Deep Learning

Introdução a Redes Convolucionais

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

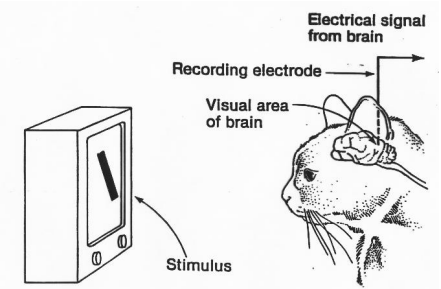
agosto de 2026

Overview

Visão Geral

Motivação Histórica

- David Hubel e Torsten Wiesel realizaram experimentos implantando eletrodos no córtex visual de gatos[a].
- Aparentemente o processamento começa identificando padrões simples, como arestas.
  - Vídeo
- Camadas seguintes combinam esses padrões em estruturas mais complexas.



[a] David H. Hubel e Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". Em: The Journal of Physiology 160.1 (1962), pp. 106–154.

Redes Convolucionais

- São redes que incluem operações de convolução.
  - Usadas principalmente em imagens.
- Criadas em 1989 por Yann LeCun, com a LeNet[1].
- Explodiram em popularidade após 2012, com a AlexNet[2].
  - Treinamento em GPUs.
  - Mais dados.
  - Mais poder computacional.

[1] Yann LeCun et al. "Handwritten digit recognition with a back-propagation network". Em: NeurIPS. vol. 2. 1989.

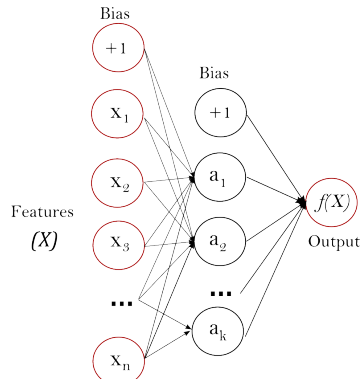
[2] Alex Krizhevsky, Ilya Sutskever e Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". Em: NeurIPS. vol. 25. 2012.

Problemas com MLP

Imagens têm **três propriedades** que sugerem o uso de arquiteturas específicas:

1. Alta dimensionalidade

- Uma imagem 512×512 resulta em 262.144 pixels.
- Com 100 unidades na primeira camada oculta de um MLP, teríamos 26.214.400 pesos.
- Em `float32` (4 bytes por peso), $\approx$ 100 MB **só** na primeira camada.



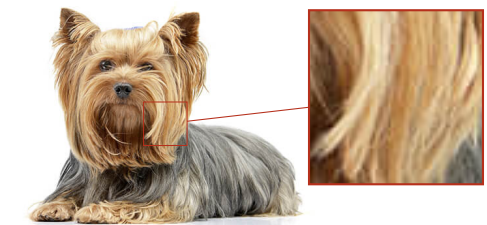
MLP totalmente conectado: milhões de pesos.

Problemas com MLP

Imagens têm **três propriedades** que sugerem o uso de arquiteturas específicas:

2. Alta coesão espacial

- Pixels próximos têm valores **altamente correlacionados** (veja o *zoom* ao lado).
- Uma mesma característica (ex.: uma orelha) pode aparecer em **vários locais** da imagem.



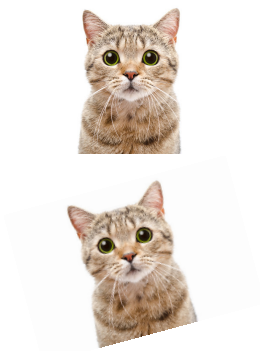
Vizinhança quase uniforme → valores semelhantes.

Problemas com MLP

Imagens têm **três propriedades** que sugerem o uso de arquiteturas específicas:

3. Semântica estável a transformações geométricas

- Translações, rotações e deformações **não mudam** o conteúdo: continua sendo um gato.



Original e rotacionado: ainda é “gato”.

Camadas Convolucionais

Camadas convolucionais trazem diversos benefícios:

- Processam regiões da imagem separadamente, compartilhando pesos.
- Usam menos parâmetros do que camadas totalmente conectadas.
- Exploram a correlação entre pixels próximos.
- Não precisam reaprender padrões em locais diferentes da imagem.

Visão Geral

Invariância

- Uma função f aplicada a uma imagem x é dita **invariante** a uma transformação t se:

$$f\big[t[x]\big] = f[x]$$

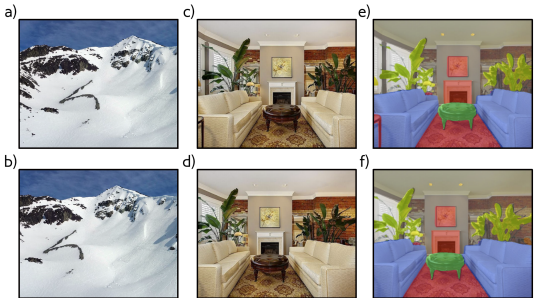
- Ou seja, independente da transformação aplicada na entrada, o resultado é o mesmo.
- Exemplo: classificar a imagem como “gato” não deveria depender de o gato estar à esquerda ou à direita do quadro.

Equivariância

- Uma função f aplicada a uma imagem x é dita **equivariante** a uma transformação t se:

$$f\big[t[x]\big] = t\big[f[x]\big]$$

- Ou seja, aplicar a transformação na entrada equivale a aplicá-la na saída.
- Exemplo: transladar a imagem de entrada deve resultar na mesma segmentação transladada.



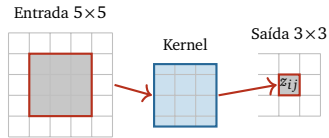
Visão Geral

Convolução 2D

- A convolução transforma uma imagem de entrada x em uma imagem de saída z. Cada posição (i, j) resulta de uma combinação linear de uma janela 3×3 ao redor:

$$z_{i,j} = \sum_{m=1}^{3} \sum_{n=1}^{3} w_{m,n}\, x_{i+m-2,\, j+n-2}$$

- Os pesos são **compartilhados** por toda a entrada.
- Esse conjunto de pesos é chamado de *kernel* ou *filtro*.

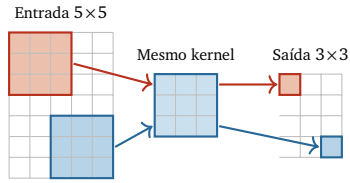


Compartilhamento de Pesos

- **O mesmo kernel** $w_{m,n}$ é deslizado por toda a entrada.
- O diagrama anterior mostrava apenas uma aplicação do kernel; aqui ilustramos a aplicação em duas posições distintas.
- Para qualquer posição (i, j):

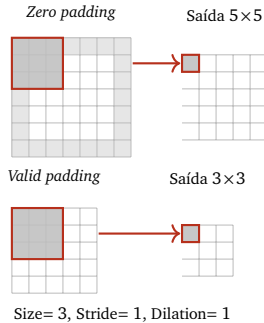
$$z_{i,j} = \sum_{m,n} w_{m,n}\, x_{i+m-2,\, j+n-2}$$

- Não existem pesos próprios por $x_{i,j}$, os mesmos nove valores reaparecem em cada janela.



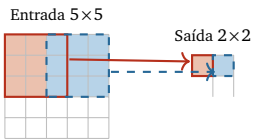
Padding

- Como lidar com os extremos da entrada?
  - *Zero padding*: completar a entrada com zeros.
  - *Valid padding*: aplicar a convolução apenas onde o *kernel* cabe inteiramente.
  - Entrada “circular” ou periódica (pouco utilizado).



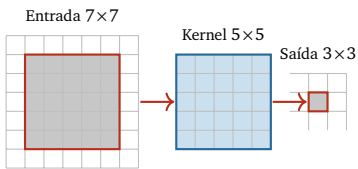
Stride

- *Stride* é o deslocamento do *kernel* a cada passo.
- Pode ser interpretado como *downsampling* quando maior que 1.
- Exemplo: Size= 3, Stride= 2 produz uma saída com aproximadamente metade da resolução da entrada em cada dimensão.



Kernel Size

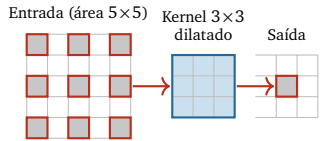
- *Kernel size* é o tamanho do filtro.
- Permite integrar informações espaciais mais distantes.
  - Mas aumenta o número de pesos a aprender.
- Exemplo: Size= 5, Stride= 1 olha uma janela 5×5 ao redor de cada posição para produzir cada saída.



Size= 5, Stride= 1, Dilation= 1

Dilation

- *Dilation* integra informações mais distantes **sem** aumento do número de parâmetros.
  - Equivalente a aumentar o *kernel* deixando alguns pesos zerados.
- Exemplo (convenção dos frameworks como PyTorch): Size= 3, Stride= 1, Dilation= 2 olha posições ($i\pm2$, $j\pm2$) em vez de ($i\pm1$, $j\pm1$).
  - Nessa convenção, Dilation= 1 corresponde ao kernel denso (sem dilatação) e Dilation= d insere $d-1$ zeros entre os pesos em cada eixo.



Size= 3, Stride= 1, Dilation= 2

Visão Geral

Camada Convolucional

- Uma camada convolucional computa a convolução, soma o *bias* e aplica uma função de ativação:

$$h_{i,j} = \varphi\left[b + \sum_{m=1}^{3}\sum_{n=1}^{3} w_{m,n}\, x_{i+m-2,\, j+n-2}\right]$$

- b e os nove pesos $w_{m,n}$ do *kernel* são os parâmetros aprendidos.
- A soma cobre toda a janela $K \times K$ ao redor de (i, j).

Camada Convolucional vs Totalmente Conectada

- A camada convolucional é um caso especial de camada totalmente conectada.

$$h_{i,j} = \varphi\left[b + \sum_{m=1}^{3}\sum_{n=1}^{3} w_{m,n}\, x_{i+m-2,\, j+n-2}\right]$$

Convolucional

$$h_i = \varphi\left[b_i + \sum_{j=1}^{D} w_{ij}\, x_j\right]$$

Totalmente Conectada

- Na camada totalmente conectada, x é a imagem achatada em um vetor de D elementos.
- A convolução é uma camada totalmente conectada com pesos compartilhados e a maior parte dos pesos zerados (esparsa e estruturada).

Múltiplos Canais e *Feature Maps*

- Tipicamente temos diversos canais de entrada e saída em uma camada convolucional.
- As ativações resultantes de um filtro são armazenadas em unidades ocultas tipicamente denominadas *feature maps* (ou canais).
- A aplicação de filtros diferentes na mesma entrada gera *feature maps* diferentes.
- Em camadas posteriores, cada nova convolução atua sobre todos os *feature maps* simultaneamente.

Parâmetros Treináveis

- Considerando C_i canais de entrada e C_o canais de saída, com *kernels* $K \times K$:

$$w \in \mathbb{R}^{C_i \times C_o \times K \times K}, \qquad b \in \mathbb{R}^{C_o}$$

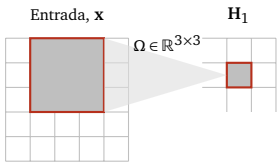
Exemplo

Para 3 canais de entrada, 5 canais de saída e *kernels* 3×3 temos $3 \cdot 5 \cdot 3 \cdot 3 = 135$ pesos nos *kernels*, mais 5 *bias*.

Visão Geral

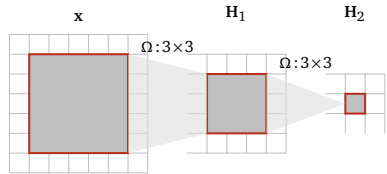
Campo Receptivo (*Receptive Field*)

- Ao realizar uma convolução, cada unidade oculta do *feature map* é gerada levando em conta apenas algumas das entradas.
  - Damos o nome de **Campo Receptivo** à região da entrada considerada por uma unidade oculta.
- Para um *kernel* 3×3, cada unidade de $\mathbf{H}_1$ enxerga uma janela 3×3 da entrada.



Crescimento do Campo Receptivo

- Empilhando convoluções sucessivas, o campo receptivo cresce camada a camada.
  - Cada unidade de $\mathbf{H}_2$ enxerga uma janela 3×3 de $\mathbf{H}_1$, que por sua vez vem de uma janela 5×5 na entrada.
  - Isso permite construir representações cada vez mais "globais" da imagem.
- Um campo receptivo pequeno pode perder informações relevantes.
- Convoluções com *stride* maior que 1 aceleram esse crescimento, ao custo de reduzir a resolução espacial.



Fórmula do Campo Receptivo

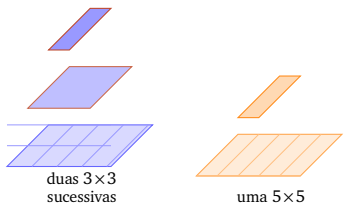
- Para uma sequência de camadas com tamanho de kernel K_l e *stride* S_l, o campo receptivo RF_l na camada l obedece à recursão:

$$RF_l = RF_{l-1} + (K_l - 1) \prod_{i<l} S_i, \qquad RF_0 = 1.$$

- Cada nova camada acrescenta $K_l - 1$ posições do espaço original, multiplicadas pelo produto acumulado dos *strides* anteriores.
- **Exemplo**, três convoluções 3×3 com $S = 1$:
  - $RF_1 = 1 + (3 - 1) \cdot 1 = 3$.
  - $RF_2 = 3 + (3 - 1) \cdot 1 = 5$.
  - $RF_3 = 5 + (3 - 1) \cdot 1 = 7$.
- Se a segunda camada usa $S_2 = 2$, então $RF_3 = 3 + (3 - 1) \cdot 1 + (3 - 1) \cdot 2 = 9$, *stride* acelera o crescimento do campo receptivo.

Duas 3×3 ou uma 5×5?

- Mesmo campo receptivo, diferentes propriedades.
- Resposta: **duas convoluções** 3×3[a].
  - Menos parâmetros: $2 \cdot (3 \cdot 3) = 18$ pesos contra $5 \cdot 5 = 25$.
  - Mais não linearidade: duas ativações em vez de uma.



[a] Karen Simonyan e Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". Em: arXiv preprint arXiv:1409.1556 (2014).

Visão Geral

Treinamento

- *Backprop* funciona naturalmente para convoluções.
  - As derivadas parciais com relação a cada peso compartilhado precisam ser **acumuladas** (somadas) sobre todas as posições onde esse peso atua.

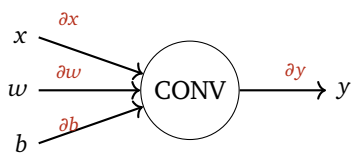
Para um *kernel* 2×2 com pesos $W_{m,n}$ aplicado a uma entrada $X_{i,j}$, a contribuição do gradiente é:

$$\partial W_{1,1} = \sum_{i,j} X_{i,j}\, \partial h_{i,j}$$

$$\partial W_{1,2} = \sum_{i,j} X_{i,j+1}\, \partial h_{i,j}$$

$$\partial W_{2,1} = \sum_{i,j} X_{i+1,j}\, \partial h_{i,j}$$

$$\partial W_{2,2} = \sum_{i,j} X_{i+1,j+1}\, \partial h_{i,j}$$



O Resultado do Treinamento

- Os *feature maps* das primeiras camadas mostram arestas, cantos e formas primitivas.
- Em camadas mais internas vemos formas mais complexas: olhos, narizes, partes de objetos.
  - Repare que aumenta também o campo receptivo.
- Esse comportamento hierárquico não é programado: emerge naturalmente do treinamento.

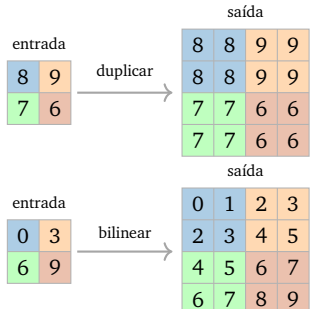
Visão Geral

Downsampling

- Subamostrar espacialmente a imagem pode ter o efeito de aumentar o campo receptivo de uma unidade oculta.
- As principais formas de subamostragem em CNN são:
  1. Usar *stride* maior que 1 na convolução.
  2. *Max pooling*: mantém apenas o maior valor de cada janela.
  3. *Average pooling*: usa a média dos valores de cada janela.

Upsampling

- Sobreamostrar espacialmente pode ser útil quando a saída da rede também é uma imagem (segmentação, super resolução, geração).
- As principais formas de sobreamostragem em CNN são:
  1. Duplicar os valores conhecidos.
  2. *Max unpooling*: insere o valor na posição original do *max pooling* e zera as demais.
  3. Interpolação bilinear.
  4. Convolução transposta.



Convolução Transposta

- Operação aprendível para aumentar a resolução espacial.
- Cada elemento da entrada multiplica o *kernel* inteiro e o resultado é depositado em uma região da saída, com sobreposições somadas.
- Parâmetros: *kernel size*, *padding*, *stride*. *Stride* maior que 1 **expande** a saída.

Convolução Transposta: Exemplo

- Filtro 3×3, *padding* $= 1$, *stride* $= 2$. Cada célula da entrada projeta o *kernel* inteiro multiplicado por seu valor; sobreposições são somadas.

entrada **X**

2	1
3	2

kernel **w**

1	2	1
2	0	1
0	2	1

resultado 4×4

0	2		
4	2		

Convolução Transposta: Exemplo

- Filtro 3×3, *padding* $= 1$, *stride* $= 2$. Cada célula da entrada projeta o *kernel* inteiro multiplicado por seu valor; sobreposições são somadas.

entrada **X**

2	1
3	2

kernel **w**

1	2	1
2	0	1
0	2	1

resultado 4×4

0	2+2	0	1
4	2+0	2	1

Convolução Transposta: Exemplo

- Filtro 3×3, *padding* $= 1$, *stride* $= 2$. Cada célula da entrada projeta o *kernel* inteiro multiplicado por seu valor; sobreposições são somadas.

entrada **X**

2	1
3	2

kernel **w**

1	2	1
2	0	1
0	2	1

resultado 4×4

0	2+2	0	1
4+6	2+0+3	2	1
0	3		
6	3		

Convolução Transposta: Exemplo

- Filtro 3×3, *padding* $= 1$, *stride* $= 2$. Cada célula da entrada projeta o *kernel* inteiro multiplicado por seu valor; sobreposições são somadas.

entrada **X**

2	1
3	2

kernel **w**

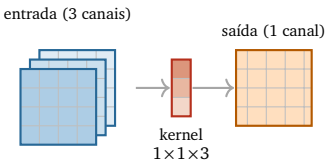
1	2	1
2	0	1
0	2	1

resultado 4×4

0	2+2	0	1
4+6	2+0+3+2	2+4	1+2
0	3+4	0	2
6	3+0	4	2

Convolução 1×1

- Algumas vezes queremos mudar o número de canais sem agrupamento espacial.
  - Nesses casos podemos aplicar a convolução 1×1.
- Cada posição espacial é independente: cada saída $h'_{i,j,c}$ é uma combinação linear dos canais de entrada $h_{i,j,1\ldots C_i}$.
- Útil para reduzir ou expandir a profundidade do tensor sem alterar largura e altura.



Visão Geral

Implementação: Versão Ingênua

- Itera explicitamente por linhas, colunas, canais e *kernel*:

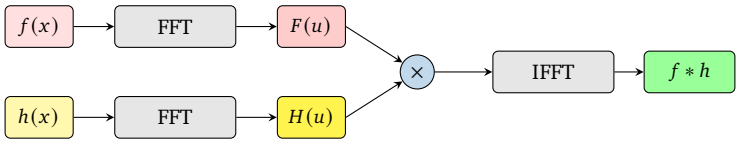
```
# Versao com um unico canal de saida (C_o = 1).
# Para multiplos canais de saida bastaria envolver tudo
# em mais um loop sobre co e indexar feat_map_out[m, n, co].
for m in range(0, M):           # linhas
    for n in range(0, N):       # colunas
        for c in range(0, C):   # canais de entrada
            for i in range(0, K):     # tamanho kernel
                for j in range(0, K): # tamanho kernel
                    feat_map_out[m, n] += kernel[i, j, c] * feat_map_in[m+i, n+j, c]
        feat_map_out[m, n] += bias
```

- Simples de entender, mas extremamente lenta e sem aproveitar paralelismo de hardware.

Implementação: via Transformada de Fourier

- Versão complexa e raramente utilizada na prática.
- Convolução no domínio espacial é equivalente a multiplicação no domínio da frequência:

$$g(x) = f(x) * h(x) \qquad \Longleftrightarrow \qquad G(u) = F(u)\,H(u)$$



- Útil para convoluções com *kernels* grandes (acima de 11×11)[3].

[3] Nicolas Vasilache et al. "Fast convolutional nets with fbfft: A GPU performance evaluation". Em: arXiv preprint arXiv:1412.7580 (2014).

Implementação: `im2col` e Matriz de Toeplitz

- Versão utilizada na prática em GPUs e bibliotecas otimizadas.
- Existe uma forma inteligente de organizar a matriz de pesos para obter o resultado da convolução com uma única multiplicação de matrizes.
- Uma **matriz de Toeplitz** é uma matriz cujas diagonais principal e subdiagonais são constantes:

$$\begin{bmatrix} 5 & 27 & 2 & -56 & 0 & 1 \\ -3 & 5 & 27 & 2 & -56 & 0 \\ 8 & -3 & 5 & 27 & 2 & -56 \\ 1 & 8 & -3 & 5 & 27 & 2 \\ -9 & 1 & 8 & -3 & 5 & 27 \\ 3 & -9 & 1 & 8 & -3 & 5 \end{bmatrix}$$

`im2col`: Passo 1 – Entrada e filtro

- Definimos a entrada X (3×3) e o filtro w (2×2):

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix} \qquad w = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

`im2col`: Passo 2 – Tamanho da saída

- Modo *full*: $(X_h + w_h - 1) \times (X_w + w_w - 1)$. Para X 3×3 e w 2×2:

$$(3 + 2 - 1) \times (3 + 2 - 1) = 4 \times 4$$

- Convenções de tamanho:
  - **full** (*signal processing*): saída maior que a entrada; considera todas as janelas em que entrada e kernel se sobrepõem em pelo menos um pixel.
  - **valid** (típico em CNNs): saída $(X_h - w_h + 1) \times (X_w - w_w + 1)$; considera apenas janelas inteiramente dentro da entrada.
  - **same**: com *padding* adequado, mantém o tamanho original da entrada.

`im2col`: Passo 3 – Zero padding do filtro

- Aplicamos *zero padding* no filtro até atingir o tamanho da saída 4×4:

$$w = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ w_{1,1} & w_{1,2} & 0 & 0 \\ w_{2,1} & w_{2,2} & 0 & 0 \end{bmatrix}$$

`im2col`: Passo 4 – Matrizes de Toeplitz por linha

- Para cada linha do filtro paddado, construímos uma matriz de Toeplitz F_k (tamanho saída$_h \times$ entrada$_w = 4 \times 3$). Começamos pela última linha.

F_0

$w_{2,1}$	0	0
$w_{2,2}$	$w_{2,1}$	0
0	$w_{2,2}$	$w_{2,1}$
0	0	$w_{2,2}$

F_1

$w_{1,1}$	0	0
$w_{1,2}$	$w_{1,1}$	0
0	$w_{1,2}$	$w_{1,1}$
0	0	$w_{1,2}$

F_2

0	0	0
0	0	0
0	0	0
0	0	0

F_3

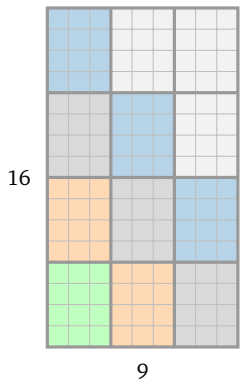
0	0	0
0	0	0
0	0	0
0	0	0

- F_0 vem da última linha do filtro paddado, F_1 da penúltima, e assim por diante.

`im2col`: Passo 5 – Matriz de bloco Toeplitz

- Empilhamos as F_k em uma matriz por blocos. O tamanho final é 16×9.

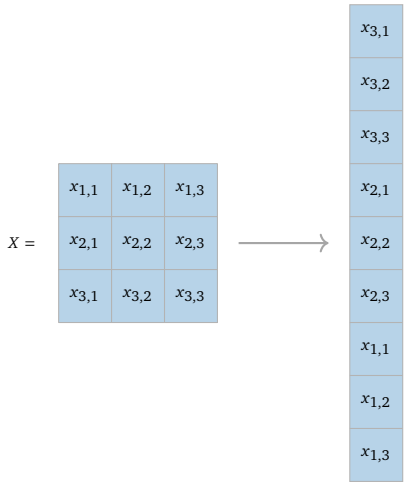
$$\text{Bloco Toeplitz} = \begin{bmatrix} F_0 & 0 & 0 \\ F_1 & F_0 & 0 \\ F_2 & F_1 & F_0 \\ F_3 & F_2 & F_1 \end{bmatrix}$$



- Colunas = linhas da imagem original (3); linhas = linhas da imagem de saída vezes a largura ($4 \cdot 4 = 16$).

`im2col`: Passo 6 – Vetorização da entrada

- Vetorizamos X concatenando as linhas **de baixo para cima**:



`im2col`: Passo 7 – Multiplicação e *reshape*

- Multiplicamos a matriz de bloco Toeplitz pelo vetor de entrada e fazemos *reshape* para a forma da saída:

$$\underbrace{T}_{16\times 9} \cdot \underbrace{\vec{X}}_{9\times 1} = \underbrace{\vec{y}}_{16\times 1} \xrightarrow{\text{reshape}} \underbrace{Y}_{4\times 4}$$

- Cada linha do produto $T\vec{X}$ já contém o resultado de uma posição da convolução; basta reorganizar em uma matriz 4×4.
- O resultado é exatamente igual ao da convolução tradicional, calculado em **uma única multiplicação de matrizes**.

Por que `im2col` é a escolha prática

- Reduz a convolução a uma multiplicação de matrizes (`GEMM`), o tipo de operação que GPUs e bibliotecas BLAS otimizam ao máximo.
- Permite usar a mesma rotina para qualquer combinação de *kernel*, *stride* e *padding*, alterando apenas o layout dos dados.
- Custa memória extra para armazenar as “cópias” das janelas, mas o ganho em velocidade compensa em larga escala.

Visão Geral

Referências I

- Sugere se **fortemente** a leitura do Capítulo 10 de *Understanding Deep Learning*[4].
  - Disponível em `https://udlbook.github.io/udlbook/`.
- Aula baseada também nos seguintes materiais:
  - Capítulo 10 de *Understanding Deep Learning*, Simon J. D. Prince.
  - *Machine Learning* @ Berkeley.
  - `https://github.com/alisaaalehi/convolution_as_multiplication`.
  - `https://github.com/vdumoulin/conv_arithmetic`.

[1] David H. Hubel e Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". Em: The Journal of Physiology 160.1 (1962), pp. 106–154.

[2] Alex Krizhevsky, Ilya Sutskever e Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". Em: NeurIPS. Vol. 25. 2012.

[3] Yann LeCun et al. "Handwritten digit recognition with a back-propagation network". Em: NeurIPS. Vol. 2. 1989.

[4] Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023.

[5] Karen Simonyan e Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". Em: arXiv preprint arXiv:1409.1556 (2014).

[6] Nicolas Vasilache et al. "Fast convolutional nets with fbfft: A GPU performance evaluation". Em: arXiv preprint arXiv:1412.7580 (2014).

[4] Simon J. D. Prince. Understanding Deep Learning. MIT Press, 2023.