Lecture 05b

Introducción a las CNNs parte 2 - Arquitecturas de CNNs

Controlar el tamaño de salida además del paso

- **Padding (controla el tamaño de la salida junto con stride)**
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - VGG16 (simple, CNN profunda)
  - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

Relleno

output size

input size

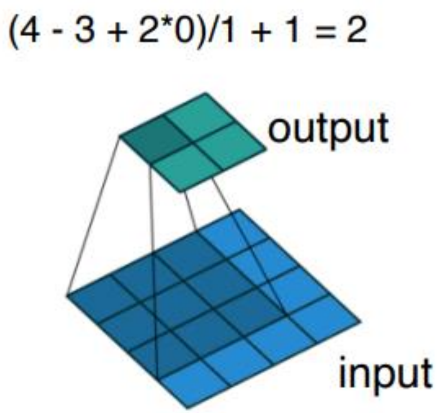
padding pixels per side

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1$$

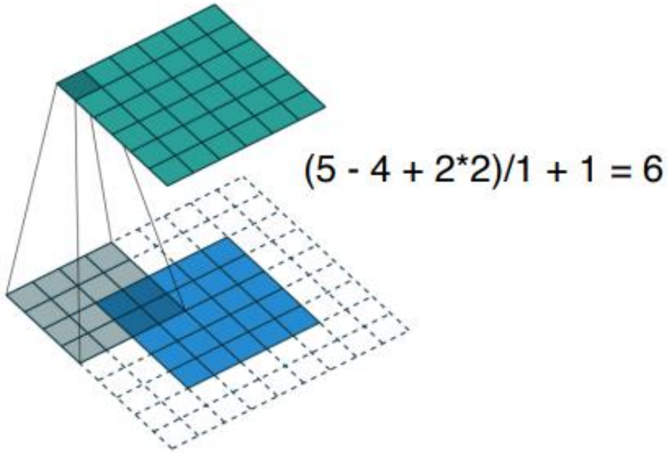
kernel size

"floor" function

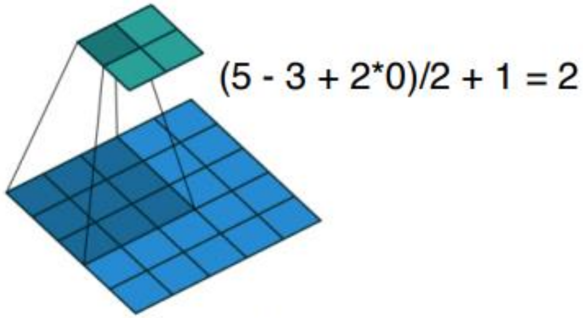
stride



No padding, stride=1



padding=2, stride=1



No padding, stride=2

Altamente recomendado

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

https://arxiv.org/abs/1603.07285

Jerga de Padding

Convolución "valid": Sin padding (el mapa de características se reduce)

Convolución "same": Padding tal que el tamaño de salida es igual al tamaño de entrada

Convenciones comunes de tamaño de kernel:

3x3, 5x5, 7x7 (a veces 1x1 en capas posteriores para reducir canales)

Padding

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1$$

Suponga que desea utilizar una operación convolucional con paso 1 y mantener las dimensiones de entrada en el mapa de características de salida:

¿Cuánto padding se necesita para que el tamaño del mapa de características sea igual al tamaño de entrada?

$$o = i + 2p - k + 1$$

$$\Leftrightarrow p = (o - i + k - 1)/2$$

$$\Leftrightarrow p = (k - 1)/2$$

Padding

$$o = i + 2p - k + 1$$

$$\Leftrightarrow p = (o - i + k - 1)/2$$

$$\Leftrightarrow p = (k - 1)/2$$

Probablemente explica por qué las convenciones de tamaño de kernel comunes son 3x3, 5x5, 7x7 (a veces 1x1 en capas posteriores para reducir canales)

Conceptos familiares ahora en 2D

- Padding (controla el tamaño de la salida junto con stride)
- **Dropout 2D y batchnorm**
- Arquitecturas comunes
    - VGG16 (simple, CNN profunda)
    - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

Dropout 2D

- Problema con el dropout regular y las CNN: Es probable que los píxeles adyacentes estén muy correlacionados (por lo tanto, es posible que no ayuden a reducir la "dependencia" tanto como se pretendía originalmente con el dropout)

- Por lo tanto, puede ser mejor eliminar mapas de características completos.

La idea proviene de
Tompson, Jonathan, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler.
"Efficient object localization using convolutional networks." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 648-656. 2015.

https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Tompson_Efficient_Object_Localization_2015_CVPR_paper.html

Dropout 2D

- Dropout2d descartará mapas de características completos (canales)

```python
import torch

m = torch.nn.Dropout2d(p=0.5)
input = torch.randn(1, 3, 5, 5)
output = m(input)
```

```python
output
```

```
tensor([[[[-0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000, -0.0000,  0.0000, -0.0000,  0.0000],
          [ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.0000, -0.0000, -0.0000]],

         [[-3.5274,  0.8163,  0.2440,  1.2410,  1.5022],
          [-1.2455,  6.3875, -2.6224,  0.0261,  1.7487],
          [ 1.6471,  0.7444, -2.1941, -2.0119, -1.5232],
          [ 0.3720, -1.5606,  0.7630,  0.9177, -0.1387],
          [-1.2817, -3.5804,  0.4367, -0.1384, -0.8148]],

         [[-0.0000, -0.0000, -0.0000, -0.0000,  0.0000],
          [ 0.0000, -0.0000, -0.0000, -0.0000,  0.0000],
          [ 0.0000, -0.0000,  0.0000, -0.0000, -0.0000],
          [-0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.0000,  0.0000,  0.0000]]]])
```

BatchNorm 2D

BatchNorm1d

CLASS `torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

BatchNorm2d

CLASS `torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Fuente: https://pytorch.org/docs/stable/nn.html

BatchNorm 2D

BatchNorm1d **Las entradas son tensores de rango 2: [N, num_features)**

CLASS `torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

BatchNorm2d **Las entradas son tensores de rango 4: [N, C, H, W]**

CLASS `torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

En BatchNorm2d, la desviación media y estándar se calculan para N * H * W, es decir, sobre la dimensión del canal

Fuente: https://pytorch.org/docs/stable/nn.html

BatchNorm 2D

En BatchNorm2d, la desviación media y estándar se calculan para N * H * W, es decir, sobre la dimensión del canal

```
[1]: import torch.nn as nn
     import torch.nn.functional as F


     class Model(nn.Module):
         def __init__(self):
             super(Model, self).__init__()

             self.cn1 = nn.Conv2d(3, 192, kernel_size=5,
                                  stride=1, padding=2, bias=False)
             self.bn1 = nn.BatchNorm2d(192)

     # ...
```

```
[2]: model = Model()
```

```
[3]: model.bn1.weight.size()
```

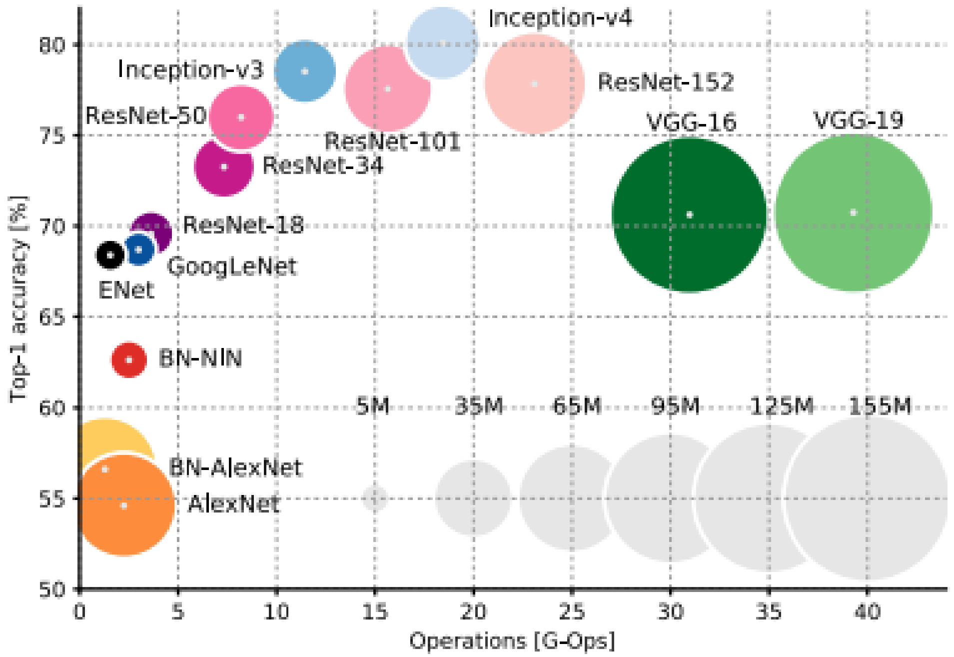
```
[3]: torch.Size([192])
```

Revisión de las arquitecturas comunes

- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- **Arquitecturas comunes**
  - VGG16 (simple, CNN profunda)
  - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

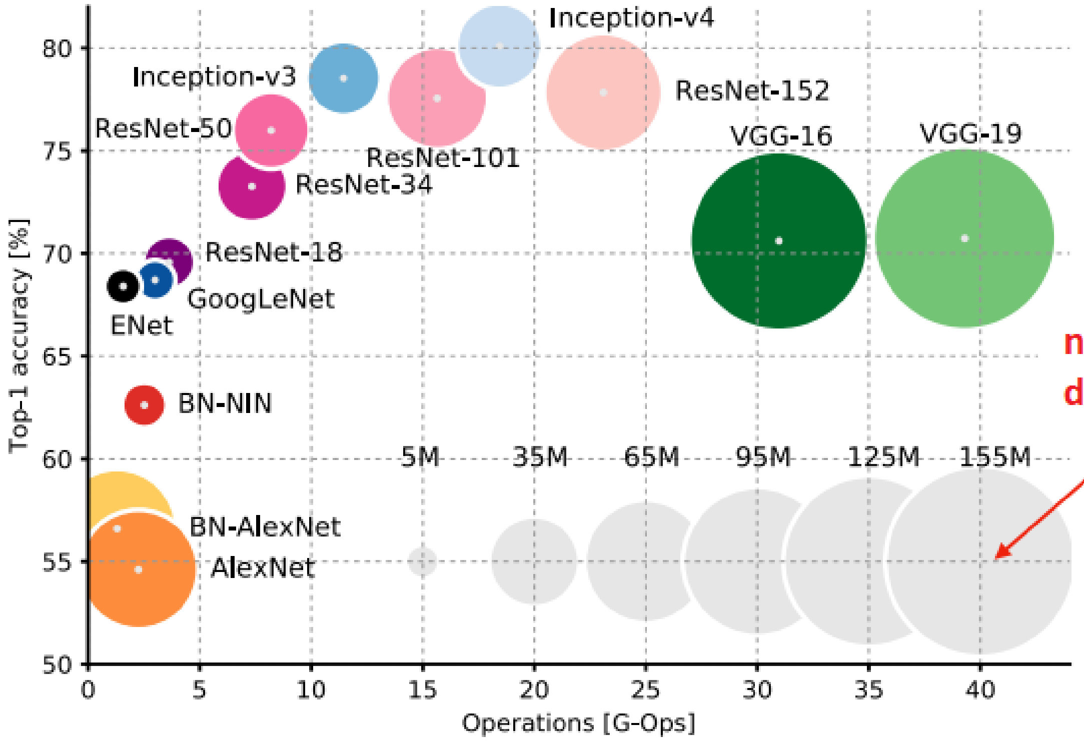
Revisión de las arquitecturas comunes

Discutiremos algunas arquitecturas CNN comunes adicionales ya que el campo evolucionó bastante desde 2012.



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical Figure 1: Top1 vs. network. applications. Single-crop top-1 vali- arXiv preprint arXiv:1605.07678.

Revisión de las arquitecturas comunes



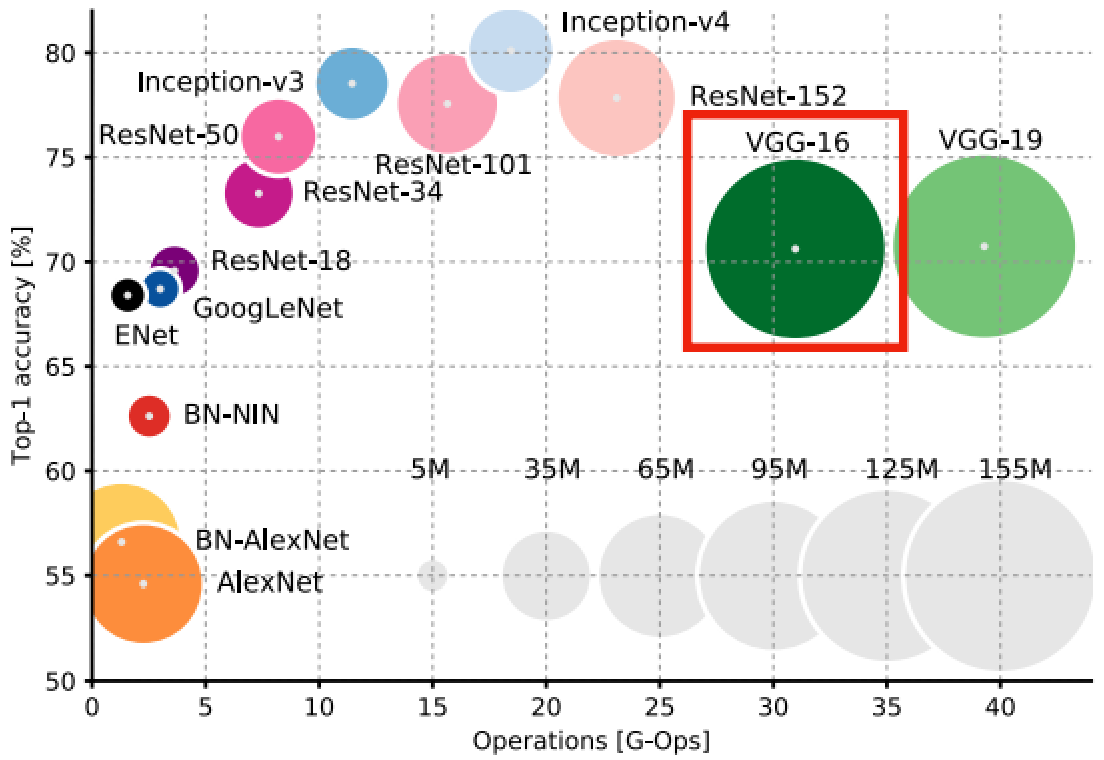
número de parámetros
del modelo

Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical
Figure 1: Top1 vs. network. applications. Single-crop top-1 vali- arXiv preprint arXiv:1605.07678.

Agregando más capas

- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - **VGG16 (simple, CNN profunda)**
  - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

Revisión de las arquitecturas comunes



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical
Figure 1: Top1 vs. network. applications. Single-crop top-1 vali- arXiv preprint arXiv:1605.07678.

VGG-16

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

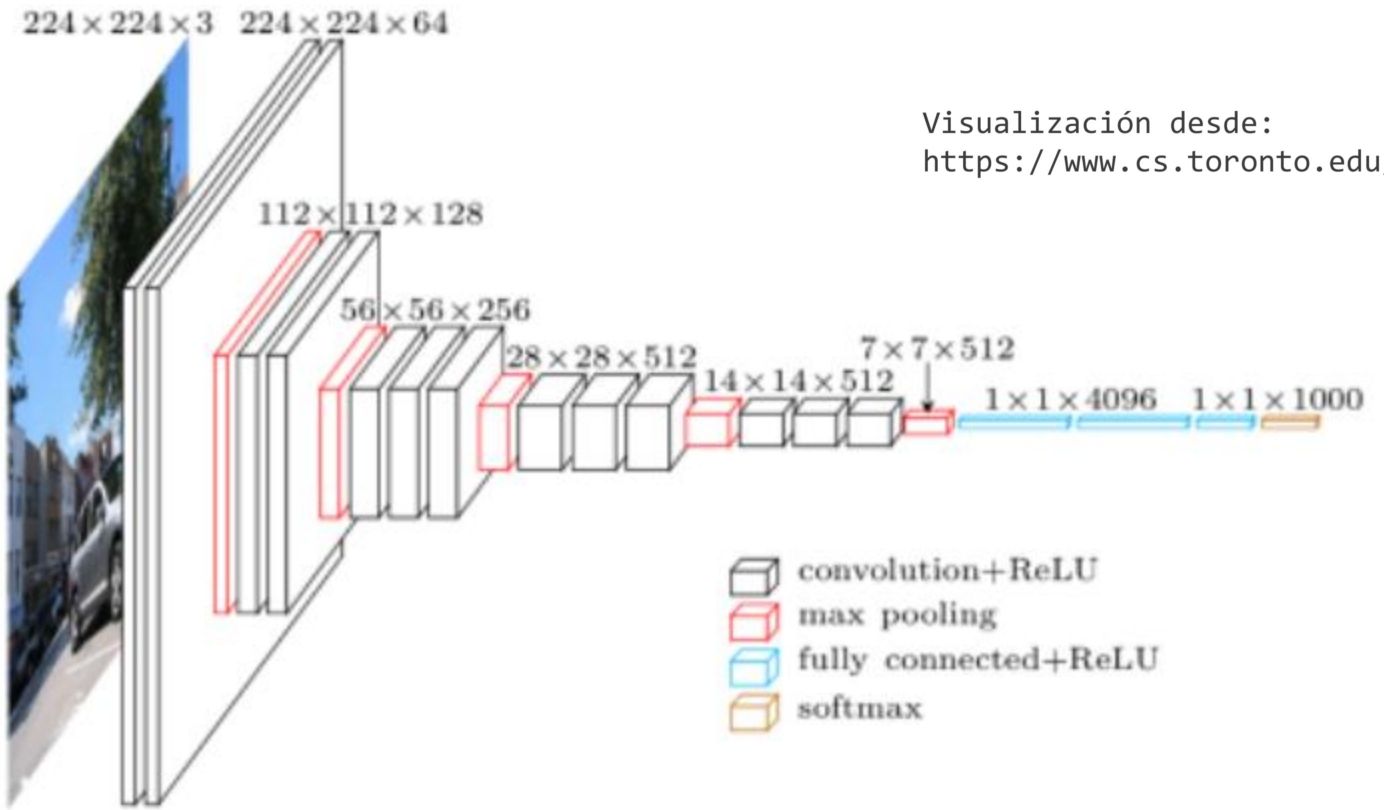
Ventajas:
Arquitectura muy simple, conv 3x3, stride = 1, padding "same" , MaxPooling de 2x2

Desventaja:
Gran cantidad de parámetros
y lento (ver diapositiva anterior)

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

https://arxiv.org/abs/1409.1556

VGG-16



Visualización desde:
https://www.cs.toronto.edu/~frossard/post/vgg16/

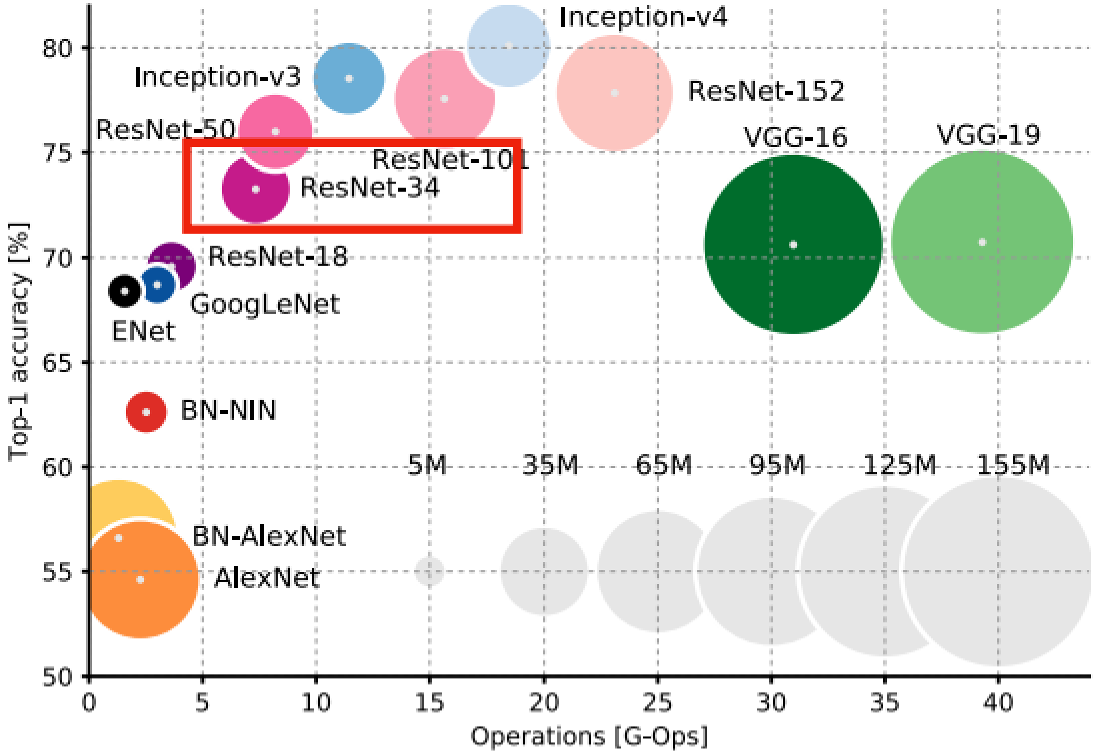
Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

https://arxiv.org/abs/1409.1556

¿Podemos agregar más capas? CNN con un simple truco

- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - VGG16 (simple, CNN profunda)
  - **ResNet y skip connections**
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

Revisión de las arquitecturas comunes



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical Figure 1: Top1 vs. network. applications. Single-crop top-1 vali- arXiv preprint arXiv:1605.07678.

ResNets

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

Con su simple truco de permitir omitir conexiones (la posibilidad de aprender funciones de identidad y omitir capas que no son útiles), ResNets permite implementar arquitecturas muy, muy profundas

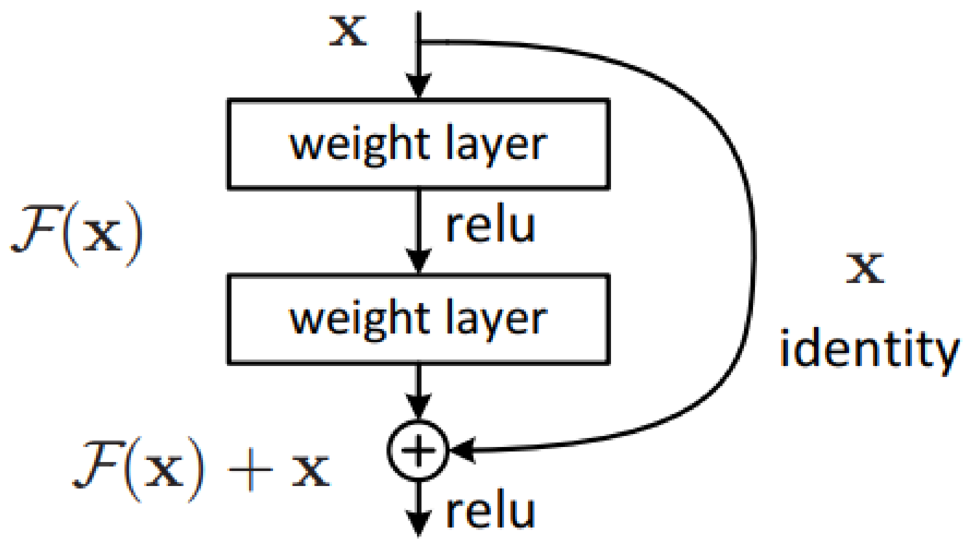
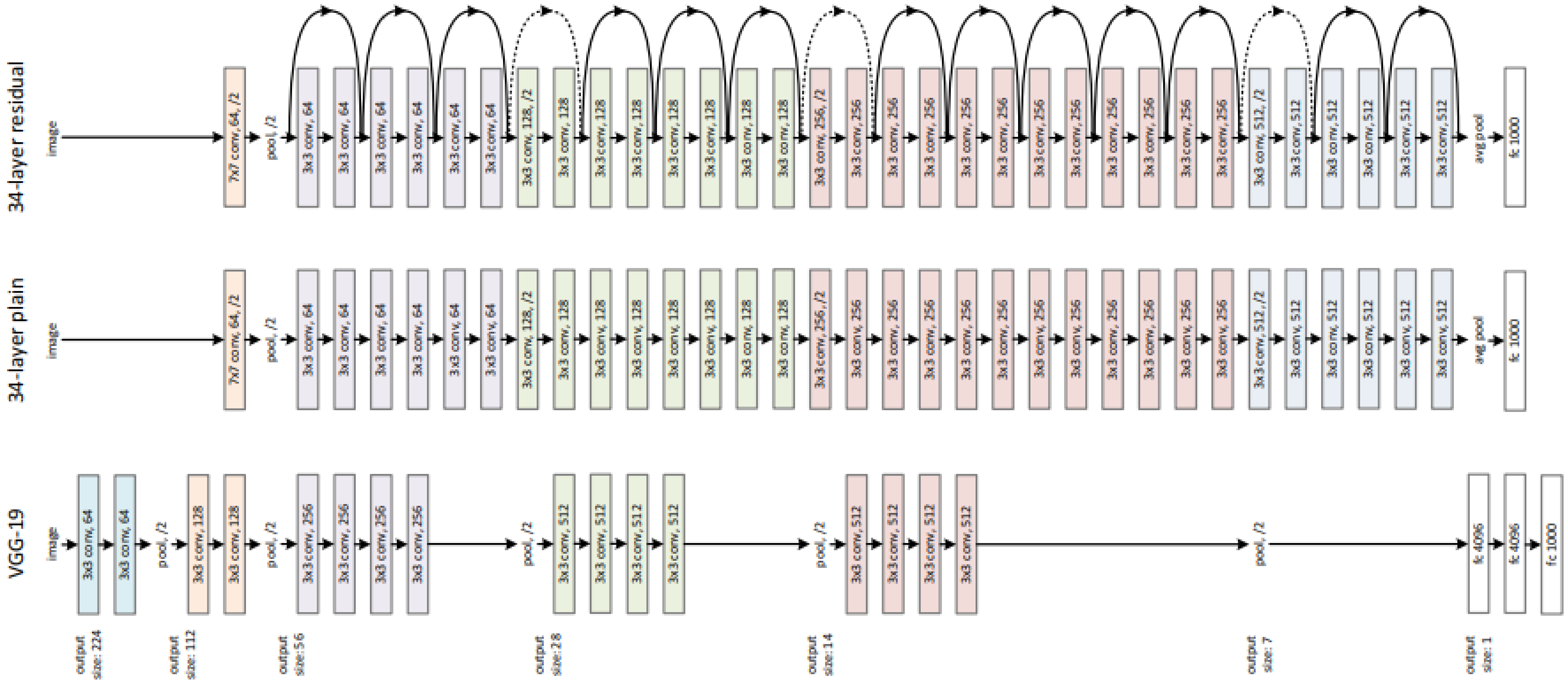


Figure 2. Residual learning: a building block.

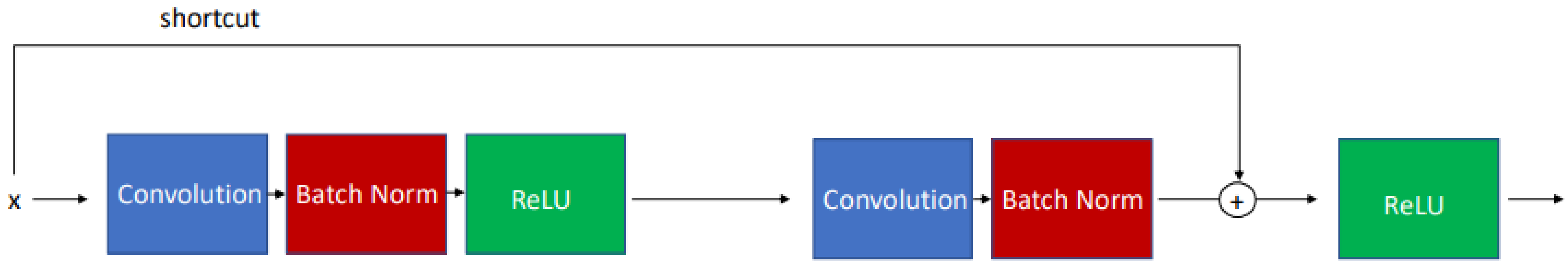
ResNets

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

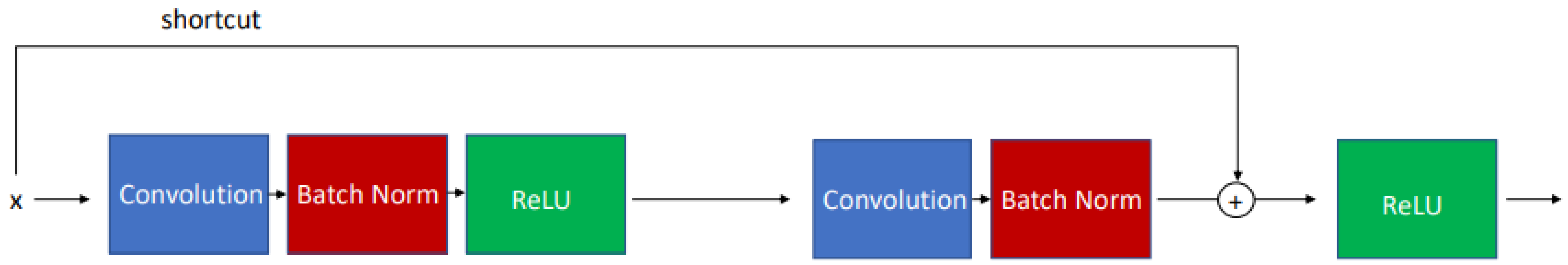


ResNets



In general: $a^{(l+2)} = \sigma\left(z^{(l+2)} + a^{(l)}\right)$

ResNets



$$a^{(l+2)} = \sigma\left(z^{(l+2)} + a^{(l)}\right)$$
$$= \sigma\left(a^{(l+1)}W^{(l+2)} + b^{(l+2)} + a^{(l)}\right)$$

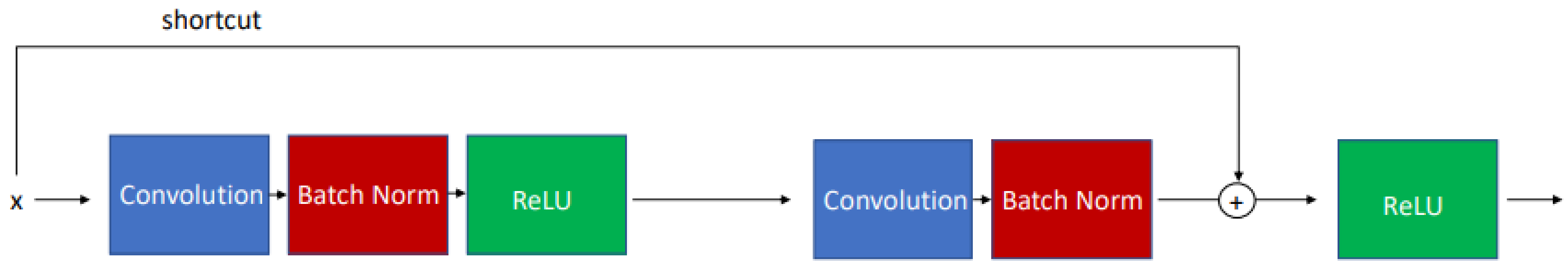
Si todos los pesos y el sesgo son cero, entonces

$$= \sigma\left(a^{(l)}\right) = a^{(l)}$$

Función identidad

debido a ReLU

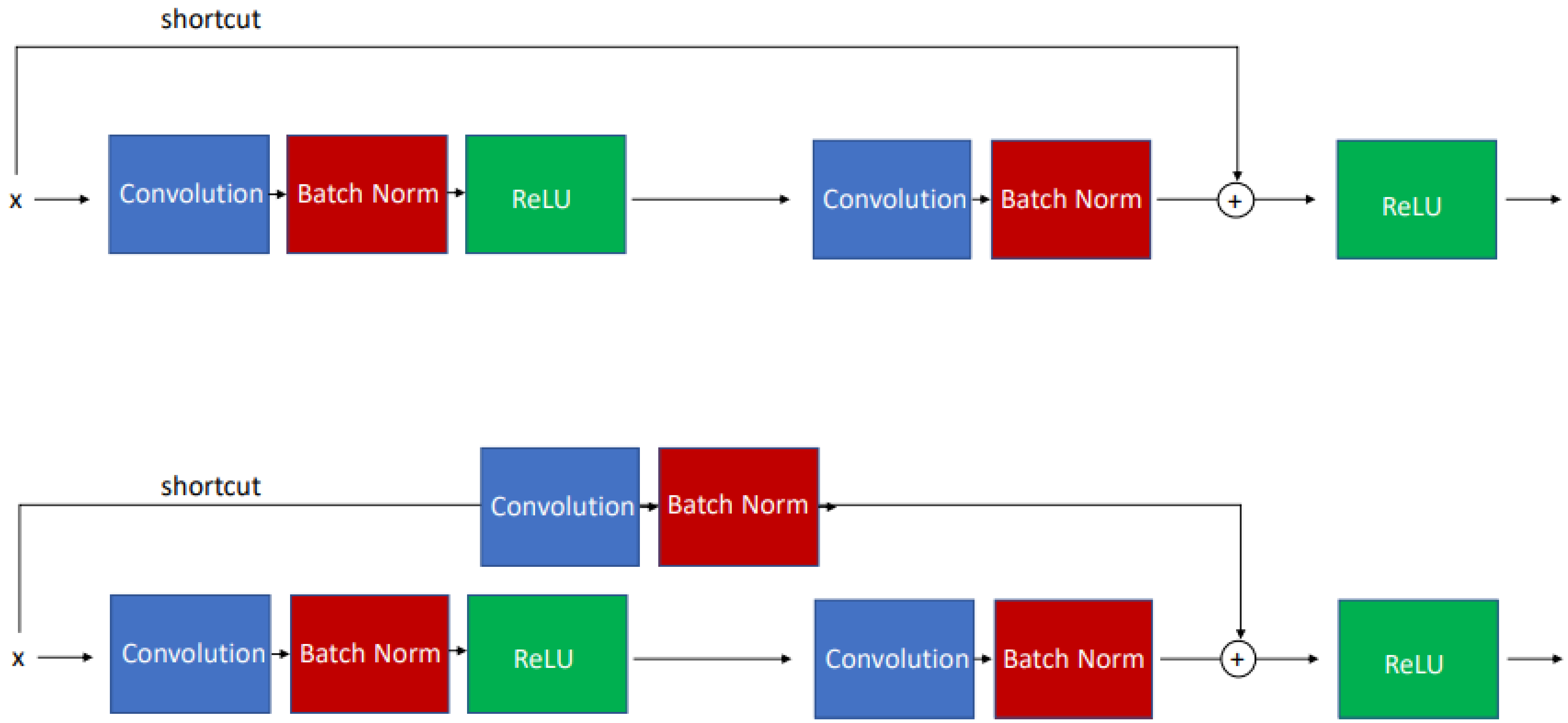
ResNets



$$a^{(l+2)} = \sigma\left(z^{(l+2)} + a^{(l)}\right)$$

Suponemos que tienen la misma dimensión (por ejemplo., A través de "la misma" convolución)

ResNets



Bloques residuales alternativos con conexiones de salto, de modo que la entrada pasada a través del atajo se redimensione a las dimensiones de la salida de la ruta principal

Simplificando CNNs

- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - VGG16 (simple, CNN profunda)
  - ResNet y skip connections
- **Reemplazando Max-Pooling con capas convolucionales**
- Capas convolucionales en lugar de completamente conectadas
- Transfer learning

"Red totalmente convolucional"

Springenberg, Jost Tobias, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "Striving for simplicity: The all convolutional net." arXiv preprint arXiv:1412.6806 (2014).

https://arxiv.org/abs/1412.6806

Idea clave: Reemplace Max Pooling por convoluciones escalonadas (es decir, capas convolucionadas con paso = 2)

Podemos pensar en las "convoluciones escalonadas" como agrupaciones que se pueden aprender

Agrupación promedio global en la última capa

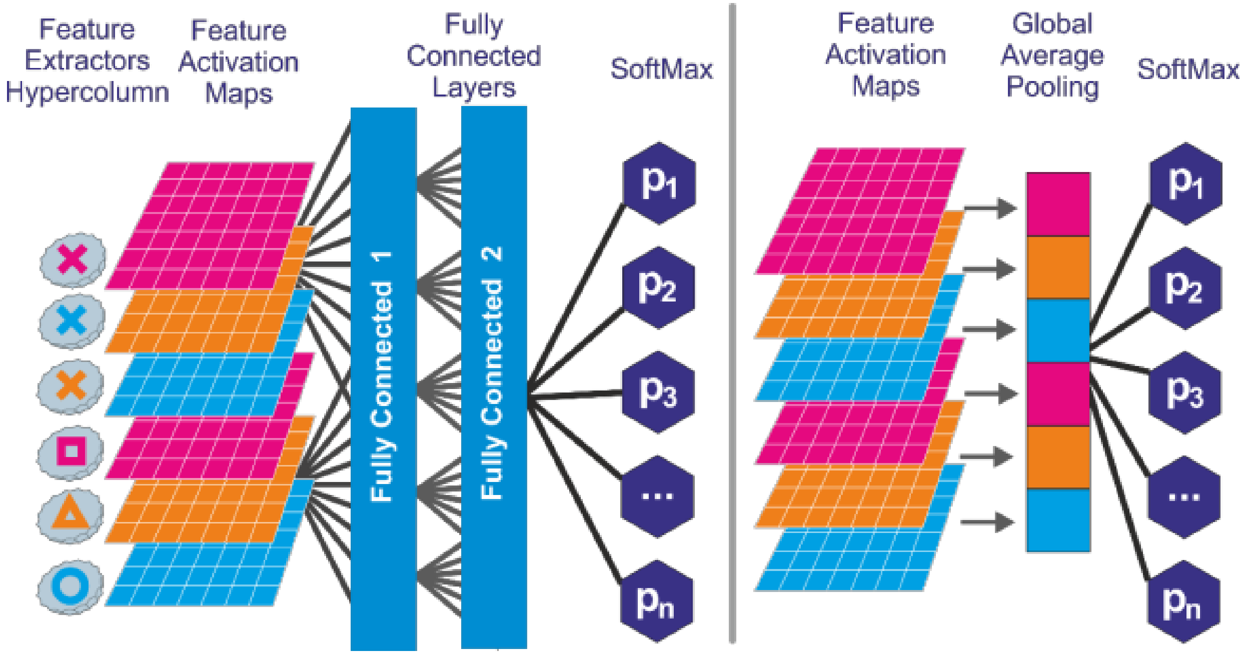


Figure 16: Global average pooling layer replacing the fully connected layers. The output layer implements a Softmax operation with $p_1, p_2, \ldots, p_n$ the predicted probabilities for each class.

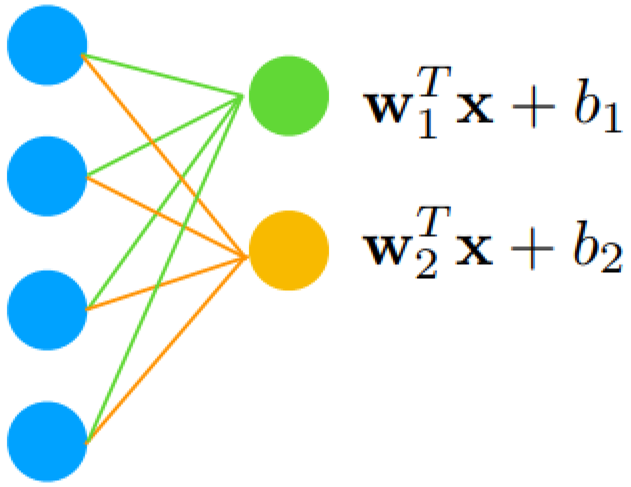
Fuente de la imagen: Singh, Anshuman Vikram. "Content-based image retrieval using deep learning." (2015). http://scholarworks.rit.edu/theses/8828/

Código de Ejemplo

Simplificando CNNs parte 2

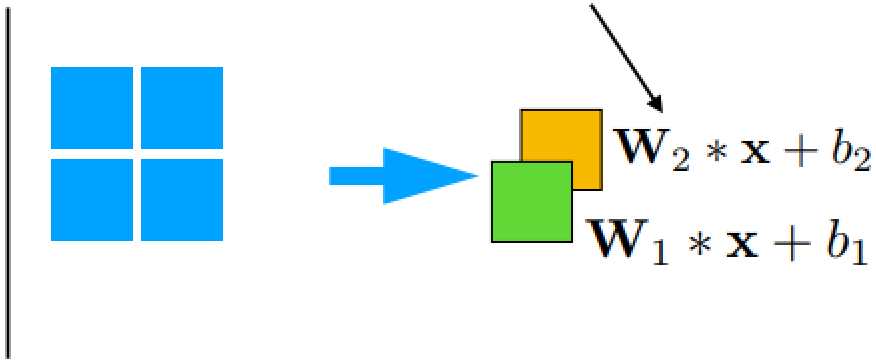
- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - VGG16 (simple, CNN profunda)
  - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- **Capas convolucionales en lugar de completamente conectadas**
- Transfer learning

Es posible reemplazar capas completamente conectadas por capas convolucionales



Fully connected layer

Recuerde, estos también involucran productos punto entre los campos receptivos y los núcleos.



$$\text{where } \mathbf{W}_1 = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{1,3} & w_{1,4} \end{bmatrix}$$

$$\mathbf{W}_2 = \begin{bmatrix} w_{2,1} & w_{2,2} \\ w_{2,3} & w_{2,4} \end{bmatrix}$$

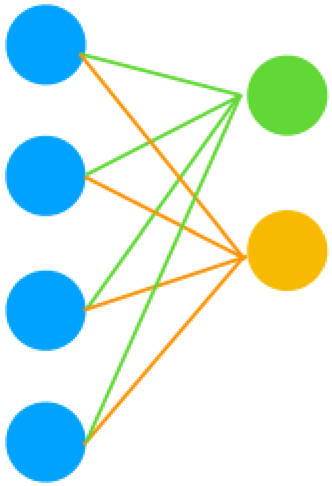
```python
import torch
```

Assume we have a 2x2 input image:

```python
inputs = torch.tensor([[[[1., 2.],
                         [3., 4.]]]])

inputs.shape
```

```
torch.Size([1, 1, 2, 2])
```

NCHW



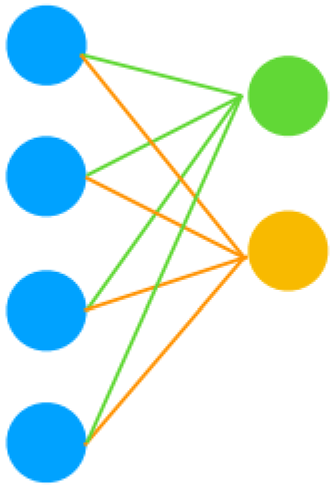
$$\mathbf{w}_1^T \mathbf{x} + b_1$$

$$\mathbf{w}_2^T \mathbf{x} + b_2$$

```python
fc = torch.nn.Linear(4, 2)

weights = torch.tensor([[1.1, 1.2, 1.3, 1.4],
                        [1.5, 1.6, 1.7, 1.8]])
bias = torch.tensor([1.9, 2.0])
fc.weight.data = weights
fc.bias.data = bias
```

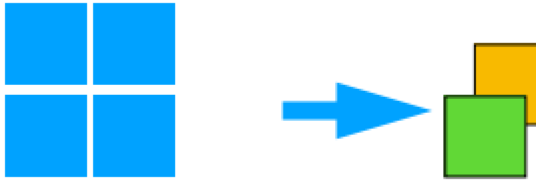
```python
torch.relu(fc(inputs.view(-1, 4)))
```

```
tensor([[14.9000, 19.0000]], grad_fn=<ReluBackward0>)
```



$$\mathbf{w}_1^T\mathbf{x} + b_1$$

$$\mathbf{w}_2^T\mathbf{x} + b_2$$



$$\mathbf{W}_2 * \mathbf{x} + b_2$$

$$\mathbf{W}_1 * \mathbf{x} + b_1$$

```
import torch
```

Assume we have a 2x2 input image:

```
inputs = torch.tensor([[[[1., 2.],
                         [3., 4.]]]])

inputs.shape
```

```
torch.Size([1, 1, 2, 2])
```

```
fc = torch.nn.Linear(4, 2)

weights = torch.tensor([[1.1, 1.2, 1.3, 1.4],
                        [1.5, 1.6, 1.7, 1.8]])
bias = torch.tensor([1.9, 2.0])
fc.weight.data = weights
fc.bias.data = bias
```

```
torch.relu(fc(inputs.view(-1, 4)))
```

```
tensor([[14.9000, 19.0000]], grad_fn=<ReluBackward0>)
```

```
kernel_size = inputs.squeeze(dim=(0)).squeeze(dim=(0)).size()
kernel_size
```

```
torch.Size([2, 2])
```

```
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=2,
                       kernel_size=kernel_size)
print(conv.weight.size())
print(conv.bias.size())
```

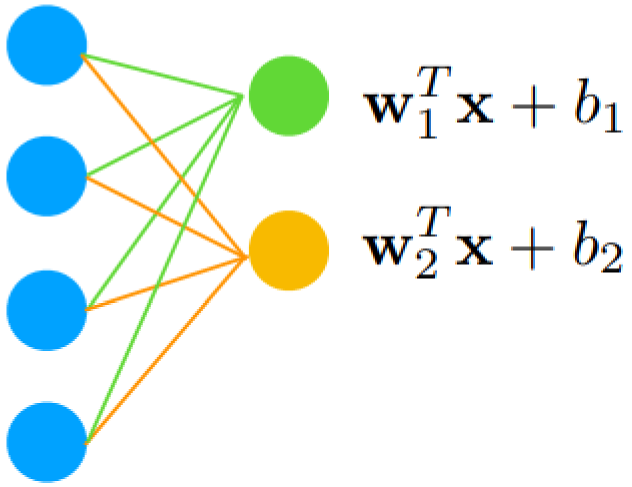
```
torch.Size([2, 1, 2, 2])
torch.Size([2])
```

```
# use same values as before
conv.weight.data = weights.view(2, 1, 2, 2)
conv.bias.data = bias
```

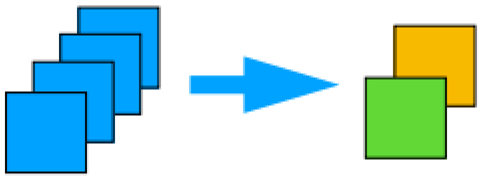
```
torch.relu(conv(inputs))
```

```
tensor([[[[14.9000]],

         [[19.0000]]]], grad_fn=<ReluBackward0>)
```

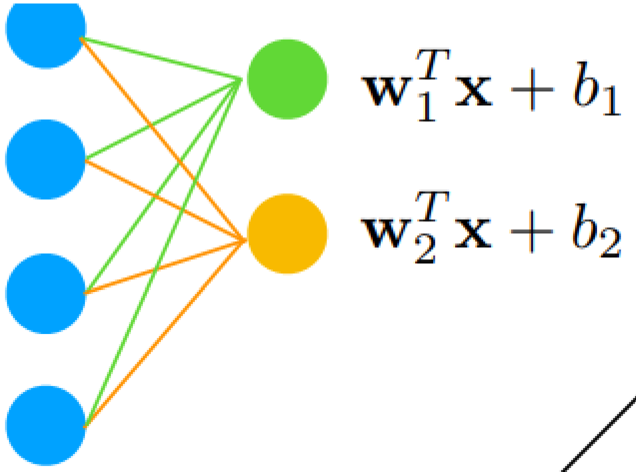
Es posible reemplazar capas completamente conectadas por capas convolucionales

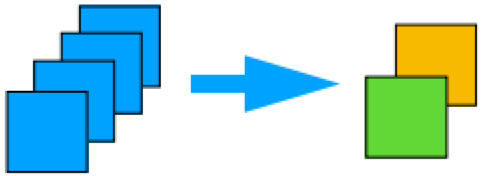


Fully connected layer



O bien, podemos concatenar las entradas en imágenes 1x1 con 4 canales y luego usar 2 núcleos (recuerde, cada núcleo también tiene 4 canales)





```
import torch
```

Assume we have a 2x2 input image:

```
inputs = torch.tensor([[[[1., 2.],
                         [3., 4.]]]])

inputs.shape
```

```
torch.Size([1, 1, 2, 2])
```

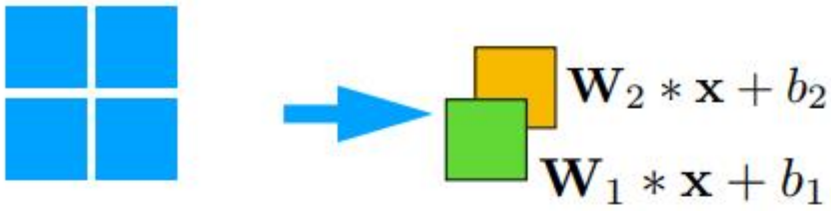
```
fc = torch.nn.Linear(4, 2)

weights = torch.tensor([[1.1, 1.2, 1.3, 1.4],
                        [1.5, 1.6, 1.7, 1.8]])
bias = torch.tensor([1.9, 2.0])
fc.weight.data = weights
fc.bias.data = bias
```

```
torch.relu(fc(inputs.view(-1, 4)))
```

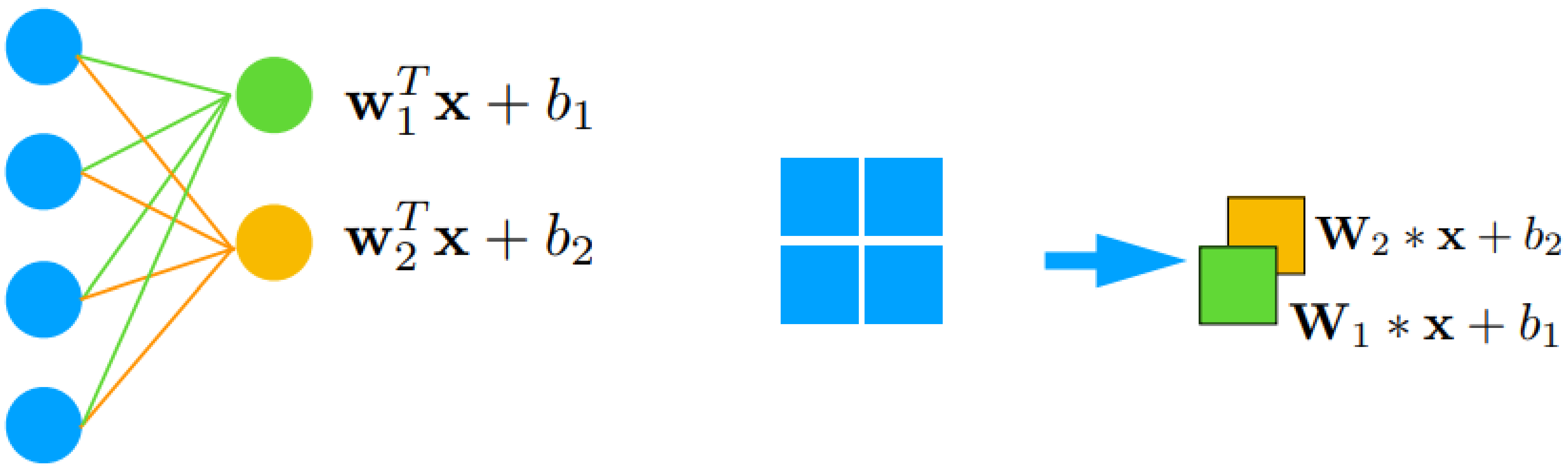
```
tensor([[14.9000, 19.0000]], grad_fn=<ReluBackward0>)
```

```
conv = torch.nn.Conv2d(in_channels=4,
                       out_channels=2,
                       kernel_size=(1, 1))

conv.weight.data = weights.view(2, 4, 1, 1)
conv.bias.data = bias
torch.relu(conv(inputs.view(1, 4, 1, 1)))
```

```
tensor([[[[14.9000]],

         [[19.0000]]]], grad_fn=<ReluBackward0>)
```




```
torch.nn.BatchNorm2d(64),
torch.nn.ReLU(inplace=True),
torch.nn.Conv2d(in_channels=64,
                out_channels=num_classes,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
                bias=False),
torch.nn.BatchNorm2d(10),
torch.nn.ReLU(inplace=True),
# Old:
# torch.nn.AdaptiveAvgPool2d(1),
# New:
torch.nn.Conv2d(in_channels=num_classes,
                out_channels=num_classes,
                kernel_size=(8, 8),
                stride=(1, 1)),
torch.nn.Flatten()
```




```
torch.nn.BatchNorm2d(64),
torch.nn.ReLU(inplace=True),
torch.nn.Conv2d(in_channels=64,
                out_channels=64,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
                bias=False),
torch.nn.BatchNorm2d(64),
torch.nn.ReLU(inplace=True),
torch.Flatten(),
torch.nn.Linear(in_features=8*8*64,
                out_features=num_classes)
```

<=>

```
torch.nn.BatchNorm2d(64),
torch.nn.ReLU(inplace=True),
torch.nn.Conv2d(in_channels=64,
                out_channels=64,
                kernel_size=(3, 3),
                stride=(1, 1),
                padding=1,
                bias=False),
torch.nn.BatchNorm2d(64),
torch.nn.ReLU(inplace=True),
torch.nn.Conv2d(in_channels=64,
                out_channels=num_classes,
                kernel_size=(8, 8),
                stride=(1, 1)),
torch.nn.Flatten()
```

¿Se pueden enseñar trucos nuevos a un perro viejo?

- Padding (controla el tamaño de la salida junto con stride)
- Dropout 2D y batchnorm
- Arquitecturas comunes
  - VGG16 (simple, CNN profunda)
  - ResNet y skip connections
- Reemplazando Max-Pooling con capas convolucionales
- Capas convolucionales en lugar de completamente conectadas
- **Transfer learning**

Transferir aprendizaje

- Una técnica que puede ser útil para sus proyectos de clase.
- Idea clave:
   * Las capas de extracción de características pueden ser útiles en general
   * Utilice un modelo previamente entrenado (por ejemplo, previamente entrenado en ImageNet)
   * Congelar los pesos: Solo entrene la última capa (o las últimas capas)
- Enfoque relacionado: Ajuste, entrene una red previamente entrenada en su conjunto de datos más pequeño

¿Qué capas reemplazar y entrenar?

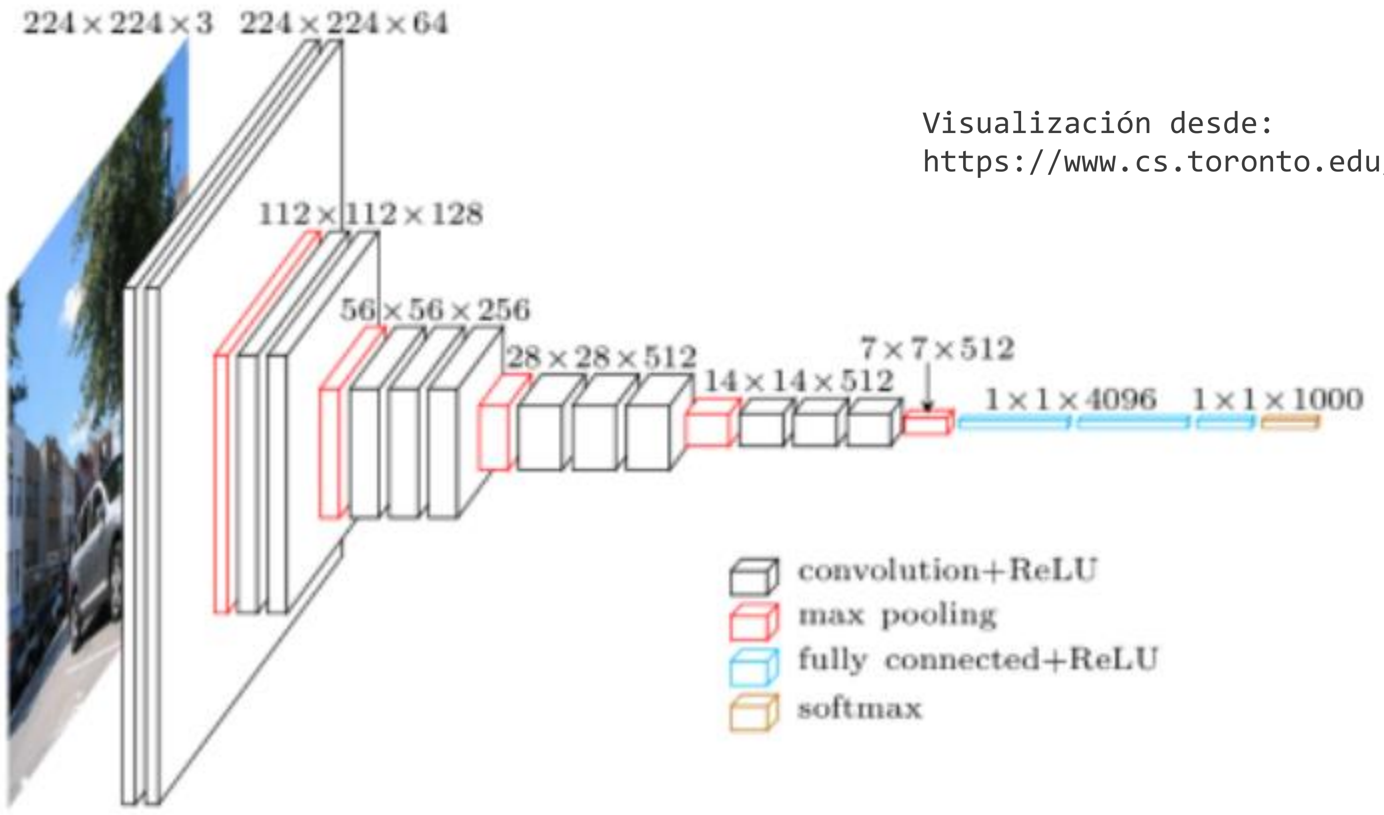
Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (pp. 1725-1732).

https://cs.stanford.edu/people/karpathy/deepvideo/

Model	**3-fold Accuracy**
Soomro et al [22]	43.9%
Feature Histograms + Neural Net	59.0%
Train from scratch	41.3%
Fine-tune top layer	64.1%
Fine-tune top 3 layers	**65.4%**
Fine-tune all layers	62.2%

Table 3: Results on UCF-101 for various Transfer Learning approaches using the Slow Fusion network.

Transfer learning

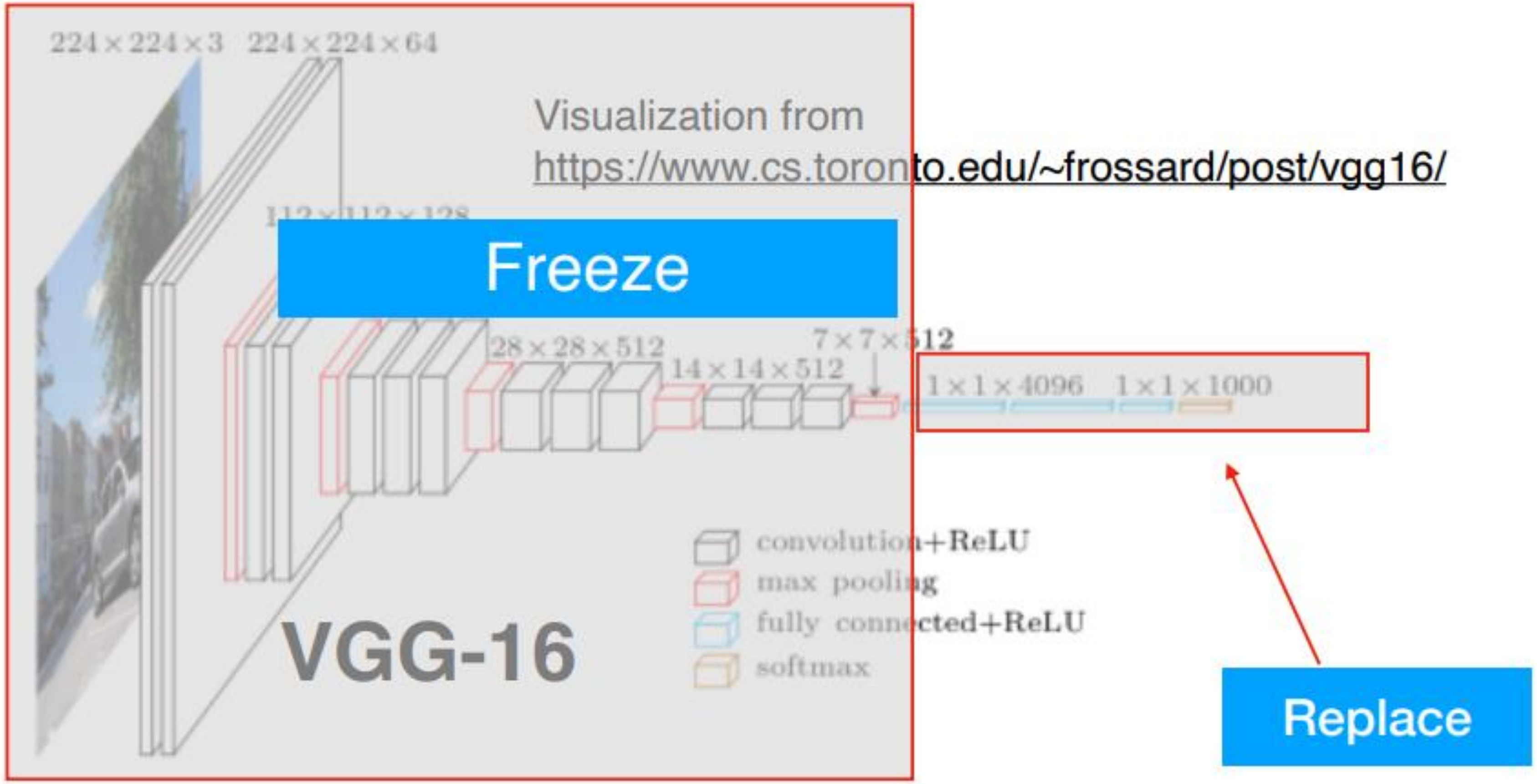


Visualización desde:
https://www.cs.toronto.edu/~frossard/post/vgg16/

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

https://arxiv.org/abs/1409.1556

Transferir aprendizaje



Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

Transfer learning

https://pytorch.org/docs/stable/torchvision/models.html

TORCHVISION.MODELS

The models subpackage contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection and video classification.

Classification

The models subpackage contains definitions for the following model architectures for image classification:

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet
- Inception v3
- GoogLeNet
- ShuffleNet v2
- MobileNet v2
- ResNeXt
- Wide ResNet
- MNASNet

Código de ejemplo de transfer learning

https://pytorch.org/docs/stable/torchvision/models.html

Instancing a pre-trained model will download its weights to a cache directory. This directory can be set using the *TORCH_MODEL_ZOO* environment variable. See `torch.utils.model_zoo.load_url()` for details.

Some models use modules which have different training and evaluation behavior, such as batch normalization. To switch between these modes, use `model.train()` or `model.eval()` as appropriate. See `train()` or `eval()` for details.

All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`. You can use the following transform to normalize:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

Código de ejemplo de transfer learning