Architectures de réseaux de neurones

CSI 4506 - automne 2026

Marcel Turcotte
Version: 4 sept. 2026 17h02

Préambule

Message du jour

Will transformers drive AI in 10 years?, Andrej Karpathy, 2025-10-27.

Andrej Karpathy est un informaticien slovaque-canadien né en 1986 à Bratislava. Il s'est installé à Toronto avec sa famille à l'âge de 15 ans. Il a obtenu un baccalauréat en informatique et en physique à l'Université de Toronto (2009), puis une maîtrise à l'Université de la Colombie-Britannique (2011), où il a travaillé avec son directeur Michiel van de Panne sur l'apprentissage de contrôleurs pour des personnages simulés physiquement. Il a ensuite obtenu un doctorat à l'Université Stanford sous la direction de Fei-Fei Li, à l'intersection de la vision par ordinateur et du traitement automatique du langage naturel.

- Membre fondateur et chercheur à **OpenAI** (2015–2017).
- Directeur principal de l'IA et de la vision pour Autopilot chez **Tesla, Inc.** (2017–2022).

- Retour à **OpenAI** en 2023 pour diriger une petite équipe travaillant à l'amélioration des modèles, avant de quitter l'entreprise en 2024.
- Fondateur d'**Eureka Labs** (depuis 2024), une entreprise consacrée à l'IA et à l'éducation.

Résultats d'apprentissage

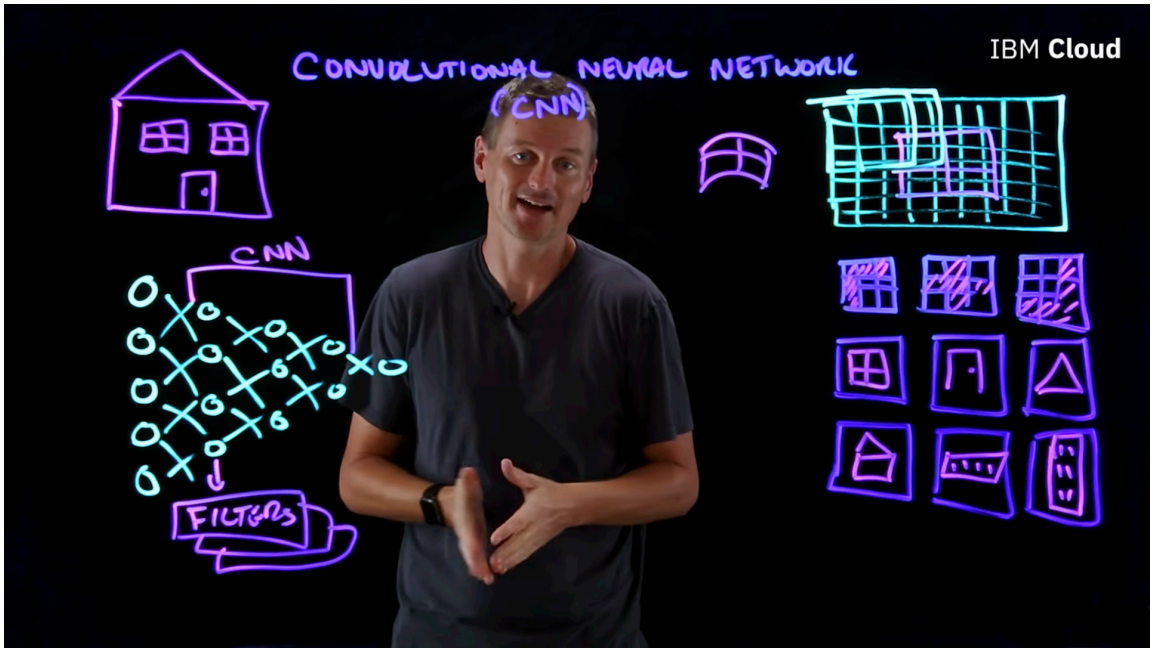
À la fin de ce cours, vous devriez pouvoir :

- **expliquer** comment la connectivité locale et le partage des paramètres confèrent un biais inductif aux réseaux convolutifs et réduisent le nombre de paramètres ;
- **calculer et suivre** le passage avant d'une couche convolutive 1D comportant plusieurs canaux d'entrée, noyaux, biais et activations ;
- **déterminer** comment la taille du noyau, le remplissage, le pas et le pooling influencent la forme de la sortie, le comportement aux frontières et l'information de position conservée ;
- **expliquer** comment l'empilement des couches compose les attributs, agrandit les champs récepteurs et alimente une tête de prédiction ;
- **distinguer** l'équivariance par translation de l'invariance, notamment les rôles de la convolution et de l'agrégation des positions.

La compétence centrale consiste à raisonner sur une valeur de sortie, puis à généraliser le même calcul aux différentes positions, aux noyaux, aux canaux et aux couches. Vous devriez pouvoir suivre les formes des tableaux et leurs indices, et expliquer les conséquences des choix architecturaux sur la modélisation.

Vous n'avez pas à dériver les gradients d'une convolution, à mémoriser les valeurs des noyaux conçus à la main, à reproduire le code de visualisation ni à mémoriser les détails d'architectures nommées comme AlexNet et VGG.

Réseau de neurones convolutif (CNN)



Une excellente vue d'ensemble des CNN.

Motivation

> « Un MLP comportant **une seule couche cachée** peut théoriquement modéliser même les **fonctions les plus complexes**, pourvu qu'il **contienne suffisamment de neurones**. Cependant, pour les problèmes complexes, les **réseaux profonds** utilisent leurs paramètres de façon beaucoup **plus efficace** que les réseaux peu profonds : ils peuvent modéliser des fonctions complexes avec **un nombre exponentiellement inférieur de neurones**, et ainsi obtenir de **bien meilleures performances** avec la même quantité de données d'entraînement. »

Géron (2019) § 10

Les réseaux de neurones étudiés jusqu'ici sont composés de neurones organisés en couches, chaque unité d'une couche étant entièrement connectée à toutes les unités de la couche suivante. Le nombre de couches et le nombre de neurones par couche peuvent varier, mais ce principe d'organisation demeure le même dans tout le réseau.

En théorie, sous des hypothèses raisonnables, un tel réseau peut approximer toute fonction complexe. En pratique, toutefois, le nombre de poids croît rapidement et les réseaux qui en résultent peuvent être difficiles à entraîner.

Pour répondre à ces difficultés, on a proposé des architectures dotées de **biais inductifs** explicites. Parmi les grandes familles de modèles figurent les architectures séquentielles et temporelles, comme les réseaux neuronaux récurrents (par exemple LSTM et GRU) et les transformeurs. Une autre famille importante regroupe les modèles

génératifs, notamment les autoencodeurs, les réseaux antagonistes génératifs et les modèles de diffusion. Enfin, les réseaux neuronaux de graphes sont conçus pour traiter des données relationnelles.

Un **biais inductif** désigne les hypothèses préalables sur lesquelles un algorithme d'apprentissage s'appuie pour généraliser des données d'entraînement à des exemples jamais observés. Il contraint ou façonne l'espace des hypothèses, ce qui permet au modèle de privilégier certaines solutions plausibles parmi plusieurs qui pourraient expliquer également bien les données observées. Sans un tel biais, un algorithme ne peut pas généraliser de façon fiable au-delà des exemples précis sur lesquels il a été entraîné.

Par exemple, la régression logistique possède un biais inductif fort puisqu'elle suppose que la relation entre l'entrée et la sortie peut être convenablement modélisée par une fonction linéaire. À l'inverse, un polynôme de degré élevé impose une hypothèse préalable plus faible, car ses paramètres lui permettent de représenter des relations beaucoup plus complexes.

Les réseaux de neurones convolutifs (ConvNets ou CNN) se caractérisent par un biais inductif fort, ce qui les rend très efficaces lorsque la tâche correspond à ce biais. Les CNN sont à la base de nombreux systèmes modernes de vision, notamment ceux des véhicules autonomes.

Nous utilisons donc les réseaux convolutifs pour introduire la notion de **conception architecturale**.

Motivation (suite)

- Considérons une **image RVB** de dimensions 224×224, relativement petite selon les normes actuelles.
- Elle contient $224 \times 224 \times 3 = 150\,528$ **attributs d'entrée**.
- Un réseau ne comportant qu'une **seule couche cachée dense** nécessiterait plus de 22 658 678 784 (**22 milliards**) de paramètres, ce qui illustre l'ampleur du calcul.

Nous supposons ici que la couche cachée contient autant d'unités que la couche d'entrée. Ce choix préserve la dimension de l'espace des attributs et évite d'imposer un goulot d'étranglement arbitraire dès la première couche. Le réseau possède ainsi une capacité suffisante pour modéliser des combinaisons complexes et non linéaires de tous les pixels avant la réduction de résolution.

Dans un réseau à propagation avant entièrement connecté de k couches contenant chacune n unités, le nombre de paramètres croît selon $\mathcal{O}(kn^2)$.

Concepts

Notre discussion portera sur les **concepts clés** suivants.

- **Biais inductif** et hypothèses topologiques
- Connectivité locale et **interactions clairsemées**
- **Partage des paramètres** et équivariance
- **Composition hiérarchique** des attributs
- **Équivariance** par translation et **invariance** par agrégation des positions

Approche pédagogique

Commencer par le traitement classique des images 2D offre une entrée intuitive et établit la motivation avant d'introduire la méthode sous-jacente. Cette approche permet de distinguer clairement l'**intuition visuelle** de la convolution des **opérations matricielles** nécessaires à sa mise en œuvre dans un réseau de neurones.

Phase 1 : l'intuition (contexte historique en 2D)

Objectif : montrer, par un retour visuel immédiat, ce que fait réellement un filtre.

- **Contenu :** appliquer à une image en niveaux de gris des filtres 2D classiques conçus à la main, par exemple les filtres de Sobel pour la détection de contours.
- **Idée essentielle :** une petite matrice de poids se déplace sur l'entrée et la transforme en carte d'attributs. Le champ récepteur est introduit intuitivement; les canaux, les lots et la rétropropagation sont d'abord laissés de côté.

Les CNN 2D constituent une architecture fondamentale pour les images. Les CNN 1D sont toutefois aussi très utilisés. Les CNN 3D, bien établis, servent surtout aux données volumétriques comme les images d'IRM et de tomodensitométrie. Pour simplifier autant que possible les équations et les représentations visuelles, nous commencerons par les CNN 1D.

Phase 2 : la mécanique (réseaux neuronaux 1D)

Objectif : isoler les mathématiques des canaux et des noyaux.

- **Transition :** en vision par ordinateur classique, des personnes choisissaient les poids des matrices; les couches convolutives, elles, les *apprennent*. Pour représenter mathématiquement cet apprentissage et les représentations profondes, nous retirons d'abord la complexité spatiale.
- **Contenu :** suivre la progression en 1D : un canal, plusieurs noyaux, plusieurs canaux, puis plusieurs couches.
- **Idée essentielle :** maîtriser le calcul des produits scalaires et les règles dimensionnelles strictes régissant C_{in} et C_{out}.

Phase 3 : la synthèse (réseaux neuronaux 2D)

Objectif : réintroduire les dimensions spatiales dans le calcul tensoriel.

- **Contenu :** réappliquer aux images de la phase 1 les mathématiques développées à la phase 2.
- **Idée essentielle :** passer à la 2D ajoute simplement un indice à la logique des canaux déjà comprise en 1D.

Traitement classique des images

Intuition et contexte historique en 2D

- Le **traitement classique des images 2D** établit la **motivation** avant d'introduire la **méthode** sous-jacente.
- Il distingue l'**intuition visuelle** des **opérations matricielles**.

Nous introduirons les **concepts d'apprentissage** plus tard; pour l'instant, nous ne faisons que traiter des images.

Un noyau parcourt une image

In [2]:
```
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
from matplotlib.patches import Rectangle
from PIL import Image

SAMPLE_RATE = 50

BLUE = "#0072B2"
ORANGE = "#D55E00"
GREEN = "#009E73"
PURPLE = "#CC79A7"
GRAY = "#5B6573"
LIGHT_GRAY = "#D8DCE2"
HIGHLIGHT = "#F0E442"

plt.rcParams.update({
    "axes.spines.top": False,
    "axes.spines.right": False,
    "axes.titleweight": "bold",
    "axes.labelsize": 12,
    "axes.titlesize": 14,
    "font.size": 12,
    "legend.frameon": False,
    "lines.linewidth": 2.2,
})

# Petit extrait en format long des fichiers de signaux inertiels UCI HAR.
```

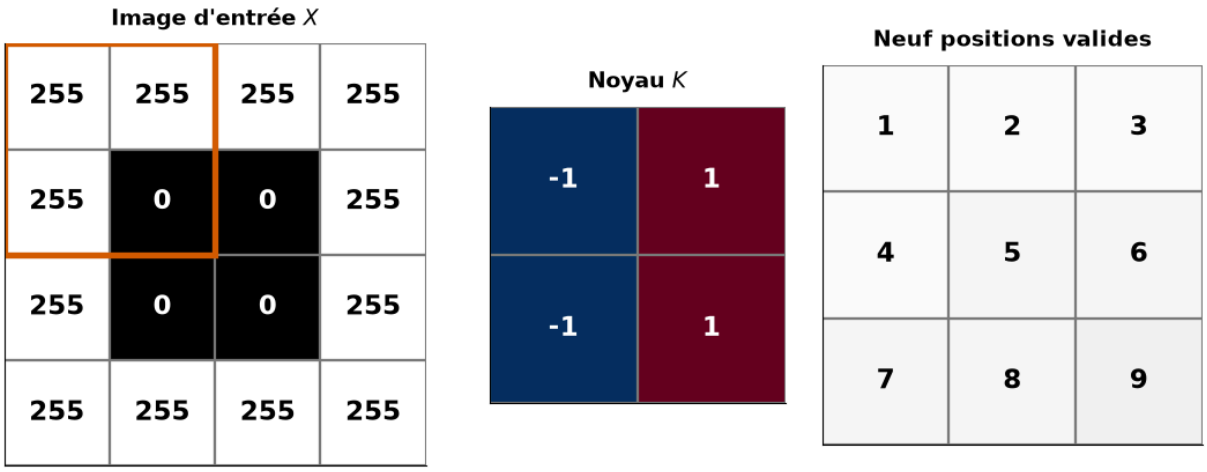
```
data_path = Path("data/uci_har_prototype.csv")
signals = pd.read_csv(data_path)

walking = (
    signals.loc[signals["activity"] == "walking"]
    .sort_values("sample")
    .reset_index(drop=True)
)
sitting = (
    signals.loc[signals["activity"] == "sitting"]
    .sort_values("sample")
    .reset_index(drop=True)
)

time = walking["sample"].to_numpy() / SAMPLE_RATE
x_walking = walking["x"].to_numpy()


def style_time_axis(ax, *, xlabel=False, ylabel=None, ylim=None):
    """Appliquer un style visuel commun aux graphiques de signaux."""
    ax.axhline(0, color=LIGHT_GRAY, linewidth=1, zorder=0)
    ax.grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
    ax.set_xlim(time[0], time[-1])
    if ylim is not None:
        ax.set_ylim(*ylim)
    if ylabel:
        ax.set_ylabel(ylabel)
    if xlabel:
        ax.set_xlabel("temps (s)")
    else:
        ax.tick_params(labelbottom=False)
```

In [3]:

```
def cross_correlation2d(X, K):
    """Appliquer le noyau 2D tel quel, par corrélation croisée valide."""
    output_rows = X.shape[0] - K.shape[0] + 1
    output_cols = X.shape[1] - K.shape[1] + 1
    Y = np.zeros((output_rows, output_cols), dtype=float)

    for i in range(output_rows):
        for j in range(output_cols):
            total = 0.0
            for u in range(K.shape[0]):
                for v in range(K.shape[1]):
                    total += X[i + u, j + v] * K[u, v]
            Y[i, j] = total

    return Y


def draw_number_matrix(ax, values, title, *, cmap="Greys", vmin=None,
                       vmax=None, text_color=None):
    """Afficher une petite matrice et ses valeurs numériques."""
    ax.imshow(values, cmap=cmap, vmin=vmin, vmax=vmax)
    rows, cols = values.shape
    norm = Normalize(vmin=np.min(values) if vmin is None else vmin,
                     vmax=np.max(values) if vmax is None else vmax)
```

```
    color_map = plt.get_cmap(cmap)
    for i in range(rows):
        for j in range(cols):
            value = 0.0 if np.isclose(values[i, j], 0.0) else values[i, j]
            ax.add_patch(Rectangle(
                (j - 0.5, i - 0.5), 1, 1,
                fill=False, edgecolor="#7A7A7A", linewidth=1.6,
            ))
            if text_color is None:
                red, green, blue, _ = color_map(norm(value))
                luminance = 0.2126 * red + 0.7152 * green + 0.0722 * blue
                cell_text_color = "black" if luminance > 0.52 else "white"
            else:
                cell_text_color = text_color
            ax.text(j, i, f"{value:g}", ha="center", va="center",
                    color=cell_text_color, fontsize=17, fontweight="bold")
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(title, pad=12)


X_image = np.array([
    [255, 255, 255, 255],
    [255,   0,   0, 255],
    [255,   0,   0, 255],
    [255, 255, 255, 255],
], dtype=float)

K_edge = np.array([
    [-1, 1],
    [-1, 1],
], dtype=float)

Y_image = cross_correlation2d(X_image, K_edge)
placement_order = np.arange(1, 10).reshape(3, 3)

fig, axes = plt.subplots(
    1, 3,
    figsize=(11.4, 4.4),
    gridspec_kw={"width_ratios": [1.25, 0.78, 1.0]},
)

draw_number_matrix(
    axes[0], X_image, "Image d'entrée $X$", cmap="Greys_r", vmin=0, vmax=255
)
axes[0].add_patch(Rectangle(
    (-0.5, -0.5), 2, 2,
    facecolor="none", edgecolor=ORANGE, linewidth=4,
))
draw_number_matrix(
    axes[1], K_edge, "Noyau $K$", cmap="RdBu_r", vmin=-1, vmax=1,
)

draw_number_matrix(
    axes[2], placement_order, "Neuf positions valides", cmap="Greys",
    vmin=0, vmax=100,
```

```
)

fig.tight_layout(w_pad=2.0)
plt.show()
```



Un **noyau** (aussi appelé filtre) est une petite matrice, généralement de taille 3×3, 5×5 ou similaire, qui **glisse sur** les données d'entrée, par exemple une image, afin d'effectuer une **corrélation croisée**.

Une image est un tableau bidimensionnel de nombres. Ici, selon l'échelle conventionnelle d'une image en niveaux de gris sur 8 bits, 0 représente un pixel noir et 255 un pixel blanc. Ces seize intensités forment une petite lettre O.

La petite matrice 2×2 est un noyau. Nous alignons d'abord son élément supérieur gauche sur le pixel supérieur gauche de l'entrée, puis nous le décalons d'une colonne à la fois. À la fin d'une ligne, nous poursuivons sur la ligne suivante. Une entrée 4×4 et un noyau 2×2 donnent $3 \times 3 = 9$ positions valides.

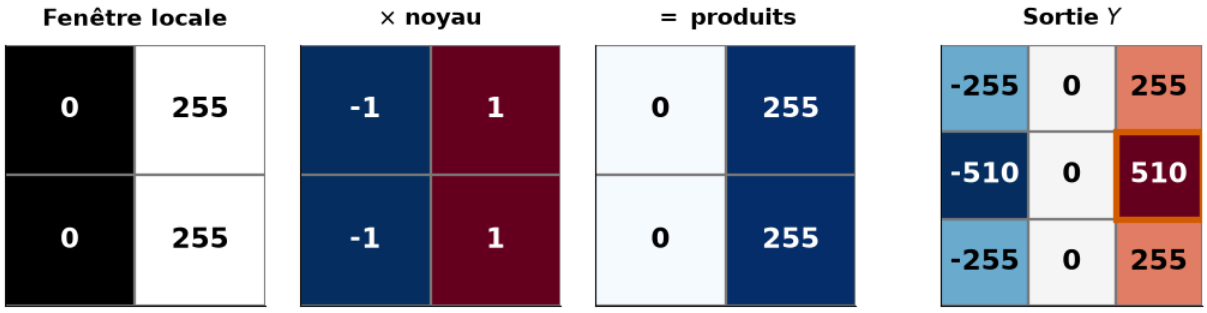
Nous ne retenons que les positions où le noyau entier chevauche l'entrée. On parle de corrélation croisée **valide**. Le traitement des frontières sera introduit plus tard.

Que calcule chaque position ?

$$Y[i, j] = \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} X[i+u, j+v] K[u, v]$$

In [4]:
```
i, j = 1, 2
patch = X_image[i:i + K_edge.shape[0], j:j + K_edge.shape[1]]
products = patch * K_edge

fig, axes = plt.subplots(
    1, 4,
    figsize=(11.2, 3.25),
    gridspec_kw={"width_ratios": [0.9, 0.9, 0.9, 1.25]},
)
```

```
draw_number_matrix(axes[0], patch, "Fenêtre locale", cmap="Greys_r", vmin=0,
draw_number_matrix(axes[1], K_edge, r"$\times$ noyau", cmap="RdBu_r", vmin=-
draw_number_matrix(axes[2], products, "$=$ produits", cmap="Blues", vmin=0,
draw_number_matrix(axes[3], Y_image, "Sortie $Y$", cmap="RdBu_r", vmin=-510,
axes[3].add_patch(Rectangle(
    (j - 0.5, i - 0.5), 1, 1,
    facecolor="none", edgecolor=ORANGE, linewidth=4,
))

fig.text(
    0.50, 0.02,
    r"$(0)(-1)+(255)(1)+(0)(-1)+(255)(1)=510$",
    ha="center", fontsize=18, color=GRAY,
)
fig.tight_layout(rect=[0, 0.11, 1, 1], w_pad=1.8)
plt.show()
```



$$(0)(-1) + (255)(1) + (0)(-1) + (255)(1) = 510$$

Chaque position produit un nombre dans la carte de réponse (**carte d'attributs**) de sortie.

À la position $(i, j) = (1, 2)$, le noyau chevauche deux pixels noirs et deux pixels blancs. Nous multiplions les éléments correspondants, puis additionnons les quatre produits. Le résultat, 510, est placé dans $Y[1, 2]$.

Le même calcul est répété à chaque position. Le bord droit du O produit des réponses positives, le bord gauche des réponses négatives, et la colonne centrale une réponse nulle. Le signe indique donc le sens du changement, et non seulement la présence d'un contour.

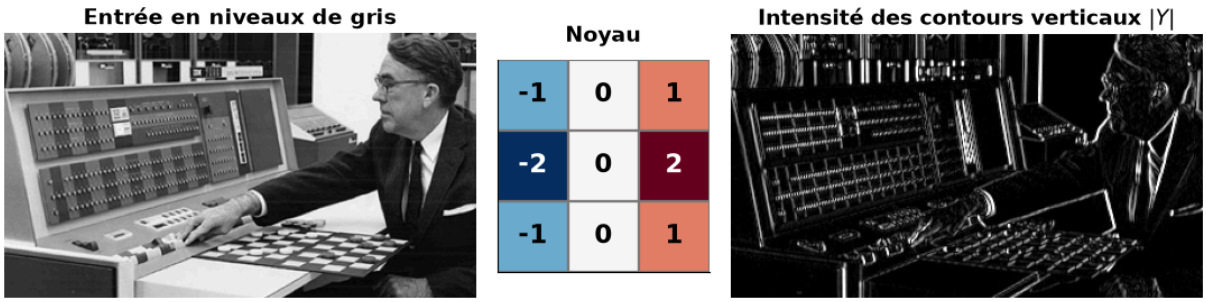
Cette opération est une **corrélation croisée**, puisque le noyau est utilisé tel quel. Dans la convolution mathématique, le noyau est d'abord renversé selon ses deux axes. Par convention, les bibliothèques d'apprentissage automatique calculent habituellement une corrélation croisée, mais appellent le résultat **convolution**. Avec un noyau de contours asymétrique, le renversement change le signe de la réponse, mais pas sa valeur absolue.

En traitement classique des images, la sortie est souvent appelée **carte de réponse**. Dans un réseau convolutif, la sortie correspondante d'un noyau est une **carte d'attributs**.

Nous n'ajoutons aucun biais ici, car il s'agit d'une opération classique conçue à la main. Une couche convolutive apprise ajoutera un biais appris à chaque carte d'attributs de sortie.

Révéler les contours verticaux

In [5]:
```
photo_path = Path("images/ibm_704.jpeg")
photo = Image.open(photo_path).convert("L")
photo.thumbnail((360, 360))
photo_gray = np.asarray(photo, dtype=float) / 255.0

K_vertical = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1],
], dtype=float)

Y_vertical = cross_correlation2d(photo_gray, K_vertical)
vertical_strength = np.abs(Y_vertical)
vertical_limit = np.percentile(vertical_strength, 99)

fig, axes = plt.subplots(
    1, 3,
    figsize=(11.5, 4.4),
    gridspec_kw={"width_ratios": [1.38, 0.62, 1.38]},
)

axes[0].imshow(photo_gray, cmap="gray", vmin=0, vmax=1)
axes[0].set_title("Entrée en niveaux de gris")
axes[0].axis("off")

draw_number_matrix(
    axes[1], K_vertical, "Noyau",
    cmap="RdBu_r", vmin=-2, vmax=2,
)

axes[2].imshow(
    vertical_strength, cmap="gray", vmin=0, vmax=vertical_limit,
)
axes[2].set_title("Intensité des contours verticaux $|Y|$")
axes[2].axis("off")

fig.tight_layout(w_pad=1.2)
plt.show()
```



Les réponses sont fortes lorsque la luminosité change de gauche à droite.

Il s'agit du noyau de Sobel x. Il estime la vitesse à laquelle l'intensité de l'image change d'une colonne à l'autre, c'est-à-dire selon la coordonnée horizontale. Un changement marqué de gauche à droite se produit à une frontière verticale; K_x est donc couramment décrit comme un détecteur de contours verticaux. Distinguer la direction de la dérivée de l'orientation du contour évite une erreur terminologique fréquente.

La réponse signée distingue une frontière sombre-claire d'une frontière claire-sombre. La diapositive affiche $|Y|$, la valeur absolue de la réponse, parce que nous cherchons ici la position des frontières verticales fortes et non leur polarité. L'échelle est limitée au 99e percentile afin que quelques pixels très intenses ne rendent pas les autres contours invisibles.

Le noyau a été choisi par une personne; il n'y a aucun apprentissage automatique. La photographie fournit une image réelle présentant des variations naturelles; la console, le damier, les vêtements et l'arrière-plan contiennent des frontières de plusieurs orientations.

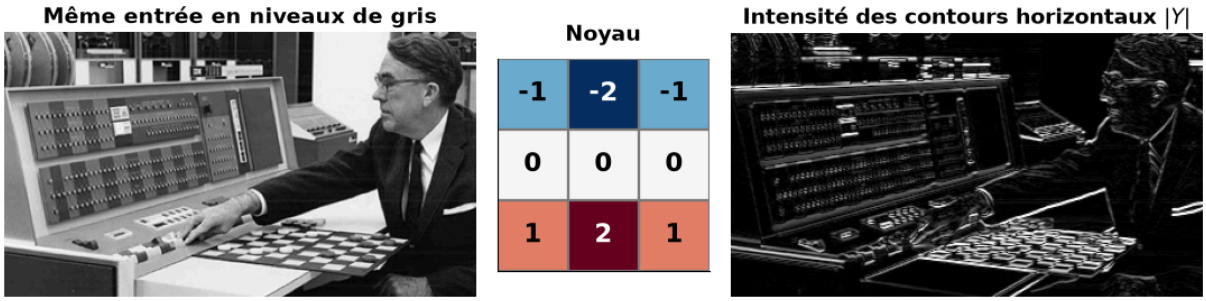
Sources :

- IBM, « The games that helped AI evolve », photographie d'Arthur Samuel devant une console d'ordinateur et un damier. https://www.ibm.com/history/early-games
- Documentation d'OpenCV, « Sobel Derivatives ». https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html

Révéler les contours horizontaux

In [6]:
```
K_horizontal = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1],
], dtype=float)

Y_horizontal = cross_correlation2d(photo_gray, K_horizontal)
horizontal_strength = np.abs(Y_horizontal)
horizontal_limit = np.percentile(horizontal_strength, 99)

fig, axes = plt.subplots(
```

```
    1, 3,
    figsize=(11.5, 4.4),
    gridspec_kw={"width_ratios": [1.38, 0.62, 1.38]},
)

axes[0].imshow(photo_gray, cmap="gray", vmin=0, vmax=1)
axes[0].set_title("Même entrée en niveaux de gris")
axes[0].axis("off")

draw_number_matrix(
    axes[1], K_horizontal, "Noyau",
    cmap="RdBu_r", vmin=-2, vmax=2,
)

axes[2].imshow(
    horizontal_strength, cmap="gray", vmin=0, vmax=horizontal_limit,
)
axes[2].set_title("Intensité des contours horizontaux $|Y|$")
axes[2].axis("off")

fig.tight_layout(w_pad=1.2)
plt.show()
```



Le traitement classique des images utilise des noyaux fixes; les réseaux convolutifs les **apprennent**.

La transposée du noyau de Sobel x donne le noyau de Sobel y. Celui-ci estime les changements d'une ligne à l'autre et répond donc fortement aux frontières horizontales. L'emploi de la même image d'entrée rend la comparaison contrôlée : seuls les poids du noyau ont changé.

Ces noyaux illustrent l'idée calculatoire centrale. Un petit ensemble de poids est appliqué à chaque voisinage local, ce qui produit une carte de réponse indiquant où apparaît le motif recherché. En traitement classique des images, des spécialistes définissent les noyaux utiles. Un réseau de neurones convolutif conserve le balayage local, mais apprend les poids et les biais de ses noyaux à partir des données.

Les noyaux appris dans la première couche peuvent ressembler à des détecteurs de contours ou de contrastes de couleur, mais un réseau convolutif n'est pas limité à la reproduction de noyaux classiques. Ses noyaux sont optimisés en fonction de l'objectif de prédiction du réseau.

Sources :

- IBM, « The games that helped AI evolve », photographie d'Arthur Samuel devant une console d'ordinateur et un damier. https://www.ibm.com/history/early-games
- Documentation d'OpenCV, « Sobel Derivatives ». https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html

Idée calculatoire centrale

> Un **petit ensemble de poids** (noyau ou filtre) est appliqué à chaque **voisinage local**; il produit une **carte de réponse**, ou carte d'attributs, qui indique où le motif recherché apparaît.

Contrairement au traitement classique des images, où les noyaux sont définis manuellement, les **réseaux de neurones convolutifs** apprennent automatiquement leurs **noyaux et leurs biais** pendant l'entraînement.

L'exemple bidimensionnel fournit l'intuition visuelle. Nous revenons maintenant aux signaux unidimensionnels afin d'introduire l'apprentissage des noyaux, les noyaux multiples, les canaux, le pas, le remplissage, le pooling et l'empilement des couches avec moins d'indices. Nous retrouverons ensuite les mêmes idées pour les images en ajoutant un seul indice spatial.

Convolution à une dimension

Où trouve-t-on des données 1D ?

> Dans quels problèmes les observations sont-elles disposées selon **un axe ordonné** ?

$$x[0],\ x[1],\ x[2],\ \ldots,\ x[L-1]$$

L'ordre importe : les valeurs voisines entretiennent généralement une relation significative.

Les données unidimensionnelles sont organisées selon un axe ordonné. Le temps est l'axe le plus familier, mais l'ordre peut aussi représenter une position dans un texte ou une séquence biologique, ou encore la distance le long d'un trajet. Les **observations voisines sont généralement liées**; permuter deux positions modifie donc le sens des données.

Voici quelques exemples familiers :

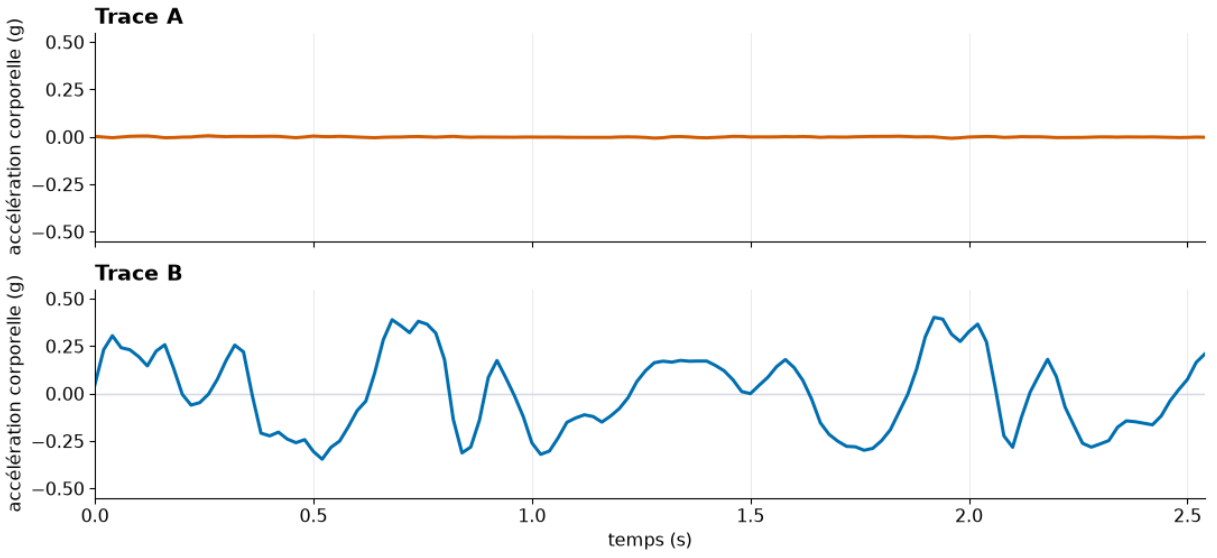
- l'amplitude d'un signal audio au fil du temps ;

- la température, les précipitations, la circulation ou des mesures financières au fil du temps ;
- les signaux d'un accéléromètre, d'un gyroscope ou d'un capteur de vibrations, ainsi que les ECG et les EEG ;
- la consommation d'électricité et d'autres signaux de compteurs intelligents ;
- un texte représenté par des caractères, des jetons ou des plongements ;
- les séquences d'ADN, d'ARN et de protéines ;
- l'altitude ou une autre mesure échantillonnée le long d'un parcours physique.

L'application aux séquences biologiques est développée séparément dans le carnet d'autoapprentissage *Prédire la charge ribosomique moyenne avec un réseau de neurones convolutif 1D*.

Quelle trace correspond à la marche ?

In [7]:
```
fig, axes = plt.subplots(2, 1, figsize=(11.5, 5.4), sharex=True)

for ax, frame, title, color in [
    (axes[0], sitting, "Trace A", ORANGE),
    (axes[1], walking, "Trace B", BLUE),
]:
    ax.plot(time, frame["x"], color=color)
    ax.set_title(title, loc="left")
    style_time_axis(
        ax,
        xlabel=ax is axes[-1],
        ylabel="accélération corporelle (g)",
        ylim=(-0.55, 0.55),
    )

fig.tight_layout(h_pad=1.0)
plt.show()
```



Deux fenêtres d'accéléromètre, d'une durée de 2,56 secondes et échantillonnées à 50 Hz.

La trace A correspond à la position assise et la trace B à la marche. L'échelle verticale commune rend visible la différence d'amplitude des mouvements sans exagérer le signal de la position assise. Il s'agit de fenêtres représentatives dont l'énergie est proche de la médiane de leur activité respective, plutôt que de cas extrêmes choisis pour maximiser la séparation visuelle.

L'expérience originale réunissait 30 volontaires portant un Samsung Galaxy S II à la taille pendant six activités. L'accéléromètre et le gyroscope étaient échantillonnés à 50 Hz. La version 1 du jeu de données publié contient à la fois des fenêtres de signaux inertiels prétraités et un tableau distinct de 561 attributs construits.

Ce prototype n'utilise **pas** le tableau de 561 attributs contenu dans `X_train.txt`. Il utilise les 128 valeurs de chaque ligne du fichier `train/Inertial Signals/body_acc_x_train.txt`. Les signaux tracés ont déjà été filtrés pour réduire le bruit, divisés en fenêtres superposées de 2,56 secondes et séparés de la composante de gravité estimée. Ce sont donc des signaux de capteurs traités, et non des enregistrements bruts intacts.

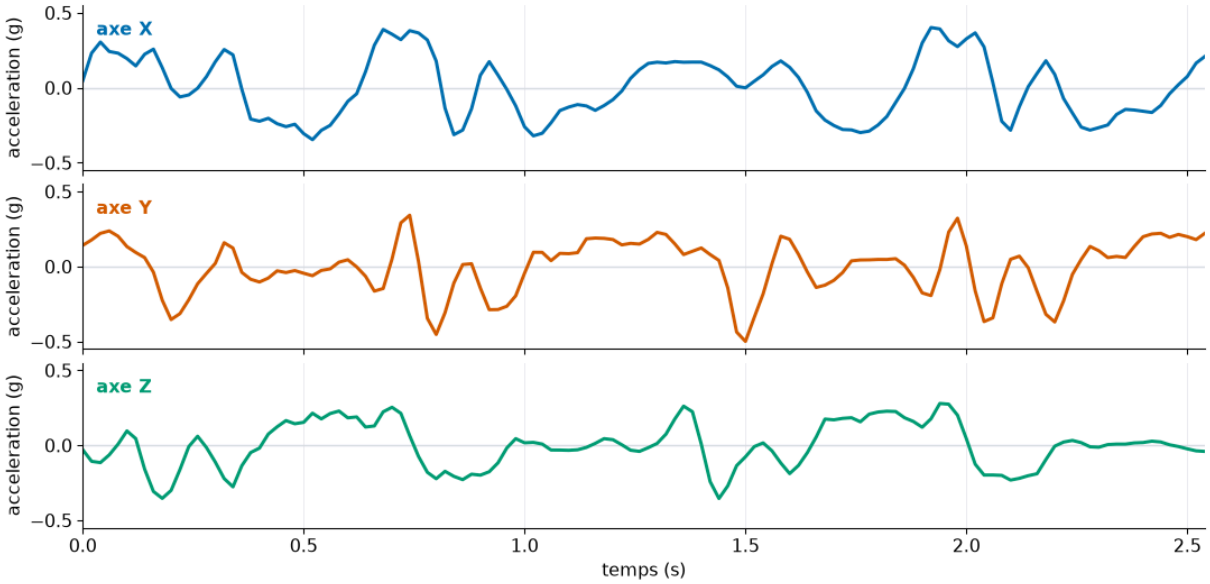
Sources :

- Reyes-Ortiz et al., *Human Activity Recognition Using Smartphones*, UCI Machine Learning Repository, DOI: 10.24432/C54S4K. https://archive.ics.uci.edu/dataset/240/humanactivityrecognitionusingsmartphones
- L'exemple de marche est l'enregistrement 3363 (indice source 3362 en base zéro, sujet 17) et celui de la position assise est l'enregistrement 395 (indice source 394 en base zéro, sujet 3) des fichiers de signaux inertiels d'entraînement.

Trois séquences d'un téléphone

In [8]:
```
fig, axes = plt.subplots(3, 1, figsize=(11.5, 5.7), sharex=True)

for ax, channel, color in zip(axes, ["x", "y", "z"], [BLUE, ORANGE, GREEN]):
    ax.plot(time, walking[channel], color=color)
    ax.text(
        0.012,
        0.82,
        f"axe {channel.upper()}",
        transform=ax.transAxes,
        color=color,
        fontweight="bold",
    )
    style_time_axis(
        ax,
        xlabel=ax is axes[-1],
```

```
        ylabel="acceleration (g)",
        ylim=(-0.55, 0.55),
    )

fig.tight_layout(h_pad=0.45)
plt.show()
```



Le téléphone fournit trois séquences simultanées : l'accélération selon ses axes X, Y et Z. Un axe n'est pas universellement « vertical » ou « horizontal »; son interprétation physique dépend de l'orientation et de la position du téléphone. Les trois canaux partagent néanmoins le même indice temporel, car ils ont été mesurés simultanément.

Le développement à un canal qui suit n'utilise temporairement que la séquence de l'axe X. Nous reviendrons à la représentation à trois canaux après avoir établi l'opération à un canal et sa mise en œuvre.

La documentation du jeu de données renvoie à une courte vidéo de l'expérience de collecte, qui montre la position du téléphone et les six activités enregistrées.

Sources :

- Reyes-Ortiz et al., *Human Activity Recognition Using Smartphones*, UCI Machine Learning Repository, DOI: 10.24432/C54S4K. https://archive.ics.uci.edu/dataset/240/humanactivityrecognitionusingsmartphones
- Vidéo de l'expérience : https://www.youtube.com/watch?v=XOEN9W05_4A

Un signal est une suite de nombres

In [9]:
```
first_values = pd.DataFrame({
    "indice i": walking["sample"].head(8),
    "temps (s)": (walking["sample"].head(8) / SAMPLE_RATE).round(2),
    "x[i] (g)": walking["x"].head(8).round(5),
```

```
})
print(first_values.to_string(index=False))
```

```
indice i  temps (s)  x[i] (g)
       0       0.00   0.04796
       1       0.02   0.23134
       2       0.04   0.30507
       3       0.06   0.24206
       4       0.08   0.23098
       5       0.10   0.19504
       6       0.12   0.14604
       7       0.14   0.22359
```

$$x[0] = 0.04796, \quad x[1] = 0.23134, \quad x[2] = 0.30507, \quad \ldots$$

L'indice indique la position; la valeur est un nombre réel ordinaire.

L'entrée de l'axe X utilisée dans les diapositives suivantes est un vecteur de 128 nombres réels. À raison de 50 mesures par seconde, l'indice progresse de 0,02 seconde à chaque position. La notation $x[i]$ désigne simplement le nombre stocké à la position i.

Le fichier `data/uci_har_prototype.csv` est un extrait compact en format long créé pour ce cours. Chaque ligne contient l'activité, l'indice en base zéro de la fenêtre source, l'identifiant du sujet, l'indice de l'échantillon et les trois valeurs d'accélération corporelle. Il a été dérivé des trois fichiers UCI `body_acc_{x,y,z}_train.txt` ; les étiquettes et les sujets proviennent respectivement de `y_train.txt` et `subject_train.txt` .

« Valeurs réelles » ne signifie pas « enregistrements bruts non traités ». Ces valeurs ont subi le prétraitement décrit dans les notes précédentes. Cette distinction importe pour interpréter ce que reçoit le modèle et les conclusions que l'on peut en tirer.

Sources :

- Les valeurs affichées proviennent de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR.

Quelles sont les hypothèses ?

- Les mesures voisines forment des **motifs locaux** significatifs.
- Un motif utile peut apparaître à **différentes positions**.
- Le même détecteur devrait être **réutilisé partout**.

Ces hypothèses constituent un **biais inductif** architectural.

Un **biais inductif** est un ensemble d'hypothèses qui guide ce qu'un modèle peut apprendre efficacement à partir d'une quantité finie de données. Une couche dense permet à chaque sortie de dépendre de toutes les entrées au moyen de poids

indépendants. Une couche convolutive suppose plutôt que les voisinages locaux importent et qu'une relation locale utile peut réapparaître à plusieurs positions.

La connectivité locale crée des **interactions clairsemées** : une activation de sortie dépend d'une courte fenêtre d'entrée plutôt que de la séquence entière. Le **partage des paramètres** réutilise ensuite les mêmes poids du noyau à chaque position. Ces contraintes **réduisent le nombre de paramètres** et font en sorte que la carte d'attributs réponde de façon cohérente lorsqu'un même motif local apparaît ailleurs.

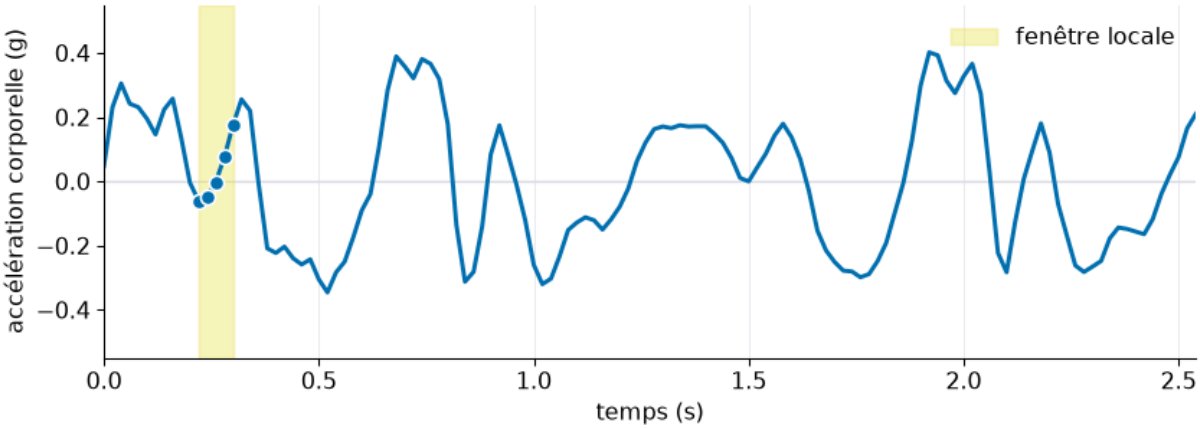
Ces hypothèses ne sont utiles que si elles correspondent au problème. **Le temps et la position spatiale possèdent une structure de voisinage significative**, alors que **des canaux de capteurs distincts ont des significations physiques différentes**. Le noyau se déplace donc selon la dimension de la séquence, mais couvre tous les canaux d'entrée; il ne glisse pas sur une permutation arbitraire des canaux.

L'architecture ne garantit pas l'apprentissage du motif recherché. Elle rend certaines solutions plus faciles à représenter et à apprendre. Le noyau conçu à la main ci-dessous rend ce biais visible avant que nous abordions l'entraînement.

Un noyau évalue un contraste local

In [10]:
```
kernel_up = np.array([-1.0, -1.0, 0.0, 1.0, 1.0])
kernel_down = -kernel_up
K = kernel_up.size

# Cette fenêtre traverse zéro; le noyau compare plutôt les niveaux relatifs.
start = 11
stop = start + K

fig, ax = plt.subplots(figsize=(9.0, 3.4))
ax.plot(time, x_walking, color=BLUE)
ax.axvspan(
    time[start],
    time[stop - 1],
    color=HIGHLIGHT,
    alpha=0.35,
    label="fenêtre locale",
)
ax.scatter(
    time[start:stop],
    x_walking[start:stop],
    color=BLUE,
    edgecolor="white",
    linewidth=1.2,
    s=58,
    zorder=3,
)
style_time_axis(
    ax,
    xlabel=True,
```

```
    ylabel="accélération corporelle (g)",
    ylim=(-0.55, 0.55),
)
ax.legend(loc="upper right")
fig.tight_layout()
plt.show()
```



$$w = [-1, -1, 0, 1, 1], \qquad y[i] = \sum_{j=0}^{K-1} x[i+j]w[j]$$

Le score est élevé lorsque les **deux dernières valeurs dépassent les deux premières**. La valeur centrale reçoit un poids nul.

Le noyau w contient cinq poids; ainsi, $K = 5$. À la position de sortie i, le score vaut

$$y[i] = -x[i] - x[i+1] + x[i+3] + x[i+4].$$

Il compare la somme des deux dernières valeurs à celle des deux premières. La valeur centrale $x[i+2]$ est ignorée. La fenêtre choisie contient deux valeurs négatives, une valeur proche de zéro et deux valeurs positives, ce qui rend la transition ascendante visuellement claire.

Le noyau ne vérifie pas le signe des valeurs d'entrée. Il répond aussi lorsque les cinq valeurs sont positives ou toutes négatives, pourvu que les deux dernières dépassent suffisamment les deux premières. Il est donc plus précis de le décrire comme un détecteur de contraste ascendant que comme un gabarit exact d'une suite particulière de signes.

Le noyau est volontairement conçu à la main afin que son comportement puisse être prédit avant le calcul. Dans un réseau de neurones convolutif, les poids du noyau sont appris à partir des données.

Le traitement classique des signaux et des images emploie le même calcul local de multiplication et de sommation, avec des poids choisis selon l'expertise du domaine. Un réseau convolutif traite ces poids comme des paramètres et apprend des détecteurs

utiles à la tâche de prédiction. Le noyau conçu ici est donc un point de départ explicatif, et non une affirmation sur ce qu'un réseau entraîné doit apprendre.

L'opération représentée est une corrélation croisée, car le noyau n'est pas renversé. Par convention, les bibliothèques d'apprentissage profond appellent cette opération une convolution. Nous pouvons expliciter la distinction avec la convolution mathématique sans modifier le passage avant qui suit.

Sources :

- Le signal tracé provient de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR.

x et y ont-ils la même longueur ?

. . .

In [11]:
```
def conv1d(x, w):

    K = w.size
    L_out = x.size - K + 1
    y = np.zeros(L_out)

    for i in range(L_out):
        for j in range(K):
            y[i] += x[i + j] * w[j]

    return y

up_map = conv1d(x_walking, kernel_up)
```

Non, pas avec cette mise en œuvre. Cette fonction effectue une corrélation croisée unidimensionnelle valide avec un canal d'entrée, un noyau, un pas unitaire, aucun remplissage et aucun biais. Elle constitue déjà un passage avant complet pour ce cas restreint.

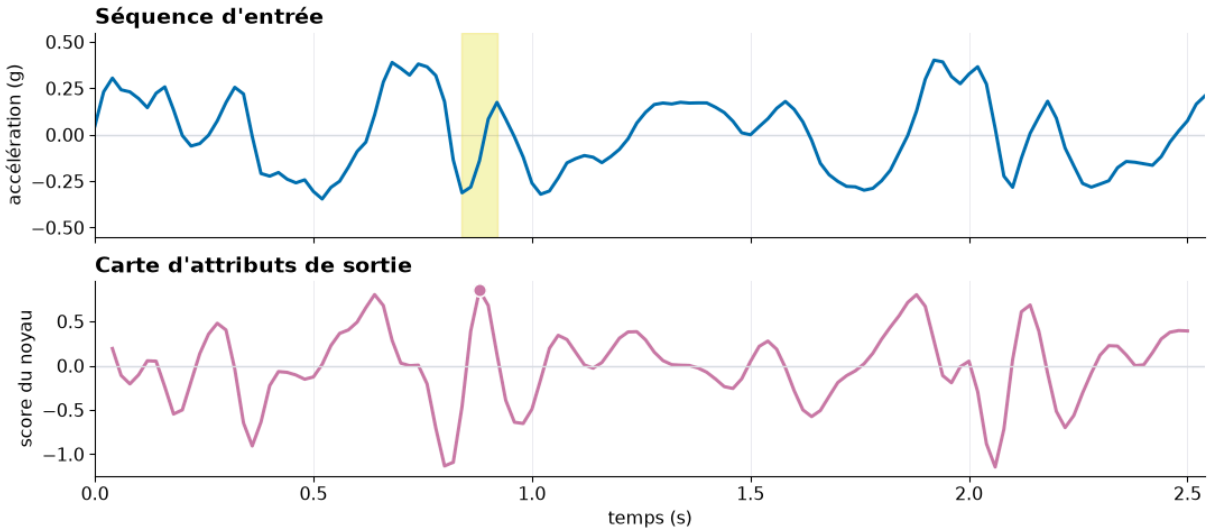
La variable de boucle i désigne une position de sortie et j une position dans le noyau. L'instruction de la boucle interne calcule un produit et l'ajoute à `y[i]`. Le code source expose ainsi la même sommation que l'équation au lieu de la dissimuler dans une opération vectorisée.

Le même vecteur `w` est utilisé à chaque position. Ce partage des paramètres permet de détecter un motif local où qu'il apparaisse dans la séquence. Pour $L = 128$ et $K = 5$, la corrélation croisée valide produit $L_{\text{out}} = 124$ scores.

Poids partagés → carte de sortie

In [12]:

```
feature_time = (np.arange(up_map.size) + (K - 1) / 2) / SAMPLE_RATE

best_up = int(np.argmax(up_map))

fig, axes = plt.subplots(
    2,
    1,
    figsize=(11.5, 5.2),
    sharex=True,
    gridspec_kw={"height_ratios": [1.05, 1]},
)

axes[0].plot(time, x_walking, color=BLUE)
axes[0].axvspan(
    time[best_up],
    time[best_up + K - 1],
    color=HIGHLIGHT,
    alpha=0.35,
)
axes[0].set_title("Séquence d'entrée", loc="left")
style_time_axis(
    axes[0],
    ylabel="accélération (g)",
    ylim=(-0.55, 0.55),
)

axes[1].plot(feature_time, up_map, color=PURPLE)
axes[1].scatter(
    feature_time[best_up],
    up_map[best_up],
    color=PURPLE,
    edgecolor="white",
    linewidth=1.2,
    s=75,
    zorder=3,
)
axes[1].set_title("Carte d'attributs de sortie", loc="left")
axes[1].axhline(0, color=LIGHT_GRAY, linewidth=1)
axes[1].grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
axes[1].set_xlim(time[0], time[-1])
axes[1].set_ylabel("score du noyau")
axes[1].set_xlabel("temps (s)")

fig.tight_layout(h_pad=0.8)
plt.show()
```



La fenêtre d'entrée surlignée produit la valeur de sortie marquée. Le déplacement de la fenêtre sur toutes les positions valides remplit le vecteur de sortie. Dans un réseau convolutif, cette sortie est appelée **carte d'attributs** : chaque position indique avec quelle intensité le motif local appris y apparaît.

Nous observons à la fois la **connectivité locale** et le **partage des paramètres**. Chaque valeur de sortie n'utilise que cinq valeurs d'entrée voisines, mais les mêmes cinq poids sont réutilisés aux 124 positions valides. À titre de comparaison, une transformation Dense de 128 valeurs d'entrée en 124 valeurs de sortie contiendrait $128 \times 124 = 15\,872$ poids distincts. Cette convolution ne contient que cinq poids, ou six paramètres appris si l'on inclut un biais partagé.

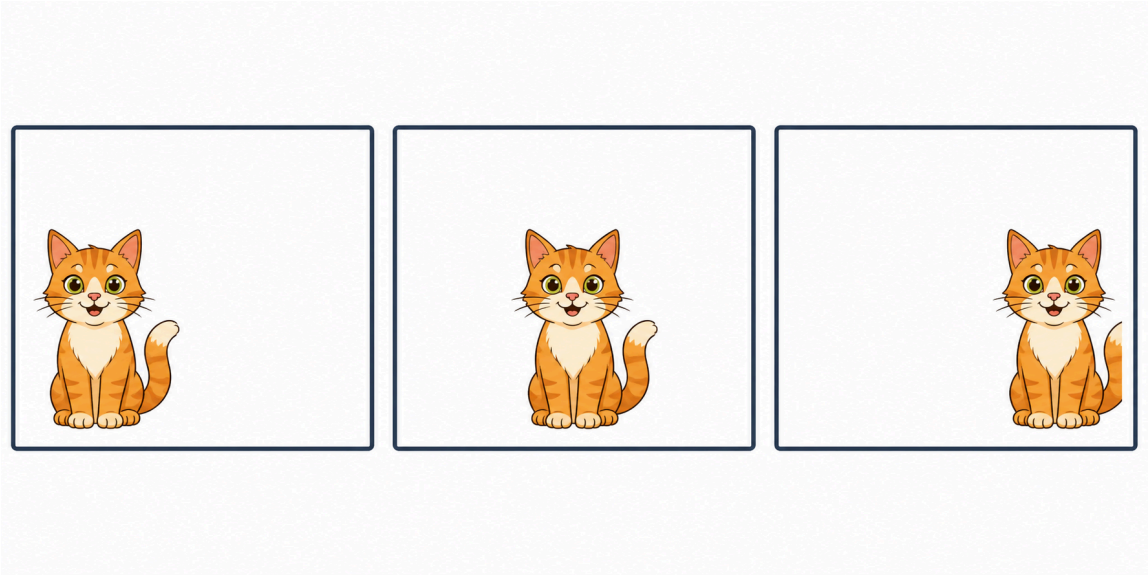
Cette réduction ne vient pas de l'élimination de positions d'entrée. Elle vient de l'obligation d'appliquer partout la même règle locale. Cette contrainte architecturale explique à la fois l'efficacité paramétrique et l'équivariance par translation.

Chaque score est placé horizontalement au centre de sa fenêtre d'entrée de cinq échantillons. Ce choix facilite la visualisation, même si le tableau lui-même est simplement indexé de `up_map[0]` à `up_map[123]`.

Sources :

- Le signal tracé provient de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR.

La position change-t-elle la réponse ?



La position change; l'énoncé « cette image contient un chat » ne change pas.

Les trois cadres contiennent le même objet à trois positions horizontales. Un classifieur devrait idéalement produire la même décision « chat présent » dans les trois cas. Cette propriété souhaitée de la décision finale est l'invariance par translation : déplacer l'objet ne change pas la catégorie attribuée à l'image.

Une carte d'attributs convolutive se comporte autrement. Lorsque le chat se déplace, la forte activation associée au chat devrait se déplacer avec lui. La représentation interne est donc équivariante, et non invariante, par translation. Distinguer ces deux propriétés évite l'affirmation imprécise selon laquelle la convolution rendrait à elle seule un réseau invariant par translation.

Même si l'illustration est bidimensionnelle, la même distinction s'applique à la séquence de l'accéléromètre. Le déplacement temporel d'un motif associé à la marche devrait déplacer sa réponse dans la carte d'attributs, tandis que la décision portant sur la fenêtre entière pourrait demeurer inchangée.

Sources :

- L'illustration du chat a été générée pour ce cours avec le générateur d'images d'OpenAI.

Équivariance

Soit T_Δ une translation de l'entrée de Δ positions.

$$F(T_\Delta X) = T_\Delta F(X)$$

Une carte d'attributs convolutive se déplace avec le motif d'entrée.

T_Δ est un opérateur de translation : il décale chaque position de Δ. Pour une convolution ou une corrélation croisée de pas unitaire sur une séquence infinie, une entrée translatée produit une carte d'attributs translatée de la même manière. Il s'agit de l'équivariance par translation, exprimée par $F(T_\Delta X) = T_\Delta F(X)$.

L'équation décrit des propriétés idéalisées. Les frontières finies et le remplissage peuvent perturber l'équivariance exacte près des extrémités. Une convolution de pas S n'est exactement équivariante que pour les translations compatibles avec sa grille d'échantillonnage, par exemple les décalages multiples de S.

Comment conserver la longueur ?

...

In [13]:
```python
def conv1d(x, w, padding=(0, 0)):

    P_left, P_right = padding
    x_pad = np.pad(x, (P_left, P_right), mode="constant")
    K = w.size
    L_out = x_pad.size - K + 1
    y = np.zeros(L_out)

    for i in range(L_out):
        for j in range(K):
            y[i] += x_pad[i + j] * w[j]

    return y

up_map_same = conv1d(x_walking, kernel_up, padding=(2, 2))
```

$$L_{\text{out}} = L + P_{\text{left}} + P_{\text{right}} - K + 1$$

Le remplissage prolonge l'entrée avant le déplacement du noyau. Ici, `padding=(2, 2)` ajoute deux zéros au début et deux à la fin. Comme le noyau est de longueur impaire $K = 5$, ce remplissage équilibré place une sortie à chaque position originale et préserve la longueur.

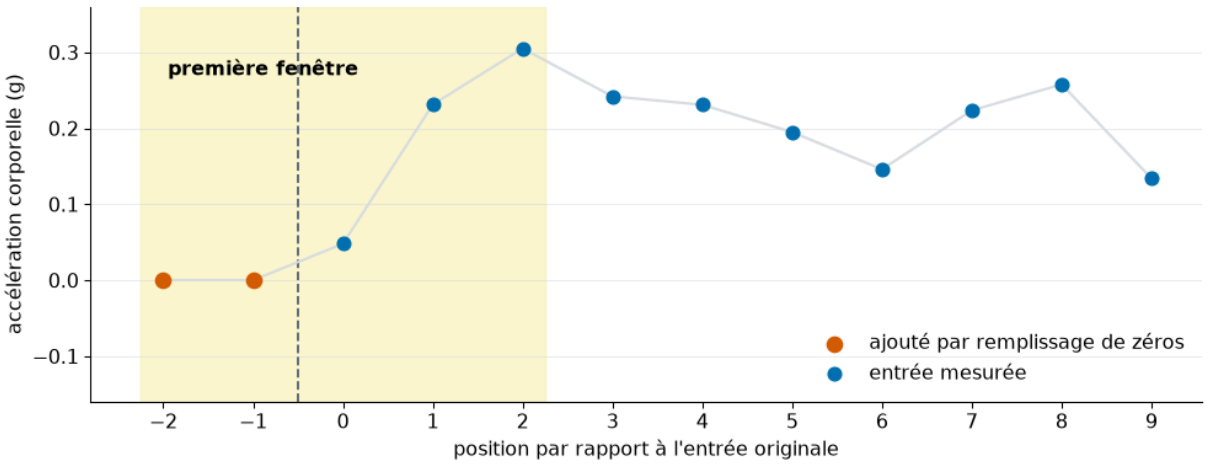
Numériquement, $L = 128$, $K = 5$ et deux zéros ajoutés de chaque côté donnent $L_{\text{out}} = 128 + 2 + 2 - 5 + 1 = 128$.

Le remplissage n'a pas à être équilibré. Ajouter des valeurs uniquement au début ou à la fin modifie l'alignement entre les positions d'entrée et la sortie. Le remplissage « same » consiste généralement à choisir la quantité totale requise pour préserver la longueur; avec un noyau de longueur impaire et un pas unitaire, le répartir également des deux côtés est le choix le plus naturel.

La mise en œuvre redéfinit maintenant `conv1d` en ajoutant un argument facultatif. Sa valeur par défaut `(0, 0)` redonne la fonction de corrélation croisée valide précédente.

Valeurs non mesurées

In [14]:
```
P = 2
shown = 10
padded_index = np.arange(-P, shown)
padded_values = np.concatenate([np.zeros(P), x_walking[:shown]])

fig, ax = plt.subplots(figsize=(10.5, 4.2))
ax.axvspan(-P - 0.25, 2.25, color=HIGHLIGHT, alpha=0.25)
ax.axvline(-0.5, color=GRAY, linestyle="--", linewidth=1.4)
ax.plot(padded_index, padded_values, color=LIGHT_GRAY, linewidth=1.5)
ax.scatter(
    padded_index[:P],
    padded_values[:P],
    color=ORANGE,
    s=85,
    label="ajouté par remplissage de zéros",
    zorder=3,
)
ax.scatter(
    padded_index[P:],
    padded_values[P:],
    color=BLUE,
    s=65,
    label="entrée mesurée",
    zorder=3,
)
ax.text(-1.95, 0.27, "première fenêtre", fontweight="bold")
ax.set_xticks(padded_index)
ax.set_xlabel("position par rapport à l'entrée originale")
ax.set_ylabel("accélération corporelle (g)")
ax.set_ylim(-0.16, 0.36)
ax.grid(axis="y", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
ax.legend(loc="lower right")
fig.tight_layout()
plt.show()
```



Le remplissage par zéros, réflexion, réplication ou périodicité encode différentes hypothèses aux frontières. Les sorties près des frontières peuvent contenir des artefacts.

Le remplissage crée des valeurs qui n'ont pas été mesurées. À la première position du noyau, deux des cinq valeurs sont des zéros artificiels. La sortie dépend donc en partie de la convention choisie pour les frontières, et non seulement des données observées.

Le remplissage par zéros suppose que les valeurs à l'extérieur de l'intervalle observé sont nulles. Le remplissage par réflexion reproduit en miroir les valeurs près de la frontière; le remplissage par réplication répète la valeur extrême; le remplissage circulaire relie la fin de la séquence à son début. Chaque choix convient à certains problèmes et en déforme d'autres.

Dans une couche, l'effet est localisé près de la frontière, mais il peut se propager dans un réseau profond. Le remplissage est donc une décision de modélisation, et non une simple astuce de gestion des dimensions.

Sources :

- Les valeurs mesurées proviennent de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR. Les zéros orange ont été ajoutés pour l'illustration.

Que se passe-t-il si i avance de 2 ?

...

In [15]:
```python
def conv1d(x, w, padding=(0, 0), stride=1):

    P_left, P_right = padding
    x_pad = np.pad(x, (P_left, P_right), mode="constant")
    K = w.size
    starts = range(0, x_pad.size - K + 1, stride)
    y = np.zeros(len(starts))

    for t, i in enumerate(starts):
        for j in range(K):
            y[t] += x_pad[i + j] * w[j]

    return y
```

...

$$L_{\text{out}} = \left\lfloor \frac{L + P_{\text{left}} + P_{\text{right}} - K}{S} \right\rfloor + 1$$

Le pas S est la distance entre deux positions successives du noyau. Avec `stride=1`, le début i de la fenêtre d'entrée parcourt $0, 1, 2, \ldots$. Avec `stride=2`, il parcourt $0, 2, 4, \ldots$; la moitié des positions possibles sont donc omises dans cet exemple.

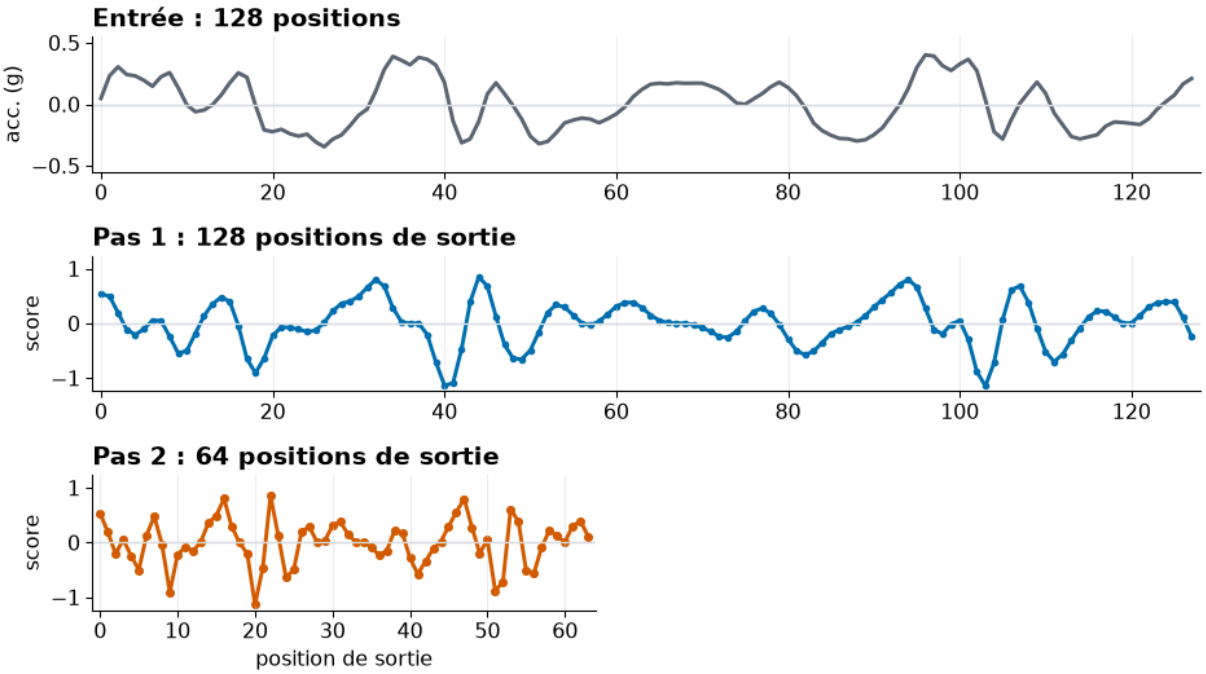
L'indice de sortie t demeure consécutif : $0, 1, 2, \ldots$. Le début de la fenêtre d'entrée servant à le calculer est $i = tS$. Distinguer t de i rend explicite la relation entre la sortie raccourcie et l'entrée originale.

Le plancher de la formule de longueur élimine toute dernière fenêtre incomplète. Pour $L = 128$, $K = 5$, un remplissage équilibré $(2, 2)$ et $S = 2$, la sortie contient 64 positions.

Quel effet a un pas de 2 ?

In [16]:
```
stride1_map = conv1d(x_walking, kernel_up, padding=(2, 2), stride=1)
stride2_map = conv1d(x_walking, kernel_up, padding=(2, 2), stride=2)

input_position = np.arange(x_walking.size)
stride1_position = np.arange(stride1_map.size)
stride2_position = np.arange(stride2_map.size)
score_limit = 1.08 * max(np.abs(stride1_map).max(), np.abs(stride2_map).max(

fig = plt.figure(figsize=(11.5, 6.0))
grid = fig.add_gridspec(3, 2, width_ratios=[1, 1], hspace=0.62)
input_ax = fig.add_subplot(grid[0, :])
stride1_ax = fig.add_subplot(grid[1, :])
stride2_ax = fig.add_subplot(grid[2, 0])
empty_ax = fig.add_subplot(grid[2, 1])
empty_ax.axis("off")

input_ax.plot(input_position, x_walking, color=GRAY)
input_ax.set_title("Entrée : 128 positions", loc="left")
input_ax.set_ylabel("acc. (g)")
input_ax.set_ylim(-0.55, 0.55)
input_ax.set_xlim(-1, 128)

stride1_ax.plot(stride1_position, stride1_map, color=BLUE)
stride1_ax.scatter(stride1_position, stride1_map, color=BLUE, s=10)
stride1_ax.set_title("Pas 1 : 128 positions de sortie", loc="left")
stride1_ax.set_ylabel("score")
stride1_ax.set_ylim(-score_limit, score_limit)
stride1_ax.set_xlim(-1, 128)

stride2_ax.plot(stride2_position, stride2_map, color=ORANGE)
stride2_ax.scatter(stride2_position, stride2_map, color=ORANGE, s=18)
stride2_ax.set_title("Pas 2 : 64 positions de sortie", loc="left")
stride2_ax.set_ylabel("score")
stride2_ax.set_xlabel("position de sortie")
stride2_ax.set_ylim(-score_limit, score_limit)
stride2_ax.set_xlim(-1, 64)

for ax in [input_ax, stride1_ax, stride2_ax]:
```

```
    ax.axhline(0, color=LIGHT_GRAY, linewidth=1)
    ax.grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)

fig.tight_layout()
plt.show()
```



Un pas de 2 produit **deux fois moins de positions**.

Le remplissage a été choisi de sorte qu'un pas de 1 produise un score aligné sur chaque position d'entrée. Un pas de 2 utilise les mêmes poids et la même entrée remplie, mais n'évalue qu'une position sur deux. Le résultat contient 64 scores plutôt que 128. Dans cette représentation centrée sur les dimensions, le tableau de 64 valeurs est volontairement tracé avec la moitié de la largeur physique des tableaux de 128 valeurs.

Si les deux sorties étaient tracées en fonction du temps écoulé, elles couvriraient le même intervalle de 2,56 secondes; la sortie de pas 2 contiendrait simplement deux fois moins d'échantillons. Tracer l'indice de sortie rend plutôt le changement de longueur immédiatement visible.

Il s'agit d'un sous-échantillonnage temporel. Il réduit le stockage et les calculs des couches suivantes, mais diminue aussi la résolution de position et peut manquer des changements entre les positions échantillonnées. Dire que la sortie est « la moitié de x » serait trompeur : il s'agit d'un résumé local appris comportant deux fois moins de positions, et non d'un vecteur contenant une valeur sur deux de l'entrée originale.

Sources :

- Le signal d'entrée provient de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR. Les cartes d'attributs ont été calculées par le code affiché.

Noyaux et canaux multiples

Plusieurs motifs locaux

Contraste ascendant

$$w^{(0)} = [-1, -1, 0, 1, 1]$$

La première paire est soustraite de la dernière.

Contraste descendant

$$w^{(1)} = [1, 1, 0, -1, -1]$$

La dernière paire est soustraite de la première.

$$W = \begin{bmatrix} | & | \\ w^{(0)} & w^{(1)} \\ | & | \end{bmatrix} \in \mathbb{R}^{K \times C_{\text{out}}}$$

Comment modéliser plusieurs motifs locaux ?

Le premier noyau produit un score positif lorsque la dernière paire dépasse la première. Le second est son opposé et produit donc un score positif pour le contraste inverse.

Cette paire symétrique est utile sur le plan pédagogique, car la relation entre les sorties est prévisible.

Placer les deux vecteurs de longueur K en colonnes produit une matrice W de forme $K \times C_{\text{out}}$. Ici, $C_{\text{out}} = 2$ puisque deux noyaux produisent deux **canaux de sortie**. Un noyau produit une carte d'attributs; ajouter des noyaux permet donc à une couche de représenter plusieurs types d'indices locaux.

L'idée qu'un noyau appris corresponde à un motif facile à nommer constitue un premier modèle utile, mais pas une garantie. Les noyaux appris ne sont pas nécessairement opposés l'un à l'autre, et leur représentation conjointe peut être plus significative qu'un noyau considéré isolément.

Combien de noyaux une couche devrait-elle utiliser ?

- L'**expertise du domaine** peut suggérer un petit ensemble interprétable.
- Plus souvent, C_{out} est un **hyperparamètre** choisi à l'aide des données de validation.
- **Davantage de noyaux** augmentent la capacité de représentation, le nombre de paramètres, la mémoire requise et les calculs.

Il n'existe aucun nombre universellement correct de noyaux. Dans un système conçu à la main, la connaissance du domaine peut suggérer un petit ensemble de contrastes, de fréquences ou de motifs. Dans un réseau convolutif appris, on choisit généralement un nombre candidat de canaux de sortie, on entraîne le modèle, puis on l'évalue sur des données de validation.

Un nombre insuffisant de noyaux peut créer un goulot d'étranglement de représentation. Un nombre excessif peut gaspiller des ressources et favoriser le surapprentissage lorsque les données sont limitées. Le choix tient donc compte de la performance prédictive, mais aussi du temps d'entraînement, de la mémoire, de la latence d'inférence et de la taille du modèle.

La boucle acquiert un indice de noyau

In [17]:
```
def conv1d(x, W, b, padding=(0, 0), stride=1):

    P_left, P_right = padding
    x_pad = np.pad(x, (P_left, P_right), mode="constant")
    K, C_out = W.shape
    starts = range(0, x_pad.size - K + 1, stride)
    Y = np.zeros((len(starts), C_out))

    for t, i in enumerate(starts):
        for m in range(C_out):
            Y[t, m] = b[m]
            for j in range(K):
                Y[t, m] += x_pad[i + j] * W[j, m]

    return Y
```

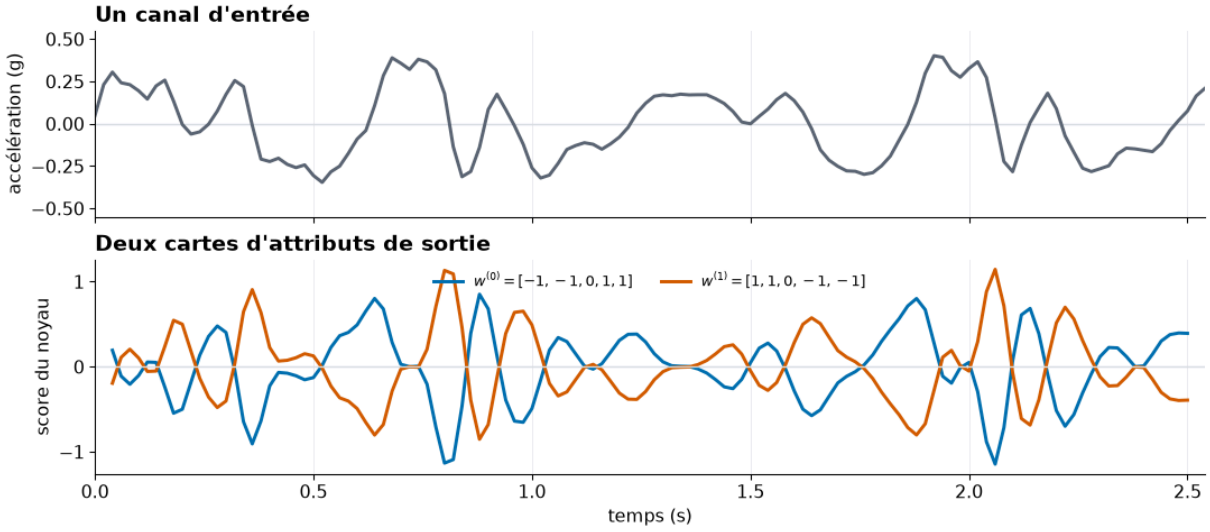
La fonction révisée ajoute l'indice de noyau m. `W[j, m]` sélectionne la position j du noyau m, `b[m]` son biais et `Y[:, m]` la carte d'attributs correspondante. Initialiser `Y[t, m]` avec le biais, puis y accumuler les K produits reproduit directement l'équation affichée. Les arguments de remplissage et de pas demeurent disponibles; la mise en œuvre s'enrichit donc en ajoutant successivement les concepts.

L'entrée reste unidimensionnelle, de forme (L). La matrice des noyaux a la forme (K, C_{out}) et la sortie, la forme $(L_{\text{out}}, C_{\text{out}})$. L'exemple utilise des biais nuls, aucun remplissage et un pas unitaire afin de comparer directement les deux cartes à la carte valide précédente.

Un noyau → une carte de sortie

In [18]:
```
kernels = np.column_stack([kernel_up, kernel_down])
bias = np.zeros(2)
feature_maps = conv1d(x_walking, kernels, bias)
feature_time = (np.arange(feature_maps.shape[0]) + (K - 1) / 2) / SAMPLE_RAT
```

```
fig, axes = plt.subplots(
    2,
    1,
    figsize=(11.5, 5.2),
    sharex=True,
    gridspec_kw={"height_ratios": [1, 1.15]},
)

axes[0].plot(time, x_walking, color=GRAY)
axes[0].set_title("Un canal d'entrée", loc="left")
style_time_axis(
    axes[0],
    ylabel="accélération (g)",
    ylim=(-0.55, 0.55),
)

axes[1].plot(
    feature_time,
    feature_maps[:, 0],
    color=BLUE,
    label=r"$w^{(0)}=[-1,-1,0,1,1]$",
)
axes[1].plot(
    feature_time,
    feature_maps[:, 1],
    color=ORANGE,
    label=r"$w^{(1)}=[1,1,0,-1,-1]$",
)
axes[1].axhline(0, color=LIGHT_GRAY, linewidth=1)
axes[1].grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
axes[1].set_xlim(time[0], time[-1])
axes[1].set_title("Deux cartes d'attributs de sortie", loc="left")
axes[1].set_ylabel("score du noyau")
axes[1].set_xlabel("temps (s)")
axes[1].legend(loc="upper center", ncol=2, fontsize=9.5)

fig.tight_layout(h_pad=0.7)
plt.show()
```



Les deux cartes d'attributs sont tracées sur les mêmes axes pour rendre leur relation visible. Comme le second noyau est exactement l'opposé du premier, sa carte est aussi l'opposé de la première. Une transition ascendante qui produit un score bleu positif produit un score orange négatif de même amplitude; une transition descendante inverse ces signes.

Les valeurs explicites des noyaux demeurent dans la légende. Les sorties restent ainsi reliées au calcul plutôt que de paraître comme des courbes inexpliquées.

Sources :

- Le signal tracé provient de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR.

Formes et signification des objets

Objet	Signification	Forme
X	séquence d'entrée	(L, C_{in})
W	poids des noyaux	$(K, C_{\text{in}}, C_{\text{out}})$
b	un biais par canal de sortie	(C_{out})
Y	cartes d'attributs de sortie	$(L_{\text{out}}, C_{\text{out}})$

$$Y[t, m] = b[m] + \sum_{j=0}^{K-1} \sum_{c=0}^{C_{\text{in}}-1} \widetilde{X}[tS + j, c] W[j, c, m]$$

Les symboles décrivent à la fois la taille des tableaux et la signification de chaque indice :

- L est le nombre de positions d'entrée et t une position de sortie ;
- K est la longueur du noyau et j une position dans celui-ci ;
- C_{in} est le nombre de canaux d'entrée et c en sélectionne un ;
- C_{out} est le nombre de noyaux et m sélectionne une carte d'attributs de sortie ;
- S est le pas; ainsi, $i = tS$ est le début correspondant du noyau dans l'entrée remplie $\widetilde{X}$.

Avec P_{left} et P_{right} positions de remplissage,

$$L_{\text{out}} = \left\lfloor \frac{L + P_{\text{left}} + P_{\text{right}} - K}{S} \right\rfloor + 1.$$

L'équation forme tous les produits entre la fenêtre d'entrée de taille $K \times C_{\text{in}}$ et les poids correspondants du noyau m, additionne ces produits, puis ajoute le biais $b[m]$. Le tilde distingue l'entrée remplie de l'entrée originale. En l'absence de remplissage et lorsque $S = 1$, l'équation se réduit à l'expression de corrélation valide précédente.

Les minuscules x, w et y désignaient le cas particulier à un canal et un noyau. Les majuscules X, W et Y désignent les tableaux généraux. Il s'agit de conventions de notation, et non d'opérations mathématiques différentes.

Une fenêtre couvre tous les canaux d'entrée

In [19]:
```python
def conv1d(X, W, b, padding=(0, 0), stride=1):

    P_left, P_right = padding
    X_pad = np.pad(X, ((P_left, P_right), (0, 0)), mode="constant")
    K, C_in, C_out = W.shape
    starts = range(0, X_pad.shape[0] - K + 1, stride)
    Y = np.zeros((len(starts), C_out))

    for t, i in enumerate(starts):
        for m in range(C_out):
            Y[t, m] = b[m]
            for j in range(K):
                for c in range(C_in):
                    Y[t, m] += X_pad[i + j, c] * W[j, c, m]

    return Y
```

Le nouvel indice de boucle c sélectionne un canal d'entrée. Pour une position de sortie t et un noyau m fixés, les deux boucles internes parcourent chaque position j et chaque canal c de la fenêtre $K \times C_{\text{in}}$. Chaque produit est ajouté au même scalaire `Y[t, m]`, initialisé avec le biais $b[m]$.

Une expression comme `np.sum(patch * W[:, :, m])` est plus courte, mais elle masque les deux sommations distinctes qui constituent l'idée nouvelle de cette diapositive. La mise en œuvre est donc volontairement littérale. Une bibliothèque vectoriserait ces boucles et traiterait des lots; ces améliorations d'ingénierie ne modifient pas le passage avant mathématique représenté ici.

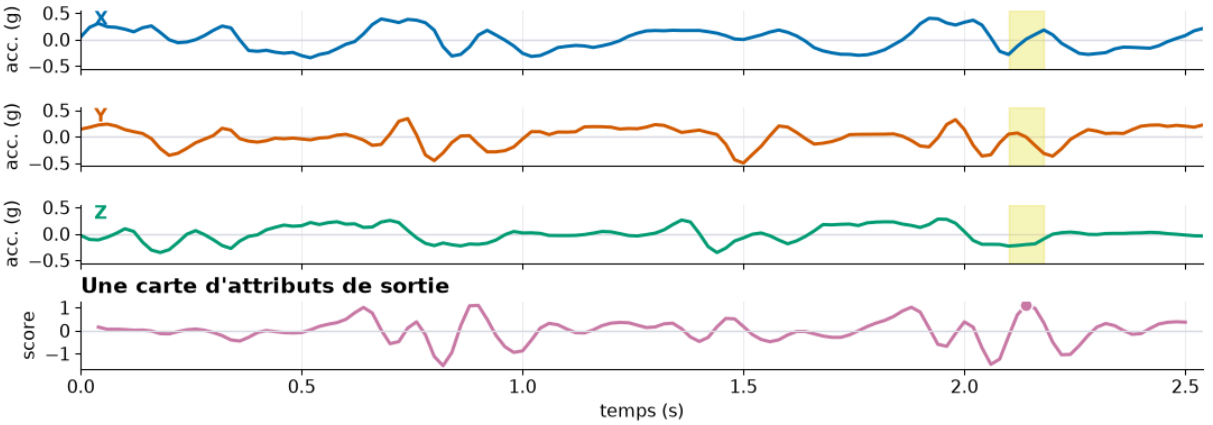
Le carnet redéfinit successivement `conv1d`. Son exécution du début à la fin reproduit donc la progression conceptuelle : chaque définition remplace le cas restreint précédent par un cas plus général.

Un noyau couvre tous les canaux d'entrée

In [20]:

```
X = walking.loc[:, ["x", "y", "z"]].to_numpy()

# Un noyau illustratif : cinq positions, trois canaux, une carte de sortie.
W = np.stack(
    [kernel_up, -0.6 * kernel_up, 0.4 * kernel_up],
    axis=1,
)[:, :, np.newaxis]
b = np.zeros(1)
Y = conv1d(X, W, b, padding=(0, 0), stride=1)

multi_map = Y[:, 0]
multi_feature_time = (
    np.arange(multi_map.size) + (K - 1) / 2
) / SAMPLE_RATE
best_multi = int(np.argmax(multi_map))
start = best_multi
stop = start + K

fig, axes = plt.subplots(
    4,
    1,
    figsize=(11.5, 4.2),
    sharex=True,
    gridspec_kw={"height_ratios": [1, 1, 1, 1.15]},
)

for ax, channel, color in zip(
    axes[:3], ["x", "y", "z"], [BLUE, ORANGE, GREEN]
):
    ax.plot(time, walking[channel], color=color)
    ax.axvspan(
        time[start],
        time[stop - 1],
        color=HIGHLIGHT,
        alpha=0.35,
    )
    ax.text(
        0.012,
        0.76,
        channel.upper(),
        transform=ax.transAxes,
        color=color,
        fontweight="bold",
    )
    style_time_axis(ax, ylabel="acc. (g)", ylim=(-0.55, 0.55))

axes[3].plot(multi_feature_time, multi_map, color=PURPLE)
axes[3].scatter(
    multi_feature_time[best_multi],
    multi_map[best_multi],
    color=PURPLE,
    edgecolor="white",
    linewidth=1.2,
    s=70,
    zorder=3,
```

```
)
axes[3].axhline(0, color=LIGHT_GRAY, linewidth=1)
axes[3].grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
axes[3].set_xlim(time[0], time[-1])
axes[3].set_ylabel("score")
axes[3].set_xlabel("temps (s)")
axes[3].set_title("Une carte d'attributs de sortie", loc="left")

fig.tight_layout(h_pad=0.35)
plt.show()
```



La bande jaune est alignée sur les trois canaux, car ceux-ci sont des mesures simultanées. Pour une position de sortie, la fenêtre d'entrée surlignée a la forme $K \times C_{\text{in}} = 5 \times 3$. Le noyau 0 contient un vecteur de cinq poids pour chaque canal, et les 15 produits contribuent à un seul score de sortie.

Le noyau illustratif utilise $w^{(0)}$ sur X, $-0.6w^{(0)}$ sur Y et $0.4w^{(0)}$ sur Z. Il sert à concrétiser la dimension des canaux, et non à prétendre que ces poids choisis à la main classifient correctement la marche. Un réseau entraîné apprendrait toutes ces valeurs à partir d'exemples étiquetés.

Sources :

- Les signaux tracés proviennent de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR.

Empilement et pooling

Que représentent X et Y ?

$$X \xrightarrow{\text{convolution}+b} Z \xrightarrow{g} Y$$

$$X^{(\ell+1)} = Y^{(\ell)}$$

- X : entrée d'une couche ;
- Z : scores de préactivation ;

- $Y = g(Z)$: activations de sortie ;
- les canaux de $Y^{(\ell)}$ deviennent les canaux d'entrée de la couche suivante.

Les symboles X et Y désignent l'entrée et la sortie d'une seule couche, et non l'entrée et la prédiction finale du réseau entier. Z représente le résultat de la convolution et de l'ajout du biais avant l'application de la fonction d'activation g. Un réseau compose les couches séquentiellement : le tableau complet des activations de sortie de la couche ℓ devient le tableau complet d'entrée de la couche $\ell + 1$.

Dans la première couche convolutive, $X^{(0)}$ contient les canaux du signal mesuré, par exemple les axes X, Y et Z de l'accéléromètre. Dans une couche ultérieure, les canaux d'entrée ne sont plus des axes bruts du capteur; ce sont les cartes d'attributs produites par la couche précédente. Ainsi, $C_{\text{in}}^{(\ell+1)} = C_{\text{out}}^{(\ell)}$.

La fonction `conv1d` affichée renvoie Z, les scores de préactivation. L'exemple suivant utilise $g(z) = \mathrm{ReLU}(z) = \max(0, z)$. ReLU s'applique élément par élément; elle modifie donc les valeurs sans changer la longueur de la séquence ni le nombre de canaux.

Les diapositives précédentes utilisaient directement Y pour les scores de convolution, car aucune non-linéarité n'avait encore été introduite; il s'agit du cas particulier $g(z) = z$. Dorénavant, Z et Y rendent cette distinction explicite.

Combiner des cartes d'attributs

In [21]:
```python
def relu(Z):
    return np.maximum(Z, 0)

X0 = x_walking[:, np.newaxis]
W1 = kernels[:, np.newaxis, :]
b1 = np.zeros(2)
Z1 = conv1d(X0, W1, b1)
Y1 = relu(Z1)

W2 = np.zeros((K, 2, 1))

W2[:2, 0, 0] = 0.5     # indice ascendant plus tôt
W2[-2:, 1, 0] = 0.5    # indice descendant plus tard

b2 = np.array([-0.8])
Z2 = conv1d(Y1, W2, b2)
Y2 = relu(Z2)
```

La première couche applique les noyaux de contraste ascendant et descendant déjà introduits. Sa sortie $Z^{(1)}$ contient des scores signés. ReLU remplace les scores négatifs par zéro et produit les activations non négatives $Y^{(1)}$ de deux canaux : l'un pour les indices ascendants, l'autre pour les indices descendants.

Le noyau de la deuxième couche couvre les deux canaux. Ses poids positifs aux deux premières positions du canal 0 favorisent un indice ascendant plus tôt. Ses poids positifs aux deux dernières positions du canal 1 favorisent un indice descendant plus tard. Il répond donc à une séquence d'attributs appris, et non directement aux valeurs brutes d'accélération.

Il s'agit d'une composition hiérarchique des attributs : la couche 1 décrit des contrastes locaux, alors que la couche 2 décrit un motif composé de ces contrastes. Les couches profondes apprennent ainsi des « motifs de motifs ». Lors de l'entraînement, ces représentations intermédiaires sont choisies selon leur utilité pour la prédiction finale plutôt que définies manuellement comme une étape indépendante de construction des attributs.

La hiérarchie vient de la composition de couches apprises et de non-linéarités. Le pooling ou le pas peuvent modifier la résolution et permettre aux couches ultérieures de couvrir une plus grande portion de l'entrée originale, mais le sous-échantillonnage ne crée pas à lui seul une hiérarchie d'attributs appris.

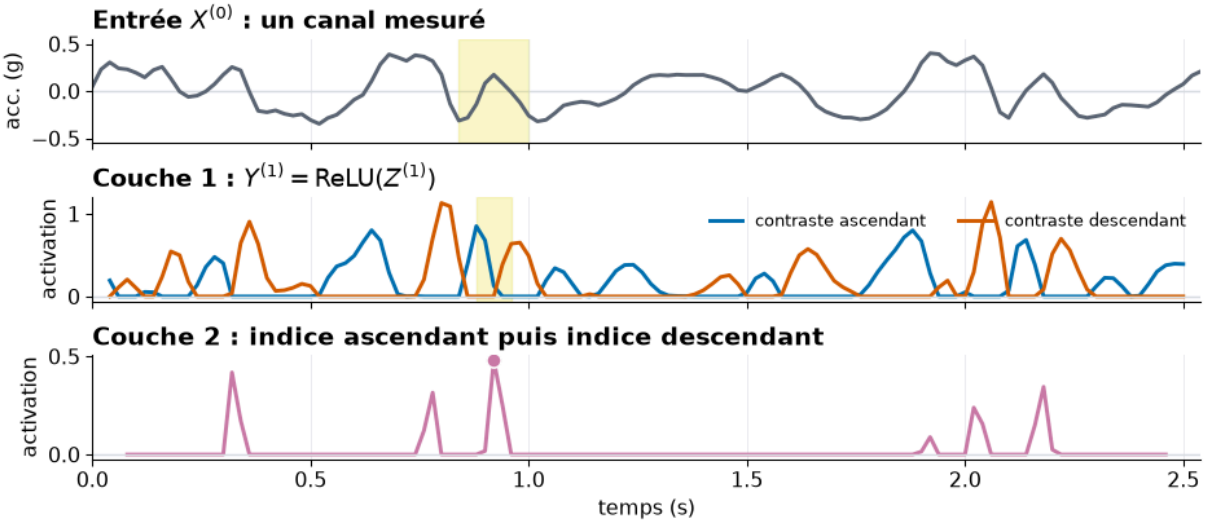
Le biais -0.8 exige une quantité suffisante d'indices combinés avant que la ReLU de la deuxième couche devienne positive. Le biais joue ici un rôle concret : il modifie le seuil d'activation tout en demeurant partagé entre toutes les positions de sortie.

Ces poids sont choisis à la main pour rendre le calcul interprétable. Ils illustrent un passage avant complet, et non un classifieur de la marche entraîné.

Les couches détectent des motifs de motifs

In [22]:
```
time1 = (np.arange(Y1.shape[0]) + (K - 1) / 2) / SAMPLE_RATE
time2 = (np.arange(Y2.shape[0]) + (K - 1)) / SAMPLE_RATE
best_pattern = int(np.argmax(Y2[:, 0]))
receptive_start = best_pattern
receptive_stop = best_pattern + 2 * K - 1

fig, axes = plt.subplots(3, 1, figsize=(10.0, 4.5), sharex=True)

axes[0].plot(time, X0[:, 0], color=GRAY)
axes[0].axvspan(
    time[receptive_start],
    time[receptive_stop - 1],
    color=HIGHLIGHT,
    alpha=0.28,
)
axes[0].set_title(r"Entrée $X^{(0)}$ : un canal mesuré", loc="left")
style_time_axis(axes[0], ylabel="acc. (g)", ylim=(-0.55, 0.55))

axes[1].plot(time1, Y1[:, 0], color=BLUE, label="contraste ascendant")
axes[1].plot(time1, Y1[:, 1], color=ORANGE, label="contraste descendant")
axes[1].axvspan(
    time1[best_pattern],
```

```
    time1[best_pattern + K - 1],
    color=HIGHLIGHT,
    alpha=0.28,
)
axes[1].set_title(
    r"Couche 1 : $Y^{(1)}=\operatorname{ReLU}(Z^{(1)})$",
    loc="left",
)
axes[1].set_ylabel("activation")
axes[1].axhline(0, color=LIGHT_GRAY, linewidth=1)
axes[1].grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
axes[1].legend(loc="upper right", ncol=2, fontsize=9.5)

axes[2].plot(time2, Y2[:, 0], color=PURPLE)
axes[2].scatter(
    time2[best_pattern],
    Y2[best_pattern, 0],
    color=PURPLE,
    edgecolor="white",
    linewidth=1.2,
    s=70,
    zorder=3,
)
axes[2].set_title(
    "Couche 2 : indice ascendant puis indice descendant",
    loc="left",
)
axes[2].set_ylabel("activation")
axes[2].set_xlabel("temps (s)")
axes[2].axhline(0, color=LIGHT_GRAY, linewidth=1)
axes[2].grid(axis="x", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)
axes[2].set_xlim(time[0], time[-1])

fig.tight_layout(h_pad=0.65)
plt.show()
```



La première couche reçoit un canal de signal et renvoie deux cartes d'activation. Ces deux colonnes sont précisément les deux canaux d'entrée de la deuxième couche. À

chaque position de cette deuxième couche, la fenêtre a la forme 5×2 : cinq positions voisines provenant de chacune des cartes d'activation de la première couche.

La région jaune du panneau central est la fenêtre à deux canaux qui produit l'activation marquée de la deuxième couche. La région jaune du panneau supérieur est son champ récepteur de neuf positions dans l'entrée originale. Le noyau de la couche 2 peut combiner des attributs qui apparaissent à des positions relatives et dans des canaux différents.

Les deux couches utilisent ici une corrélation croisée valide. La longueur de la séquence passe donc de 128 à 124, puis à 120. Le remplissage pourrait préserver la longueur et un pas supérieur pourrait la réduire plus rapidement, sans changer ce qui constitue l'entrée de la couche suivante.

Sources :

- L'entrée mesurée provient de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR. Les deux tableaux d'activations ont été calculés par le code affiché.

Que modifie le biais ?

$$Y[i, m] = \sum_{j=0}^{K-1} x[i + j] W[j, m] + b[m]$$

Le scalaire $b[m]$ décale toute la carte d'attributs produite par le noyau m.

Le biais n'est pas un autre motif temporel. Il existe **un biais par canal de sortie**, et le même scalaire est ajouté à chaque position. Le biais ne peut donc pas encoder l'endroit où apparaît un motif; une translation de l'entrée déplace toujours la réponse correspondante de la carte d'attributs.

Sans non-linéarité subséquente, le biais décale simplement chaque score d'une carte. Avec une unité linéaire rectifiée, $\mathrm{ReLU}(s + b) = \max(0, s + b)$, il modifie le score nécessaire pour qu'une activation devienne positive. On peut donc l'interpréter comme une composante du seuil d'activation.

Fenêtre locale ou champ récepteur ?

- Une **fenêtre locale** (*patch* ou *window*) est la tranche de l'entrée d'une couche lue par le noyau.
- Le **champ récepteur** d'une unité regroupe les positions de l'entrée originale qui peuvent l'influencer.

Avec un pas unitaire et sans dilatation,

$$R_1 = K_1, \qquad R_2 = R_1 + (K_2 - 1).$$

Deux noyaux de longueur 5 donnent à la deuxième couche un champ récepteur de $5 + (5 - 1) = 9$ positions.

Dans la première couche, une fenêtre de cinq positions d'entrée est aussi un champ récepteur de cinq positions originales. Dans ce cas particulier, les termes peuvent sembler interchangeables. Dans une couche plus profonde, une fenêtre de cinq positions contient toutefois cinq valeurs de la carte précédente, et chacune résume déjà plusieurs positions originales. Le champ récepteur profond est donc plus grand que la fenêtre de la couche courante.

Pour des pas arbitraires, il est utile de suivre l'espacement J_ℓ entre les unités adjacentes, mesuré en positions de l'entrée originale. En partant de $R_0 = 1$ et $J_0 = 1$,

$$R_\ell = R_{\ell-1} + (K_\ell - 1)J_{\ell-1}, \qquad J_\ell = J_{\ell-1}S_\ell.$$

Il s'agit du champ récepteur théorique : l'ensemble des positions qui pourraient influencer une activation selon l'architecture. L'intensité de l'influence réelle dépend aussi des poids appris et des non-linéarités.

Résumer le voisinage

. . .

In [23]:
```python
def max_pool1d(y, pool_size=2, stride=2):

    L_out = (len(y) - pool_size) // stride + 1
    p = np.zeros(L_out)

    for t in range(L_out):
        i = t * stride
        window = y[i : i + pool_size]
        p[t] = max(window)

    return p
```

$$p[t] = \max_{0 \le j < P} y[tS + j]$$

Peut-on résumer les activations voisines sans apprendre un autre ensemble de poids ?

La question motivante est : « Peut-on résumer les activations voisines sans apprendre un autre ensemble de poids ? » Le max-pooling répond par l'affirmative en appliquant un résumé fixe indépendamment à chaque carte d'attributs. Aucun poids ni biais n'est appris.

Ici, P est la taille de la fenêtre de pooling, S le pas et t la position de sortie; `i = t * stride` est le début de la fenêtre d'entrée correspondante. Le code sépare délibérément ces deux indices, comme dans l'exemple du pas pour la convolution.

La longueur de sortie utilise la division entière pour mettre en œuvre

$$L_{\text{out}} = \left\lfloor \frac{L - P}{S} \right\rfloor + 1.$$

Toute dernière fenêtre incomplète est ignorée. Dans la boucle, `window` rend le voisinage local visible et la fonction `max` de Python en sélectionne la plus grande valeur. Une compréhension de liste NumPy équivalente serait plus courte, mais masquerait l'allocation de la sortie, la relation $i = tS$ et la répétition de l'opération locale.

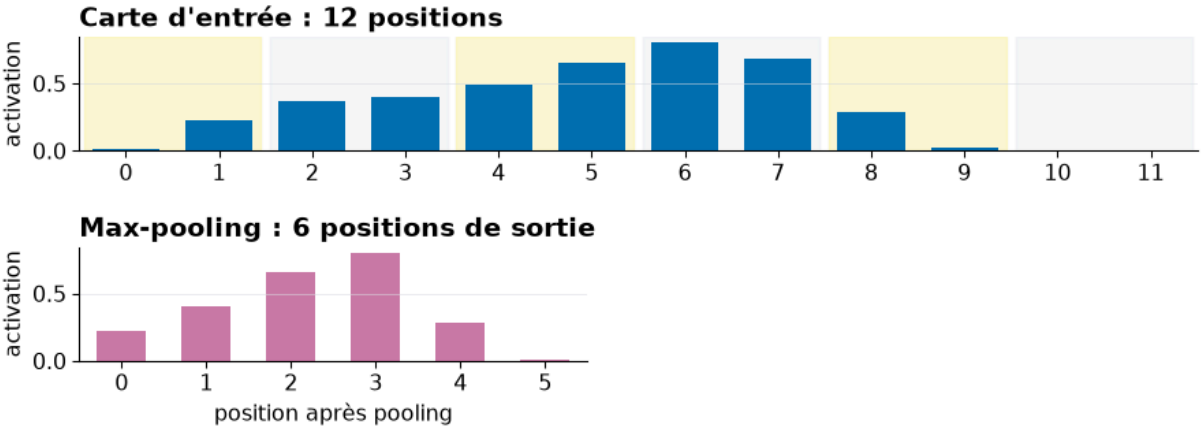
La fonction accepte une carte d'attributs. Pour un tableau multicanal, la même opération s'applique indépendamment à chaque canal; le pooling modifie donc normalement la longueur sans changer le nombre de canaux.

Le pooling résume les attributs déjà calculés par une couche convolutive précédente; il n'apprend pas de nouveaux détecteurs. Le max-pooling conserve la réponse la plus forte de chaque fenêtre, tandis que le pooling moyen conserve la réponse moyenne. Ce choix détermine l'information préservée lors du sous-échantillonnage.

Le pooling réduit la longueur de la séquence

In [24]:
```
# Sélectionner une courte région contenant des activations positives variées
pool_start = 24
pool_input = np.maximum(up_map[pool_start : pool_start + 12], 0)
pool_output = max_pool1d(pool_input, pool_size=2, stride=2)

fig = plt.figure(figsize=(11.0, 3.2))
grid = fig.add_gridspec(2, 2, width_ratios=[1, 1], hspace=0.85)
input_ax = fig.add_subplot(grid[0, :])
pooled_ax = fig.add_subplot(grid[1, 0])
empty_ax = fig.add_subplot(grid[1, 1])
empty_ax.axis("off")

for pair in range(6):
    color = HIGHLIGHT if pair % 2 == 0 else LIGHT_GRAY
    input_ax.axvspan(
        2 * pair - 0.45,
        2 * pair + 1.45,
        color=color,
        alpha=0.22,
    )

input_ax.bar(np.arange(12), pool_input, color=BLUE, width=0.72)
input_ax.set_title("Carte d'entrée : 12 positions", loc="left")
input_ax.set_ylabel("activation")
```

```
input_ax.set_xticks(np.arange(12))
input_ax.set_xlim(-0.5, 11.5)
input_ax.grid(axis="y", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)

pooled_ax.bar(np.arange(6), pool_output, color=PURPLE, width=0.58)
pooled_ax.set_title("Max-pooling : 6 positions de sortie", loc="left")
pooled_ax.set_ylabel("activation")
pooled_ax.set_xlabel("position après pooling")
pooled_ax.set_xticks(np.arange(6))
pooled_ax.set_xlim(-0.5, 5.5)
pooled_ax.grid(axis="y", color=LIGHT_GRAY, linewidth=0.7, alpha=0.55)

fig.tight_layout()
plt.show()
```



Avec une fenêtre et un pas de deux, chaque paire adjacente est remplacée par son maximum; 12 activations deviennent donc six.

Il s'agit d'une réduction dimensionnelle de la représentation : les couches ultérieures stockent et traitent six activations plutôt que douze. La couche de pooling ne possède aucun paramètre appris et ne change pas le nombre de paramètres des couches convolutives précédentes. Elle peut réduire le nombre de paramètres d'une tête Flatten–Dense ultérieure, puisque celle-ci reçoit alors un vecteur plus court. Une tête de pooling global élimine entièrement la dimension de longueur.

Le pooling conserve le fait qu'une forte activation s'est produite dans un petit voisinage, tout en éliminant sa position exacte dans la paire et toutes les valeurs non maximales. Il peut donc offrir une tolérance limitée aux petites translations, mais ne rend pas le réseau entier exactement invariant par translation.

Le pooling entraîne également une perte d'information. Une représentation plus petite peut réduire les calculs ultérieurs et la capacité du modèle, mais elle n'empêche pas intrinsèquement le surapprentissage. Son utilité dépend de la pertinence, pour la tâche, des détails de position éliminés.

Il est également imprécis de décrire le pooling en général comme une réduction du bruit. Le pooling moyen peut lisser les fluctuations à petite échelle, alors que le max-pooling

peut être sensible à une seule grande valeur positive aberrante. Le max-pooling n'est stable que si les perturbations ne changent pas la valeur gagnante de la fenêtre.

Le sous-échantillonnage permet aux couches ultérieures de travailler à une résolution plus grossière et augmente la portion de l'entrée originale pouvant influencer une activation profonde. Il peut favoriser des représentations multiéchelles, mais la hiérarchie apprise provient de l'empilement de couches convolutives et de non-linéarités.

Les entrées tracées utilisent une ReLU afin que les valeurs puissent être lues comme des activations non négatives. Le max-pooling peut aussi s'appliquer à des valeurs signées; sa définition arithmétique reste la même.

Sources :

- Les activations ont été calculées à partir de l'enregistrement 3363 des fichiers de signaux inertiels d'entraînement UCI HAR avec le noyau de contraste ascendant affiché.

Effet sur la longueur

	Opération	Paramètres	Canaux
Pas	somme pondérée	poids du noyau et biais	peuvent changer
Pooling	maximum ou moyenne fixe	aucun	généralement inchangés

Le pas et le pooling réduisent tous deux la longueur, mais différemment.

Une convolution avec un pas supérieur à un combine l'extraction d'attributs et le sous-échantillonnage. Ses poids appris déterminent l'information locale conservée, tandis que le nombre de noyaux détermine le nombre de canaux de sortie. Le pooling applique une règle fixe après le calcul des attributs, normalement de façon indépendante à chaque canal.

Aucune des deux opérations n'est obligatoire dans toutes les architectures. Les réseaux modernes utilisent souvent des convolutions à pas supérieur plutôt que des couches de pooling distinctes. Les introduire toutes deux en 1D rend le choix de modélisation explicite : la règle de sous-échantillonnage devrait-elle être apprise, ou devrait-elle être un résumé local fixe, comme un maximum ou une moyenne ?

La longueur du noyau, le nombre de canaux de sortie, le remplissage, le pas, la taille de la fenêtre de pooling et la profondeur du réseau sont des hyperparamètres architecturaux, et non des valeurs apprises par rétropropagation. L'expertise du domaine peut limiter les choix plausibles; la performance de validation et les contraintes de calcul guident ensuite leur sélection.

Des attributs aux prédictions

Comment les cartes d'attributs prédisent-elles ?

$$Y \in \mathbb{R}^{L_{\text{out}} \times C_{\text{out}}} \xrightarrow{\text{pooling global}} h \in \mathbb{R}^{C_{\text{out}}} \xrightarrow{\text{Dense}} z \in \mathbb{R} \xrightarrow{\sigma} \hat{p}_{\text{marche}}$$

$$h[m] = \max_i Y[i, m], \qquad z = \sum_m v[m] h[m] + a, \qquad \hat{p}_{\text{marche}} = \sigma(z).$$

> Devons-nous savoir **où** un attribut est apparu, ou seulement **s'il** est apparu ?

La dernière couche convolutive renvoie toujours une séquence : une activation à chaque position de chacune des cartes d'attributs de sortie. Une classification de la séquence entière, comme « marche », exige un résumé de taille fixe et une règle de prédiction.

Le max-pooling global produit un nombre par carte d'attributs. La valeur $h[m]$ représente la plus forte activation de l'attribut m dans toute la séquence. Le pooling moyen global constitue une autre possibilité; il enregistre plutôt l'activation moyenne. Les deux éliminent la position exacte et rendent donc la tête indépendante de la position. La nuance des diapositives sur la translation demeure : le remplissage, les frontières et le pas peuvent modifier l'invariance exacte.

La couche Dense combine les indices issus du pooling. Ses poids $v[m]$ indiquent comment l'attribut m contribue à la décision « marche », et le biais a modifie le seuil de décision. Pour une classification binaire, la sigmoïde transforme le logit scalaire z en un nombre compris entre zéro et un. Pour plusieurs activités, la couche Dense produirait un logit par classe, suivi d'un softmax.

La notation $\hat{p}_{\text{marche}}$ souligne qu'il s'agit de la sortie du réseau entier. Elle se distingue de Y, qui désigne la sortie d'une seule couche.

Que change une translation ?

Soit T_Δ une translation de l'entrée de Δ positions.

Équivariance

$$F(T_\Delta X) = T_\Delta F(X)$$

Une carte d'attributs convolutive se déplace avec le motif d'entrée.

Invariance

$$G(T_\Delta Y) = G(Y)$$

Une sortie qui agrège les positions peut préserver la décision finale.

T_Δ est un opérateur de translation : il décale chaque position de Δ. Pour une convolution ou une corrélation croisée de pas unitaire sur une séquence infinie, une entrée translatée produit une carte d'attributs translatée de la même manière. Il s'agit de l'équivariance par translation, exprimée par $F(T_\Delta X) = T_\Delta F(X)$.

Une fonction invariante élimine l'information de position associée à cette translation. Par exemple, le max-pooling global $G(Y) = \max_i Y[i]$ donne le même scalaire lorsque la carte d'attributs est translatée : $G(T_\Delta Y) = G(Y)$. Un classifieur fondé sur une telle agrégation peut donc prendre la même décision de catégorie à différentes positions.

Les équations décrivent des propriétés idéalisées. Les frontières finies et le remplissage peuvent perturber l'équivariance exacte près des extrémités. Une convolution de pas S n'est exactement équivariante qu'aux translations compatibles avec sa grille d'échantillonnage, comme les décalages multiples de S. Le pooling local peut accroître la tolérance aux petits décalages, mais ne produit pas à lui seul une invariance globale exacte par translation.

La prédiction exige une tête

In [25]:
```python
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def predict_walking(Y, v, a):
    C_out = Y.shape[1]
    h = np.zeros(C_out)
    for m in range(C_out):
        h[m] = max(Y[:, m])  # une valeur par carte d'attributs

    z = h @ v + a            # couche Dense
    return sigmoid(z)
```

$$\text{signal} \to \text{couches convolutives} \to \text{pooling global} \to \text{Dense} \to \text{prédiction}$$

Si la position importe, remplacez le **pooling global** par `h = Y.reshape(-1)`, appelé **aplatissement** (*flatten*).

Cette fonction constitue le calcul avant complet d'une tête de prédiction binaire. Si Y a la forme $(L_{\text{out}}, C_{\text{out}})$, la boucle parcourt chaque carte d'attributs m et place son maximum global dans $h[m]$. Le vecteur obtenu a la forme $(C_{\text{out}},)$. Les poids Dense v ont la même longueur et le biais a est un scalaire.

Dans l'illustration actuelle à deux couches, $Y^{(2)}$ possède un seul canal; la tête recevrait donc une seule valeur issue du pooling. En pratique, une dernière couche convolutive

comporte normalement plusieurs canaux de sortie, ce qui permet à la couche Dense de combiner les indices de plusieurs attributs appris.

Nous n'attribuons aucune valeur numérique à v et a, puisque ces paramètres doivent être appris. Des valeurs commodes choisies à la main illustreraient le calcul, mais ne produiraient pas un classifieur de la marche entraîné. Pendant l'entraînement, les noyaux et les biais convolutifs, les poids Dense et le biais Dense sont optimisés conjointement à partir d'exemples étiquetés au moyen d'une perte de classification, comme l'entropie croisée binaire.

L'aplatissement est une autre possibilité légitime. Il transforme les $L_{\text{out}}C_{\text{out}}$ activations en un seul vecteur avant la couche Dense, en conservant quel attribut est apparu à quelle position. La tête obtenue contient davantage de paramètres, exige une longueur d'entrée fixe et apprend des poids propres à chaque position. Le pooling global n'utilise que C_{out} valeurs, peut accepter des séquences de longueurs différentes et correspond à la question « cet attribut est-il apparu quelque part ? »

Consultez le carnet d'autoapprentissage *Prédire la charge ribosomique moyenne avec un réseau de neurones convolutif 1D* pour un exemple utilisant l'aplatissement.

Pour Q classes d'activités, remplacez v par une matrice comportant une colonne par classe, remplacez a par un vecteur de biais de longueur Q, puis appliquez softmax aux logits obtenus. Le fonctionnement des couches Dense, de la sigmoïde, de softmax et de l'entropie croisée a déjà été développé dans le cours sur les réseaux denses; il n'est pas nécessaire de le réintroduire ici.

Apprentissage

$$\text{exemples étiquetés} \rightarrow \text{passage avant} \rightarrow \text{perte}$$

$$\text{perte} \rightarrow \text{rétropropagation} \rightarrow \text{mise à jour} \rightarrow \text{répéter}$$

- **Appris :** chaque poids des noyaux, chaque biais des canaux de sortie ainsi que les poids et le biais de la tête de prédiction.
- **Spécifiés :** le remplissage, le pas, la règle de pooling, la fonction d'activation et les dimensions des couches.

Le réseau complet est entraîné **de bout en bout**.

La procédure d'apprentissage est la même que pour les réseaux denses. Un passage avant produit des prédictions, la perte les compare aux étiquettes et la rétropropagation indique comment la perte dépend de chaque paramètre appris. Un optimiseur met ensuite les paramètres à jour, et le processus se répète sur les exemples d'entraînement.

Les paramètres comprennent les poids et les biais de chaque couche convolutive, ainsi que ceux de la tête de prédiction. Ils sont appris conjointement : un noyau précoce n'est

utile que si ses activations aident les couches suivantes à réduire la perte de prédiction finale.

Le partage des paramètres ne crée pas une copie distincte du noyau à chaque position. Il existe un seul ensemble partagé de poids, et l'entraînement combine l'information provenant de toutes les positions où ces poids ont été utilisés.

Le pooling, le remplissage, le pas et ReLU participent au calcul avant, mais ne possèdent aucun poids ni biais appris sous les formes présentées ici. Leurs types et dimensions, ainsi que la longueur des noyaux, le nombre de canaux et la profondeur du réseau, sont des hyperparamètres architecturaux choisis par la personne qui conçoit le modèle.

Les noyaux de l'exemple filé ont été choisis à la main pour rendre chaque passage avant interprétable. Dans un classifieur de la marche entraîné, les noyaux et la tête de prédiction seraient plutôt appris à partir de nombreuses fenêtres de signaux étiquetées. Aucune nouvelle dérivation de la rétropropagation n'est requise dans ce cours.

Plus de dimensions, plus d'indices

$$\begin{aligned}&\text{1D: } W[j, c, m]\\&\text{2D: } W[j_1, j_2, c, m]\\&\text{3D: } W[j_1, j_2, j_3, c, m]\end{aligned}$$

Chaque noyau :

1. sélectionne une fenêtre locale ;
2. multiplie les valeurs et les poids correspondants ;
3. additionne sur les positions locales et les canaux d'entrée ;
4. produit une valeur dans une carte de sortie.

Plus de dimensions ajoutent des indices, et non une nouvelle idée.

Le passage des séquences aux images ou aux volumes n'introduit pas un principe différent. Il ajoute un indice de position locale pour chaque dimension spatiale ou temporelle. Un noyau d'image 2D se déplace selon deux axes; un noyau de volume 3D, selon trois. Les indices des canaux d'entrée et de sortie conservent exactement les mêmes rôles.

Par exemple, en omettant la notation du remplissage pour alléger l'écriture, une couche bidimensionnelle calcule

$$Y[i_1, i_2, m] = b[m] + \sum_{j_1=0}^{K_1-1} \sum_{j_2=0}^{K_2-1} \sum_{c=0}^{C_{\text{in}}-1} X[i_1 S_1 + j_1, i_2 S_2 + j_2, c] W[j_1, j_2, c, m].$$

Par rapport à l'équation 1D, la seule nouvelle opération mathématique est la somme sur le second indice de position locale j_2. Une mise en œuvre littérale ajouterait pareillement une boucle spatiale. Chaque noyau couvre toujours tous les canaux d'entrée et produit une carte d'attributs de sortie.

Cette correspondance explique pourquoi nous développons soigneusement le cas unidimensionnel. Une fois les données, la fenêtre locale, les poids des noyaux, les sommes et les cartes de sortie compris en 1D, les équations de dimension supérieure deviennent des prolongements du même motif plutôt que de nouvelles constructions.

Un téléphone peut-il reconnaître la marche ?

$$X \to \text{attributs locaux} \to \text{hiérarchie} \to \text{résumé global} \to \hat{p}_{\text{marche}}$$

> Oui, une fois les noyaux, les biais et la tête de prédiction appris à partir d'exemples étiquetés.

L'entrée est une courte séquence multicanal d'accéléromètre. Les noyaux convolutifs utilisent la connectivité locale et le partage des paramètres pour détecter des motifs utiles où qu'ils apparaissent. Plusieurs noyaux créent plusieurs cartes d'attributs, et chaque noyau couvre tous les canaux de son entrée.

L'empilement des couches compose les attributs locaux en motifs de motifs tout en agrandissant le champ récepteur. Un pas supérieur ou un pooling local peut réduire la résolution et les calculs ultérieurs, au prix d'une perte explicite de détails de position. Le pooling global demande ensuite si chaque attribut final est apparu quelque part, et la tête Dense combine ces indices en une prédiction de marche.

La rétropropagation et un optimiseur apprennent conjointement chaque noyau, chaque biais convolutif et chaque paramètre de la tête de prédiction à partir d'exemples étiquetés. L'architecture apporte le biais inductif : localité, partage, structure des canaux et choix de l'information de position conservée ou éliminée.

L'exemple filé montre toutes les parties de ce calcul, mais ne constitue pas un système entraîné de reconnaissance d'activités. Ses poids choisis à la main rendent le passage avant inspectable. Entraîner la même architecture sur de nombreuses fenêtres étiquetées en ferait un modèle prédictif.

Le même raisonnement s'étend aux images, aux volumes et aux données de dimension supérieure. Ces cas ajoutent des indices de position sans changer les idées sous-jacentes.

Prologue

Résumé

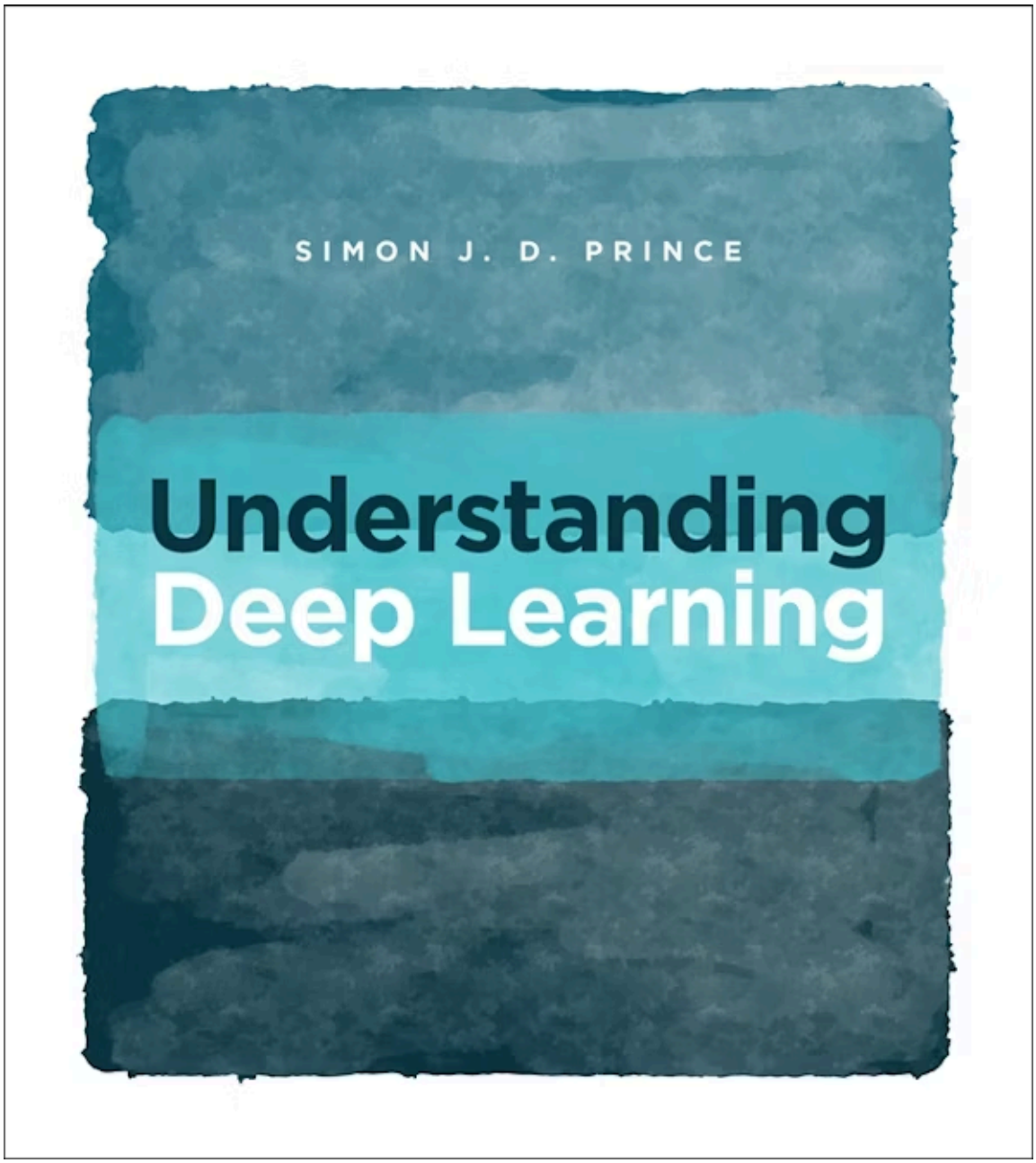
- **Localité et partage :** le même noyau examine chaque fenêtre locale et produit une carte d'attributs avec relativement peu de paramètres.
- **Canaux :** chaque noyau couvre tous les canaux d'entrée et produit une carte d'attributs de sortie.
- **Géométrie :** le remplissage contrôle les frontières; le pas et le pooling contrôlent la résolution et les détails de position conservés.

Résumé (suite)

- **Hiérarchie :** l'empilement des couches compose des attributs simples et agrandit les champs récepteurs.
- **Prédiction :** la convolution est équivariante par translation; l'agrégation des positions peut favoriser des prédictions invariantes, et une tête de prédiction produit la sortie finale.
- **Apprentissage :** la rétropropagation apprend conjointement chaque noyau, chaque poids et chaque biais.

Le même calcul s'étend des séquences 1D aux images 2D et aux volumes 3D en ajoutant des indices spatiaux.

Lectures complémentaires

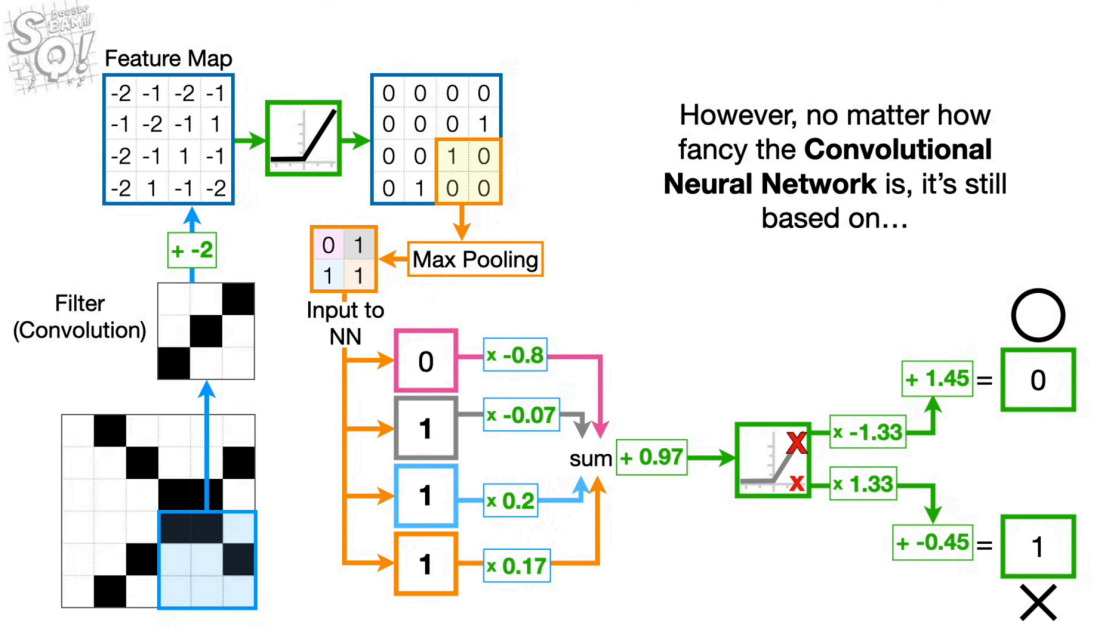


- Understanding Deep Learning (Prince 2023) est un manuel récent consacré aux concepts fondamentaux de l'apprentissage profond.
- Il part des principes de base et aborde ensuite des sujets contemporains comme les transformeurs, les modèles de diffusion, les réseaux neuronaux de graphes, les autoencodeurs, les réseaux antagonistes et l'apprentissage par renforcement.
- Il vise à rendre ces concepts compréhensibles sans s'attarder excessivement aux détails théoriques.
- Il comprend 68 exercices sous forme de carnets Python.
- Le livre suit un modèle « lire d'abord, payer plus tard ».

Ressources

- **A guide to convolution arithmetic for deep learning**
- Auteurs : Vincent Dumoulin et Francesco Visin
- Dernière révision : 11 janvier 2018
  - arXiv:1603.07285
  - Dépôt GitHub

StatQuest



La vidéo présente un exemple simple qui distingue les images de la lettre O de celles de la lettre X au moyen d'un seul filtre. Ce choix simplifie l'explication et la rend facile à suivre. En pratique, toutefois, chaque couche convolutive contient généralement des dizaines, voire des centaines de filtres.

Prochain cours

- Nous introduirons la **recherche**.

Références

Géron, Aurélien. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd éd. O'Reilly Media.

Krizhevsky, Alex, Ilya Sutskever, et Geoffrey E Hinton. 2012. « ImageNet Classification with Deep Convolutional Neural Networks ». In *Advances in Neural Information Processing Systems*, édité par F. Pereira, C. J. Burges, L. Bottou, et K. Q. Weinberger,

vol. 25. Curran Associates, Inc.
https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a
Paper.pdf.

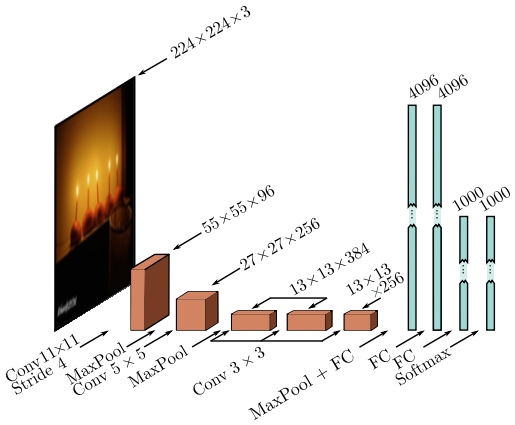
Prince, Simon J. D. 2023. *Understanding Deep Learning*. The MIT Press.
http://udlbook.com.

Russell, Stuart, et Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4e éd. Pearson. http://aima.cs.berkeley.edu/.

Simonyan, Karen, et Andrew Zisserman. 2015. « Very Deep Convolutional Networks for Large-Scale Image Recognition ». *International Conference on Learning Representations*.

Annexe : réseaux de neurones convolutifs bien connus

AlexNet

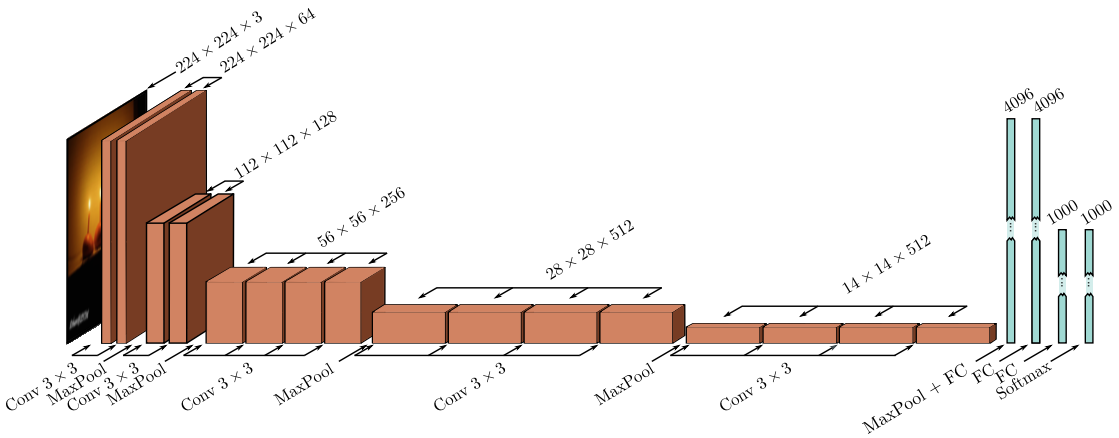


Krizhevsky et al. (2012)

Attribution : Prince (2023)

AlexNet comporte huit couches dotées de paramètres apprenables : cinq couches convolutives suivies de trois couches entièrement connectées. L'architecture comprend aussi des couches de max-pooling, des fonctions d'activation ReLU et du dropout afin d'améliorer l'entraînement et de réduire le surapprentissage.

VGG



Simonyan et Zisserman (2015)

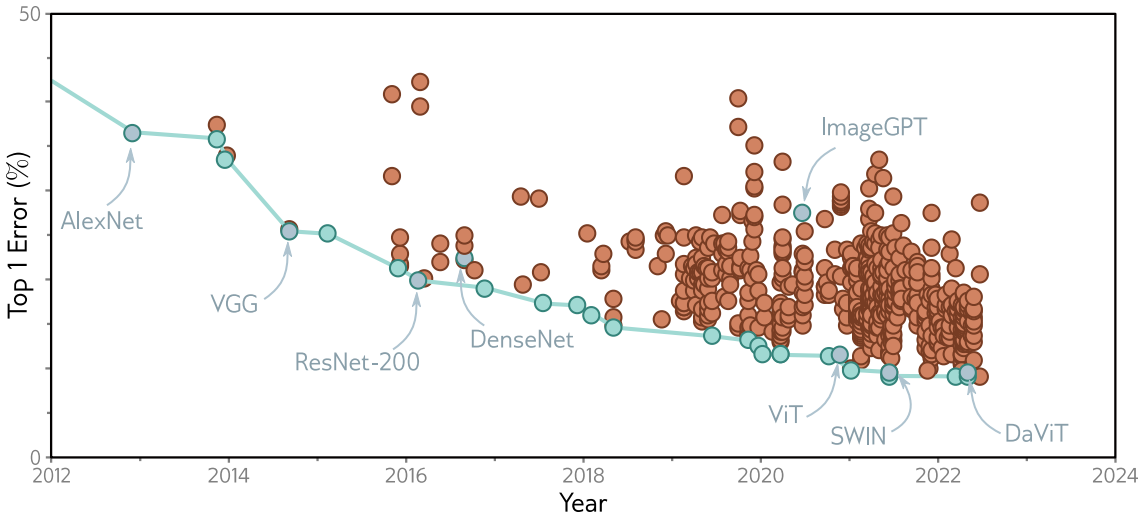
Attribution : Prince (2023)

Des renseignements complémentaires sont disponibles ici.

> Les réseaux convolutifs établissent alors l'état de l'art en reconnaissance visuelle. Ce projet étudie l'effet de leur profondeur sur leur exactitude dans un contexte de reconnaissance d'images à grande échelle. [traduction]

> Notre principale contribution est une évaluation rigoureuse de réseaux de profondeur croissante. Elle montre qu'une augmentation à 16–19 couches de poids, nettement plus que dans les travaux antérieurs, améliore sensiblement les configurations précédentes. Pour réduire le nombre de paramètres de réseaux si profonds, nous utilisons de très petits filtres 3×3 dans toutes les couches convolutives, avec un pas de 1. Consultez notre publication pour plus de détails. [traduction]

Performance des ConvNets



Attribution : Prince (2023)

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (**SI**GE)

Université d'Ottawa