{\text{KL}}(q(X) \parallel p(X)) \\ &\leq \mathbb{D}_{\text{KL}}(q(X, Z) \parallel p(X, Z)). \end{aligned}$$

Remember that  $q(X)$  is the data distribution, and  $q(Z | X = x)$  is the distribution of the latent encoding  $f(x)$ . We want to maximize

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

or equivalently minimize

$$\begin{aligned} \mathbb{E}_{q(X)} \left[ \log q(X) - \log p(X) \right] &= \mathbb{D}_{\text{KL}}(q(X) \| p(X)) \\ &\leq \mathbb{D}_{\text{KL}}(q(X, Z) \| p(X, Z)). \end{aligned}$$

We will minimize this latter bound, that can be rewritten as

$$\begin{aligned} \mathbb{D}_{\text{KL}}(q(X, Z) \| p(X, Z)) &= \\ \mathbb{E}_{q(X)} \left[ \mathbb{D}_{\text{KL}}(q(Z | X) \| p(Z)) \right] &- \mathbb{E}_{q(X, Z)} \left[ \log p(X | Z) \right] + \mathbb{H}(q(X)). \end{aligned}$$

Kingma and Welling use Gaussians with diagonal covariance for both  $q(Z | X)$  and  $p(X | Z)$ :

- the encoder maps a data point from the signal space  $\mathbb{R}^c$  to [the parameters of] a Gaussian in the latent space  $\mathbb{R}^d$

$$f : \mathbb{R}^c \rightarrow \mathbb{R}^{2d}$$
$$x \mapsto \left( \mu_1^f, \dots, \mu_d^f, \sigma_1^f, \dots, \sigma_d^f \right),$$

Kingma and Welling use Gaussians with diagonal covariance for both  $q(Z | X)$  and  $p(X | Z)$ :

- the encoder maps a data point from the signal space  $\mathbb{R}^c$  to [the parameters of] a Gaussian in the latent space  $\mathbb{R}^d$

$$f : \mathbb{R}^c \rightarrow \mathbb{R}^{2d}$$
$$x \mapsto (\mu_1^f, \dots, \mu_d^f, \sigma_1^f, \dots, \sigma_d^f),$$

- the decoder maps a latent value from  $\mathbb{R}^d$  to [the parameters of] a Gaussian in the signal space  $\mathbb{R}^c$

$$g : \mathbb{R}^d \rightarrow \mathbb{R}^{2c}$$
$$z \mapsto (\mu_1^g, \dots, \mu_c^g, \sigma_1^g, \dots, \sigma_c^g).$$

We have to minimize

$$\mathcal{L} = \hat{\mathbb{E}}_{q(X)} \left[ \mathbb{D}_{\text{KL}} (q(Z | X) \| p(Z)) \right] - \hat{\mathbb{E}}_{q(X, Z)} \left[ \log p(X | Z) \right].$$

We have to minimize

$$\mathcal{L} = \hat{\mathbb{E}}_{q(X)} \left[ \mathbb{D}_{\text{KL}}(q(Z | X) \| p(Z)) \right] - \hat{\mathbb{E}}_{q(X, Z)} \left[ \log p(X | Z) \right].$$

The first term is the average of

$$\mathbb{D}_{\text{KL}}(q(Z | X = x) \| p(Z)) = -\frac{1}{2} \sum_d \left( 1 + 2 \log \sigma_d^f(x) - \left( \mu_d^f(x) \right)^2 - \left( \sigma_d^f(x) \right)^2 \right).$$

over the  $x_n$ s.

We have to minimize

$$\mathcal{L} = \hat{\mathbb{E}}_{q(X)} \left[ \mathbb{D}_{\text{KL}}(q(Z | X) \| p(Z)) \right] - \hat{\mathbb{E}}_{q(X, Z)} \left[ \log p(X | Z) \right].$$

The first term is the average of

$$\mathbb{D}_{\text{KL}}(q(Z | X = x) \| p(Z)) = -\frac{1}{2} \sum_d \left( 1 + 2 \log \sigma_d^f(x) - \left( \mu_d^f(x) \right)^2 - \left( \sigma_d^f(x) \right)^2 \right).$$

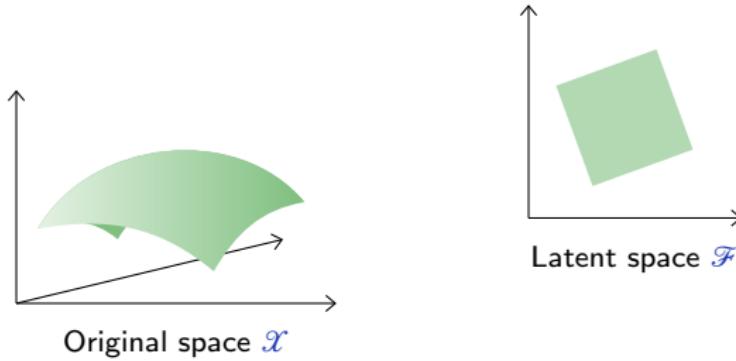
over the  $x_n$ s.

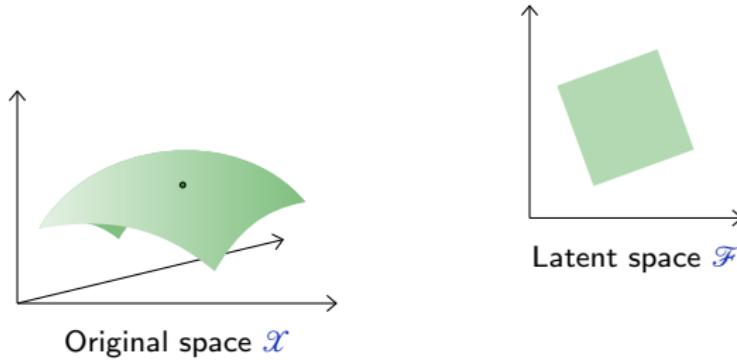
The second term of  $\mathcal{L}$  is the average of

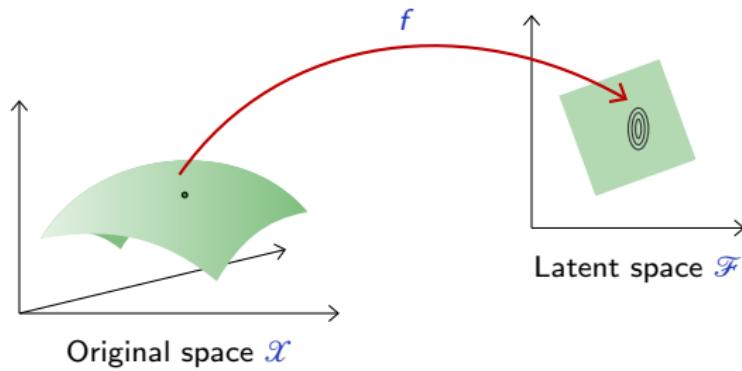
$$-\log p(X = x | Z = z) = \frac{1}{2} \sum_d \left( \log 2\pi + 2 \log \sigma_d^g(z) + \frac{(x_d - \mu_d^g(z))^2}{(\sigma_d^g(z))^2} \right)$$

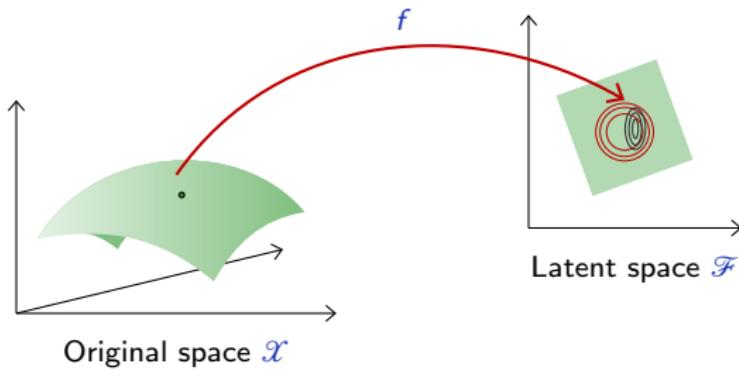
over the  $x_n$ , with one  $z_n$  sampled for each (could be more)

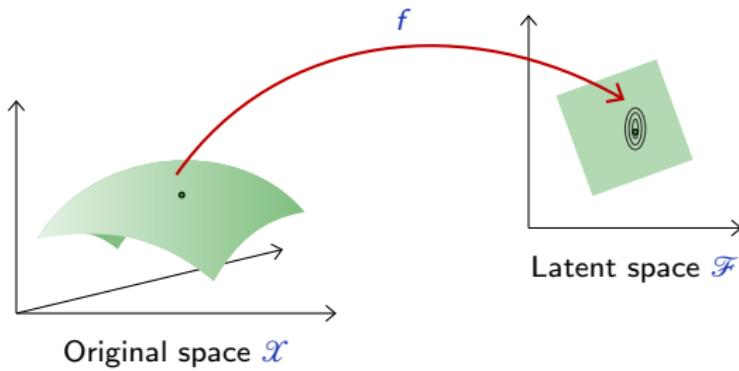
$$z_n \sim \mathcal{N} \left( \mu^f(x_n), \sigma^f(x_n) \right), \quad n = 1, \dots, N.$$

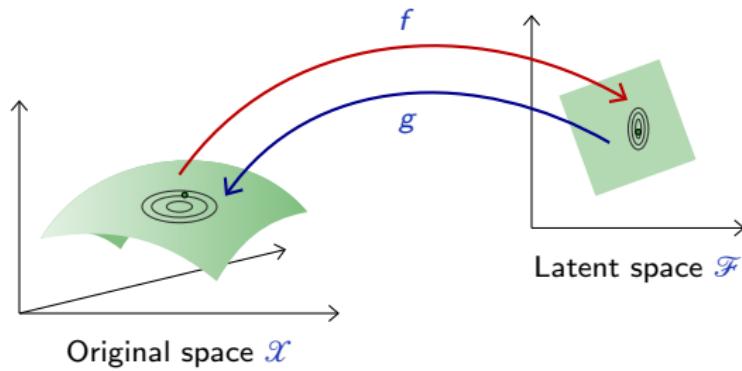












Regarding implementation: the encoder now maps to twice the number of dimensions, which corresponds to the  $\mu^f$ 's and  $\log((\sigma^f)^2)$ 's.

Regarding implementation: the encoder now maps to twice the number of dimensions, which corresponds to the  $\mu^f$ 's and  $\log((\sigma^f)^2)$ 's.

Also, as in Kingma and Welling (2013), **we use a fixed variance of 1 for the decoder**. So it outputs the  $\mu^g$ 's alone, and its dimension remains unchanged.

The first term of the loss is the average of

$$\mathbb{D}_{\text{KL}}(q(Z | X = x) \| p(Z)) = -\frac{1}{2} \sum_d \left( 1 + 2 \log \sigma_d^f(x) - \left( \mu_d^f(x) \right)^2 - \left( \sigma_d^f(x) \right)^2 \right).$$

over the  $x_n$ .

The first term of the loss is the average of

$$\mathbb{D}_{\text{KL}}(q(Z | X = x) \| p(Z)) = -\frac{1}{2} \sum_d \left( 1 + 2 \log \sigma_d^f(x) - \left( \mu_d^f(x) \right)^2 - \left( \sigma_d^f(x) \right)^2 \right).$$

over the  $x_n$ .

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)
```

Since we use a constant variance of 1 for the decoder, the second term of  $\mathcal{L}$  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.

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

Since we use a constant variance of 1 for the decoder, the second term of  $\mathcal{L}$  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.

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

This can be implemented as

```
std_f = torch.exp(0.5 * logvar_f)
z = torch.empty_like(mu_f).normal_() * 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)
```

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.empty_like(mu_f).normal_() * 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
```

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.empty_like(mu_f).normal_() * 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  $\mu^f$  and  $\mu^g$  as prediction.

Original

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 8 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

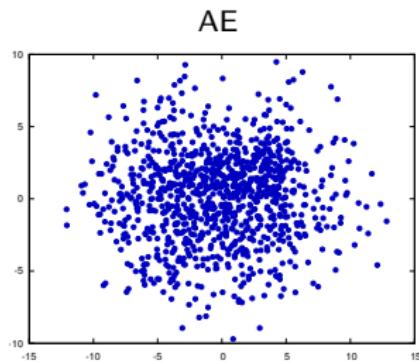
Autoencoder reconstruction ( $d = 32$ )

7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 8 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

Variational Autoencoder reconstruction ( $d = 32$ )

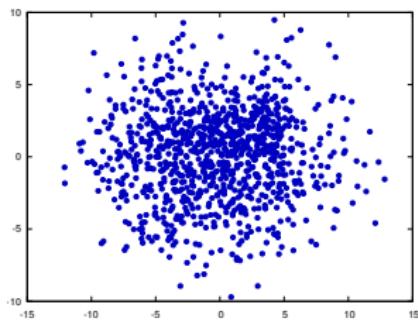
7 2 1 0 4 1 4 9 5 9 0 6  
9 0 1 5 9 7 8 4 9 6 6 5  
4 0 7 4 0 1 3 1 3 4 7 2

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

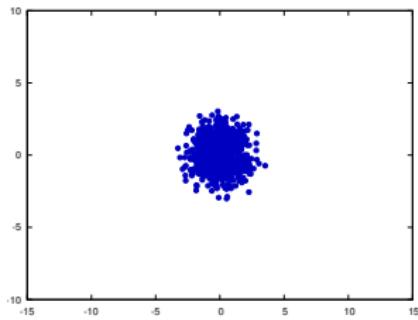


We can look at two latent features to check that they are Normal for the VAE.

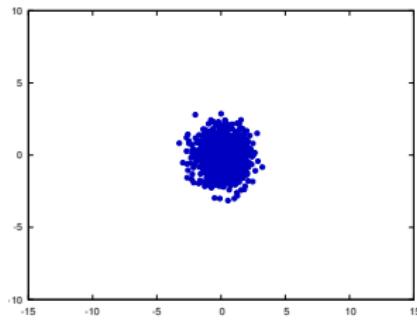
AE



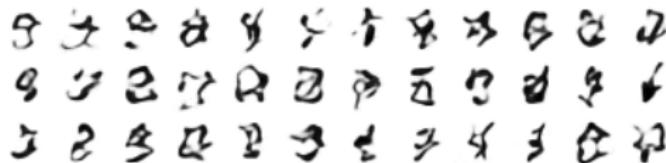
VAE



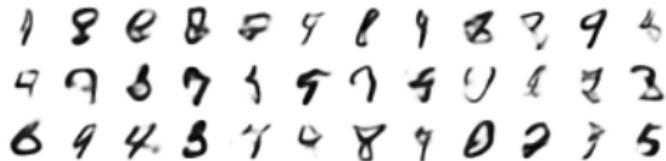
$\mathcal{N}(0, 1)$



Autoencoder sampling ( $d = 32$ )



Variational Autoencoder sampling ( $d = 32$ )



The end

## References

- D. P. Kingma and M. Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.
- P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research (JMLR)*, 11:3371–3408, 2010.

## EE-559 – Deep learning

### 9.4. Non-volume preserving networks

François Fleuret

<https://fleuret.org/ee559/>

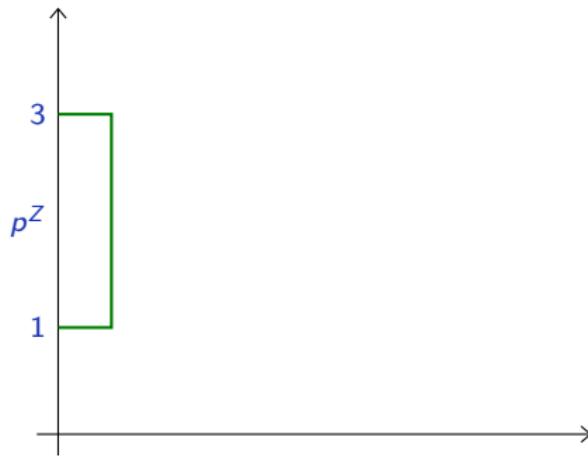
Mon Feb 18 13:36:25 UTC 2019

A standard result of probability theory is that if  $f$  is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) |J_f(x)|.$$

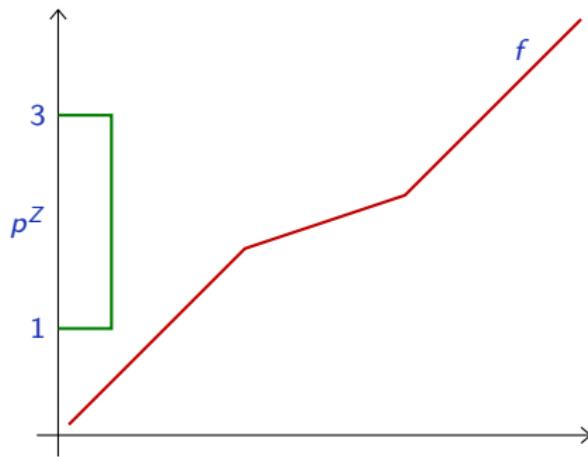
A standard result of probability theory is that if  $f$  is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) |J_f(x)|.$$



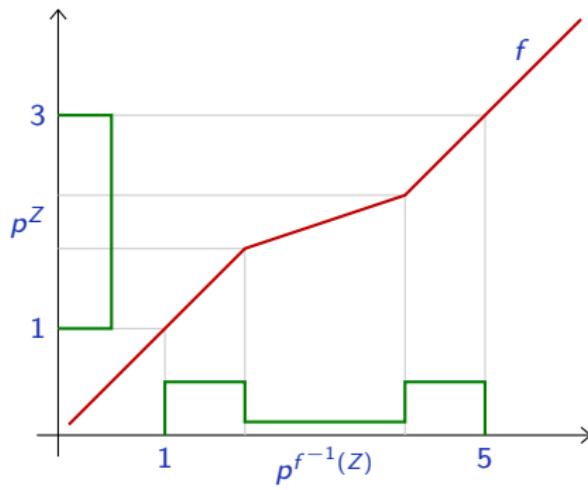
A standard result of probability theory is that if  $f$  is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) |J_f(x)|.$$



A standard result of probability theory is that if  $f$  is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) |J_f(x)|.$$



From this equality, if  $f$  is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of  $f^{-1}(Z)$  fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n)$$

From this equality, if  $f$  is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of  $f^{-1}(Z)$  fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n) = \sum_n \log \left( p^Z(f(x_n)) |J_f(x_n)| \right).$$

From this equality, if  $f$  is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of  $f^{-1}(Z)$  fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n) = \sum_n \log \left( p^Z(f(x_n)) |J_f(x_n)| \right).$$

If we are able to do so, then we can synthesize a new  $X$  by sampling  $Z \sim \mathcal{N}(0, 1)$  and computing  $f^{-1}(Z)$ .

If  $Z \sim \mathcal{N}(0, I)$ ,

$$\log p^Z(f(x_n)) = -\frac{1}{2} (\|f(x_n)\|^2 + d \log 2\pi).$$

If  $Z \sim \mathcal{N}(0, I)$ ,

$$\log p^Z(f(x_n)) = -\frac{1}{2} (\|f(x_n)\|^2 + d \log 2\pi).$$

And remember that if  $f$  is a composition of functions

$$f = f^{(K)} \circ \dots \circ f^{(1)}$$

we have

$$J_f(x) = \prod_{k=1}^K J_{f^{(k)}} \left( f^{(k-1)} \circ \dots \circ f^{(1)}(x) \right),$$

so

$$\log |J_f(x)| = \sum_{k=1}^K \log \left| J_{f^{(k)}} \left( f^{(k-1)} \circ \dots \circ f^{(1)}(x) \right) \right|.$$

If  $f^{(k)}$  are standard layers computing  $f^{-1}(z)$  is impossible, and computing  $|J_f(x)|$  is intractable.

If  $f^{(k)}$  are standard layers computing  $f^{-1}(z)$  is impossible, and computing  $|J_f(x)|$  is intractable.

Dinh et al. (2014) introduced the **coupling layers** to address both issues.

The resulting Non-Volume Preserving network (NVP) is an example of a **Normalizing flow** (Rezende and Mohamed, 2015).

We use here the formalism from Dinh et al. (2016).

Given a dimension  $d$ , a Boolean vector  $b \in \{0, 1\}^d$  and two mappings

$$s : \mathbb{R}^d \rightarrow \mathbb{R}^d$$

$$t : \mathbb{R}^d \rightarrow \mathbb{R}^d,$$

We use here the formalism from Dinh et al. (2016).

Given a dimension  $d$ , a Boolean vector  $b \in \{0, 1\}^d$  and two mappings

$$s : \mathbb{R}^d \rightarrow \mathbb{R}^d$$

$$t : \mathbb{R}^d \rightarrow \mathbb{R}^d,$$

we define a [fully connected] coupling layer as the transformation

$$c : \mathbb{R}^d \rightarrow \mathbb{R}^d$$

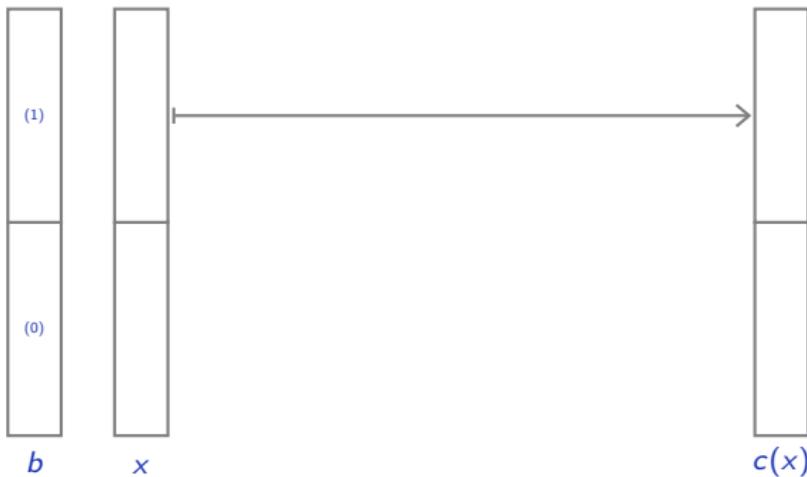
$$x \mapsto b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

where  $\exp$  is component-wise, and  $\odot$  is the Hadamard component-wise product.

The expression

$$c(x) = b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

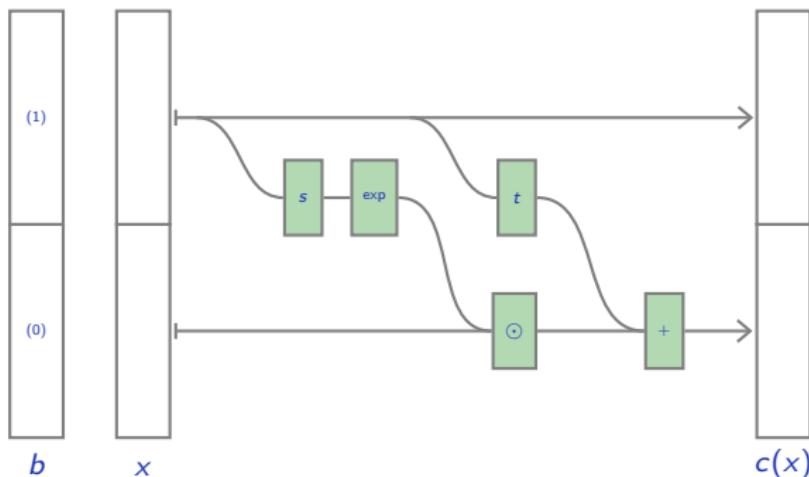
can be understood as: forward  $b \odot x$  unchanged,



The expression

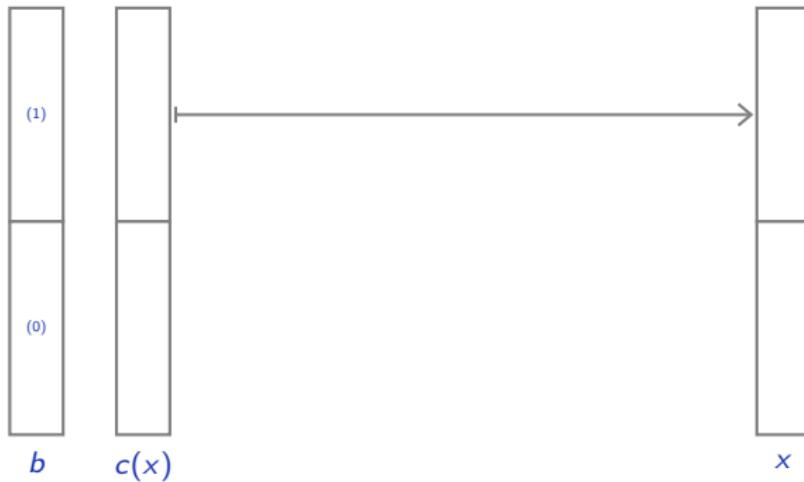
$$c(x) = b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

can be understood as: forward  $b \odot x$  unchanged, and apply to  $(1 - b) \odot x$  an invertible transformation parametrized by  $b \odot x$ .



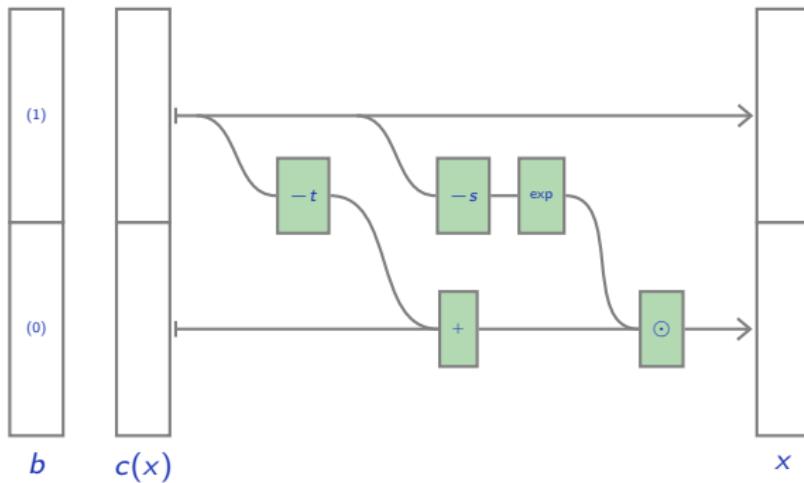
The consequence is that  $c$  is invertible, and if  $y = c(x)$

$$x = b \odot y + (1 - b) \odot \left( y - t(b \odot y) \right) \odot \exp(-s(b \odot y)).$$



The consequence is that  $c$  is invertible, and if  $y = c(x)$

$$x = b \odot y + (1 - b) \odot \left( y - t(b \odot y) \right) \odot \exp(-s(b \odot y)).$$



The second property of this mapping is the simplicity of its Jacobian determinant.

The second property of this mapping is the simplicity of its Jacobian determinant. Since

$$c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right)$$

we have,  $\forall i, j, x$ ,

$$\begin{aligned} b_i = 1 &\Rightarrow c_i(x) = x_i \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \end{aligned}$$

The second property of this mapping is the simplicity of its Jacobian determinant. Since

$$c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right)$$

we have,  $\forall i, j, x$ ,

$$\begin{aligned} b_i = 1 &\Rightarrow c_i(x) = x_i \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \end{aligned}$$

and

$$b_i = 0 \Rightarrow c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x)$$

The second property of this mapping is the simplicity of its Jacobian determinant. Since

$$c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right)$$

we have,  $\forall i, j, x$ ,

$$\begin{aligned} b_i = 1 &\Rightarrow c_i(x) = x_i \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \end{aligned}$$

and

$$\begin{aligned} b_i = 0 &\Rightarrow c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x) \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = \left( 1_{\{i=j\}} + x_i \underbrace{\frac{\partial s_i(b \odot x)}{\partial x_j}}_{0 \text{ if } b_j=0} \right) \exp(s_i(b \odot x)) + \underbrace{\frac{\partial t_i(b \odot x)}{\partial x_j}}_{0 \text{ if } b_j=0} \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \exp(s_i(b \odot x)) + b_j \left( \dots \right). \end{aligned}$$

The second property of this mapping is the simplicity of its Jacobian determinant. Since

$$c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right)$$

we have,  $\forall i, j, x$ ,

$$\begin{aligned} b_i = 1 &\Rightarrow c_i(x) = x_i \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \end{aligned}$$

and

$$\begin{aligned} b_i = 0 &\Rightarrow c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x) \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = \left( 1_{\{i=j\}} + x_i \underbrace{\frac{\partial s_i(b \odot x)}{\partial x_j}}_{0 \text{ if } b_j=0} \right) \exp(s_i(b \odot x)) + \underbrace{\frac{\partial t_i(b \odot x)}{\partial x_j}}_{0 \text{ if } b_j=0} \\ &\Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \exp(s_i(b \odot x)) + b_j \left( \dots \right). \end{aligned}$$

Hence  $\frac{\partial c_i}{\partial x_j}$  can be non-zero only if  $i = j$ , or  $(1 - b_i)b_j = 1$ .

If we re-order both the rows and columns of the Jacobian to put first the non-zeros entries of  $b$ , and then the zeros, it becomes lower triangular

$$J_c(x) = \begin{pmatrix} 1 & & & & \\ & \ddots & & & (0) \\ & & 1 & & \\ \hline & & & \exp(s_k(x \odot b)) & \\ (\neq 0) & & & & \ddots \\ & & & & \exp(s_{k'}(x \odot b)) \end{pmatrix}$$

its determinant remains unchanged, and we have

$$\begin{aligned} \log |J_{f^{(k)}}(x)| &= \sum_{i:b_i=0} s_i(x \odot b) \\ &= \sum_i ((1 - b) \odot s(x \odot b))_i. \end{aligned}$$

```
dim = 6

x = torch.empty(1, dim).normal_().requires_grad_()
b = torch.zeros(1, dim)
b[:, :dim//2] = 1.0

s = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())
t = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())

c = b * x + (1 - b) * (x * s(b * x).exp() + t(b * x))

j = torch.cat([torch.autograd.grad(c_k, x, retain_graph=True)[0] for c_k in c[0]])

print(j)
```

```

dim = 6

x = torch.empty(1, dim).normal_().requires_grad_()
b = torch.zeros(1, dim)
b[:, :dim//2] = 1.0

s = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())
t = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())

c = b * x + (1 - b) * (x * s(b * x).exp() + t(b * x))

j = torch.cat([torch.autograd.grad(c_k, x, retain_graph=True)[0] for c_k in c[0]])

print(j)

prints

tensor([[ 1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
       [ 0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  0.0000],
       [ 0.0000,  0.0000,  1.0000,  0.0000,  0.0000,  0.0000],
       [ 0.6073,  0.3393,  0.0282,  0.9897,  0.0000,  0.0000],
       [-0.3223, -0.0081,  0.1224,  0.0000,  1.4373,  0.0000],
       [ 0.0301,  0.3185,  0.0665,  0.0000,  0.0000,  1.0295]])
```

To recap, with  $f^{(k)}, k = 1, \dots, K$  coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and  $x_n^{(0)} = x_n$  and  $x_n^{(k)} = f^{(k)}(x_n^{(k-1)})$ ,

To recap, with  $f^{(k)}, k = 1, \dots, K$  coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and  $x_n^{(0)} = x_n$  and  $x_n^{(k)} = f^{(k)}(x_n^{(k-1)})$ , we train by maximizing

$$\mathcal{L}(f) = \sum_n \left( -\frac{1}{2} \left( \|x_n^{(K)}\|^2 + d \log 2\pi \right) + \sum_{k=1}^K \log |J_{f^{(k)}}(x_n^{(k-1)})| \right),$$

with

$$\log |J_{f^{(k)}}(x)| = \sum_i \left( (1 - b^{(k)}) \odot s^{(k)}(x \odot b^{(k)}) \right)_i.$$

To recap, with  $f^{(k)}, k = 1, \dots, K$  coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and  $x_n^{(0)} = x_n$  and  $x_n^{(k)} = f^{(k)}(x_n^{(k-1)})$ , we train by maximizing

$$\mathcal{L}(f) = \sum_n \left( -\frac{1}{2} \left( \|x_n^{(K)}\|^2 + d \log 2\pi \right) + \sum_{k=1}^K \log |J_{f^{(k)}}(x_n^{(k-1)})| \right),$$

with

$$\log |J_{f^{(k)}}(x)| = \sum_i \left( (1 - b^{(k)}) \odot s^{(k)}(x \odot b^{(k)}) \right)_i.$$

And to sample we just need to generate  $Z \sim \mathcal{N}(0, I)$  and compute  $f^{-1}(Z)$ .

A coupling layer can be implemented with

```
class NVPCouplingLayer(nn.Module):
    def __init__(self, map_s, map_t, b):
        super(NVPcouplingLayer, self).__init__()
        self.map_s = map_s
        self.map_t = map_t
        self.b = b.clone().unsqueeze(0)

    def forward(self, x_and_logdetjac):
        x, logdetjac = x_and_logdetjac
        s, t = self.map_s(self.b * x), self.map_t(self.b * x)
        logdetjac += ((1 - self.b) * s).sum(1)
        y = self.b * x + (1 - self.b) * (torch.exp(s) * x + t)
        return (y, logdetjac)

    def invert(self, x):
        s, t = self.map_s(self.b * x), self.map_t(self.b * x)
        return self.b * x + (1 - self.b) * (torch.exp(-s) * (x - t))
```

The `forward` here computes both the image of  $x$  and the update on the accumulated determinant of the Jacobian, *i.e.*

$$(x, u) \mapsto (f(x), u + |J_f(x)|).$$

We can then define a complete network with one-hidden layer tanh MLPs for the  $s$  and  $t$  mappings

```
class NVPNet(nn.Module):
    def __init__(self, dim, hidden_dim, depth):
        super(NVPNet, self).__init__()
        b = torch.empty(dim)
        self.layers = nn.ModuleList()
        for d in range(depth):
            if d%2 == 0:
                i = torch.randperm(b.numel())[0:b.numel() // 2]
                b.zero_()[i] = 1
            else:
                b = 1 - b
            map_s = nn.Sequential(nn.Linear(dim, hidden_dim), nn.Tanh(),
                                  nn.Linear(hidden_dim, dim))
            map_t = nn.Sequential(nn.Linear(dim, hidden_dim), nn.Tanh(),
                                  nn.Linear(hidden_dim, dim))
            self.layers.append(NVPCouplingLayer(map_s, map_t, b))

    def forward(self, x_and_logdetjac):
        for m in self.layers: x_and_logdetjac = m(x_and_logdetjac)
        return x_and_logdetjac

    def invert(self, x):
        for m in reversed(self.layers): x = m.invert(x)
        return x
```

And the log-proba of individual samples of a batch

```
def LogProba(x_and_logdetjac):
    (x, logdetjac) = x_and_logdetjac
    log_p = logdetjac - 0.5 * x.pow(2).add(math.log(2 * math.pi)).sum(1)
    return log_p
```

Training is achieved by maximizing the mean log-proba

```
batch_size = 100

model = NVPNet(dim = 2, hidden_dim = 2, depth = 4)
optimizer = optim.Adam(model.parameters(), lr = 1e-2)

for e in range(args.nb_epochs):

    acc_loss = 0

    for b in range(0, nb_train_samples, batch_size):
        output = model((input[b:b+batch_size], 0))
        loss = - LogProba(output).mean()
        acc_loss = acc_loss + loss.item()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Training is achieved by maximizing the mean log-proba

```
batch_size = 100

model = NVPNet(dim = 2, hidden_dim = 2, depth = 4)
optimizer = optim.Adam(model.parameters(), lr = 1e-2)

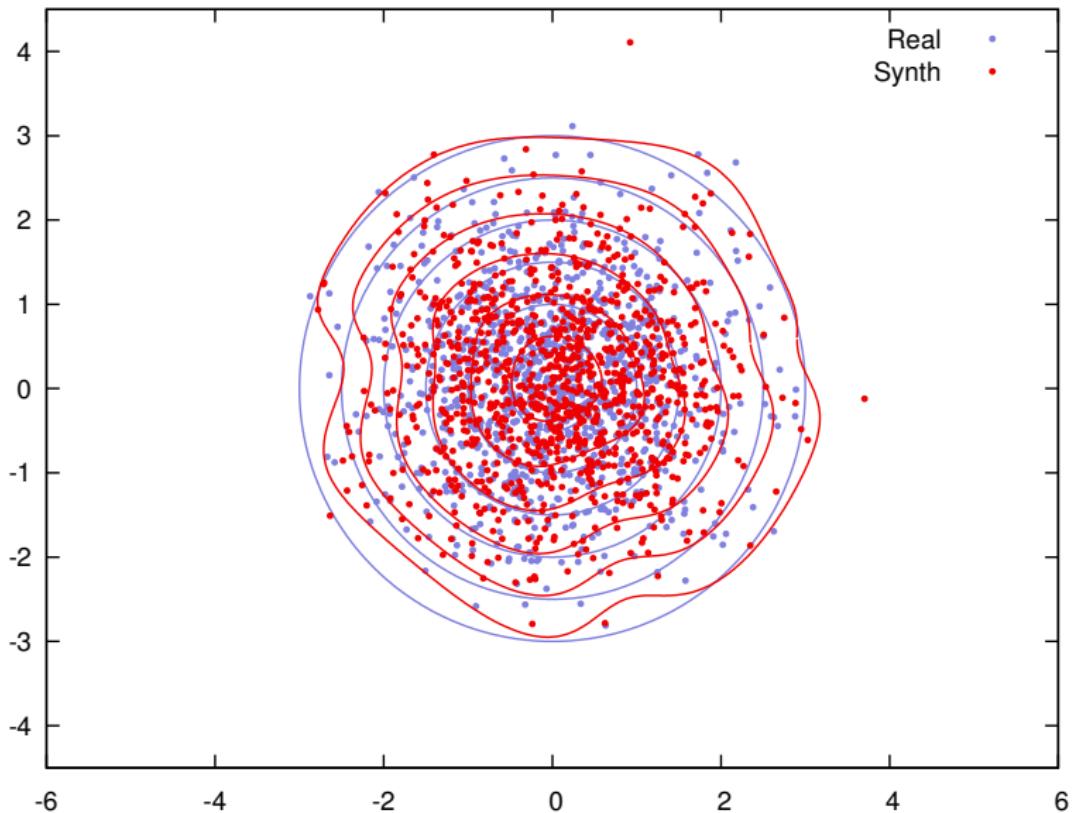
for e in range(args.nb_epochs):

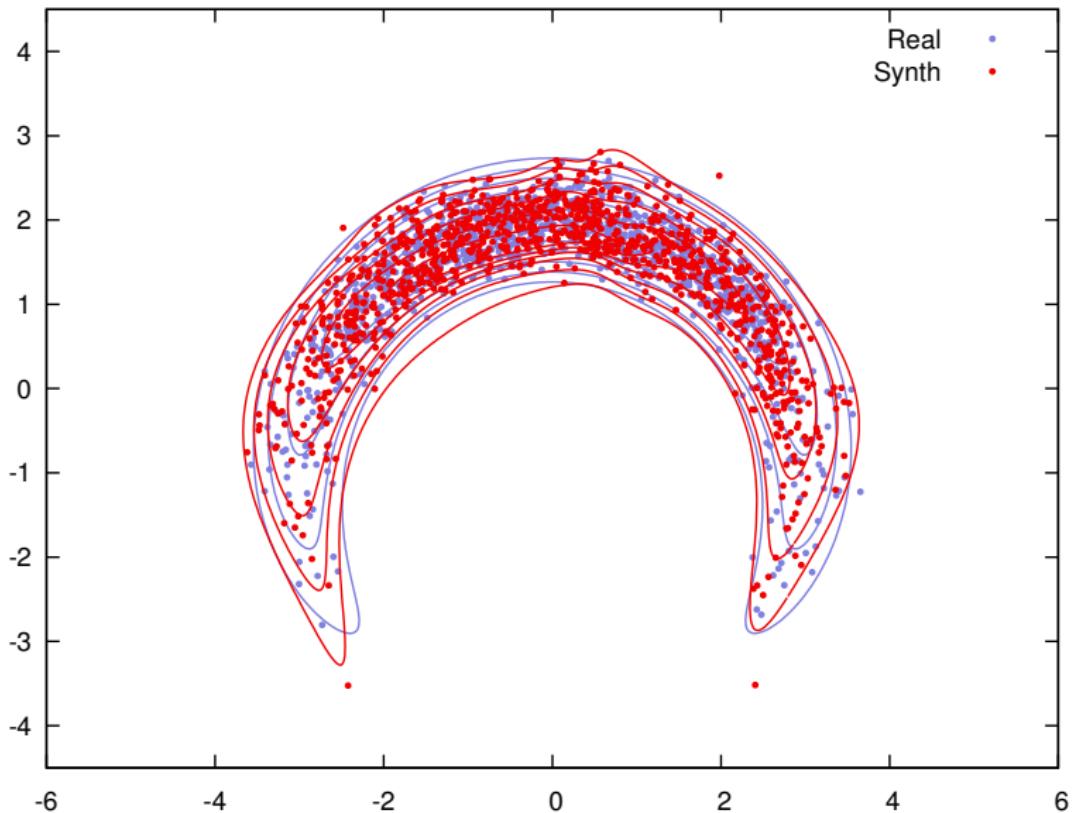
    acc_loss = 0

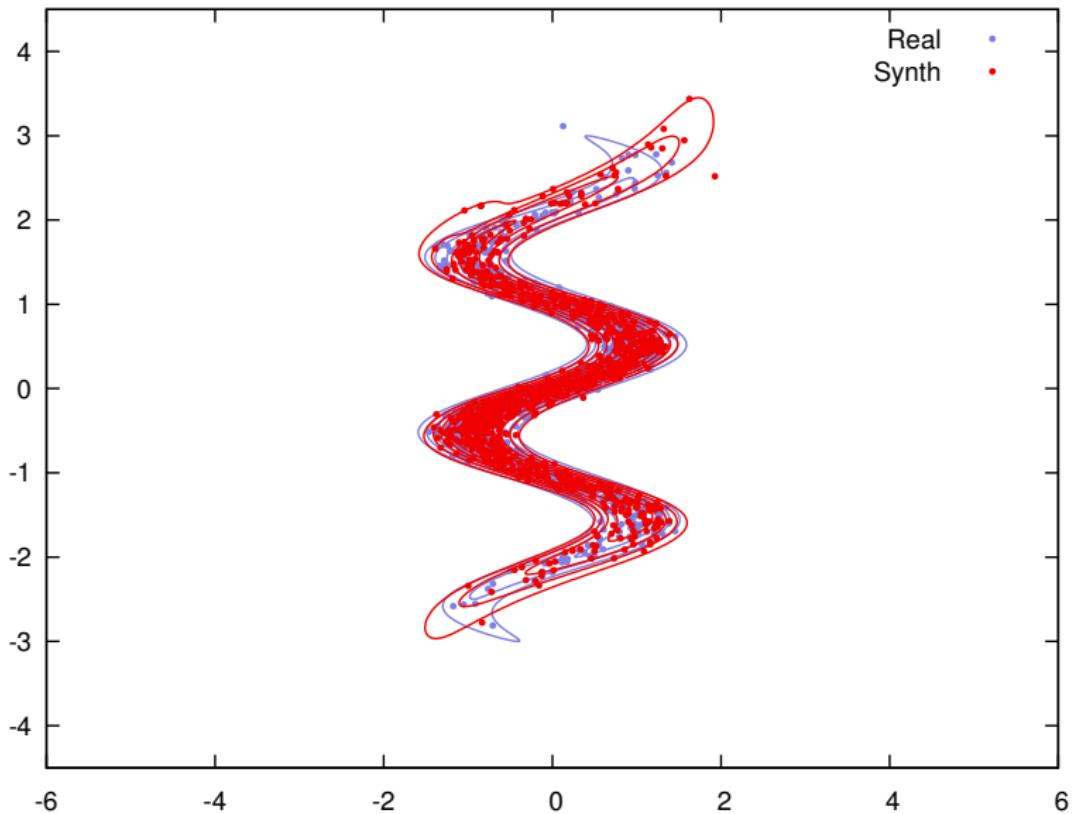
    for b in range(0, nb_train_samples, batch_size):
        output = model((input[b:b+batch_size], 0))
        loss = - LogProba(output).mean()
        acc_loss = acc_loss + loss.item()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

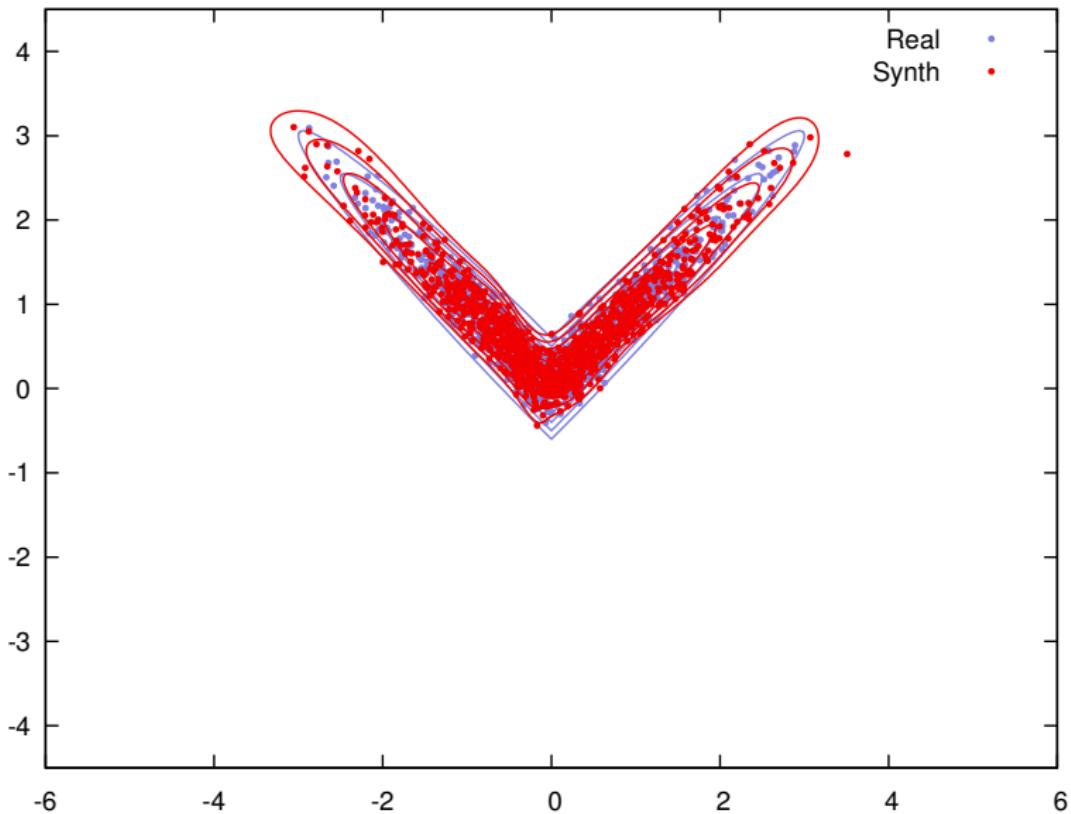
Finally, we can sample according to  $p_{f^{-1}(Z)}$  with

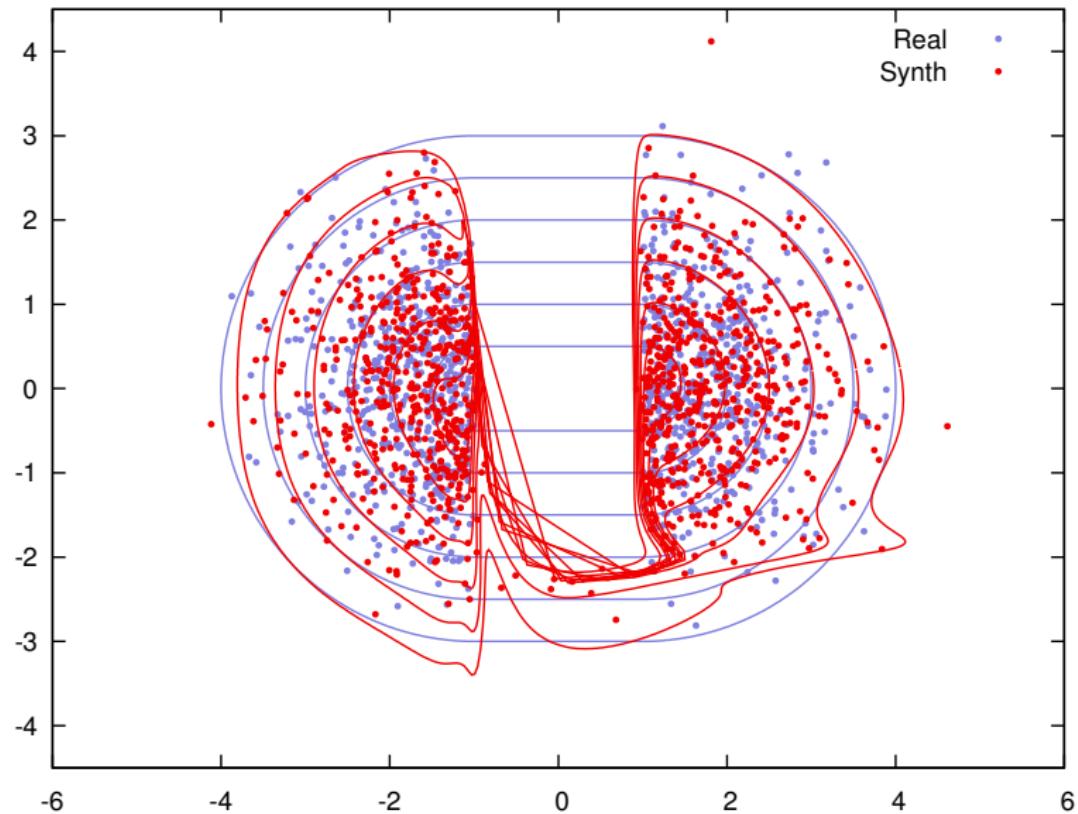
```
z = torch.empty(nb_train_samples, 2).normal_()
x = model.invert(z)
```











Dinh et al. (2016) apply this approach to convolutional layers by using *bs* consistent with the activation map structure, and reducing the map size while increasing the number of channels.

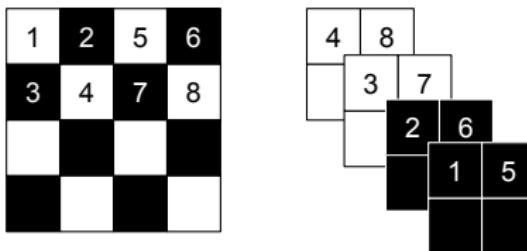
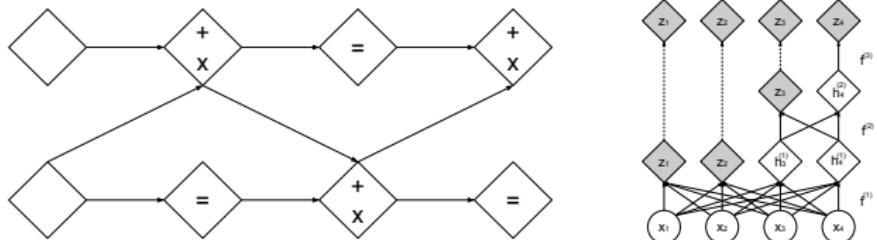


Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the  $4 \times 4 \times 1$  tensor (on the left) into a  $2 \times 2 \times 4$  tensor (on the right). Before the squeezing operation, a checkerboard pattern is used for coupling layers while a channel-wise masking pattern is used afterward.

(Dinh et al., 2016)

They combine these layers by alternating masks, and branching out half of the channels at certain points to forward them unchanged.



(a) In this alternating pattern, units which remain identical in one transformation are modified in the next.

(b) Factoring out variables.  
At each step, half the variables are directly modeled as Gaussians, while the other half undergo further transformation.

Figure 4: Composition schemes for affine coupling layers.

(Dinh et al., 2016)

The structure for generating images consists of

- $\times 2$  stages
  - $\times 3$  checkerboard coupling layers,
  - a squeezing layer,
  - $\times 3$  channel coupling layers,
  - a factor-out layer.
- $\times 1$  stage
  - $\times 4$  checkerboard coupling layers
  - a factor-out layer.

The  $s$  and  $t$  mappings get more complex in the later layers.

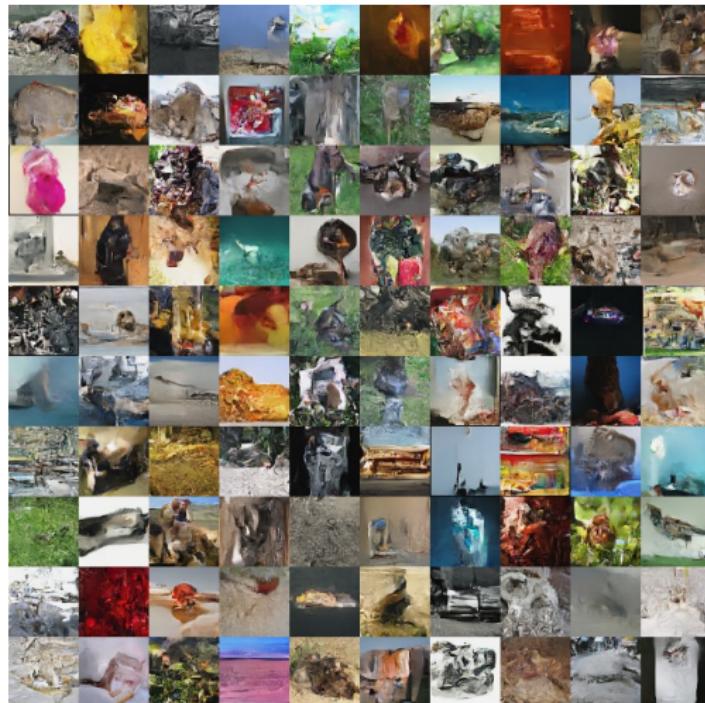


Figure 7: Samples from a model trained on *Imagenet* ( $64 \times 64$ ).

(Dinh et al., 2016)



Figure 8: Samples from a model trained on *CelebA*.

(Dinh et al., 2016)

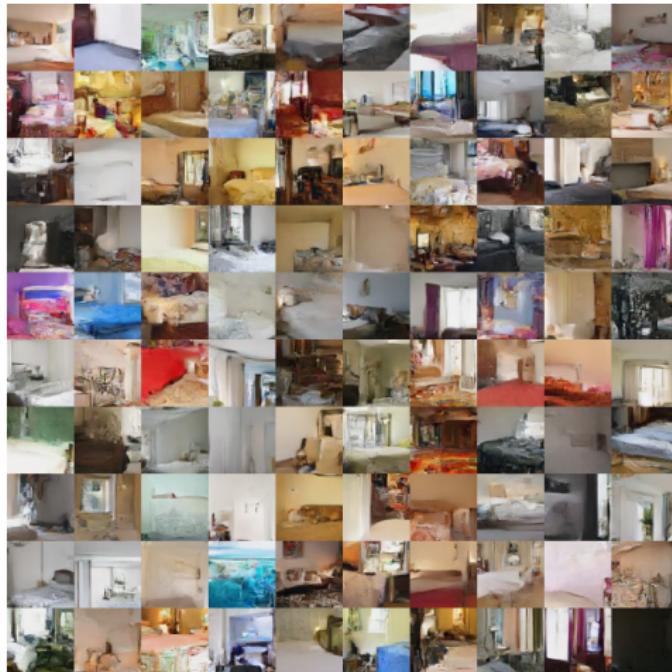


Figure 9: Samples from a model trained on *LSUN* (*bedroom* category).

(Dinh et al., 2016)



Figure 10: Samples from a model trained on *LSUN* (*church outdoor* category).

(Dinh et al., 2016)

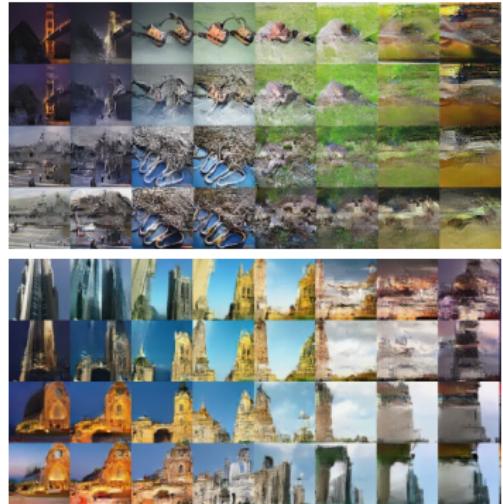
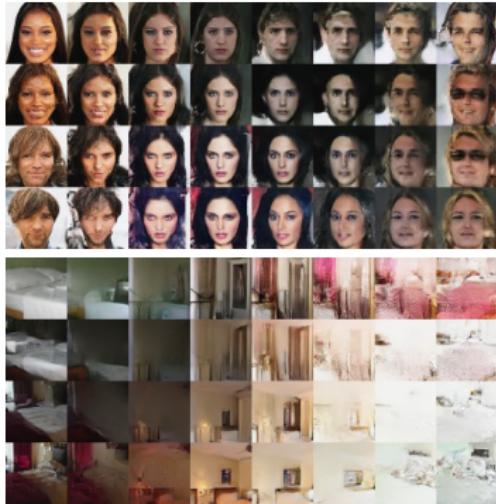


Figure 6: Manifold generated from four examples in the dataset. Clockwise from top left: CelebA, Imagenet ( $64 \times 64$ ), LSUN (tower), LSUN (bedroom).

(Dinh et al., 2016)

The end

## References

- L. Dinh, D. Krueger, and Y. Bengio. NICE: non-linear independent components estimation. *CoRR*, abs/1410.8516, 2014.
- L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real NVP. *CoRR*, abs/1605.08803, 2016.
- D. Rezende and S. Mohamed. Variational inference with normalizing flows. *CoRR*, abs/1505.05770, 2015.