Neural Network Architectures

CSI 4106 - Fall 2026

Marcel Turcotte
Version: Sep 13, 2026 16:21

Preamble

Message of the Day

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

Andrej Karpathy is a Slovak-Canadian computer scientist born in 1986 in Bratislava, who moved to Toronto with his family at the age of 15. He obtained a bachelor's degree in computer science and physics from the University of Toronto (2009) and a master's degree from the University of British Columbia (2011), where he worked with his advisor Michiel van de Panne on learning controllers for physically simulated figures. He then completed a Ph.D. at Stanford University under the supervision of Fei-Fei Li, focusing on the intersection of computer vision and natural language processing.

- Founding member and researcher at **OpenAI** (2015 – 2017).
- Senior Director of AI and Autopilot Vision at **Tesla, Inc.** (2017 – 2022).
- Returned to **OpenAI** in 2023 to lead a small team working on improving models, before leaving the company in 2024.

- Founder of **Eureka Labs** (since 2024), a startup focused on AI and education.

Learning outcomes

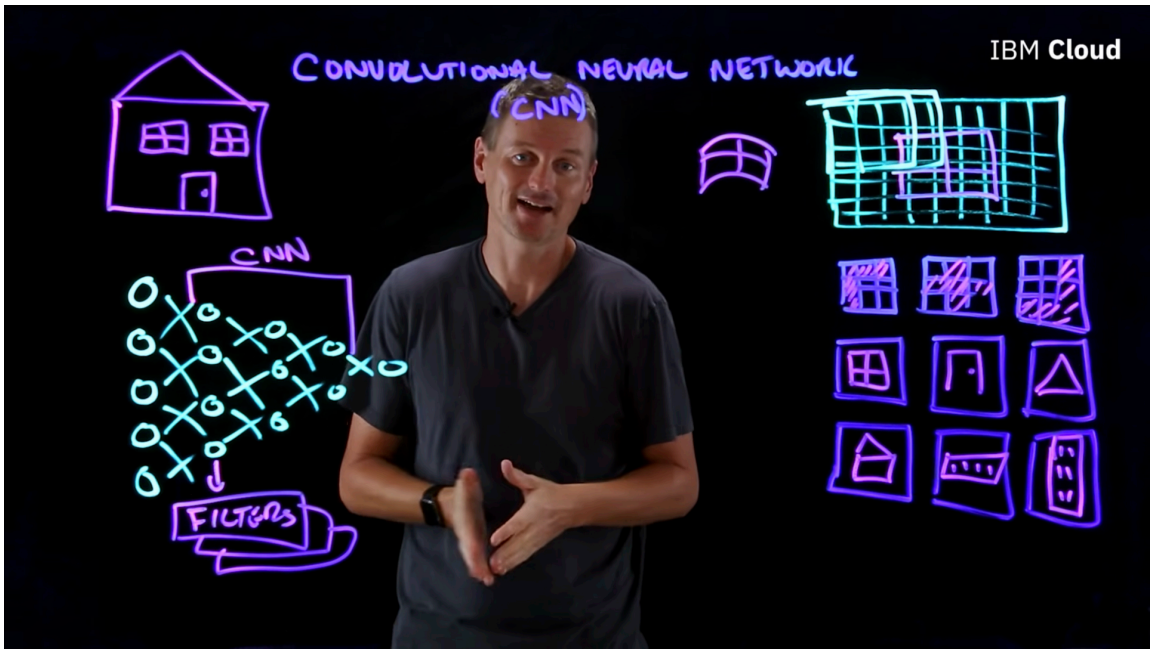
By the end of this lecture, you should be able to:

- **Explain** how local connectivity and parameter sharing give convolutional networks an inductive bias and reduce parameter counts.
- **Compute and trace** the forward pass of a 1D convolutional layer with multiple input channels, kernels, biases, and activations.
- **Determine** how kernel size, padding, stride, and pooling affect output shape, boundary behaviour, and retained positional information.
- **Explain** how stacked layers compose features, enlarge receptive fields, and supply a prediction head.
- **Distinguish** translation equivariance from invariance, including the roles of convolution and position aggregation.

The central skill is reasoning about one output value and then generalizing the same computation across positions, kernels, channels, and layers. You should be able to follow the array shapes and indices and explain the modelling consequences of architectural choices.

You are not expected to derive convolutional gradients, memorize hand-crafted kernel values, reproduce plotting code, or memorize the details of named architectures such as AlexNet and VGG.

Convolutional Neural Network (CNN)



An excellent high-level overview of CNNs.

Motivation

> "An MLP with just **one hidden layer** can theoretically model even the most **complex functions**, provided it **has enough neurons**. But for complex problems, **deep networks** have a much **higher parameter efficiency** than shallow ones: they can model complex functions **using exponentially fewer neurons** than shallow nets, allowing them to reach much **better performance** with the same amount of training data."

Géron (2019) § 10

The neural networks discussed thus far comprise neurons arranged in layers, with each unit in a given layer fully connected to every unit in the immediately subsequent layer. Although the number of layers and the number of neurons within each layer may vary, this organizational principle remains consistent throughout the network.

In theory, such a neural network can approximate any complex function under moderate, reasonable constraints. However, in practice, the number of weights grows rapidly, and the resulting networks can be difficult to train.

To address these challenges, architectures with explicit **inductive biases** have been proposed. Foundational model families include sequence and temporal architectures, such as recurrent neural networks (e.g., LSTM and GRU) and transformers. Another major class comprises generative models, including autoencoders, generative adversarial networks, and diffusion models. Finally, graph neural networks form a broader class of models designed to operate on relational data.

Inductive bias refers to the prior assumptions that a learning algorithm relies on to generalize from training data to unseen examples. It constrains or shapes the hypothesis space, enabling the model to preferentially select among multiple plausible solutions that may fit the observed data equally well. Without such bias, an algorithm cannot reliably generalize beyond the exact examples on which it was trained.

For example, logistic regression exhibits a strong inductive bias because it assumes that the input-output relationship can be adequately modeled by a linear function. In contrast, a high-degree polynomial imposes a weaker prior assumption, as its parameters allow it to represent substantially more complex relationships.

Convolutional neural networks (ConvNets, CNNs) are a class of neural networks characterized by a strong inductive bias, making them highly effective for tasks that align with this bias. CNNs underpin many modern vision systems, including those used in autonomous vehicles.

We therefore use convolutional neural networks to introduce the concept of **architectural design**.

Motivation (continued)

- Consider an **RGB image** with dimensions 224×224, which is relatively small by contemporary benchmarks.
- The image consists of $224 \times 224 \times 3 = 150,528$ **input features**.
- A neural network with merely a **single hidden dense layer** would require over 22,658,678,784 (**22 billion**) parameters, highlighting the computational complexity involved.

Here, we assume that the hidden layer contains the same number of units as the input layer, a common design choice. This preserves the dimensionality of the feature space and avoids imposing an arbitrary information bottleneck in the first layer. It also ensures that the network has sufficient capacity to model complex, nonlinear combinations of all pixels prior to downsampling.

In a fully connected feedforward neural network with k layers, if each layer contains n units, the number of parameters scales as $\mathcal{O}(kn^2)$.

Concepts

Our discussion will center around the following **key concepts**.

- **Inductive bias** and topological priors
- Local connectivity and **sparse interactions**
- **Parameter sharing** and equivariance
- Hierarchical **feature composition**
- Translation **equivariance** and **invariance** through position aggregation

Pedagogical approach

Beginning with classical 2D image processing provides an intuitive entry point and establishes motivation before introducing the underlying methodology. This approach effectively distinguishes the **visual intuition* of convolution from the** matrix operations** required for neural network implementation.

Phase 1: The Intuition (2D Historical Context)

Goal: Establish what a filter actually does using immediate visual feedback.

- **The Content:** Demonstrate classic, hand-crafted 2D filters (e.g., Sobel edge detection) on a standard grayscale image.

- **The Takeaway:** A small, sliding weight matrix transforms the input into a feature map. The notion of the receptive field is introduced intuitively, while channels, batches, and backpropagation are initially omitted.

2D CNNs are a foundational architecture for image data. However, 1D CNNs are also widely used. 3D CNNs are well established and are primarily applied to volumetric data, such as MRI and CT images in medical imaging. To keep the equations and visual representation as simple as possible, we begin with 1D CNNs.

Phase 2: The Mechanics (1D Neural Networks)

Goal: Isolate the mathematics of channels and kernels.

- **The Pivot:** While classic computer vision relied on humans guessing the correct matrix weights, convolutional layers *learn* them. To implement this learning mathematically—specifically how we handle deep representations—you strip away the spatial complexity.
- **The Content:** Execute the 1D signal progression (Single Channel -> Multi-Channel -> Multiple Kernels -> Stacked Layers).
- **The Takeaway:** Master the underlying dot-product arithmetic and the strict dimensional rules governing C_{in} and C_{out}.

Phase 3: The Synthesis (2D Neural Networks)

Goal: Re-introduce spatial dimensions to the rigorous tensor math.

- **The Content:** Bring the math from Phase 2 back to the images from Phase 1.
- **The Takeaway:** Realize that moving to 2D simply adds an iterator to existing, well-understood 1D channel logic.

Classical image processing

Intuition/2D Historical Context

- **Classical 2D image processing** establishes **motivation** before introducing the underlying **methodology**.
- Distinguishes the **visual intuition** from the **matrix operations**.

We will introduce **learning concepts** later; for now, we are simply processing images.

How does a kernel scan an image?

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

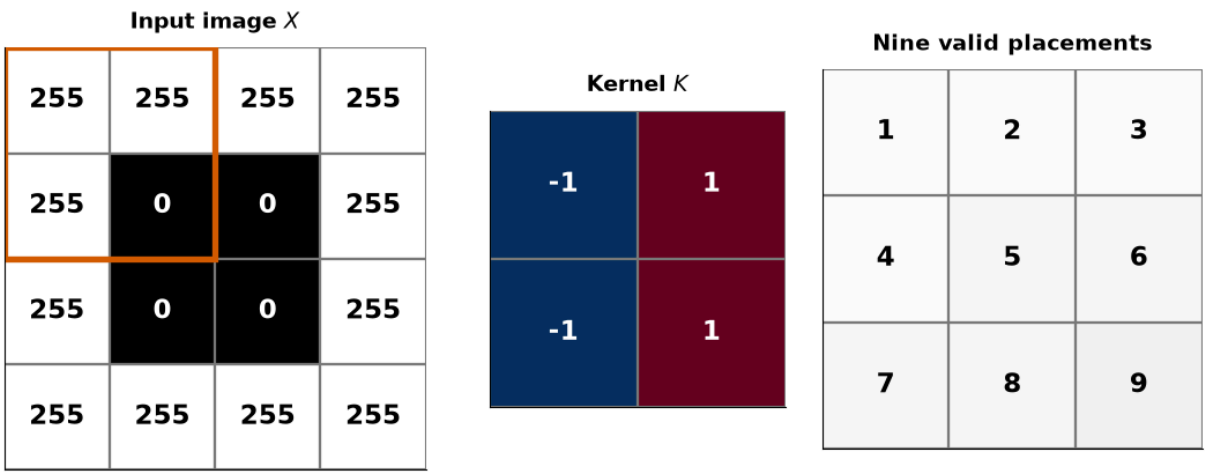
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
from matplotlib.patches import Rectangle
from PIL import Image

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,
})

# A small, long-form extract from the UCI HAR inertial-signal files.
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):
    """Apply a shared visual style to the signal plots."""
    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("time (s)")
    else:
        ax.tick_params(labelbottom=False)
```

In [3]:

```
def cross_correlation2d(X, K):
    """Apply a 2D kernel as written, using valid cross-correlation."""
    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):
    """Display a small matrix with its numerical values."""
    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, "Input image $X$", 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, "Kernel $K$", cmap="RdBu_r", vmin=-1, vmax=1,
)

draw_number_matrix(
    axes[2], placement_order, "Nine valid placements", cmap="Greys",
    vmin=0, vmax=100,
)

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



A **kernel** (aka filter) is a small matrix, usually 3×3, 5×5, or similar in size, that **slides over** the input data (such as an image) to perform **cross-correlation**.

An image is a two-dimensional array of numbers. Here, 0 represents a black pixel and 255 represents a white pixel, following the conventional range for an 8-bit grayscale image. These sixteen intensities form a small letter O.

The small 2×2 matrix is a kernel. We begin with its upper-left entry aligned with the upper-left input pixel, then move it one column at a time. After reaching the end of a row, we continue on the next row. With a 4×4 input and a 2×2 kernel, there are $3 \times 3 = 9$ valid placements.

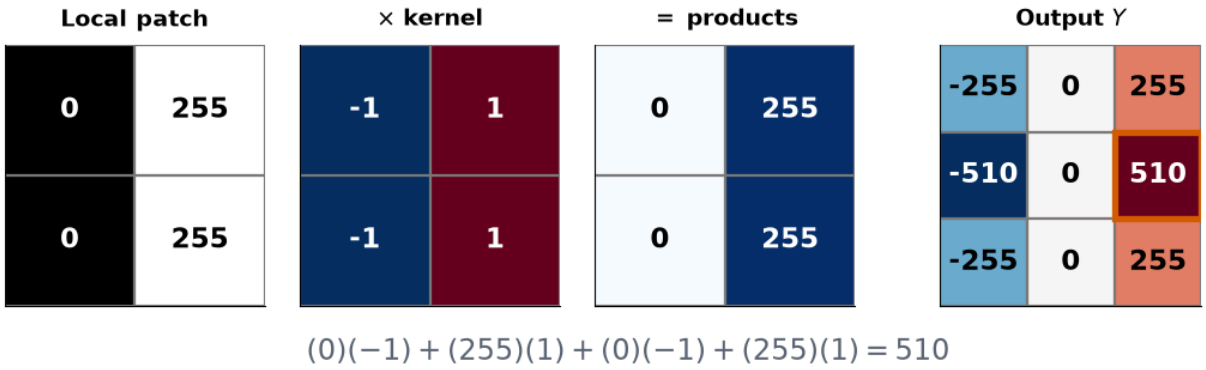
Only positions where the complete kernel overlaps the input are used. This is called **valid** cross-correlation. Boundary handling is deliberately postponed until later.

What does each placement compute?

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

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, "Local patch", cmap="Greys_r", vmin=0, vm
draw_number_matrix(axes[1], K_edge, r"$\times$ kernel", cmap="RdBu_r", vmin=
draw_number_matrix(axes[2], products, "$=$ products", cmap="Blues", vmin=0,
draw_number_matrix(axes[3], Y_image, "Output $Y$", 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()
```



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

Every placement produces one number in the output response map (**feature map**).

At placement $(i, j) = (1, 2)$, the kernel overlaps two black pixels and two white pixels. Corresponding entries are multiplied, and the four products are summed. The result, 510, is stored at $Y[1, 2]$.

The same calculation is repeated at each placement. The right edge of the O produces positive responses, the left edge produces negative responses, and the central column produces zero. The sign therefore records the direction of the change, not merely the presence of an edge.

This operation is **cross-correlation**, because the kernel is used as written. In mathematical convolution, the kernel is first reversed along both axes. Machine-learning libraries ordinarily compute cross-correlation but call the result **convolution** by convention. With an asymmetric edge kernel, reversal changes the sign of the response, although its absolute magnitude is unchanged.

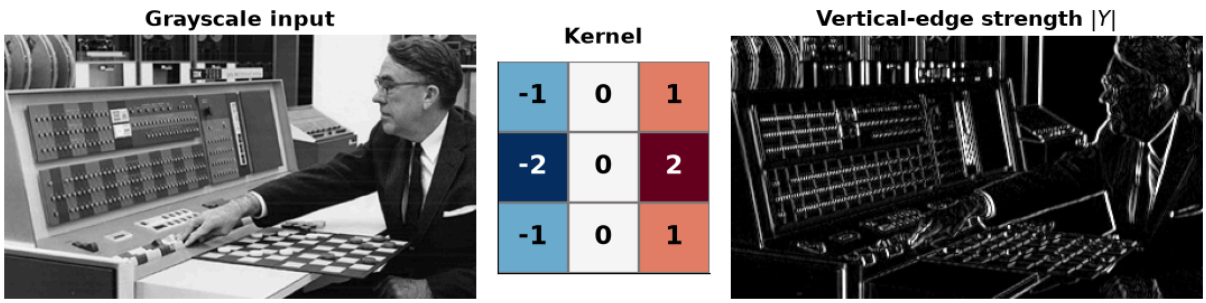
The output is often called a **response map** in classical image processing. In a convolutional network, the corresponding output of a kernel is a **feature map**.

No bias is added here because this is a hand-crafted classical image-processing operation. A learned convolutional layer will add a learned bias to each output feature map.

Which kernel reveals vertical edges?

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("Grayscale input")
axes[0].axis("off")

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

axes[2].imshow(
    vertical_strength, cmap="gray", vmin=0, vmax=vertical_limit,
)
axes[2].set_title("Vertical-edge strength $|Y|$")
axes[2].axis("off")

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



Large responses occur where brightness changes from left to right.

This is the Sobel x kernel. It estimates how rapidly image intensity changes across columns—the horizontal coordinate. A strong left-to-right change occurs at a vertical boundary, so K_x is commonly described as a vertical-edge detector. Keeping the derivative direction and edge orientation distinct avoids an otherwise common terminology error.

The signed response distinguishes a dark-to-light boundary from a light-to-dark boundary. The slide displays $|Y|$, the absolute response, because the immediate question is where strong vertical boundaries occur rather than their polarity. The clipping value is the 99th percentile so that a few very strong pixels do not make the remaining edges invisible.

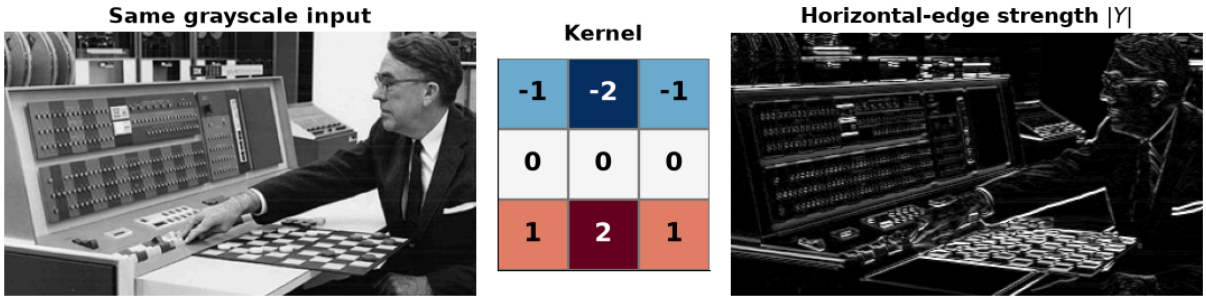
The kernel was chosen by a person; no machine learning is involved. The photograph provides a real image with natural variation, while the computer console, checkerboard, clothing, and background contain boundaries at several orientations.

Sources:

- IBM, "The games that helped AI evolve," photograph of Arthur Samuel at a computer console with a checkers board. https://www.ibm.com/history/early-games
- OpenCV documentation, "Sobel Derivatives." https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html

How can we reveal horizontal edges?

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("Same grayscale input")
axes[0].axis("off")

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

axes[2].imshow(
    horizontal_strength, cmap="gray", vmin=0, vmax=horizontal_limit,
)
axes[2].set_title("Horizontal-edge strength $|Y|$")
axes[2].axis("off")

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



Classical image processing uses fixed kernels; convolutional networks **learn** them.

Transposing the Sobel x kernel gives the Sobel y kernel. It estimates changes from one row to the next, so it responds strongly to horizontal boundaries. Using the same input image makes the comparison controlled: only the kernel weights have changed.

These kernels illustrate the central computational idea. A small set of weights is applied to every local neighbourhood, producing a response map that records where the selected pattern occurs. Classical image processing relies on domain experts to specify useful kernels. A convolutional neural network retains the local scanning operation but learns its kernel weights and biases from data.

Learned first-layer kernels may resemble edge or colour-contrast detectors, but a convolutional network is not restricted to reproducing classical kernels. Its kernels are optimized for the network's prediction objective.

Sources:

- IBM, "The games that helped AI evolve," photograph of Arthur Samuel at a computer console with a checkers board. https://www.ibm.com/history/early-games
- OpenCV documentation, "Sobel Derivatives." https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html

Central computational idea

> A **small set of weights** (kernel/filter) is applied to every **local neighbourhood**, producing a **response (feature) map** that records where the selected pattern occurs.

Unlike in image processing, where kernels are manually specified, **convolutional neural networks** learn their **kernels and biases automatically** during training.

The two-dimensional example provides the visual intuition. We now step back to one-dimensional signals so that kernel learning, multiple kernels, channels, stride, padding, pooling, and stacked layers can be introduced with fewer indices. The same ideas will then return to images by adding one spatial index.

Convolution in one dimension

Where do 1D data appear?

> What problems have observations arranged along **one ordered axis**?

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

The order matters: neighbouring values usually have a meaningful relationship.

One-dimensional data are organized along one ordered axis. Time is the most familiar axis, but the ordering can also represent position in text, position in a biological sequence, or distance along a route. **Neighbouring observations are usually related**, so exchanging two positions changes the meaning of the data.

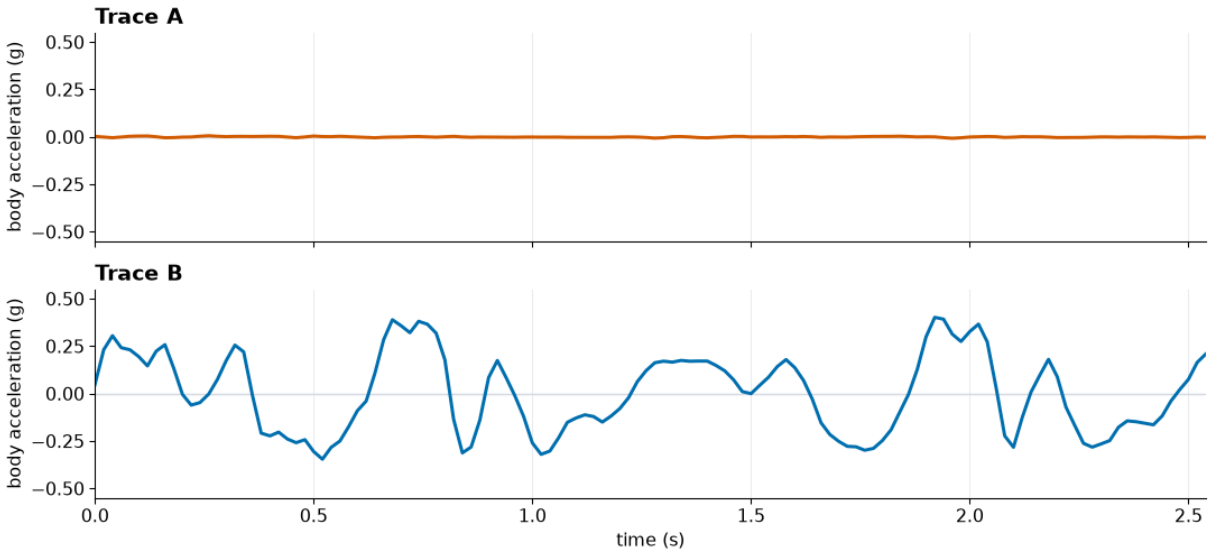
Familiar examples include:

- audio amplitude over time;
- temperature, rainfall, traffic, or financial measurements over time;
- accelerometer, gyroscope, vibration, ECG, and EEG signals;
- electricity consumption and other smart-meter signals;
- text represented as characters, tokens, or embeddings;
- DNA, RNA, and protein sequences; and
- elevation or another measurement sampled along a physical path.

The biological-sequence application is developed separately in the self-study notebook *Predicting Mean Ribosome Load with a 1D Convolutional Neural Network*.

Which trace comes from walking?

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="body acceleration (g)",
        ylim=(-0.55, 0.55),
    )

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



Two processed, 2.56-second accelerometer windows sampled at 50 Hz.

Trace A is sitting and Trace B is walking. The common vertical scale makes the difference in movement amplitude visible without exaggerating the sitting signal. These are representative windows near the median signal energy for their respective activities, rather than extreme cases selected for maximum visual separation.

The original experiment involved 30 volunteers wearing a Samsung Galaxy S II on the waist while performing six activities. The accelerometer and gyroscope were sampled at 50 Hz. The released version 1 dataset contains both preprocessed inertial-signal windows and a separate table of 561 engineered features.

This prototype does **not** use the 561-feature table in `X_train.txt`. It uses the 128 values in each row of `train/Inertial Signals/body_acc_x_train.txt`. The plotted signals have already been noise-filtered, divided into overlapping 2.56-second windows, and separated from the estimated gravity component. They are therefore processed sensor signals rather than untouched raw recordings.

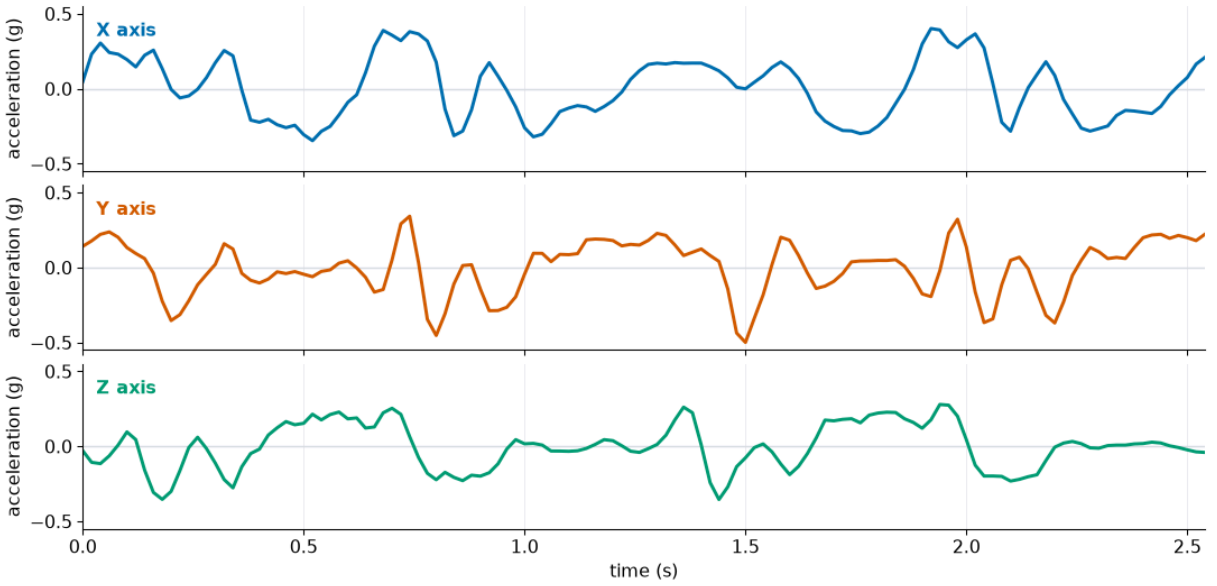
Sources:

- 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
- The walking example is record 3363 (zero-based source index 3362, subject 17) and the sitting example is record 395 (zero-based source index 394, subject
    1. in the training inertial-signal files.

A phone records three sequences

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"{channel.upper()} axis",
        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()
```



The phone supplies three simultaneous sequences: acceleration along its X, Y, and Z axes. An axis is not universally “vertical” or “horizontal”; its physical interpretation depends on the phone’s orientation and placement. All three channels nevertheless share the same time index because they were measured simultaneously.

The single-channel development below temporarily uses only the X-axis sequence. The three-channel representation returns after the one-channel operation and its implementation are established.

The dataset documentation links to a short video of the data-collection experiment, which shows the phone placement and the six recorded activities.

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
- Experiment video: https://www.youtube.com/watch?v=XOEN9W05_4A

A signal is an ordered list of numbers

In [9]:
```
first_values = pd.DataFrame({
    "index i": walking["sample"].head(8),
    "time (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))
```

```
index i  time (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
```

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

The index records position; the value is an ordinary real number.

The X-axis input used in the following slides is a vector of 128 real numbers. At 50 readings per second, the index increases by 0.02 seconds at every step. The notation $x[i]$ simply means the number stored at position i.

The file `data/uci_har_prototype.csv` is a compact, long-form extract made for this lecture. Each row contains the activity, the zero-based index of the source window, the subject identifier, the sample index, and the three body-acceleration values. It was derived from the three UCI files `body_acc_{x,y,z}_train.txt`, with labels and subjects joined from `y_train.txt` and `subject_train.txt`.

“Actual values” does not mean “unprocessed raw recordings.” The values have undergone the preprocessing described in the preceding notes. The distinction matters when interpreting what the model receives and what conclusions can be drawn from it.

Sources:

- The displayed values come from record 3363 of the UCI HAR training inertial-signal files.

What should the model assume?

- Nearby measurements form meaningful **local patterns**.
- A useful pattern may occur at **different positions**.
- The same detector should be **reused everywhere**.

These assumptions form an architectural **inductive bias**.

An **inductive bias** is a set of assumptions that guides what a model can learn efficiently from finite data. A Dense layer permits every output to depend on every input through unrelated weights. A convolutional layer instead assumes that local neighbourhoods matter and that a useful local relationship can recur at different positions.

Local connectivity creates **sparse interactions**: one output activation depends on a short input window rather than on the complete sequence. **Parameter sharing** then reuses the same kernel weights at every window position. These constraints **reduce the number of parameters** and make the feature map respond consistently when the same local pattern appears elsewhere.

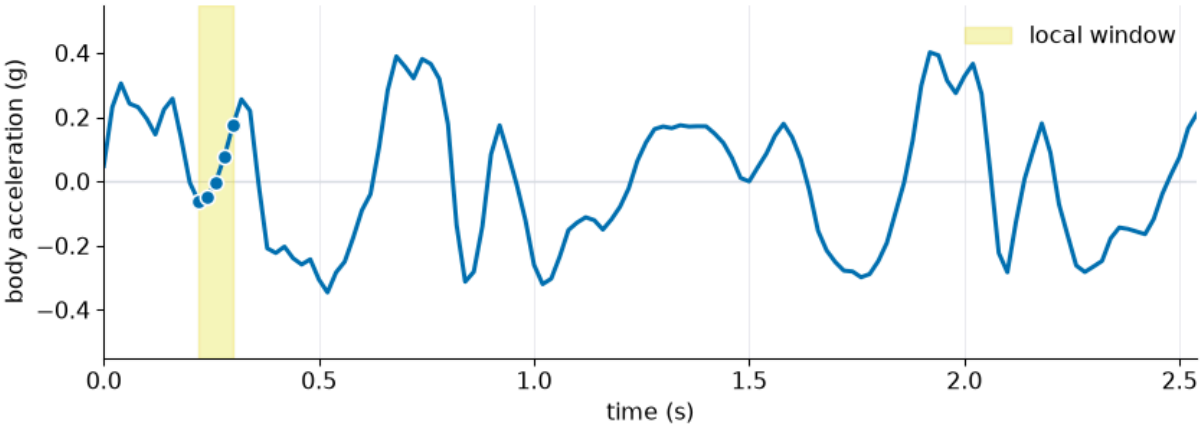
The assumptions are useful only when they match the problem. **Time and spatial position have a meaningful neighbourhood structure**, whereas **distinct sensor channels have different physical meanings**. A kernel therefore moves along the sequence dimension but spans all input channels; it is not reused by sliding across an arbitrary permutation of channels.

The architecture does not guarantee that the desired pattern will be learned. It makes certain solutions easier to represent and learn. The following hand-crafted kernel makes that bias visible before training is discussed.

One kernel scores one local contrast

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

# This window happens to cross zero; the kernel itself compares relative lev
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="local window",
)
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="body acceleration (g)",
    ylim=(-0.55, 0.55),
)
ax.legend(loc="upper right")
fig.tight_layout()
plt.show()
```



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

The score is large when the **last two values exceed the first two**. The middle value receives weight zero.

The kernel w contains five weights, so $K = 5$. At output position i, the score is

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

It compares the combined level of the last two values with the combined level of the first two values. The middle value $x[i+2]$ is ignored. The selected window happens to contain two negative values, one value near zero, and two positive values, which makes the upward transition visually clear.

The kernel does not test the signs of the input values. It also responds when all five values are positive or all five are negative, provided that the last two are sufficiently larger than the first two. It is therefore more precise to describe it as an upward-contrast detector than as an exact template for a particular sequence of signs.

The kernel is intentionally hand-crafted so that its behaviour can be predicted before performing the arithmetic. In a convolutional neural network, the kernel weights are learned from data.

Classical signal and image processing use the same local multiply-and-sum arithmetic with weights chosen from domain knowledge. A convolutional neural network treats those weights as parameters and learns useful detectors for the prediction task. The hand-crafted kernel here is therefore an explanatory starting point, not a claim about what a trained network must learn.

The operation shown here is cross-correlation because the kernel is not reversed. Deep-learning libraries conventionally call this operation convolution. The distinction between mathematical convolution and the deep-learning convention can be made explicit without changing the forward pass that follows.

Sources:

- The plotted signal comes from record 3363 of the UCI HAR training inertial-signal files.

Do x and y have the same length?

. . .

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)
```

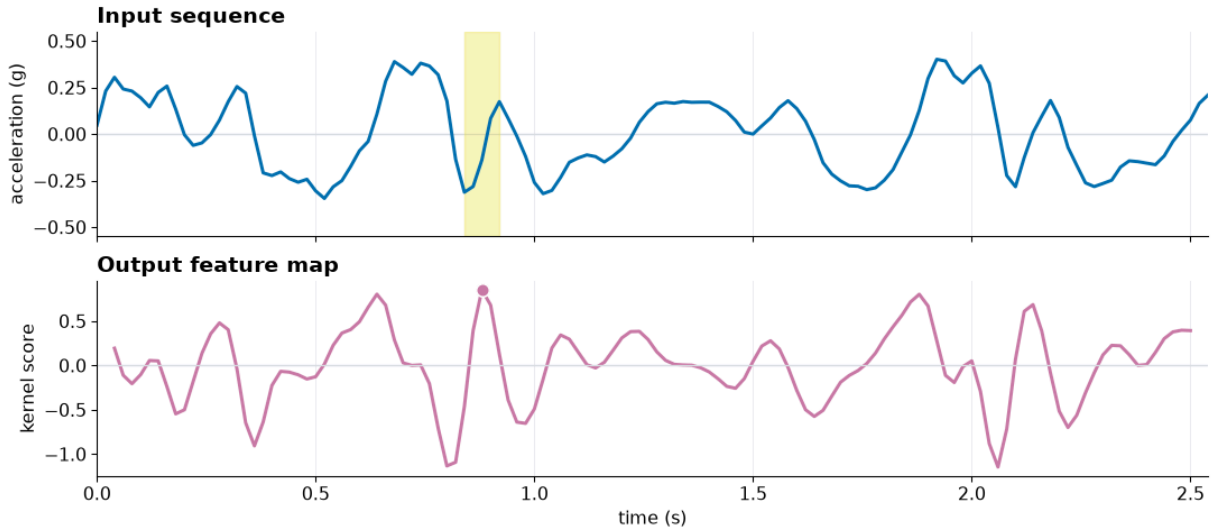
The answer is no for this implementation. This function performs valid one-dimensional cross-correlation with one input channel, one kernel, unit stride, no padding, and no bias. It is already a complete forward pass for this restricted case.

The loop variable i identifies an output position, and j identifies a position within the kernel. The update on the inner line forms one product and adds it to `y[i]`. Consequently, the source code displays the same summation as the equation rather than hiding it in a vectorized operation.

The same vector `w` is used at every position. This parameter sharing is what allows a local pattern to be detected wherever it occurs in the sequence. For $L = 128$ and $K = 5$, valid cross-correlation produces $L_{\text{out}} = 124$ scores.

Shared weights create a feature map

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("Input sequence", loc="left")
style_time_axis(
    axes[0],
    ylabel="acceleration (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("Output feature map", 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("kernel score")
axes[1].set_xlabel("time (s)")

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



The highlighted input window produces the marked output value. Moving the window across every valid position fills the output vector. In a convolutional network this output is called a **feature map**: each position records how strongly the learned local pattern is present there.

This is both **local connectivity** and **parameter sharing**. Each output value uses only five neighbouring input values, yet the same five weights are reused at all 124 valid positions. For comparison, a Dense mapping from 128 input values to 124 output values would contain $128 \times 124 = 15{,}872$ distinct weights. This convolution contains only five weights, or six learned parameters if one shared bias is included.

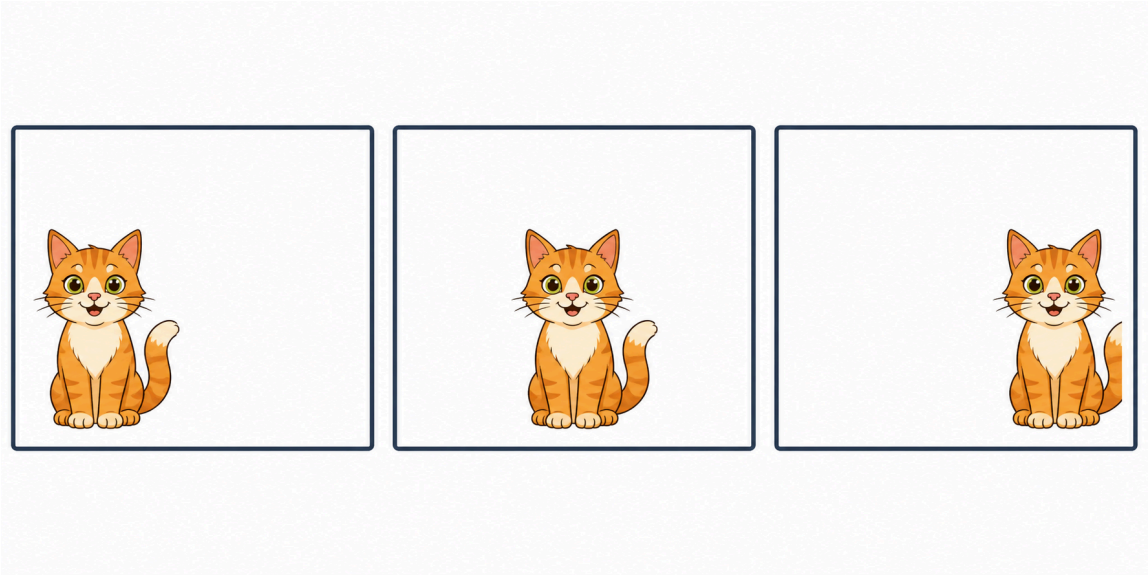
The reduction is not achieved by discarding input positions. It comes from requiring the same local rule to apply everywhere. That architectural constraint is the source of both parameter efficiency and translation equivariance.

The horizontal placement of each score is shown at the centre of its five-sample input window. This choice is useful for visualization, although the array itself is indexed simply as `up_map[0]` through `up_map[123]`.

Sources:

- The plotted signal comes from record 3363 of the UCI HAR training inertial-signal files.

Does location change the answer?



The location changes; the statement “this image contains a cat” does not.

The three frames contain the same object at three horizontal positions. A classifier should ideally give the same “cat present” decision for all three. This desired property of the final decision is translation invariance: changing the object’s position does not change the category assigned to the image.

A convolutional feature map behaves differently. When the cat moves, the strong cat-related activation should move with it. The internal representation is therefore translation equivariant rather than invariant. Keeping these two properties separate avoids the imprecise claim that convolution itself makes a network translation invariant.

Although the illustration is two-dimensional, the same distinction applies to the accelerometer sequence. Shifting a walking-related pattern in time should shift its feature-map response, while a window-level activity decision may remain unchanged.

Sources:

- The cat illustration was generated for this lecture using OpenAI image generation.

Equivariance

Let T_Δ translate an input by Δ positions.

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

A convolutional feature map moves with the input pattern.

T_Δ is a translation operator: it shifts every position by Δ. For stride-one convolution or cross-correlation on an infinite sequence, a translated input produces an identically translated feature map. This is translation equivariance, expressed by $F(T_\Delta X) = T_\Delta F(X)$.

The equation describes idealized properties. Finite boundaries and padding can disturb exact equivariance near the edges. A stride-S convolution is exactly equivariant only to translations compatible with its sampling grid, such as shifts by multiples of S.

How can y keep the length of x?

In [13]:
```
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))
```

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

Padding extends the input before the kernel is moved across it. Here, `padding=(2, 2)` adds two zeros to the start and two zeros to the end. Because the kernel has odd length $K = 5$, this balanced padding places one output at every original input position and preserves the length.

Numerically, $L = 128$, $K = 5$, and two padded zeros on each side give $L_{\text{out}} = 128 + 2 + 2 - 5 + 1 = 128$.

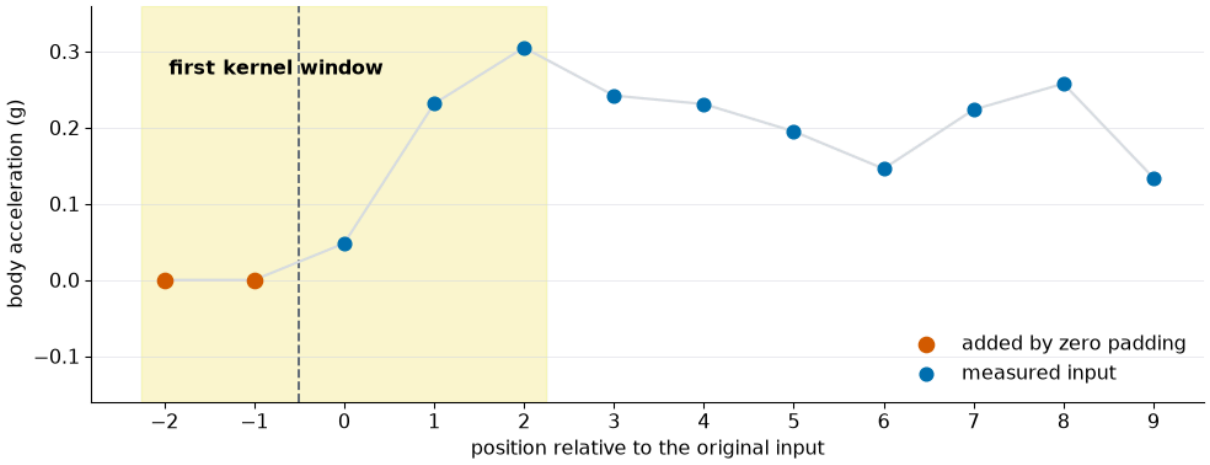
Padding need not be balanced. Adding values only at the start or only at the end shifts which input positions align with the output. “Same” padding usually means choosing the total padding required to preserve the length; for an odd-length kernel and unit stride, splitting it equally is the most natural choice.

The implementation now redefines `conv1d` by adding one optional argument. Its default value `(0, 0)` recovers the preceding valid cross-correlation function.

Values that were not measured

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="added by zero padding",
    zorder=3,
)
ax.scatter(
    padded_index[P:],
    padded_values[P:],
    color=BLUE,
    s=65,
    label="measured input",
    zorder=3,
)
ax.text(-1.95, 0.27, "first kernel window", fontweight="bold")
ax.set_xticks(padded_index)
ax.set_xlabel("position relative to the original input")
ax.set_ylabel("body acceleration (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()
```



Zero, reflection, replication, and circular padding encode different boundary assumptions. Boundary outputs can contain artefacts.

Padding creates values that were not measured. In the first kernel placement, two of the five values are artificial zeros. The resulting output therefore depends partly on the chosen boundary convention rather than only on observed data.

Zero padding assumes that values outside the observed interval are zero. Reflection padding mirrors values near the boundary; replication padding repeats the edge value;

and circular padding connects the end of the sequence to its start. Each choice is appropriate for some problems and misleading for others.

The effect is localized near the boundary in one layer, but it can propagate in a deep network. Padding is therefore a modelling decision, not merely a shape-management trick.

Sources:

- The measured values come from record 3363 of the UCI HAR training inertial-signal files. The orange zeros were added for illustration.

What happens if i advances by 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
```

...

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

The stride S is the distance between successive kernel placements. With `stride=1`, the input-window start i advances through $0, 1, 2, \ldots$. With `stride=2`, it advances through $0,2,4,` and half of the possible placements are skipped in this example.

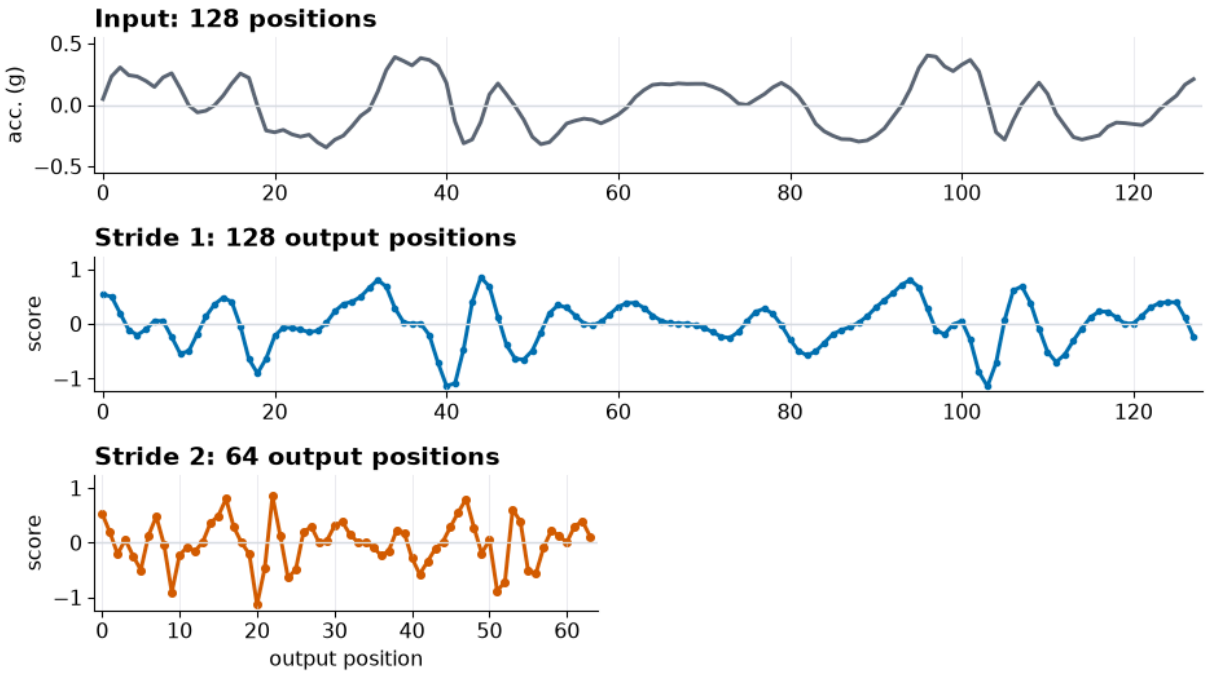
The output index t remains consecutive: $0, 1, 2, \ldots$. The input-window start used to compute it is $i = tS$. Distinguishing t from i makes the relationship between the shorter output and the original input explicit.

The floor in the length formula discards any final partial window. For $L = 128$, $K = 5$, balanced padding $(2, 2)$, and $S = 2$, the output contains 64 positions.

What happens when the stride is 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("Input: 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("Stride 1: 128 output positions", 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("Stride 2: 64 output positions", loc="left")
stride2_ax.set_ylabel("score")
stride2_ax.set_xlabel("output position")
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()
```



Stride 2 produces **half as many positions**, not values that are half as large.

Padding was selected so that stride 1 produces one score aligned with every input position. Increasing the stride to 2 uses the same kernel weights and the same padded input, but evaluates only every second placement. The result contains 64 scores rather than 128. In this shape-focused view, the 64-value array is deliberately drawn at half the physical width of the 128-value arrays.

If both outputs were plotted against elapsed time, they would cover the same 2.56-second interval: stride 2 would simply contain half as many samples across that interval. Plotting output index instead makes the change in array length immediately visible.

This is temporal downsampling. It reduces storage and computation in later layers, but it also removes positional resolution and can miss changes that occur between sampled placements. The phrase "half of x" would be misleading: the output is a learned local summary with half as many positions, not a vector containing every second original input value.

Sources:

- The input signal comes from record 3363 of the UCI HAR training inertial-signal files. The feature maps were computed by the displayed code.

Multiple kernels and channels

Multiple local patterns

Upward contrast

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

The first pair is subtracted from the last pair.

Downward contrast

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

The last pair is subtracted from the first pair.

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

How can multiple local patterns be modelled?

The first kernel gives a positive score when the last pair exceeds the first pair. The second is its negative and therefore gives a positive score for the opposite contrast.

This symmetric pair is pedagogically useful because the relationship between the outputs is predictable.

Stacking the two length-K vectors as columns produces a matrix W with shape $K \times C_{\text{out}}$. Here $C_{\text{out}} = 2$ because two kernels produce two **output channels**. One kernel produces one feature map, so adding kernels allows a layer to represent several kinds of local evidence.

The idea that one learned kernel corresponds to one easily named pattern is a useful first model, not a guarantee. Learned kernels need not be negatives of one another, and their joint representation can be more meaningful than any single kernel viewed in isolation.

How many kernels should a layer use?

- **Domain expertise** can suggest a small, interpretable set.
- More commonly, C_{out} is a **hyperparameter** selected using validation data.
- **More kernels** increase representational capacity, parameter count, memory, and computation.

There is no universal correct number of kernels. In a hand-designed system, domain knowledge may suggest a compact set of contrasts, frequencies, or motifs. In a learned convolutional network, the usual practice is to choose a candidate number of output channels, train the model, and evaluate it on validation data.

Too few kernels can create a representational bottleneck. Too many can waste resources and can contribute to overfitting when data are limited. The choice therefore reflects predictive performance as well as constraints on training time, memory, inference latency, and model size.

The loop gains one kernel index

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
```

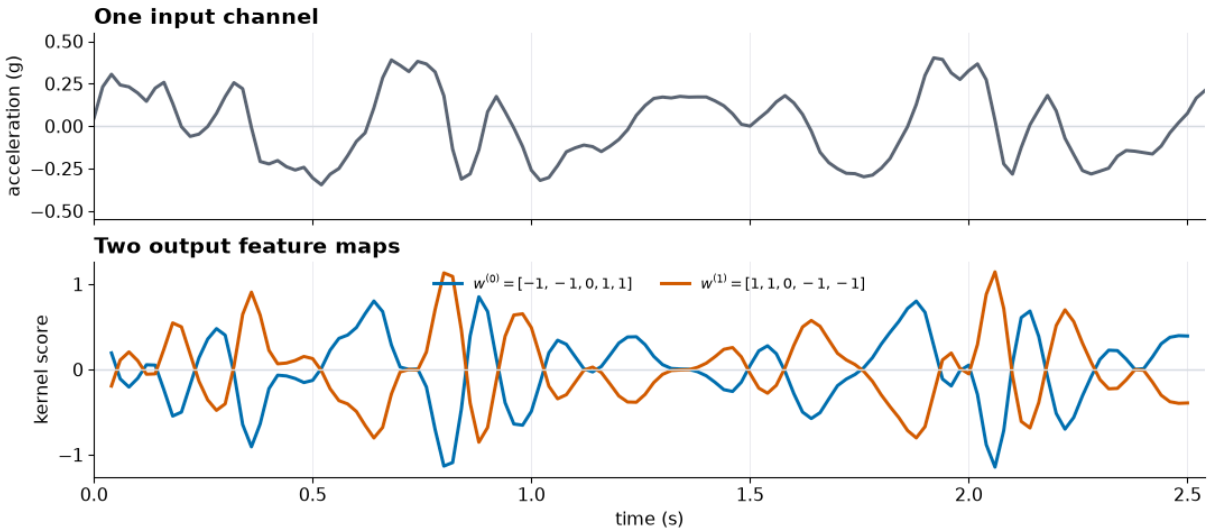
The revised function adds the kernel index m. `W[j, m]` selects position j within kernel m, `b[m]` selects its bias, and `Y[:, m]` is the corresponding feature map. Initializing `Y[t, m]` with the bias and then accumulating the K products mirrors the displayed equation directly. The padding and stride arguments remain available, so the implementation grows by successively adding concepts rather than replacing them with unrelated code.

The input is still one-dimensional, with shape (L). The kernel matrix has shape (K, C_{out}) and the output has shape $(L_{\text{out}}, C_{\text{out}})$. This example uses zero biases, no padding, and unit stride so that the two maps can be compared directly with the earlier valid feature map.

Each kernel creates one feature map

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("One input channel", loc="left")
style_time_axis(
    axes[0],
    ylabel="acceleration (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("Two output feature maps", loc="left")
axes[1].set_ylabel("kernel score")
axes[1].set_xlabel("time (s)")
axes[1].legend(loc="upper center", ncol=2, fontsize=9.5)

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



The two feature maps are plotted on the same axes so that their relationship is visible. Because the second kernel is exactly the negative of the first, its feature map is also the negative of the first feature map. An upward transition that produces a positive blue score produces an equally negative orange score, and a downward transition reverses those signs.

The explicit kernel values remain in the legend. This keeps the outputs tied to the computation rather than presenting the feature maps as unexplained curves.

Sources:

- The plotted signal comes from record 3363 of the UCI HAR training inertial-signal files.

Shapes keep everything interpretable

Object	Meaning	Shape
X	input sequence	(L, C_{in})
W	kernel weights	$(K, C_{\text{in}}, C_{\text{out}})$
b	one bias per output channel	(C_{out})
Y	output feature maps	$(L_{\text{out}}, C_{\text{out}})$

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

The symbols describe both the sizes of the arrays and the meaning of each index:

- L is the number of input positions and t is an output position;
- K is the kernel length and j is a position inside the kernel;
- C_{in} is the number of input channels and c selects one;
- C_{out} is the number of kernels and m selects one output feature map; and
- S is the stride, so $i = tS$ is the corresponding kernel start in the padded input $\widetilde{X}$.

With P_{left} and P_{right} padded positions,

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

The equation forms every product between the $K \times C_{\text{in}}$ input patch and the corresponding weights for kernel m, sums those products, and adds the bias $b[m]$. The tilde distinguishes the padded input from the original input. When there is no padding and $S = 1$, the equation reduces to the earlier valid-correlation expression.

Lowercase x, w, and y denoted the single-channel, single-kernel special case. Uppercase X, W, and Y denote the general arrays. These are notation conventions rather than different mathematical operations.

A patch spans all input channels

In [19]:
```
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
```

The new loop index c selects an input channel. For a fixed output position t and kernel m, the two inner loops visit every position j and every channel c in the $K \times C_{\text{in}}$ patch. Each product is added to the same scalar `Y[t, m]`, which starts at the bias $b[m]$.

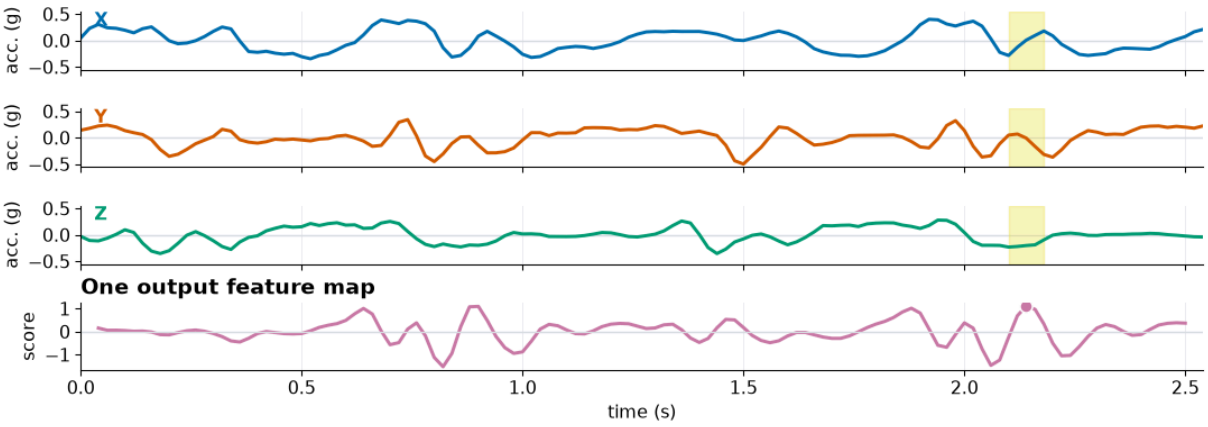
An expression such as `np.sum(patch * W[:, :, m])` is shorter, but it conceals the two distinct summations that are the main new idea on this slide. The implementation is therefore intentionally literal. A library implementation would vectorize these loops and support batches; those engineering improvements do not change the mathematical forward pass represented here.

The notebook successively redefines `conv1d`. Executing it from top to bottom therefore mirrors the conceptual progression: each definition replaces the previous restricted case with a more general one.

A kernel spans every input channel

In [20]:
```
X = walking.loc[:, ["x", "y", "z"]].to_numpy()

# One illustrative kernel: five positions, three channels, one output map.
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("time (s)")
axes[3].set_title("One output feature map", loc="left")

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



The yellow band is aligned across all three channels because the channels are simultaneous measurements. For one output position, the highlighted input patch has shape $K \times C_{\text{in}} = 5 \times 3$. Kernel 0 contains one five-weight vector for each channel, and all 15 products contribute to a single output score.

The illustrative kernel uses $w^{(0)}$ on X, $-0.6w^{(0)}$ on Y, and $0.4w^{(0)}$ on Z. Its purpose is to make the channel dimension concrete, not to claim that these hand-selected weights classify walking accurately. A trained network would learn all of these values from labelled examples.

Sources:

- The plotted signals come from record 3363 of the UCI HAR training inertial-signal files.

Stacking and pooling

What do X and Y represent?

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

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

- X: input to one layer;
- Z: pre-activation scores;
- $Y = g(Z)$: output activations; and
- the channels of $Y^{(\ell)}$ become the channels of the next layer's input.

The symbols X and Y describe the input and output of a single layer, not the input and final prediction of the entire network. Z names the result of the convolution and bias addition before the activation function g is applied. A network composes layers sequentially: the complete output activation array of layer ℓ becomes the complete input array of layer $\ell + 1$.

For the first convolutional layer, $X^{(0)}$ contains the measured signal channels. In the walking example these could be the X, Y, and Z accelerometer axes. For a later layer, the input channels are not raw sensor axes; they are the feature maps produced by the preceding layer. Consequently, $C_{\text{in}}^{(\ell+1)} = C_{\text{out}}^{(\ell)}$.

The displayed `conv1d` returns Z, the pre-activation scores. The next example uses $g(z) = \mathrm{ReLU}(z) = \max(0, z)$. ReLU is applied element by element, so it changes values without changing the sequence length or the number of channels.

Earlier slides used Y directly for the convolution scores because no nonlinearity had yet been introduced; that is the special case $g(z) = z$. From this point onward, Z and Y make the distinction explicit.

How to combine feature maps?

In [21]:
```
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     # earlier upward evidence
W2[-2:, 1, 0] = 0.5    # later downward evidence

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

The first layer applies the upward- and downward-contrast kernels already introduced. Its output $Z^{(1)}$ contains signed scores. ReLU sets negative scores to zero, producing nonnegative activations $Y^{(1)}$ with two channels: one for upward evidence and one for downward evidence.

The second-layer kernel spans both channels. Its positive weights on the first two positions of channel 0 reward earlier upward evidence. Its positive weights on the final two positions of channel 1 reward later downward evidence. It therefore responds to a sequence of learned features rather than directly to raw acceleration values.

This is hierarchical feature composition: layer 1 describes local contrasts, whereas layer 2 describes a pattern made from those contrasts. In this sense, deeper layers learn “patterns of patterns.” When the model is trained, these intermediate representations

are selected for their usefulness to the final prediction rather than specified manually as an independent feature-engineering step.

Hierarchy comes from composing learned layers and nonlinearities. Pooling or stride can change the resolution and help later layers cover a wider portion of the original input, but downsampling alone does not create a hierarchy of learned features.

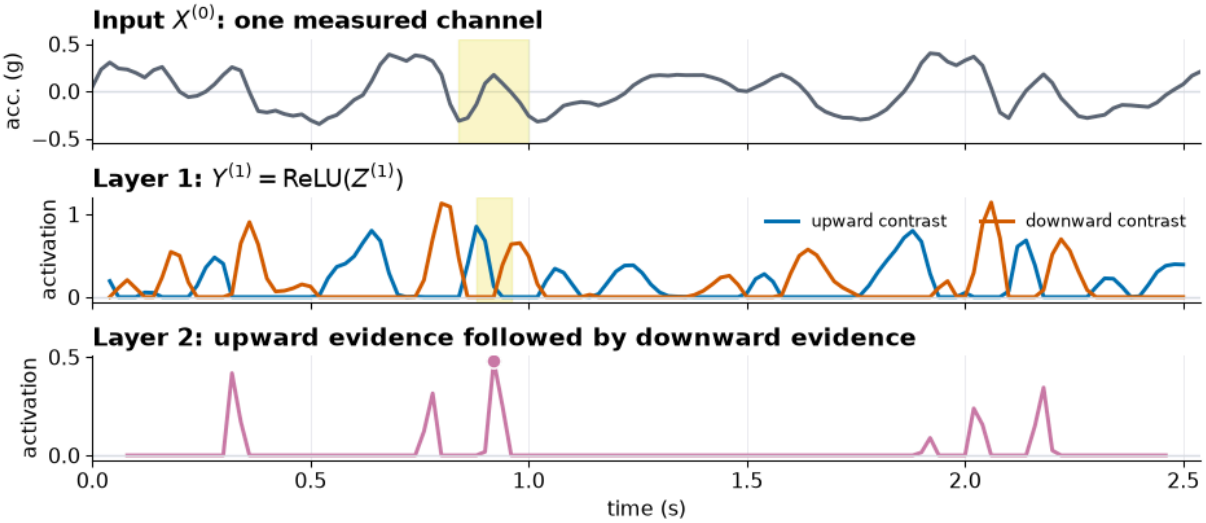
The bias -0.8 requires enough combined evidence before the second-layer ReLU becomes positive. This gives the bias term a concrete role: it changes the activation threshold while remaining shared over every output position.

These weights are hand-selected to keep the computation interpretable. They illustrate a complete forward pass, not a trained walking classifier.

Layers detect patterns of patterns

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"Input $X^{(0)}$: one measured channel", loc="left")
style_time_axis(axes[0], ylabel="acc. (g)", ylim=(-0.55, 0.55))

axes[1].plot(time1, Y1[:, 0], color=BLUE, label="upward contrast")
axes[1].plot(time1, Y1[:, 1], color=ORANGE, label="downward contrast")
axes[1].axvspan(
    time1[best_pattern],
    time1[best_pattern + K - 1],
    color=HIGHLIGHT,
    alpha=0.28,
)
axes[1].set_title(
    r"Layer 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(
    "Layer 2: upward evidence followed by downward evidence",
    loc="left",
)
axes[2].set_ylabel("activation")
axes[2].set_xlabel("time (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()
```



The first layer receives one signal channel and returns two activation maps. Those two columns are exactly the two input channels of the second layer. At each second-layer position, the patch has shape 5×2: five nearby positions from each first-layer activation map.

The yellow region in the middle panel is the two-channel patch that produces the marked second-layer activation. The yellow region in the top panel is its nine-position receptive field in the original input. The layer-2 kernel can combine features that occur at different relative positions and in different channels.

Both layers use valid cross-correlation here. The sequence length therefore changes from 128 to 124 to 120. Padding could preserve the length, and stride could reduce it more rapidly, without changing what constitutes the next layer's input.

Sources:

- The measured input comes from record 3363 of the UCI HAR training inertial-signal files. Both activation arrays were computed by the displayed code.

What does the bias change?

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

One scalar $b[m]$ shifts the entire feature map produced by kernel m.

The bias is not another temporal pattern. There is **one bias per output channel**, and the same scalar is added at every output position. Consequently, the bias cannot encode where a pattern occurs; translation of the input still translates the corresponding feature-map response.

Without a following nonlinearity, the bias simply shifts every score in one feature map. With a rectified linear unit, $\mathrm{ReLU}(s + b) = \max(0, s + b)$, the bias changes the score required for an activation to become positive. It can therefore be interpreted as part of the activation threshold.

Patch or receptive field?

- A **patch** or **window** is the slice of a layer's current input used at one kernel placement.
- A unit's **receptive field** is the set of original-input positions that can influence it.

For unit stride and no dilation,

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

With two kernels of length 5, one second-layer activation can depend on $5 + (5 - 1) = 9$ original positions.

In the first layer, a patch of five input positions is also a receptive field of five original positions. The terms can appear interchangeable in that special case. In a deeper layer, however, a five-position patch contains five preceding feature-map values, and each of those values already summarizes several original positions. The deeper receptive field is therefore larger than the current-layer patch.

For arbitrary strides, it is helpful to track the spacing J_ℓ between adjacent units measured in original-input positions. Starting with $R_0 = 1$ and $J_0 = 1$,

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

This is the theoretical receptive field: the set of positions that could influence an activation according to the architecture. The magnitude of the actual influence also depends on the learned weights and nonlinearities.

Summarizing the neighbourhood

...

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
```

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

Can nearby activations be summarized without learning another set of weights?

The motivating question is: “Can nearby activations be summarized without learning another set of weights?” Max pooling answers yes by applying a fixed summary independently to each feature map. No weights or biases are learned.

Here P is the pool size, S is the stride, t is the output position, and `i = t * stride` is the start of the corresponding input window. The code deliberately separates those two indices, just as the stride example did for convolution.

The output length uses integer division to implement

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

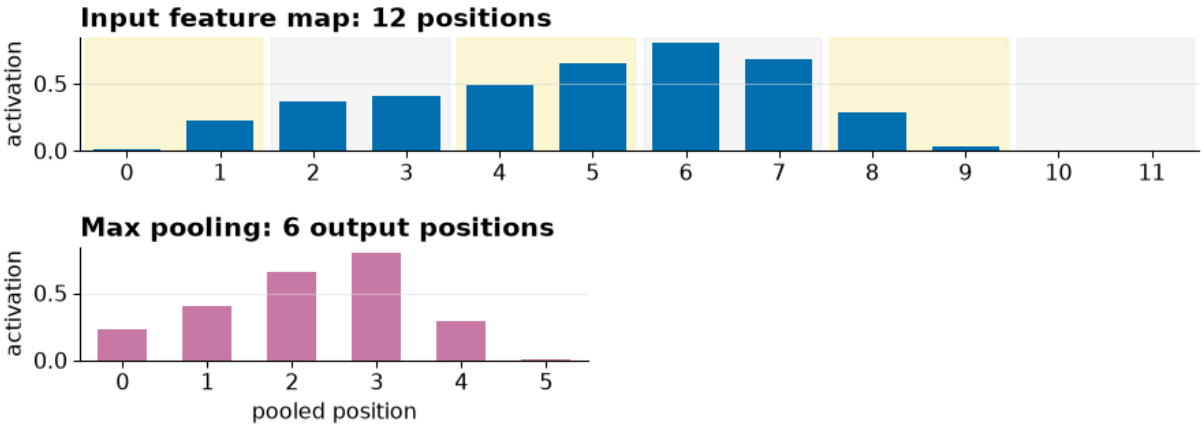
Any final partial window is discarded. Inside the loop, `window` makes the local neighbourhood visible and Python’s `max` selects its largest value. An equivalent NumPy list comprehension would be shorter, but it would hide the output allocation, the relationship $i = tS$, and the repeated local operation.

The function accepts one feature map. For a multichannel array, the same operation is applied independently to every channel, so pooling normally changes the sequence length without changing the number of channels.

Pooling summarizes features that a preceding convolutional layer has already computed; it does not learn new feature detectors. Max pooling retains the strongest response in each window. Mean pooling instead retains the average response. The choice determines what information is preserved during downsampling.

Pooling reduces sequence length

In [24]:
```
# Select a short region containing varied positive activations.
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("Input feature map: 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 output positions", loc="left")
pooled_ax.set_ylabel("activation")
pooled_ax.set_xlabel("pooled position")
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()
```



With a pool size and stride of two, every adjacent pair is replaced by its maximum, so 12 activations become six.

This is dimensionality reduction in the representation: later layers store and process six activations rather than twelve. The pooling layer itself has no learned parameters, and it does not change the parameter count of preceding convolutional layers. It can reduce the number of parameters in a subsequent Flatten–Dense head because that head then receives a shorter vector. A global pooling head removes the sequence-length dimension entirely.

Pooling retains whether a strong activation occurred in a small neighbourhood while discarding the exact within-pair position and every non-maximum value. This can provide limited tolerance to small translations, but it does not make the whole network exactly translation invariant.

Pooling is also an information-loss operation. A smaller representation can reduce later computation and model capacity, but it does not inherently prevent overfitting. Its value depends on whether the discarded positional detail is irrelevant to the task.

It is similarly imprecise to describe pooling in general as noise reduction. Mean pooling can smooth short-scale fluctuations, whereas max pooling can be sensitive to a single large positive outlier. Max pooling is stable only to perturbations that do not change the winning value in a window.

Downsampling lets later layers operate at a coarser resolution and increases how much of the original input can influence a deeper activation. This can support multiscale representations, but the learned hierarchy itself arises from stacking convolutional layers and nonlinearities.

The plotted inputs use a ReLU so that the values can be read as nonnegative activations. Max pooling can also be applied to signed values; the arithmetic definition is the same.

Sources:

- The activations were computed from record 3363 of the UCI HAR training inertial-signal files using the displayed upward-contrast kernel.

Impact on length

	Operation	Parameters	Channels
Stride	weighted sum	kernel weights and bias	may change
Pooling	fixed maximum or mean	none	usually unchanged

Stride and pooling both reduce length, but differently

A strided convolution combines feature extraction and downsampling. Its learned weights determine which local information is retained, and the number of kernels determines the number of output channels. Pooling applies a fixed rule after features have been computed, normally treating each channel independently.

Neither operation is mandatory in every architecture. Modern networks often use strided convolutions instead of separate pooling layers. Introducing both in one dimension makes the modelling choice visible: should the downsampling rule be learned, or should it be a fixed local summary such as a maximum or mean?

Kernel length, number of output channels, padding, stride, pool size, and network depth are architectural hyperparameters rather than values learned by backpropagation. Domain knowledge can constrain plausible choices; validation performance and computational requirements then guide their selection.

From features to predictions

How do feature maps predict?

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

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

> Do we need to know **where** a feature occurred, or only **whether** it occurred?

The final convolutional layer still returns a sequence: one activation at each output position in each output feature map. A sequence-level classification such as “walking” requires a fixed-size summary and a prediction rule.

Global max pooling produces one number per output feature map. The value $h[m]$ records the strongest activation of feature m anywhere in the sequence. Global mean pooling is another possibility; it records the average activation instead. Both discard the exact position and therefore make the head position-agnostic. The qualification from the translation slides still applies: padding, boundaries, and stride can affect exact invariance.

The Dense layer combines the pooled feature evidence. Its weights $v[m]$ say how feature m contributes to the walking decision, and the bias a changes the decision threshold. For binary classification, the sigmoid converts the scalar logit z into a number between zero and one. For multiple activities, the Dense layer would return one logit per class, followed by a softmax.

The notation $\hat{p}_{\text{walking}}$ emphasizes that this is the output of the entire network. It is distinct from Y, which denotes the output of one layer.

What changes under translation?

Let T_Δ translate an input by Δ positions.

Equivariance

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

A convolutional feature map moves with the input pattern.

Invariance

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

A position-aggregating readout can preserve the final decision.

T_Δ is a translation operator: it shifts every position by Δ. For stride-one convolution or cross-correlation on an infinite sequence, a translated input produces an identically translated feature map. This is translation equivariance, expressed by $F(T_\Delta X) = T_\Delta F(X)$.

An invariant function discards the position information relevant to that translation. For example, global maximum pooling $G(Y) = \max_i Y[i]$ gives the same scalar when the feature map is translated: $G(T_\Delta Y) = G(Y)$. A classifier built on such an aggregation can therefore make the same category decision at different positions.

The equations describe idealized properties. Finite boundaries and padding can disturb exact equivariance near the edges. A stride-S convolution is exactly equivariant only to translations compatible with its sampling grid, such as shifts by multiples of S. Local

pooling can increase tolerance to small shifts but is not, by itself, exact global translation invariance.

Prediction requires a head

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])  # one value per feature map

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

$$\text{signal} \to \text{convolutional layers} \to \text{global pooling} \to \text{Dense} \to \text{prediction}$$

If position matters, replace **global pooling** with `h = Y.reshape(-1)`, called **flatten**.

This function is the complete forward computation of a binary prediction head. If Y has shape $(L_{\text{out}}, C_{\text{out}})$, the loop visits each feature map m and stores its global maximum in $h[m]$. The resulting vector has shape $(C_{\text{out}},)$. The Dense weights v have the same length, and the bias a is a scalar.

In the current two-layer illustration, $Y^{(2)}$ has one channel, so the head would receive one pooled value. A practical final convolutional layer normally has several output channels, allowing the Dense layer to combine evidence from several learned features.

No numerical values are assigned to v and a here because those parameters must be learned. Choosing convenient values by hand could demonstrate arithmetic, but it would not produce a trained walking classifier. During training, the convolutional kernels, convolutional biases, Dense weights, and The Dense bias is optimized jointly from labelled examples using a classification loss such as binary cross-entropy.

Flattening is a legitimate alternative. It turns all $L_{\text{out}}C_{\text{out}}$ activations into one vector before the Dense layer, retaining which feature occurred at which position. The resulting head has more parameters, expects a fixed input length, and learns position-specific weights. Global pooling uses only C_{out} values, can accommodate different sequence lengths, and fits the question “did this feature occur anywhere?”

Consult the self-study notebook *Predicting Mean Ribosome Load with a 1D Convolutional Neural Network* for an example where flattening is used.

For Q activity classes, replace v with a matrix having one column per class, replace a with a length-Q bias vector, and apply softmax to the resulting logits. The mechanics of

Dense layers, sigmoid, softmax, and cross-entropy have already been developed in the dense-network lecture, so they need not be reintroduced here.

How are all parameters learned?

$$\text{labelled examples} \rightarrow \text{forward pass} \rightarrow \text{loss}$$

$$\text{loss} \rightarrow \text{backpropagation} \rightarrow \text{parameter update} \rightarrow \text{repeat}$$

- **Learned:** every kernel weight, every output-channel bias, and the prediction-head weights and bias.
- **Specified:** padding, stride, pooling rule, activation function, and layer sizes.

The complete network is trained **end to end**.

The learning procedure is the same one used for Dense networks. A forward pass produces predictions, the loss compares those predictions with the labels, and backpropagation communicates how the loss depends on every learned parameter. An optimizer then updates the parameters, and the process repeats over training examples.

The parameters include the weights and biases of every convolutional layer as well as the weights and bias in the prediction head. They are learned jointly: an early kernel is useful only insofar as its activations help later layers reduce the final prediction loss.

Parameter sharing does not create a separate copy of a kernel at every position. There is one shared set of kernel weights, and training combines the evidence arising from every position where those weights were used.

Pooling, padding, stride, and ReLU participate in the forward computation but have no learned weights or biases in the forms shown here. Their types and sizes, along with kernel length, channel counts, and network depth, are architectural hyperparameters selected by the model designer.

The kernels in the running example were chosen by hand to make each forward pass interpretable. In a trained walking classifier, those kernels and the prediction head would instead be learned from many labelled signal windows. No new backpropagation derivation is required for this lecture.

More dimensions add indices

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

Each kernel still:

1. selects a local patch;
2. multiplies matching values and weights;
3. sums over local positions and input channels; and
4. produces one value in one output feature map.

More dimensions add indices, not a new idea.

Moving from sequences to images or volumes does not introduce a different principle. It adds one local-position index for every spatial or temporal dimension. A 2D image kernel moves over two axes; a 3D volume kernel moves over three. The input-channel and output-channel indices retain exactly the same roles.

For example, omitting padding notation for readability, a two-dimensional layer computes

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

Compared with the 1D equation, the only new mathematical operation is the sum over the second local-position index j_2. A literal implementation similarly adds one spatial loop. Every kernel still spans all input channels and produces one output feature map.

This correspondence is the reason for developing the one-dimensional case carefully. Once the data, local patch, kernel weights, sums, and output feature maps are understood in 1D, the higher-dimensional equations are extensions of the same pattern rather than new constructions.

Can a phone recognize walking?

$$X \to \text{local features} \to \text{feature hierarchy} \to \text{global summary} \to \hat{p}_{\text{walking}}$$

> Yes—after the kernels, biases, and prediction head have been learned from labelled examples.

The input is a short multichannel accelerometer sequence. Convolutional kernels use local connectivity and parameter sharing to detect useful patterns wherever they occur. Multiple kernels create multiple feature maps, and every kernel spans all channels of its input.

Stacking layers composes local features into patterns of patterns while increasing the receptive field. Stride or local pooling can reduce resolution and later computation, with an explicit loss of positional detail. Global pooling then asks whether each final feature occurred anywhere, and the Dense head combines that evidence into a walking prediction.

Backpropagation and an optimizer learn every kernel, convolutional bias, and prediction-head parameter jointly from labelled examples. The architecture contributes the inductive bias: locality, sharing, channel structure, and the choice of how positional information is retained or discarded.

The running example demonstrates all parts of this computation but is not itself a trained activity-recognition system. Its hand-selected weights make the forward pass inspectable. Training on many labelled windows would turn the same architecture into a predictive model.

The same reasoning extends to images, volumes, and higher-dimensional data. Those cases add positional indices; they do not change the underlying ideas.

Prologue

Summary

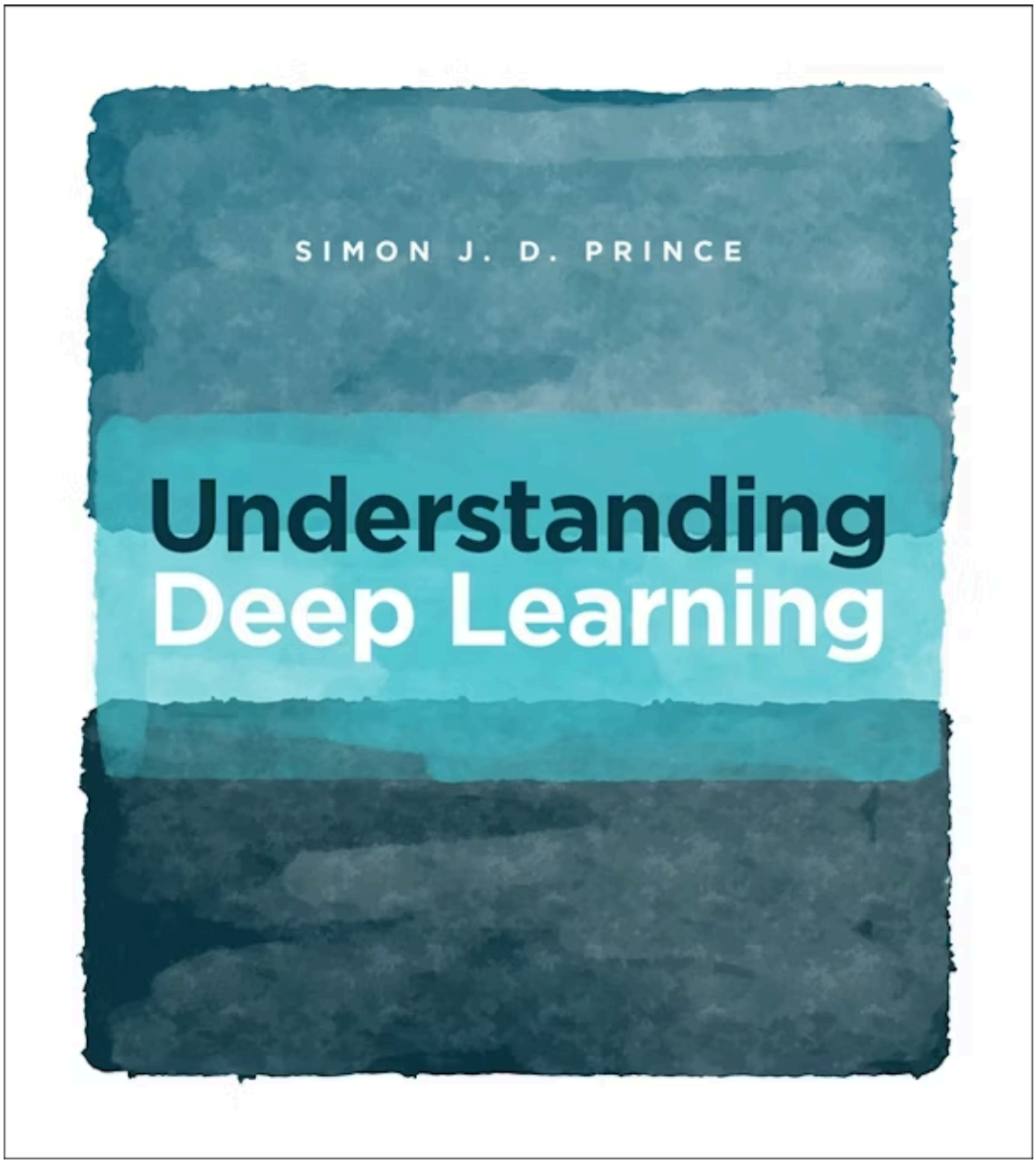
- **Locality and sharing:** the same kernel examines every local patch, producing a feature map with relatively few parameters.
- **Channels:** each kernel spans all input channels and produces one output feature map.
- **Geometry:** padding controls boundaries; stride and pooling control resolution and retained positional detail.

Summary (continued)

- **Hierarchy:** stacked layers compose simpler features and enlarge receptive fields.
- **Prediction:** convolution is translation equivariant; position aggregation can support invariant predictions, and a prediction head produces the final output.
- **Learning:** backpropagation learns every kernel, weight, and bias jointly.

The same computation extends from 1D sequences to 2D images and 3D volumes by adding spatial indices.

Further Reading

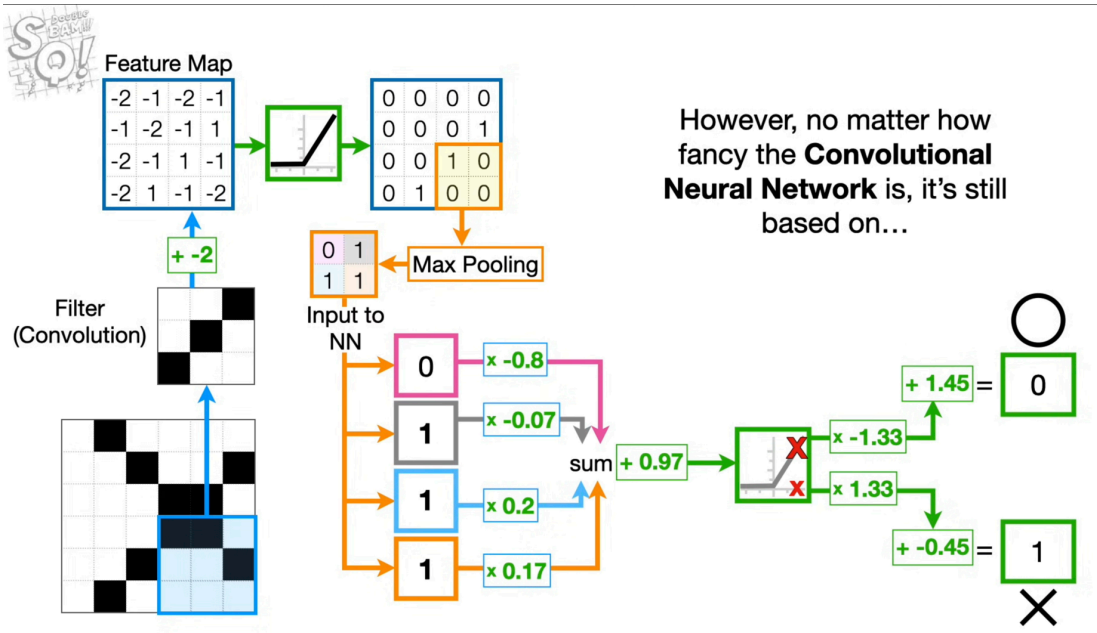


- Understanding Deep Learning (Prince 2023) is a recently published textbook focused on the foundational concepts of deep learning.
- It begins with fundamental principles and extends to contemporary topics such as transformers, diffusion models, graph neural networks, autoencoders, adversarial networks, and reinforcement learning.
- The textbook aims to help readers comprehend these concepts without delving excessively into theoretical details.
- It includes sixty-eight Python notebook exercises.
- The book follows a “read-first, pay-later” model.

Resources

- **A guide to convolution arithmetic for deep learning**
- Authors: Vincent Dumoulin and Francesco Visin
- Last revised: 11 Jan 2018
  - arXiv:1603.07285
  - GitHub Repository

StatQuest



The video presents a straightforward example that differentiates between images of the letter O and the letter X, utilizing a single filter for this purpose. This approach simplifies the explanation, making it easy to follow. In practical applications, however, each convolutional layer typically contains dozens or even hundreds of filters.

Next lecture

- We will introduce **search**.

References

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

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." In *Advances in Neural Information Processing Systems*, edited by F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger, vol. 25.

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

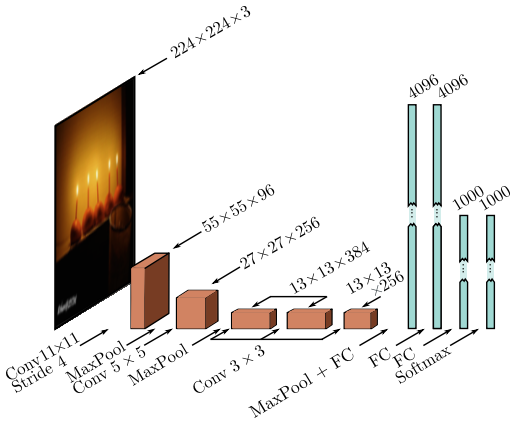
Prince, Simon J. D. 2023. *Understanding Deep Learning*. The MIT Press.
http://udlbook.com.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

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

Appendix: Well-known convolutional neural networks

AlexNet

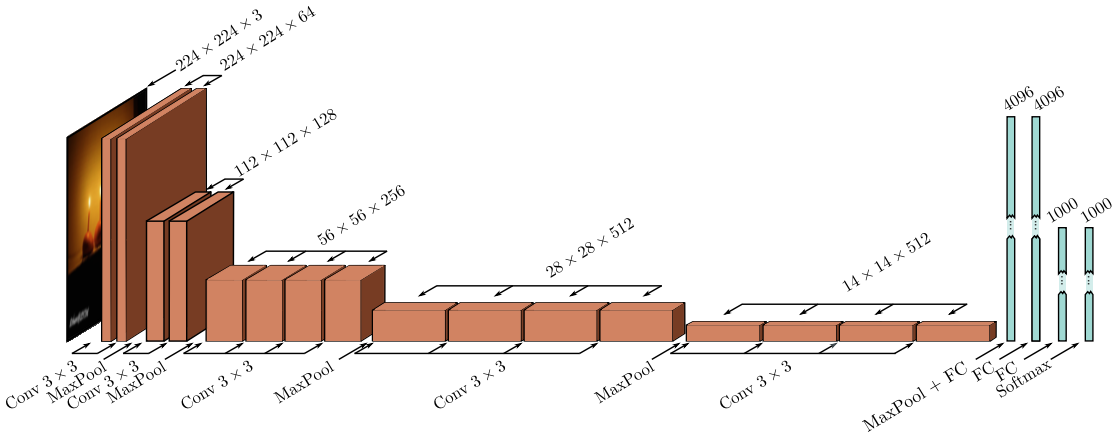


Krizhevsky et al. (2012)

Attribution: Prince (2023)

AlexNet consists of eight layers with learnable parameters: five convolutional layers followed by three fully connected layers. The architecture also includes max-pooling layers, ReLU activation functions, and dropout to improve training performance and reduce overfitting.

VGG



Simonyan and Zisserman (2015)

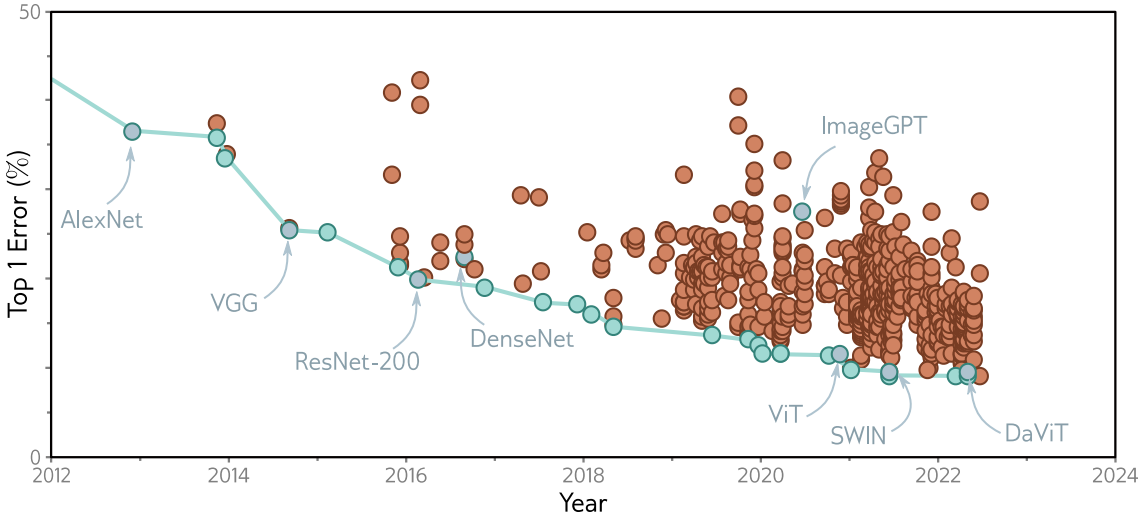
Attribution: Prince (2023)

Complementary information can be found here.

> Convolutional networks (ConvNets) currently set the state of the art in visual recognition. The aim of this project is to investigate how the ConvNet depth affects their accuracy in the large-scale image recognition setting.

> Our main contribution is a rigorous evaluation of networks of increasing depth, which shows that a significant improvement on the prior-art configurations can be achieved by increasing the depth to 16-19 weight layers, which is substantially deeper than what has been used in the prior art. To reduce the number of parameters in such very deep networks, we use very small 3×3 filters in all convolutional layers (the convolution stride is set to 1). Please see our publication for more details.

ConvNets Performance



Attribution: Prince (2023)

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa