

Aula 7

Introdução e motivação

Como vimos na Aula 6, o processamento em redes neurais clássicas consiste em "achatar" a entrada num vetor unidimensional e então aplicar sucessivas camadas fully-connected, calculando scores, loss e retropropagando gradientes para ajustar os pesos. Embora esse método funcione bem em domínios como séries temporais ou vetores de características, ele se revela ineficiente para imagens, que são naturalmente mais complexas. Em vez de simples vetores, imagens são representadas como **tensores tridimensionais**, com três dimensões principais: altura (H), largura (W) e número de canais, que chamamos de C.

Esse valor C indica **quantas camadas de informação** existem em cada posição da imagem. Em imagens em preto e branco, por exemplo, C = 1, ou seja, há apenas um valor por pixel, representando a intensidade. Em imagens coloridas no padrão RGB, C = 3, pois cada pixel é composto por três números, um para o canal vermelho, um para o verde e um para o azul. E em contextos mais avançados, como visão 3D ou segmentação de cenas, esse número pode ser ainda maior, com canais representando profundidade, classes semânticas ou características extraídas pela própria rede.

Quando transformamos um quadrante 2×2 de uma imagem num vetor unidimensional, perdemos completamente a noção de vizinhança espacial entre pixels, conforme ilustrado na **Figura 1**.

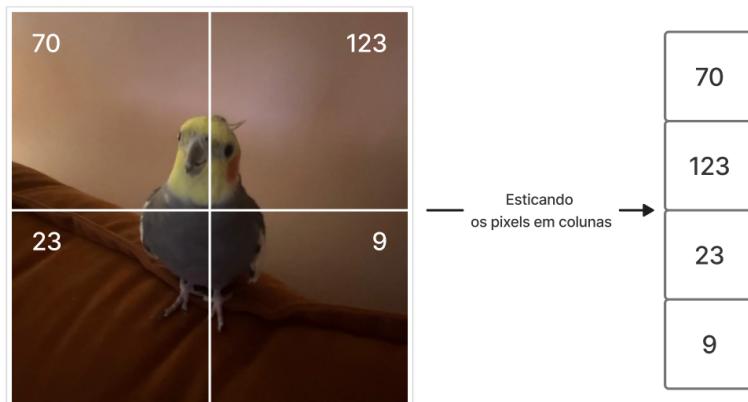


Figura 1: Esticando os pixels de um quadrante 2×2 para um vetor.

Isso obriga a rede a reaprender do zero todas as correlações locais, como por exemplo o fato de que bordas horizontais costumam ser importantes, aumentando drasticamente o número de parâmetros e degradando a eficiência do treinamento e da inferência.

As CNNs surgem para resolver esse problema. Em vez de alimentar diretamente o vetor achata do no forward pass, elas aplicam uma sequência de operações espaciais, como convolução, pooling e normalização, que exploram padrões locais e preservam a estrutura do tensor de entrada. Dessa forma, filtros convolucionais aprendem automaticamente a detectar bordas, texturas e formas em regiões específicas da imagem, reduzindo o número de parâmetros necessários e acelerando a convergência.

Neste material, vamos entender por que e como esses operadores funcionam no mundo tridimensional das imagens, explorando em sequência a intuição da convolução, os operadores básicos (stride, padding, campo receptivo), as técnicas de pooling e normalização, as arquiteturas clássicas como o LeNet-5 e, ao final, tecnologias mais modernas como o YOLO.

Intuição da convolução com exemplo

Agora que entendemos por que as CNNs são necessárias, vamos ver como elas funcionam na prática. O principal operador dessas redes é a **convolução**, uma operação que permite detectar padrões locais, como bordas, cantos e texturas. Para isso, usamos um pequeno conjunto de pesos chamado **filtro** (ou *kernel*) que percorre a imagem, aplicando multiplicações ponto a ponto sobre regiões locais e gerando como saída um novo mapa, o chamado **feature map** (ou mapa de características).

Podemos pensar nesse processo como uma janela que se move sobre a imagem, realizando pequenos cálculos em cada posição e armazenando o resultado. É justamente esse mecanismo que permite à rede detectar, por exemplo, onde há uma mudança brusca de intensidade, indicando uma borda.

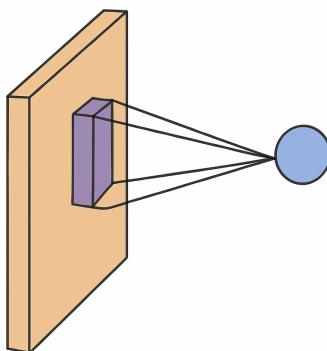


Figura 2: Imagem do campo receptivo em uma convolução. Cada ponto da saída (círculo azul) é influenciado por uma pequena região da entrada (janela roxa), permitindo à rede capturar padrões locais como bordas e texturas.

Para entender como isso funciona, vamos usar como exemplo o porquinho *Waddles*, da série *Gravity Falls*. Abaixo, temos sua imagem convertida para tons de cinza, com uma grade sobreposta. O

quadrado mais escuro destaca uma **região 3×3** da imagem, que será usada no exemplo numérico.

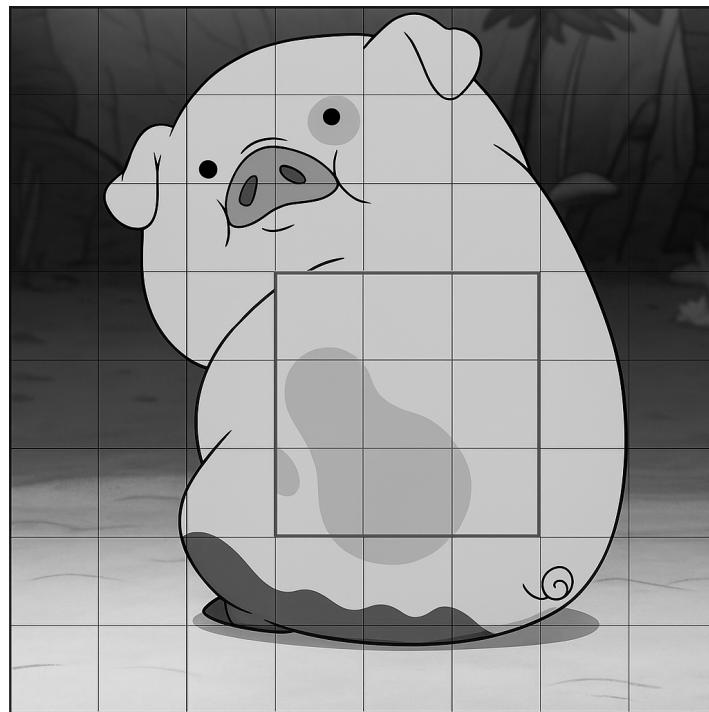


Figura 3: Imagem do Waddles com destaque para o bloco 3×3 usado no exemplo numérico.

Os valores numéricos dessa região, extraídos diretamente dos tons de cinza da imagem, formam a seguinte matriz:

$$X = \begin{bmatrix} 189.59 & 194.95 & 190.90 \\ 172.70 & 185.25 & 193.31 \\ 175.50 & 168.63 & 186.22 \end{bmatrix}$$

Agora, aplicaremos sobre esse bloco um filtro comum de detecção de bordas verticais:

$$F = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Esse filtro é conhecido por **realçar bordas verticais**. Ele faz isso ao somar as intensidades da direita da região e subtrair as da esquerda, ignorando o centro. Vamos calcular o valor da convolução entre esse filtro e o bloco extraído:

$$\text{Resultado} = \left(\begin{bmatrix} 189.59 & 194.95 & 190.90 \\ 172.70 & 185.25 & 193.31 \\ 175.50 & 168.63 & 186.22 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \right) \Rightarrow \\ = [32.64]$$

Esse valor, **32.64**, representa a **força da borda vertical** naquela região da imagem. Um valor alto significa que há uma variação intensa entre o lado esquerdo e o direito, ou seja, provavelmente

há uma borda. O resultado seria armazenado no mapa de características (*feature map*) na posição correspondente a essa região da imagem original.

Esse processo é repetido para todas as regiões 3×3 da imagem, e ao final teremos um novo "mapa" com as ativações do filtro em cada posição. A CNN pode usar diferentes filtros para detectar diferentes tipos de padrões, e o mais interessante: esses filtros são aprendidos automaticamente durante o treinamento da rede.

Arquitetura convolucional

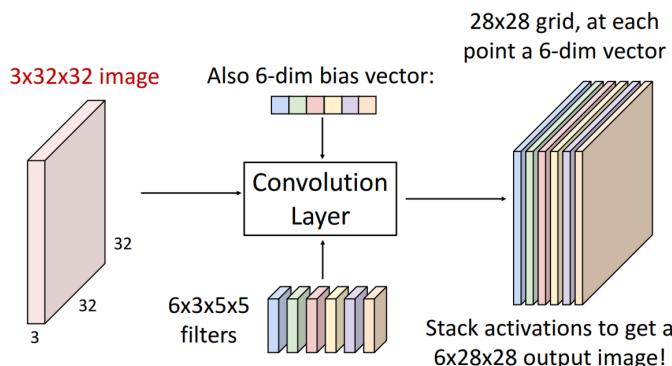


Figura 4: Primeira camada convolucional da rede, com 6 filtros $3 \times 5 \times 5$ e sem padding.

Já vimos, na seção anterior, como um filtro pode ser aplicado manualmente para destacar padrões locais em uma imagem. No exemplo do Waffles, utilizamos um filtro 3×3 para detectar bordas horizontais, realizando multiplicações ponto a ponto sobre uma região da imagem e somando os resultados. Esse tipo de filtro é um exemplo direto do que uma CNN aprende nas camadas iniciais, como bordas, linhas e transições bruscas de intensidade.

Ao treinar uma rede convolucional com imagens reais, o modelo aprende filtros semelhantes ao do exemplo, mas também muitos outros, com diferentes direções, cores e complexidades. Cada filtro é um pequeno bloco tridimensional de pesos que percorre toda a imagem, canal por canal, aplicando operações locais. O resultado de cada filtro é um mapa de ativação 2D, que mostra onde aquele padrão foi encontrado na imagem. Como cada filtro detecta um tipo específico de padrão, o empilhamento dos mapas de ativação gera um volume tridimensional de saída.

Podemos visualizar esse volume como uma grade (por exemplo, 28×28) em que, em cada coordenada (i, j) , há um *vetor* de 6 componentes (uma por filtro) indicando o grau de afinidade daquele patch com cada template aprendido. Esses vetores de características locais alimentam as camadas seguintes, que combinam padrões simples para formar representações cada vez mais complexas.

Esse padrão foi inspirado em evidências da neurociência. O experimento clássico de Hubel e Wiesel, premiado com o Nobel de 1981, observou a ativação de neurônios no córtex visual de gatos em resposta a estímulos visuais. Utilizando microeletrodos, os pesquisadores identificaram que certos

neurônios simples disparavam apenas quando uma linha com orientação específica era apresentada em uma região precisa do campo visual. Já neurônios mais complexos reagiam a estímulos mais elaborados, como o movimento de uma barra em uma direção. Esse comportamento hierárquico é exatamente o que as CNNs reproduzem: camadas iniciais capturam padrões simples e, ao empilhar mais camadas, a rede passa a representar formas e conceitos cada vez mais complexos.

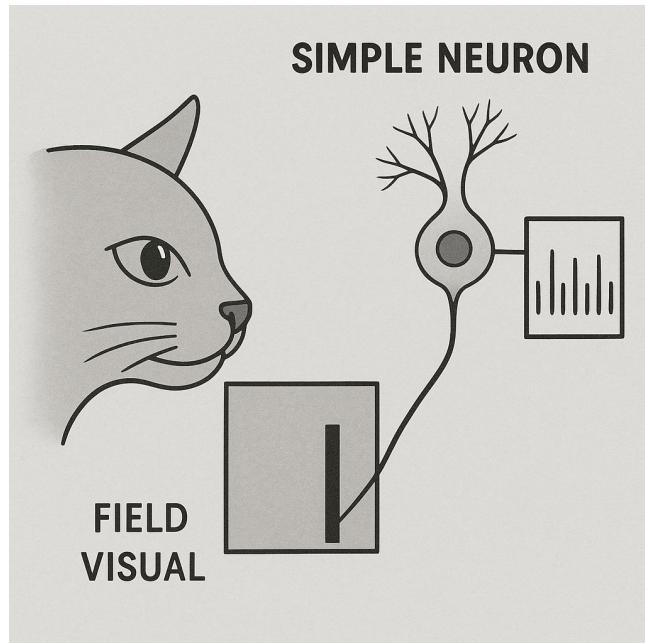


Figura 5: Neurônio simples ativado por uma linha vertical no canto inferior direito do campo visual.

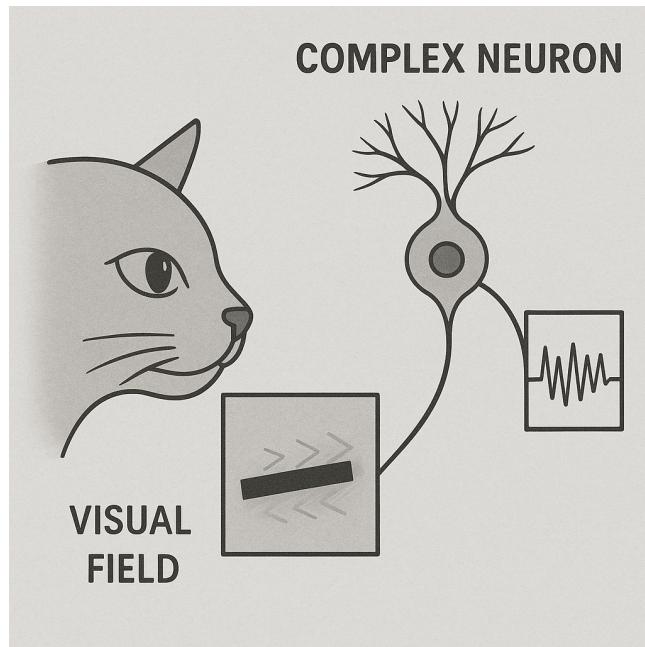


Figura 6: Neurônio complexo respondendo ao movimento de uma barra horizontal.

Padding e Receptive Field

Para fixar os conceitos de **padding** e **Receptive Field**, vamos trabalhar com um exemplo simples: uma imagem de 5×5 pixels (área azul) sendo processada por um kernel (também chamado de filtro)

de 3×3 , com *stride* $S = 1$. O *stride* indica o tamanho do "passo" com que o kernel se desloca sobre a imagem. Para todos os exemplos desta seção, manteremos $S = 1$, o que significa que o kernel avança um pixel por vez. Dessa forma, você não precisa se preocupar com as variações e os diferentes efeitos de outros valores de *stride* neste momento. Este tópico, incluindo seu impacto nas dimensões do mapa de características, será explorado em detalhes na próxima seção.

$$\begin{array}{|c|c|c|c|c|} \hline 7 & 2 & 3 & 3 & 8 \\ \hline 4 & 5 & 3 & 8 & 4 \\ \hline 3 & 3 & 2 & 8 & 4 \\ \hline 2 & 8 & 7 & 2 & 7 \\ \hline 5 & 4 & 4 & 5 & 4 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 6 & -9 & -8 \\ \hline -3 & -2 & -3 \\ \hline -3 & 0 & \\ \hline \end{array}$$

Figura 7 (a): o kernel nunca toca a borda, percorrendo apenas o interior da área azul.

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel

0	-1	0
-1	5	-1
0	-1	0

114	328	-26	470	158
53	266	-61	-30	344
403	116	-47	295	244
108	-135	256	-128	344
314	346	279	153	421

Figura 7 (b): Moldura de zeros (área cinza) de um pixel em cada lado, o que chamamos de *padding*.

Convolução sem padding ($P = 0$) Ao aplicar o kernel 3×3 sobre a imagem 5×5 sem bordas adicionais, o centro do kernel só pode visitar todas as combinações $i, j \in \{1, 2, 3\}$, totalizando $3 \times 3 = 9$ posições. Consequentemente, a saída tem dimensão

$$W_{\text{out}} = W_{\text{in}} - K + 1 = 5 - 3 + 1 = 3,$$

isto é, um mapa 3×3 . Perceba que os pixels nas bordas não participam e parte da informação original é descartada logo na primeira camada.

Convolução com padding de zeros ($P = 1$)

Inserindo uma borda de um pixel de zeros ao redor da imagem, o *canvas* torna-se 7×7 , embora os valores reais permaneçam nos 5×5 centrais. Agora, o centro do kernel pode visitar todas as coordenadas $i, j \in \{1, 2, 3, 4, 5\}$, totalizando $5 \times 5 = 25$ posições. Assim, a saída mantém as mesmas dimensões da entrada:

$$W_{\text{out}} = \frac{W_{\text{in}} - K + 2P}{S} + 1 = (5 + 2 \cdot 1) - 3 + 1 = 5,$$

resultando em um mapa 5×5 . O padding devolve às bordas o mesmo peso dos pixels centrais, preservando a resolução espacial.

Same padding em geral

Para qualquer kernel de tamanho ímpar K e *stride* $S = 1$, garantimos que a saída tenha a mesma dimensão da entrada definindo

$$P = \frac{K - 1}{2}.$$

Por exemplo, se $K = 3$ então $P = 1$; se $K = 5$ então $P = 2$; e assim por diante. Com essa escolha, o cálculo

$$W_{\text{out}} = \frac{W_{\text{in}} - K + 2P}{S} + 1$$

sempre resulta em $W_{\text{out}} = W_{\text{in}}$.

Receptive Field

Como vimos na seção anterior **Arquitetura Convolucional**, filtros convolucionais operam sobre regiões locais da imagem de entrada e geram como saída mapas de características que preservam a estrutura espacial.

Nesta subseção, primeiro apresentamos a fórmula genérica que calcula o campo receptivo teórico em qualquer configuração de camadas convolucionais ou de pooling. Em seguida, mostramos como essa fórmula se reduz ao nosso exemplo contínuo de três camadas 3×3 com **stride 1** e **padding 1**.

Fórmula geral do campo receptivo

Antes de detalhar cada termo, listamos as definições iniciais

$$\text{RF}_0 = 1, \quad j_0 = 1$$

onde RF_0 representa o tamanho do campo receptivo antes de qualquer camada (ou seja, um único pixel) e j_0 indica o salto entre neurônios na camada de entrada (um pixel).

A cada camada ℓ , definimos:

$$j_\ell = j_{\ell-1} \cdot s_\ell$$

onde s_ℓ é o **stride** da camada ℓ . Este salto j_ℓ diz quantos pixels na imagem original correspondem a um deslocamento de um neurônio para o outro na camada ℓ .

$$\text{RF}_\ell = \text{RF}_{\ell-1} + (k_\ell - 1) j_{\ell-1}$$

em que k_ℓ é o tamanho do kernel (por exemplo, 3 se for um filtro 3×3) na camada ℓ . O termo $(k_\ell - 1)$ indica em quantos pixels o campo receptivo se expande além do ponto de contato, e a multiplicação

por $j_{\ell-1}$ leva esse acréscimo à escala da imagem original.

A expressão acima pode ser combinada para gerar a forma fechada do campo receptivo após L camadas:

$$\text{RF}_{\text{final}} = 1 + \sum_{l=1}^L (k_l - 1) \prod_{i=1}^{l-1} s_i$$

Aqui:

- L é o número total de camadas que influenciam o campo receptivo (camadas convolucionais e/ou de pooling).
- Para cada l , k_l é o tamanho do kernel na camada l .
- Para cada i , s_i é o stride da camada i .
- $\prod_{i=1}^{l-1} s_i$ é o produto de todos os strides desde a primeira camada até a camada $l - 1$. Quando $l = 1$, este produto é vazio e definido como 1.

Cada camada pode ter stride maior que 1 (por exemplo, convolução com *stride* 2 ou pooling 2×2 com *stride* 2). Nestes casos, o salto j_ℓ cresce e faz com que o campo receptivo aumente mais rapidamente. Nesta fórmula geral, basta inserir o valor de cada s_i correspondente.

Caso particular: três camadas 3×3 , $\text{stride} = 1$, $\text{padding} = 1$

No nosso exemplo contínuo, cada camada ℓ tem:

$$k_\ell = 3, \quad s_\ell = 1, \quad p_\ell = 1$$

e todos os strides são iguais a 1. Isso implica que, para cada l , temos:

$$\prod_{i=1}^{l-1} s_i = 1.$$

Assim, a expressão fechada se torna:

$$\text{RF}_{\text{final}} = 1 + \sum_{l=1}^3 (3 - 1) \cdot 1 = 1 + 2 + 2 + 2 = 7.$$

Ou seja, três camadas consecutivas de $3 \times 3/\text{stride} = 1/\text{padding} = 1$ produzem um campo receptivo teórico de 7×7 . Abaixo, ilustramos em detalhes cada etapa:

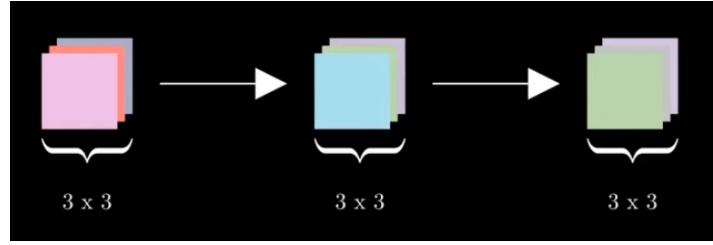


Figura 8: Crescimento do *Receptive Field* em três camadas 3×3 .

Camada 1:

$\text{RF}_0 = 1$ e $j_0 = 1$. Como $k_1 = 3$ e $j_0 = 1$, temos

$$\text{RF}_1 = \text{RF}_0 + (3 - 1) j_0 = 1 + 2 \cdot 1 = 3.$$

Portanto, cada neurônio da primeira camada “vê” exatamente 3×3 pixels da imagem original.

Camada 2:

Primeiro calculamos o salto:

$$j_1 = j_0 \cdot s_1 = 1 \cdot 1 = 1.$$

Como $k_2 = 3$ e $j_1 = 1$, o campo receptivo na camada 2 é

$$\text{RF}_2 = \text{RF}_1 + (3 - 1) j_1 = 3 + 2 \cdot 1 = 5.$$

Isso mostra que cada neurônio da segunda camada “vê” um bloco 5×5 da imagem original.

Camada 3:

Calculamos o salto novamente:

$$j_2 = j_1 \cdot s_2 = 1 \cdot 1 = 1.$$

Como $k_3 = 3$ e $j_2 = 1$, obtemos

$$\text{RF}_3 = \text{RF}_2 + (3 - 1) j_2 = 5 + 2 \cdot 1 = 7.$$

Logo, o campo receptivo teórico final é 7×7 .

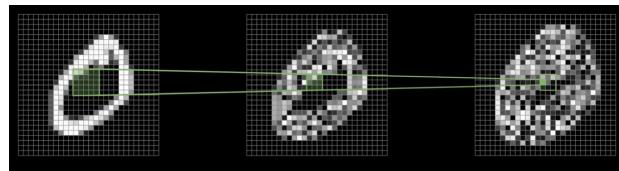


Figura 9: Crescimento do *Receptive Field* em profundidade.

A ativação de um neurônio em uma camada mais profunda (à direita) depende de uma região cada vez maior da imagem de entrada (à esquerda), como indicado pelas conexões verdes. Essa expansão

progressiva define o *campo receptivo efetivo* da rede.

Receptive Field teórico vs. efetivo

Embora o cálculo do campo receptivo teórico (por exemplo, 7×7) indique a região máxima da entrada que pode influenciar a ativação de um neurônio, na prática a rede **não** utiliza cada pixel dessa região com a mesma intensidade. Isso nos leva aos conceitos de **Receptive Field teórico** e **Receptive Field efetivo**.

- **Receptive Field teórico** é o tamanho máximo da região da imagem de entrada que, pela arquitetura da rede (kernéis, strides, etc.), pode chegar a influenciar um neurônio específico.
- **Receptive Field efetivo** é a porção dentro desse campo teórico que realmente exerce influência significativa, pois durante o treinamento os gradientes tendem a se concentrar no centro do campo receptivo, fazendo com que áreas periféricas tenham influência muito reduzida. Muitas vezes, a sensibilidade assume uma forma semelhante a uma distribuição aproximada de Gauss, com pico no centro.

A Figura a seguir ilustra um exemplo em que, mesmo tendo 7×7 pixels no campo teórico, apenas o sub-bloco central contribui de fato para a ativação final:

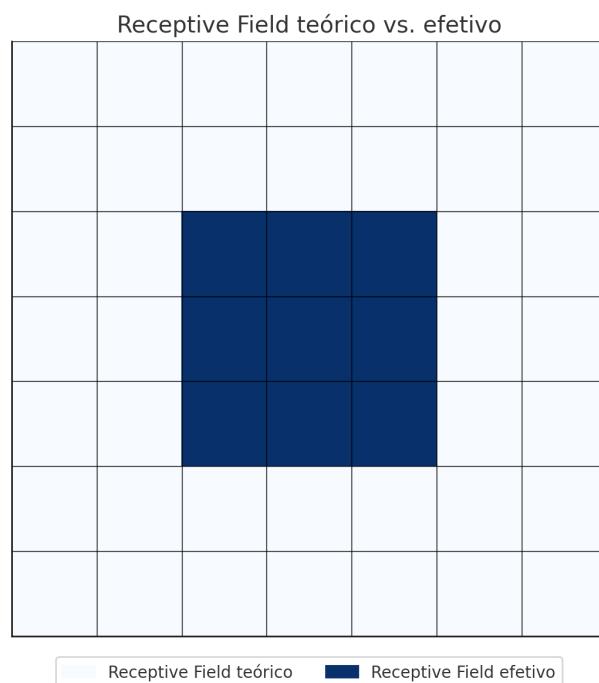


Figura 10: Distribuição típica do gradiente: o *Receptive Field* efetivo (azul escuro) ocupa apenas parte do campo teórico 7×7 .
(Exemplo didático)

Comentário sobre Stride e Pooling

No exemplo acima, todos os $s_\ell = 1$ e não usamos pooling. A partir do próximo capítulo, ao tratarmos

de pooling (por exemplo, pooling 2×2 com **stride** 2), veremos que o salto j_ℓ dobra a cada aplicação de pooling, o que faz o campo receptivo crescer mais rapidamente. Em linhas gerais:

$$j_\ell = j_{\ell-1} \cdot s_\ell, \quad \text{RF}_\ell = \text{RF}_{\ell-1} + (k_\ell - 1) j_{\ell-1}.$$

Se $s_\ell > 1$, o centro de visão de cada neurônio fica mais afastado na imagem original, aumentando o tamanho do campo de forma multiplicativa. Essa situação será explorada em detalhes quando abordarmos o tópico “Pooling e suas implicações no Receptive Field”.

Stride e convoluções especiais

Stride

Introduzimos o hiperparâmetro **stride** S para controlar o espaçamento com que o kernel percorre a imagem. Em vez de deslocar o kernel uma posição por vez, avançamos de S em S .

Por exemplo, considere uma entrada 7×7 e um kernel 3×3 com stride = 2. Nesse caso, o kernel cabe em apenas três posições horizontais e três verticais, produzindo uma saída 3×3 .

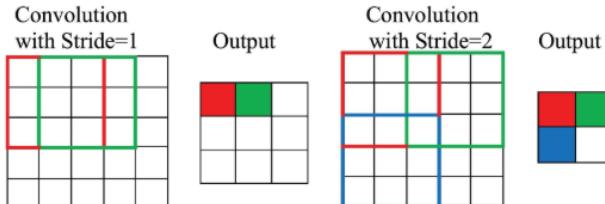


Figura 11: Comparativo entre convolução 3×3 com stride = 1 (esquerda) e stride = 2 (direita), mostrando as posições do kernel e o mapa de saída correspondente.

Cada aplicação de stride = 2 dobra o passo espacial da convolução, aumentando mais rapidamente o **Receptive Field** sem precisar empilhar tantas camadas.

Com $\text{stride} > 1$, reduzimos a resolução do mapa de características e também o custo computacional.

Na prática, escolhemos S e P de modo que o valor acima seja inteiro. Embora implementações possam truncar ou arredondar, arquiteturas bem projetadas evitam esse tipo de ajuste.

Exemplo:

Entrada $3 \times 32 \times 32$, com 10 filtros 5×5 , stride = 1, padding = 2.

$$W_{\text{out}} = \frac{32 - 5 + 2 \cdot 2}{1} + 1 = 32$$

A saída mantém a mesma resolução da entrada. Como usamos 10 filtros, a profundidade resultante será 10, formando um tensor $10 \times 32 \times 32$.

Em resumo: o **padding** preserva tamanho, o **stride** controla a resolução e ambos afetam o crescimento do Receptive Field.

Usamos $\text{stride} = 1$ para manter detalhes espaciais e $\text{stride} = 2$ (ou pooling) quando queremos agregar contexto e reduzir custo computacional.

Convoluçãoes especiais

Além das convoluções padrão, existem variantes que permitem ampliar o **Receptive Field** ou reduzir parâmetros sem perder capacidade de representação.

Dilated Convolutions

Nas **dilated convolutions**, inserimos espaços entre os elementos do kernel, controlados pelo hiperparâmetro de dilatação D . Isso permite que um kernel $K \times K$ cubra uma área maior na entrada sem aumentar o número de parâmetros.

O tamanho efetivo do kernel dilatado é:

$$K_{\text{eff}} = K + (K - 1) \cdot (D - 1)$$

Exemplo com $K = 3$ e $D = 2$:

$$K_{\text{eff}} = 3 + (3 - 1)(2 - 1) = 5$$

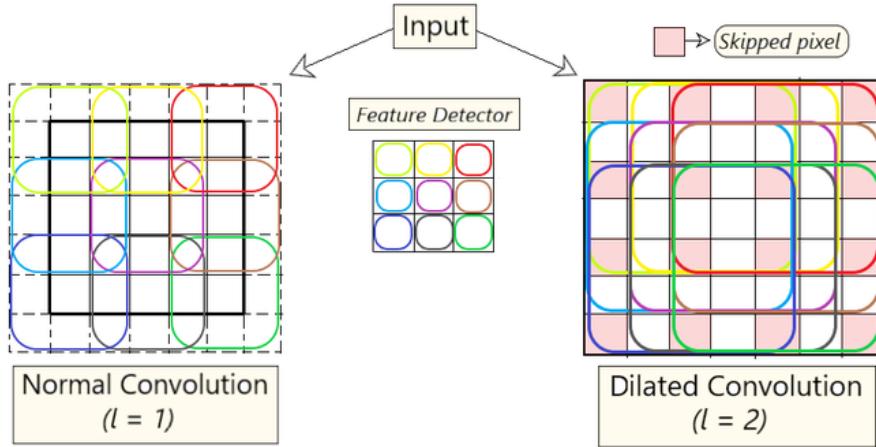


Figura 12: Comparativo entre convolução normal ($D = 1$) e dilatada ($D = 2$). As cores indicam os deslocamentos dos filtros. Os quadrados rosa representam os pixels ignorados devido à dilatação.

Com dilation = 2, o kernel 3×3 cobre uma área equivalente a 5×5 , mas com o mesmo número de parâmetros.

Cada elemento ativo do kernel é separado por um pixel "vazio", ampliando o alcance da rede sem reduzir a resolução da imagem de entrada. Essa técnica é especialmente útil em tarefas como **segmentação semântica**, onde precisamos captar contexto amplo sem perder precisão nos detalhes locais.

Convoluções 1×1

As convoluções 1×1 aplicam um filtro sobre cada posição do mapa de características, operando como uma transformação linear dos canais naquele ponto.

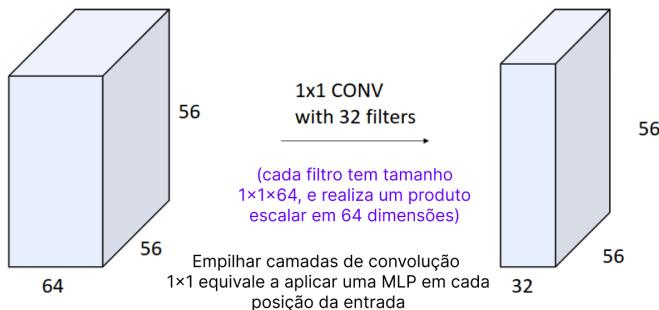


Figura 13: Convolução 1×1 com 64 canais de entrada e 32 filtros, resultando em saída $32 \times H \times W$.

São úteis para:

- Reduzir ou expandir o número de canais;
- Combinar saídas de convoluções anteriores;
- Agir como adaptadores entre blocos de diferentes profundidades.

Essa técnica é amplamente usada em arquiteturas como Inception, ResNet e MobileNet.

Depthwise Separable Convolutions

Esse tipo de convolução divide o processo em duas etapas:

1. **Depthwise convolution:** aplica um kernel $K \times K$ separadamente em cada canal da entrada. Isso gera mapas intermediários com a mesma profundidade.
2. **Pointwise convolution:** usa um kernel 1×1 para combinar os canais, projetando para a profundidade desejada.

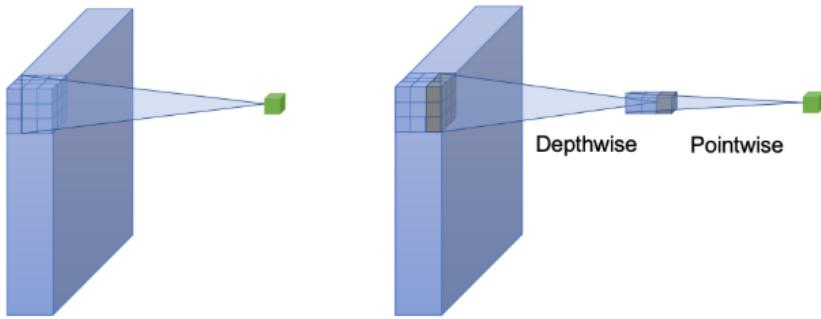


Figura 14: Comparação entre convolução padrão (à esquerda) e convolução depthwise separável (à direita). A primeira aplica filtros 3D diretamente; a segunda divide o processo em filtragem por canal (depthwise) e combinação com 1×1 (pointwise).

Comparando parâmetros:

$$\text{Conv normal: } K^2 \cdot C_{\text{in}} \cdot C_{\text{out}}$$

$$\text{Depthwise separável: } K^2 \cdot C_{\text{in}} + C_{\text{in}} \cdot C_{\text{out}}$$

Essa abordagem reduz drasticamente o custo computacional, mantendo o desempenho em muitas tarefas. É usada em arquiteturas como o **MobileNet**.

Grouped Convolutions

Em **grouped convolutions**, dividimos os canais de entrada em G grupos. Cada grupo é processado por um subconjunto específico de filtros. Se temos C_{in} canais e C_{out} filtros finais, cada grupo trata C_{in}/G canais e gera C_{out}/G saídas.

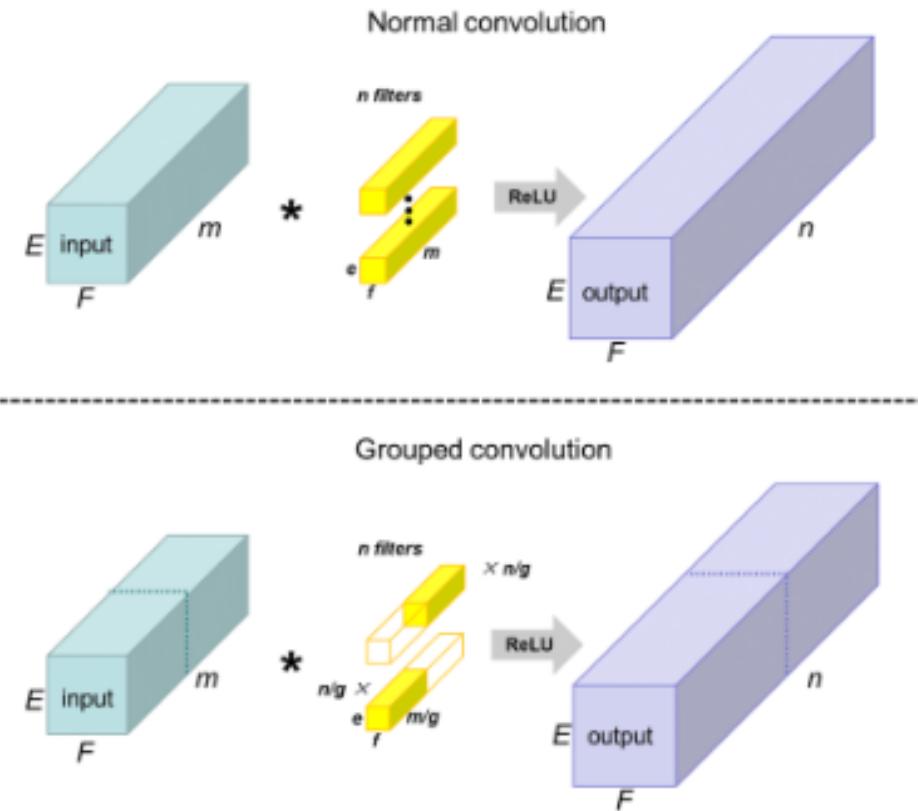


Figura 15: Comparação entre convolução padrão (acima) e convolução agrupada (abaixo). Em grouped convolutions, a entrada e os filtros são divididos em grupos independentes, reduzindo o custo computacional.

Essa técnica é útil em redes com restrições de memória ou que precisam aproveitar paralelismo de hardware, como no AlexNet e ResNeXt.

Essas variações, **dilated**, **1×1**, **separable** e **grouped convolutions**, tornam as CNNs mais eficientes e controláveis, ampliando o Receptive Field ou reduzindo parâmetros conforme necessário.

Nas próximas seções, abordaremos o **pooling** em detalhes. Não é necessário dominar todos os aspectos dessas convoluções por agora, são apenas exemplificações de como esse universo pode se expandir.

Pooling

Intuição

Como vimos na seção anterior, é por meio da operação de convolução que as CNNs são capazes de extrair características, como bordas e outros tipos de padrões das imagens. Contudo, imagine que tenhamos uma imagem de $3 \times 1280 \times 720$, que é uma imagem HD: são 2.764.800 valores de pixel que a nossa rede precisa analisar! Logo, precisamos de alguma forma de reduzir esse número de valores à medida que passamos pelas camadas de convolução.

Nesse sentido, o **pooling** entra como um mecanismo de redução de resolução espacial. Ele *não* é aplicado como um filtro treinável (ou kernel), isto é, não há pesos aprendidos; em vez disso, usa-se uma *janela deslizante* de tamanho $K \times K$ que percorre o *feature map*. Em cada posição dessa janela, calcula-se um *único* valor agregado (máximo, média, mínimo etc.) que substituirá todos os K^2 valores originais daquela área. Assim:

1. A janela se desloca (controlada por um *stride* S) sobre todo o mapa de ativação.
2. Para cada bloco $K \times K$, aplica-se a função de pooling escolhida e obtém-se um valor.
3. O resultado forma um novo mapa de características de menor resolução.

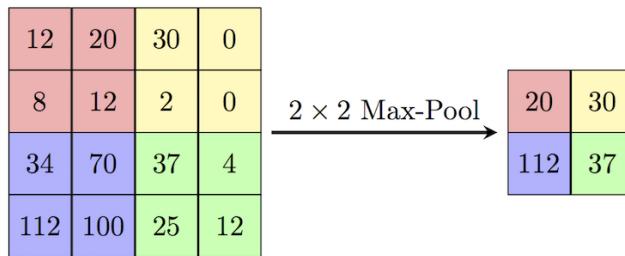


Figura 16: Exemplo de max pooling com janela 2×2 .

Dessa forma, preservamos as características extraídas pelo filtro convolucional, pois ao pegar a média ou o máximo de uma região mantemos um valor representativo daquela área. Além disso, reduzimos drasticamente a quantidade de ativações que as camadas posteriores precisam manipular.

Diferente das convoluções, o pooling não possui parâmetros treináveis, apenas *hiperparâmetros* como tamanho da janela (K) e *stride* (S). Portanto, adiciona eficiência sem aumentar o número de pesos da rede.

Tipos de Pooling

Antes de listar as variações em tópicos, vale um panorama sequencial. O método mais popular é o **max pooling** (Presente na Figura 16): ele varre o feature map em blocos e, em cada bloco, propaga o valor de maior intensidade. Isso realça as ativações dominantes, ajudando a rede a se concentrar nas partes mais salientes do sinal. Quando se deseja uma visão mais suave, usa-se o **average pooling**, que devolve a média dos valores dentro da janela: útil quando a informação está distribuída de maneira relativamente homogênea. Em casos raros, pode-se empregar **min pooling** para destacar regiões de menor ativação, por exemplo em mapas de saliência negativos ou detecção de falhas. Independentemente da escolha, todas as variantes compartilham a mesma mecânica de janela deslizante e redução espacial.

- **Max-Pooling:**

- Seleciona o valor máximo dentro de uma janela;
- Preserva características mais ativas e proeminentes;
- É o tipo mais comum e geralmente oferece melhor desempenho em tarefas de classificação.

- **Average-Pooling:**

- Calcula a média dos valores dentro da janela;
- Suaviza o mapa de características, preservando informações contextuais;
- Mais adequado quando a informação está distribuída uniformemente na região;
- Exemplo: para a janela $\begin{bmatrix} 3 & 8 \\ 5 & 2 \end{bmatrix}$, a saída é $(3 + 8 + 5 + 2)/4 = 4,5$.

- **Min-Pooling:**

- Seleciona o valor mínimo dentro da janela;
- Útil para destacar regiões com baixa ativação;
- Pode ser aplicado em contextos específicos onde características de ausência são importantes.

É comum utilizar janelas de pooling com tamanho 2×2 e stride = 2, o que garante que as regiões não se sobreponham. Isso simplifica os cálculos e evita redundância entre blocos vizinhos.

Pooling e convoluções com stride

Assim como convoluções com **stride maior que 1** reduzem a resolução espacial ao “pular” posições, o pooling também diminui o tamanho do mapa de ativações, mas por *agregação* e não por *pesos treináveis*. Pode-se enxergar o pooling como uma “convolução determinística” cujo kernel executa uma função fixa (máximo, média etc.) em cada região.

Pooling e ativação (ReLU)

Como relembrado na subseção **Funções de Ativação** da seção *Arquitetura Convolucional*, a **ReLU** (Rectified Linear Unit) é a função

$$f(x) = \max(0, x).$$

Em outras palavras, ela deixa passar os valores positivos do mapa de ativação e zera tudo que for negativo. Pense na ReLU como uma válvula: só abre para números maiores que zero. Isso ajuda a rede a

- manter o gradiente “vivo” em camadas profundas,

- acelerar o treinamento,
- produzir mapas mais esparsos (muitos valores viram zero), tornando a detecção de padrões mais simples.

Mesmo que o *max pooling* já seja uma operação não linear, é comum encerrar o bloco $Conv \rightarrow ReLU \rightarrow Pool$ com mais uma $ReLU$, ficando $Conv \rightarrow ReLU \rightarrow Pool \rightarrow ReLU$. Isso garante que todos os blocos entreguem saídas não negativas para a próxima convolução e padroniza o design. Arquiteturas mais antigas, como o LeNet-5, dispensavam essa $ReLU$ extra; porém, nas redes modernas e profundas (AlexNet, VGG, ResNet) ela se tornou praxe por custar praticamente nada em tempo de execução e ajudar a estabilizar o aprendizado.

Conclusões sobre Pooling

A principal função do pooling é a **redução de dimensionalidade espacial** e a introdução de **invariância a pequenas translações**. Por exemplo, considere:

$$\begin{matrix} 20 & 3 \\ 10 & 4 \end{matrix} \Rightarrow \text{Max-Pooling} = 20$$

Se o valor 20 mover-se para:

$$\begin{matrix} 4 & 3 \\ 10 & 20 \end{matrix} \Rightarrow \text{Max-Pooling ainda} = 20$$

O resultado permanece idêntico, mostrando que pequenas mudanças de posição não afetam a saída. Assim, o pooling torna redes convolucionais mais eficientes, robustas e generalizáveis, reduzindo resolução espacial sem sacrificar informação essencial.

LeNet-5

Agora, para ver como todas as operações se combinam em uma CNN prática, apresentamos a **LeNet-5**, projetada para reconhecer dígitos manuscritos em imagens 32×32 em tons de cinza (1 canal). A Figura 17, a seguir, ilustra o encadeamento das camadas conforme a convenção original de LeCun:

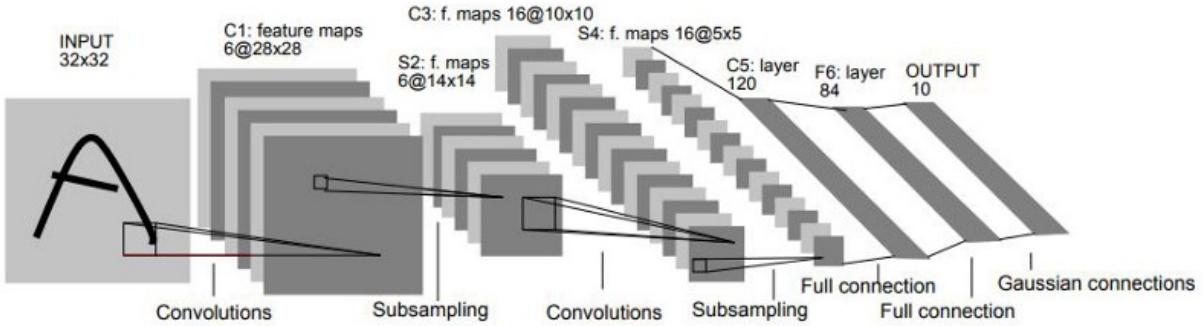


Figura 17: Diagrama da LeNet-5, onde “C” indica camada de convolução, “S” indica subsampling (pooling) e “F” indica camada totalmente conectada.

- **1^a Camada (C1 → ReLU)** A entrada é uma imagem 32×32 em escala de cinza. Aplica-se uma convolução com 6 filtros 5×5 , stride = 1 e sem *padding*. Isso reduz o tamanho espacial de 32×32 para 28×28 , pois $28 = 32 - 5 + 1$. Cada filtro gera um *feature map* diferente, resultando em $6 \times 28 \times 28$. Em seguida, aplica-se a função de ativação ReLU: ela mantém valores positivos e zera valores negativos, introduzindo não linearidade imediatamente após a convolução.
- **2^a Camada (S2)** Realiza *max pooling* com janela 2×2 e stride = 2. Cada *feature map* de 28×28 é reduzido para 14×14 , gerando uma saída de $6 \times 14 \times 14$. O pooling localiza, a cada bloco 2×2 , o valor máximo e o propaga, garantindo invariância a pequenas translações e diminuindo o número de ativações que serão processadas adiante.
- **3^a Camada (C3 → ReLU)** A entrada agora é o tensor $6 \times 14 \times 14$. Aplica-se uma segunda convolução com 16 filtros 5×5 , stride = 1 e sem *padding*. O tamanho espacial cai de 14×14 para 10×10 (pois $10 = 14 - 5 + 1$). Assim, a saída é $16 \times 10 \times 10$. Depois, vem novamente a ReLU, garantindo que cada mapa de ativação C3 tenha valores não negativos.
- **4^a Camada (S4)** Executa *max pooling* 2×2 com stride = 2. Cada *feature map* 10×10 passa para 5×5 , produzindo $16 \times 5 \times 5$. Nesse ponto, o tamanho espacial já caiu de 32 para 5, enquanto a profundidade (número de canais) subiu de 1 para 16.
- **5^a Camada (C5 → ReLU)** A entrada é $16 \times 5 \times 5$. Aplica-se uma última convolução 5×5 que cobre toda a extensão espacial 5×5 . Cada filtro gera um único valor escalar, porque a janela do kernel coincide exatamente com o tamanho do mapa. No total, há 120 filtros, resultando em $120 \times 1 \times 1$. Depois, a ReLU transforma esse tensor em um vetor \mathbb{R}^{120} , já que as dimensões espaciais são 1×1 .
- **6^a Camada (F6 → ReLU)** Recebe o vetor \mathbb{R}^{120} . Trata-se de uma camada totalmente conectada com 84 neurônios. A ReLU aplicada aqui transforma o vetor de entrada em outro vetor \mathbb{R}^{84} com valores não negativos. Essa camada funciona como *hidden-layer* de uma MLP, sintetizando as características extraídas pelas camadas anteriores.

- **7^a Camada (Saída)** Por fim, uma camada totalmente conectada de 10 neurônios: um para cada dígito de 0 a 9. Cada unidade gera uma pontuação não normalizada; após aplicar *softmax*, obtemos probabilidades de classificação para as dez classes. Assim, a saída final é um vetor \mathbb{R}^{10} .

Observando essa sequência, percebe-se que o tamanho espacial de cada saída é crucial para definir qual camada vem a seguir: quando os *feature maps* chegam a 1×1 , não faz mais sentido usar outra convolução, pois ela não reduziria mais a dimensão espacial. Assim, passamos para camadas totalmente conectadas. Além disso, ao final da etapa de neurônios totalmente conectados, a quantidade de unidades deve coincidir com o número de classes do problema: no caso, 10, uma para cada dígito.

Vale lembrar que, embora a LeNet-5 original de LeCun, pesquisador pioneiro em redes neurais convolucionais, utilizasse *average pooling* e função de ativação sigmoide nos trabalhos iniciais, a convenção ensinada aqui adapta esses pontos para **max pooling** e **ReLU**, prática que se generalizou porque acelera a convergência e melhora o fluxo de gradientes em redes profundas. Essa adaptação mantém a estrutura arquitetural original, mas incorpora avanços empíricos que se mostraram mais eficazes em treinamentos modernos, sobretudo em redes mais profundas e com datasets maiores.

Batch Normalization

Conforme avançamos no empilhamento de camadas convolucionais, observamos que o fluxo de informações na rede começa a enfrentar problemas de instabilidade, mesmo com o uso de ReLU, padding, stride e pooling. Um dos principais responsáveis por isso é o **internal covariate shift**, isto é, a mudança na distribuição das ativações internas ao longo do treinamento.

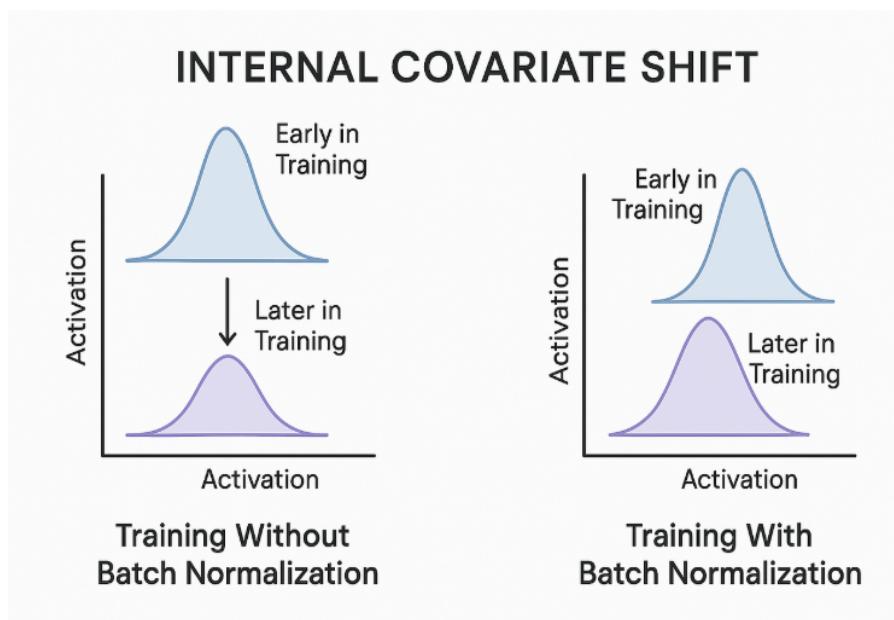


Figura 17: Mudança na distribuição das ativações ao longo do tempo (internal covariate shift). À esquerda, sem normalização, as distribuições variam muito entre camadas. À direita, com BatchNorm, elas permanecem estáveis.

Esse fenômeno ocorre porque, a cada atualização de pesos nas camadas iniciais, a distribuição estatística das ativações muda, afetando negativamente a aprendizagem nas camadas subsequentes. Isso exige estratégias adicionais como taxas de aprendizado menores, inicializações específicas e evita o uso de funções como **sigmoid**, que saturam facilmente.

Para mitigar esse problema, Ioffe e Szegedy propuseram o **Batch Normalization (BN)**, que normaliza as ativações de uma camada com base nas estatísticas de um mini-batch, antes da aplicação da função de ativação. Isso estabiliza a distribuição das ativações e acelera o treinamento.

O funcionamento matemático é direto. Dado um *mini-batch* de ativações x_1, x_2, \dots, x_m , calculamos sua média e variância:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

Em seguida, normalizamos cada valor:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Para permitir que a rede aprenda representações mais flexíveis, aplicamos uma transformação linear com parâmetros treináveis:

$$y_i = \gamma \hat{x}_i + \beta$$

Com isso, a rede pode manter ou ajustar o efeito da normalização conforme necessário. Durante o treinamento, usamos médias e variâncias por mini-batch; já na inferência, utilizamos estatísticas acumuladas globalmente, para garantir consistência.

Batch Normalization

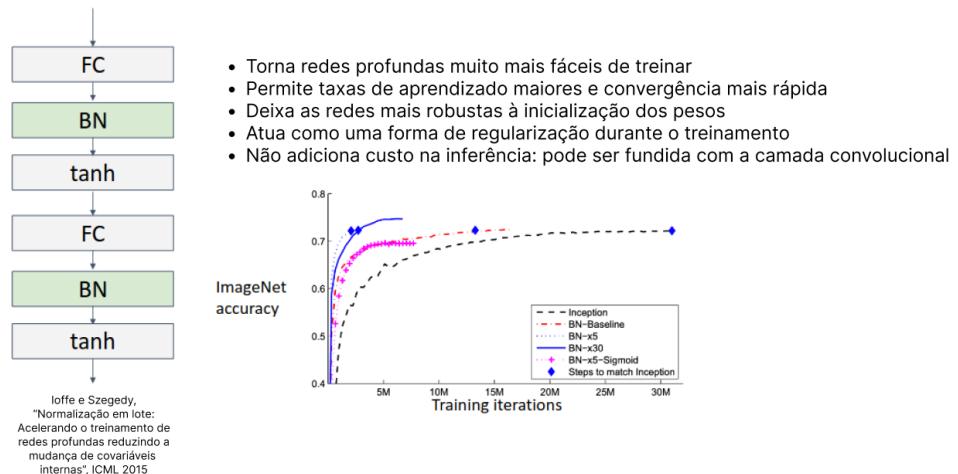


Figura 18: Impacto da Batch Normalization no desempenho de redes profundas.

Além da estabilização, o BN oferece benefícios adicionais. Ao reduzir a sensibilidade das ativa-

ções a variações nos pesos, permite o uso de *learning rates* maiores, acelera a convergência e evita ficar preso em platôs da função de perda.

Outro efeito é a leve **regularização** introduzida pela variação estatística entre os mini-batches. Esse ruído atua de forma semelhante ao Dropout, ajudando a reduzir overfitting. Em algumas arquiteturas modernas, o uso do BN elimina totalmente a necessidade do Dropout.

Também é possível treinar redes com funções de ativação mais sensíveis, como **sigmoid** e **tanh**, pois o BN garante que suas entradas fiquem centradas e com variância unitária, evitando a saturação.

Por fim, o BN é totalmente diferenciável e compatível com o processo de **backpropagation**, sendo integrado diretamente à computação de gradientes. Em redes convolucionais, como vimos na LeNet-5, a normalização é feita *canal por canal*, preservando a estrutura espacial e aplicando a mesma normalização a todos os pixels de um canal.

O Batch Normalization tornou-se um componente essencial em redes modernas como **ResNet**, **MobileNet** e **YOLO**, contribuindo para o avanço no treinamento de redes profundas com desempenho estável e eficiente.

Aplicação prática: YOLO em VRUs

Para encerrar nosso material, vamos agora acompanhar uma demonstração prática da aplicação do **YOLO**, uma das arquiteturas mais modernas para detecção de objetos, no contexto de **VRUs** (*Vulnerable Road Users*), como pedestres, ciclistas e motociclistas.

Esse exemplo prático será apresentado durante a aula, com vídeos reais sendo processados pelo modelo para ilustrar, na prática, como os conceitos aprendidos ao longo da apostila se aplicam em um cenário real.

Acompanhe atentamente a aula para entender como essas técnicas se conectam em uma pipeline moderna de visão computacional.