EE-311—Apprentissage et intelligence artificielle

9. Réseaux profonds et convolutifs

François Fleuret (ML : modifs 2022–25)

`https://moodle.epfl.ch/course/view.php?id=16090`

2 Mai 2025 (compiled 1er mai 2025)

idiap RESEARCH INSTITUTE

EPFL

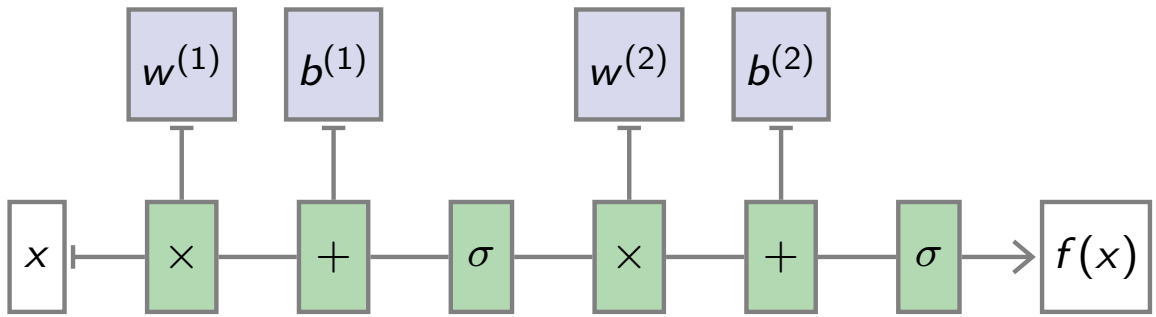
Réseaux profonds et convolutifs : Contenu

Extension des perceptrons multi-couches et apprentissage profond :

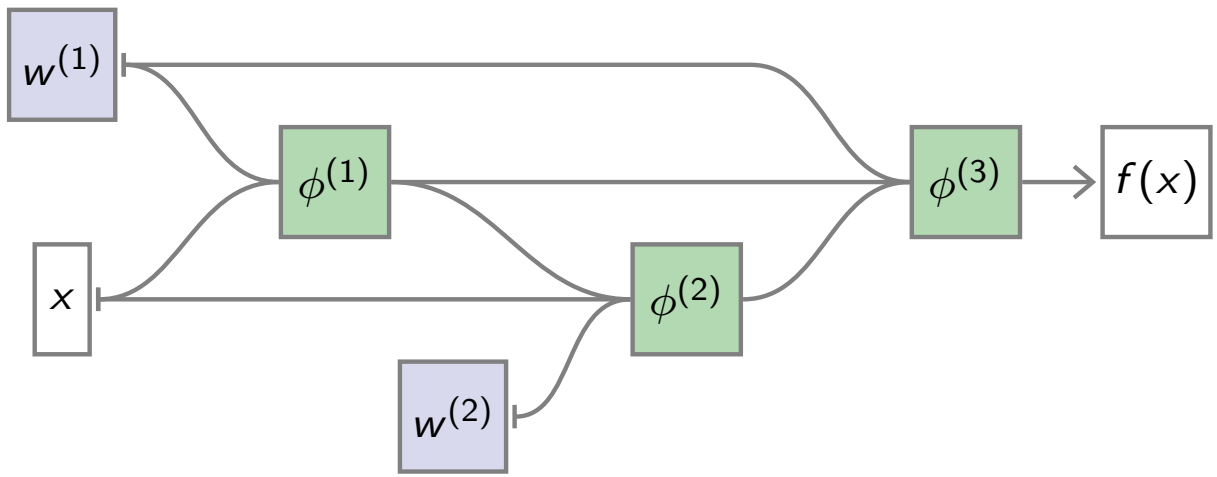
- Généralisation des architectures
- Autograd
- Descente de gradient stochastique
- Convolutions et *pooling*
- Réseaux neuronaux convolutionnels
- Exemples de réseaux
  - Auto-encodeurs, représentations latentes
  - Modèles génératifs antagonistes
- Analyse des modèles profonds
  - Images d'entrées qui maximisent l'activité d'unités du réseau
  - Cartographie des régions d'une images qui contribuent le plus à la prédiction

Graphe acyclique orienté (Directed Acyclic Graph (DAG)) : une généralisation du perceptron multi-couche

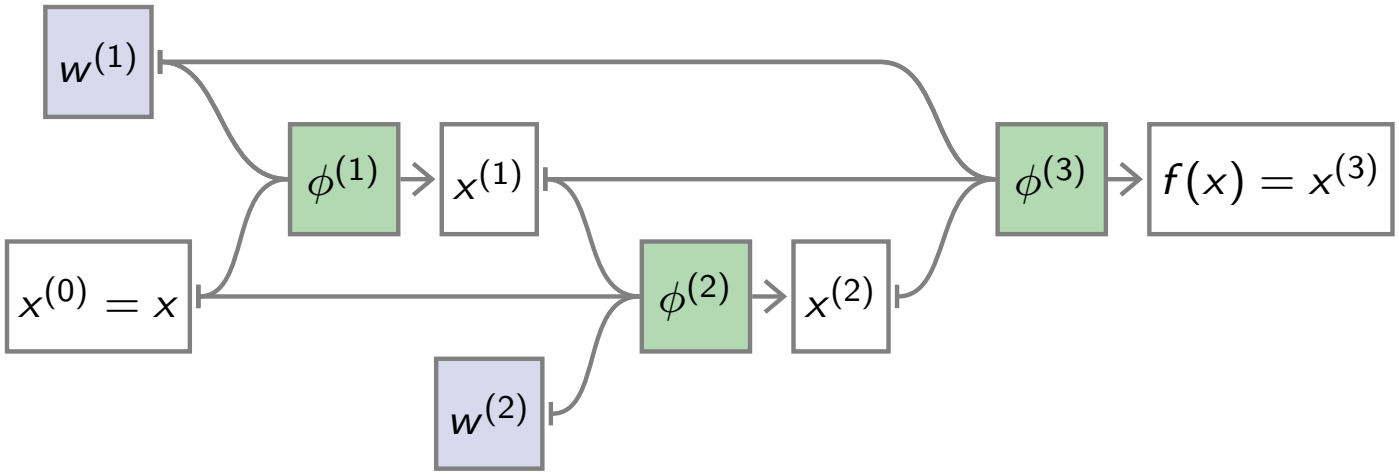
Un perceptron multi-couche



peut être généralisé à un "graphe acyclique orienté" d'opérateurs



Propagation vers l'avant pour un exemple de DAG



$$
\begin{aligned}
x^{(0)} &= x \\
x^{(1)} &= \phi^{(1)}\left(x^{(0)}; w^{(1)}\right) \\
x^{(2)} &= \phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) \\
f(x) = x^{(3)} &= \phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right)
\end{aligned}
$$

Convention de notation pour dérivations

Si $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)$, on note :

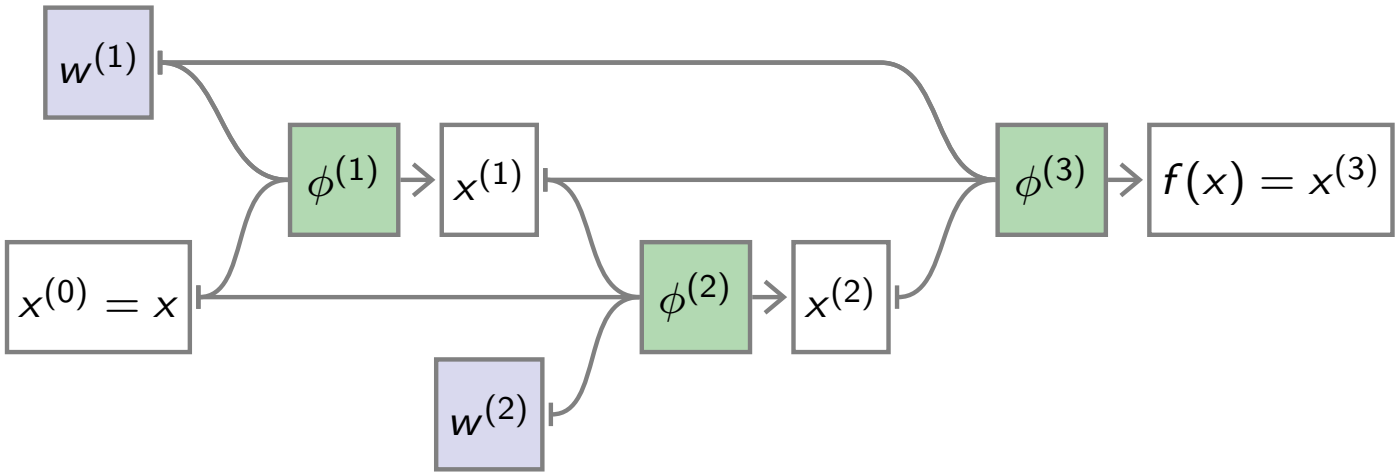
$$\left[\frac{\partial a}{\partial b}\right] = J_\phi = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

Si $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R, c_1, \ldots, c_S)$, on note :

$$\left[\frac{\partial a}{\partial c}\right] = J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

Rétro-propagation du gradient pour exemple de DAG (1/2)

D'abord on regarde les dérivées par rapport aux activations



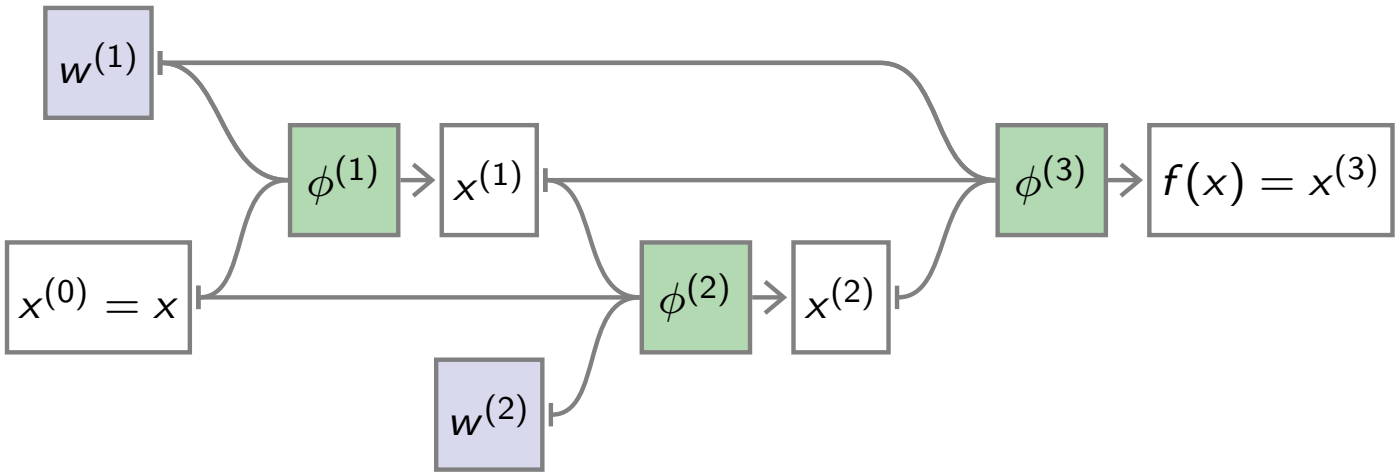
$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right] \left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}} \left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(1)}}\right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}}\right] \left[\frac{\partial \ell}{\partial x^{(2)}}\right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}}\right] \left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(2)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(2)}}\right] + J_{\phi^{(3)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(0)}}\right] = \left[\frac{\partial x^{(1)}}{\partial x^{(0)}}\right] \left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(2)}}{\partial x^{(0)}}\right] \left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(1)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(2)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

Rétro-propagation du gradient pour exemple GAO (2/2)

Puis les dérivées par rapport aux paramètres



$$\left[\frac{\partial \ell}{\partial w^{(1)}}\right] = \left[\frac{\partial x^{(1)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(3)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(1)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(3)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial w^{(2)}}\right] = \left[\frac{\partial x^{(2)}}{\partial w^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(2)}|w^{(2)}}\left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

Outils nécessaire pour faire une descente de gradient

Donc en pratique, si nous disposons d'une librairie "d'opérateurs tensoriels" pour calculer

$$\begin{aligned} & (x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w) \\ \forall c, \ & (x_1, \ldots, x_d, w) \mapsto J_{\phi|x_c}(x_1, \ldots, x_d; w) \\ & (x_1, \ldots, x_d, w) \mapsto J_{\phi|w}(x_1, \ldots, x_d; w), \end{aligned}$$

nous pouvons construire des modèles ayant la forme de graphes orientés acycliques quelconques, et nous pourrons calculer la réponse et faire une descente de gradient pour en optimiser les paramètres.

Outils permettant de combiner des opérateurs tensoriels, d'effectuer des dérivations automatiquement

Implémenter un modèle de type DAG arbitraire est complexe et sujet à de nombreuses erreurs.

Plusieurs librairies existent qui fournissent les fonctionnalités nécessaires et permettent de combiner des opérateurs tensoriels de manière arbitraire et de calculer des dérivations automatiquement.

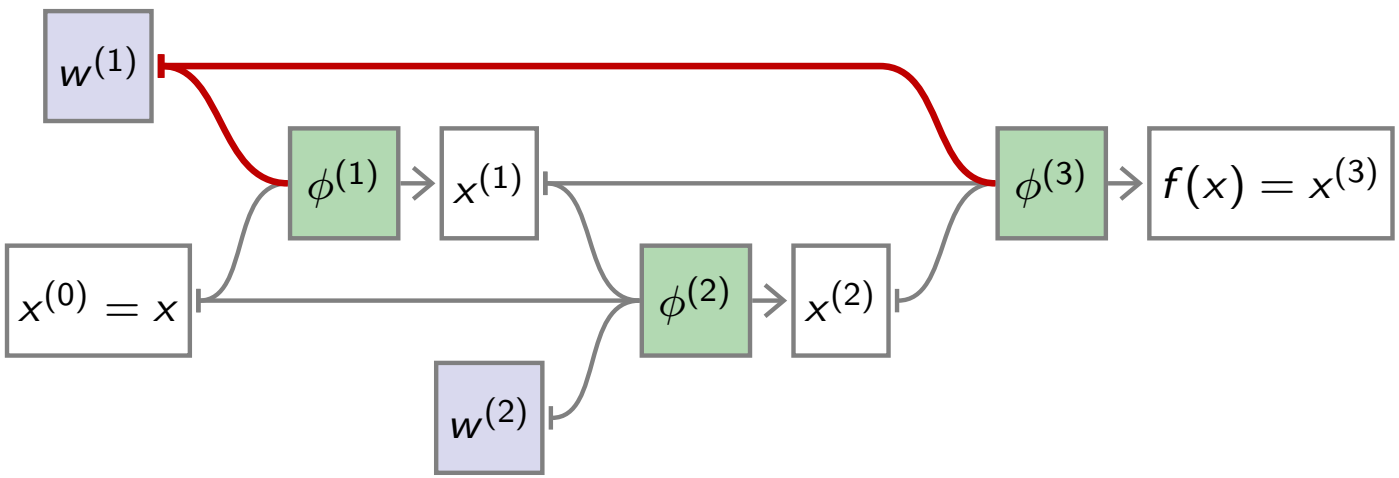
	Language(s)	**Licence**	**Principal développeur**
PyTorch	**Python**	BSD	Facebook
TensorFlow	Python, C++	Apache	Google
JAX	Python	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
Torch	Lua	BSD	Facebook
CNTK	Python, C++	MIT	Microsoft
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

Paramètres partagés

Partage de poids

Cette formulation généralisée sous forme d'un graphe d'opérateurs, autorise implicitement qu'un paramètre module plusieurs opérateurs.

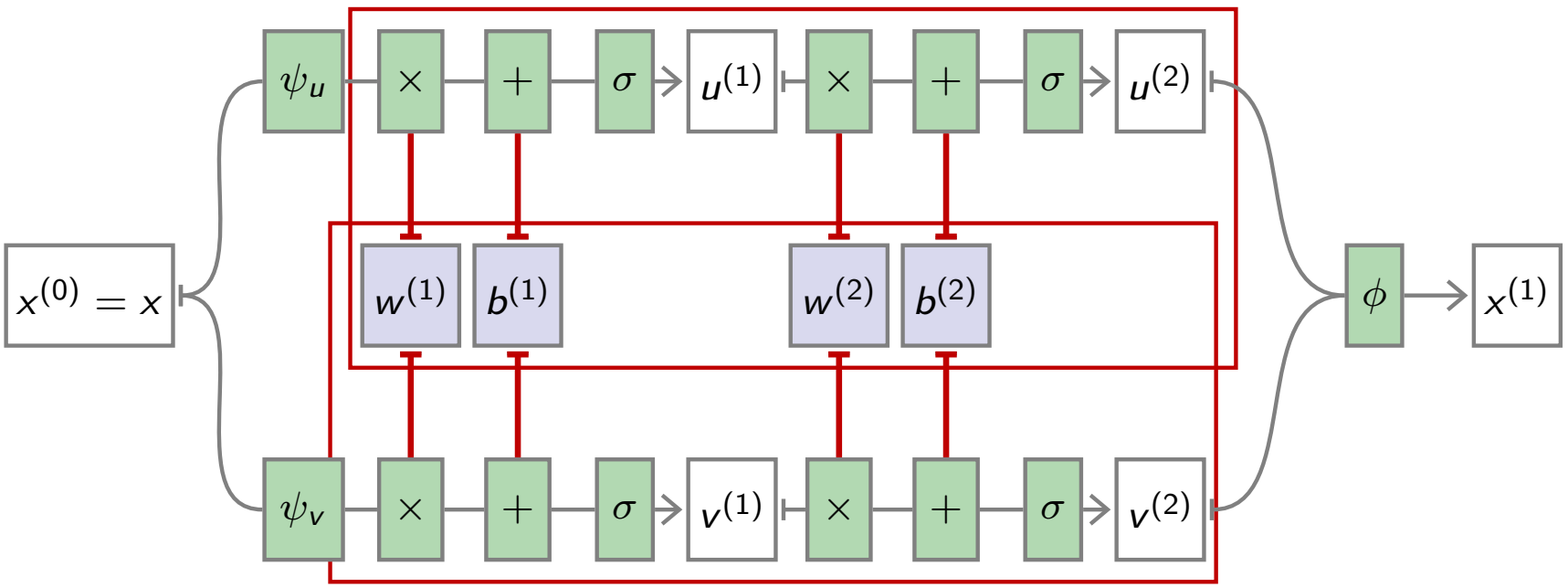
Dans notre exemple $w^{(1)}$ module $\phi^{(1)}$ et $\phi^{(3)}$.



On parle de **partage de poids**.

Partage de poids : exemple des réseaux siamois

Cela permet en particulier de concevoir des **réseaux siamois** où un réseau complet est répliqué plusieurs fois.



Autograd

Calcul automatique des dérivées

Conceptuellement, la propagation vers l'avant est une succession "classique" d'opérations tensorielles. Le graphe est nécessaire uniquement pour calculer les dérivées en déroulant le calcul à l'envers.

Une manière très élégante de calculer des dérivées consiste à construire automatiquement et dynamiquement, pendant le calcul, le graphe nécessaire à calculer des dérivées.

Ce mécanisme d'"autograd" a deux avantages majeurs :

- une syntaxe plus simple, des opérations classiques de manipulation de tenseurs en Python suffisent, et
- une plus grande flexibilité : comme le graphe est dynamique, il peut changer à chaque calcul.

Calcul automatique des dérivées avec PyTorch

C'est ce qu'offre par exemple PyTorch, qui est très proche de NumPy pour la manipulation de tenseurs.

Il suffit d'indiquer (avec `requires_grad_`) que l'on va avoir besoin de calculer des dérivées par rapport aux composantes d'un tenseur pour que PyTorch construise le graphe nécessaire.

```
>>> t = torch.tensor([1., 2., 4.])
>>> t.requires_grad_()
tensor([1., 2., 4.], requires_grad=True)
>>> s = sum(t**2)
>>> torch.autograd.grad(s, t)
(tensor([2., 4., 8.]),)
```

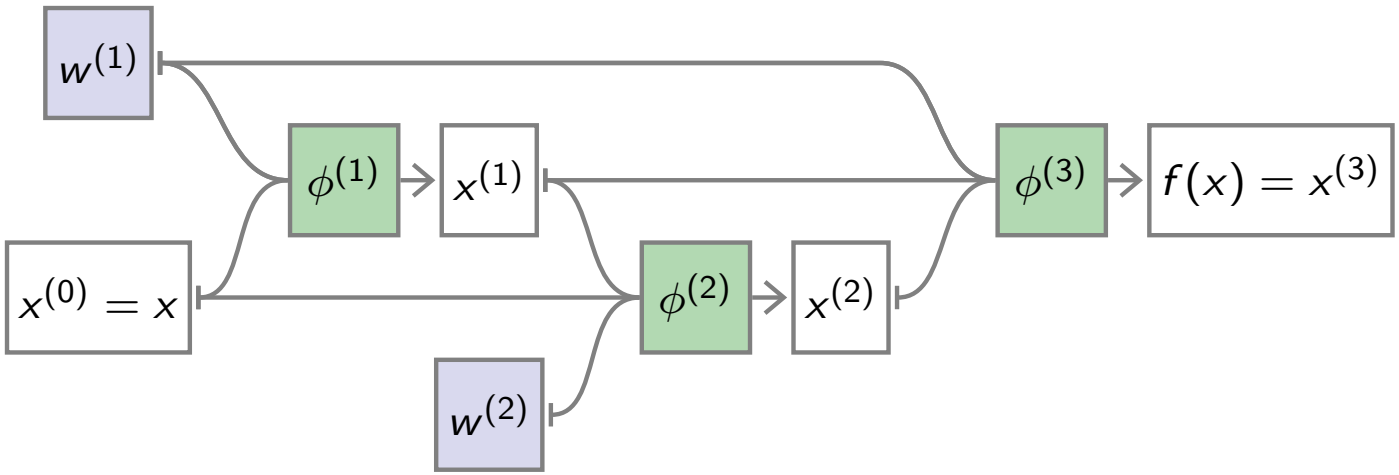
Accumulation des des gradients

La fonction `Tensor.backward()` accumule les gradients dans les champs `grad` des tenseurs et est souvent plus pratique pour traiter des gros modèles.

```
>>> x = torch.tensor([ 0.0, 0.1, 0.2 ]).requires_grad_()
>>> u = sum(torch.log(1/(x+1)))
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([-1.0000, -0.9091, -0.8333])
```

autograd appliqué à l'exemple de DAG

Nous pouvons donc exécuter l'évaluation et la rétro-propagation avec



$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$

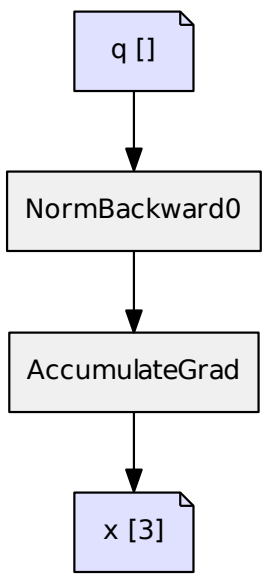
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

```
w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.empty(5).normal_()

x0 = x
x1 = w1 @ x0
x2 = x0 + w2 @ x1
x3 = w1 @ (x1 + x2)

q = x3.norm()

q.backward()
```

Visualisation du graphe construit lors d'un calcul tensoriel

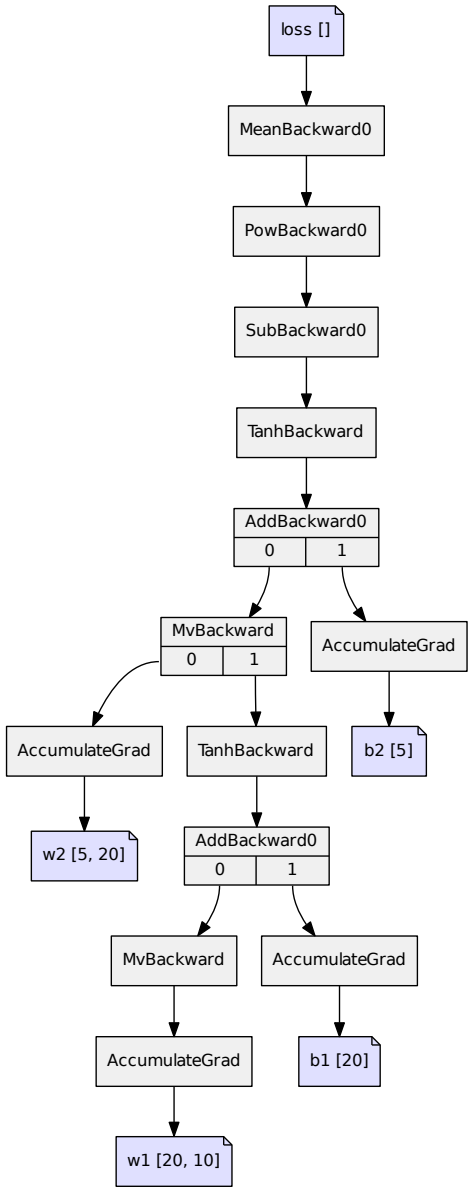
On peut visualiser le graphe qui est construit pendant un calcul tensoriel :

```
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```



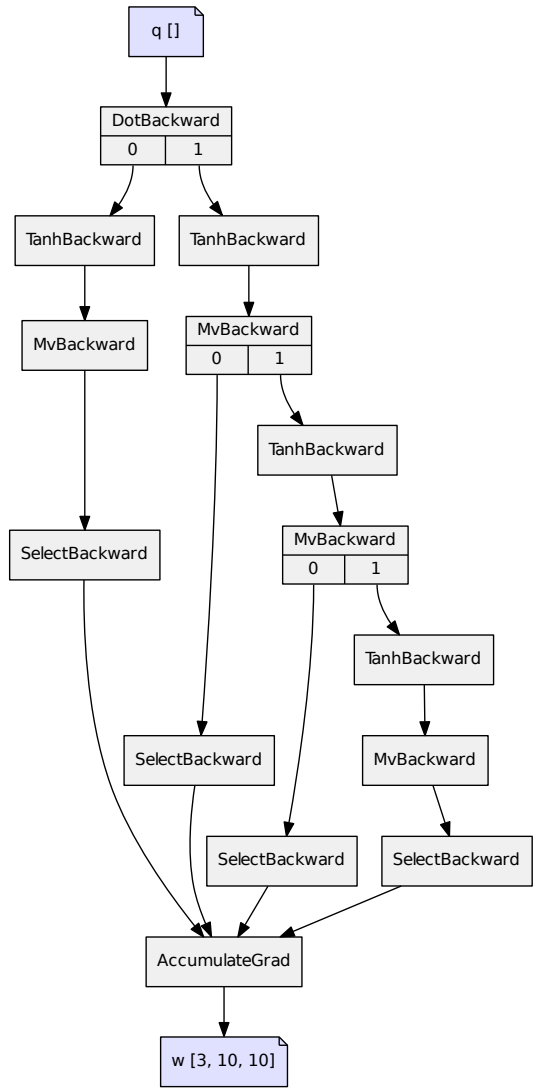
et un autre exemple

```
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()

x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)

target = torch.rand(5)

loss = (y - target).pow(2).mean()
```



exemple, suite

```
w = torch.rand(3, 10, 10).requires_grad_()

def blah(k, x):
    for i in range(k):
        x = torch.tanh(w[i] @ x)
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```



Descente de gradient stochastique

Rappel : algorithme de descente de gradient classique

Nous avons vu que pour minimiser un coût

$$\mathscr{L}(w) = \sum_{n=1}^{N} \underbrace{\ell(f(x_n; w), y_n)}_{\ell_n(w)}$$

l'algorithme classique itératif de descente de gradient a la forme

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t).$$

Limitation de l'implémentation directe et mini-batches

Une implémentation directe serait

```
for k in range(nb_epochs):
    output = model.forward(x)
    model.compute_grad(dloss(y, output))
    model.gradient_descent_step(eta)
```

L'occupation mémoire de cet algorithme est proportionnelle au nombre d'exemples. Cela peut être évité en traitant des "mini batches" :

```
for k in range(nb_epochs):
    model.zero_grad()
    for b in range(0, x.shape[0], nb):
        output = model.forward(x[b:b+nb])
        model.accumulate_grad(dloss(y[b:b+nb], output))
    model.gradient_descent_step(eta)
```

Considérations à propos du gradient

Bien que cela soit formellement raisonnable de calculer un gradient exactement, en pratique :

- Cela prend énormément de temps.
- C'est une estimation empirique et toute somme partielle serait un estimateur sans biais de la même quantité (avec une variance plus grande).
- Il est calculé itérativement

$$\nabla \mathscr{L}(w_t) = \sum_{n=1}^{N} \nabla \ell_n(w_t),$$

et quand ℓ_n est calculé, nous avons déjà $\ell_1, \ldots, \ell_{n-1}$ à disposition et pourrions obtenir un estimé de w^* plus à jour que w_t.

Argument intuitif pour approximer le gradient

Prenons un cas idéal où l'ensemble d'apprentissage est en réalité composé du même ensemble de M exemples repliqué K fois.
Nous avons alors :

$$\begin{aligned}\mathscr{L}(w) &= \sum_{n=1}^{N} \ell(f(x_n; w), y_n) \\ &= \sum_{k=1}^{K} \sum_{m=1}^{M} \ell(f(x_m; w), y_m) \\ &= K \sum_{m=1}^{M} \ell(f(x_m; w), y_m).\end{aligned}$$

Donc, au lieu de faire la somme complète, nous pouvons faire la somme sur seulement M exemples, et multiplier le résultat par K.

Bien que cela soit un cas idéal, la redondance dans tout ensemble d'exemples est telle que des comportements de ce type se rencontrent en pratique.

Descente de gradient stochastique

La **descente de gradient stochastique** consiste à mettre à jour les paramètres en utilisant le gradient du coût calculé sur des exemples individuels

$$w_{t+1} = w_t - \eta \nabla \ell_{n(t)}(w_t).$$

Cette approche est peu efficace computationellement car elle utilise mal la mémoire cache. Il vaut donc mieux traiter les exemples par groupes.

Descente de gradient stochastique par mini-batches

La **descente de gradient stochastique par mini-batches** est la procédure standard d'optimisation des paramètres pour l'apprentissage profond. Elle consiste à parcourir les exemples d'apprentissage par groupes, et à mettre à jour les paramètres du modèle à chaque fois :

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t).$$

L'ordre $n(t, b)$ dans lequel les exemples sont visités peut être séquentiel ou aléatoire, avec ou sans remplacement.

Le comportement aléatoire de cette procédure permet de s'échapper de minima locaux.

De procédure exacte à implémentation stochastique

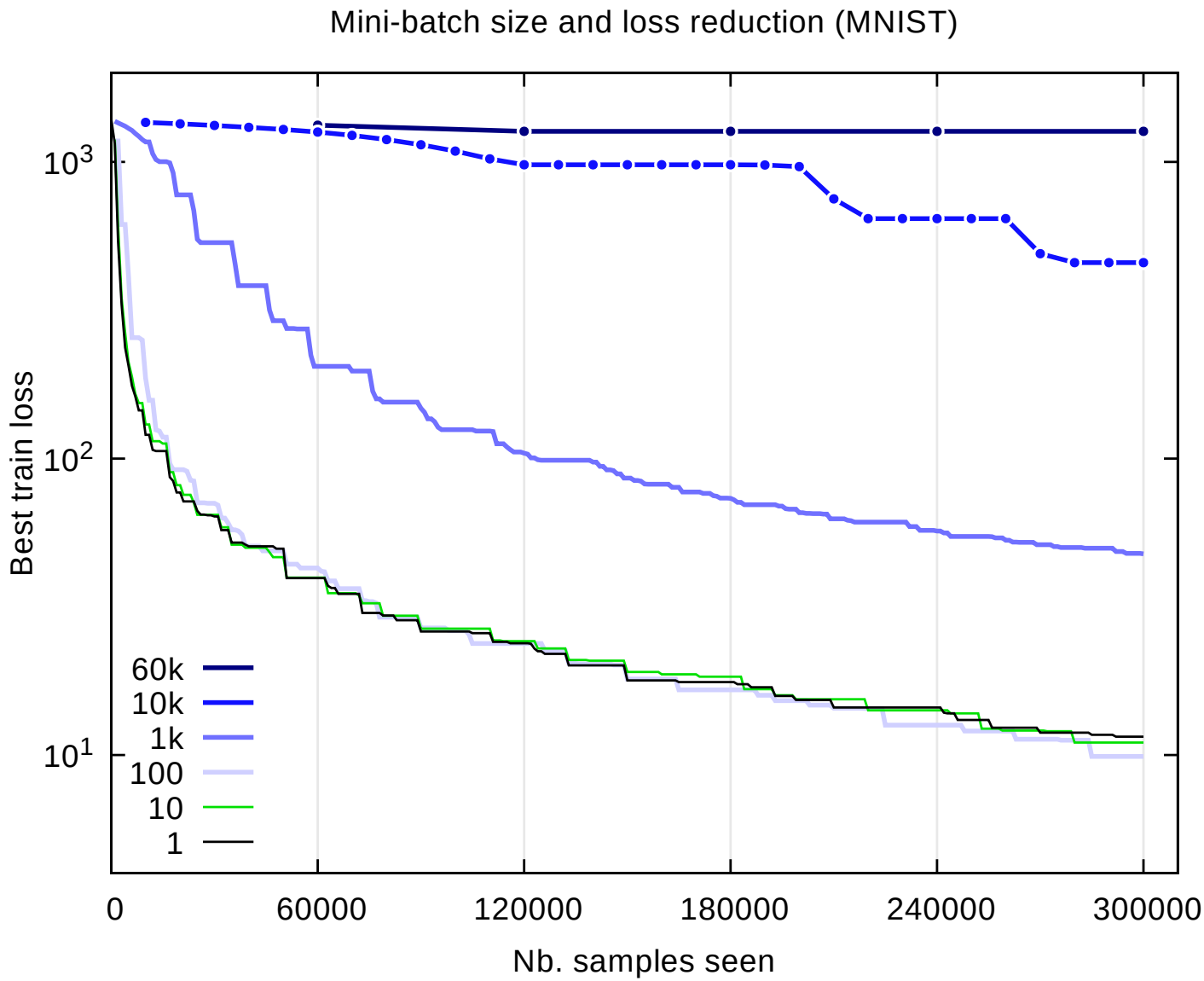
Notre procédure exacte par mini-batches

```
for k in range(nb_epochs):
    model.zero_grad()
    for b in range(0, x.shape[0], nb):
        output = model.forward(x[b:b+nb])
        model.accumulate_grad(dloss(y[b:b+nb], output))
    model.gradient_descent_step(eta)
```

peut être modifiée en une implémentation de la descente de gradient stochastique par mini-batches

```
for k in range(nb_epochs):
    for b in range(0, x.shape[0], nb):
        output = model.forward(x[b:b+nb])
        model.compute_grad(dloss(y[b:b+nb], output))
        model.gradient_descent_step(eta)
```

Evaluation du loss en fonction du nombre d'exemples vus pour différentes tailles de mini-batch



Convolutions

Limitation des réseaux entièrement connectés

S'ils étaient traités comme des signaux sans structure, les images ou les échantillons sonores demanderaient des modèles de tailles excessive.

Par exemple, une couche linéaire qui prendrait une image 256×256 couleurs en entrée et produirait un signal de même taille en sortie aurait

$$(256 \times 256 \times 3)^2 \simeq 3.87 \times 10^{10}$$

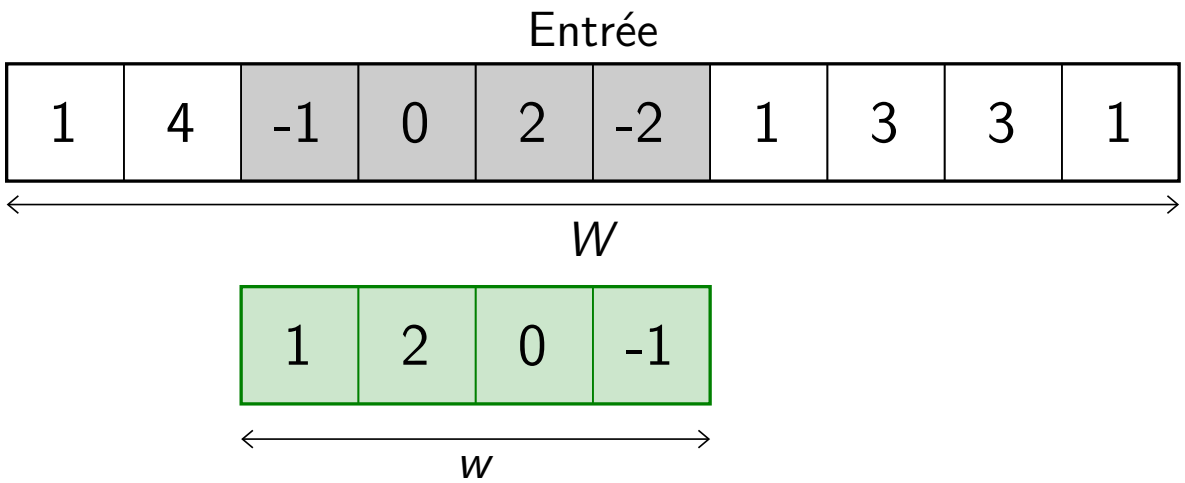
coefficients, avec l'occupation mémoire ($\simeq$150Gb !) et l'excès de capacité correspondants

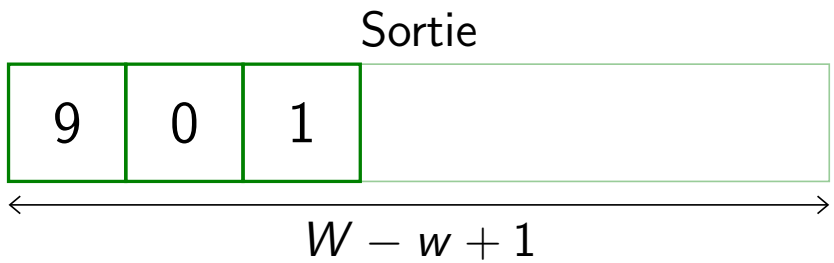
La stationarité des signaux motive l'utilisation de convolutions

Un modèle entièrement connecté serait incohérent avec l'intuition que les signaux tel que les images ou sons ont une certaines "stationarité" : une représentation qui est adéquate à un endroit l'est ailleurs.

Une couche convolutionnelle repose sur cette idée et applique le même opérateur linéaire "partout" dans le signal d'entrée pour calculer sa sortie.

“Convolution” 1D

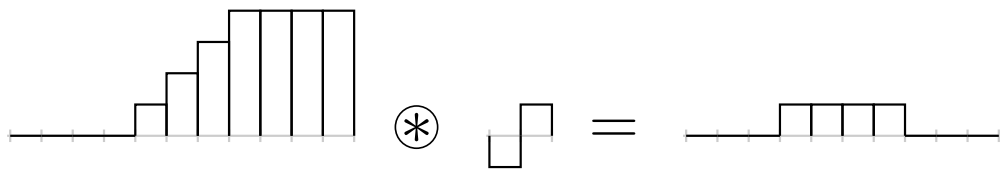




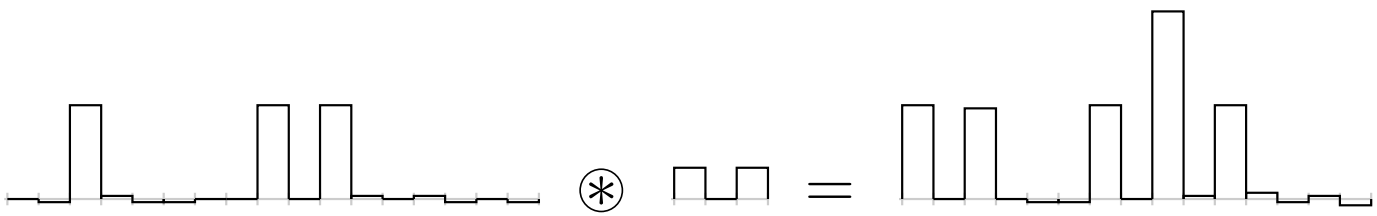
Interprétation de quelques convolutions

Une convolution peut calculer en particulier un opérateur différentiel discret, *e.g.*

$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



ou “détecteur de motif”, *e.g.*



Note : au sens “signaux et systèmes,” on devrait parler ici d’opérations de corrélation. En effet, on note que le noyau n’est pas retourné lorsqu’il est glissé.

Convolutions en dimensions > 1

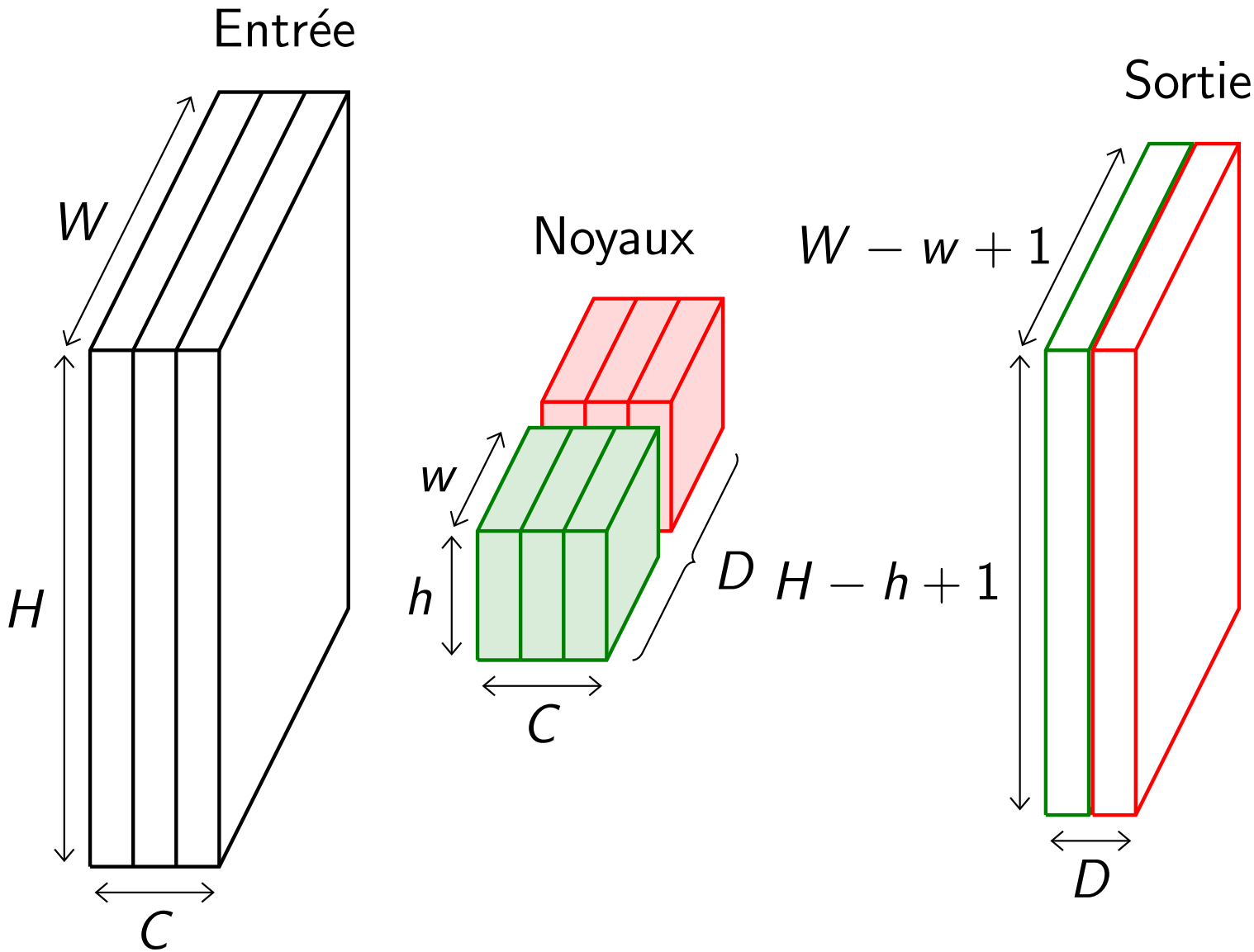
Cette opération se généralise naturellement à des signaux de plus grande dimension.

La forme la plus fréquente dans les réseaux convolutifs opère sur un tenseur à trois dimensions qui représente un signal 2d multi-canal. Le noyau de convolution se déplace sur les lignes et les colonnes mais pas sur les canaux.

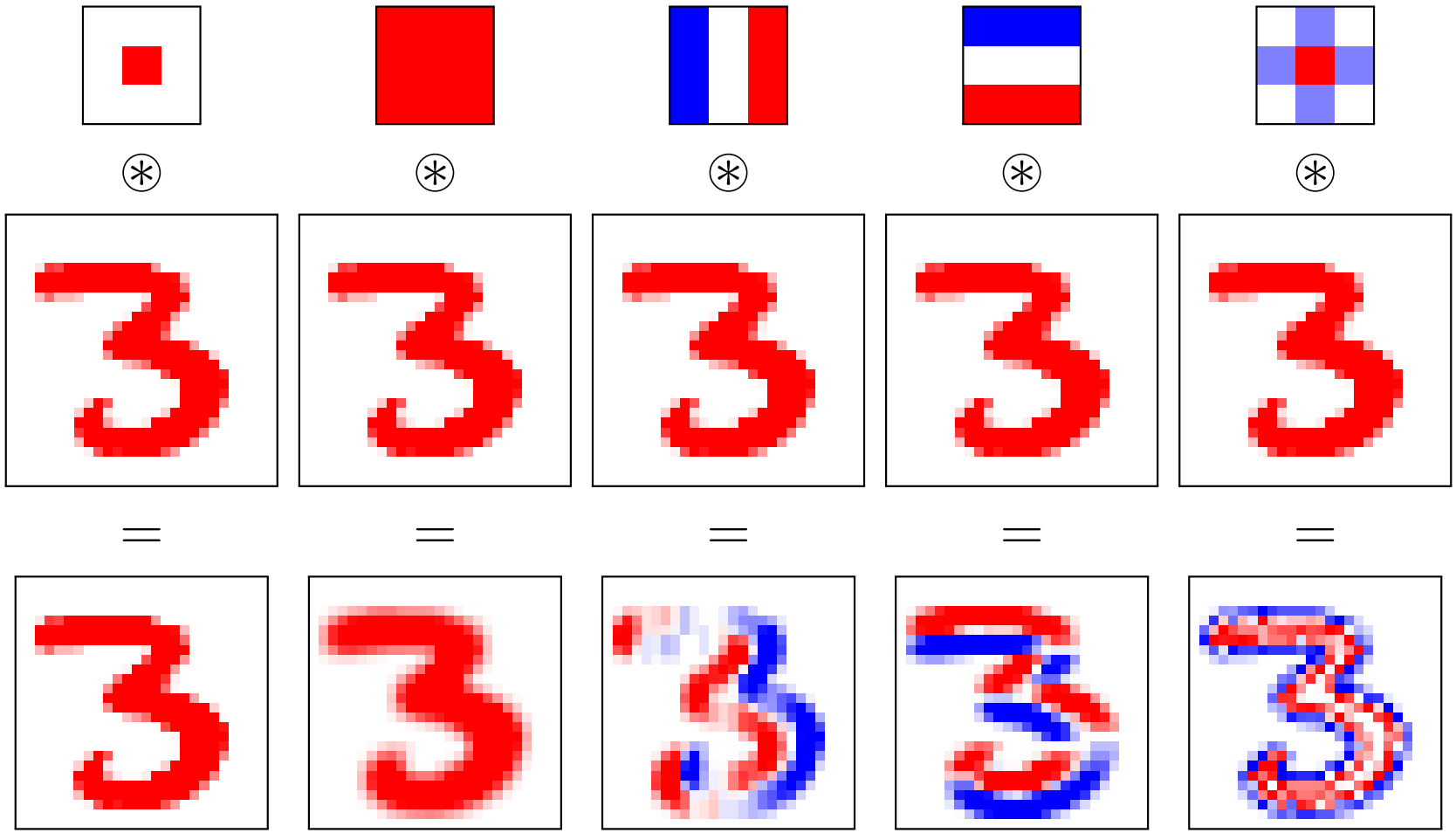
Si le tenseur en entrée est de taille $C \times H \times W$, et le kernel est $C \times h \times w$, le tenseur de sortie sera $(H - h + 1) \times (W - w + 1)$.

Une couche convolutionnelle classique combine D convolutions de ce type et génère en sortie un tenseur de taille $D \times (H - h + 1) \times (W - w + 1)$.

Convolutions en 2D (e.g. images en couleur avec 3 canaux)



Exemples : Floutage (léger, fort), mise en évidence de bords (verticaux, horizontaux, toutes orientations)



Pooling

But du pooling

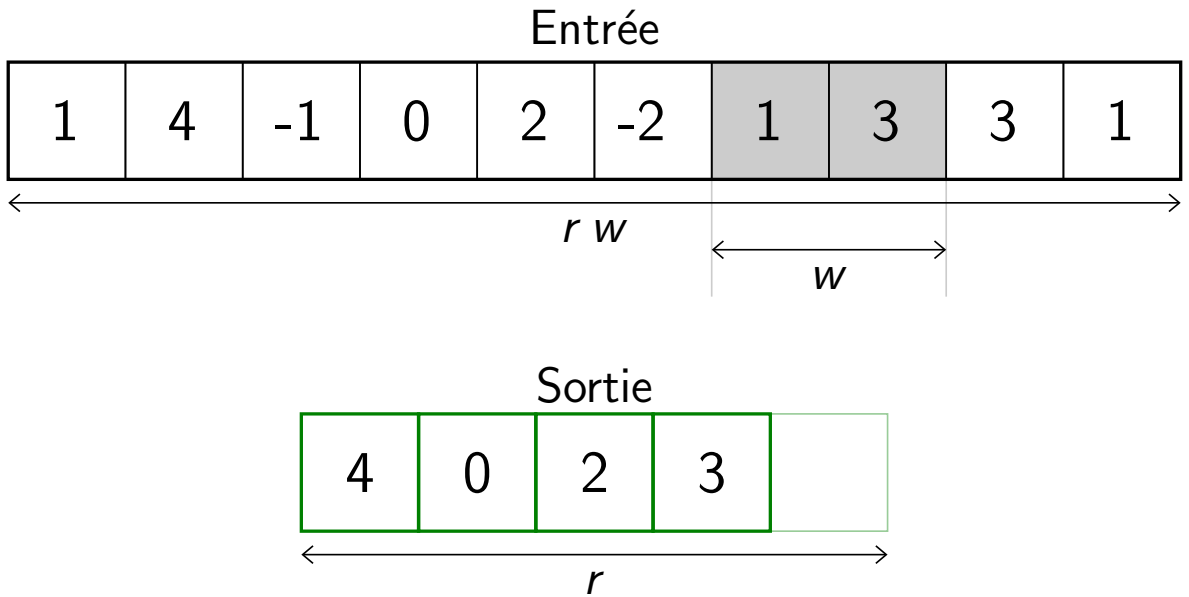
La manière traditionnelle pour obtenir un signal de petite dimension (*e.g.* quelques valeurs) en partant d'un signal de grande dimension (*e.g.* une image) consiste à utiliser des opérations de **pooling**.

De telles opérations visent à regrouper plusieurs valeurs en une seule qui est plus "informative".

Max- et average pooling

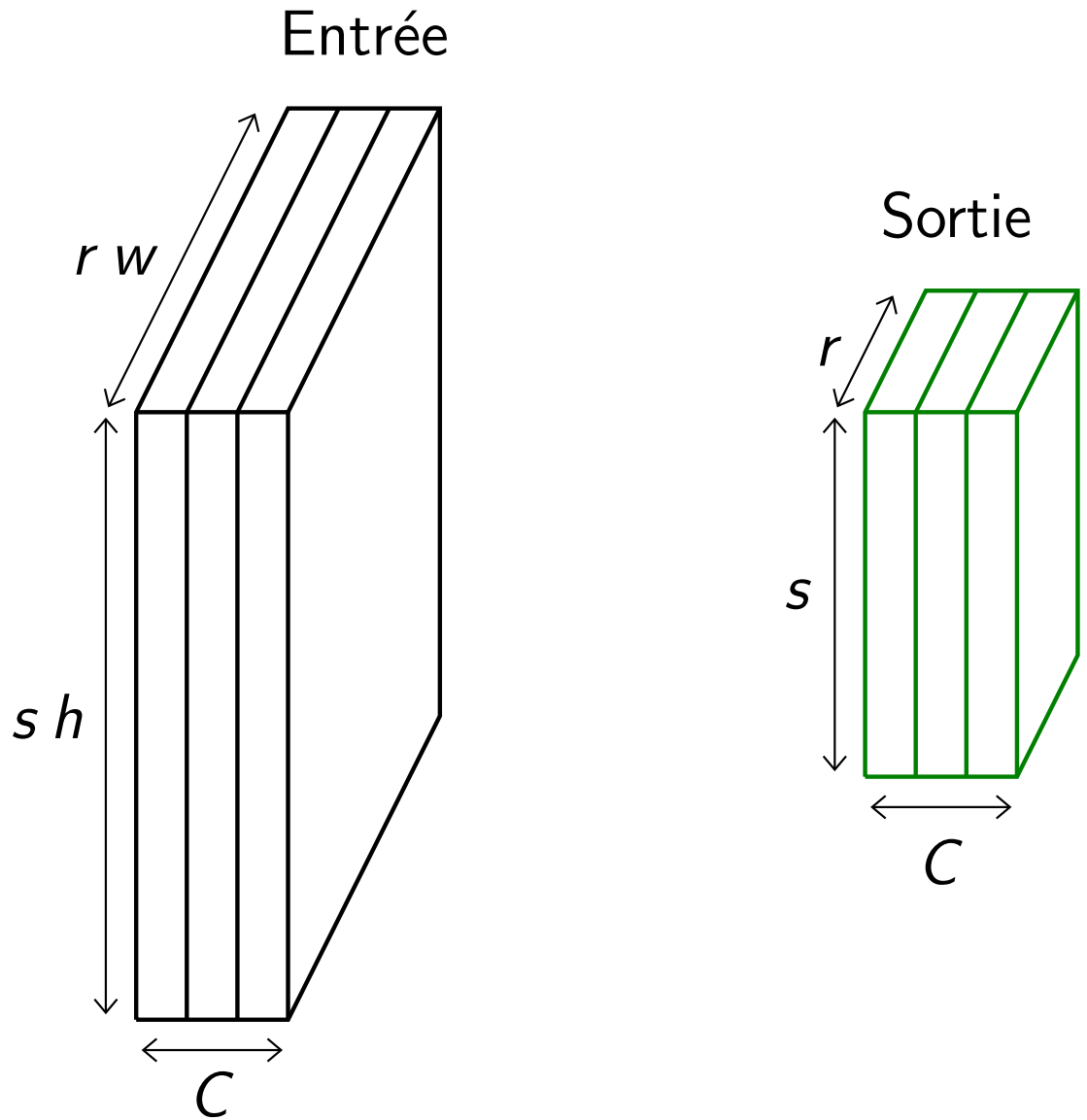
Le type de pooling le plus classique est le **max-pooling**, qui calcule la valeur maximale dans des blocs disjoints.

Par exemple, en 1d avec un noyau de taille 2



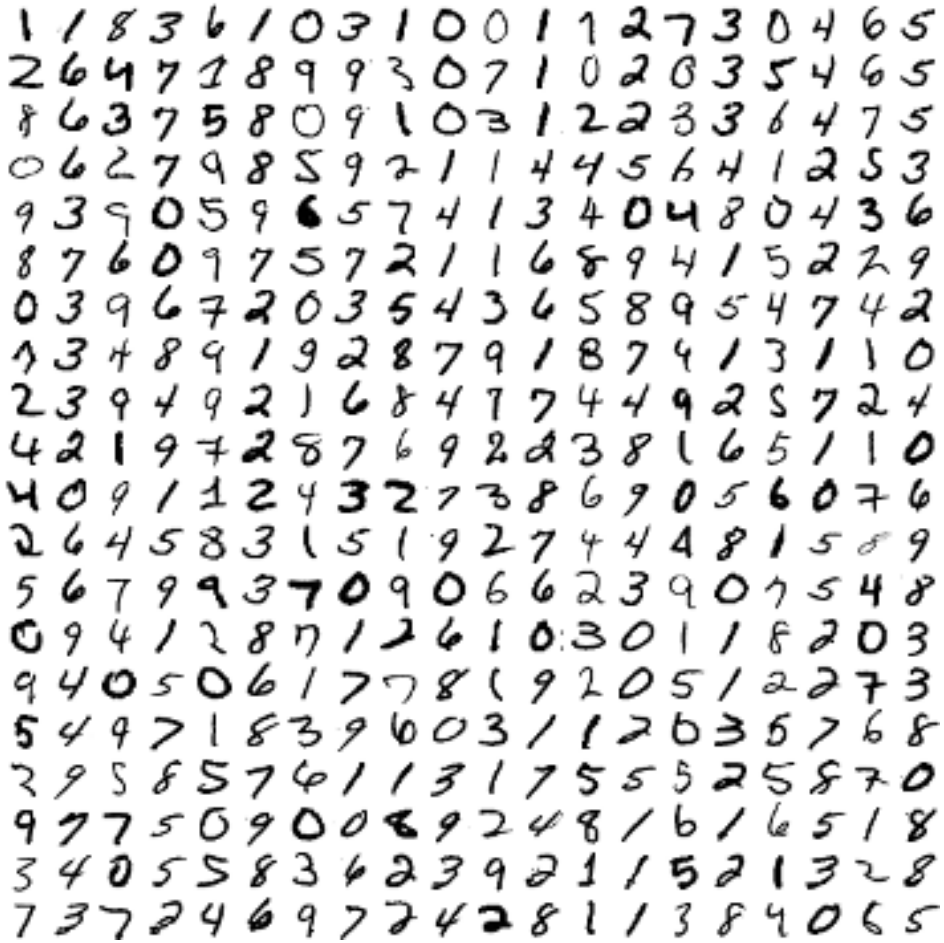
L'**average pooling** calcule lui une valeur moyenne par bloc. Ce dernier est donc un opérateur linéaire.

Opération de pooling en 2D



Exemple complet

MNIST



(leCun et al., 1998)

Implémentation PyTorch

```
model = nn.Sequential(
    nn.Conv2d(1, 32, 5),
    nn.ReLU(),
    nn.MaxPool2d(3),
    nn.Conv2d(32, 64, 5),
    nn.ReLU(),
    nn.MaxPool2d(2),
    Shape1D(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr = 1e-2)

for e in range(nb_epochs):
    for input, target in data_loader_iterator(train_loader):
        output = model(input)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Training time $<$10s (GPU), error $\simeq$1%

Exemple de performance

```
import PIL, torch, torchvision

to_tensor = torchvision.transforms.ToTensor()
img = to_tensor(PIL.Image.open('example_images/blacklab.jpg'))
img = img.unsqueeze(0)
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()
output = alexnet(img)

scores, indexes = output.view(-1).sort(descending = True)

class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())

for k in range(12):
    print(f'#{k+1} {scores[k].item():.02f} {class_names[indexes[k].item()]}')
```

Exemple de performance

```
#1 (12.26) Weimaraner
#2 (10.95) Chesapeake Bay retriever
#3 (10.87) Labrador retriever
#4 (10.10) Staffordshire bullterrier, Staffordshire bull terrier
#5 (9.55) flat-coated retriever
#6 (9.40) Italian greyhound
#7 (9.31) American Staffordshire terrier, Staffordshire terrier
#8 (9.12) Great Dane
#9 (8.94) German short-haired pointer
#10 (8.53) Doberman, Doberman pinscher
#11 (8.35) Rottweiler
#12 (8.25) kelpie
```

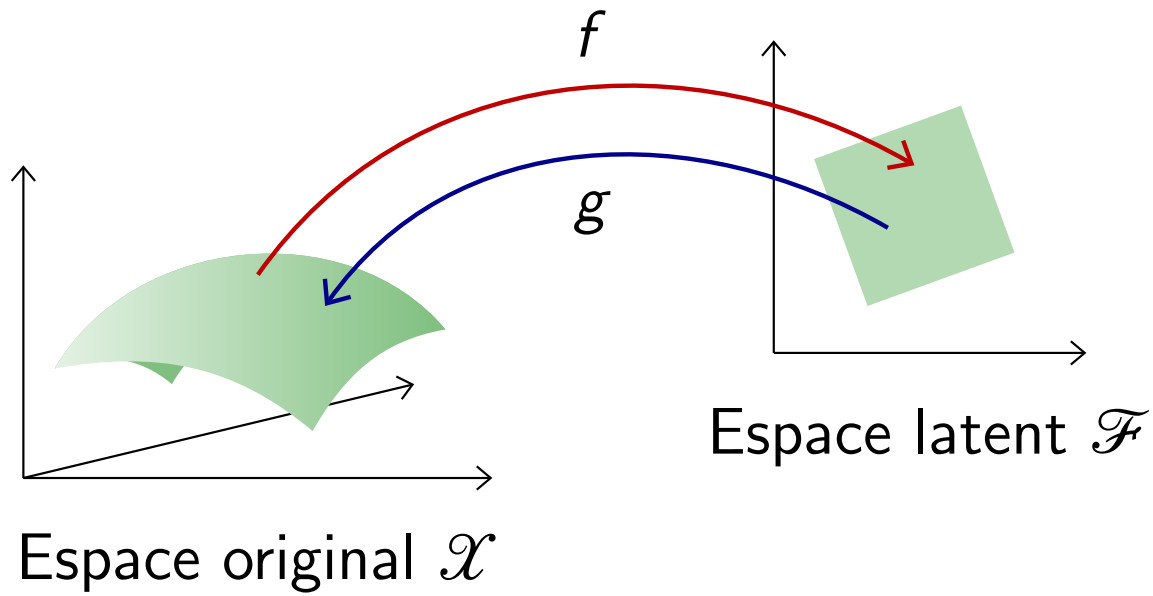
Weimaraner

Chesapeake Bay retriever

Auto-encodeurs

Auto-encodeurs : définition

Un auto-encodeur a une entrée et une sortie de même dimension, et se comporte comme l'identité sur les données. Il est souvent composé d'un **encodeur** qui va de l'espace de départ dans un **espace latent** et d'un **décodeur** qui revient dans l'espace de départ.



Si l'espace latent est de plus petite dimension, un autoencodeur doit réduire la redondance dans le signal.

Fonctions de coût pour les auto-encodeurs

Une fonction de coût classique pour entraîner un auto-encodeur est l'erreur quadratique. Avec q la distribution des données sur $\mathcal{X}$ on voudrait donc

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

Étant donnés deux modèles $f(\cdot\,; w_f)$ et $g(\cdot\,; w_g)$, les entraîner consiste à minimiser une estimation empirique de cette fonction de coût

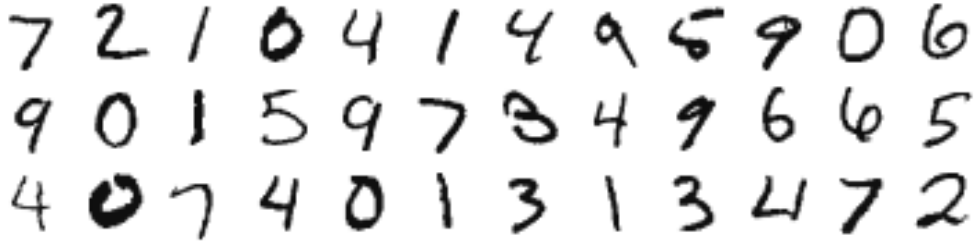
$$\hat{w}_f, \hat{w}_g = \operatorname*{argmin}_{w_f, w_g} \frac{1}{N}\sum_{n=1}^{N} \|x_n - g(f(x_n; w_f); w_g)\|^2 .$$

Auto-encodeurs : implémentation

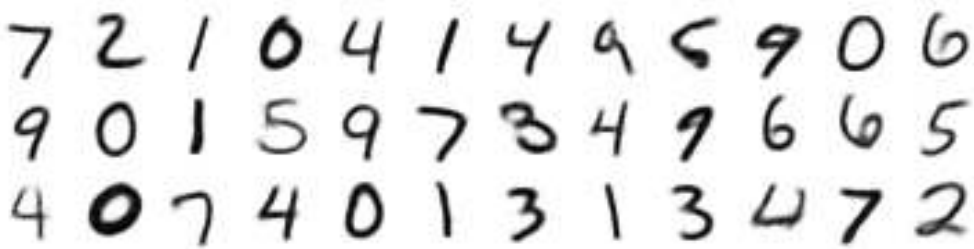
```
AutoEncoder (
  (encoder): Sequential (
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
    (3): ReLU (inplace)
    (4): Conv2d(32, 32, kernel_size=(4, 4), stride=(2, 2))
    (5): ReLU (inplace)
    (6):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

Reconstructions à partir de représentations réduites

X (original samples)



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

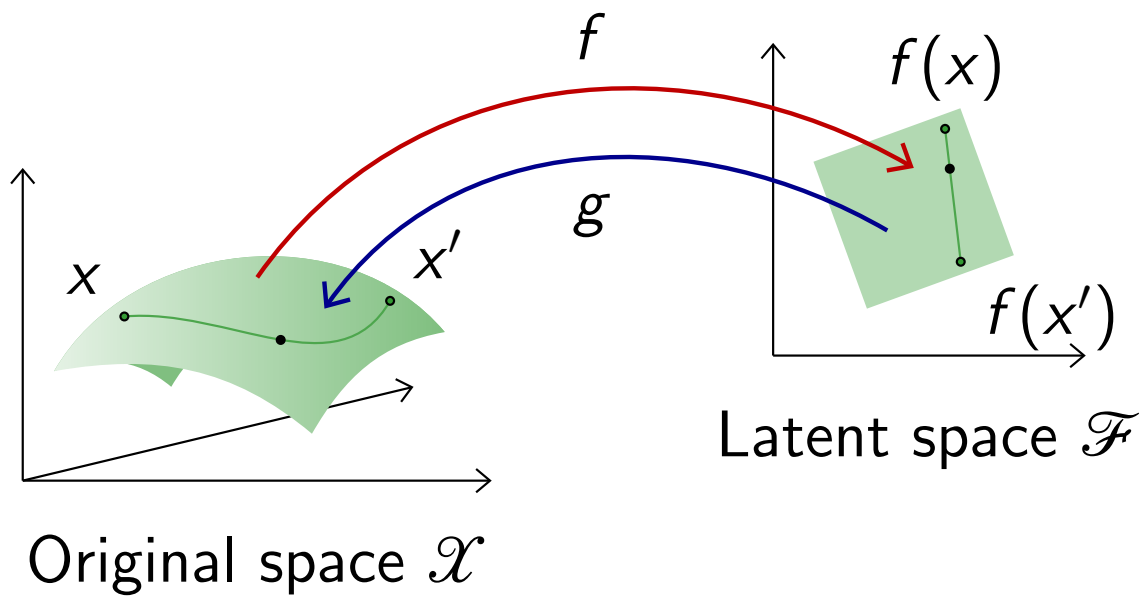


$g \circ f(X)$ (PCA, $d = 8$)

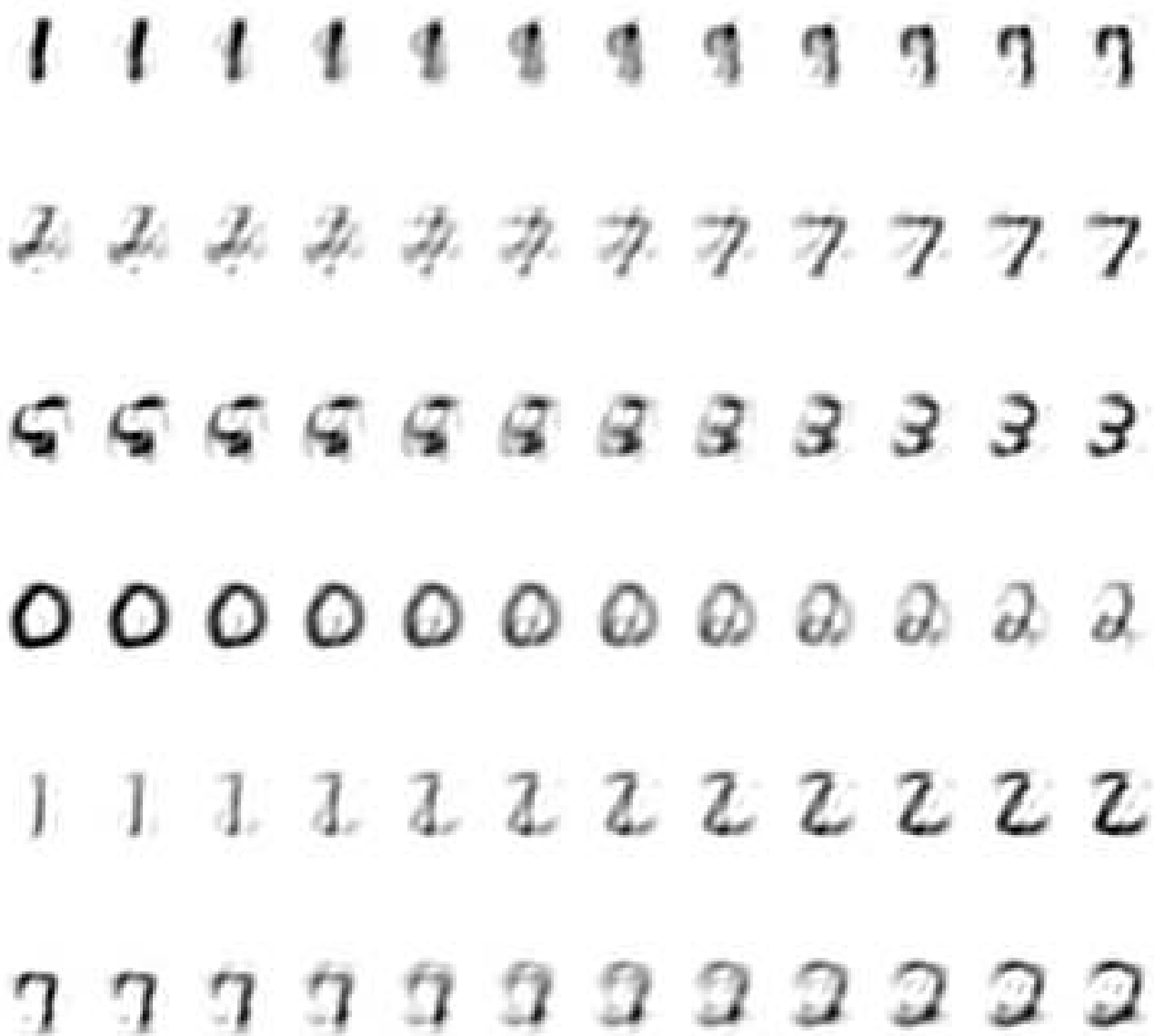
Interpolation dans l'espace latent

Pour mieux comprendre la représentation qui émerge dans cet auto-encodeur, nous pouvons prendre au hasard deux exemples de test x and x' et interpoler des exemples le long du segment qui les joint leurs images dans l'espace latent.

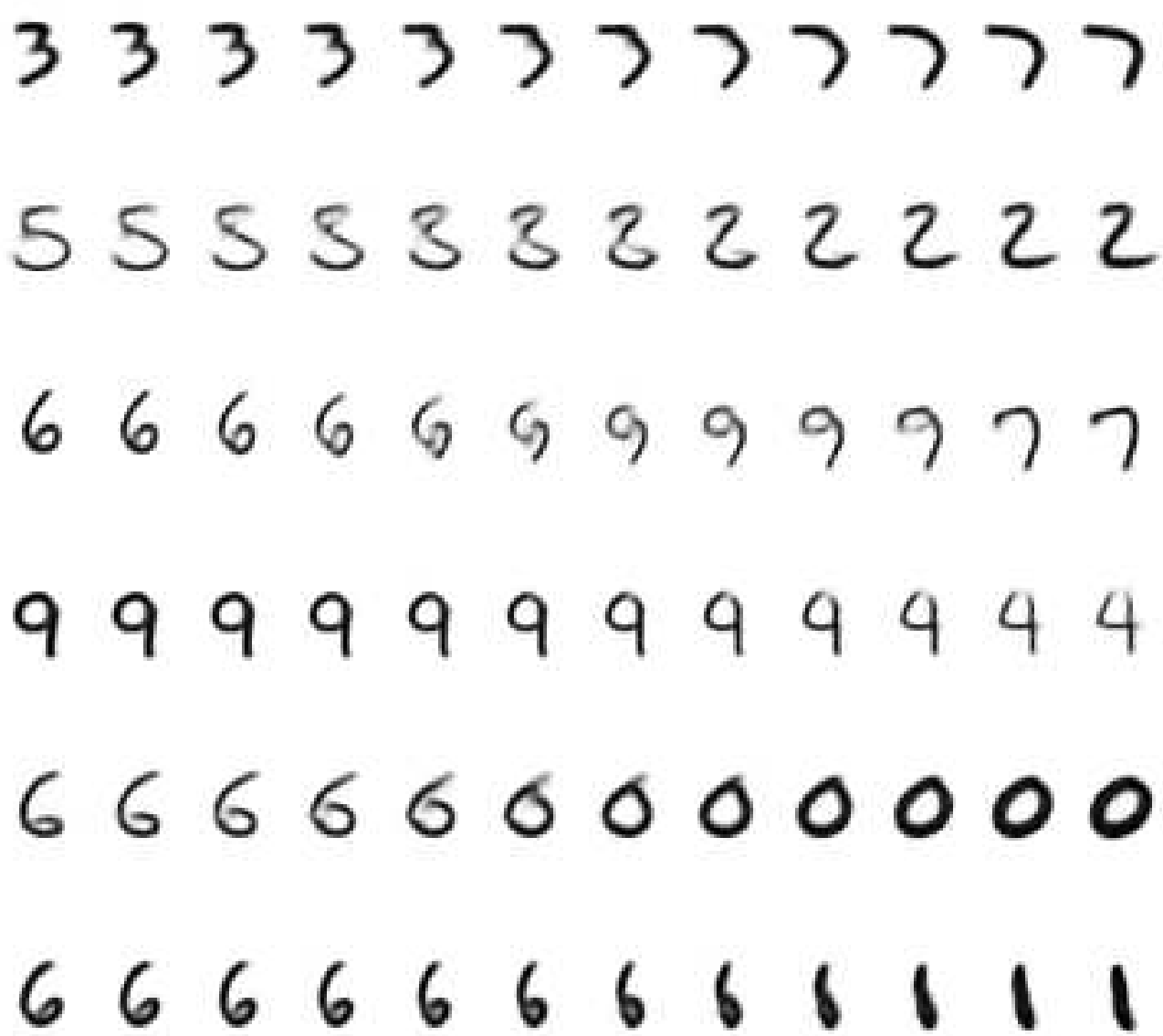
$$\forall x, x' \in \mathcal{X}^2,\ \alpha \in [0,1],\quad \xi(x, x', \alpha) = g((1-\alpha)f(x) + \alpha f(x')).$$



Interpolation avec PCA ($d = 32$)



Interpolation avec l'auto-encodeur ($d = 8$)

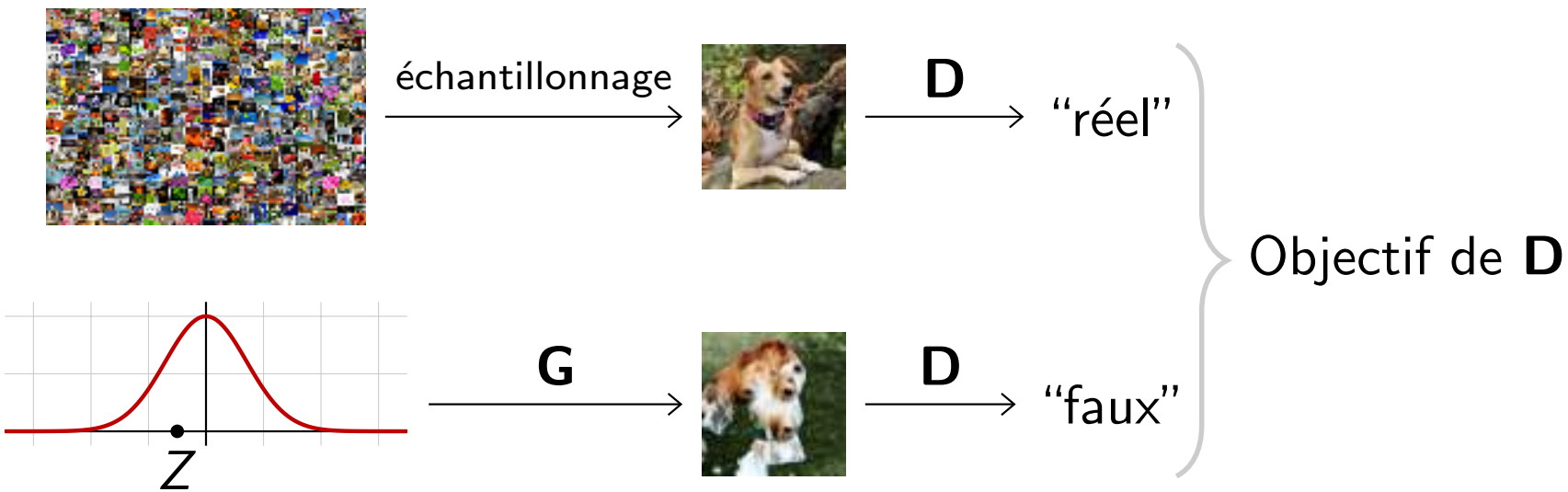


Modèles génératifs antagonistes

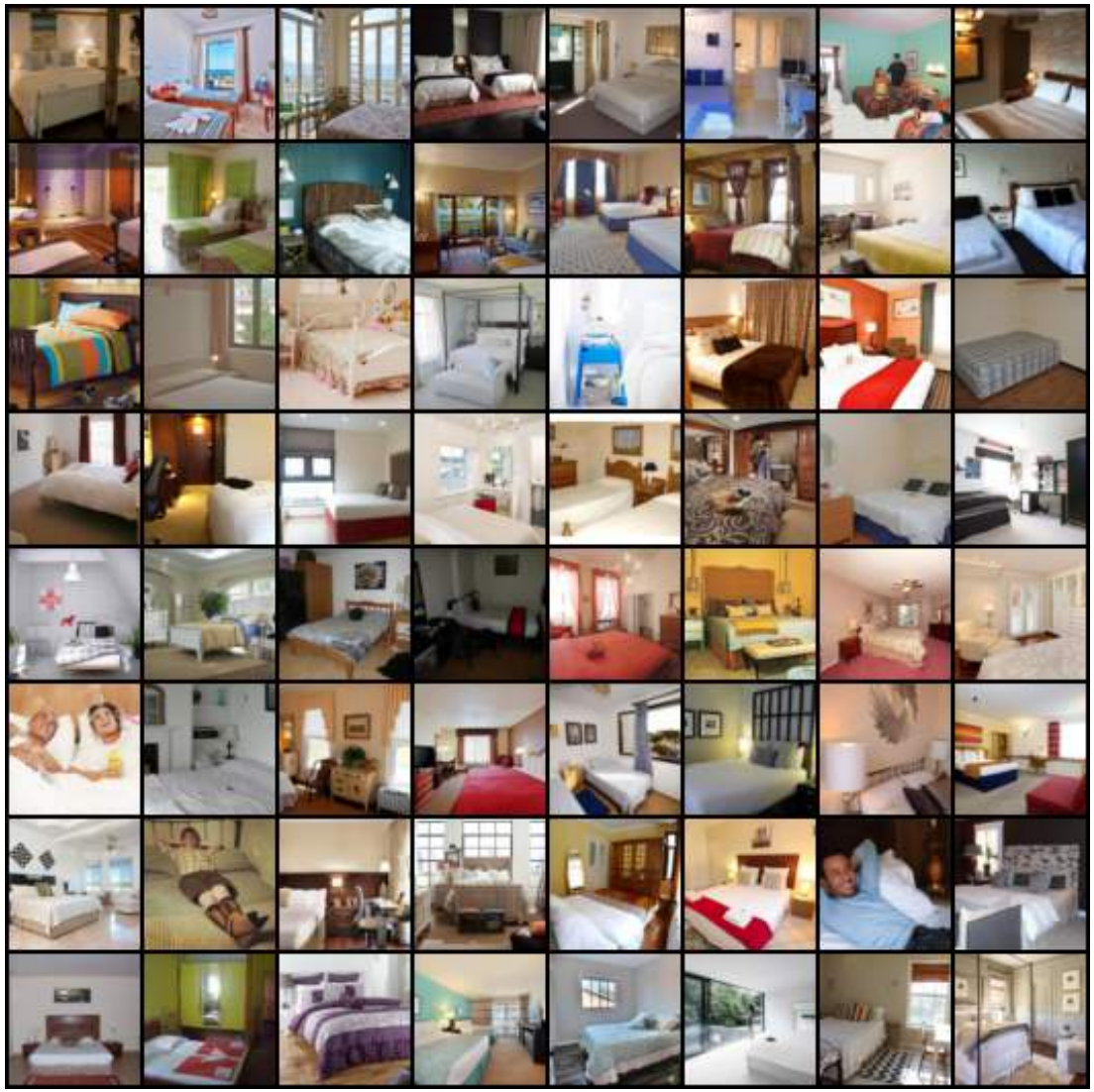
Modèles génératifs antagonistes : principe

Une approche très puissante pour modéliser des distributions en grande dimension consiste à utiliser des **modèles antagonistes** entraînés de manière couplée :

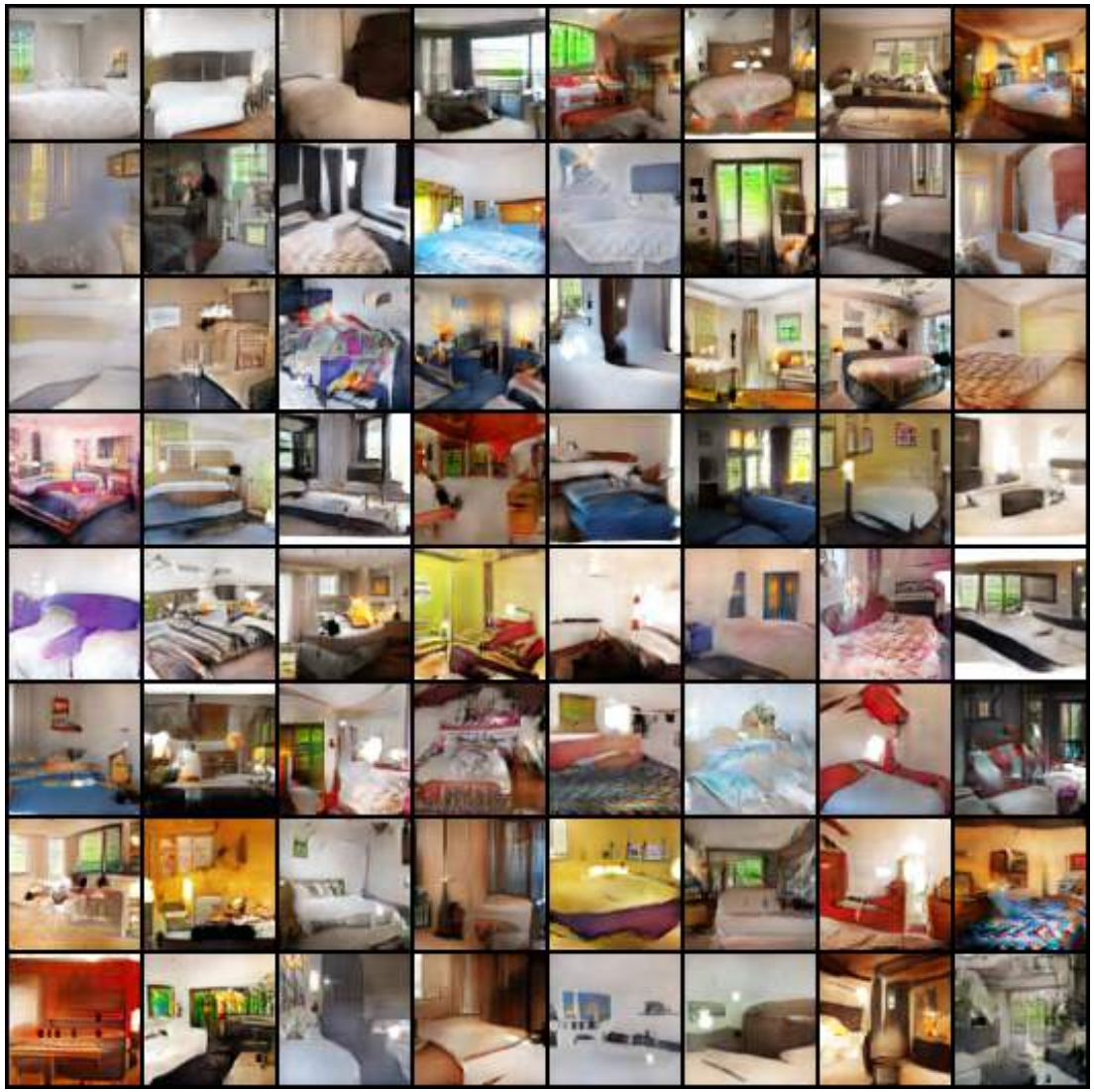
- Le **discriminateur D** doit classifier les exemples comme "réels" ou "faux",
- le **générateur G** doit transformer une distribution simple et fixée *a priori* en une distribution de points que **D** classe comme "réels".



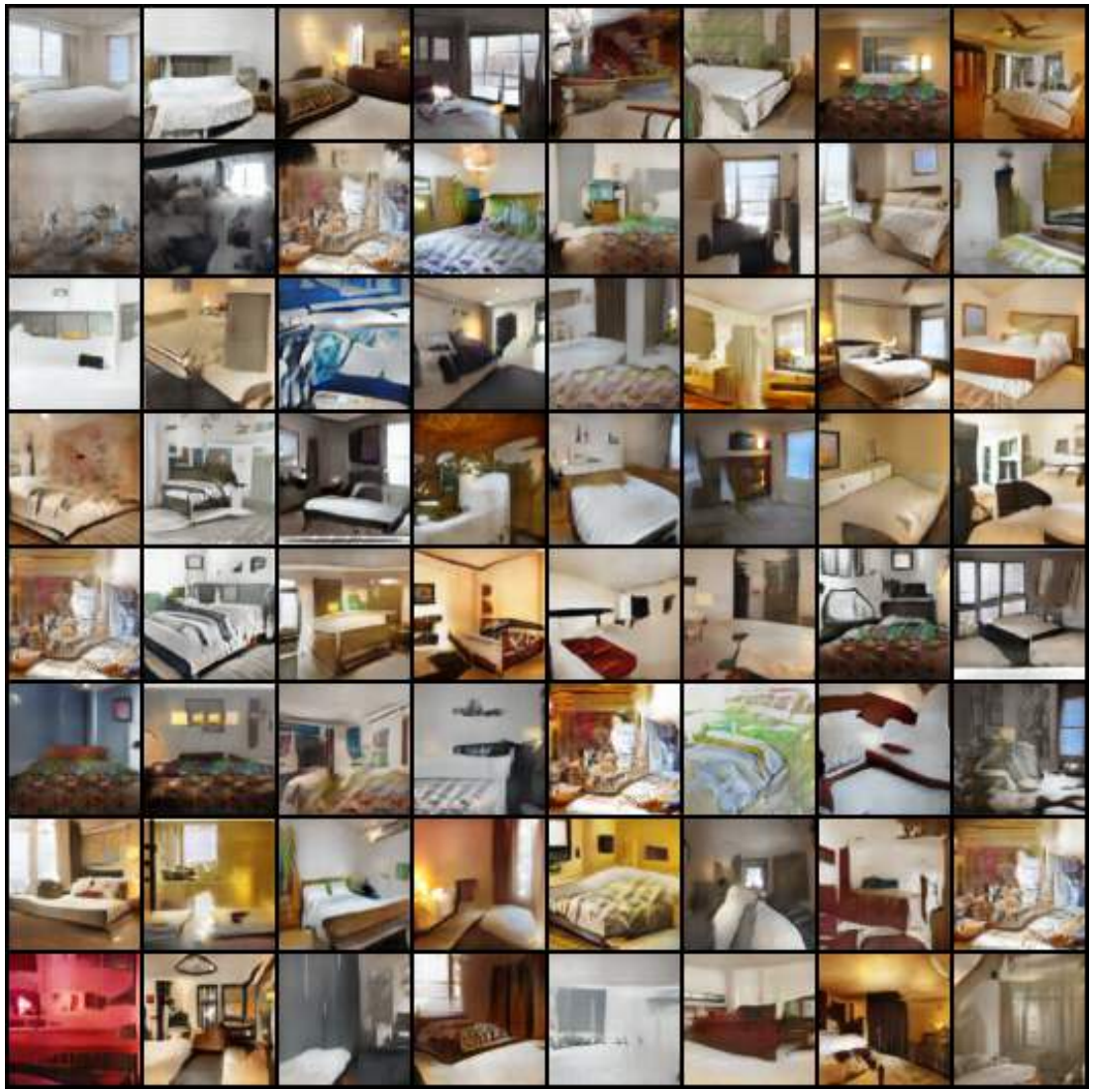
Dans cette approche les deux modèles ont donc des objectifs antagonistes.



Vraies images de la classe “bedroom” dans la base de donnée Large-scale Scene Understanding



Exemples d'images générées après 1 époque (3M images)



Exemples d'images générées après 20 époques

Exemples d'images générées par un modèle génératif antagoniste

(Karras et al., 2018)

Analyse d'un modèle profond

Interprétation d'un réseau par remontée du gradient

Il est difficile de comprendre quelles sont les représentations et les calculs qui résultent de l'entraînement d'un réseau profond.

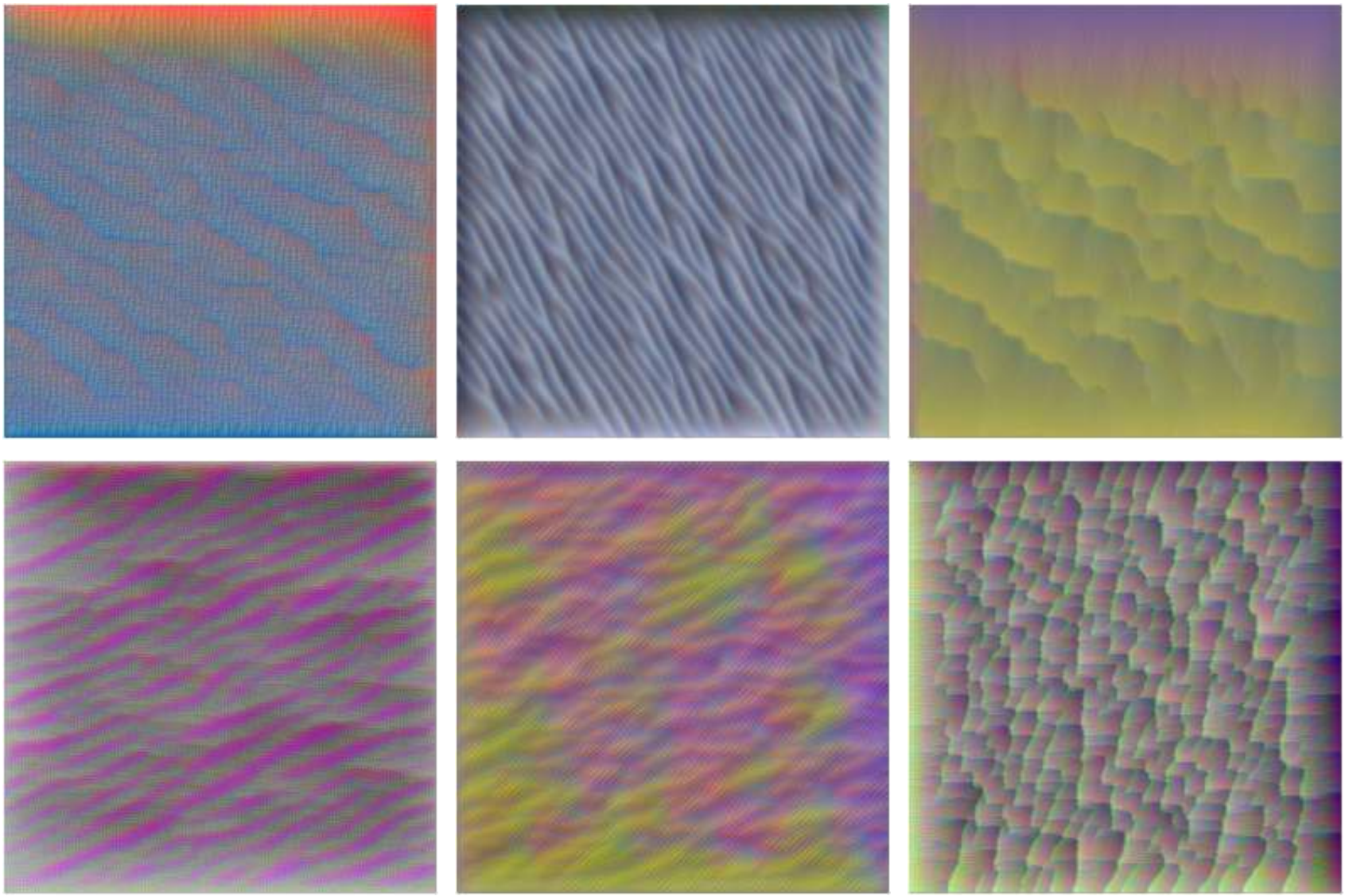
Une manière directe de visualiser ce que détecte une unité particulière consiste à optimiser l'entrée du réseau pour maximiser l'activité de l'unité.

Cela se fait avec une **remontée** du gradient :

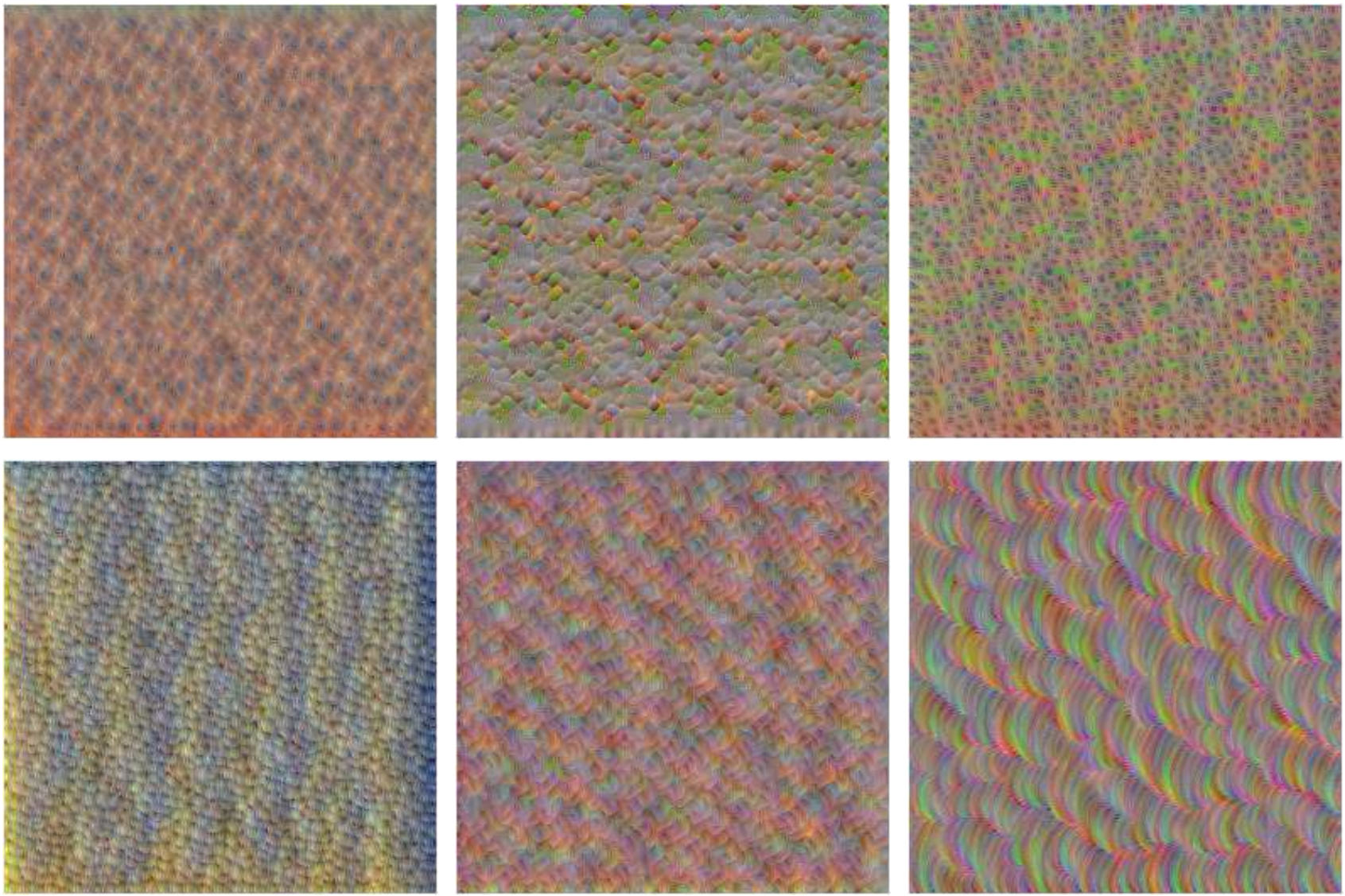
$$x_{k+1} = x_k + \eta \nabla f_c(x_k)$$

où f_c est l'activation à maximiser, qui peut être une valeur interne au réseau ou un des scores de sortie, et x_k est l'entrée après k itérations.

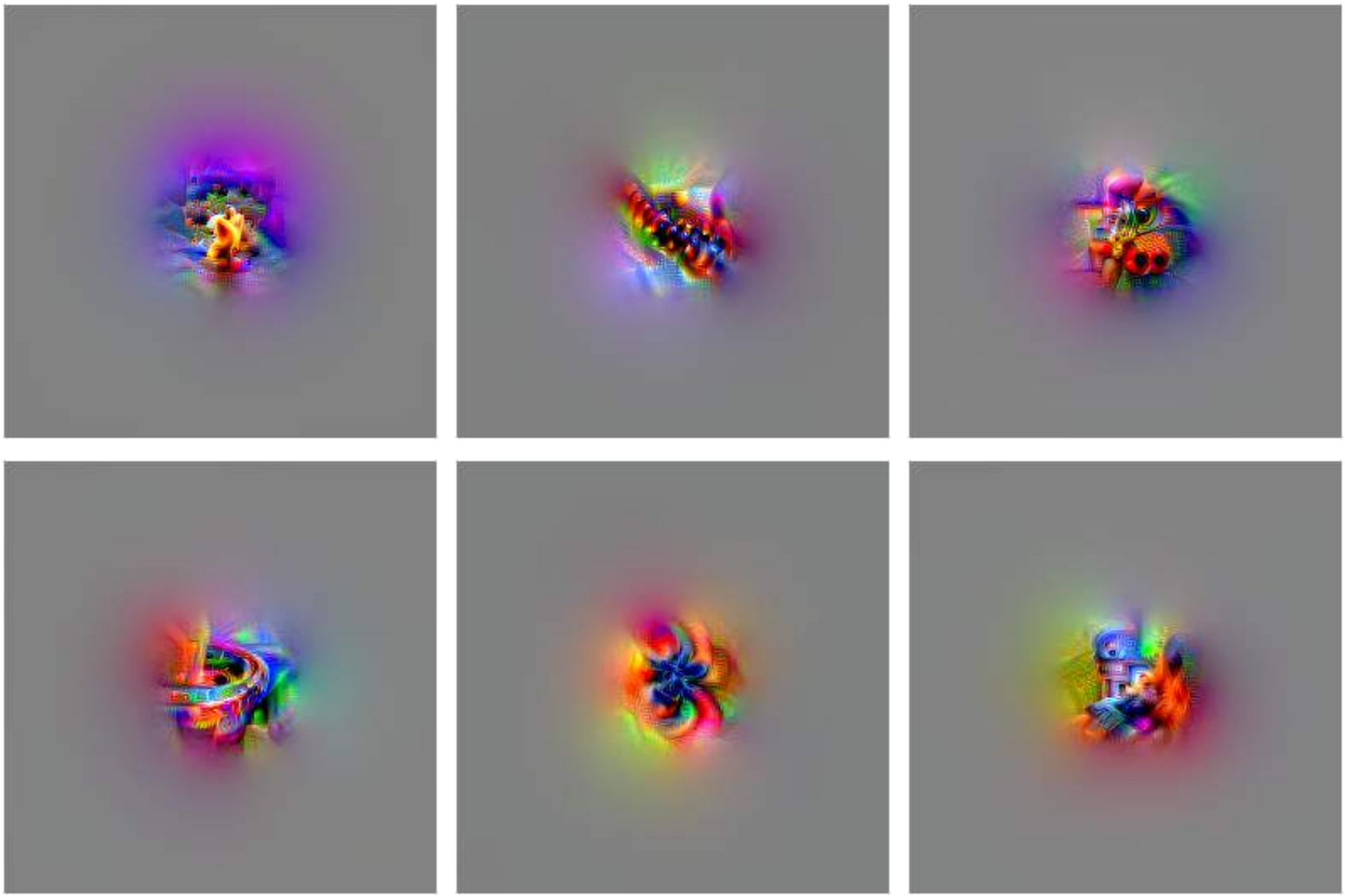
Images d'entrée optimisées chacunes pour maximiser un des canaux de la 4ème couche de convolutions d'un réseau VGG-16



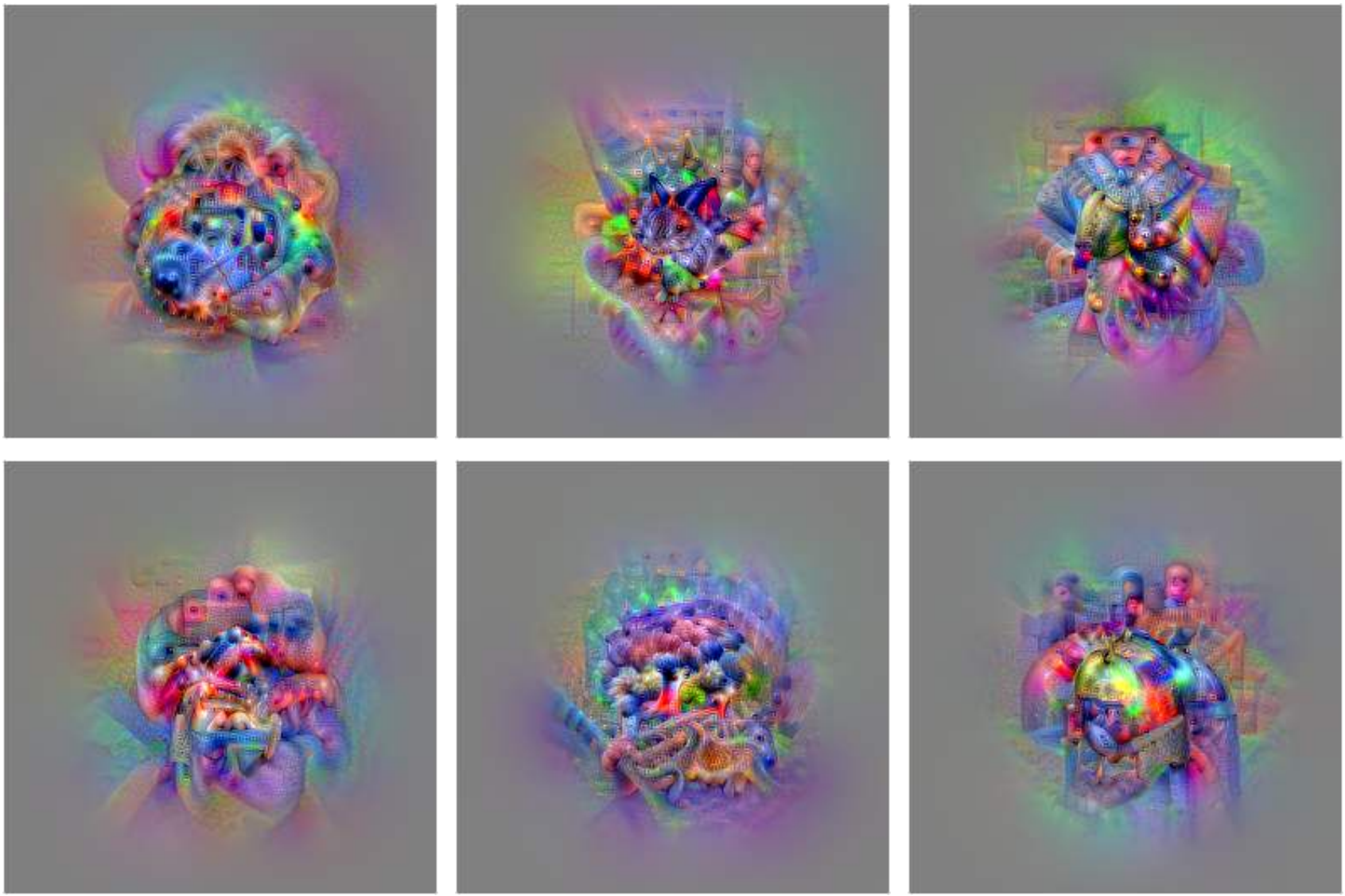
Images d'entrée optimisées chacunes pour maximiser un des canaux de la 7ème couche de convolutions d'un réseau VGG-16



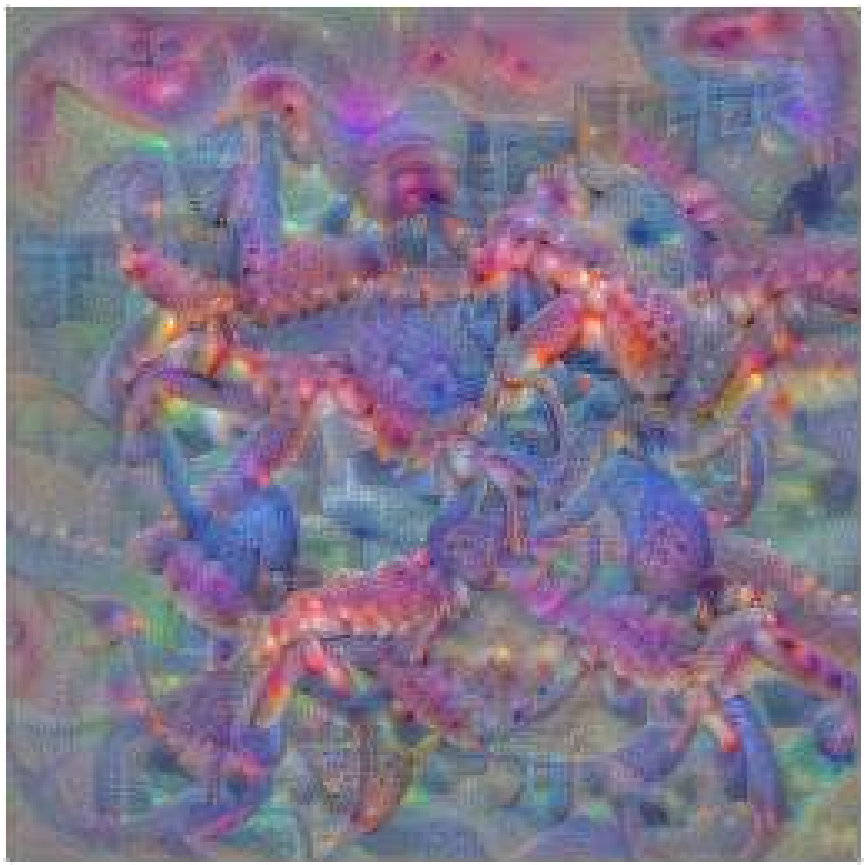
Images d'entrée optimisées chacunes pour maximiser une unité de la 10ème couche de convolutions d'un réseau VGG-16



Images d'entrée optimisées chacunes pour maximiser une unité de la 13ème, et dernière, couche de convolutions d'un réseau VGG-16



Images d'entrée optimisées chacunes pour maximiser une unité de sortie d'un réseau VGG-16

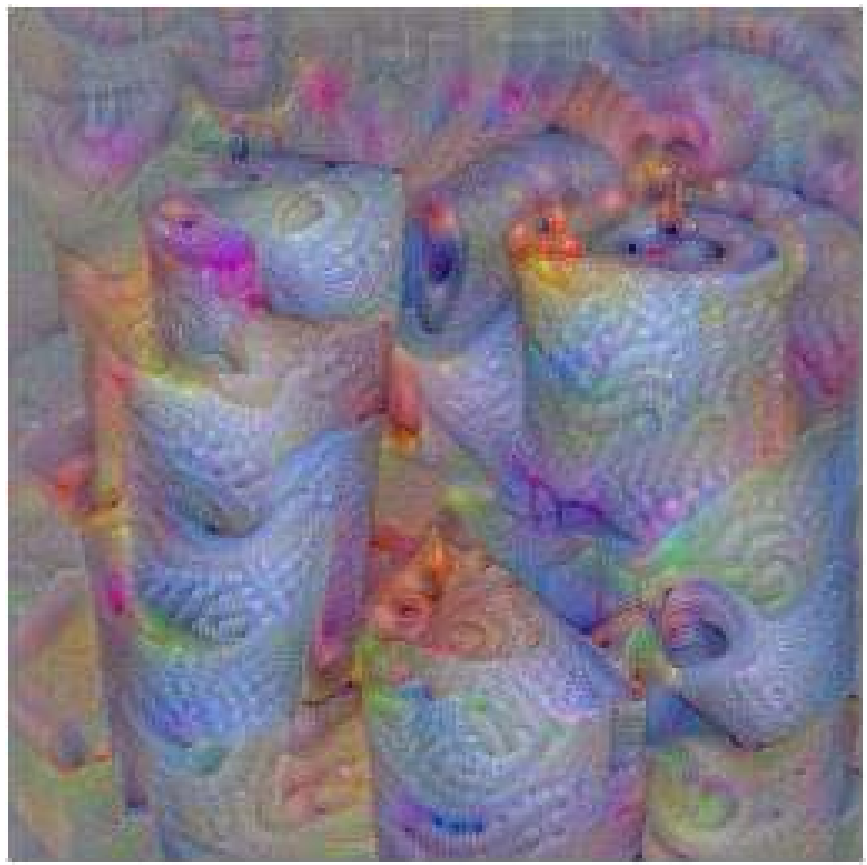


"King crab"

"Samoyed" (that's a fluffy dog)

Images d'entrée optimisées chacunes pour maximiser une unité de sortie d'un réseau VGG-16

"Hourglass"



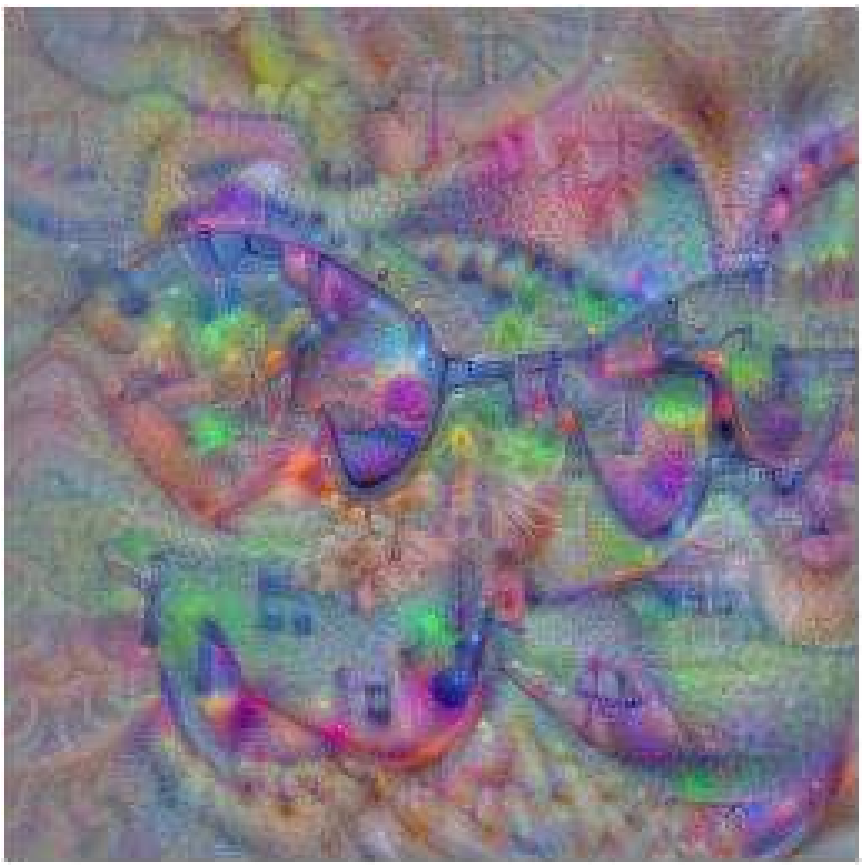
"Paper towel"

Images d'entrée optimisées chacunes pour maximiser une unité de sortie d'un réseau VGG-16

"Ping-pong ball"

"Steel arch bridge"

Images d'entrée optimisées chacunes pour maximiser une unité de sortie d'un réseau VGG-16



"Sunglass"

"Geyser"

Images d'entrée optimisées pour maximiser certaines unités d'un réseau VGG-16 : conclusions

Les résultats montrent que les paramètres d'un réseau entraîné pour des tâches de classification encodent assez d'information pour générer des parties identifiables de grandes tailles.

Ils montrent également les limitations du modèle des contraintes globales comme la symétrie ou la cardinalité.

Détermination des parties d'une image d'entrée qui ont contribué le plus à la prédiction obtenue

Une autre classe de méthodes estiment l'importance des différentes parties d'un signal d'entrée dans la modulation de la prédiction du modèle.

Le *Gradient-weighted Class Activation Mapping* (Grad-CAM) proposé par Selvaraju et al. (2016) visualise cette importance en considérant une couche interne du réseau, en général, proche de la sortie du modèle.

Grad-CAM : approche

On veut établir une carte qui indique quelles régions de l'image contribuent à la valeur de sortie y^c correspondant à une classe c.

On fixe une couche de convolution (e.g. la dernière couche de convolution). Dans cette couche, on indexe les canaux par $k \in \{1, \ldots, C\}$ et $A^k \in \mathbb{R}^{H \times W}$ sont les activation du canal k dans la couche considérée. On calcule un poids pour chaque canal k :

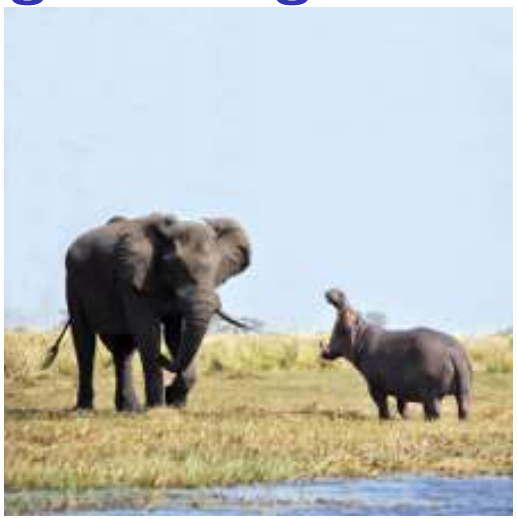
$$\alpha_k^c = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \frac{\partial y^c}{\partial A_{i,j}^k},$$

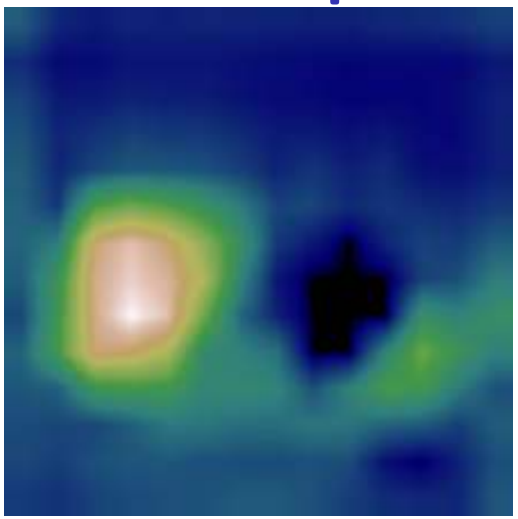
qui favorise les canaux en fonction de l'importance du gradient de la sortie y^c par rapport aux activations $A_{i,j}^k$

Et la carte de localisation finale est obtenue en calculant une somme pondérée (et rectifiée) des activations A^k :

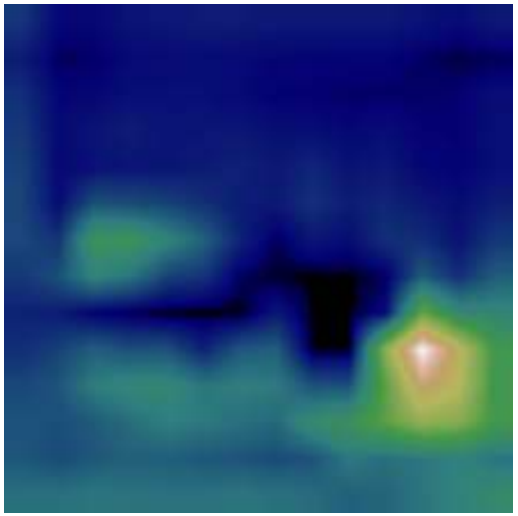
$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_{k=1}^{C} \alpha_k^c A^k\right).$$

Images et régions d'activations pour deux classes

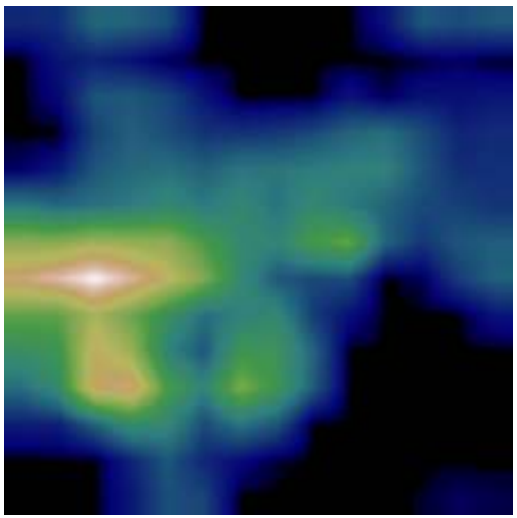




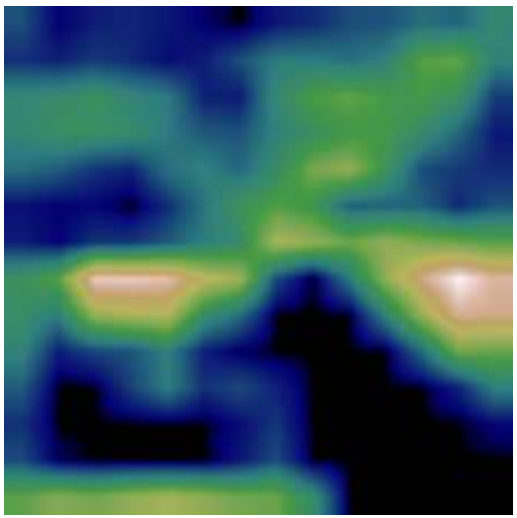
African elephant



Hippopotamus

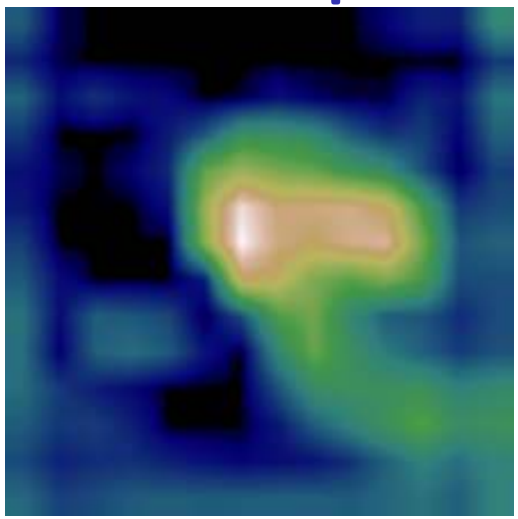


Ox

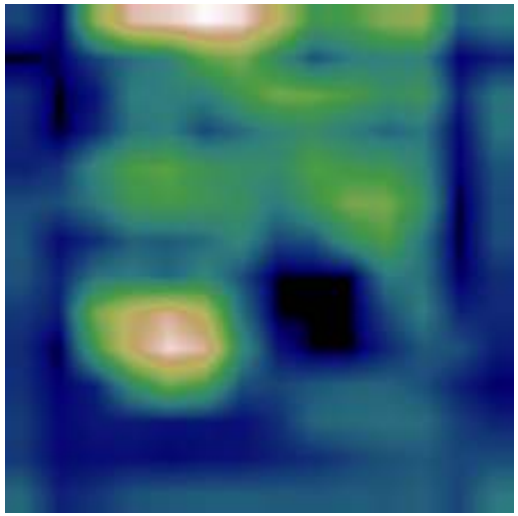


Fountain

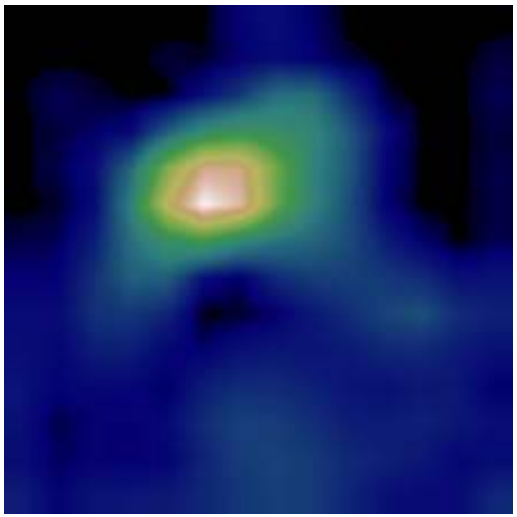
Images et régions d'activations pour deux classes



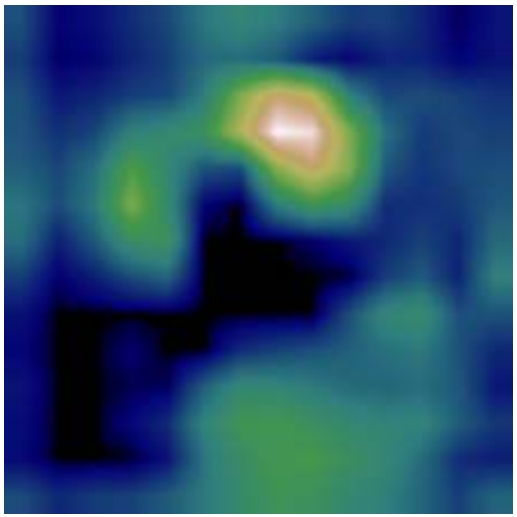
Coffee mug



Bagel



Bee



Daisy

Résumé : réseaux profonds et convolutionnels

- Les graphes acycliques orientés (direct acyclic graphs (DAG) sont des généralisation du perceptron multi-couche
- Implémentation à l'aide d'opérateurs tensoriels et calculs de gradient automatisés
- Descente de gradient stochastique et mini-batches : accélération de convergence et évitement des minima locaux
- Opérations de convolutions et *pooling* : les réseaux neuronaux convolutionnels sont souvent mieux adaptés que des réseaux entièrement connectés
- Auto-encodeurs et représentations latentes : représentations parcimonieuses et possibilités d'interpolation
- Modèles génératifs antagonistes : générateur d'images contre discriminateur vrai/faux
- Analyse des modèles profonds :
    - Images d'entrées qui maximisent l'activité d'unités du réseau
    - Cartographie des régions d'une images qui contribuent le plus à la prédiction

References

T. Karras, S. Laine, and T. Aila. **A style-based generator architecture for generative adversarial networks**. CoRR, abs/1812.04948, 2018.

Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. **Gradient-based learning applied to document recognition**. Proceedings of the IEEE, 86(11) :2278–2324, 1998.

R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. **Grad-cam : Visual explanations from deep networks via gradient-based localization**. CoRR, abs/1610.02391, 2016.