

# Treinamento e Otimização de ConvNets

Gabriel Simões

Jônatas Wehrman

Rodrigo Barros

# Roteiro

- Transfer Learning
- Otimização de Convoluçãoes
- CPU vs. GPU
- Precisão de ponto-flutuante vs. desempenho

# *Transfer Learning*

“Precisamos utilizar uma base de dados gigante”

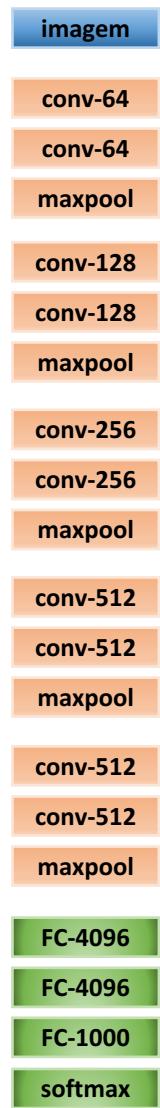


“Precisamos treinar uma ConvNet do zero”



*“Transfer learning (inductive transfer) é uma área do aprendizado de máquina que foca em armazenar conhecimento adquirido ao se resolver determinada tarefa para posterior aplicação em problema diferente mas relacionado.”*

# Transfer Learning em ConvNets

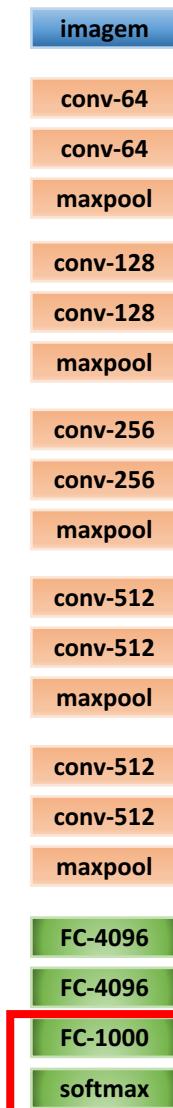


(Pré-) treinar no ImageNet (ILSVRC-2012)

<https://github.com/BVLC/caffe/wiki/Model-Zoo>

<https://github.com/tensorflow/models>

<https://github.com/pytorch/vision>

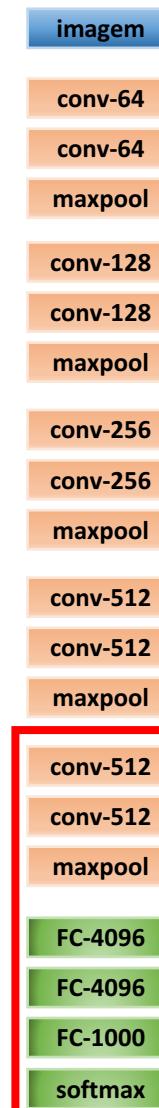


Se dataset for pequeno, utilizar rede como **extrator de features!**

"congelar" estas camadas

```
layer {  
    name: "conv5"  
    type: "Convolution"  
    bottom: "conv4"  
    top: "conv5"  
    param {  
        lr_mult: 0  
        decay_mult: 0  
    }  
    param {  
        lr_mult: 0  
        decay_mult: 0  
    }  
    convolution_param {  
        num_output: 256  
        pad: 1  
        kernel_size: 3  
        group: 2  
    }  
}
```

Treinar aqui!  
(classificador linear ou jogar *features* num SVM)



Dataset médio: **fine-tuning!**

Quanto mais dados, mais rede pode ser re-treinada...

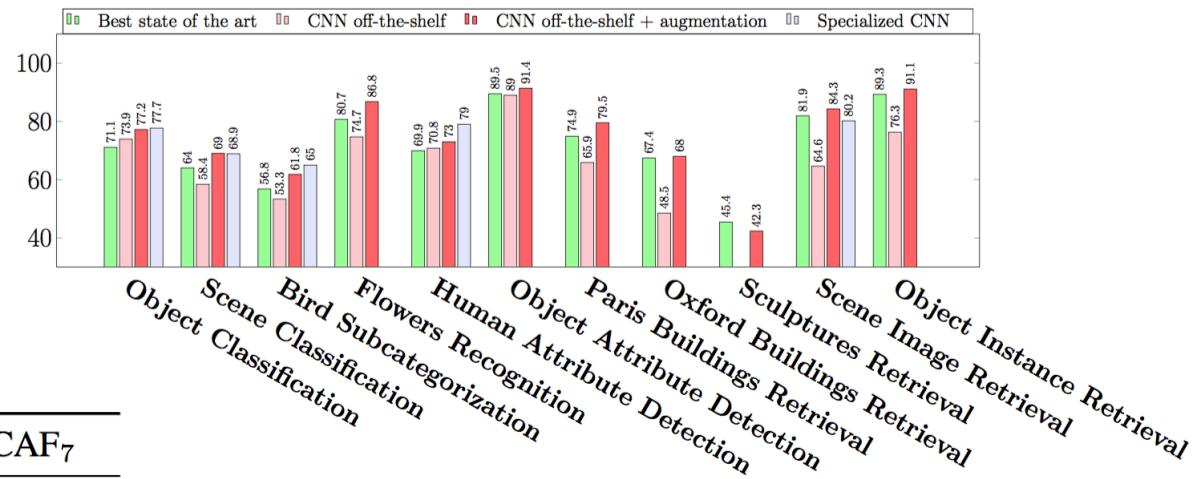
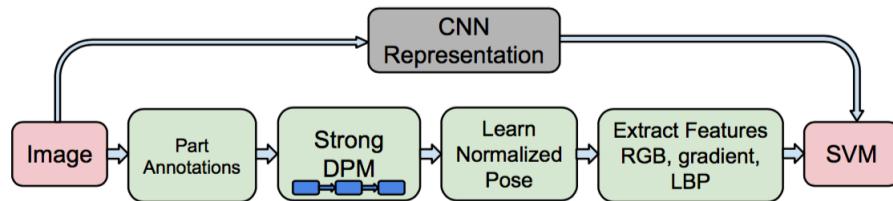
"congelar" estas camadas

Treinar aqui!

Dica: utilizar 1/10 da *learning rate* original na última camada e 1/100 nas demais camadas

# *Transfer Learning* em ConvNets

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, Stefan Carlsson. **CNN Features Off-the-Shelf: An Astounding Baseline for Recognition.** CVPR Workshops 2014, 512-519.



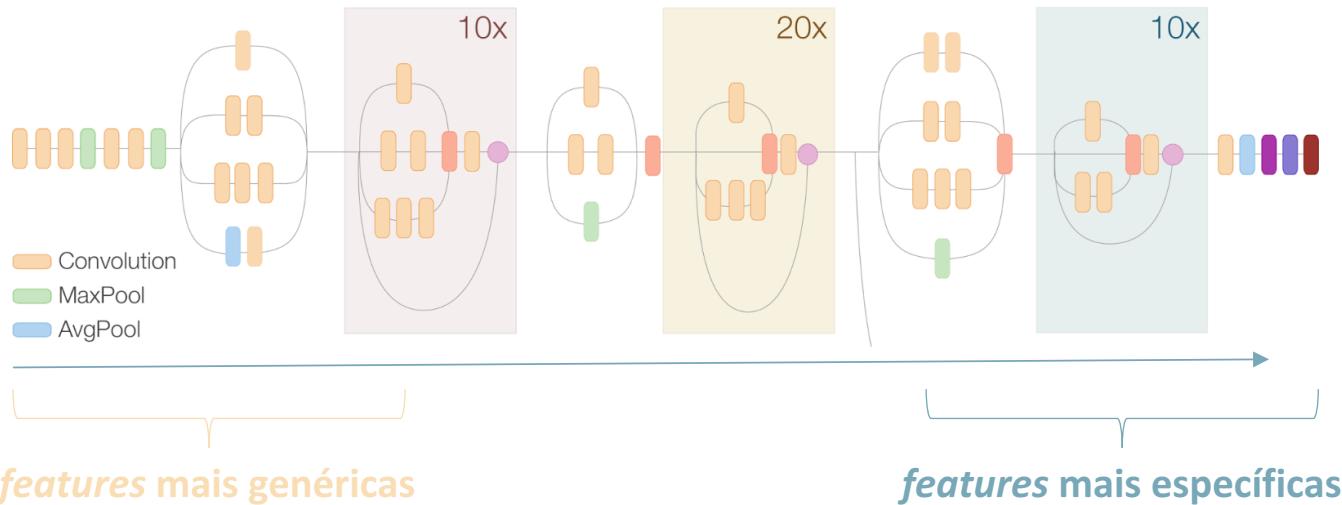
	DeCAF <sub>6</sub>	DeCAF <sub>7</sub>
LogReg	<b>40.94 ± 0.3</b>	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3
Xiao et al. (2010)		38.0

**Table 3.** Average accuracy per class on SUN-397 with 50 training samples and 50 test samples per class, across two hidden layers of the network and two classifiers. Our result from the training protocol/classifier combination with the best validation accuracy – Logistic Regression with DeCAF<sub>7</sub> – is shown in **bold**.

Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, Trevor Darrell.  
**DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition.** ICML 2014, 647-655.

# Transfer Learning em ConvNets

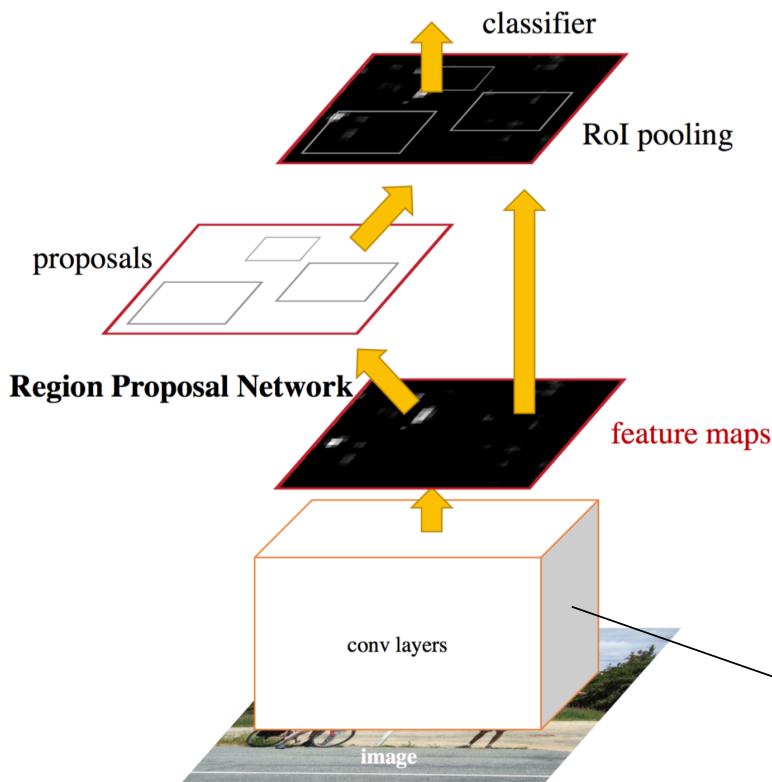
Compressed View



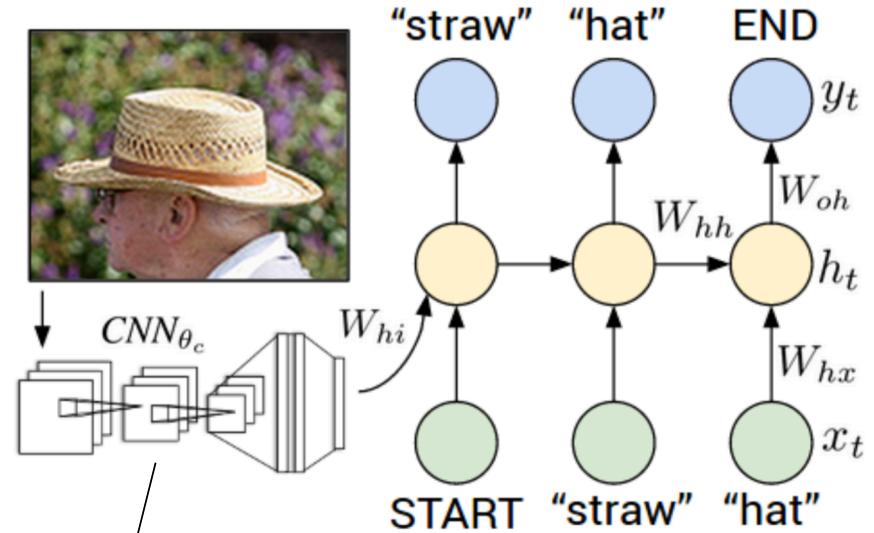
	Dados similares ao contexto em questão	Dados muito diferentes
Poucos dados	Usar um classificador linear na saída	
Muitos dados	Fine-tune algumas camadas...	Fine-tune algumas camadas a mais...

# Transfer Learning em ConvNets

Transfer Learning em ConvNets é a regra (e não exceção!!)



Andrej Karpathy, Fei-Fei Li. Deep visual-semantic alignments for generating image descriptions. CVPR 2015, 3128-3137



redes pré-treinadas no ImageNet!!

Shaoqing Ren, Kaiming He, Ross B. Girshick, Jian Sun.  
Faster R-CNN: Towards Real-Time Object Detection with  
Region Proposal Networks. NIPS 2015, 91-99

# *Transfer Learning* em ConvNets

## Conclusão:

1. Encontre dataset muito grande (>1M) com dados similares aos do seu interesse e (pré-) treine uma ConvNet lá! (ou pegue o modelo treinado de algum repositório de modelos, e.g. model zoo)
2. Faça transfer learning para o seu dataset (atenção para escolha da estratégia mais adequada de acordo com o que foi apresentado)

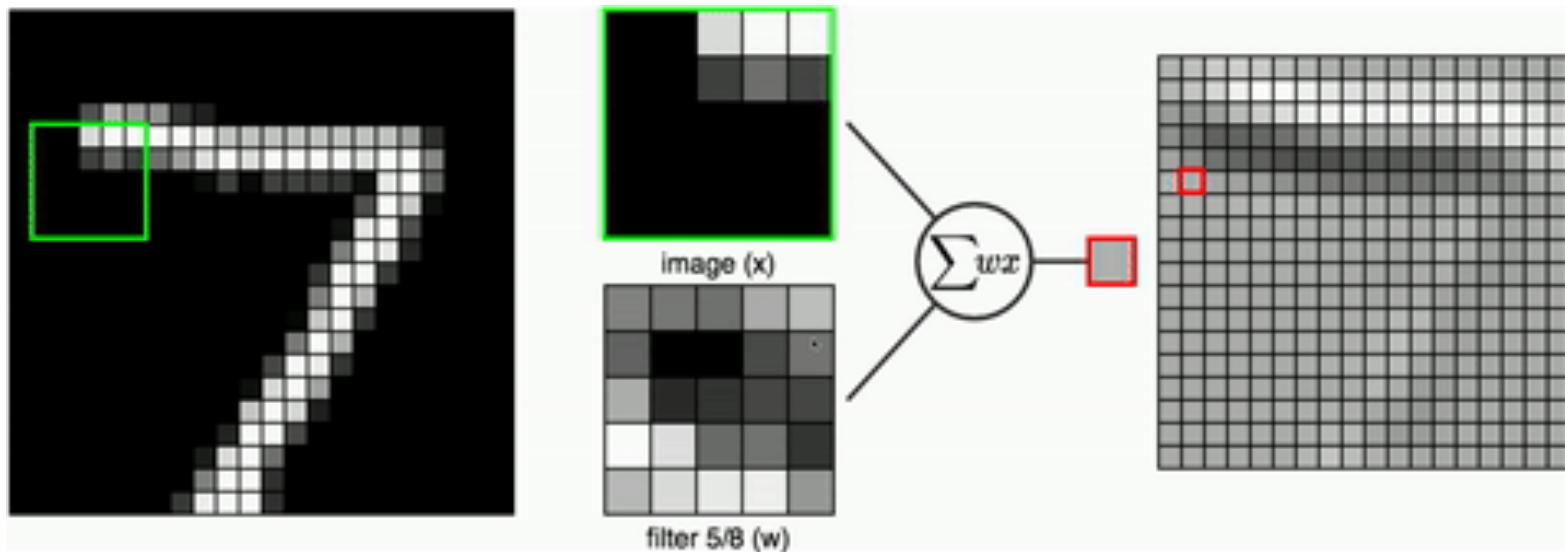
Caffe → <https://github.com/BVLC/caffe/wiki/Model-Zoo>

TensorFlow → <https://github.com/tensorflow/models>

PyTorch → <https://github.com/pytorch/vision>

# O Retorno das Convoluçãoções

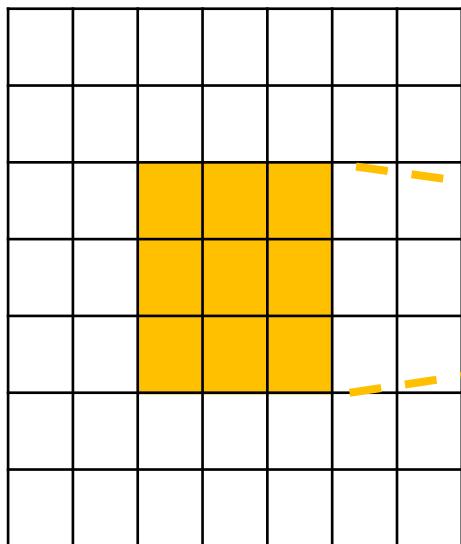
## Parte I: Otimização de Projeto



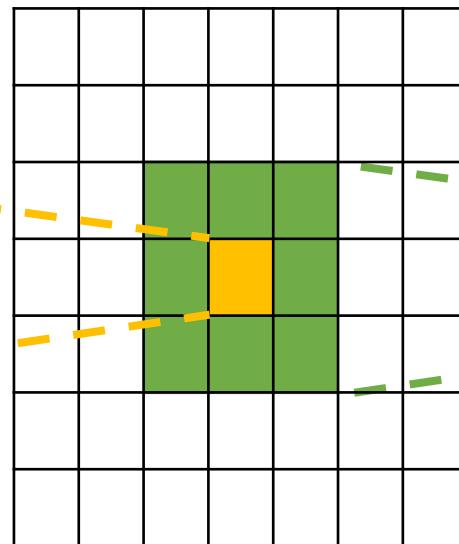
# Otimização de Projeto de Convoluçãoes

Imagine que existem duas convoluções sendo feitas em sequência (stride 1) ...

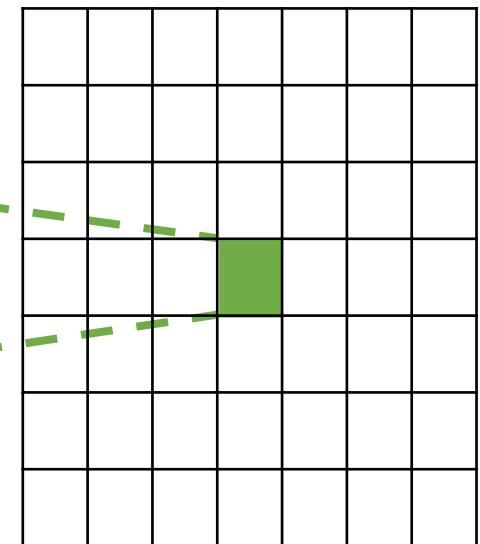
Cada neurônio enxerga uma região  $3 \times 3$  do mapa de ativação anterior



Entrada



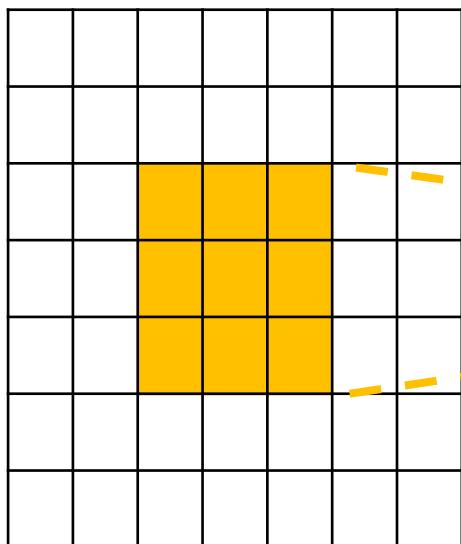
Convolução 1



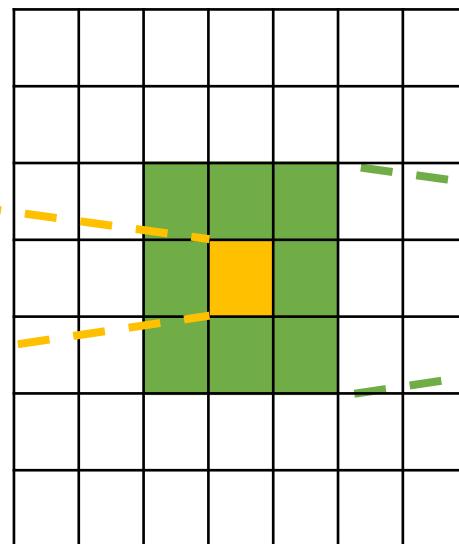
Convolução 2

# Otimização de Projeto de Convoluçãoes

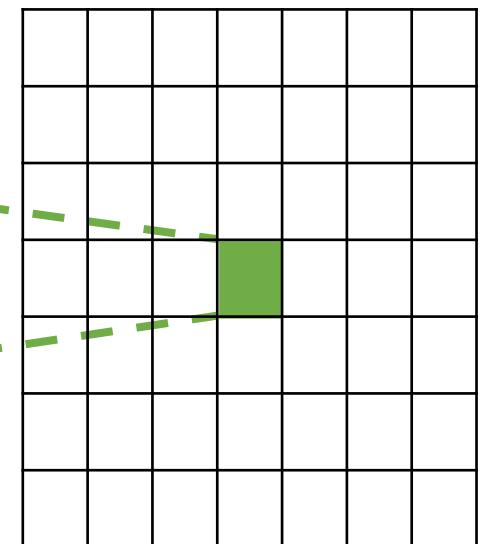
Qual o tamanho da região da entrada que um neurônio na segunda convolução enxerga?



Entrada



Convolução 1

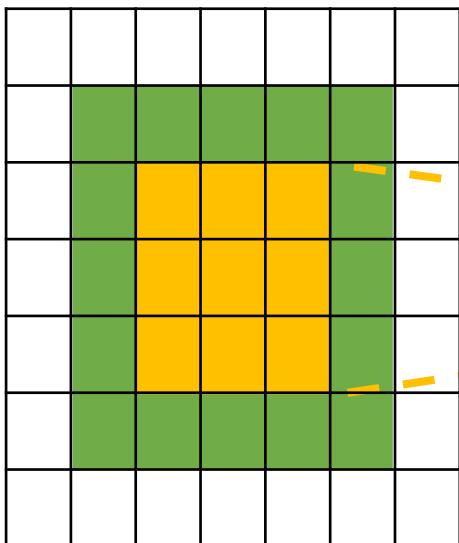


Convolução 2

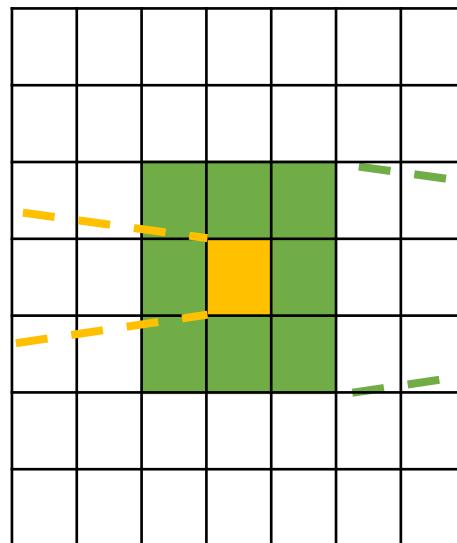
# Otimização de Projeto de Convoluçãoes

Qual o tamanho da região da entrada que um neurônio na segunda convolução enxerga?

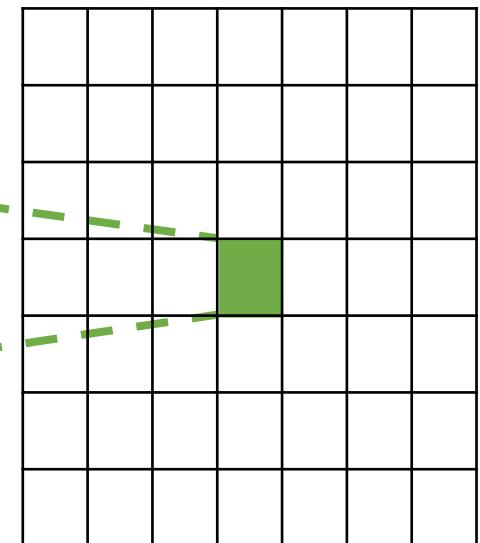
R:  $5 \times 5$



Entrada



Convolução 1



Convolução 2

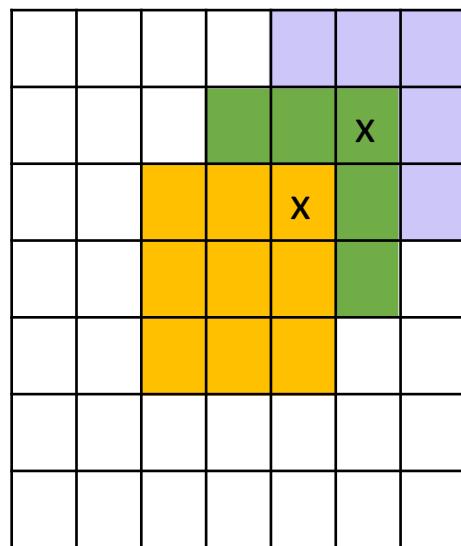
# Otimização de Projeto de Convoluçãoes

E se empilharmos 3 camadas de convolução  $3 \times 3$ , qual tamanho da região de entrada que os neurônios na terceira camada enxergarão?

# Otimização de Projeto de Convoluções

E se empilharmos 3 camadas de convolução  $3 \times 3$ , qual tamanho da região de entrada que os neurônios na terceira camada enxergarão?

R:  $7 \times 7$



Note que 3 convoluções  $3 \times 3$  tem poder de representação similar a uma convolução  $7 \times 7$

# Otimização de Projeto de Convoluçãoes

Assumindo uma entrada de tamanho  $L \times A \times C$  e que utilizaremos convoluções com  $C$  filtros para preservar a profundidade (*stride 1, zero-padding* para preservar  $L, A$ )

Uma convolução  $7 \times 7$

$$(7 \times 7 \times C) \times C$$

$$49C^2$$

3 convoluções  $3 \times 3$

$$[(3 \times 3 \times C) \times C] \times 3$$

Qtde de pesos?  
(sem *bias*)

$$27C^2$$

# Otimização de Projeto de Convoluçãoes

Assumindo uma entrada de tamanho  $L \times A \times C$  e que utilizaremos convoluções com  $C$  filtros para preservar a profundidade (*stride 1, zero-padding* para preservar  $L, A$ )

Uma convolução  $7 \times 7$

$$(7 \times 7 \times C) \times C$$

$$49C^2$$

Qtde de pesos?  
(sem *bias*)

3 convoluções  $3 \times 3$

$$[(3 \times 3 \times C) \times C] \times 3$$

$$27C^2$$

Menos parâmetros e mais não-linearidade!!!  
(excelente!!!)

# Otimização de Projeto de Convoluçãoes

Assumindo uma entrada de tamanho  $L \times A \times C$  e que utilizaremos convoluções com  $C$  filtros para preservar a profundidade (*stride 1, zero-padding* para preservar  $L, A$ )

Uma convolução  $7 \times 7$

$$(7 \times 7 \times C) \times C$$

$$49C^2$$

$$49LAC^2$$

3 convoluções  $3 \times 3$

$$[(3 \times 3 \times C) \times C] \times 3$$

$$27C^2$$

$$27LAC^2$$

Qtde de pesos?  
(sem *bias*)

Qtde de multiplicações-somas?

# Otimização de Projeto de Convoluçãoes

Assumindo uma entrada de tamanho  $L \times A \times C$  e que utilizaremos convoluções com  $C$  filtros para preservar a profundidade (*stride 1, zero-padding* para preservar  $L, A$ )

Uma convolução  $7 \times 7$

$$(7 \times 7 \times C) \times C$$

$$49C^2$$

Qtde de pesos?  
(sem *bias*)

3 convoluções  $3 \times 3$

$$[(3 \times 3 \times C) \times C] \times 3$$

$$27C^2$$

Qtde de multiplicações-somas?

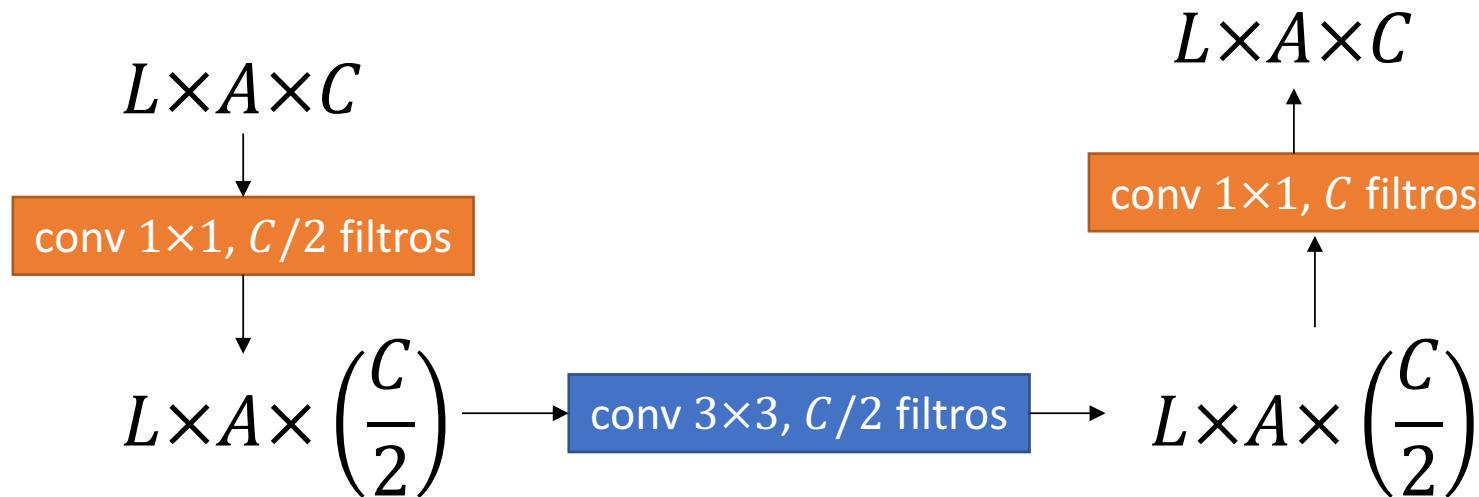
$$49LAC^2$$

$$27LAC^2$$

Menos computação e mais não-linearidade!!!  
(excelente!!!)

# Otimização de Projeto de Convoluções

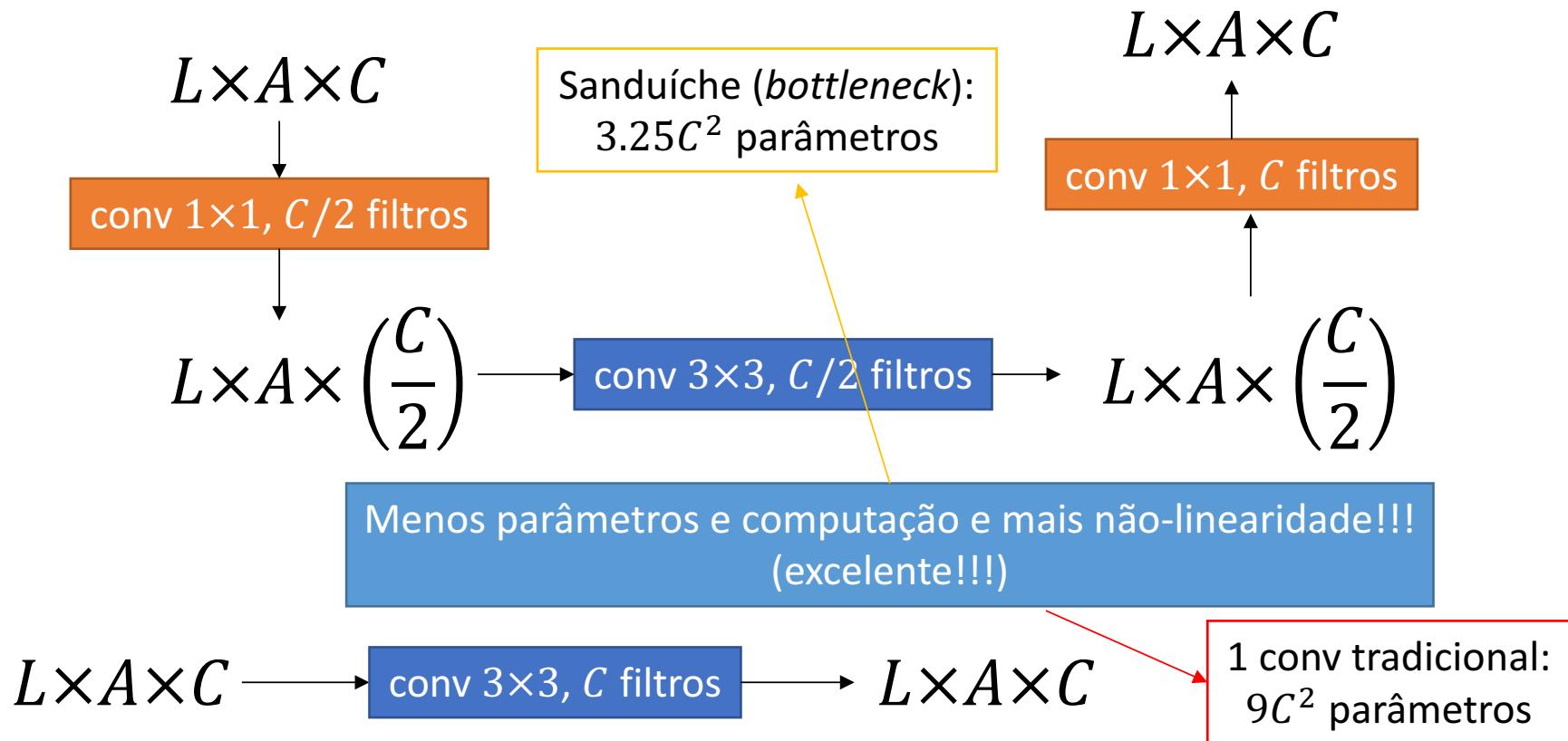
O poder dos filtros  $1 \times 1$



1. convolução  $1 \times 1$  *bottleneck* (gargalo) para redução de dimensionalidade
2. convolução  $3 \times 3$  em menor dimensão
3. convolução  $1 \times 1$  para restaurar dimensão original

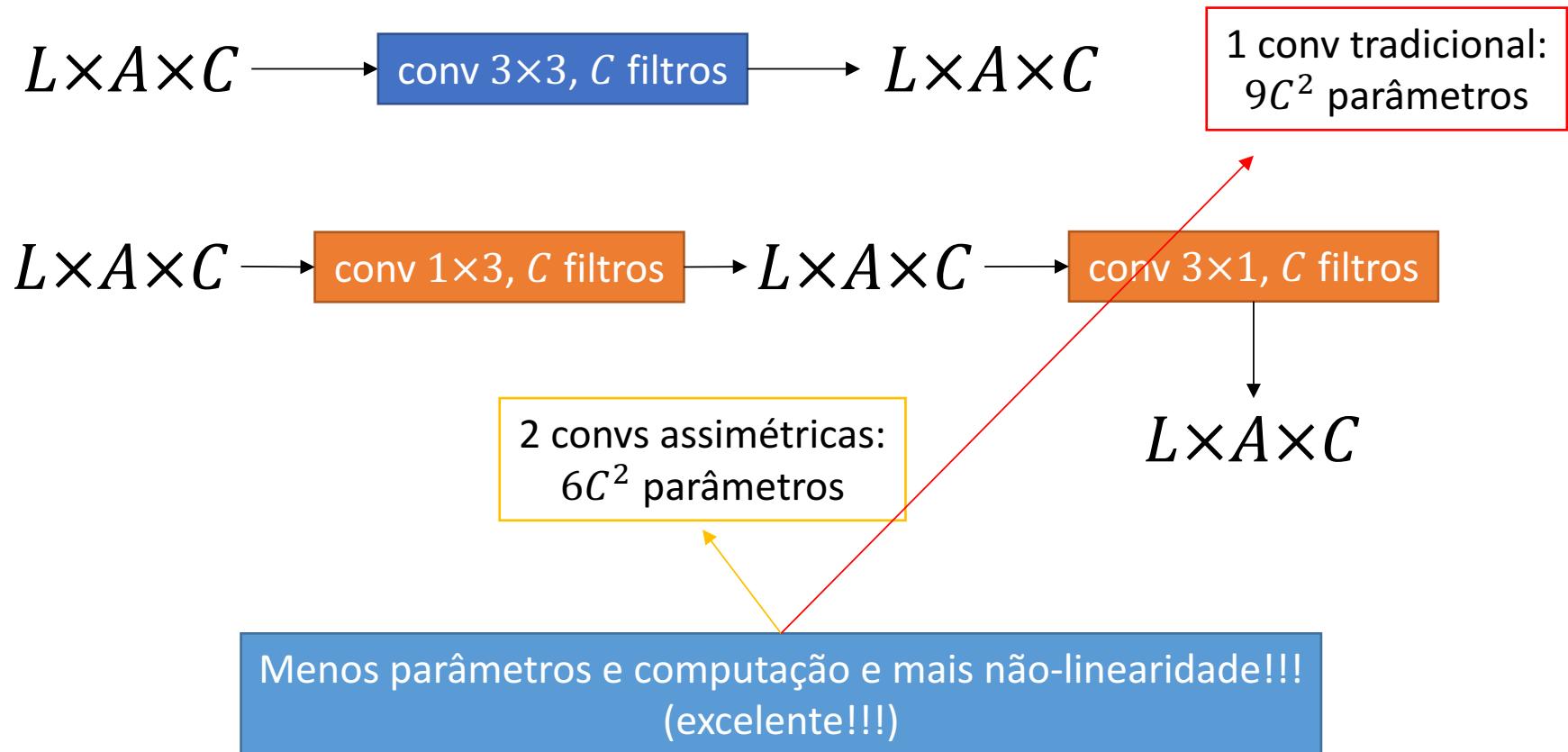
# Otimização de Projeto de Convoluçãoes

O poder dos filtros  $1 \times 1$



# Otimização de Projeto de Convoluçãoes

Filtros  $3 \times 3$ : conseguimos otimizar ainda mais?



# Otimização de Projeto de Convoluçãoes

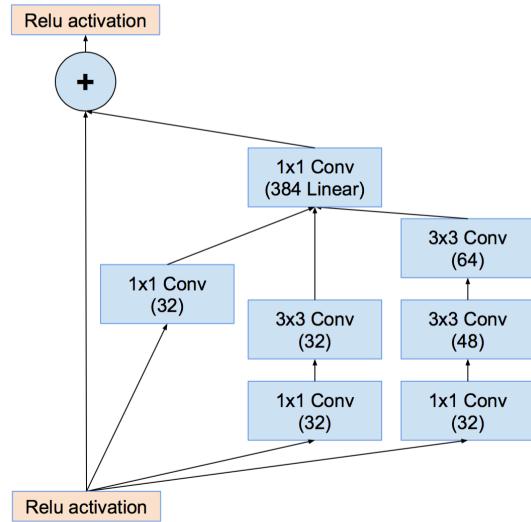


Figure 16. The schema for  $35 \times 35$  grid (Inception-ResNet-A) module of the Inception-ResNet-v2 network.

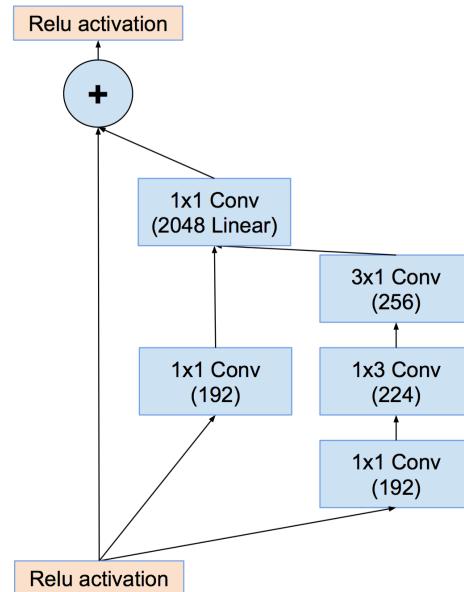


Figure 19. The schema for  $8 \times 8$  grid (Inception-ResNet-C) module of the Inception-ResNet-v2 network.

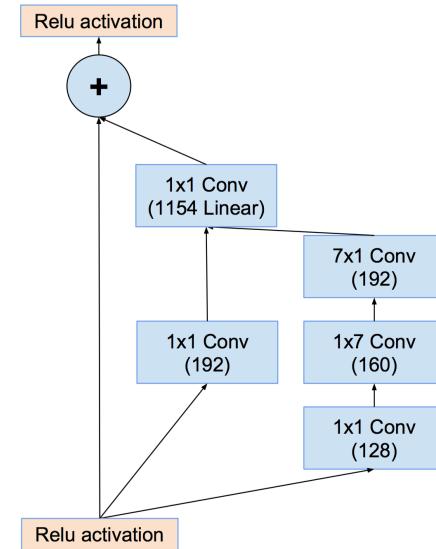


Figure 17. The schema for  $17 \times 17$  grid (Inception-ResNet-B) module of the Inception-ResNet-v2 network.

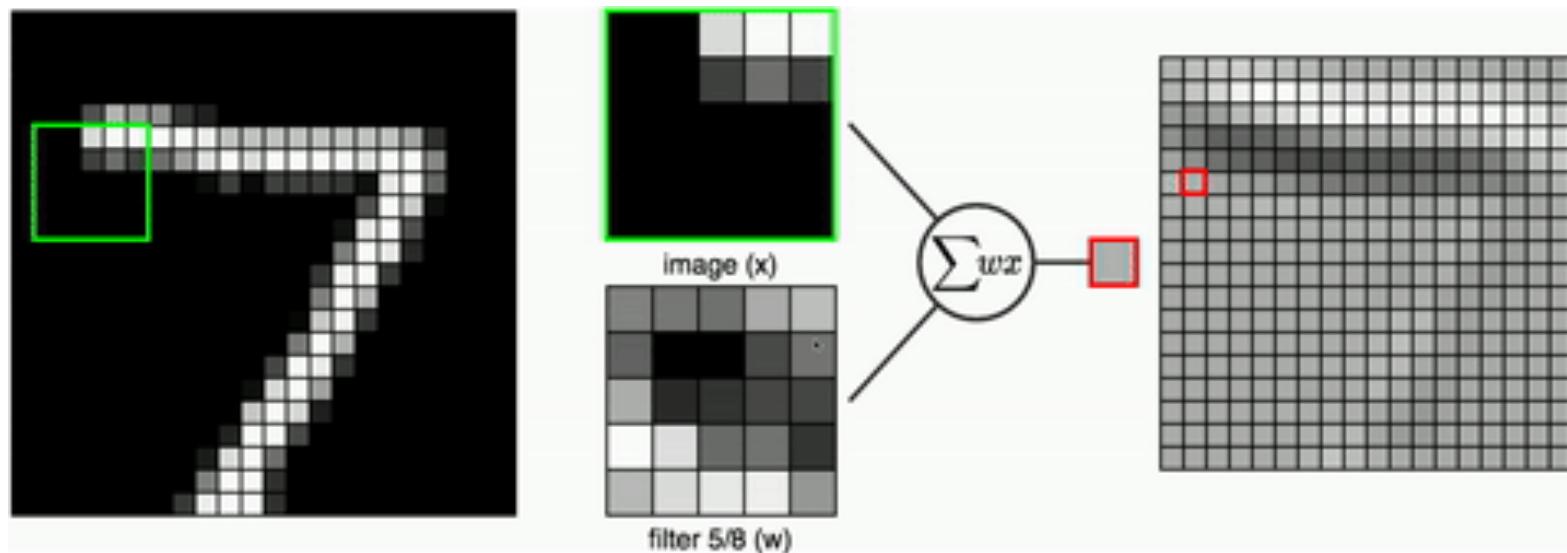
# Otimização de Projeto de Convoluções

## Conclusão:

1. Substitua convoluções grandes ( $5 \times 5$ ,  $7 \times 7$ ) por pilhas de convoluções  $3 \times 3$
2. Utilize convoluções  $1 \times 1$  "*bottleneck*" para ganhar eficiência!
3. Divida convoluções  $N \times N$  em  $N \times 1$  e  $1 \times N$
4. Todos os casos acima resultam em menos parâmetros e computação, com mais não-linearidade! =)

# O Retorno das Convoluçãoções

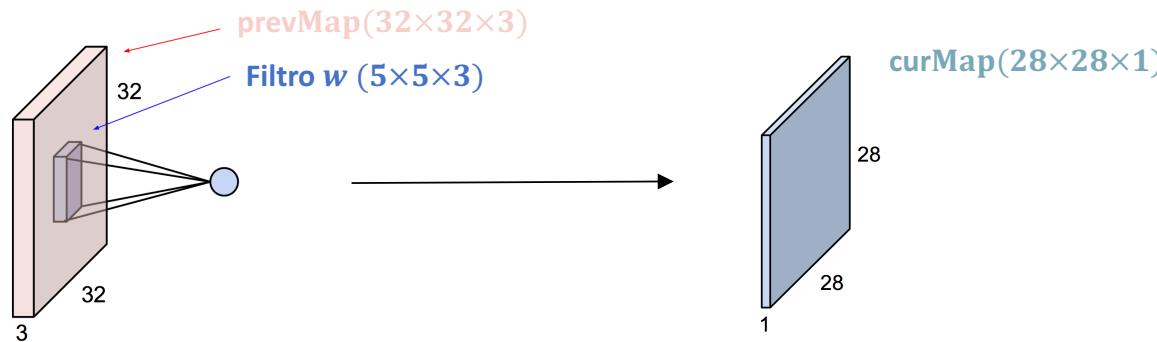
Parte II: Como implementar convoluções eficientes?



# Implementação Ingênua: 5 laços

```
for i in [0, A')  
    for j in [0, L')  
        for c in [0, C)  
            for l in [0, K)  
                for m in [0 to K)  
                    curMap[i, j] += w[l, m, c] × prevMap[i + l, j + m, c]  
curMap[i, j] += b
```

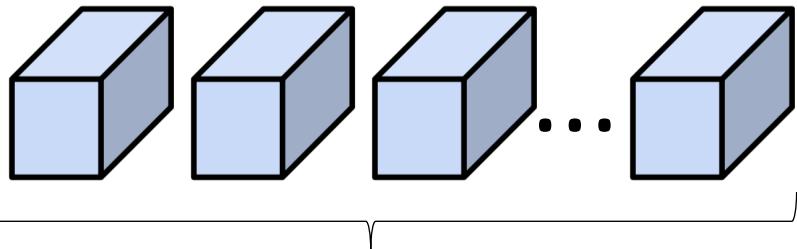
Pouco eficiente =(  
Conseguimos fazer melhor?



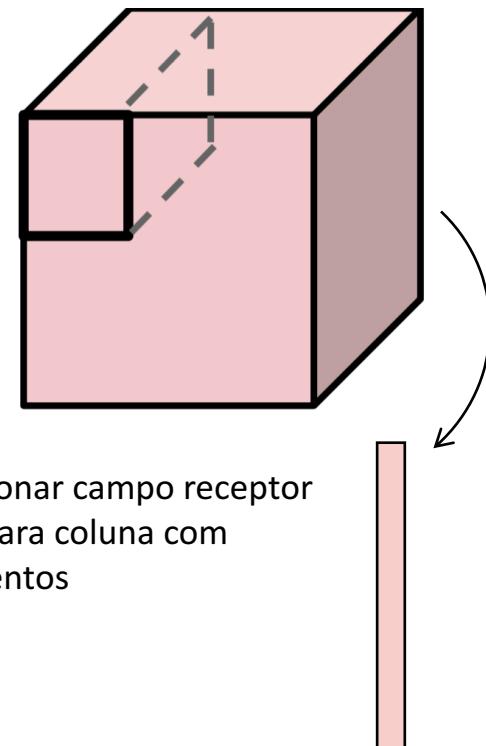
PS: estamos ignorando *stride* e utilização de múltiplos filtros

# im2col

- Ideia básica: transformar convolução em multiplicação de matrizes
- Motivação: existem rotinas otimizadas para multiplicar matrizes em múltiplas plataformas!



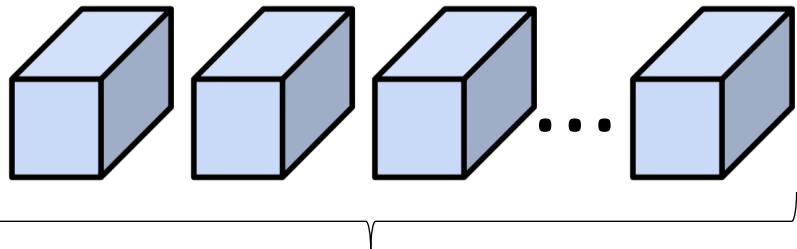
*Feature Map ( $N \times N \times C$ )*



Redimensionar campo receptor  
 $F \times F \times C$  para coluna com  
 $F^2 C$  elementos

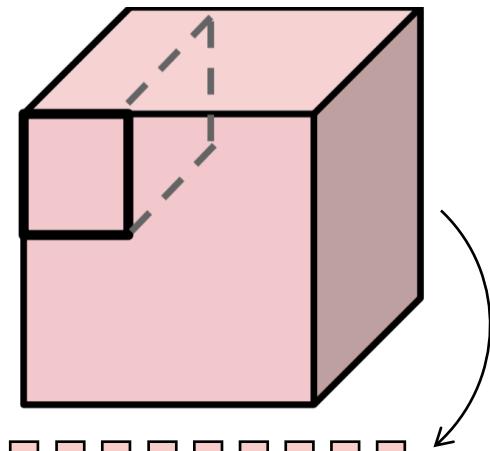
# im2col

- Ideia básica: transformar convolução em multiplicação de matrizes
- Motivação: existem rotinas otimizadas para multiplicar matrizes em múltiplas plataformas!

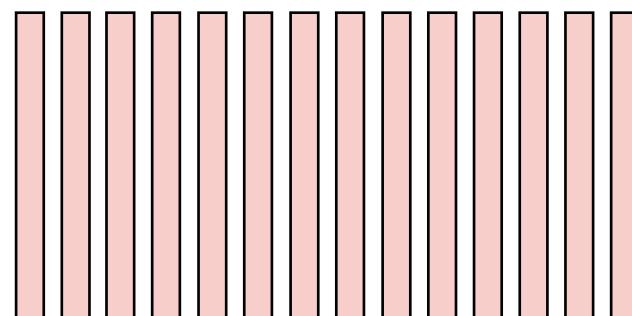


$D$  filtros ( $F \times F \times C$ )

*Feature Map ( $N \times N \times C$ )*



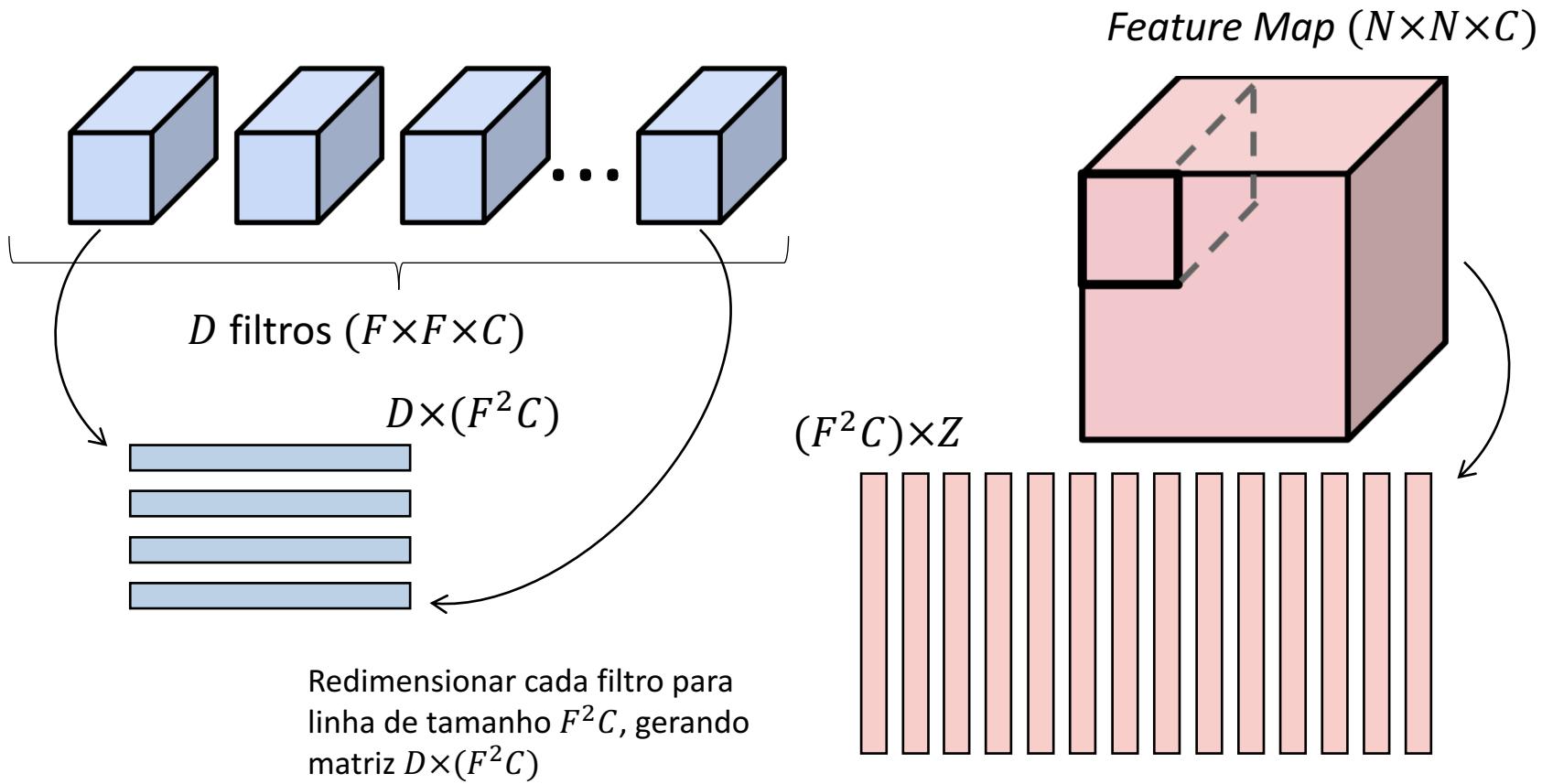
$(F^2 C) \times Z$



Repetir para todas colunas  
gerando matriz  $(F^2 C) \times Z$   
( $Z$  campos receptores)

# im2col

- Ideia básica: transformar convolução em multiplicação de matrizes
- Motivação: existem rotinas otimizadas para multiplicar matrizes em múltiplas plataformas!



# Detalhes de Implementação

Ache a CPU!  
(unidade de processamento central)



# Detalhes de Implementação

Ache a CPU!  
(unidade de processamento central)



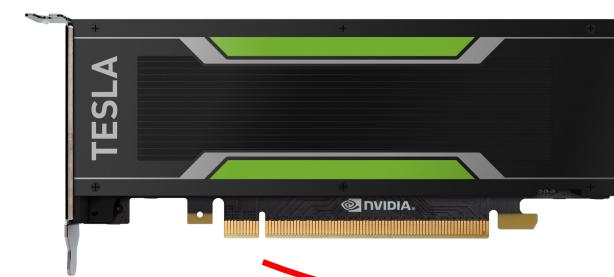
# Detalhes de Implementação

Ache a GPU!  
(unidade de processamento gráfico)

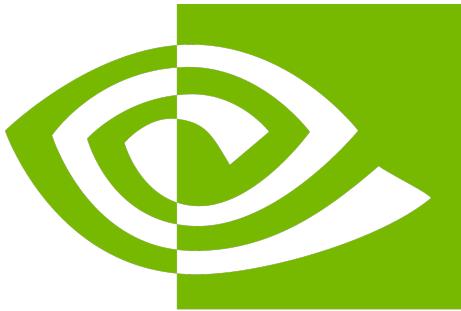


# Detalhes de Implementação

Ache a GPU!  
(unidade de processamento gráfico)



# Detalhes de Implementação

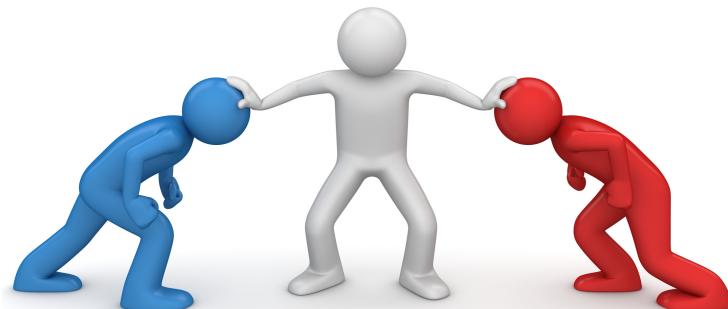


# NVIDIA®

vs

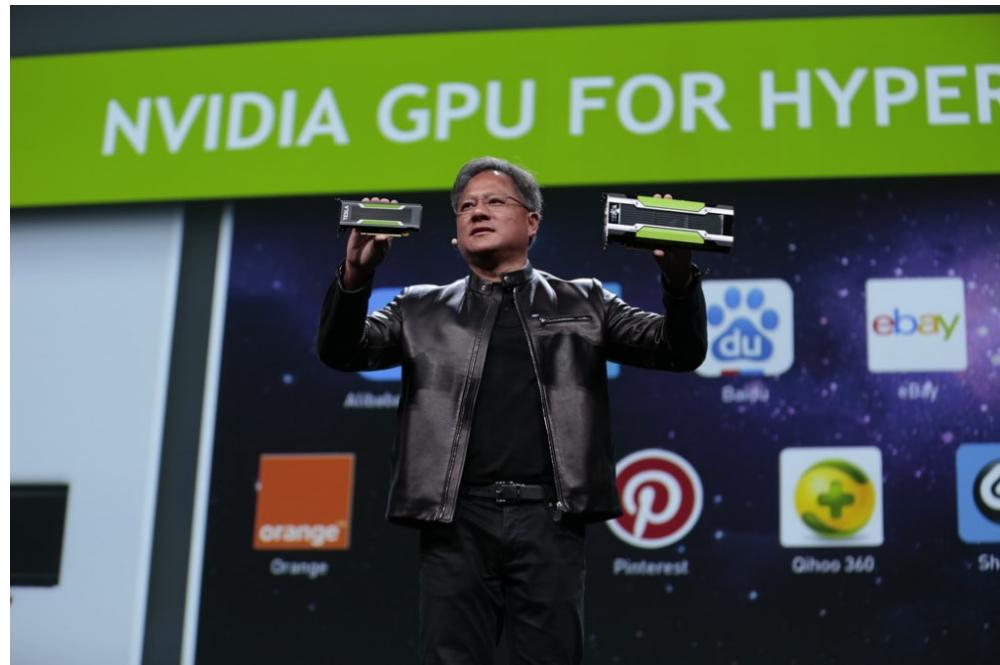
# AMD

NVIDIA está ganhando disparado a briga pelo domínio de *deep learning*

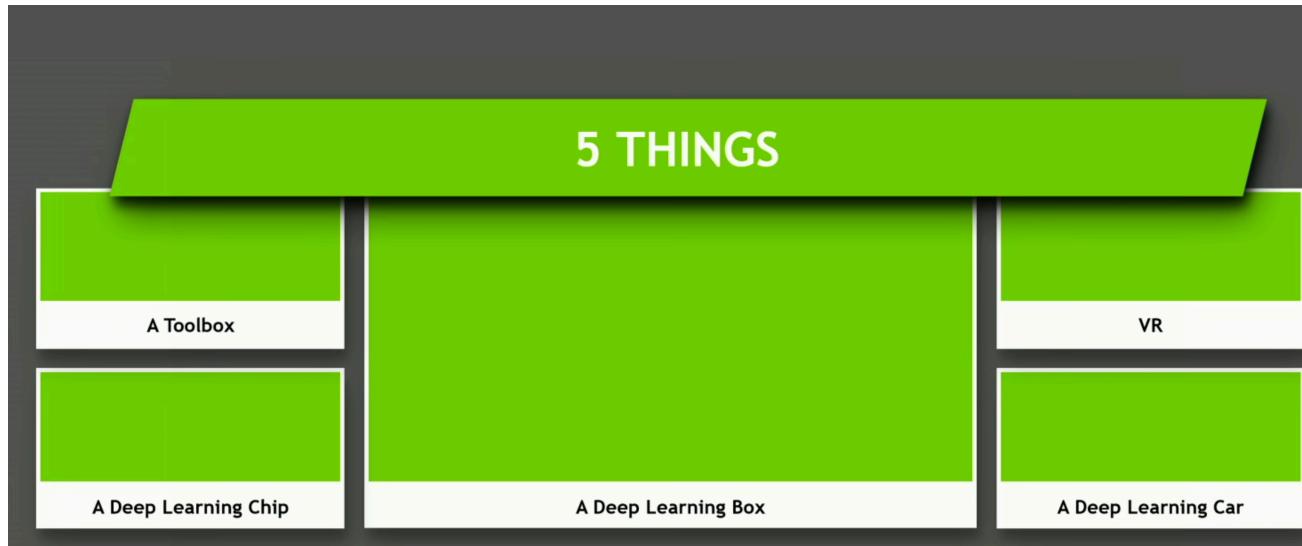


# Detalhes de Implementação

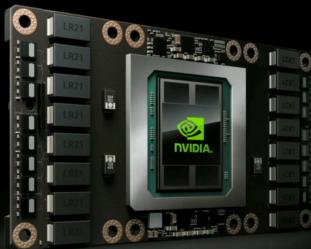
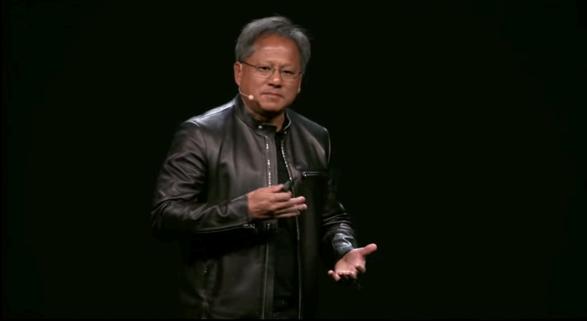
[https://www.youtube.com/watch?v=ddBIF1fnvIM&index=5&list=PLZHnYvH1qtOZPJtv1WNYk0TU4L3M\\_rnj4](https://www.youtube.com/watch?v=ddBIF1fnvIM&index=5&list=PLZHnYvH1qtOZPJtv1WNYk0TU4L3M_rnj4)



CEO da NVIDIA (Jen-Hsun Huang) na GTC 2016



# Detalhes de Implementação



**TESLA P100**  
THE MOST ADVANCED  
HYPERSCALE DATACENTER GPU EVER BUILT

150B XTORS | 5.3TF FP64 | 10.6TF FP32 | 21.2TF FP16 | 14MB SM RF | 4MB L2 Cache



**NVIDIA DGX-1**  
WORLD'S FIRST DEEP LEARNING SUPERCOMPUTER

Engineered for deep learning | 170TF FP16 | 8x Tesla P100 | NVLink hybrid cube mesh | Accelerates major AI frameworks

# Detalhes de Implementação

	DUAL XEON	DGX-1
FLOPS (CPU + GPU)	3 TF	170 TF
AGGREGATE NODE BW	76 GB/s	768 GB/s
ALEXNET TRAIN TIME	150 HOURS	2 HOURS
TRAIN IN 2 HOURS	>250 NODES*	1 NODE

“250 SERVERS  
IN-A-BOX”

\* Caffe Training on Multi-node Distributed-memory Systems Based on Intel® Xeon® Processor E5 Family (extrapolated)  
Geirnady Fedorov (Intel)’s picture submitted by Geirnady Fedorov (Intel), Valdim P. (Intel) on October 29, 2015  
<http://software.intel.com/en-us/articles/caffe-training-on-multi-node-distributed-memory-systems-based-on-intel-xeon-processor-e5>

# Detalhes de Implementação

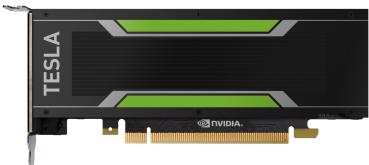
## DGX-1 ARCHITECTURE ADVANCEMENTS

System Level Specifications	NVIDIA DGX-1 with Tesla P100	NVIDIA DGX-1 with Tesla V100
TFLOPS (FP16)	170	960
CUDA Cores	28,672	40,960
Tensor Cores	--	5,120
NVLink vs PCIe Speed-up	5X	10X
Deep Learning Training Speed-up	1X	3X

# Detalhes de Implementação



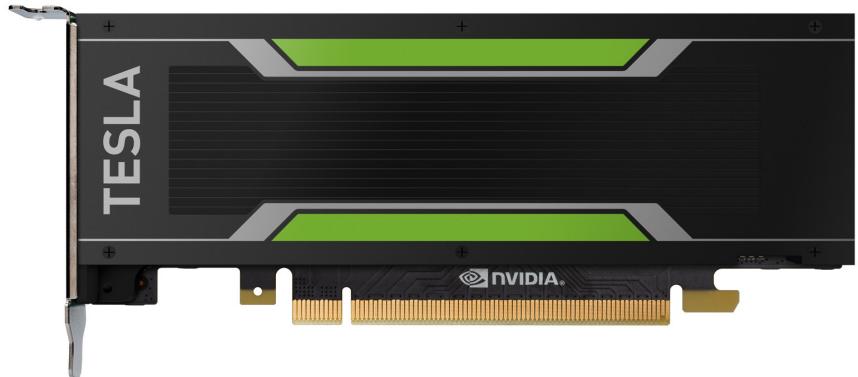
poucos cores (1-16) muito rápidos, excelente para processamento sequencial



muitos cores (milhares) mais lentos, originalmente destinados apenas a processamento gráfico... excelente para processamento em paralelo

	# Cores	Clock Speed	Memory	Price
<b>CPU</b> (Intel Core i7-7700k)	4 (8 threads with hyperthreading )	4.4 GHz	Shared with system	\$339
<b>CPU</b> (Intel Core i7-6950X)	10 (20 threads with hyperthreading )	3.5 GHz	Shared with system	\$1723
<b>GPU</b> (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	\$1200
<b>GPU</b> (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	\$399

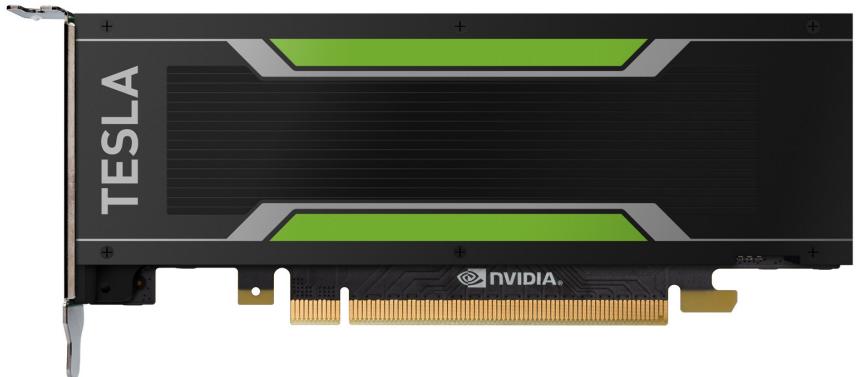
# Detalhes de Implementação



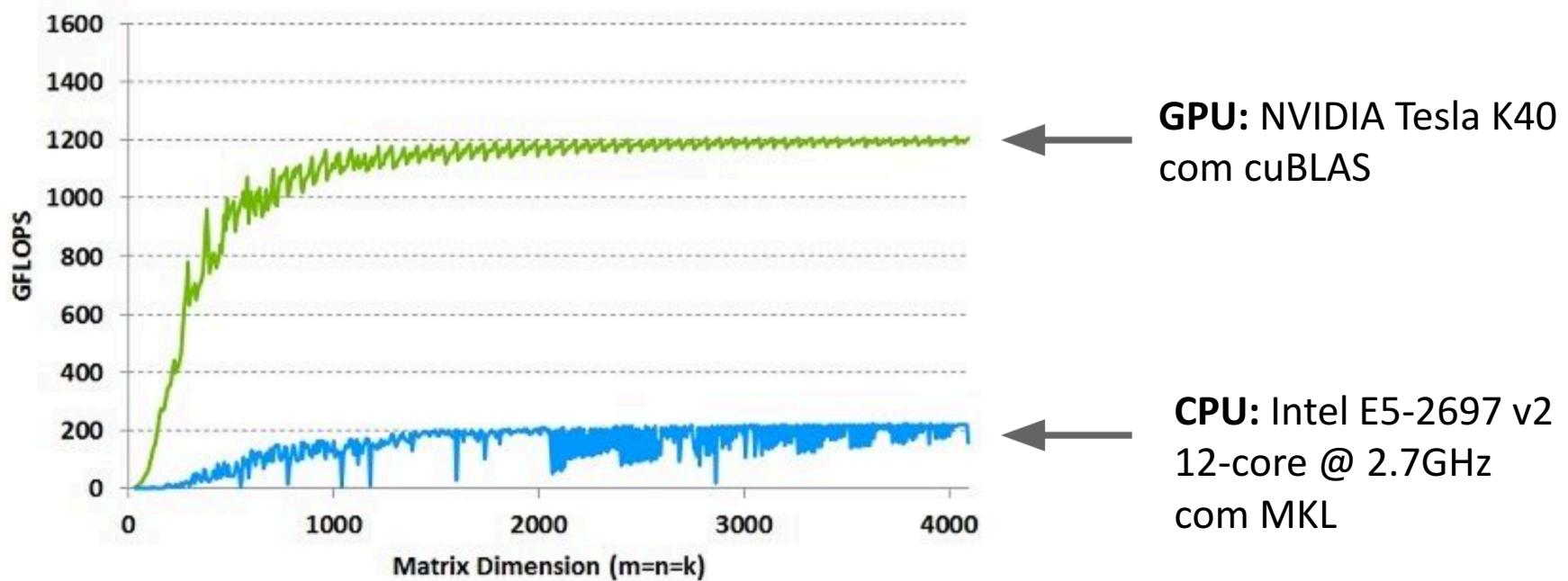
## Programação em GPUs:

- CUDA (proprietário NVIDIA)
  - Gera código C que roda direto na GPU
  - APIs de alto nível: cuBLAS, cuFFT, cuDNN, etc.
- OpenCL
  - Similar a CUDA, mas roda em qualquer GPU
  - Frequentemente mais lento =(
- Udacity ("Introduction to Parallel Programming")
  - <https://www.udacity.com/course/cs344>
- Para *deep learning*, utilize as bibliotecas/APIs prontas....

# Detalhes de Implementação

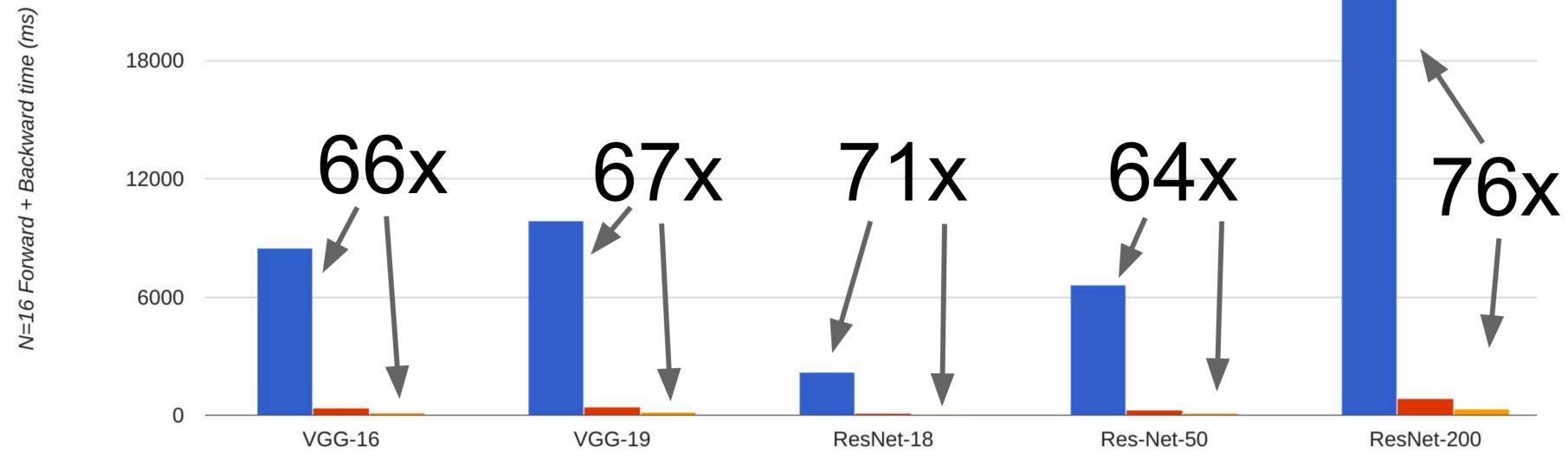
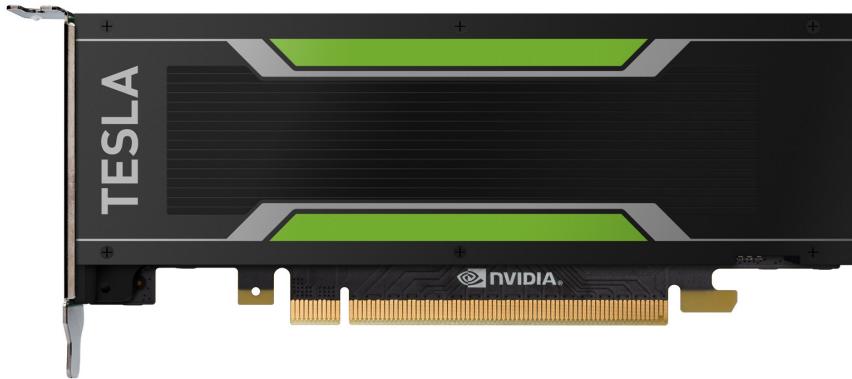


Multiplicação de matrizes com GPUs:

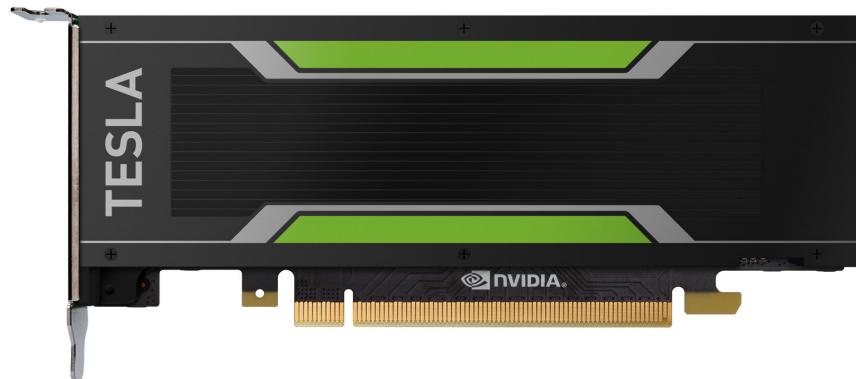


# Detalhes de Implementação

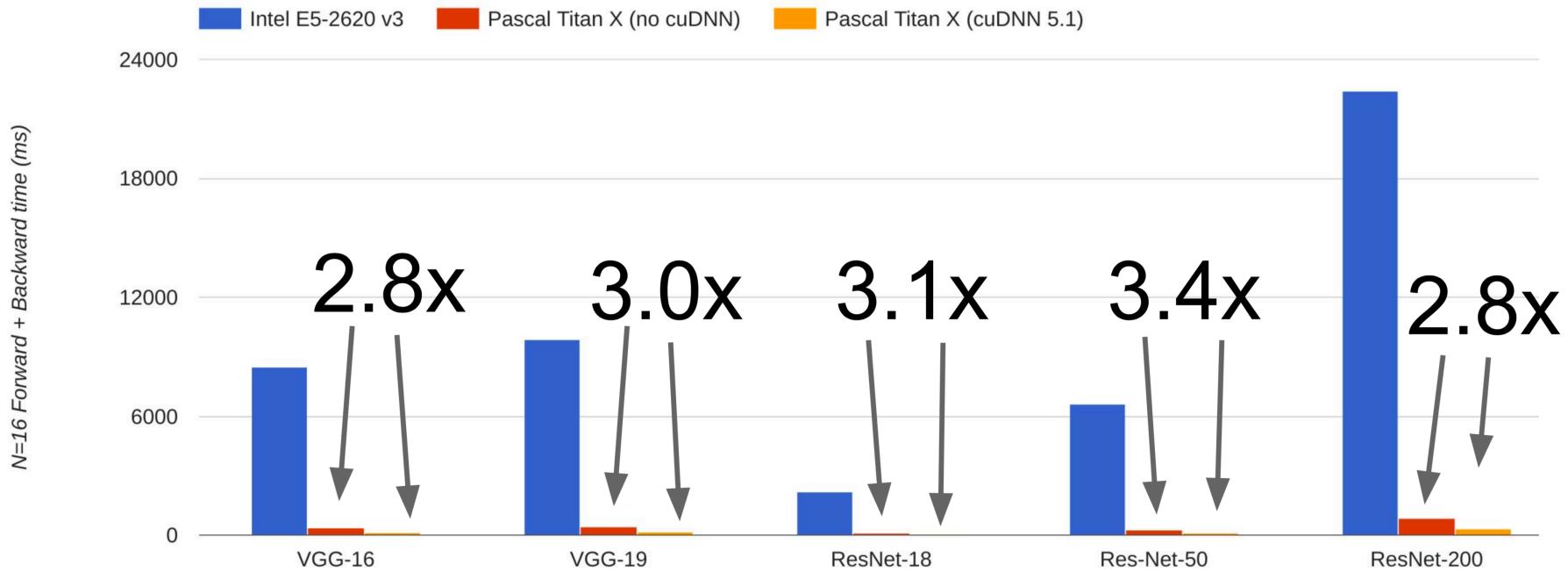
GPUs são excelentes para convoluções:



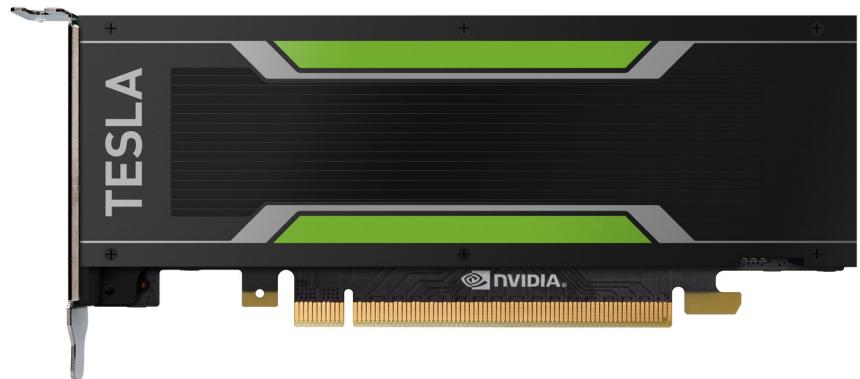
# Detalhes de Implementação



cuDNN muito melhor do que CUDA não-otimizado



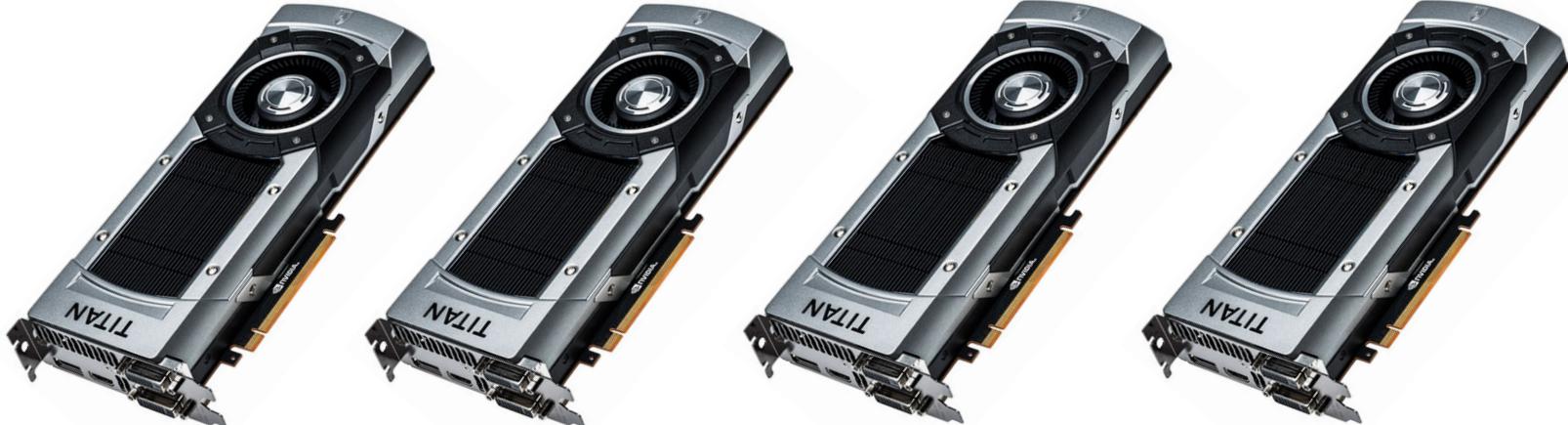
# Detalhes de Implementação



Mesmo com GPUs, treinamento de CNNs pode ser lento!!!

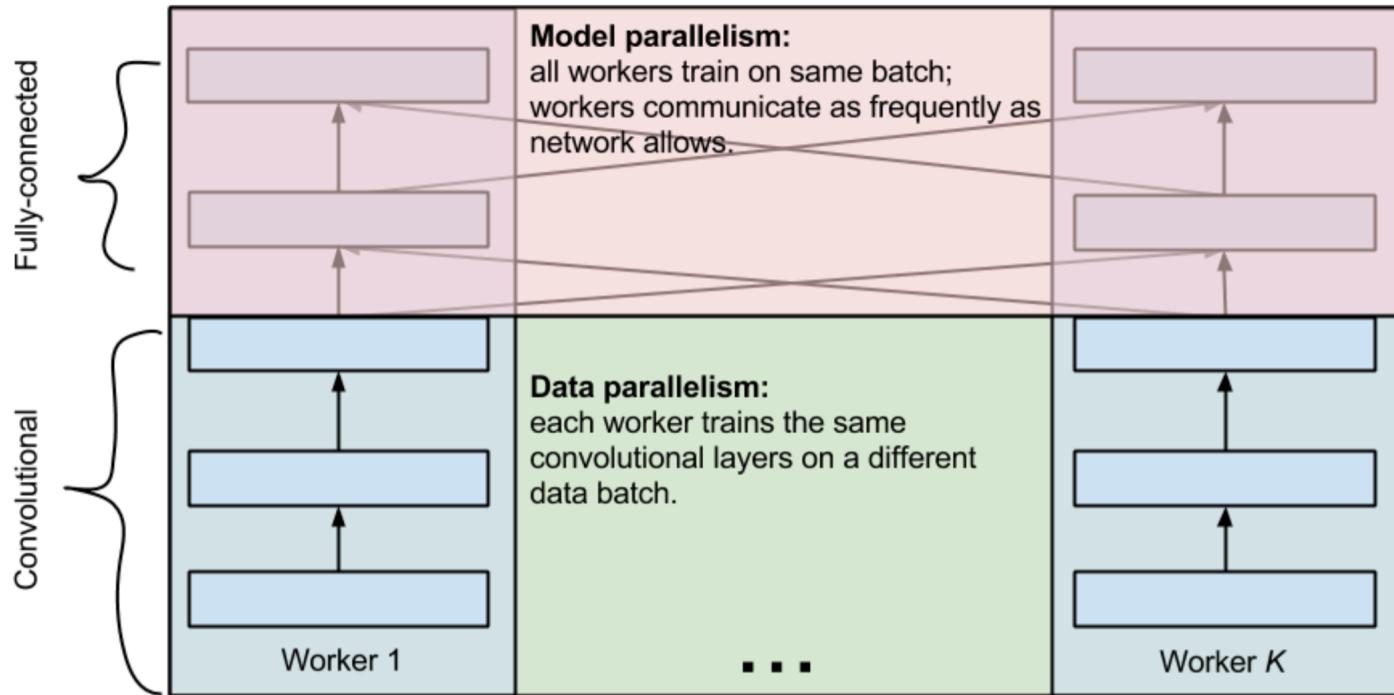
ResNet-101: 2-3 semanas com 4 GPUs!!

Batches de imagens são divididos entre as GPUs!



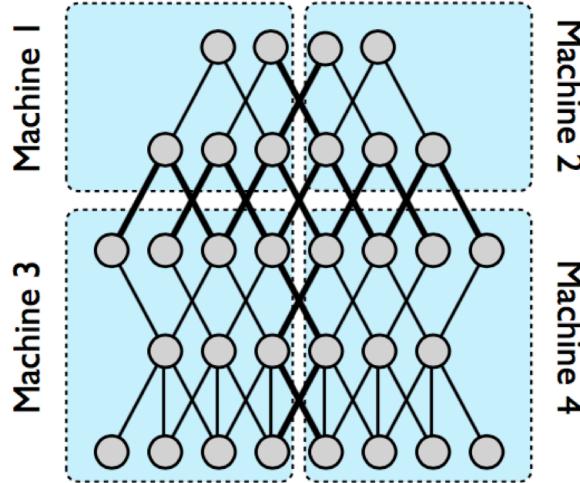
# Detalhes de Implementação

Alex Krizhevsky: [One weird trick for parallelizing convolutional neural networks](#). CoRR abs/1404.5997 (2014)



# Detalhes de Implementação

Google: Treinamento Distribuído em CPU



Paralelismo do Modelo

Figure 1: An example of model parallelism in DistBelief. A five layer deep neural network with local connectivity is shown here, partitioned across four machines (blue rectangles). Only those nodes with edges that cross partition boundaries (thick lines) will need to have their state transmitted between machines. Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once. Within each partition, computation for individual nodes will be parallelized across all available CPU cores.

# Detalhes de Implementação

Google: Atualização Síncrona vs Assíncrona

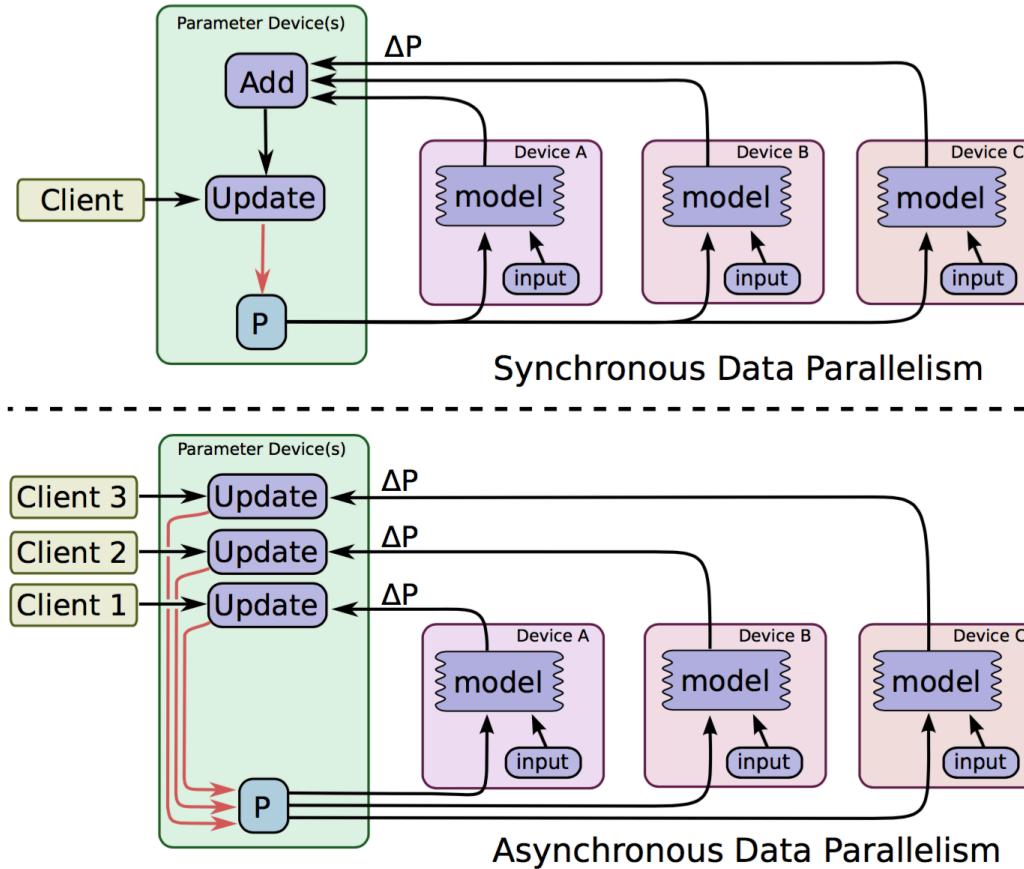


Figure 7: Synchronous and asynchronous data parallel training

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: **TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems**. CoRR abs/1603.04467 (2016)

# Detalhes de Implementação

## Gargalos de *Hardware*



### 1. Comunicação CPU ↔ GPU

**CPU** → thread carregando dados e fazendo data augmentation

**GPU** → passagens *forward* e *backward* pela rede

# Detalhes de Implementação

## Gargalos de *Hardware*



### 2. Gargalo CPU↔ disco

**Disco rígido (HD)** é muito lento para ser utilizado em deep learning



**Utilizar SSD** → imagens pré-processadas são idealmente armazenadas contiguamente em arquivos e lidas como stream de bytes do SSD



# Detalhes de Implementação

## Gargalos de *Hardware*



### 3. Memória da GPU

Tesla M40 e P40: **24GB**

Tesla K40 (K80), Titan X: **12GB**

GTX 1080 Ti: **11GB**

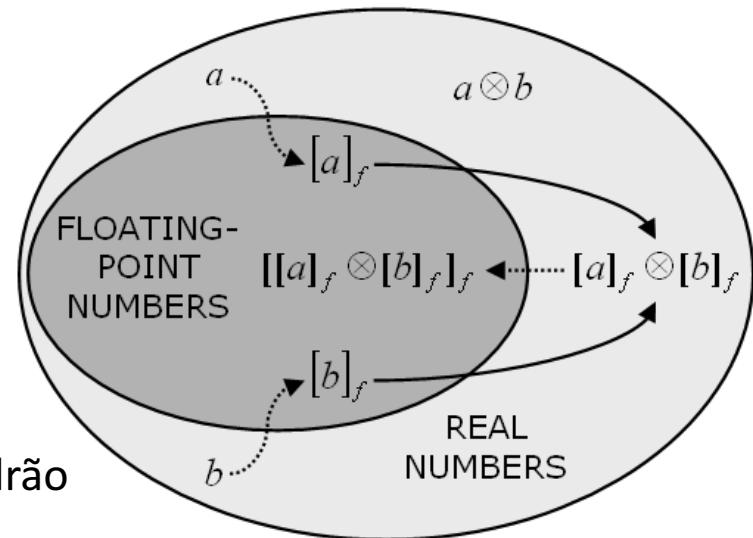
GTX 980 Ti: **6GB**

Ex: AlexNet precisa de  $\approx 3\text{GB}$  para batch de 256 imagens

# Detalhes de Implementação

## Precisão de Ponto Flutuante

- Ponto flutuante de precisão dupla (64 bits) é o padrão para programação nos dias de hoje
- Para CNNs, no entanto, precisão simples (32 bits) é tipicamente utilizado nos frameworks por questões de desempenho



# Detalhes de Implementação

## Precisão de Ponto Flutuante

AlexNet (One Weird Trick paper) - Input 128x3x224x224

Library	Class	Time (ms)	forward (ms)	backward (ms)
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	71	25	46
Nervana-neon-fp16	ConvLayer	78	25	52
CuDNN[R4]-fp32 (Torch)	cudnn.SpatialConvolution	81	27	53
TensorFlow	conv2d	81	26	55
Nervana-neon-fp32	ConvLayer	87	28	58
fbfft (Torch)	fbnn.SpatialConvolution	104	31	72

<https://github.com/soumith/convnet-benchmarks>

- Muito provavelmente o novo padrão para CNNs será meia-precisão (16 bits)
  - cuDNN já suporta cálculos com fp-16
  - cuDNN e Nervana (adquirida recentemente pela Intel) têm as soluções mais rápidas atualmente
  - Placas da NVIDIA (Pascal) dão suporte a fp-16

GoogleNet V1 - Input 128x3x224x224

Library	Class	Time (ms)	forward (ms)	backward (ms)
Nervana-neon-fp16	ConvLayer	230	72	157
Nervana-neon-fp32	ConvLayer	270	84	186
TensorFlow	conv2d	445	135	310
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	462	112	349
CuDNN[R4]-fp32 (Torch)	cudnn.SpatialConvolution	470	130	340
Chainer	Convolution2D	687	189	497
Caffe	ConvolutionLayer	1935	786	1148
CL-nn (Torch)	SpatialConvolutionMM	7016	3027	3988
Caffe-CLGreenTea	ConvolutionLayer	9462	746	8716

# Detalhes de Implementação

## Precisão de Ponto Flutuante

Quanto podemos comprometer de precisão para ter desempenho computacional maior?

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, Pritish Narayanan: **Deep Learning with Limited Numerical Precision**. ICML 2015:  
1737–1746

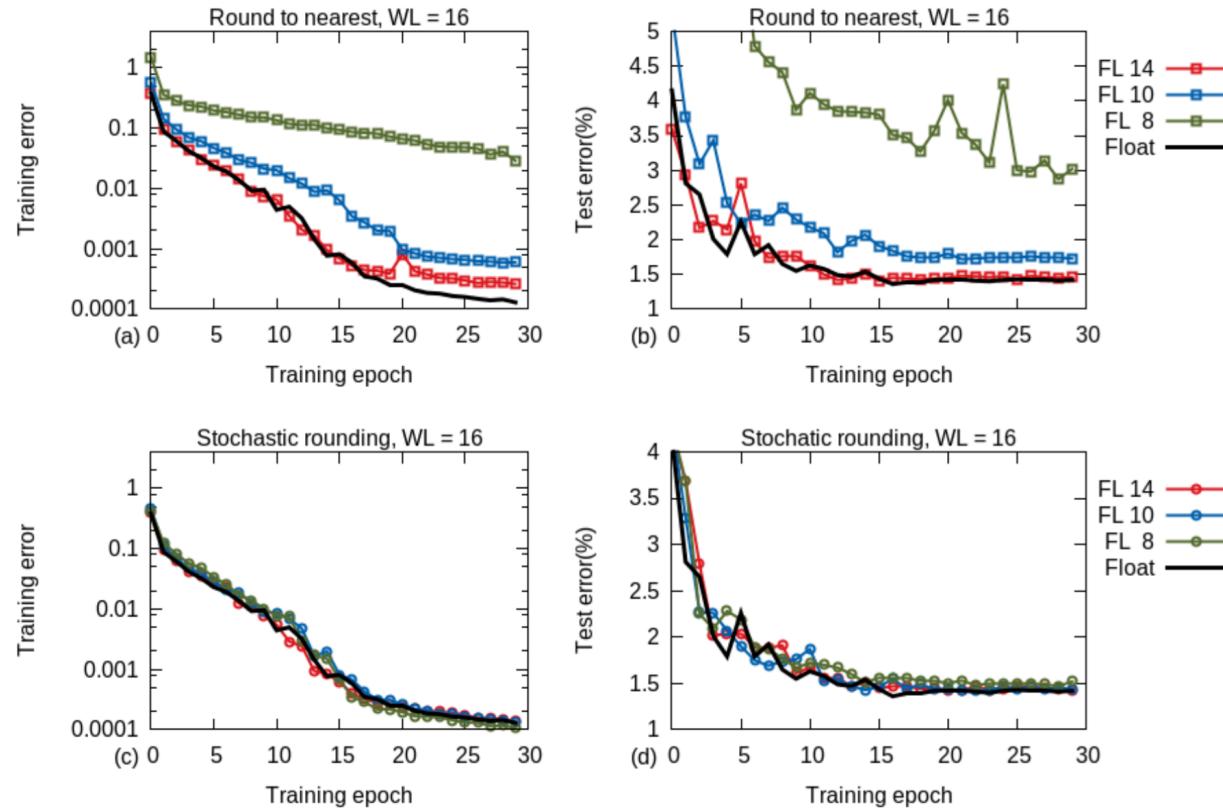


Figure 1. MNIST dataset using fully connected DNNs: Training error (a, c) and the test error (b, d) for training using fixed-point number representation and rounding mode set to either “Round to nearest” (top) or “Stochastic rounding” (bottom). The word length for fixed-point numbers WL is kept fixed at 16 bits and results are shown for three different fractional (integer) lengths: 8(8), 10(6), and 14(2) bits. Results using float are also shown for comparison.

# Detalhes de Implementação

## Precisão de Ponto Flutuante

Quanto podemos comprometer de precisão para ter desempenho computacional maior?

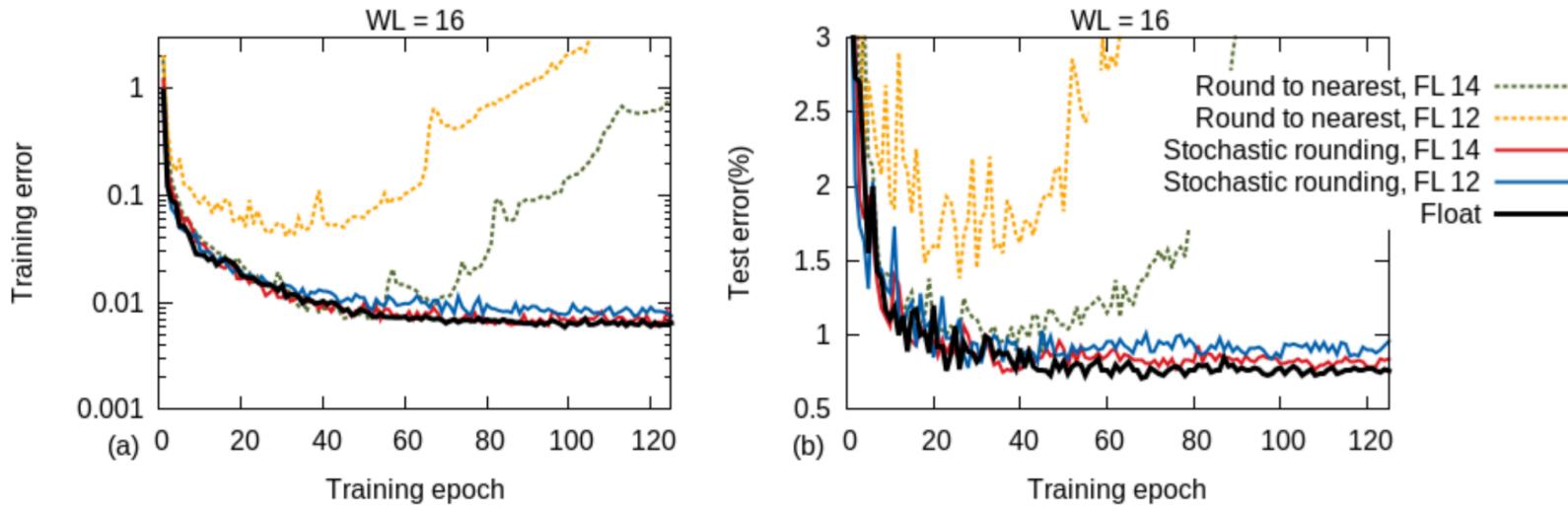


Figure 2. MNIST dataset using CNNs: Training error (a) and the test error (b) for training using fixed-point number representation and rounding mode set to either “Round to nearest” or “Stochastic rounding”. The word length for fixed-point numbers WL is kept fixed at 16 bits and results are shown for different fractional (integer) lengths for weights and weight updates: 12(4), and 14(2) bits. Layer outputs use  $\langle 6, 10 \rangle$  format in all cases. Results using `float` are also shown for comparison.

# Detalhes de Implementação

## Precisão de Ponto Flutuante

Quanto podemos comprometer de precisão para ter desempenho computacional maior?

Matthieu Courbariaux, Yoshua Bengio, Jean-Pierre David: **Training Deep Neural Networks with Low Precision Multiplications.** ICLR 2015.

- 10 bits para ativações (propagação *forward*)
- 12 bits para atualização de parâmetros

### 11 CONCLUSION AND FUTURE WORKS

We have shown that:

- Very low precision multipliers are sufficient for training deep neural networks.
- Dynamic fixed point seems well suited for training deep neural networks.
- Using a higher precision for the parameters during the updates helps.

Our work can be exploited to:

- Optimize memory usage on general-purpose hardware (Gray *et al.*, 2015).
- Design very power-efficient hardware dedicated to deep learning.

Format	Prop.	Up.	PI MNIST	MNIST	CIFAR-10	SVHN
Goodfellow <i>et al.</i> (2013a)	32	32	0.94%	0.45%	11.68%	2.47%
Single precision floating point	32	32	1.05%	0.51%	14.05%	2.71%
Half precision floating point	16	16	1.10%	0.51%	14.14%	3.02%
Fixed point	20	20	1.39%	0.57%	15.98%	2.97%
Dynamic fixed point	10	12	1.28%	0.59%	14.82%	4.95%

Table 4: Test set error rates of single and half floating point formats, fixed and dynamic fixed point formats on the permutation invariant (PI) MNIST, MNIST (with convolutions, no distortions), CIFAR-10 and SVHN datasets. **Prop.** is the bit-width of the propagations and **Up.** is the bit-width of the parameters updates. The single precision floating point line refers to the results of our experiments. It serves as a baseline to evaluate the degradation brought by lower precision.

# Detalhes de Implementação

## Precisão de Ponto Flutuante

Quanto podemos comprometer de precisão para ter desempenho computacional maior?

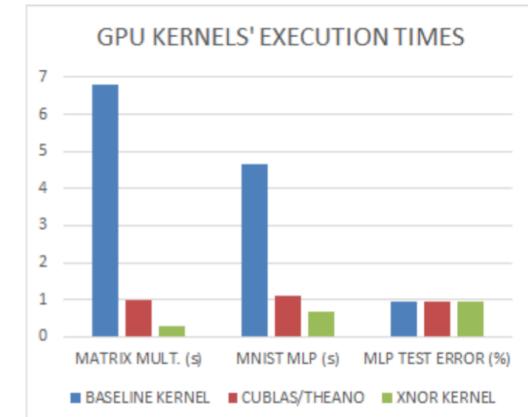
Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, Yoshua Bengio: **Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1**. CoRR abs/1602.02830, 2016.

- Treinamento com pesos e ativações de 1 bit!!!!!
- Ativações e pesos são binarizados para +1 ou -1 (função sinal)
- Multiplicações rápidas com XNOR bit-a-bit
- (Gradientes utilizam maior precisão, obviamente)

Table 1. Classification test error rates of DNNs trained on MNIST (MLP architecture without unsupervised pretraining), CIFAR-10 (without data augmentation) and SVHN.

Data set	MNIST	SVHN	CIFAR-10
Binarized activations+weights, during training and test			
BNN (Torch7)	1.40%	2.53%	10.15%
BNN (Theano)	0.96%	2.80%	11.40%
Committee Machines' Array [Baldassi et al., 2015]	1.35%	-	-
Binarized weights, during training and test			
BinaryConnect [Courbariaux et al., 2015]	$1.29 \pm 0.08\%$	2.30%	9.90%
Binarized activations+weights, during test			
EBP [Cheng et al., 2015]	$2.2 \pm 0.1\%$	-	-
Bitwise DNNs [Kim & Smaragdis, 2016]	1.33%	-	-
Ternary weights, binary activations, during test			
(Hwang & Sung, 2014)	1.45%	-	-
No binarization (standard results)			
Maxout Networks [Goodfellow et al.]	0.94%	2.47%	11.68%
Network in Network [Lin et al.]	-	2.35%	10.41%
Gated pooling [Lee et al., 2015]	-	1.69%	7.62%

Figure 3. The first three columns represent the time it takes to perform a  $8192 \times 8192 \times 8192$  (binary) matrix multiplication on a GTX750 Nvidia GPU, depending on which kernel is used. We can see that our XNOR kernel is 23 times faster than our baseline kernel and 3.4 times faster than cuBLAS. The next three columns represent the time it takes to run the MLP from Section 2 on the full MNIST test set. As MNIST's images are not binary, the first layer's computations are always performed by the baseline kernel. The last three columns show that the MLP accuracy does not depend on which kernel is used.



# Detalhes de Implementação

## Conclusão:

1. GPUs são muito mais rápidas que CPUs!
2. Cuidado com os gargalos CPU-GPU e CPU-disco
3. Precisão mais baixa acelera o processo e parece funcionar bem!
  - a) Padrão atual é 32 bits, 16 bits em breve
  - b) Pro futuro.... redes binárias?

Material adaptado do original gentilmente cedido por Andrej Karpathy, Justin Johnson, Serena Yeung e Fei Fei Li