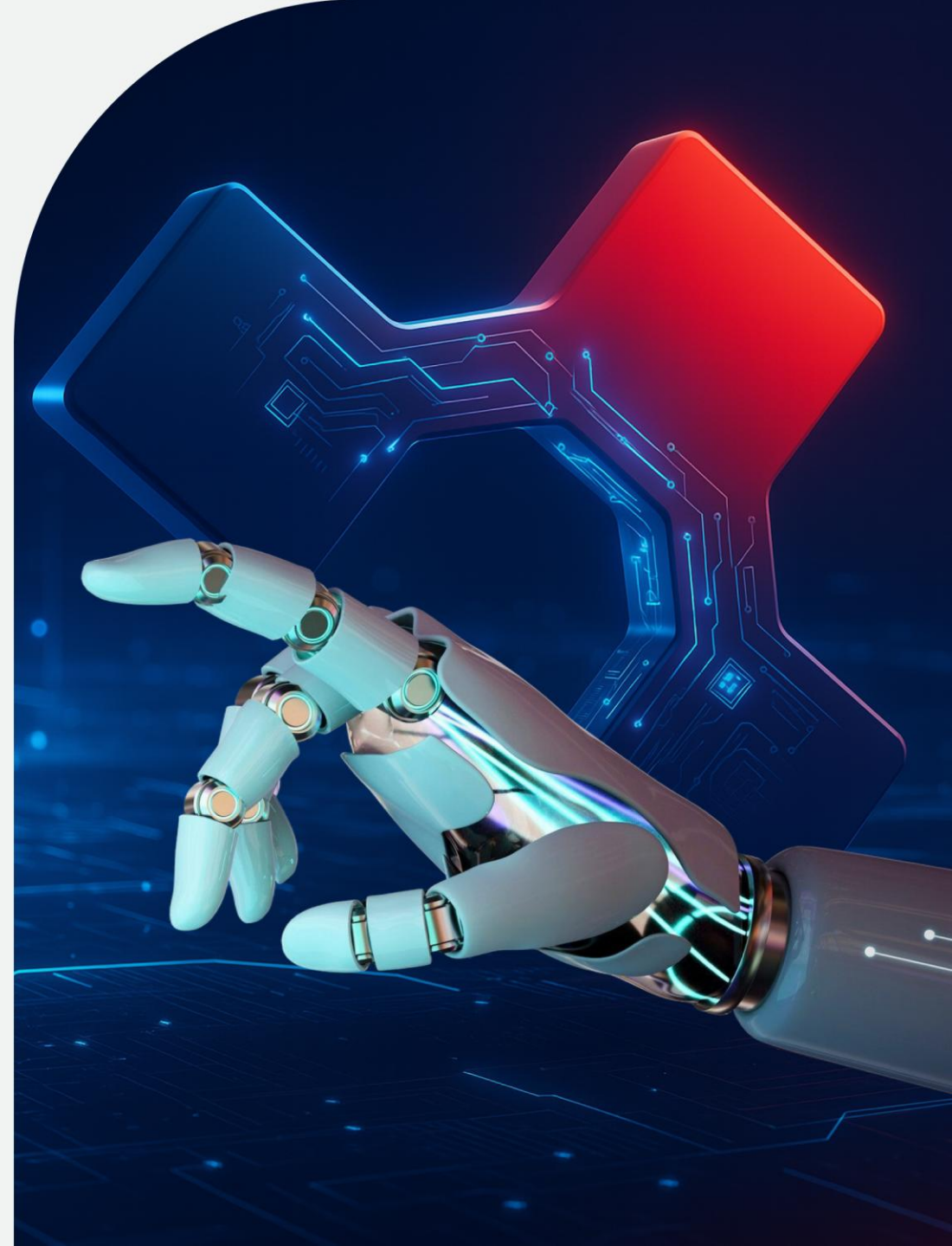




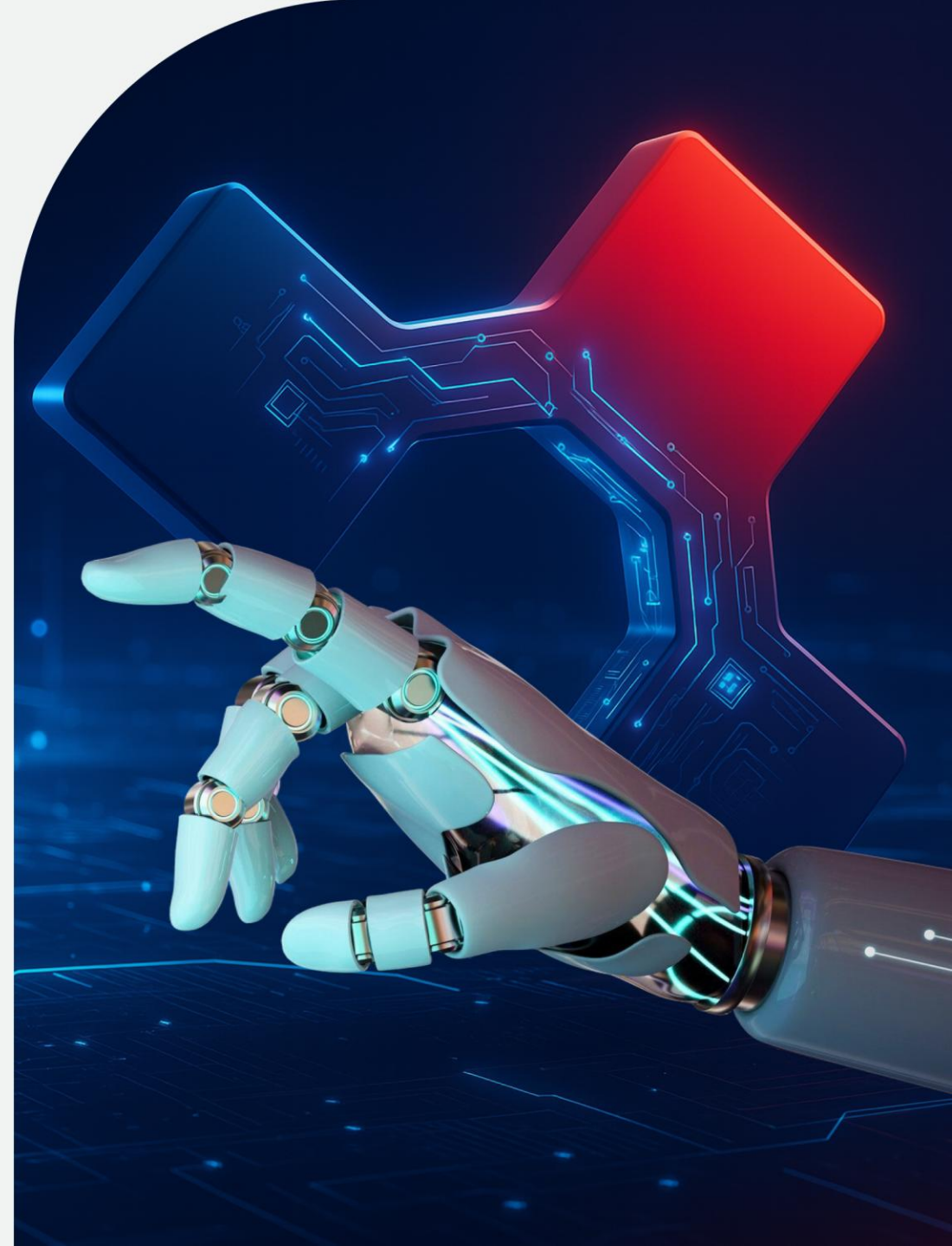
Núcleo de Capacitação em Inteligência Artificial





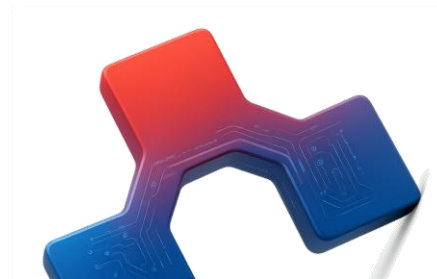
Deep Computer Vision Using Convolutional Neural Networks

The Architecture of the Visual Cortex, Convolutional Layers, Filters, Stacking Multiple Feature Maps, Implementing Convolutional Layers with PyTorch, Pooling Layers, Implementing Pooling Layers with PyTorch, CNN Architectures, LeNet-5, AlexNet, GoogLeNet, ResNet, Xception, SNet, Other Noteworthy Architectures, Choosing the Right CNN Architecture, GPU RAM Requirements: Inference Versus Training, Reversible Residual Networks (RevNets), Implementing a ResNet-34 CNN Using PyTorch, Using TorchVision's Pretrained Models, Pretrained Models for Transfer Learning, Classification and Localization, Object Detection, Fully Convolutional Networks, You Only Look Once, Object Tracking, Semantic Segmentation



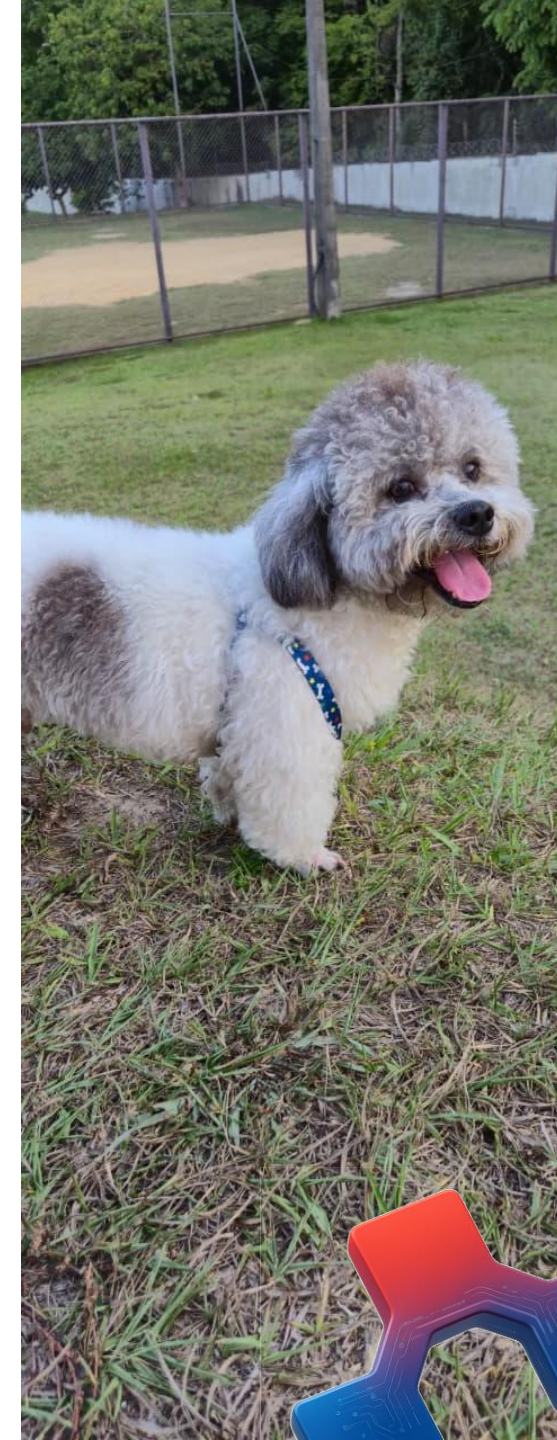


1. The Architecture of the Visual Cortex



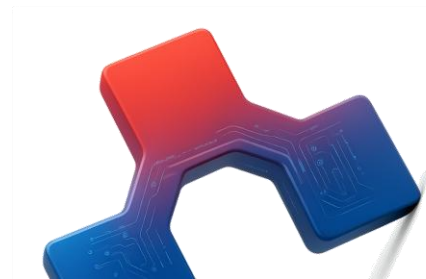


Embora computadores vençam humanos em xadrez desde 1996, tarefas “simples” para nós — como [ver um filhote numa foto](#) — só se tornaram confiáveis recentemente em computadores. A percepção humana acontece em [módulos sensoriais fora da consciência](#): quando percebemos algo, o sinal já chega com “[características](#)” de alto nível (ex.: “isso é um cachorro fofo”). Como não descrevemos passo a passo esse processo, fica claro que [percepção não é trivial](#). CNNs modelam computacionalmente esses módulos e, com mais poder de computação, grandes volumes de dados e técnicas de treino para redes profundas, alcançaram desempenho super-humano em várias tarefas visuais.





Veremos a [origem das CNNs](#), seus [blocos básicos](#) (camadas convolucionais e de pooling) e [como implementá-las](#) em PyTorch. Depois, revisaremos [arquiteturas consagradas](#) e duas tarefas além de classificação: [detecção de objetos](#) (múltiplas classes e caixas delimitadoras) e [segmentação semântica](#) (classificar cada pixel).

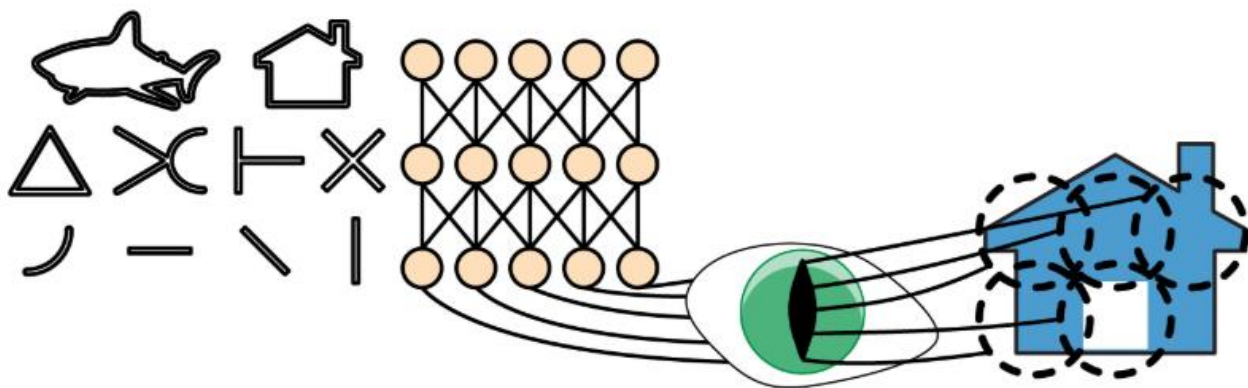


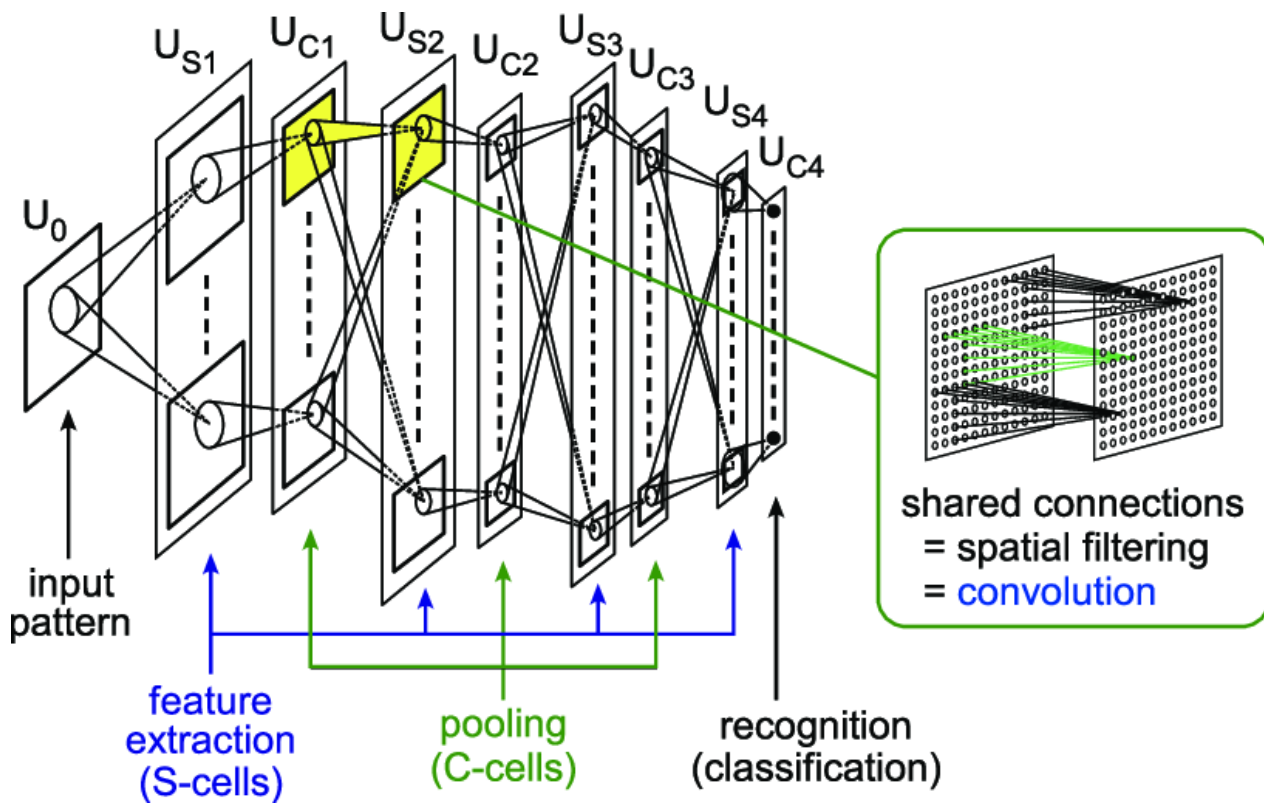




Experimentos de [Hubel & Wiesel](#) mostraram que neurônios do córtex visual possuem [campos receptivos locais](#): cada neurônio [responde a estímulos de uma pequena região do campo visual](#), e campos de vizinhos se sobrepõem. Em módulos sucessivos, campos receptivos ficam maiores e os neurônios respondem a padrões mais complexos (combinações de bordas e orientações).

Essa organização hierárquica motivou o desenho de redes que compõem padrões simples em estruturas complexas.



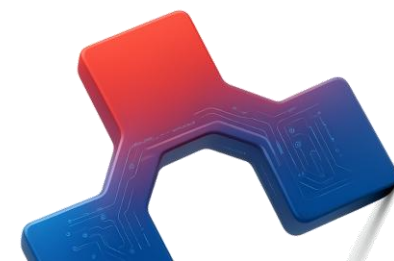
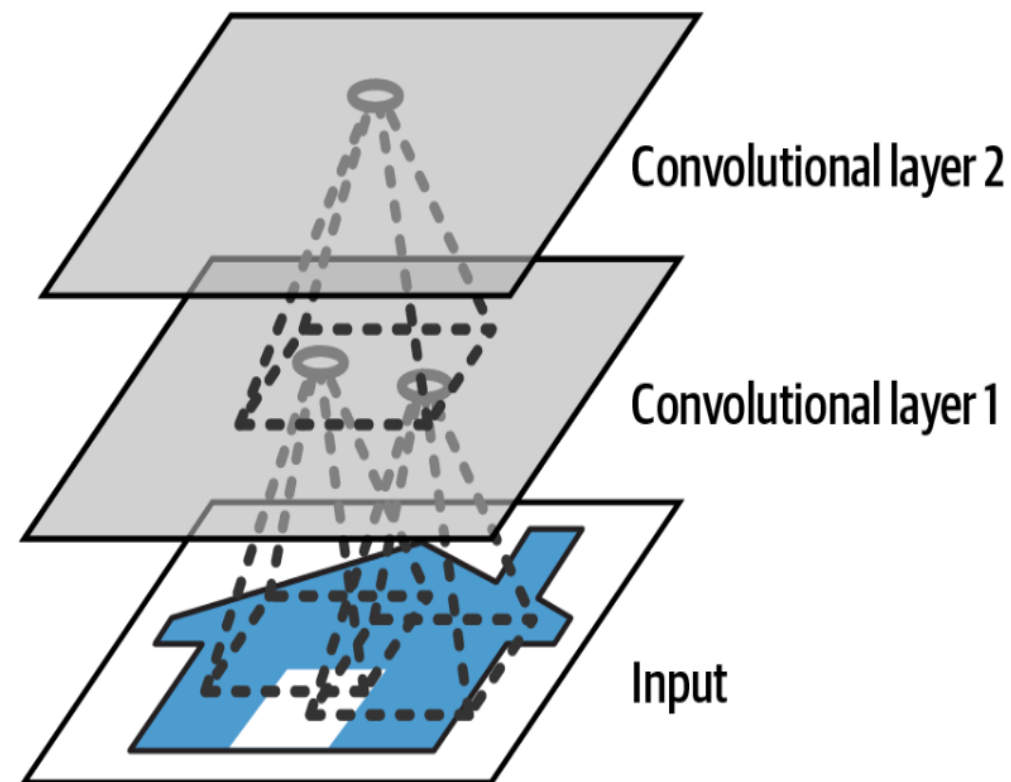


Os achados biológicos inspiraram o [Neocognitron](#) (1980), precursor das CNNs. Em 1998, LeCun et al. apresentaram a [LeNet-5](#), usada em bancos para leitura de dígitos em cheques. A arquitetura combinou blocos conhecidos (camadas totalmente conectadas, sigmoide) com dois blocos decisivos: camadas convolucionais e camadas de pooling.

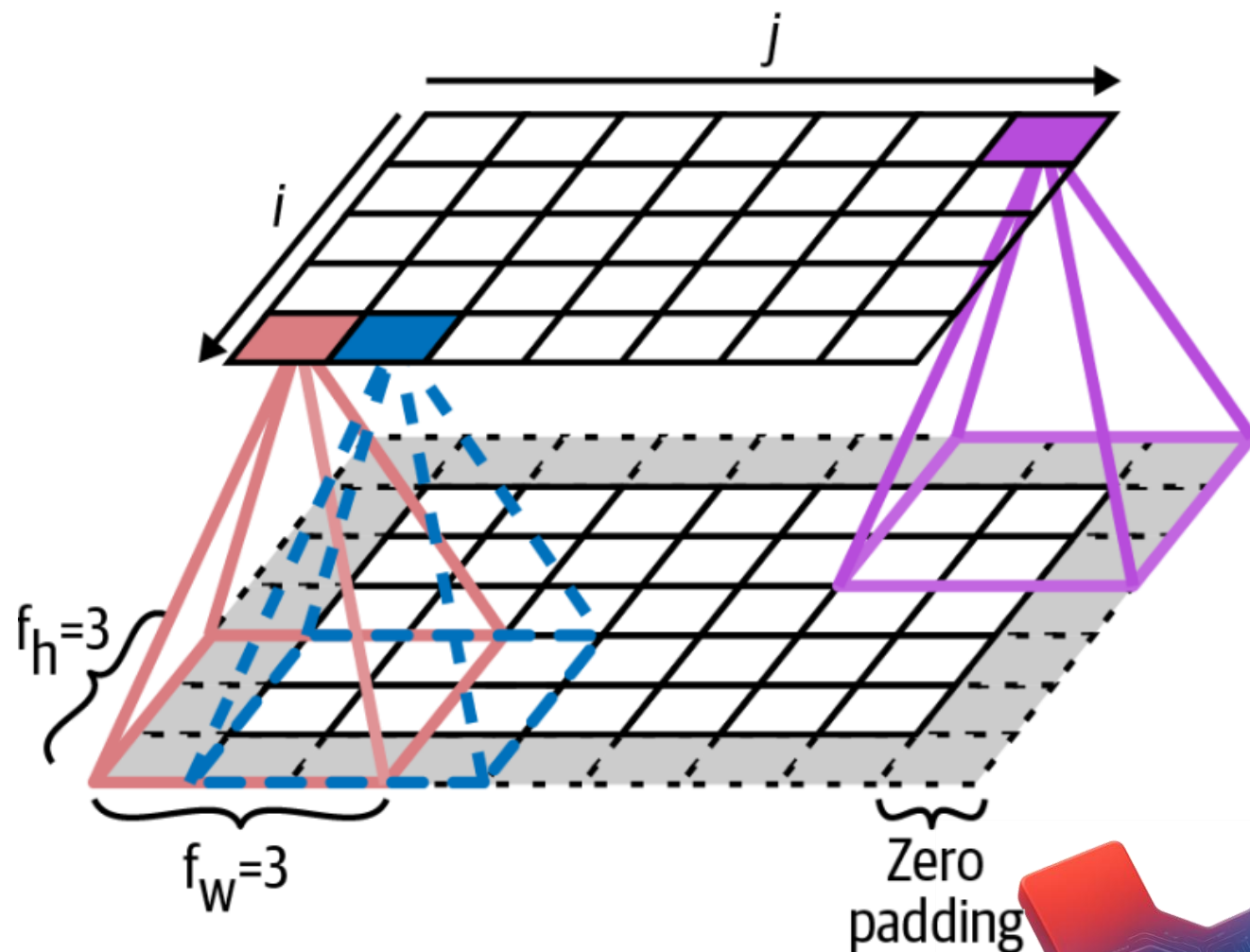


2. Convolutional Layers

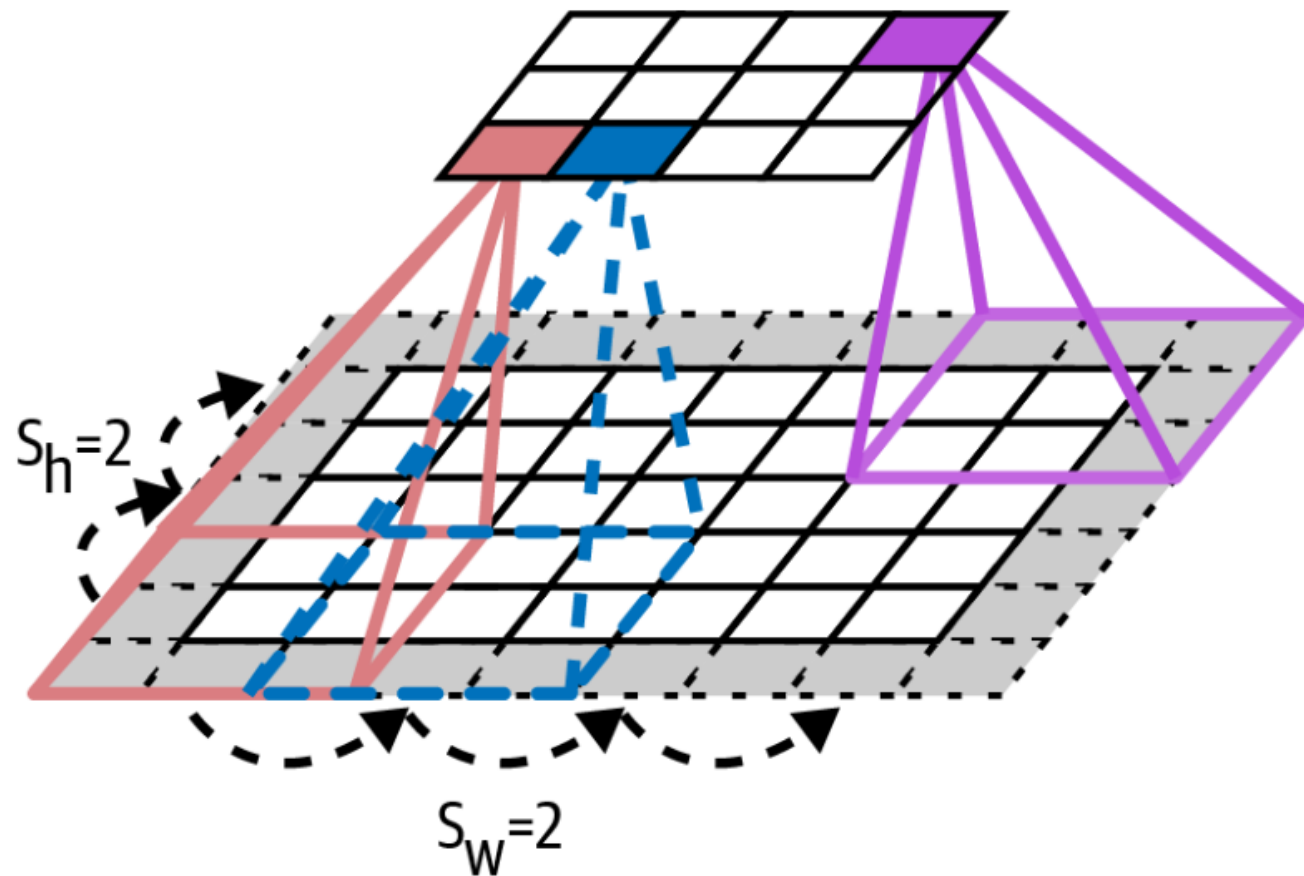
Diferente de camadas densas, a primeira camada convolucional conecta cada neurônio apenas aos pixels do seu campo receptivo. A camada seguinte conecta-se a pequenas janelas da anterior, e assim por diante. A hierarquia aprende primeiro padrões locais simples (bordas, texturas) e depois os compõe em padrões maiores (partes e objetos). Isso torna CNNs especialmente adequadas a imagens reais, repletas de objetos compostos.



Um neurônio na posição (i, j) conecta-se a uma região $f_h \times f_w$ da camada anterior (linhas i até $i + f_h - 1$; colunas j até $j + f_w - 1$). Para preservar altura e largura entre camadas, aplica-se **zero padding**: adiciona-se uma borda de zeros na entrada, evitando redução do mapa de ativação.

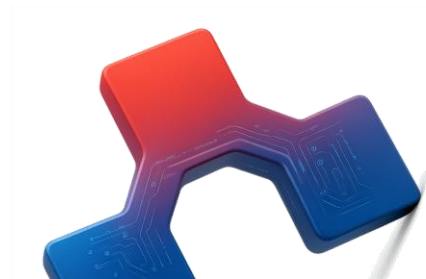


Para reduzir dimensionalidade e custo, espaça-se os campos receptivos usando **stride** (passo). Com **stride 2**, “pulam-se” posições, reduzindo a **resolução espacial**. Em geral, usa-se o mesmo stride vertical e horizontal ($s_h = s_w$). Formalmente, o neurônio em (i, j) conecta-se às **linhas** $i \times s_h$ até $i \times s_h + f_h - 1$ e **colunas** $j \times s_w$ até $j \times s_w + f_w - 1$ da camada anterior.



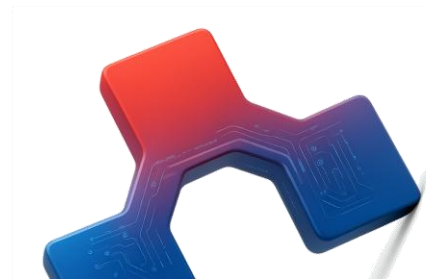
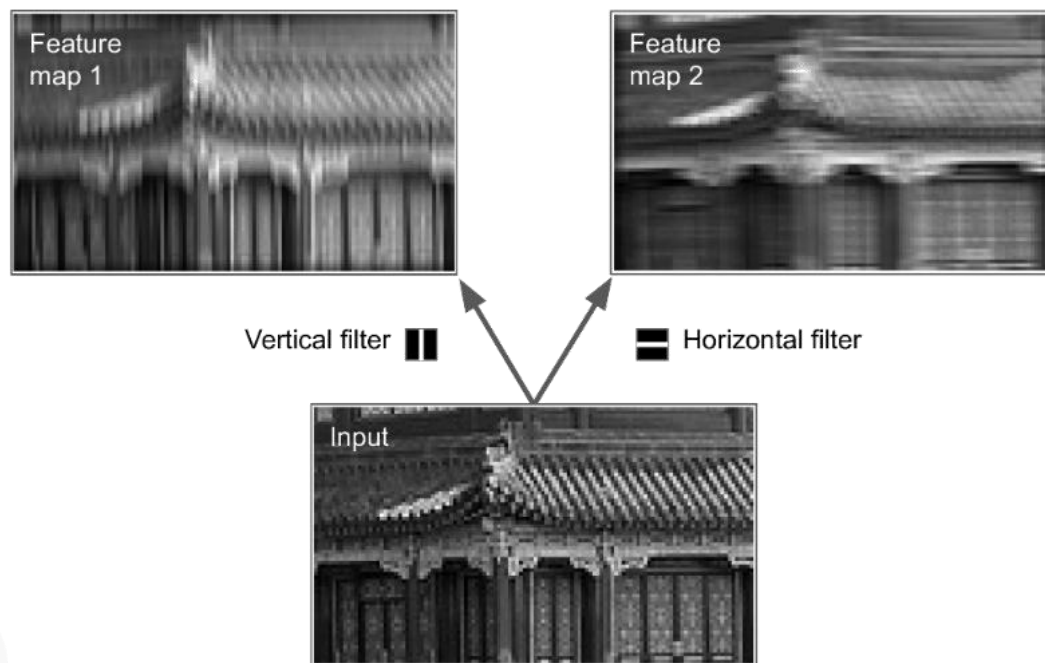


Localidade + compartilhamento de pesos criam detectores reutilizáveis e (quase) invariantes à posição; a composição hierárquica captura **estruturas cada vez mais complexas**. **Padding** controla o **tamanho espacial**; **stride** (e **pooling**) controla a **redução de resolução** e o **custo**. Resultado: modelos eficientes e expressivos para visão.

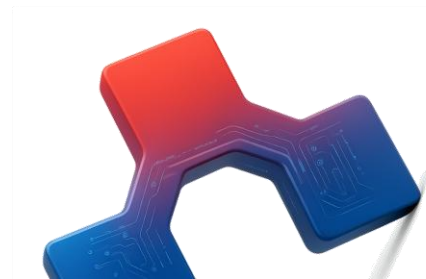
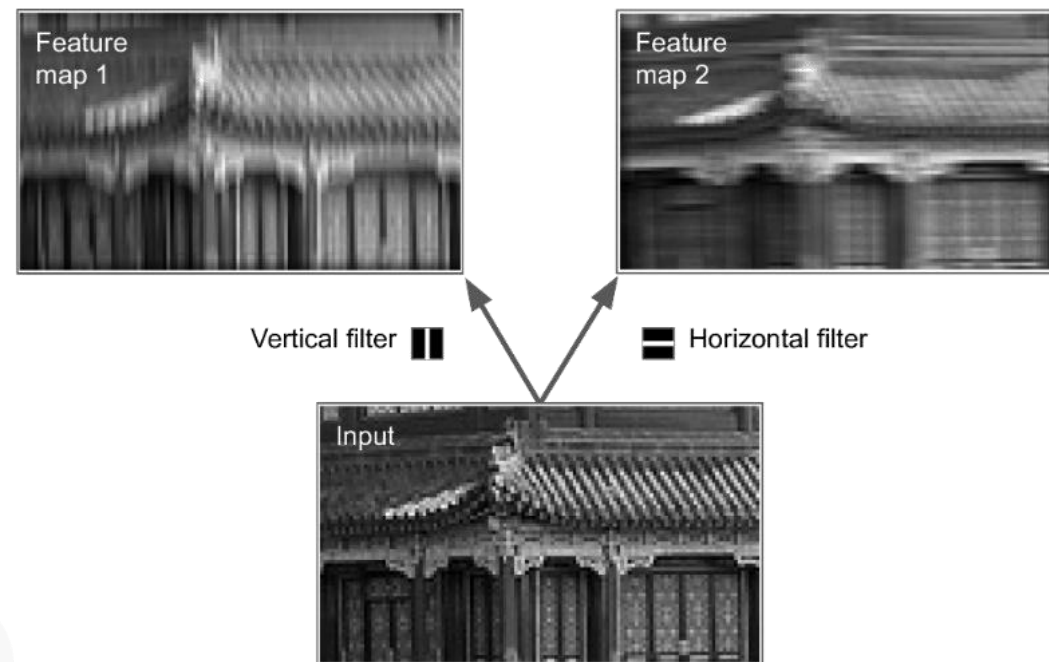


2.1 Filters (Kernel)

Os pesos de um neurônio na convolução podem ser vistos como uma “pequena imagem” do tamanho do campo receptivo — isto é, um **filtro** (também chamado de **kernel**). Exemplo: um **kernel 7×7** com a **coluna central = 1** e o **resto = 0** “presta atenção” apenas à **linha vertical central** do seu campo; outro kernel 7×7 com a **linha central = 1** e o **resto = 0** destaca **linhas horizontais**. Ao multiplicar elemento a elemento a janela da entrada pelo filtro e somar, o neurônio responde mais quando esse padrão está presente naquela região.

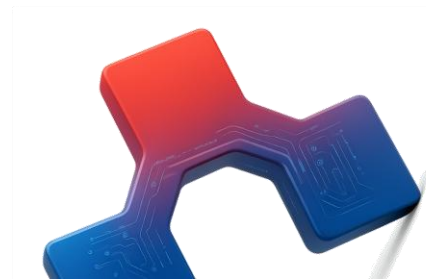


Se todos os neurônios de uma camada usarem o mesmo filtro (e o mesmo viés), varrendo toda a imagem, a saída é um mapa de características: um “mapa” que realça onde o padrão do filtro aparece. No exemplo, o filtro vertical produz um mapa que reforça linhas verticais e “apaga” o restante; o filtro horizontal faz o mesmo para linhas horizontais. Cada filtro gera um canal de saída; vários filtros → vários mapas (um por filtro).





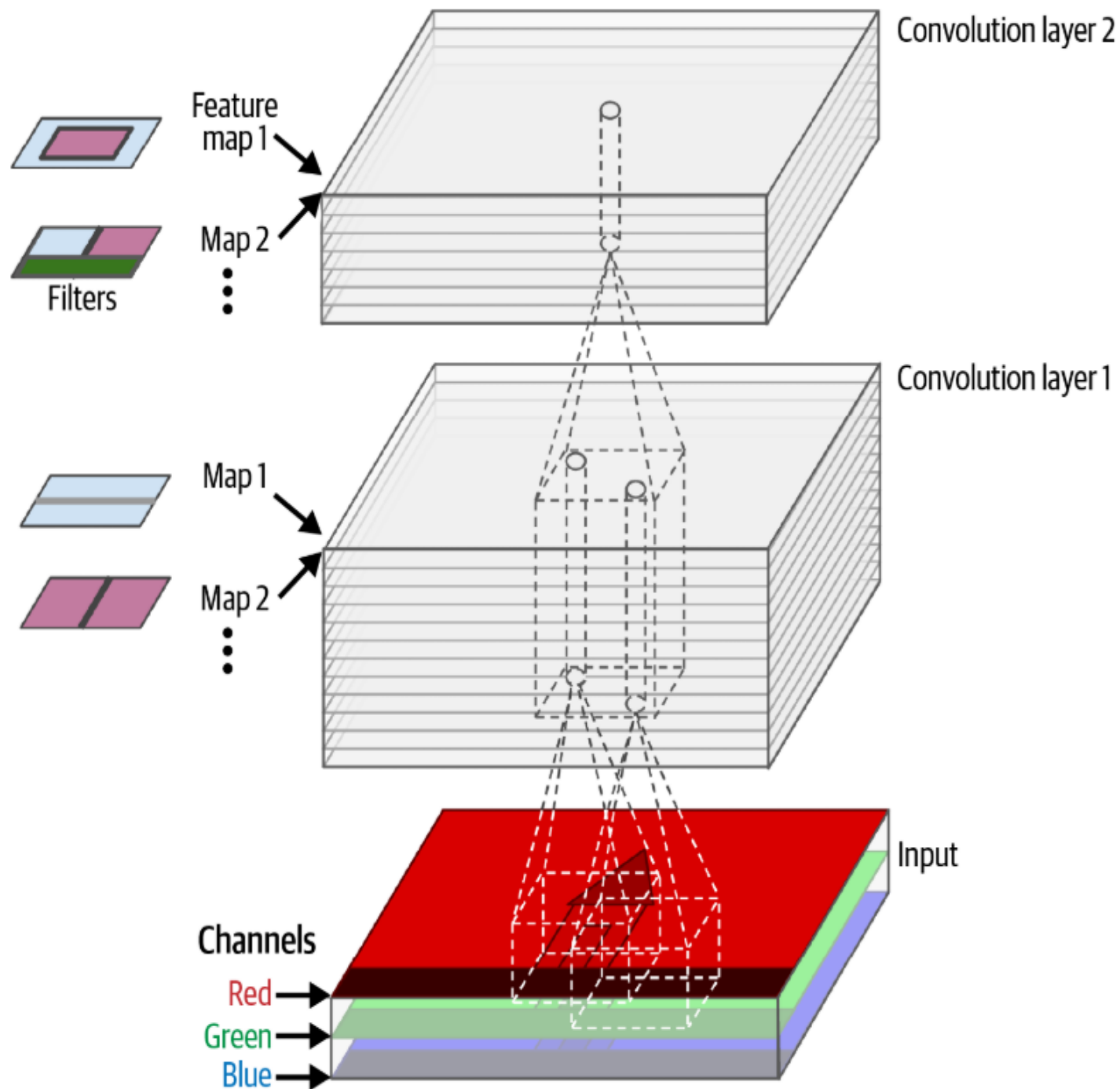
Embora possamos desenhar filtros “à mão” para entender a ideia (bordas horizontais, verticais, diagonais etc.), **na prática a rede aprende os filtros** durante o treinamento, **ajustando pesos** para maximizar o desempenho na tarefa. Camadas iniciais tendem a aprender **padrões simples** (bordas, texturas); camadas mais profundas **combinam** esses padrões em estruturas cada vez mais complexas (partes e objetos).





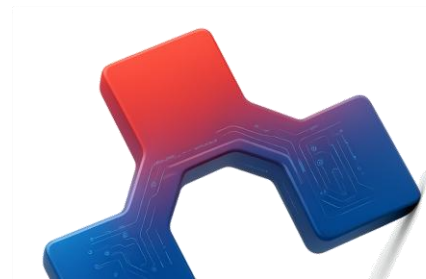
2.2 Stacking Multiple Feature Maps

Uma camada convolucional **usa vários filtros** (você escolhe quantos) e produz **um mapa de características por filtro**. Portanto, sua saída é melhor vista como um **tensor 3D: altura × largura × n° de mapas** (canais). Em cada mapa, há **um neurônio por pixel**; neurônios **do mesmo mapa** compartilham **os mesmos pesos e viés** (o mesmo kernel), enquanto mapas diferentes têm parâmetros diferentes. Assim, uma única camada consegue detectar **múltiplos padrões** em qualquer posição da entrada.



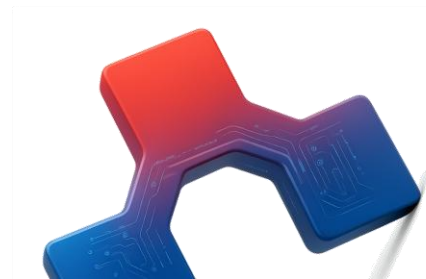



A entrada também é 3D: **um canal por banda**. Em imagens comuns, são **3 canais RGB**; imagens em tons de cinza têm **1 canal**; já aplicações específicas (ex.: sensoriamento remoto) podem ter **muitos canais** (infravermelho, etc.). **Importante:** o **campo receptivo de um neurônio** na camada conv **se estende por todos os canais** da camada anterior. Ou seja, um único filtro tem tamanho $fh \times fw \times C_{in}$ (agrega espacialmente e entre canais).





Para o neurônio na posição (i, j) do mapa k na camada l , a janela na camada $l-1$ cobre linhas $i \times sh$ até $i \times sh + fh - 1$ e colunas $j \times sw$ até $j \times sw + fw - 1$, atravessando todos os mapas (canais) da camada anterior. Repare que, dentro da mesma camada, neurônios em (i, j) de mapas diferentes usam a mesma região espacial da entrada — o que muda são os pesos (os filtros) de cada mapa.

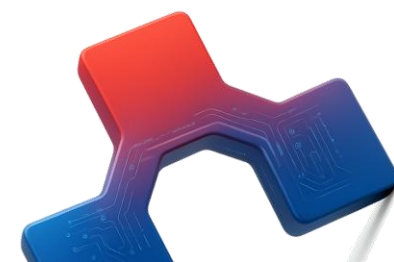




A abaixo apenas formaliza o que já descrevemos: o valor de saída $z_{i,j,k}$ é a **soma ponderada** de todos os elementos da janela 3D da entrada ($f_h \times f_w \times n^\circ$ de canais anteriores), **mais** o viés b_k . Em notação: somamos sobre deslocamentos da janela (u, v), sobre os canais anteriores k' , multiplicando cada entrada $x_{i',j',k'}$ pelo peso correspondente $w_{u,v,k',k}$, e ao final somamos b_k . É “feio” por causa dos índices, mas é simplesmente **(entrada \times pesos) + viés**, varrendo a janela.

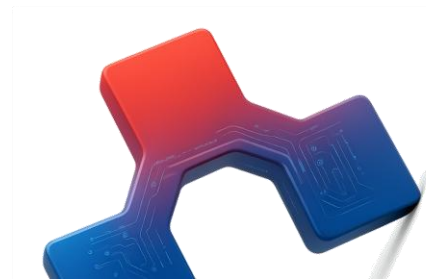
$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \quad \text{with} \quad \begin{aligned} i' &= i \times s_h + u \\ j' &= j \times s_w + v \end{aligned}$$

Para cada posição (i, j) e para cada filtro k , a conv pega **um patch** da entrada (definido pelo **stride**), multiplica **cada valor do patch** (em **todos os canais**) pelo **peso correspondente do kernel**, **soma tudo** e **adiciona o viés** — o resultado é $z_{i,j,k}$.



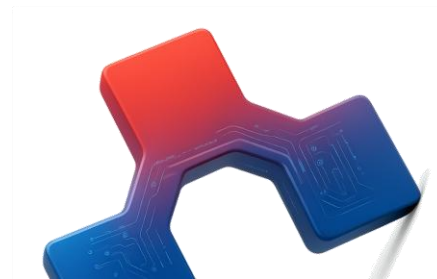


Uma camada convolucional **aplica vários filtros treináveis em paralelo**, cada um produzindo um mapa que realça um tipo de padrão (bordas, texturas, orientações, cores...). As camadas seguintes **combinam** esses mapas para compor padrões mais complexos. Na implementação em PyTorch (que virá nas próximas células), você define **quantos filtros** (isto é, out_channels) quer aprender; o framework cuida de aplicar cada kernel em **todos os canais de entrada** e empilhar os **mapas de saída**.





2.3 Implementing Convolutional Layers with PyTorch

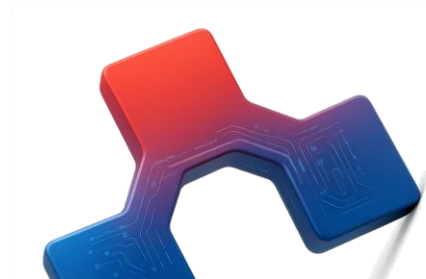




O código carrega duas imagens de exemplo com `load_sample_images()`. Elas vêm como `uint8` (0–255). Em seguida: empilha (`np.stack`), converte para tensor `float32` (`torch.tensor`) e normaliza para 0–1 (divide por 255). A forma impressa é `[2, 427, 640, 3]`: 2 imagens, altura 427, largura 640, 3 canais (RGB). Objetivo: padronizar tipo e escala de pixel para alimentar as camadas da rede.

```
import numpy as np
import torch
from sklearn.datasets import load_sample_images

sample_images = np.stack(load_sample_images()["images"])
sample_images = torch.tensor(sample_images, dtype=torch.float32) / 255
```





PyTorch usa **NCHW** (lote, canais, altura, largura). Reorganizamos $[N, H, W, C] \rightarrow [N, C, H, W]$ com `permute`, obtendo `[2, 3, 427, 640]`. Depois, aplicamos `CenterCrop((70, 120))` para recortar o centro e padronizar o tamanho espacial para `[2, 3, 70, 120]`.

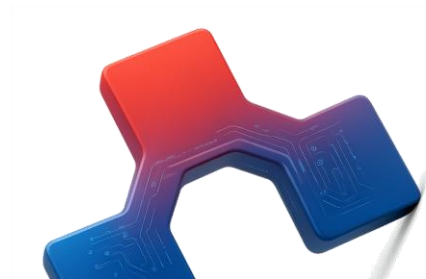
```
sample_images_permuted = sample_images.permute(0, 3, 1, 2)
sample_images_permuted.shape
```

```
torch.Size([2, 3, 427, 640])
```

```
import torchvision
import torchvision.transforms.v2 as T
```

```
cropped_images = T.CenterCrop((70, 120))(sample_images_permuted)
cropped_images.shape
```

```
torch.Size([2, 3, 70, 120])
```





Criamos `nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7)` e aplicamos em `[2, 3, 70, 120]`. A saída é `[2, 32, 64, 114]`: trocamos 3 canais RGB por 32 mapas de características (um por filtro) e a altura/largura encolhem 6 px (3 por lado), pois o padrão é sem padding (“valid”).

```
import torch.nn as nn
```

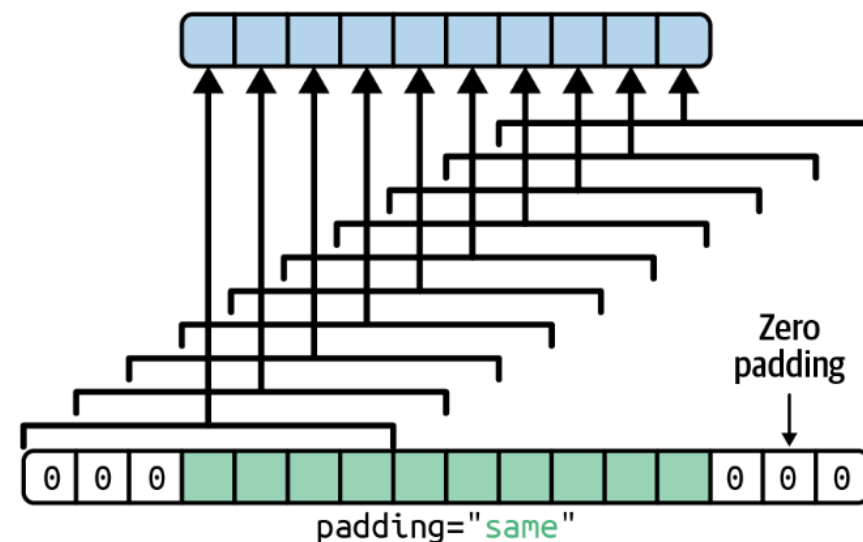
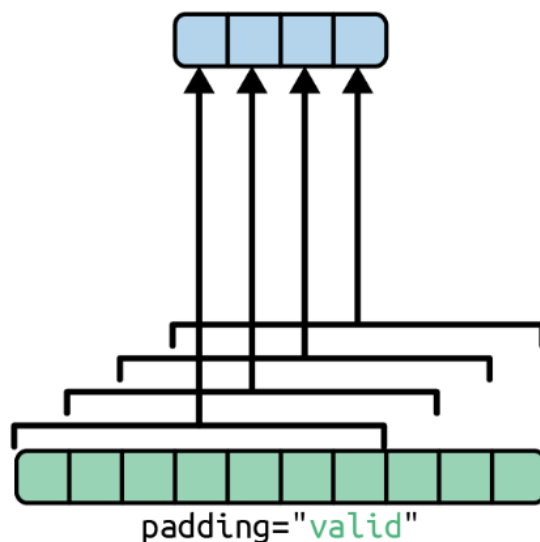
```
torch.manual_seed(42)
```

```
conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7)
```

```
fmaps = conv_layer(cropped_images)
```

```
fmaps.shape
```

```
torch.Size([2, 32, 64, 114])
```



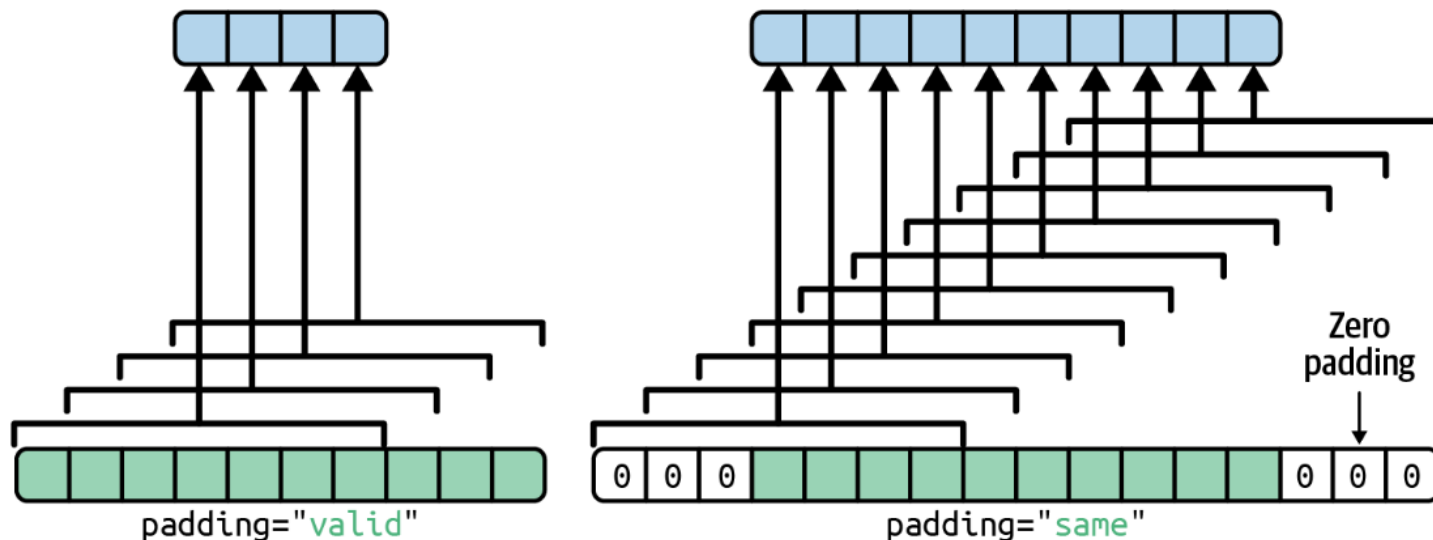


Recriando a camada com `padding="same"`, a saída mantém a mesma altura e largura da entrada: `[2, 32, 70, 120]`. Intuição: zeros nas bordas permitem “varrer” até os cantos sem perder pixels.

```
conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7,  
                        padding="same")  
fmaps = conv_layer(cropped_images)
```

```
fmaps.shape
```

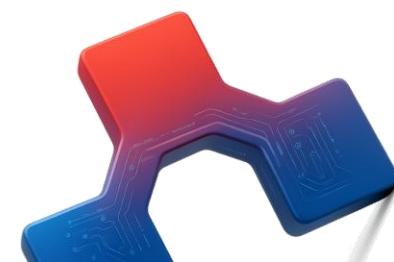
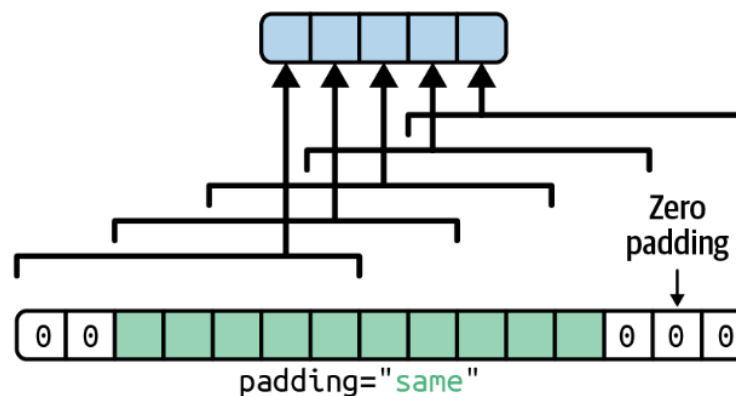
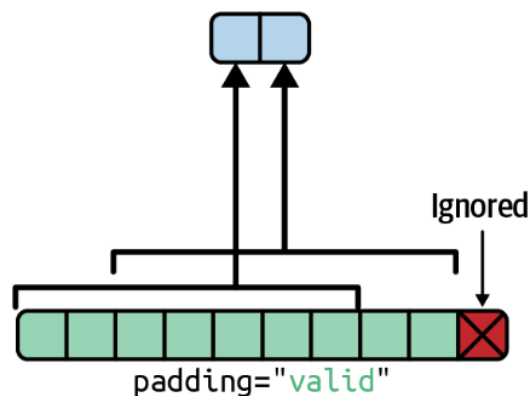
```
torch.Size([2, 32, 70, 120])
```



Com `stride=2`, a janela “pula” posições e a saída reduz ~metade em cada dimensão. Exemplo: entrada 70×120, com `kernel=7` e `padding=3` → saída 35×60. Observação: `padding="same"` não é permitido com `stride>1` no PyTorch para evitar excesso de zeros.

```
conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7, stride=2,
                        padding=3)
fmaps = conv_layer(cropped_images)
fmaps.shape
```

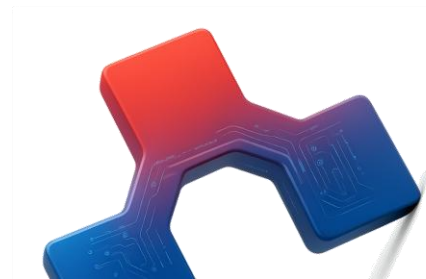
```
torch.Size([2, 32, 35, 60])
```





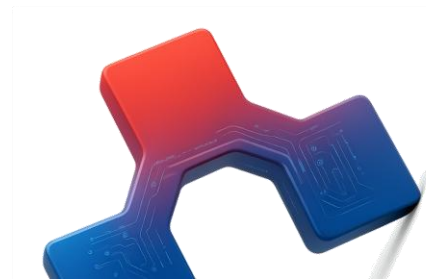
Os pesos têm formato `[out_channels, in_channels, kernel_h, kernel_w]` e os vieses têm formato `[out_channels]`. H e W da imagem não aparecem nos pesos porque o mesmo filtro é aplicado em todas as posições (compartilhamento de pesos). A camada aceita qualquer tamanho `espacial \geq kernel`, desde que o `nº de canais de entrada seja correto`.

```
conv_layer.weight.shape  
torch.Size([32, 3, 7, 7])  
  
conv_layer.bias.shape  
torch.Size([32])
```





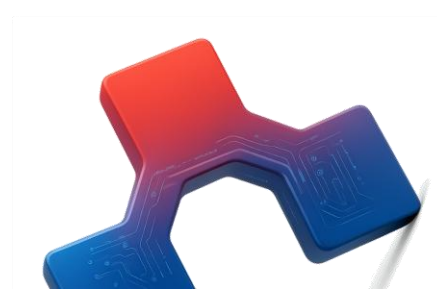
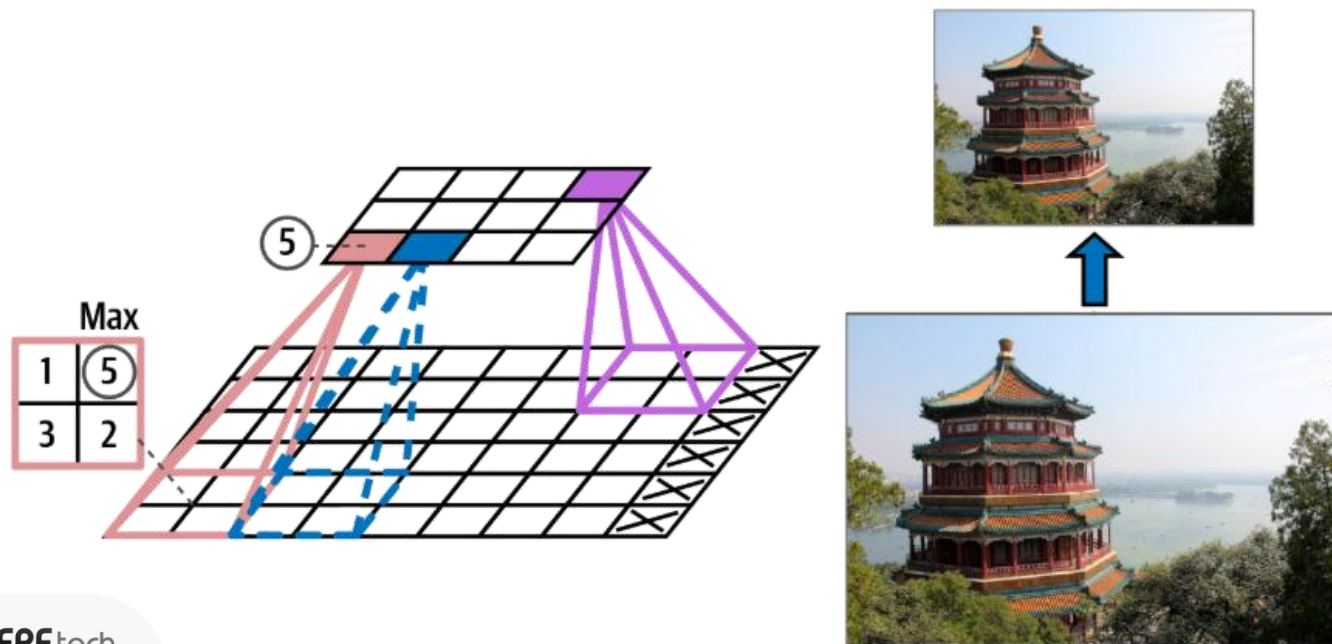
Convolução é **linear**; empilhar várias sem ativação equivale a uma única conv (baixa expressividade). Por isso, **inclua uma ativação** (ex.: ReLU) após cada conv. Para **ReLU**, use inicialização de **He/Kaiming**; **vieses podem ser zerados**.





3. Pooling Layers

Camadas de pooling fazem **subamostragem** da entrada para reduzir custo computacional, uso de memória e número de parâmetros (menor risco de overfitting). Assim como na convolução, cada neurônio vê um **campo receptivo** (definimos tamanho do kernel, stride e padding). **Diferença-chave:** pooling **não tem pesos nem vieses**; apenas agrega valores (ex.: **máximo** ou **média**).

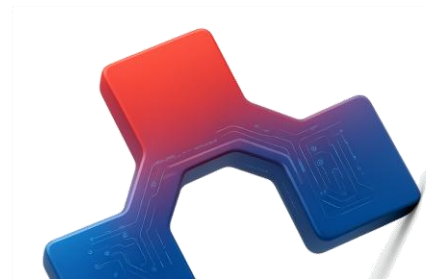




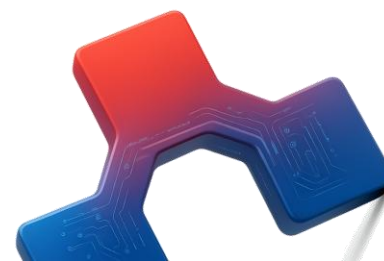
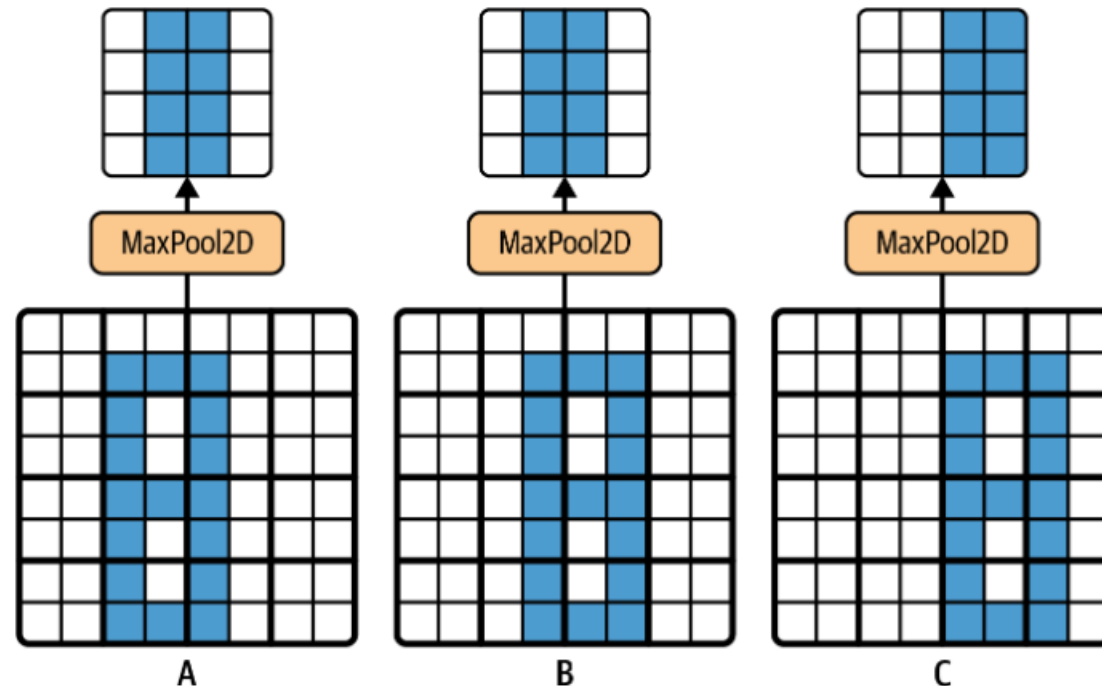
4. Implementing Pooling Layers with PyTorch,

Com **kernel 2×2** e **stride 2**, cada bloco 2×2 gera **um** número: o **máximo**. Isso reduz a **altura** e **largura** pela metade (sem padding). Ex.: bloco com {1, 5, 3, 2} → sai **5**. Resultado: mapas menores e com as ativações mais fortes preservadas.

```
max_pool = nn.MaxPool2d(kernel_size=2)
```

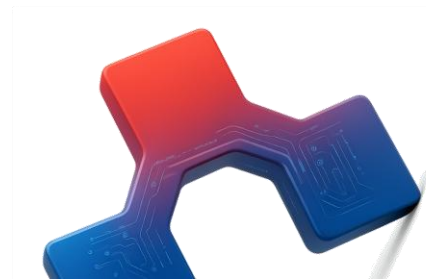


Pooling opera **independentemente em cada canal**: a **profundidade** (nº de canais) se mantém. Além da redução de custo, **max pooling** traz certa **invariância a pequenas translações**: deslocar levemente o padrão pode não mudar a saída após o pooling (ver A→B). Deslocamentos um pouco maiores mudam a saída, mas às vezes **apenas deslocam** o resultado (A→C), sugerindo invariância parcial. Isso é útil em **classificação**, quando detalhes de posição não deveriam afetar a classe





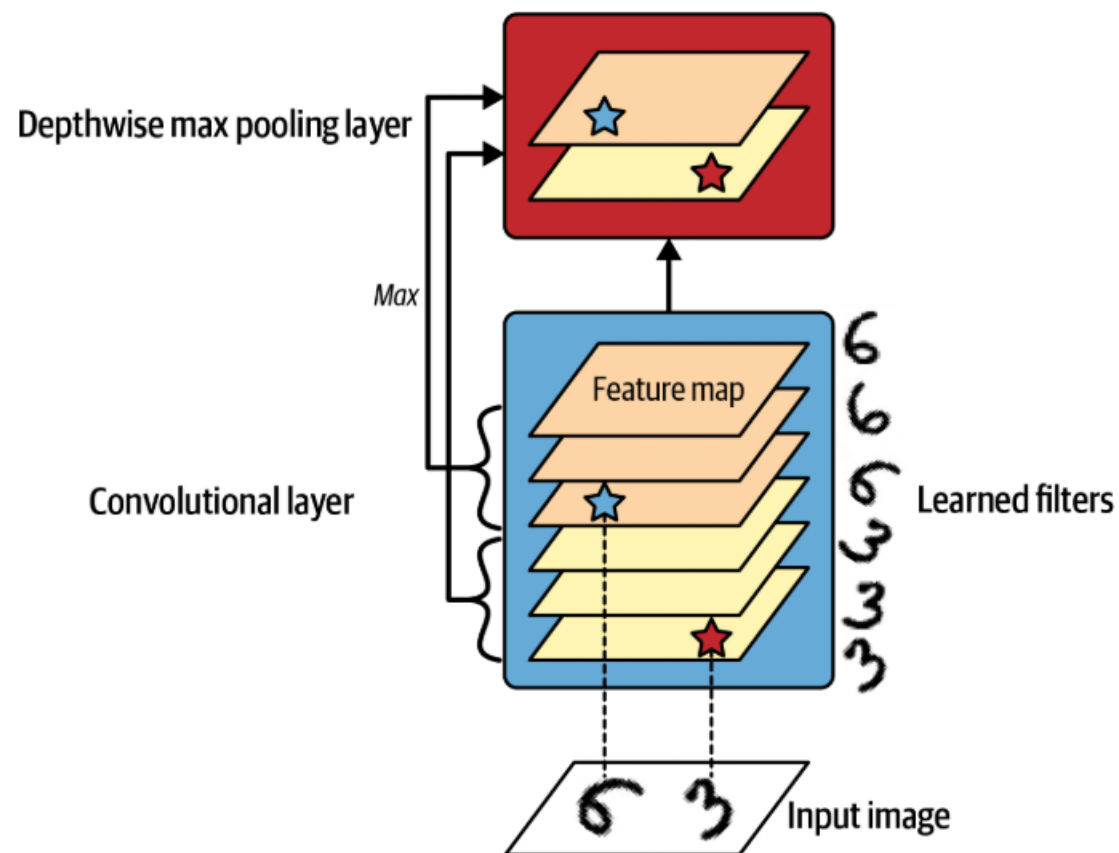
Desvantagens: mesmo com 2×2 , stride 2, a área reduz $4 \times$ (perde-se 75% dos valores). Em tarefas onde a saída deve **mover-se junto** com a entrada (ex.: **segmentação semântica**), queremos **equivariância**, não invariância. Nesses casos, usar menos pooling (ou alternativo) pode ser preferível.





Max pooling costuma performar melhor que **average pooling** em prática: mesmo perdendo mais informação que a média, ele **preserva as ativações mais fortes** (limpando sinais fracos/ruído), dá invariância mais forte e é **ligeiramente mais barato**. **Average** era comum no passado; hoje é usado mais pontualmente.

Também é possível fazer pooling ao longo da **profundidade** (entre filtros), para **aprender invariâncias a variações** (ex.: rotação, espessura, brilho, cor etc.). A ideia: vários filtros detectam **variações** do mesmo padrão; o **max pooling na profundidade** escolhe o mais forte, tornando a saída **invariante à variação** (ex.: rotação de dígitos).



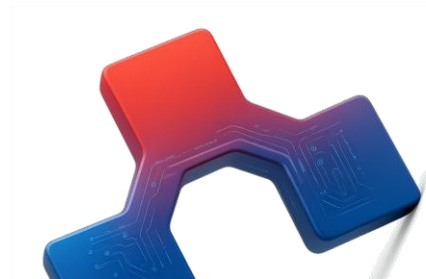


PyTorch não tem camada pronta de [depthwise max pooling](#). Abaixo, um módulo customizado usando [F.max_pool1d](#) que agrupa ao longo da dimensão de [canais](#).

```
import torch.nn.functional as F

class DepthPool(torch.nn.Module):
    def __init__(self, kernel_size, stride=None, padding=0):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding

    def forward(self, inputs):
        batch, channels, height, width = inputs.shape
        Z = inputs.view(batch, channels, height * width) # merge spatial dims
        Z = Z.permute(0, 2, 1) # switch spatial and channels dims
        Z = F.max_pool1d(Z, kernel_size=self.kernel_size, stride=self.stride,
                        padding=self.padding) # compute max pool
        Z = Z.permute(0, 2, 1) # switch back spatial and channels dims
        return Z.view(batch, -1, height, width) # unmerge spatial dims
```

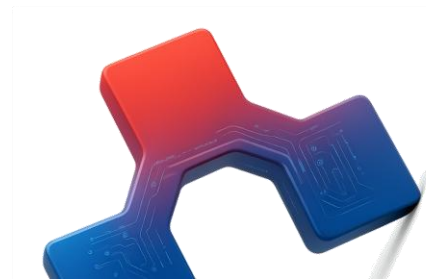




Suponha entrada $[2, 32, 70, 120]$ (B, C, H, W), $\text{kernel_size}=4$, $\text{stride}=4$, $\text{padding}=0$.

1. Une H×W: $[2, 32, 8400]$ ($70 \times 120 = 8400$)
2. Permuta para $[2, 8400, 32]$
3. `max_pool1d` ao longo de $C=32$ com $\text{kernel}/\text{stride}=4 \rightarrow [2, 8400, 8]$
4. Permuta de volta: $[2, 8, 8400]$
5. Reparte espacial: $[2, 8, 70, 120]$

Resultado: mesmos H×W, **canais reduzidos** ($32 \rightarrow 8$) por pooling na profundidade.



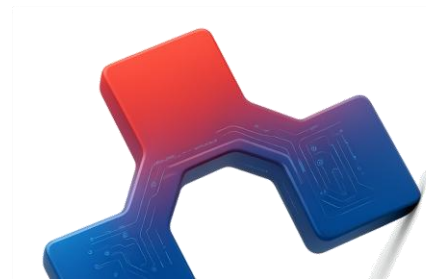


Global average pooling calcula a **média de cada mapa inteiro**, produzindo **um número por mapa** e por instância. É altamente destrutivo, mas muito útil **antes da camada de saída** (reduz parâmetros em *classification heads* modernas). Use **AdaptiveAvgPool2d** para não depender do tamanho exato da entrada.

```
global_avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
output = global_avg_pool(cropped_images)
```

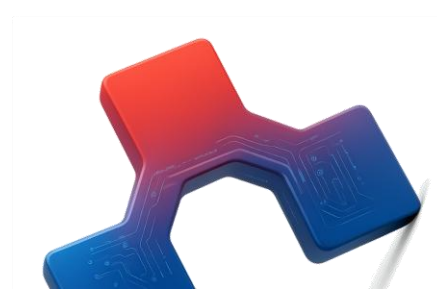
Alternativa equivalente:

```
output = cropped_images.mean(dim=(2, 3), keepdim=True)
```



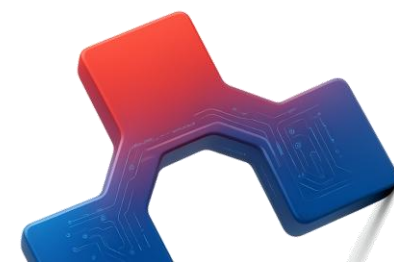
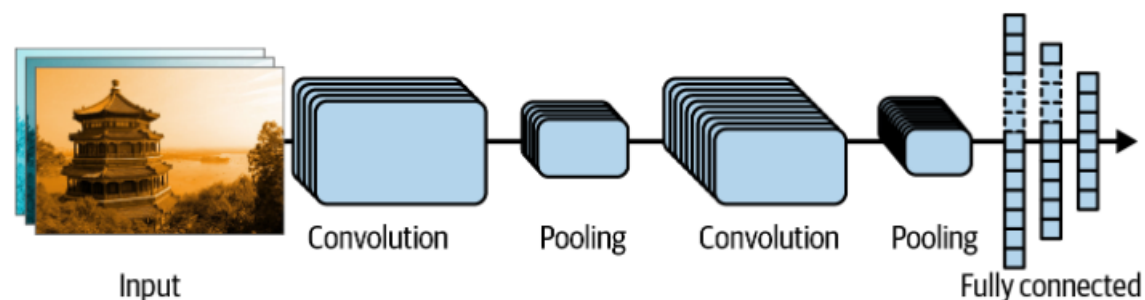


Você agora tem os blocos básicos: **Conv2d + ativação** (ReLU), **Pooling** (max/avg, inclusive global) e, quando necessário, pooling **na profundidade** customizado. A seguir, veremos **como combiná-los** em arquiteturas completas de CNNs e padrões práticos (ordem das camadas, escolha de hiperparâmetros e cabeças de classificação).



5. CNN Architectures

CNNs típicas empilham **várias convoluções + ReLU**, depois um **pooling**, repetem esse bloco algumas vezes (a imagem vai ficando **menor** em $H \times W$ e **mais profunda** em n° de mapas), e no topo conectam um **perceptron multicamada (MLP)** com camadas **densas + ReLU** até a **camada de saída** (ex.: Softmax para probabilidades, ou logits + CrossEntropyLoss).



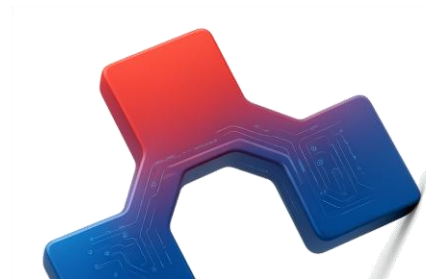


Em vez de 1 conv 5×5 , prefira 2 conv 3×3 :

- Menos parâmetros e menos FLOPs.
- Mais não linearidades (duas ReLUs), aumentando a capacidade.

Exceção comum: a primeira conv pode usar kernel maior (5×5 ou 7×7) e $\text{stride} \geq 2$ para reduzir a resolução cedo sem custo excessivo (entrada costuma ter 3 canais).

Modelo sequencial que alterna conv+ReLU e MaxPool (reduz $H \times W$), dobrando filtros após cada pooling ($64 \rightarrow 128 \rightarrow 256$). No topo: Flatten \rightarrow Dense(128) + ReLU + Dropout(0.5) \rightarrow Dense(64) + ReLU + Dropout(0.5) \rightarrow Dense(10). Como treinaremos com CrossEntropyLoss, deixamos a saída como logits (sem Softmax explícito).

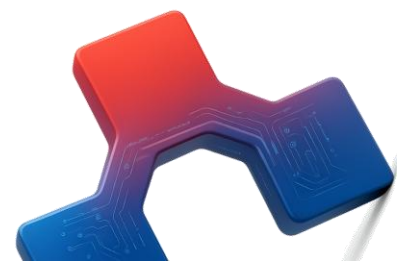




Ajudantes: `functools.partial` define `DefaultConv2d` com `kernel_size=3` e `padding="same"` para evitar repetição. Primeira conv usa `kernel 7` (maior) com `64` filtros.

```
from functools import partial

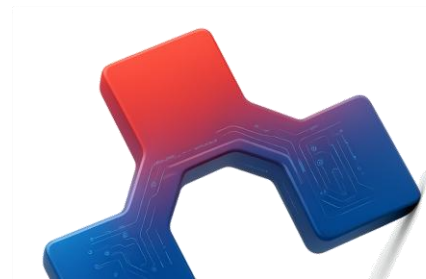
DefaultConv2d = partial(nn.Conv2d, kernel_size=3, padding="same")
model = nn.Sequential(
    DefaultConv2d(in_channels=1, out_channels=64, kernel_size=7), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=64, out_channels=128), nn.ReLU(),
    DefaultConv2d(in_channels=128, out_channels=128), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=128, out_channels=256), nn.ReLU(),
    DefaultConv2d(in_channels=256, out_channels=256), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),
    nn.Linear(in_features=2304, out_features=128), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=128, out_features=64), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=64, out_features=10),
).to(device)
```





Pooling divide $H \times W$ por 2 (por dimensão) \rightarrow área cai $4\times$. Para manter/elevar capacidade sem explodir custo, dobramos filtros depois de cada pooling (p.ex., $64 \rightarrow 128 \rightarrow 256$). Intuição: poucos padrões de baixo nível (bordas, texturas) combinam-se em muitos padrões de alto nível (partes/objetos), logo faz sentido mais mapas nas camadas profundas.

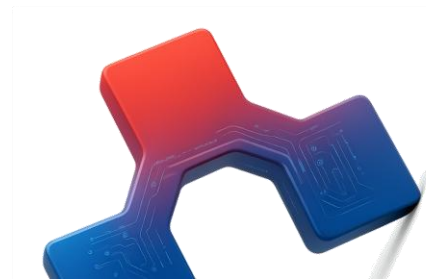
Após Flatten, usamos densas + ReLU para misturar globalmente as características. Dropout(0.5) adiciona regularização reduzindo overfitting. Em classificação com 10 classes, a última camada tem 10 unidades. Usamos CrossEntropyLoss (que internamente aplica LogSoftmax), então deixamos a saída como logits.





Treinado no Fashion-MNIST, esse modelo alcança $\sim 92\%$ de acurácia de teste (aprox.). Ao longo dos anos, arquiteturas evoluíram e baixaram o erro do ImageNet ILSVRC (top-5) de $>26\%$ para $<2,3\%$ em ~ 6 anos — imagens grandes (~ 256 px) e 1.000 classes (muitas sutis, como 120 raças de cães). Estudar os vencedores ajuda a entender o progresso em CNNs.

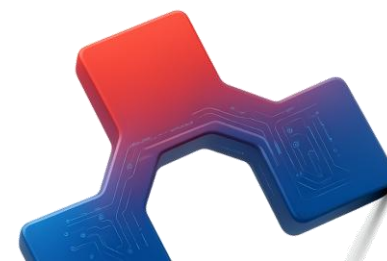
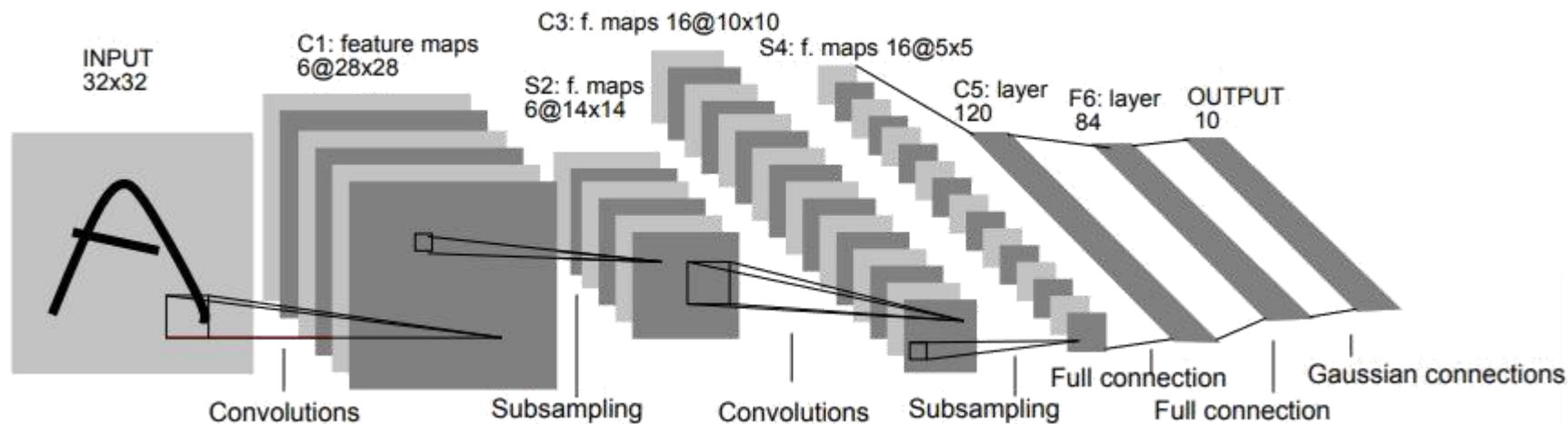
Veremos LeNet-5 (1998) e vencedoras do ILSVRC: AlexNet (2012), GoogLeNet/Inception (2014), ResNet (2015), SENet (2017). Também discutiremos VGG, Xception, ResNeXt, DenseNet, MobileNet, CSPNet, EfficientNet, ConvNeXt; *Vision Transformers* ficam para o Cap. 16.



5.1 LeNet-5

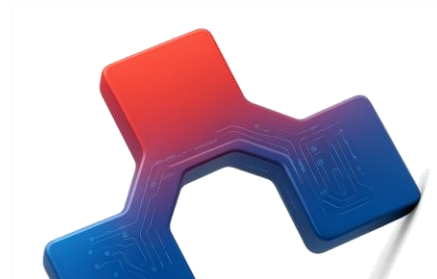
A **LeNet-5** (LeCun, 1998) é uma das CNNs mais conhecidas, popularmente usada para **reconhecimento de dígitos manuscritos (MNIST)**. Sua estrutura segue o padrão **Conv → Pool → ... → FC**, semelhante ao modelo que montamos para Fashion-MNIST: camadas convolucionais e de **pooling médio** (na época), seguidas por camadas **totalmente conectadas** para a classificação.

Diferenças para arquiteturas modernas: hoje, costuma-se usar **ReLU** (em vez de **tanh**) e **Softmax** (em vez de **RBF**) na saída, além de preferir **max pooling** e outros detalhes de projeto que evoluíram desde 1998.





Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–





A tabela descreve cada bloco:

- Entrada (In): 1 mapa (grayscale), 32×32 .
- C1 (Conv): 6 mapas, 28×28 , kernel 5×5 , stride 1, ativação \tanh .
- S2 (Avg pooling): 6 mapas, 14×14 , kernel 2×2 , stride 2, \tanh .
- C3 (Conv): 16 mapas, 10×10 , kernel 5×5 , stride 1, \tanh .
- S4 (Avg pooling): 16 mapas, 5×5 , kernel 2×2 , stride 2, \tanh .
- C5 (Conv): 120 mapas, 1×1 (após convolução), kernel 5×5 , stride 1, \tanh .
- F6 (FC): 84 unidades, \tanh .
- Out (FC): 10 saídas, RBF no artigo original (hoje usaríamos Softmax/logits).

Observações: (1) o uso de **average pooling** reflete escolhas históricas; (2) a passagem de 5×5 para C5 (1×1) aponta para o colapso espacial total antes das camadas densas; (3) a rede foi projetada para 32×32 , por isso muitas implementações aplicam **zero-padding** em imagens 28×28 (MNIST) para combinar o tamanho esperado.

