EE-559 – Deep learning

9. Autoencoders and generative models
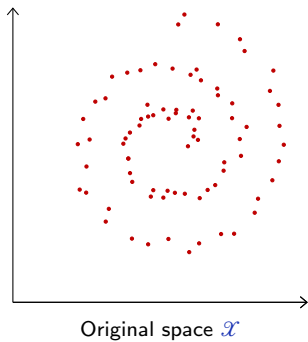
François Fleuret

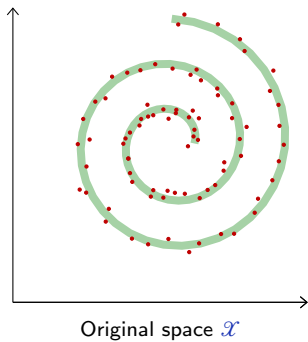https://fleuret.org/dlc/

[version of: May 17, 2018]

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.
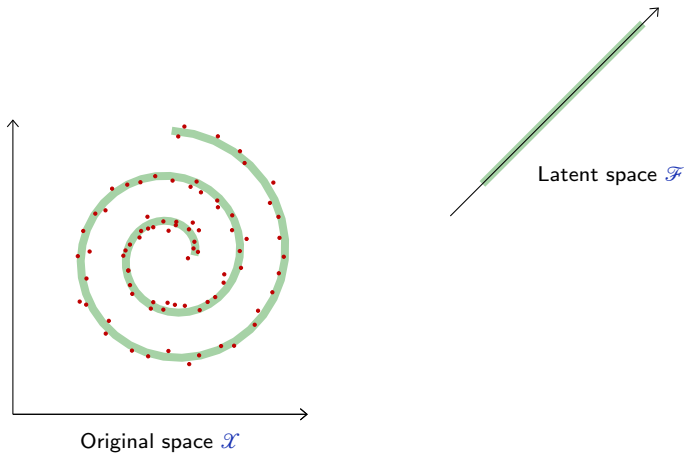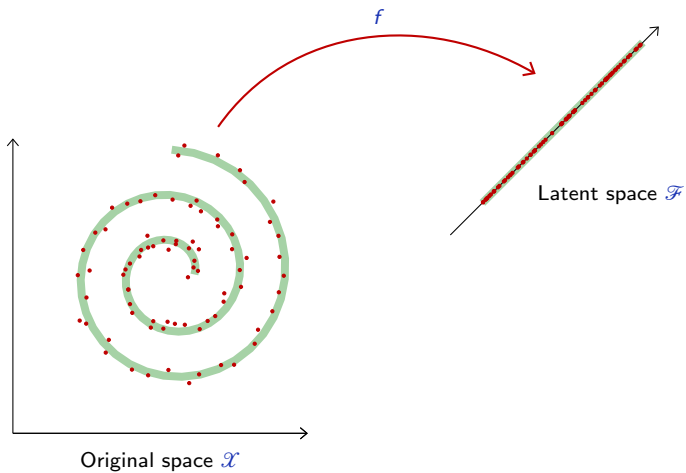
Many applications such as image synthesis, denoising, super-resolution, speech synthesis, compression, etc. require to go beyond classification and regression, and model explicitly a high dimension signal.
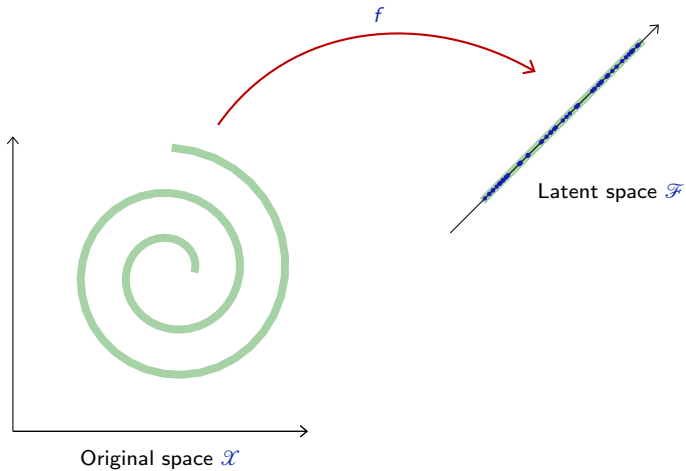
This modeling consists of finding "meaningful degrees of freedom" that describe the signal, and are of lesser dimension.

Original space $\mathcal{X}$

Original space $\mathcal{X}$

Original space $\mathscr{X}$

Latent space $\mathscr{F}$

f

Latent space $\mathscr{F}$

Original space $\mathscr{X}$

Original space $\mathcal{X}$

Latent space $\mathcal{F}$
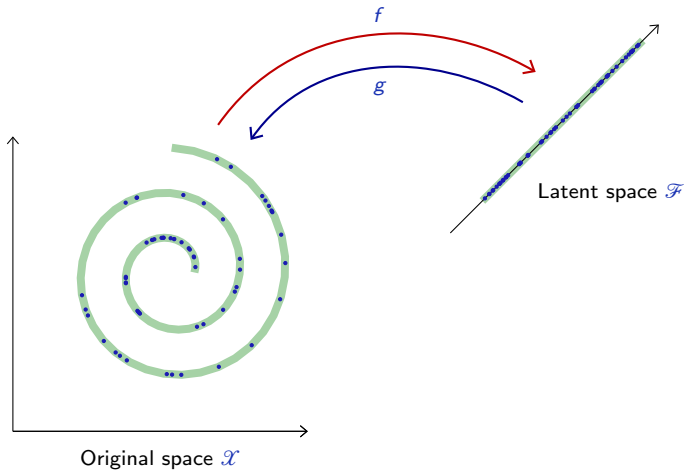
f

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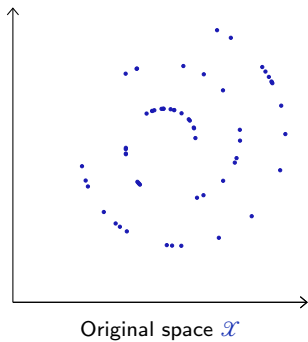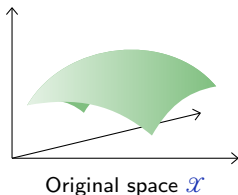
When dealing with real-world signals, this objective involves the same theoretical and practical issues as for classification or regression: defining the right class of high-dimension models, and optimizing them.

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

Autoencoders

An autoencoder combines an **encoder** f that embeds the original space $\mathscr{X}$ into a **latent** space of lower dimension $\mathscr{F}$, and a **decoder** g to map back to $\mathscr{X}$, such that their composition $g \circ f$ is [close to] the identity on the data.

An autoencoder combines an **encoder** f that embeds the original space $\mathcal{X}$ into a **latent** space of lower dimension $\mathcal{F}$, and a **decoder** g to map back to $\mathcal{X}$, such that their composition $g \circ f$ is [close to] the identity on the data.
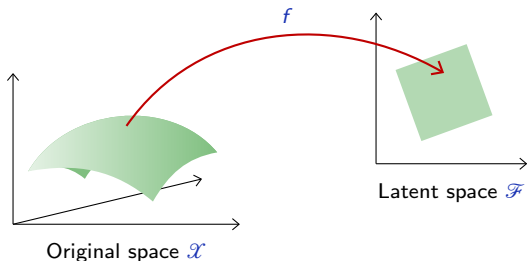

Original space $\mathcal{X}$

An autoencoder combines an **encoder** f that embeds the original space $\mathcal{X}$ into a **latent** space of lower dimension $\mathcal{F}$, and a **decoder** g to map back to $\mathcal{X}$, such that their composition $g \circ f$ is [close to] the identity on the data.

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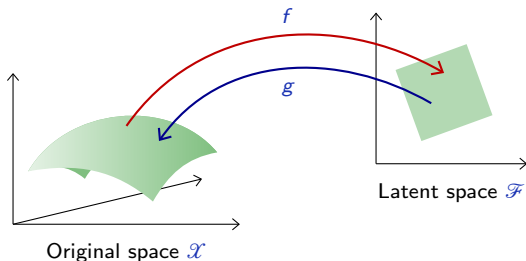
Original space $\mathcal{X}$

An autoencoder combines an **encoder** f that embeds the original space $\mathcal{X}$ into a **latent** space of lower dimension $\mathcal{F}$, and a **decoder** g to map back to $\mathcal{X}$, such that their composition $g \circ f$ is [close to] the identity on the data.
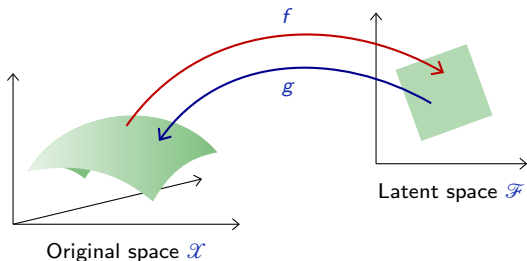
An autoencoder combines an **encoder** f that embeds the original space $\mathscr{X}$ into a **latent** space of lower dimension $\mathscr{F}$, and a **decoder** g to map back to $\mathscr{X}$, such that their composition $g \circ f$ is [close to] the identity on the data.

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 MSE loss

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

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

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

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

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

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

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

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 $\mathscr{X}$. A good autoencoder could be characterized with the MSE loss

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

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.

Transposed convolutions

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

Constructing deep generative architectures, such as the decoder of an autoencoder, 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 expend the signal from a low-dimension representation to the high-dimension signal space.

Constructing deep generative architectures, such as the decoder of an autoencoder, 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 expend 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 backward pass.

Consider a 1d convolution with a kernel κ

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

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

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

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

$$\left[\frac{\partial \ell}{\partial x}\right] = \left[\frac{\partial \ell}{\partial y}\right] * \kappa.$$
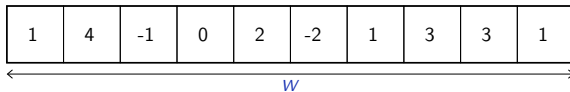
In the deep-learning field, since it corresponds to transposing the weight matrix of the equivalent fully-connected layer, it is called a **transposed convolution**.

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

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.
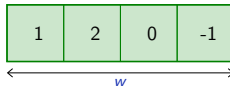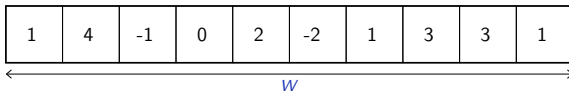
Convolution layer

Input

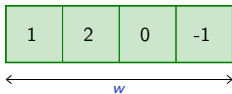| 1	4	-1	0	2	-2	1	3	3	1

$\longleftarrow \quad W \quad \longrightarrow$

Convolution layer

Input

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

W

Kernel

| 1 | 2 | 0 | -1 |

w

Convolution layer

Input

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

$\longleftarrow \qquad\qquad W \qquad\qquad \longrightarrow$

1	2	0	-1

$\longleftarrow \quad w \quad \longrightarrow$

Output

9							

$\longleftarrow \qquad W - w + 1 \qquad \longrightarrow$

Convolution layer

Input

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

W

| 1 | 2 | 0 | -1 |

w

Output

| 9 | 0 | | | | | | |

$W - w + 1$

Convolution layer

Input

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

$\longleftrightarrow$
W

| 1 | 2 | 0 | -1 |

$\longleftrightarrow$
w

Output

| 9 | 0 | 1 | | | | |

$\longleftrightarrow$
$W - w + 1$

Convolution layer

Input

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

$\xleftarrow{\hspace{5cm}}$ W $\xrightarrow{\hspace{5cm}}$

| 1 | 2 | 0 | -1 |

$\xleftarrow{\hspace{2cm}}$ w $\xrightarrow{\hspace{2cm}}$

Output

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

$\xleftarrow{\hspace{5cm}}$ $W - w + 1$ $\xrightarrow{\hspace{5cm}}$

Convolution layer

Input

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

W

1	2	0	-1

w

Output

9	0	1	3	-5					

$W - w + 1$

Convolution layer

Convolution layer

Convolution layer

Input

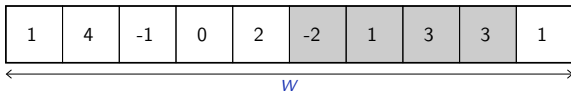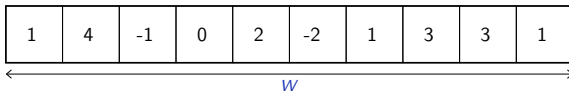| 1	4	-1	0	2	-2	1	3	3	1

$\longleftarrow \qquad\qquad W \qquad\qquad \longrightarrow$

Kernel

1	2	0	-1

$\longleftarrow \quad w \quad \longrightarrow$

Output

9	0	1	3	-5	-3	6

$\longleftarrow \qquad W - w + 1 \qquad \longrightarrow$

Transposed convolution layer

Input

2	3	0	-1

$\longleftrightarrow$
W

Kernel

1	2	-1

$\longleftrightarrow$
w

Transposed convolution layer

Input

| 2 | 3 | 0 | -1 |

W

| 1 | 2 | -1 |

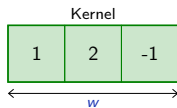| 2 | 4 | -2 |

Output

| 2 | |

$W + w - 1$
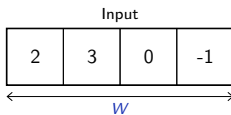
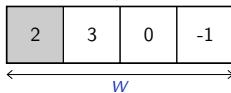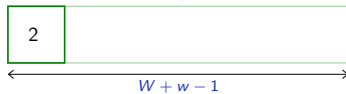# Transposed convolution layer

Transposed convolution layer

Transposed convolution layer

Transposed convolution layer

Input
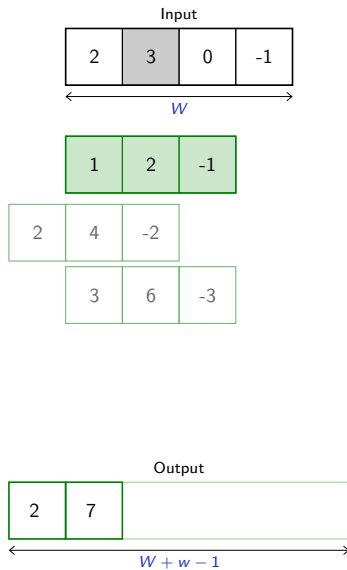
2	3	0	-1

W

2	4	-2
3	6	-3
0	0	0
-1	-2	1

Output

2	7	4	-4	-2	1

$W + w - 1$

Transposed convolution layer

Input

2	3	0	-1

W

Kernel

1	2	-1

w

Output

2	7	4	-4	-2	1

$W + w - 1$

`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 = Variable(Tensor([[[0, 0, 1, 0, 0, 0, 0]]]))
>>> k = Variable(Tensor([[[1, 2, 3]]]))
>>> F.conv1d(x, k)
Variable containing:
(0 ,.,.) =
  3  2  1  0  0
[torch.FloatTensor of size 1x1x5]
```

`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 = Variable(Tensor([[[0, 0, 1, 0, 0, 0, 0]]]))
>>> k = Variable(Tensor([[[1, 2, 3]]]))
>>> F.conv1d(x, k)
Variable containing:
(0 ,.,.) =
  3  2  1  0  0
[torch.FloatTensor of size 1x1x5]
```



```
>>> F.conv_transpose1d(x, k)
Variable containing:
(0 ,.,.) =
  0  0  1  2  3  0  0  0  0
[torch.FloatTensor of size 1x1x9]
```
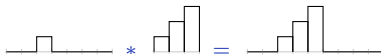
The class `torch.nn.ConvTranspose1d` implements that operation into a `torch.nn.Module`.

```
>>> x = Variable(Tensor([[[ 2, 3, 0, -1]]]))
>>> m = nn.ConvTranspose1d(1, 1, kernel_size=3)
>>> m.bias.data.zero_()

 0
[torch.FloatTensor of size 1]

>>> m.weight.data.copy_(Tensor([ 1, 2, -1 ]))

(0 ,.,.) =
  1   2  -1
[torch.FloatTensor of size 1x1x3]

>>> y = m(x)
>>> y
Variable containing:
(0 ,.,.) =
  2   7   4  -4  -2   1
[torch.FloatTensor of size 1x1x6]
```

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 $= 2$)
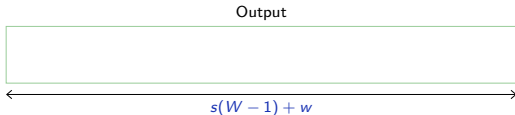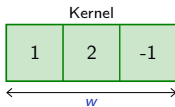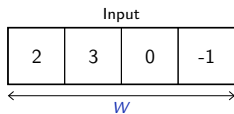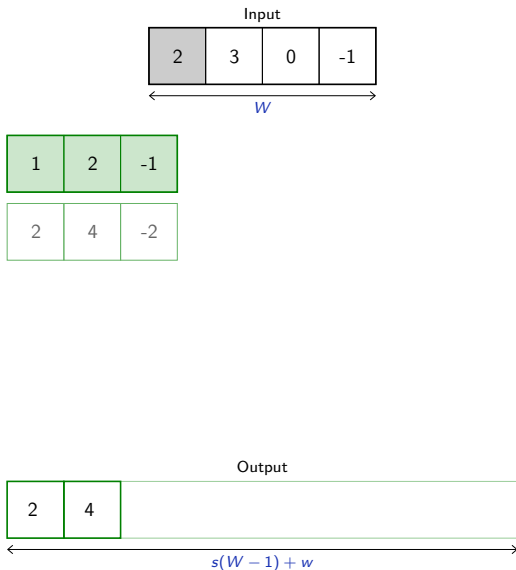
Input

2	3	0	-1

$\longleftrightarrow$
W

Kernel

1	2	-1

$\longleftrightarrow$
w

Output

$\longleftrightarrow$
$s(W - 1) + w$

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 $= 2$)

Input

2	3	0	-1

W

Kernel

1	2	-1

w

Output

2	4	1	6	-3	0	-1	-2	1

$s(W-1) + w$
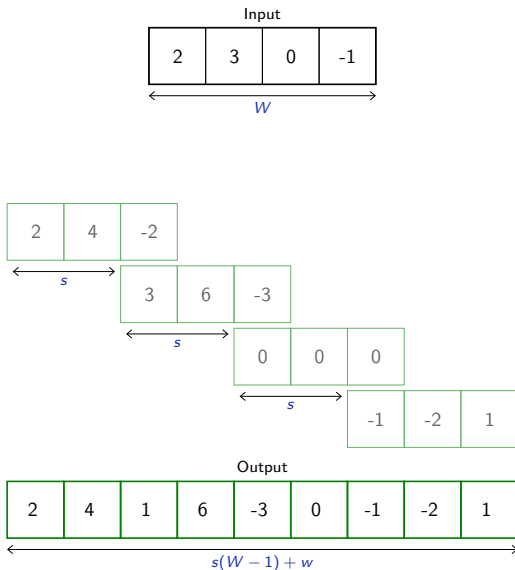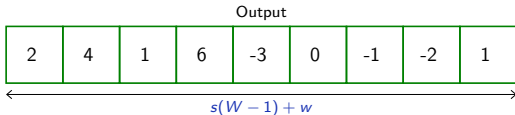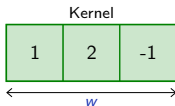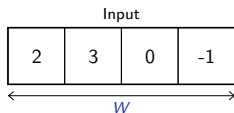
The composition of a convolution and a transposed convolution of same parameters keep the signal size [roughly] unchanged.

⚠ 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

$$\exists q \in \mathbb{N}, \ W = w + s\,q.$$
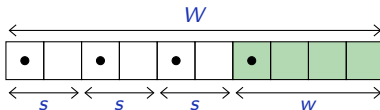
It has been observed that transposed convolutions may create some grid-structure artifact, since generated pixels are not all covered similarly.

For instance with a 4×4 kernel and stride 3
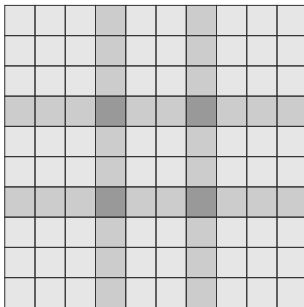
An alternative is to use an analytic up-scaling. Two standard such PyTorch modules are `nn.UpsamplingBilinear2d` and `nn.Upsample`.

```
>>> x = Variable(Tensor([[[[ 1, 2 ], [ 3, 4 ]]]]))
>>> b = nn.UpsamplingBilinear2d(scale_factor = 3)
>>> b(x)
Variable containing:
(0 ,0 ,.,.) =
  1.0000  1.2000  1.4000  1.6000  1.8000  2.0000
  1.4000  1.6000  1.8000  2.0000  2.2000  2.4000
  1.8000  2.0000  2.2000  2.4000  2.6000  2.8000
  2.2000  2.4000  2.6000  2.8000  3.0000  3.2000
  2.6000  2.8000  3.0000  3.2000  3.4000  3.6000
  3.0000  3.2000  3.4000  3.6000  3.8000  4.0000
[torch.FloatTensor of size 1x1x6x6]
```

An alternative is to use an analytic up-scaling. Two standard such PyTorch modules are `nn.UpsamplingBilinear2d` and `nn.Upsample`.

```
>>> x = Variable(Tensor([[[[ 1, 2 ], [ 3, 4 ]]]]))
>>> b = nn.UpsamplingBilinear2d(scale_factor = 3)
>>> b(x)
Variable containing:
(0 ,0 ,.,.) =
  1.0000  1.2000  1.4000  1.6000  1.8000  2.0000
  1.4000  1.6000  1.8000  2.0000  2.2000  2.4000
  1.8000  2.0000  2.2000  2.4000  2.6000  2.8000
  2.2000  2.4000  2.6000  2.8000  3.0000  3.2000
  2.6000  2.8000  3.0000  3.2000  3.4000  3.6000
  3.0000  3.2000  3.4000  3.6000  3.8000  4.0000
[torch.FloatTensor of size 1x1x6x6]

>>> u = nn.Upsample(scale_factor = 3, mode = 'nearest')
>>> u(x)
Variable containing:
(0 ,0 ,.,.) =
  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
[torch.FloatTensor of size 1x1x6x6]
```

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

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

can be replaced by

```
nn.UpsamplingBilinear2d(scale_factor = 2)
nn.Conv2d(nic, noc, kernel_size = 3, padding = 1)
```

or

```
nn.Upsample(scale_factor = 2, mode = 'nearest')
nn.Conv2d(nic, noc, kernel_size = 3, padding = 1)
```

Deep Autoencoders

A deep autoencoder combines an encoder composed of convolutional layers, and a decoder composed of the reciprocal transposed convolution layers.

A deep autoencoder combines an encoder composed of convolutional layers, and a decoder composed of the reciprocal transposed convolution layers.

To run a simple example on MNIST, we consider the following model, where dimension reduction is obtained through filter sizes and strides > 1, avoiding max-pooling layers.

```
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)

$$8 \times 1 \times 1$$
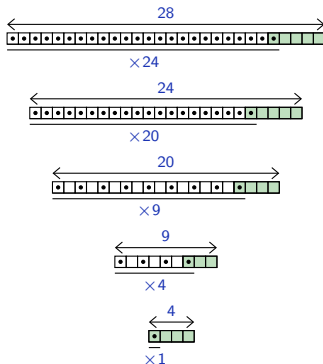
Decoder

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)`

$1 \times 28 \times 28$
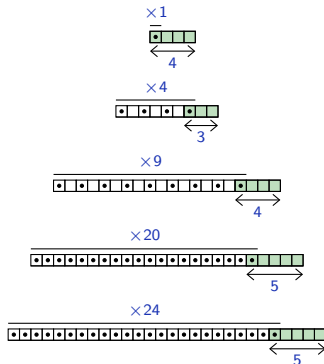
Training is achieved with MSE and Adam

```
model = AutoEncoder(embedding_dim, nb_channels)
mse_loss = nn.MSELoss()

if torch.cuda.is_available():
    model.cuda()
    mse_loss.cuda()

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

for epoch in range(args.nb_epochs):
    for input, _ in iter(train_loader):
        if torch.cuda.is_available(): input = input.cuda()
        input = Variable(input)
        output = model(input)
        loss = mse_loss(output, input)
        model.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

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

Original space $\mathcal{X}$

Latent space $\mathcal{F}$

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

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

Latent space $\mathcal{F}$

Original space $\mathcal{X}$

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

$$\forall x, x' \in \mathscr{X}^2, \ \alpha \in [0, 1], \ \ \xi(x, x', \alpha) = g((1 - \alpha)f(x) + \alpha f(x')).$$
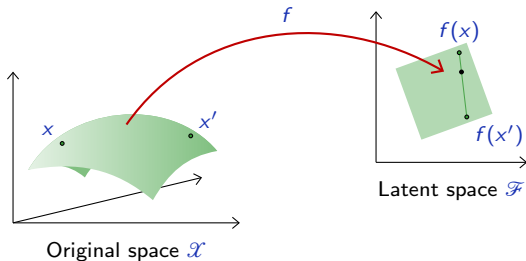
To get an intuition of the latent representation, we can pick two samples x and x' at random and interpolate samples along the line in the latent space

$$\forall x, x' \in \mathcal{X}^2, \ \alpha \in [0, 1], \ \ \xi(x, x', \alpha) = g((1-\alpha)f(x) + \alpha f(x')).$$
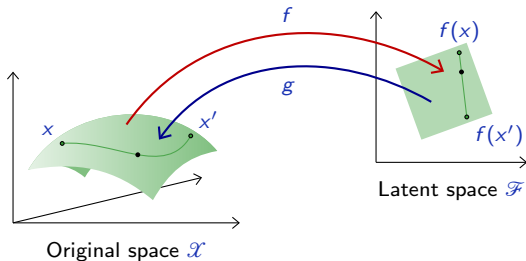
Autoencoder interpolation ($d = 8$)
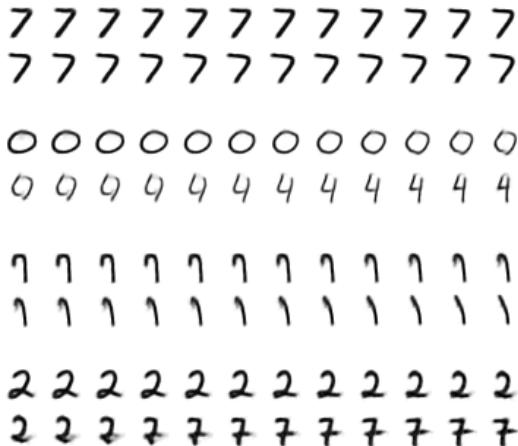
Autoencoder interpolation ($d = 32$)

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

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

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

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 $\mathscr{F}$, sample there, and map the samples into the image space $\mathscr{X}$ 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.

Latent space $\mathscr{F}$

Original space $\mathscr{X}$

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

Latent space $\mathscr{F}$

Original space $\mathscr{X}$

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

Latent space $\mathscr{F}$

Original space $\mathscr{X}$

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

We can for instance use a Gaussian model with diagonal covariance matrix.
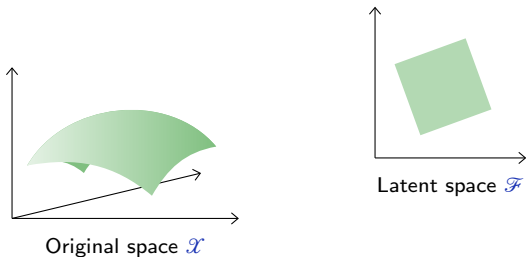
$$f(X) \sim \mathscr{N}(\hat{m}, \hat{\Delta})$$

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

Latent space $\mathscr{F}$

Original space $\mathscr{X}$
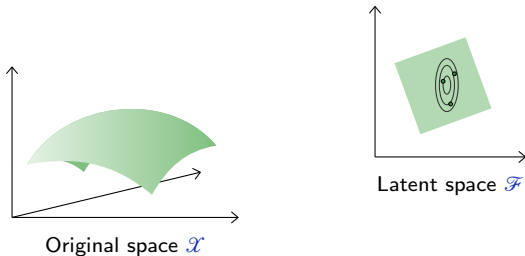
Autoencoder sampling ($d = 8$)

Autoencoder sampling ($d = 16$)

Autoencoder sampling ($d = 32$)
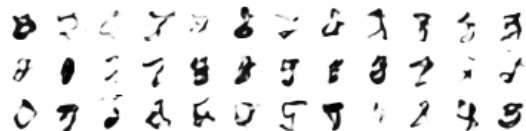
These results are unsatisfying, because the density model used on the latent space $\mathscr{F}$ is too simple and inadequate.

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

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

We have

$$
\begin{aligned}
\operatorname*{argmax}_{\theta} \; \mathbb{I}(X, Z) &= \operatorname*{argmax}_{\theta} \; \mathbb{H}(X) - \mathbb{H}(X \mid Z) \\
&= \operatorname*{argmax}_{\theta} \; -\mathbb{H}(X \mid Z) \\
&= \operatorname*{argmax}_{\theta} \; \mathbb{E}\Big[\log q_\theta(X \mid Z) \Big].
\end{aligned}
$$

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

For any distribution p we have

$$\mathbb{E}\Big[\log q_\theta(X \mid Z) \Big] \geq \mathbb{E}\Big[\log p(X \mid Z) \Big].$$

For any distribution p we have

$$\mathbb{E}\Big[\log q_\theta(X \mid Z) \Big] \geq \mathbb{E}\Big[\log p(X \mid Z) \Big].$$

So we can in particular approximate the left term with the right one by optimizing a reconstruction model p_η to make the inequality tight.

If we consider the following model for p

$$p_\eta \left(\cdot \mid Z = z \right) \sim \mathcal{N}(g(z), \sigma)$$

where g is deterministic,

If we consider the following model for p

$$p_\eta \left(\cdot \mid Z = z \right) \sim \mathcal{N}(g(z), \sigma)$$

where g is deterministic, we get

$$\mathbb{E}\Big[\log p_\eta(X \mid Z) \Big] = -\mathbb{E}\left[\frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right]$$
$$= -\mathbb{E}\left[\frac{\|X - g(f(X; \theta); \eta)\|^2}{2\sigma^2} \right].$$

If we consider the following model for p

$$p_\eta \left(\cdot \mid Z = z \right) \sim \mathcal{N}(g(z), \sigma)$$

where g is deterministic, we get

$$\mathbb{E}\Big[\log p_\eta(X \mid Z) \Big] = -\mathbb{E}\left[\frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right]$$
$$= -\mathbb{E}\left[\frac{\|X - g(f(X; \theta); \eta)\|^2}{2\sigma^2} \right].$$

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

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

If we consider the following model for p

$$p_\eta \left(\cdot \mid Z = z \right) \sim \mathcal{N}(g(z), \sigma)$$

where g is deterministic, we get

$$\mathbb{E}\Big[\log p_\eta(X \mid Z) \Big] = -\mathbb{E}\left[\frac{\|X - g(Z; \eta)\|^2}{2\sigma^2} \right]$$
$$= -\mathbb{E}\left[\frac{\|X - g(f(X; \theta); \eta)\|^2}{2\sigma^2} \right].$$

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

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

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.

Reducing dimension, or forcing sparsity is a way to push the model to maximize retained information in a constraint coding space.

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

Reducing dimension, or forcing sparsity is a way to push the model to maximize retained information in a constraint coding space.

In their work, Vincent et al. proposed to degrade the signal with noise before feeding it to the encoder, but to keep the MSE to the original signal. This forces the encoder to retain meaningful structures.
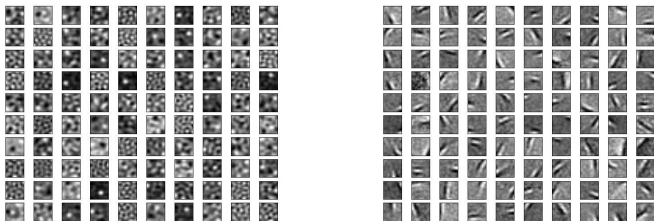
Figure 6: Weight decay vs. Gaussian noise. We show typical filters learnt from natural image patches in the over-complete case (200 hidden units). *Left:* regular autoencoder with weight decay. We tried a wide range of weight-decay values and learning rates: filters never appeared to capture a more interesting 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)
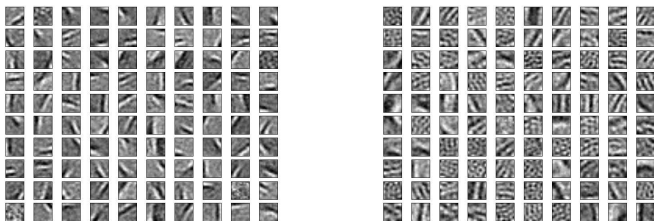
Figure 7: Filters obtained on natural image patches by denoising autoencoders using other noise types. *Left:* with 10% salt-and-pepper noise, we obtain oriented Gabor-like filters. They appear slightly less localized than when using Gaussian noise (contrast with Figure 6 right). *Right:* with 55% zero-masking 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.

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.

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.

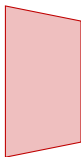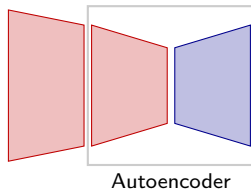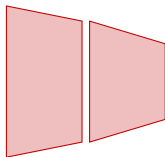A final classifying layer is added and the full structure can be fine-tuned.

Data Set	SVM$_{rbf}$	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)

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

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

We should ideally look for the g that maximizes the [empirical] log-likelihood

$$\frac{1}{N} \sum_n \log p(x_n).$$

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 on it.

We should ideally look for the g that maximizes the [empirical] log-likelihood

$$\frac{1}{N} \sum_n \log p(x_n).$$

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.

Their framework considers a **stochastic** encoder f, and decoder g, whose outputs depend on their inputs as usual but with some remaining randomness.

We can equivalently maximize (all expectations are under p):

$$\mathcal{S} = \mathbb{E}\Big[\log p(X) - \log q(X)\Big]$$

$$= \mathbb{E}\left[\log p(X, Z) + \log \frac{p(X)}{p(X, Z)} - \log q(X, Z) - \log \frac{q(X)}{q(X, Z)}\right]$$

$$= \mathbb{E}\left[\log \frac{p(X, Z)}{q(X, Z)} - \log \frac{p(Z \mid X)}{q(Z \mid X)}\right]$$

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

We can equivalently maximize (all expectations are under p):

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

While the second term cannot be properly estimated, we can use the first as a lower bound, and rewrite it

$$\begin{aligned}
\mathcal{S} &\geq -\mathbb{D}_{KL}(q(X, Z) \,\|\, p(X, Z)) \\
&= -\mathbb{E}\Big[\mathbb{D}_{KL}(q(Z \mid X) \,\|\, p(Z))\Big] + \mathbb{E}\Big[\log p(X \mid Z)\Big].
\end{aligned}$$

We can equivalently maximize (all expectations are under p):

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

While the second term cannot be properly estimated, we can use the first as a lower bound, and rewrite it

$$\begin{aligned}
\mathcal{S} &\geq -\mathbb{D}_{KL}(q(X,Z)\,\|\,p(X,Z)) \\
&= -\mathbb{E}\Big[\mathbb{D}_{KL}(q(Z\mid X)\,\|\,p(Z))\Big] + \mathbb{E}\Big[\log p(X\mid Z)\Big].
\end{aligned}$$

It pushes to match the joint distributions on (X, Z) while matching the marginals on X alone would be enough.

Kingma and Welling use Gaussians with diagonal covariance for both $q(Z \mid X)$ and $p(X \mid Z)$.

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

$$f : \mathbb{R}^c \to \mathbb{R}^{2d}$$

$$x \mapsto \left(\mu_1^f, \ldots, \mu_d^f, \sigma_1^f, \ldots, \sigma_d^f \right),$$

Kingma and Welling use Gaussians with diagonal covariance for both $q(Z \mid X)$ and $p(X \mid Z)$.

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

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

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

$$g : \mathbb{R}^d \to \mathbb{R}^{2c}$$
$$z \mapsto \left(\mu_1^g, \ldots, \mu_c^g, \sigma_1^g, \ldots, \sigma_c^g \right).$$

Original space $\mathcal{X}$

Latent space $\mathcal{F}$

Original space $\mathscr{X}$

Latent space $\mathscr{F}$

f

Latent space $\mathscr{F}$

Original space $\mathscr{X}$

Original space $\mathcal{X}$

Latent space $\mathcal{F}$

f

f

g

Latent space $\mathscr{F}$

Original space $\mathscr{X}$
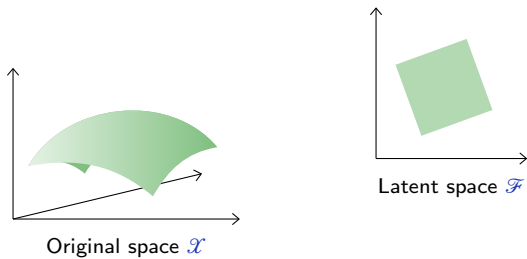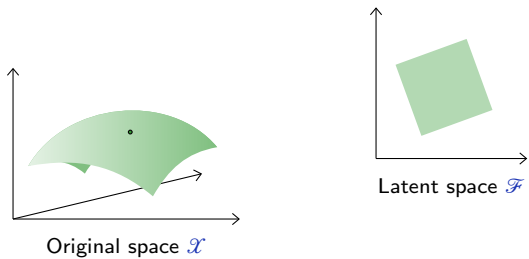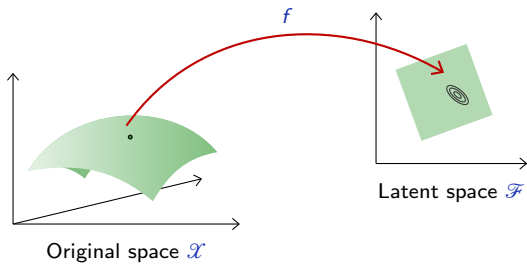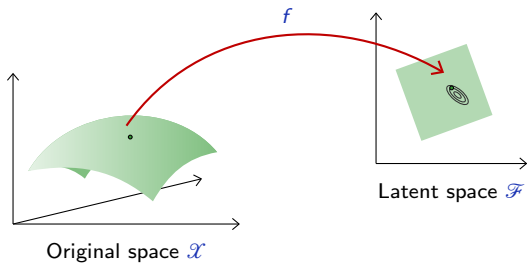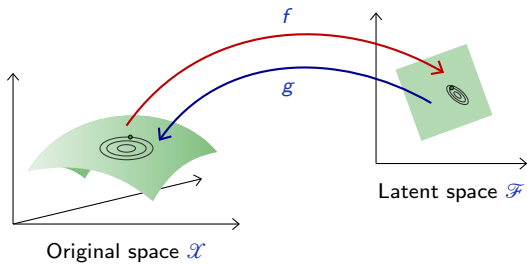
We have to minimize

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

We have to minimize

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

Since $q(Z \mid X)$ and $p(Z)$ are Gaussian, we have

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

We have to minimize

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

Since $q(Z \mid X)$ and $p(Z)$ are Gaussian, we have

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

And with

$$z_l^n \sim \mathscr{N}\left(\mu^f(x_n), \sigma^f(x_n)\right),\ n = 1,\ldots,N,\ l = 1,\ldots,L,$$

we have

$$-\hat{\mathbb{E}}\Big[\log p(X \mid Z)\Big] \simeq \frac{1}{2}\sum_{n=1}^{N}\sum_{l=1}^{L}\sum_c \frac{\left(x_{n,d} - \mu_c^g(z_l^n)\right)^2}{2\left(\sigma_c^g(z_l^n)\right)^2}$$

Kingma and Welling point out that using $L = 1$ is enough.

For MNIST, we keep our convolutional structure, but the encoder now maps to twice the number of dimensions, which corresponds to the μ^fs and σ^fs, and we use a fixed variance for the decoder.

We use Adam for training and the loss estimate for the standard autoencoder

```
output = model(input)
loss = mse_loss(output, input)
```

For MNIST, we keep our convolutional structure, but the encoder now maps to twice the number of dimensions, which corresponds to the μ^fs and σ^fs, and we use a fixed variance for the decoder.

We use Adam for training and the loss estimate for the standard autoencoder

```
output = model(input)
loss = mse_loss(output, input)
```

becomes

```
param = model.encode(input)
mu, logvar = param.split(param.size(1)//2, 1)
logvar = logvar + math.log(0.01)
std = logvar.mul(0.5).exp()

kl = - 0.5 * (1 + logvar - mu.pow(2) - logvar.exp())
kl = kl.mean()

u = Variable(mu.data.new(mu.size()).normal_())
z = u * std + mu
output = model.decode(z)

loss = mse_loss(output, input) + 0.5 * kl
```

For MNIST, we keep our convolutional structure, but the encoder now maps to twice the number of dimensions, which corresponds to the μ^fs and σ^fs, and we use a fixed variance for the decoder.

We use Adam for training and the loss estimate for the standard autoencoder

```
output = model(input)
loss = mse_loss(output, input)
```

becomes

```
param = model.encode(input)
mu, logvar = param.split(param.size(1)//2, 1)
logvar = logvar + math.log(0.01)
std = logvar.mul(0.5).exp()

kl = - 0.5 * (1 + logvar - mu.pow(2) - logvar.exp())
kl = kl.mean()

u = Variable(mu.data.new(mu.size()).normal_())
z = u * std + mu
output = model.decode(z)

loss = mse_loss(output, input) + 0.5 * kl
```

For MNIST, we keep our convolutional structure, but the encoder now maps to twice the number of dimensions, which corresponds to the μ^fs and σ^fs, and we use a fixed variance for the decoder.

We use Adam for training and the loss estimate for the standard autoencoder

```
output = model(input)
loss = mse_loss(output, input)
```

becomes

```
param = model.encode(input)
mu, logvar = param.split(param.size(1)//2, 1)
logvar = logvar + math.log(0.01)
std = logvar.mul(0.5).exp()

kl = - 0.5 * (1 + logvar - mu.pow(2) - logvar.exp())
kl = kl.mean()

u = Variable(mu.data.new(mu.size()).normal_())
z = u * std + mu
output = model.decode(z)

loss = mse_loss(output, input) + 0.5 * kl
```

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

Original

Autoencoder reconstruction ($d = 32$)

Variational Autoencoder reconstruction ($d = 32$)

Autoencoder sampling ($d = 32$)
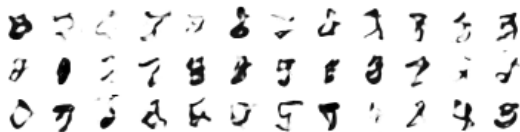
Variational Autoencoder sampling ($d = 32$)
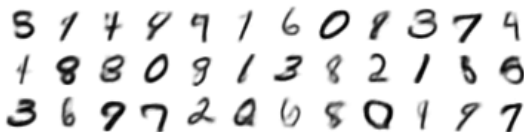
Non-Volume Preserving network

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

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) \, |J_f(x)| \, .$$
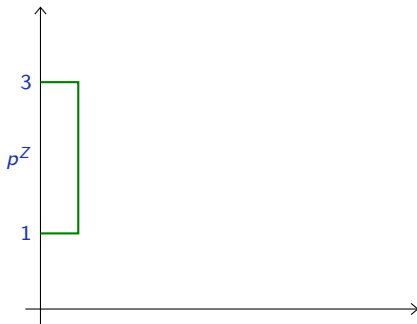
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x,\ p^{f^{-1}(Z)}(x) = p^Z(f(x))\, |J_f(x)|\,.$$
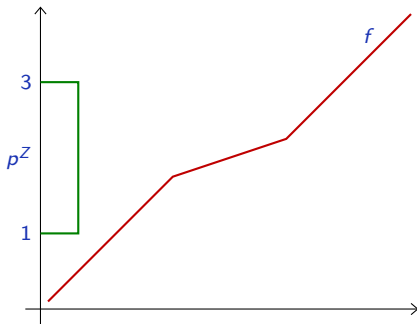
From this equality, if f is a parametric function such that we can compute [and differentiate]

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

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

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

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

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

And remember that if f is is a composition of functions

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

we have

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

so

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

If $f^{(k)}$ are standard layers we cannot compute $f^{-1}(z)$, and computing $|J_f(x)|$ is intractable.

If $f^{(k)}$ are standard layers we cannot compute $f^{-1}(z)$, 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

$$s : \mathbb{R}^d \to \mathbb{R}^d$$
$$t : \mathbb{R}^d \to \mathbb{R}^d,$$

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

$$s : \mathbb{R}^d \to \mathbb{R}^d$$
$$t : \mathbb{R}^d \to \mathbb{R}^d,$$

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

$$c : \mathbb{R}^d \to \mathbb{R}^d$$
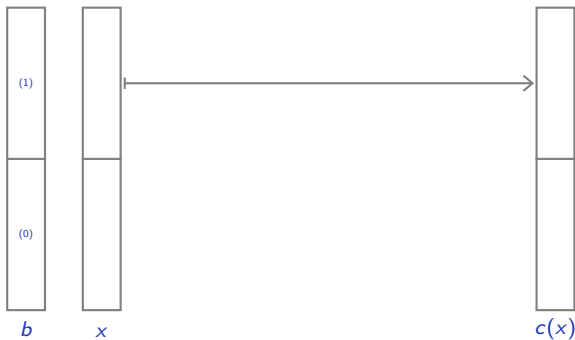$$x \mapsto b \odot x + (1 - b) \odot \left(x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

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,

b   x $c(x)$

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 $b \odot x$.
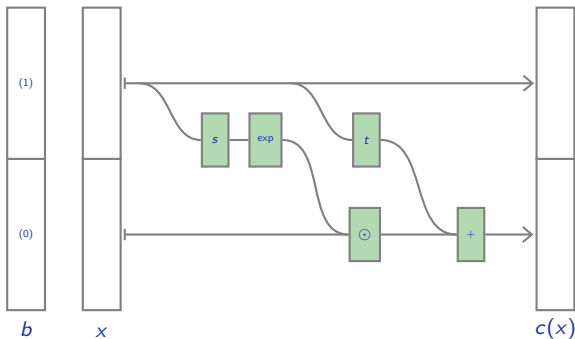
The consequence is that c is invertible, and if $y = c(x)$
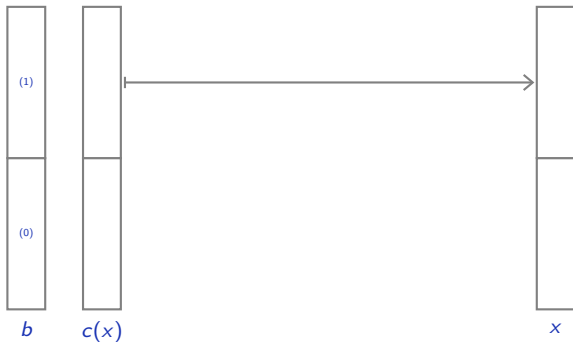
$$x = b \odot y + (1 - b) \odot \Big(y - t(b \odot y)\Big) \odot \exp(-s(b \odot y)).$$

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

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

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

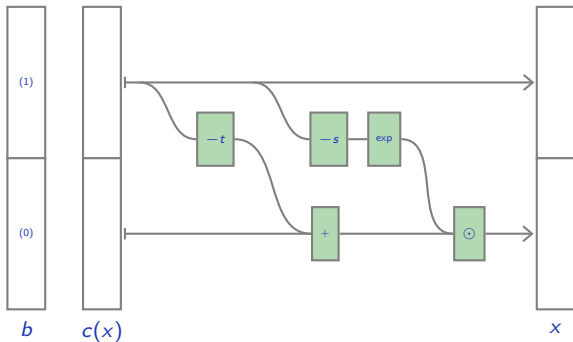The second property of this mapping is the simplicity of its Jacobian determinant. Since

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

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

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

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

$$b_i = 1 \quad \Rightarrow \quad c_i(x) = x_i$$
$$\Rightarrow \quad \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}}$$

and

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

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

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

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

$$b_i = 1 \quad \Rightarrow \quad c_i(x) = x_i$$
$$\Rightarrow \quad \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}}$$

and

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

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

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

```
dim = 6

x = Variable(Tensor(1, dim).normal_(), requires_grad = True)
b = Variable(Tensor(1, dim).zero_())
b.data.narrow(1, 0, dim//2).fill_(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 = Variable(Tensor(1, dim).normal_(), requires_grad = True)
b = Variable(Tensor(1, dim).zero_())
b.data.narrow(1, 0, dim//2).fill_(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

```
 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
-0.8182   0.5622  -0.4035   1.4700   0.0000   0.0000
 0.2610   0.1475   0.1689   0.0000   1.1358   0.0000
-0.2910   0.0287   0.0194   0.0000   0.0000   0.8508
[torch.FloatTensor of size 6x6]
```

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

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

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

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

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

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

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

with

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

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

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

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

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

with

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

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

A coupling layer can be implemented with

```
class NVPCouplingLayer(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 = Variable(b.clone().unsqueeze(0), requires_grad = False)

    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, y):
        s, t = self.map_s(self.b * y), self.map_t(self.b * y)
        return self.b * y + (1 - self.b) * (torch.exp(-s) * (y - 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(Module):
    def __init__(self, dim, hdim, depth):
        super(NVPNet, self).__init__()
        b = Tensor(dim)
        self.layers = nn.ModuleList()
        for d in range(depth):
            if d%2 == 0:
                # Tag half the dimensions
                i = torch.randperm(b.numel()).narrow(0, 0, b.numel() // 2)
                b.zero_()[i] = 1
            else:
                b = 1 - b
            map_s = nn.Sequential(nn.Linear(dim, hdim), nn.Tanh(), nn.Linear(hdim, dim))
            map_t = nn.Sequential(nn.Linear(dim, hdim), nn.Tanh(), nn.Linear(hdim, dim))
            self.layers.append(NVPCouplingLayer(map_s, map_t, b))

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

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

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

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

```
z = Variable(Tensor(nb_train_samples, dim).normal_())

x = model.invert(z).data
```
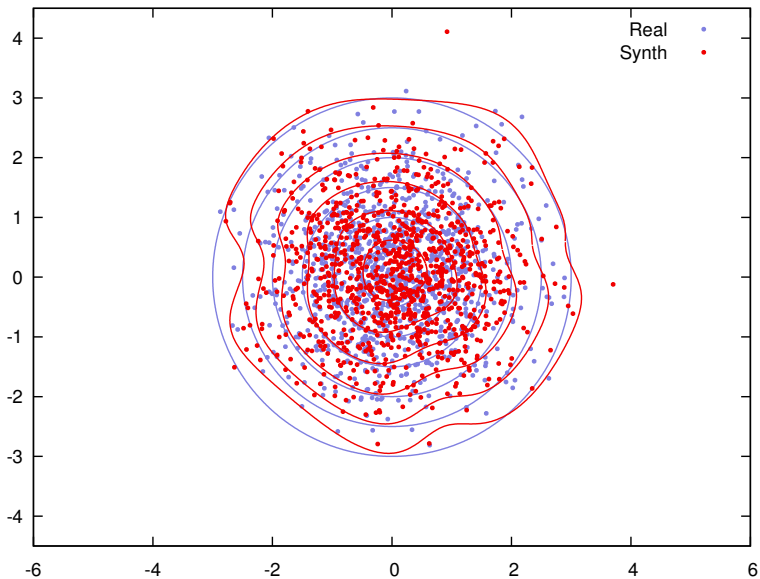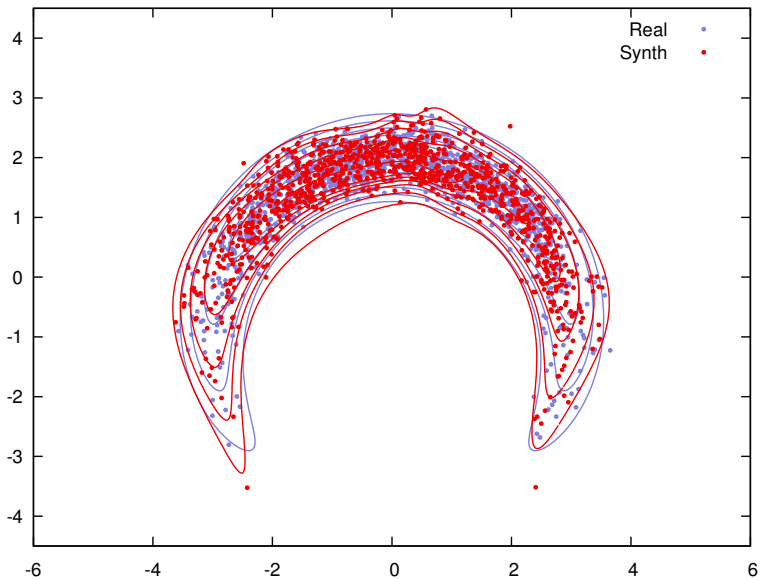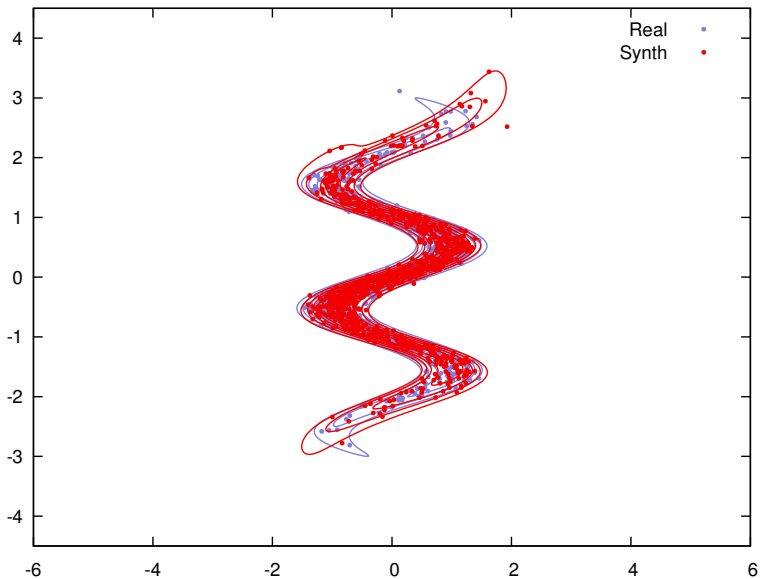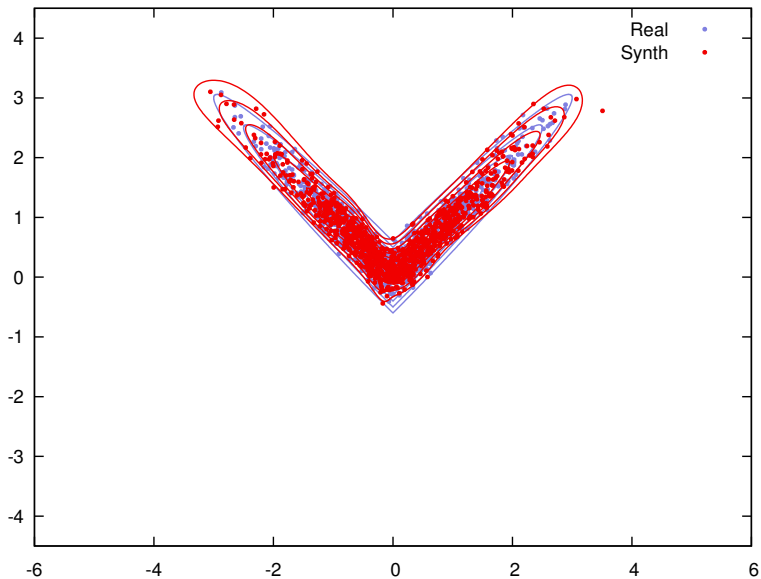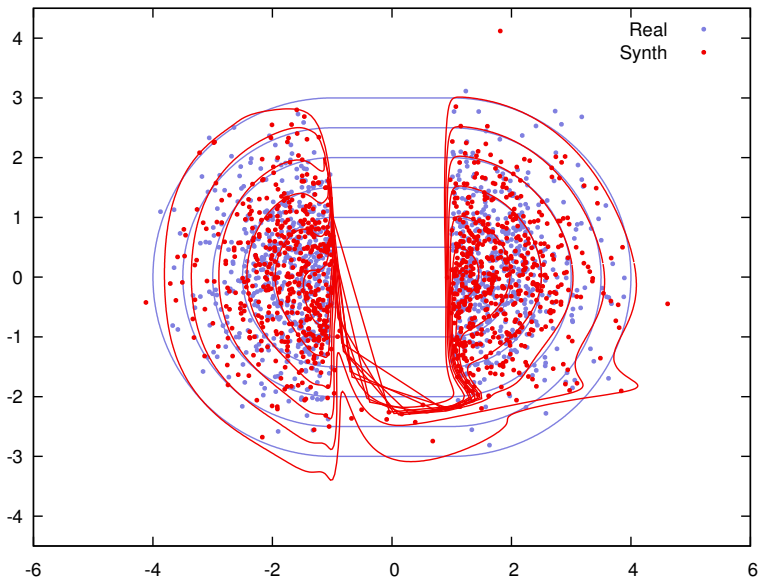
Dinh et al. (2016) apply this approach to convolutional layers by using bs consistent with the activation map structure, and reducing the map size while increasing the number of channels.
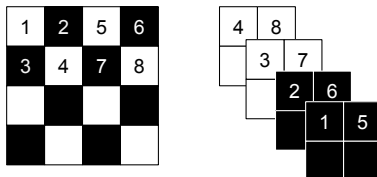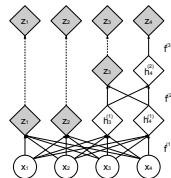

Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the $4 \times 4 \times 1$ tensor (on the left) into a $2 \times 2 \times 4$ tensor (on the right). Before the squeezing operation, 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.
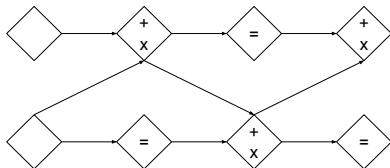
(a) In this alternating pattern, units which remain identical in one transformation are modified in the next.

(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.

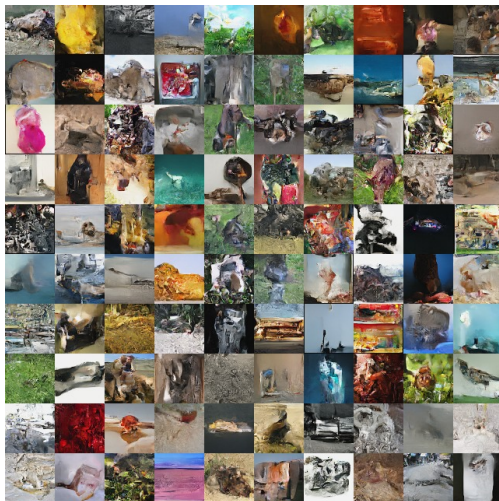The s and t mappings get more complex in the later layers.

Figure 7: Samples from a model trained on *Imagenet* (64×64).
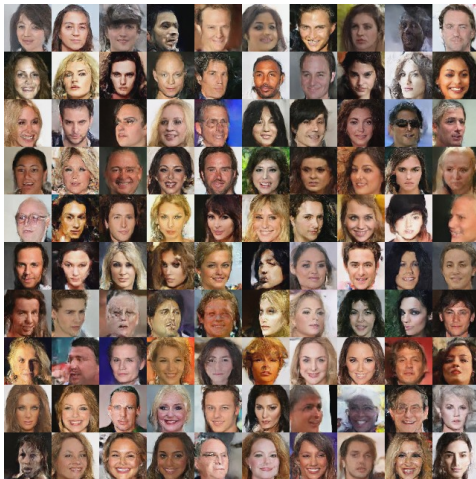
(Dinh et al., 2016)

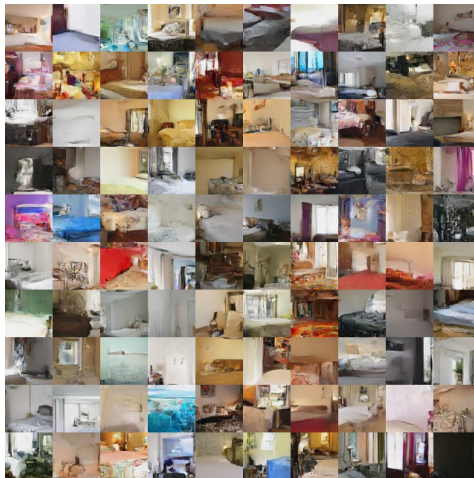Figure 8: Samples from a model trained on *CelebA*.

(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).
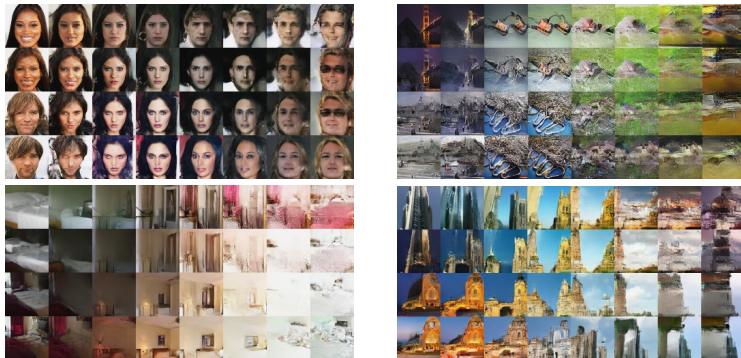
(Dinh et al., 2016)

Figure 6: Manifold generated from four examples in the dataset. Clockwise from top left: CelebA, Imagenet (64×64), 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. P. Kingma and M. Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

D. Rezende and S. Mohamed. Variational inference with normalizing flows. *CoRR*, abs/1505.05770, 2015.

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.