

Analisi di Immagini e Video (Computer Vision)

Giuseppe Manco, Francesco S. Pisani

Outline

- Reti Neurali
- CNN
- Architetture di rete

Crediti

- Slides adattate da vari corsi e libri
 - Computer Vision (I. Gkioulekas) - CS CMU Edu
 - Computational Visual Recognition (V. Ordonez), CS Virginia Edu
 - Mohamed Elgendy [Elg20]

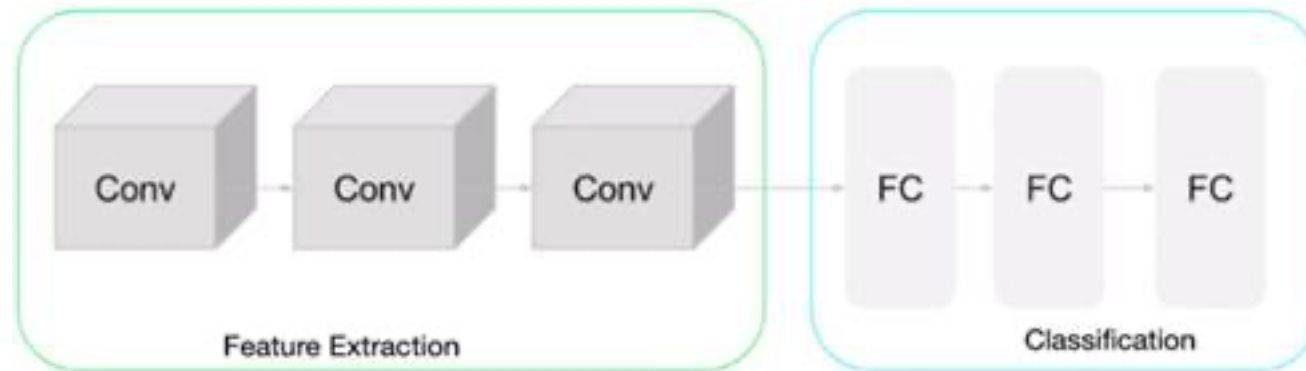
Architetture di rete

Elementi a confronto

- Nuove features
 - Cosa distingue una rete dalle altre?
 - Qual è il problema che cercano di risolvere?
- Architettura
 - Le componenti che la strutturano
- Implementazione
 - Pytorch code
- Setup
 - C'è qualche aspetto particolare che caratterizza il learning?
- Performance
 - Qual è il guadagno?

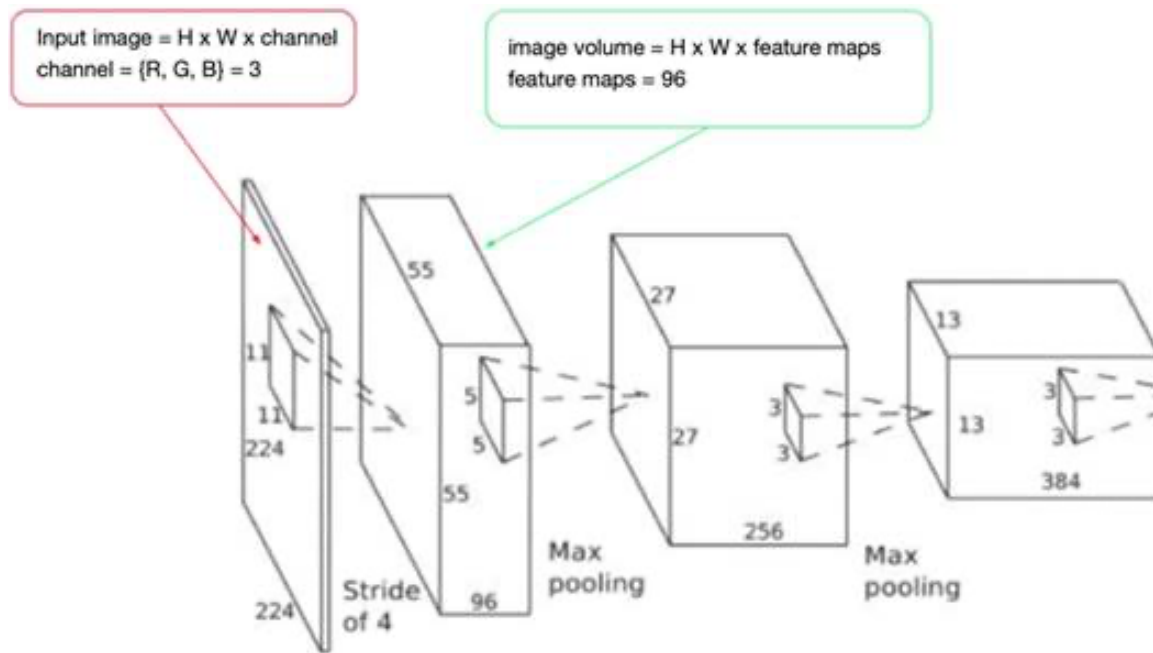
CNN design patterns

- Pattern 1

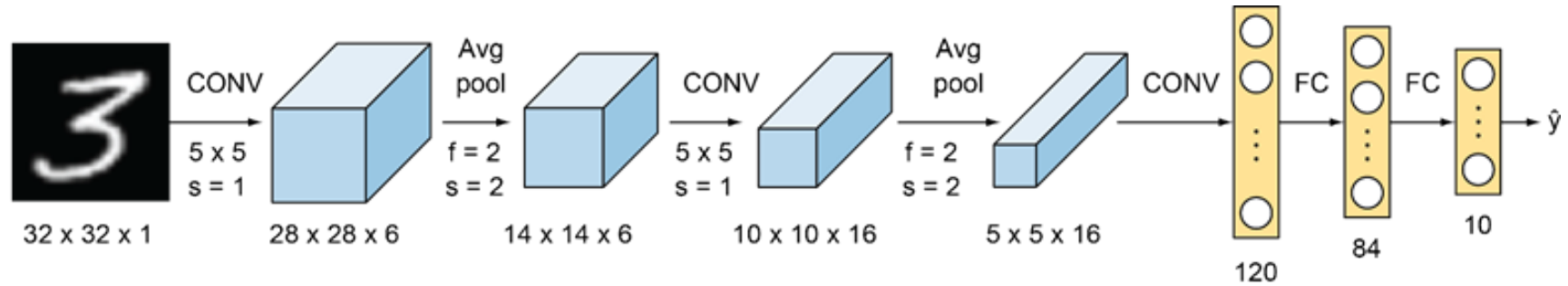


CNN design patterns

- Pattern 2



LeNet-5

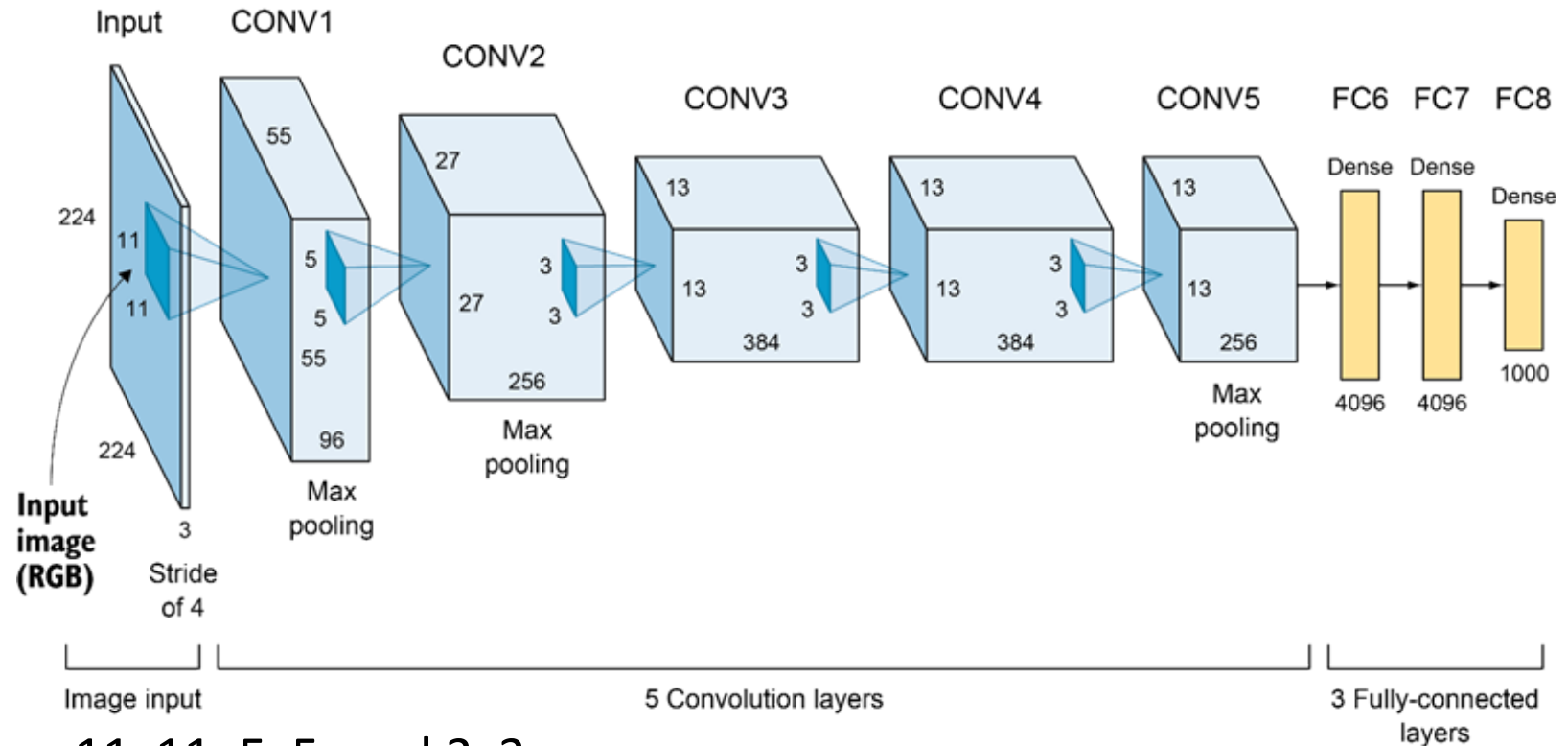


- Quanti pesi?

AlexNet

- Vincitore ImageNet Large Scale Visual Recognition Challenge (ILSVRC) del 2012
 - ImageNet dataset
 - 1.2M high-res images
 - 1,000 classi.
- Alex Krizhevsky, Geoffrey Hinton and Ilya Sutskever

AlexNet

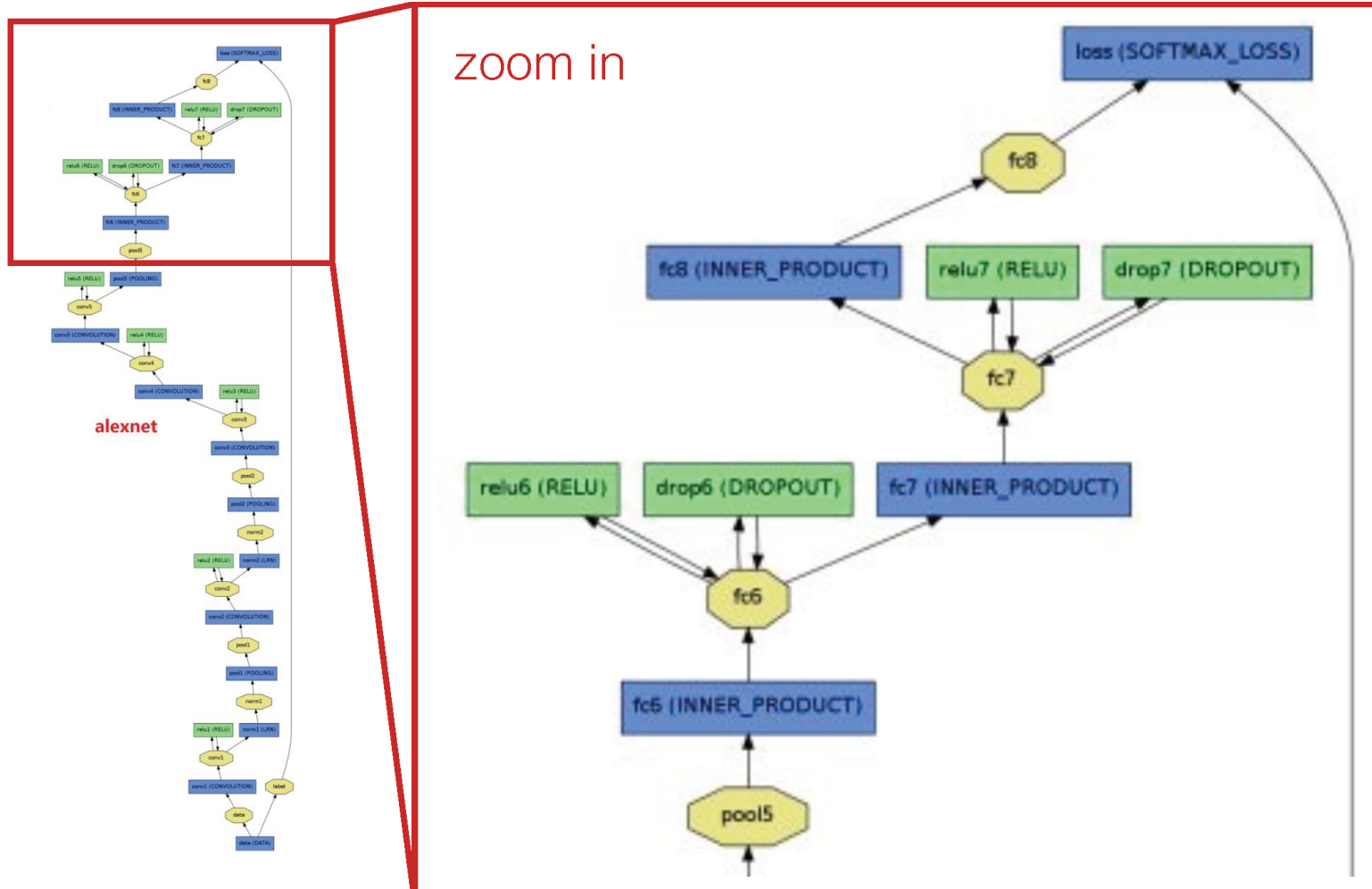


- Convolutional layers: 11x11, 5x5, and 3x3
- Max pooling layers
- Dropout layers
- ReLU activation functions
- Quanti parametri?

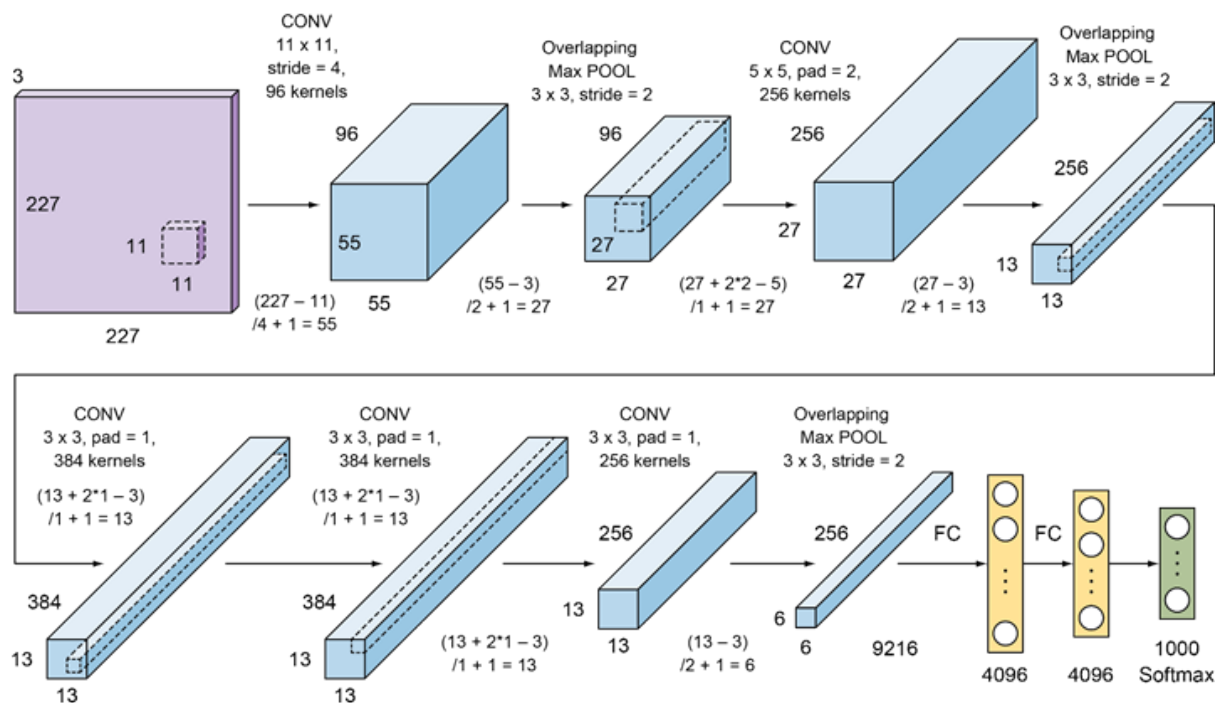
AlexNet: features

- ReLU
- Dropout
 - $p = 0.5$ nei due layers FC
- Data augmentation
 - image rotation, flipping, scaling, ...
- Local response normalization
 - Previene la crescita non limitata dovuta alle attivazioni ReLU
- Weight regularization
 - L2 con peso 0.0005

AlexNet



Alexnet in Pytorch



```

class AlexNet(nn.Module):
    def __init__(self, num_classes = 1000):
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=4),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            LRN(local_size=5, alpha=1e-4, beta=0.75, ACROSS_CHANNELS=True)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=96, out_channels=256, kernel_size=5, groups=2, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            LRN(local_size=5, alpha=1e-4, beta=0.75, ACROSS_CHANNELS=True)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=384, padding=1, kernel_size=3),
            nn.ReLU(inplace=True)
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.layer6 = nn.Sequential(
            nn.Linear(in_features=6*6*256, out_features=4096),
            nn.ReLU(inplace=True),
            nn.Dropout()
        )
        self.layer7 = nn.Sequential(
            nn.Linear(in_features=4096, out_features=4096),
            nn.ReLU(inplace=True),
            nn.Dropout()
        )
        self.layer8 = nn.Linear(in_features=4096, out_features=num_classes)

    def forward(self, x):
        x = self.layer5(self.layer4(self.layer3(self.layer2(self.layer1(x))))))
        x = x.view(-1, 6*6*256)
        x = self.layer6(self.layer7(self.layer8(x)))

        return x
    
```

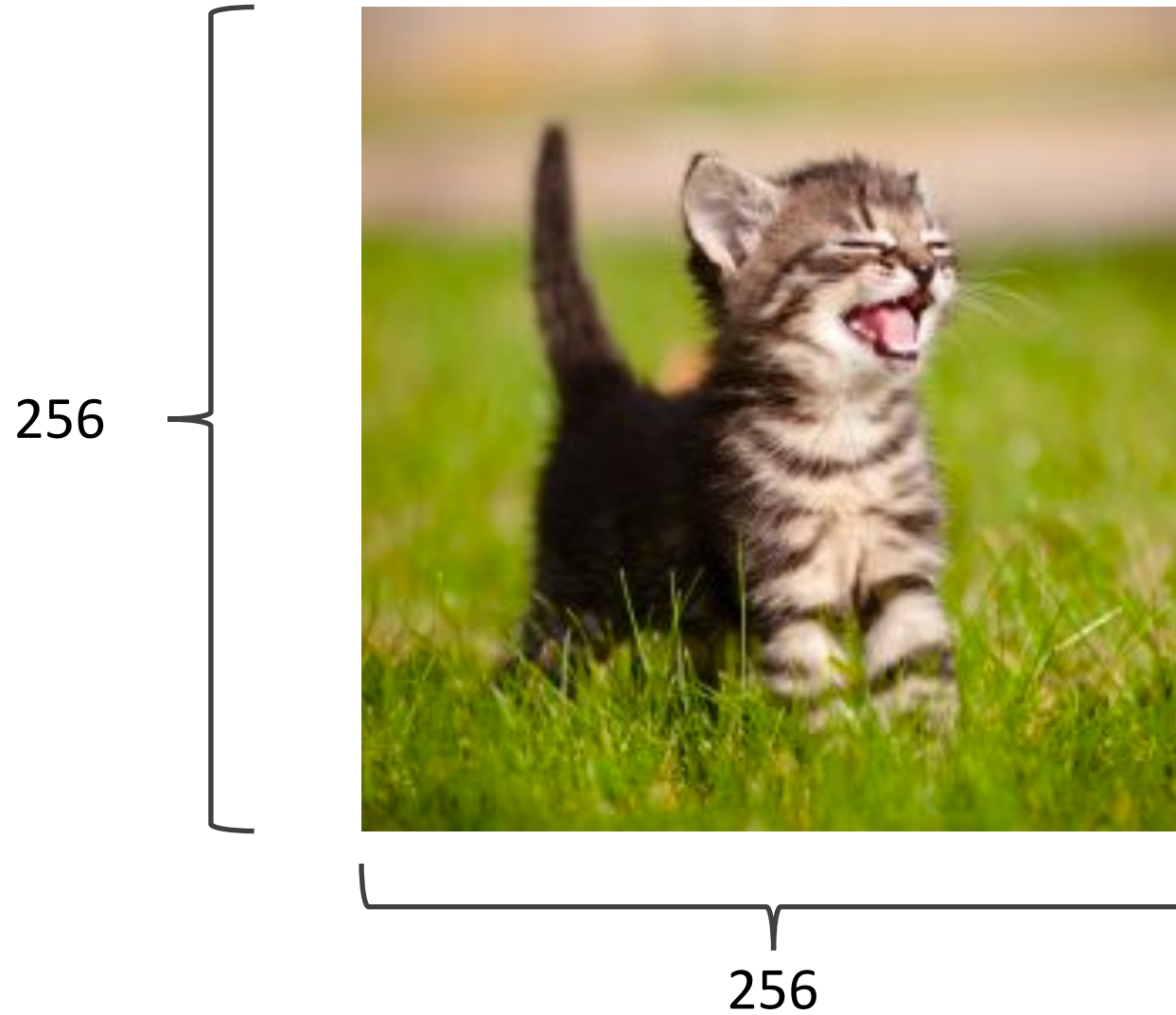
<https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py>

NOTA: Implementazione differente!

Data Augmentation

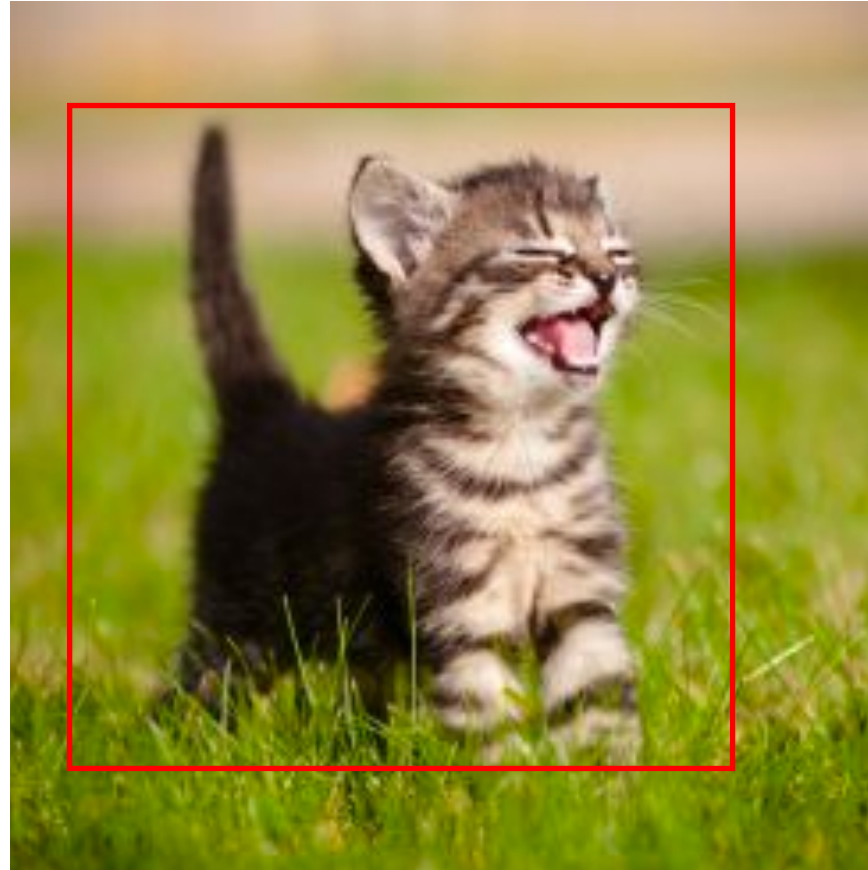


Data Augmentation



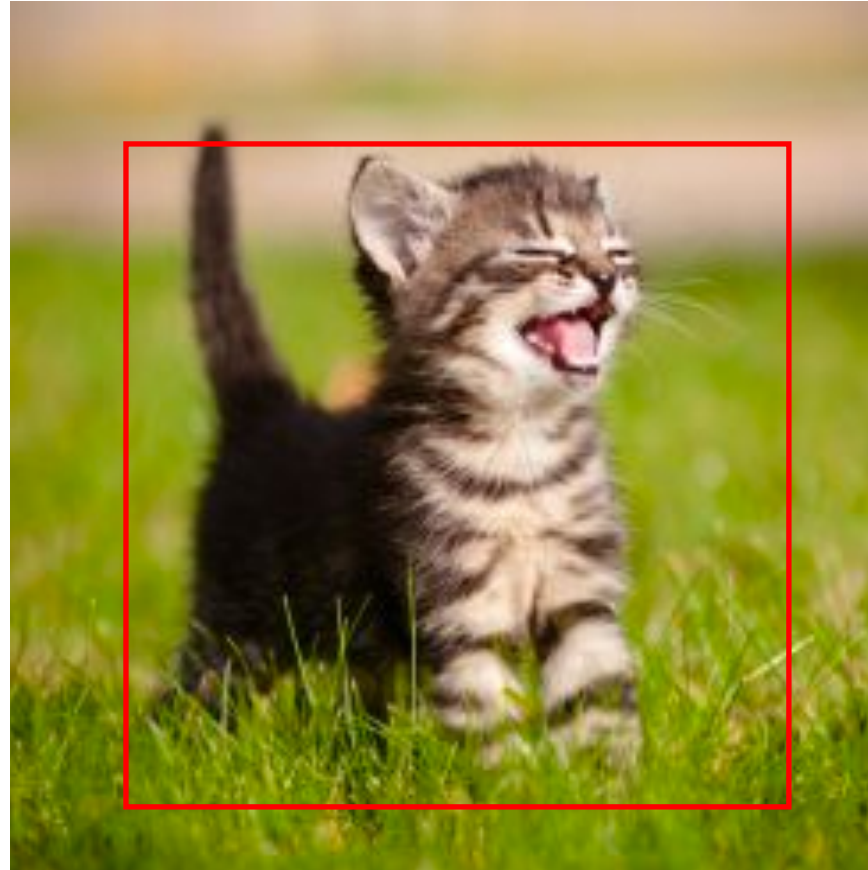
Preprocessing and Data Augmentation

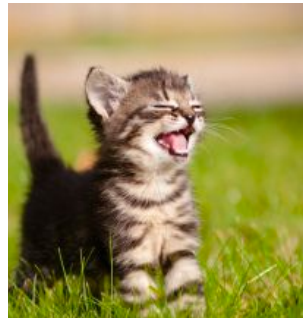
224x224



Preprocessing and Data Augmentation

224x224





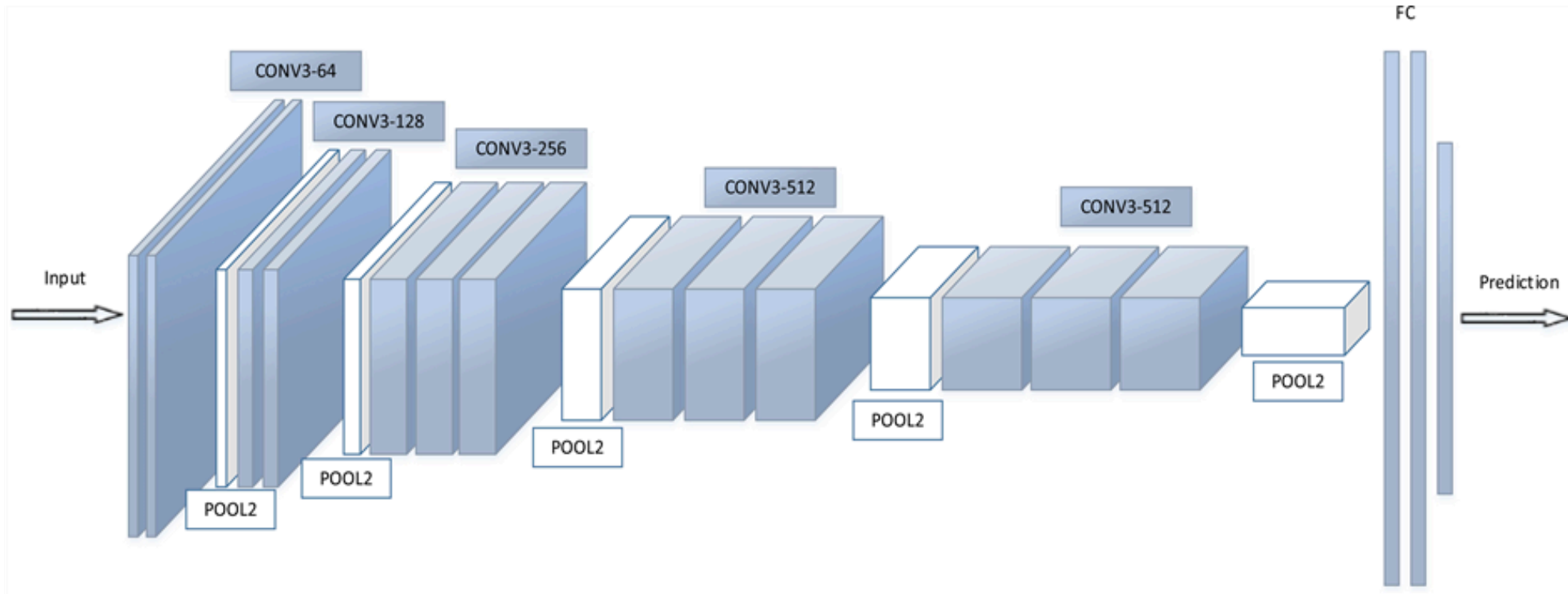
VGG Network

- Visual Geometry Group at Oxford University, 2014
 - Karen Simonyan, Andrew Zisserman
- VGG-16
 - 16 weight layers
 - 13 convolutional layers
 - 3 fully-connected layers
- Semplifica il setup degli iperparametri (kernel size, padding, strides, etc.)
 - Contiene componenti uniformi (CONV/POOL)
 - Rimpiazza i filtri di grandi dimensioni con cascate di filtri
 - Tutti i layer convoluzionali sono 3x3 con stride = 1 e padding same
 - Tutti i layer di pooling sono 2x2 pool-size con stride = 2

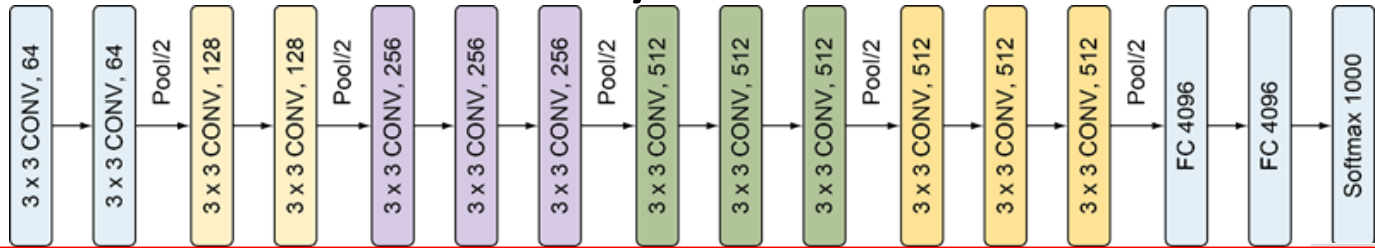
Perché cascate di filtri piccoli?

- Layer non lineari multipli apprendono features più complesse con un numero minimo di parametri
 - 3 layer di 3x3 CONV con C channels $\rightarrow 27C^2$ pesi, un layer 7x7 ne richiede $49C^2$
- Uno stack di due 3x3 CONV ha lo stesso effetto di un 5x5
 - tre 3x3 CONV hanno lo stesso effetto di un 7x7

VGG Network



VGG16 in Pytorch



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
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
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
maxpool					
FC-4096					
FC-4096					
FC-1000					

```
class VGG(nn.Module):
```

```
def __init__(self, features, num_classes=1000):
    super(VGG, self).__init__()
    self.features = features
    self.classifier = nn.Sequential(
        nn.Linear(512 * 7 * 7, 4096),
        nn.ReLU(True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(True),
        nn.Dropout(),
        nn.Linear(4096, num_classes),
    )
```

```
def forward(self, x):
    x = self.features(x)
    x = x.view(x.size(0), -1)
    x = self.classifier(x)
    return x
```

```
def make_layers(cfg, batch_norm=False):
```

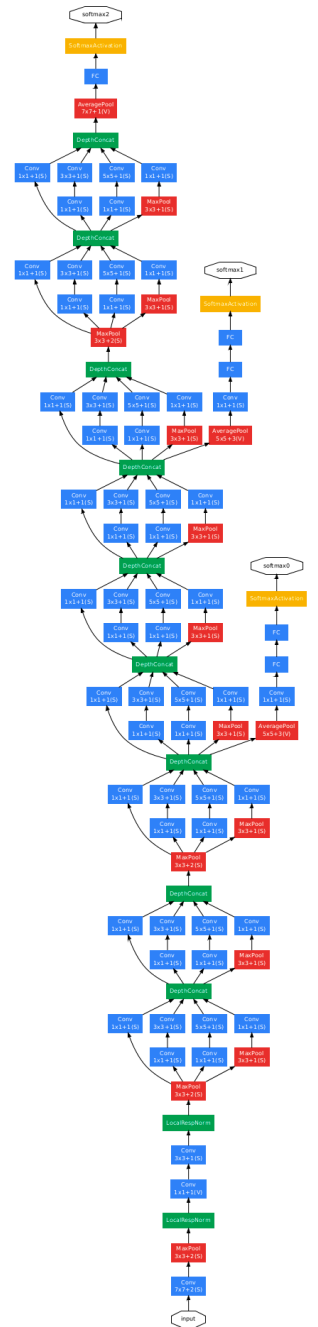
```
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

```
cfg = {
```

```
'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 'M'],
'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 'M'],
}
```

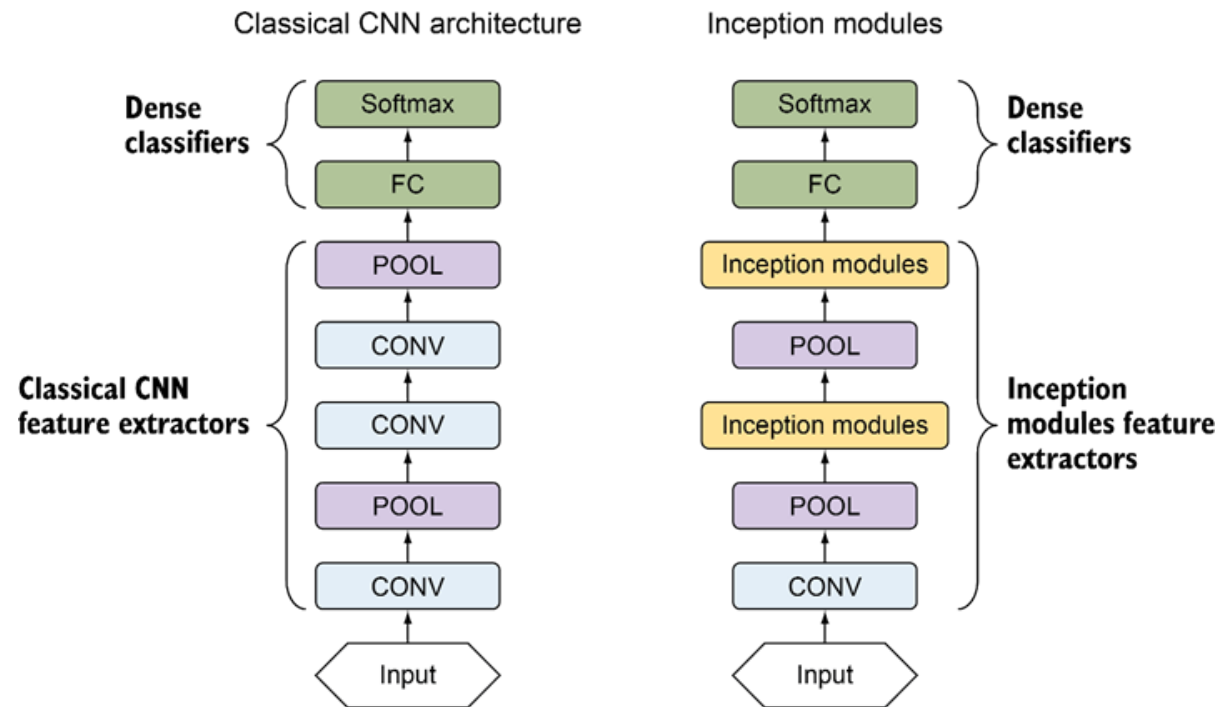
GoogLeNet

- Proposta da Google nel 2014
 - ILSVRC14
 - Inception network
 - 22 layers: più grande di VGGNet con meno parametri (da ~138M a ~13M)
 - Inception Module
 - Che size per i filtri?
 - Quando usare il pooling?
 - **Idea: combiniamoli!**



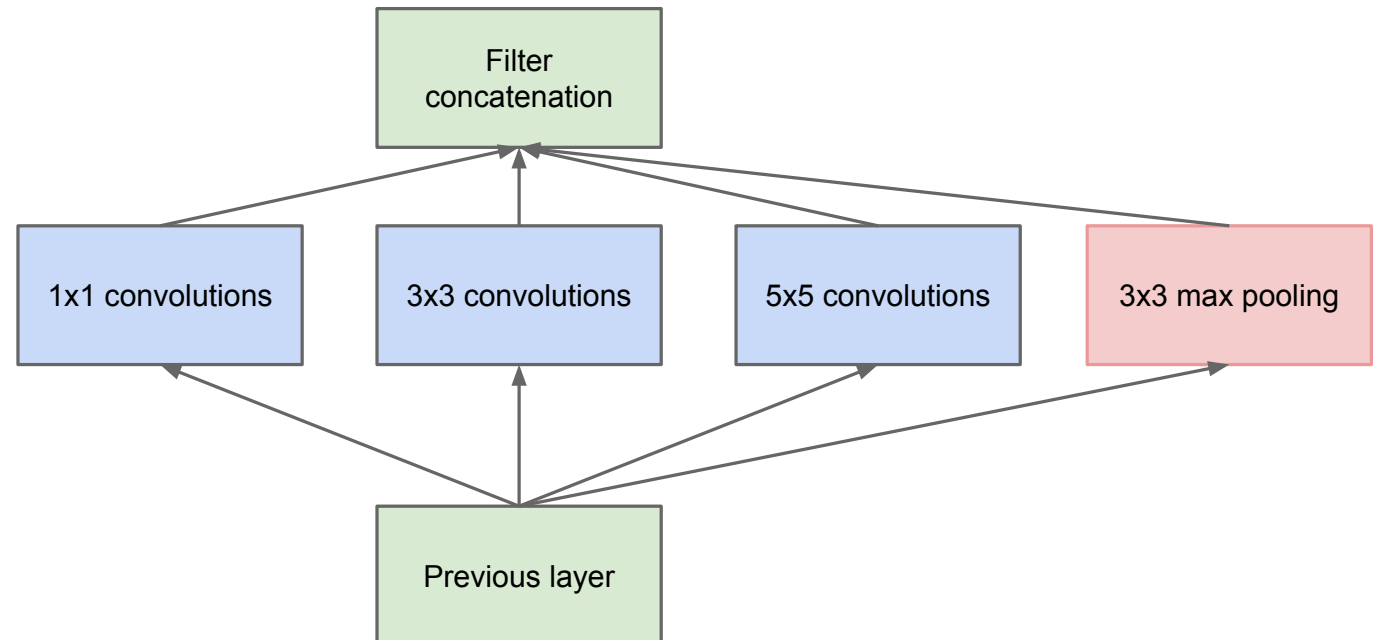
Inception Network

- Stacking di moduli inception
- Per limitare il numero di calcoli adotta un approccio di dimensionality reduction prima di ogni kernel

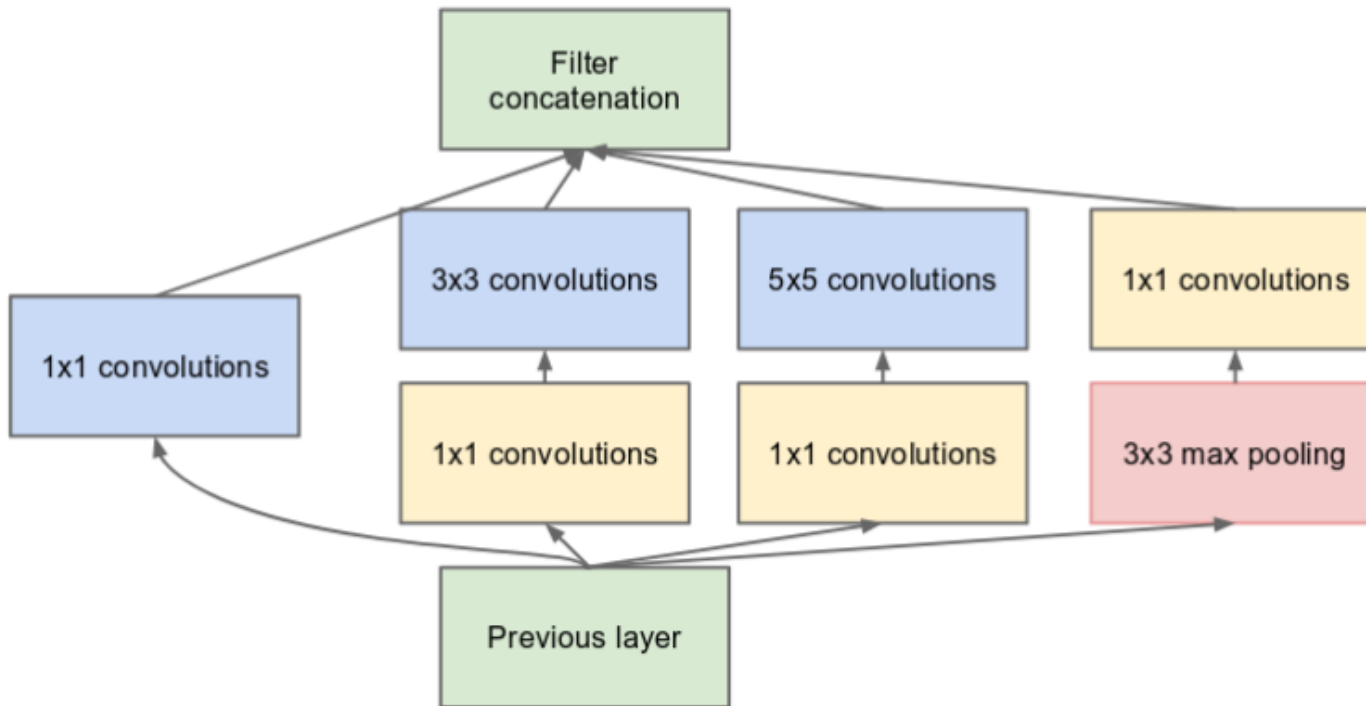


Inception Network - Module

- Quattro layer concatenati
 - 1x1 CONV
 - 3x3 CONV
 - 5x5 CONV
 - 3x3 MaxPOOL
- Quante operazioni?



Inception Network – Module



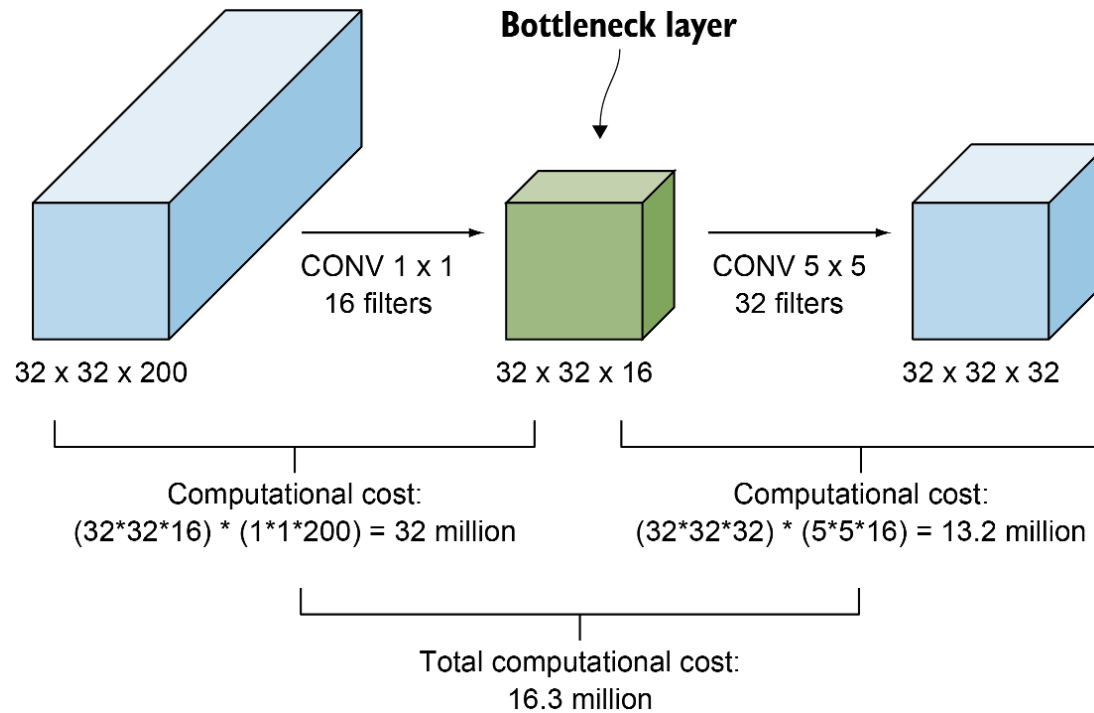
Ogni modulo contiene i filtri 1x1, 3x3, 5x5

L'output è composto dalla concatenazione dei risultati dei kernel

Un blocco MaxPool 3x3 è presente nel modulo

I blocchi in giallo (1x1) sono i blocchi di dimensionality reduction

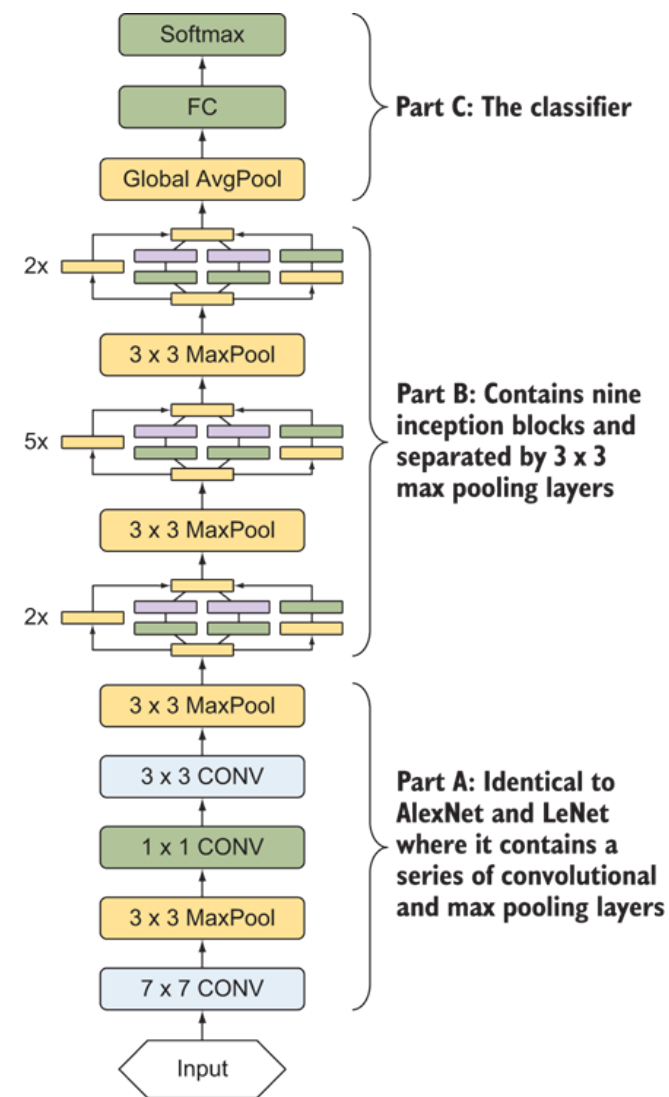
Inception Network – Complessità



~16M vs ~163M di operazioni

GoogLeNet in Pytorch

	type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
Part A	convolution	7×7/2	112×112×64	1							2.7K	34M
	max pool	3×3/2	56×56×64	0								
	convolution	3×3/1	56×56×192	2		64	192				112K	360M
	max pool	3×3/2	28×28×192	0								
Part B	inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
	inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
	max pool	3×3/2	14×14×480	0								
	inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
	inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
	inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
	inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
	inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
	max pool	3×3/2	7×7×832	0								
	inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M	
Part D	avg pool	7×7/1	1×1×1024	0								
	dropout (40%)		1×1×1024	0								
	linear		1×1×1000	1							1000K	1M
	softmax		1×1×1000	0								



GoogLeNet in Pytorch

```
class Inception(nn.Module):
    def __init__(self, in_planes, n1x1, n3x3red, n3x3, n5x5red, n5x5, pool_planes):
        super(Inception, self).__init__()
        # 1x1 conv branch
        self.b1 = nn.Sequential(
            nn.Conv2d(in_planes, n1x1, kernel_size=1),
            nn.ReLU(True),
        )

        # 1x1 conv -> 3x3 conv branch
        self.b2 = nn.Sequential(
            nn.Conv2d(in_planes, n3x3red, kernel_size=1),
            nn.ReLU(True),
            nn.Conv2d(n3x3red, n3x3, kernel_size=3, padding=1),
            nn.ReLU(True),
        )

        # 1x1 conv -> 5x5 conv branch
        self.b3 = nn.Sequential(
            nn.Conv2d(in_planes, n5x5red, kernel_size=1),
            nn.ReLU(True),
            nn.Conv2d(n5x5red, n5x5, kernel_size=3, padding=1),
            nn.ReLU(True),
        )

        # 3x3 pool -> 1x1 conv branch
        self.b4 = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            nn.Conv2d(in_planes, pool_planes, kernel_size=1),
            nn.ReLU(True),
        )

    def forward(self, x):
        y1 = self.b1(x)
        y2 = self.b2(x)
        y3 = self.b3(x)
        y4 = self.b4(x)
        return torch.cat([y1,y2,y3,y4], 1)
```

```
class GoogLeNet(nn.Module):
    def __init__(self):
        super(GoogLeNet, self).__init__()
        self.pre_layers = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(True),
            nn.MaxPool2d(3, stride=2, padding=1),
            nn.Conv2d(64, 192, kernel_size=1, stride=1),
            nn.ReLU(True),
            nn.Conv2d(192, 192, kernel_size=3, stride=1, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(3, stride=2, padding=1)
        )

        self.a3 = Inception(192, 64, 96, 128, 16, 32, 32)
        self.b3 = Inception(256, 128, 128, 192, 32, 96, 64)

        self.maxpool = nn.MaxPool2d(3, stride=2, padding=1)

        self.a4 = Inception(480, 192, 96, 208, 16, 48, 64)
        self.b4 = Inception(512, 160, 112, 224, 24, 64, 64)
        self.c4 = Inception(512, 128, 128, 256, 24, 64, 64)
        self.d4 = Inception(512, 112, 144, 288, 32, 64, 64)
        self.e4 = Inception(528, 256, 160, 320, 32, 128, 128)

        self.a5 = Inception(832, 256, 160, 320, 32, 128, 128)
        self.b5 = Inception(832, 384, 192, 384, 48, 128, 128)

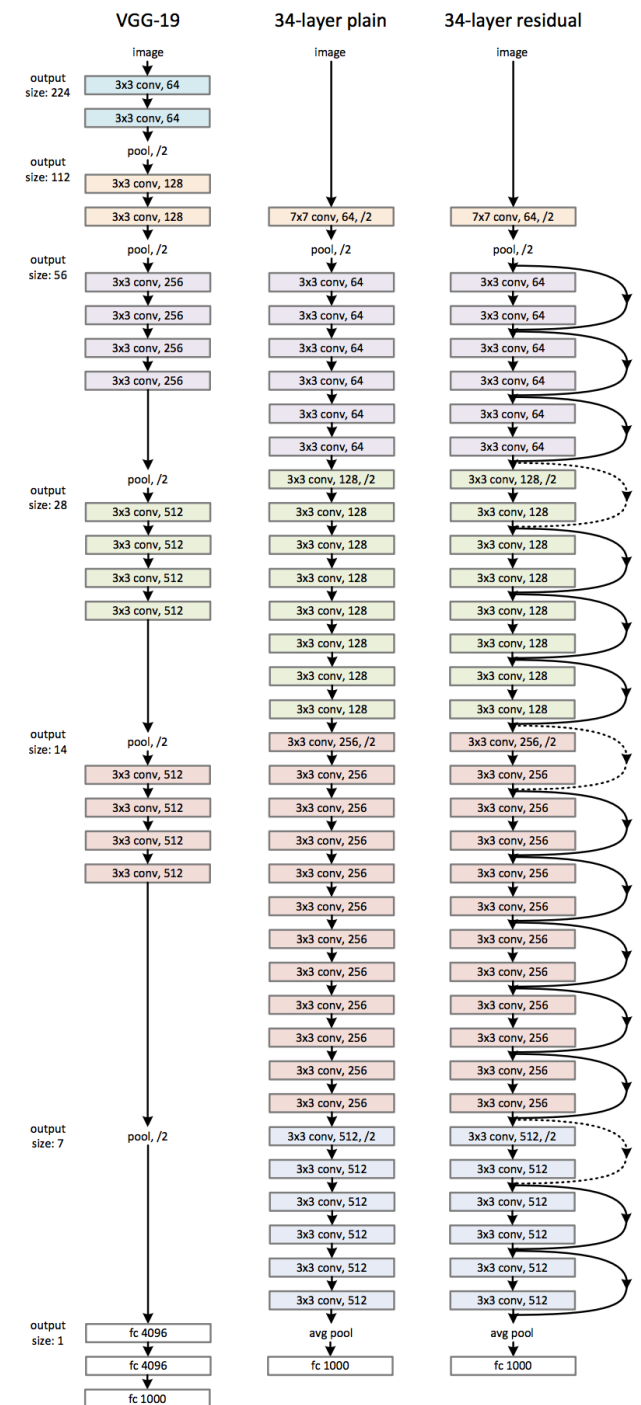
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.linear = nn.Linear(1024, 10)

        self.dropout = nn.Dropout(0.4)

    def forward(self, x):
        out = self.pre_layers(x)
        out = self.a3(out)
        out = self.b3(out)
        out = self.maxpool(out)
        out = self.a4(out)
        out = self.b4(out)
        out = self.c4(out)
        out = self.d4(out)
        out = self.e4(out)
        out = self.maxpool(out)
        out = self.a5(out)
        out = self.b5(out)
        out = self.avgpool(out)
        out = self.dropout(out)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

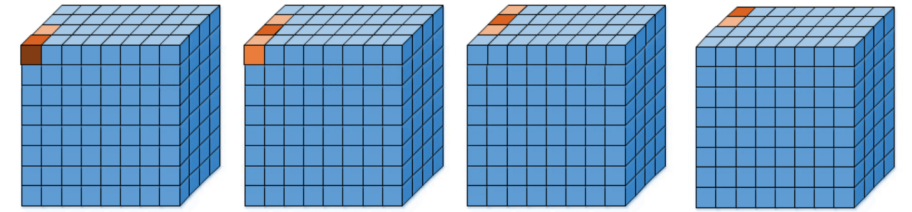
ResNet

- Introdotta nel 2015 da Kaiming He et al (Microsoft Research)
- Utilizza Skip connections chiamate residual module
- Batch normalization
 - 50, 101, and 152 weight layers
 - Complessità minore di reti più piccole come VGGNet
- Vincitore ILSVRC15
- Possiamo costruire very deep layers?
 - Vanishing gradient

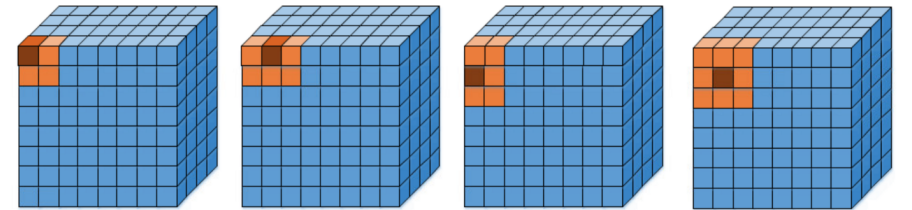


Local Response Normalization, Batch Normalization

- $$b_{x,y}^i = \frac{a_{x,y}^i}{\left(k + \alpha \sum_{i-\frac{n}{2}}^{i+\frac{n}{2}} (a_{x,y}^j)^2\right)}$$
- $$b_{x,y}^i = \frac{a_{x,y}^i}{\left(k + \alpha \sum_{x-\frac{n}{2}}^{x+\frac{n}{2}} \sum_{y-\frac{n}{2}}^{y+\frac{n}{2}} (a_{u,v}^i)^2\right)}$$



a) Inter-Channel LRN (n=2)



b) Intra-Channel LRN (n=2)

BatchNormalization Layer

- Ogni layer è un input per i layer successivi
- Problema
 - Ogni passo di backprop cambia i pesi
 - Risultato:
 - la distribuzione dei layer può cambiare durante la fase di training
 - **Covariance-shift!**
- Rimedio:
 - Normalizzazione, scala e shift

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

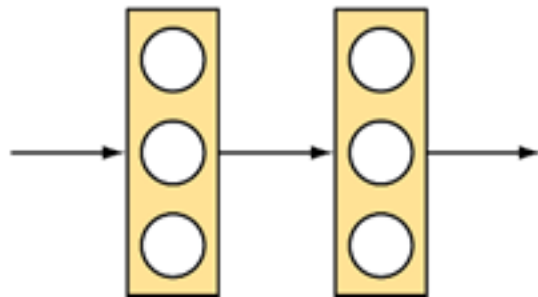
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

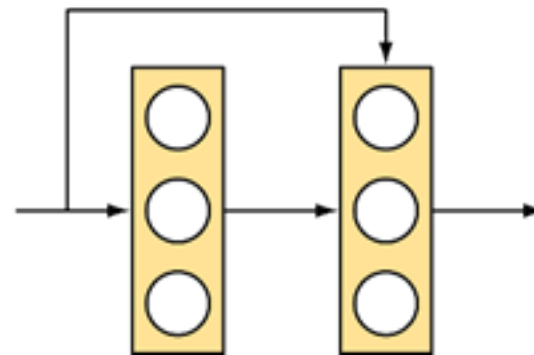
Skip Connections

- Uno shortcut che permette al gradiente di propagarsi ai layers iniziali
- Identity function
 - Ogni layer include le performance del layer precedente

Without skip connection

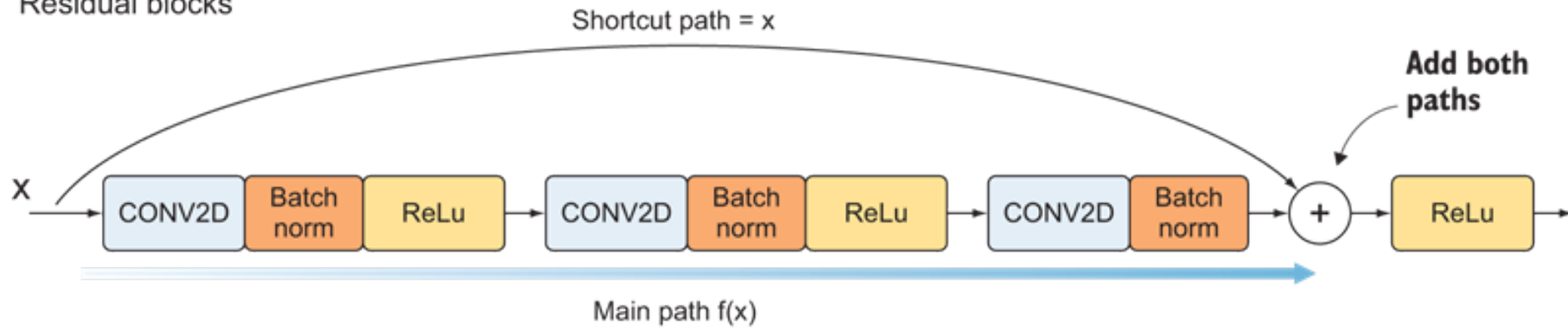


With skip connection

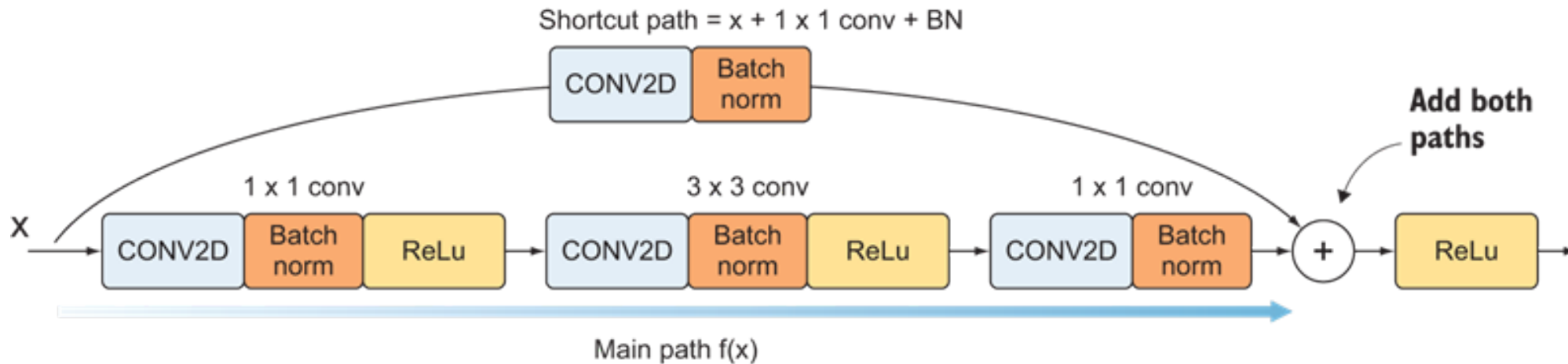


Residual blocks

Residual blocks

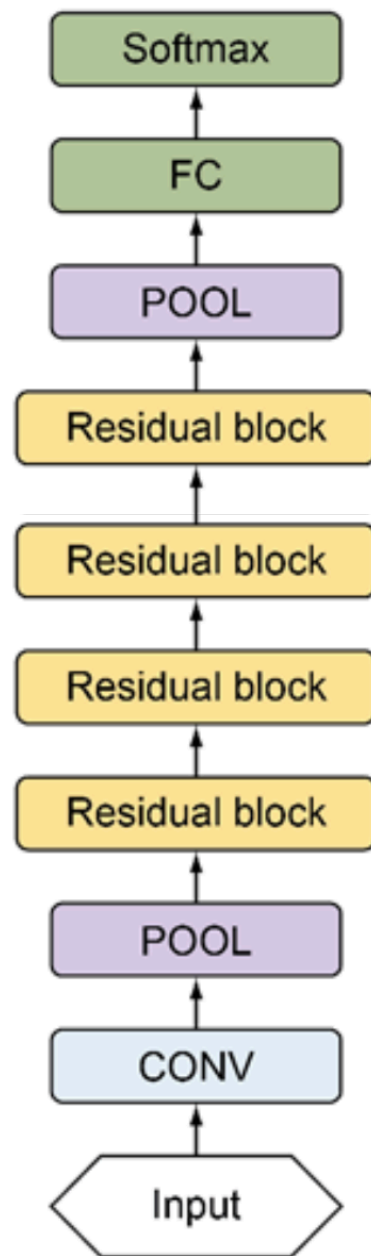


Bottleneck residual block with reduce shortcut



ResNet in Pytorch

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9



ResNet in Pytorch

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, bn_channels, stride=1, bottleneck=False):
        super(ResidualBlock, self).__init__()

        if bottleneck:
            self.expansion = 4
        else:
            self.expansion = 1

        out_channels = bn_channels * self.expansion

        if bottleneck:
            self.block = nn.Sequential(
                nn.Conv2d(in_channels, bn_channels, kernel_size=1, padding=0, bias=False),
                nn.BatchNorm2d(bn_channels),
                nn.ReLU(True),
                nn.Conv2d(bn_channels, bn_channels, kernel_size=3, stride=stride, padding=1, bias=False),
                nn.BatchNorm2d(bn_channels),
                nn.ReLU(True),
                nn.Conv2d(bn_channels, out_channels, kernel_size=1, padding=0, bias=False),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.block = nn.Sequential(
                nn.Conv2d(in_channels, bn_channels, kernel_size=3, stride=stride, padding=1, bias=False),
                nn.BatchNorm2d(bn_channels),
                nn.ReLU(True),
                nn.Conv2d(bn_channels, out_channels, kernel_size=3, padding=1, bias=False),
                nn.BatchNorm2d(out_channels),
            )

        if in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.shortcut = nn.Sequential()

    def forward(self, x):
        out = self.block(x)
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

```
class ResNet(nn.Module):
    def __init__(self, layers, bottleneck=False):
        super(ResNet, self).__init__()

        self.in_channels = 64
        self.bottleneck = bottleneck

        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.MaxPool2d(3, stride=2, padding=1)
        )

        self.conv2_x = self._make_layer(64, layers[0])
        self.conv3_x = self._make_layer(128, layers[1], stride=2)
        self.conv4_x = self._make_layer(256, layers[2], stride=2)
        self.conv5_x = self._make_layer(512, layers[3], stride=2)

        self.avgpool = nn.AvgPool2d((1, 1))
        self.fc = nn.Linear(self.in_channels*7*7, 10)

    def _make_layer(self, out_channels, blocks, stride=1):
        layers = []
        for index in range(blocks):
            if index == 0:
                block = ResidualBlock(self.in_channels, out_channels, stride, bottleneck=self.bottleneck)
            else:
                block = ResidualBlock(self.in_channels, out_channels, stride=1, bottleneck=self.bottleneck)
            layers.append(block)
            self.in_channels = out_channels*block.expansion

        return nn.Sequential(*layers)

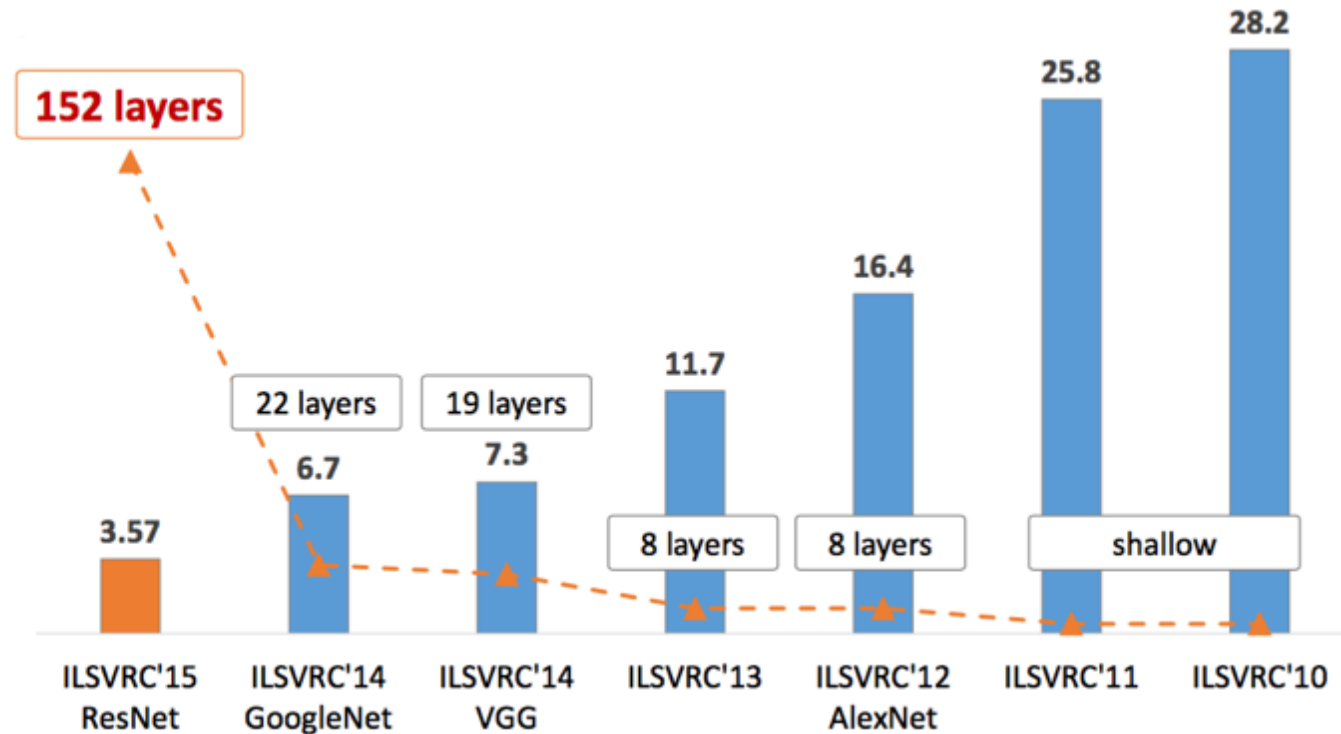
    def forward(self, x):
        x = self.conv1(x)

        x = self.conv2_x(x)
        x = self.conv3_x(x)
        x = self.conv4_x(x)
        x = self.conv5_x(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

Quali sono le performance?



Tecniche di data augmentation

- Position augmentation
 - Scaling
 - Cropping
 - Flipping
 - Padding
 - Rotation
 - Translation
 - Affine Transformation
- Color augmentation
 - Brightness
 - Contrast
 - Saturation
 - Hue



Pytorch

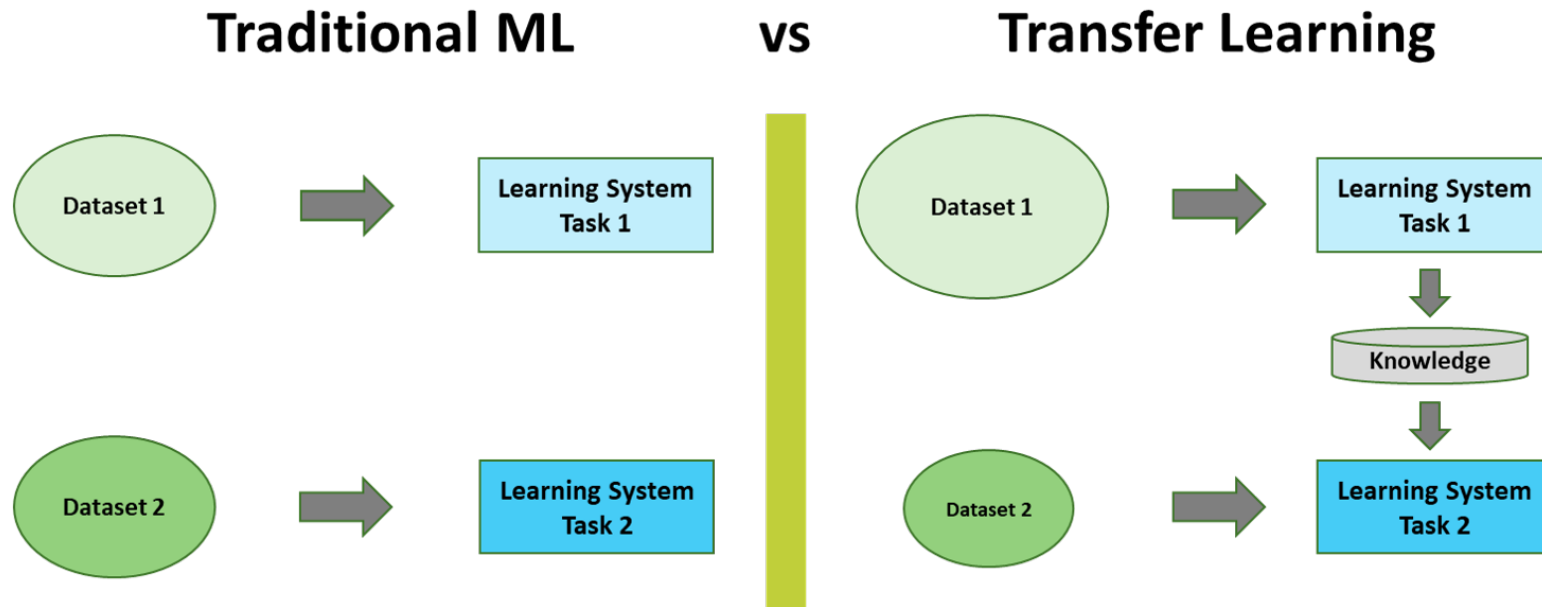
```
loader_transform = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomResizedCrop(140),
    transforms.RandomHorizontalFlip()
])

datasets.ImageFolder(root=train_dir, transform=loader_transform)
```

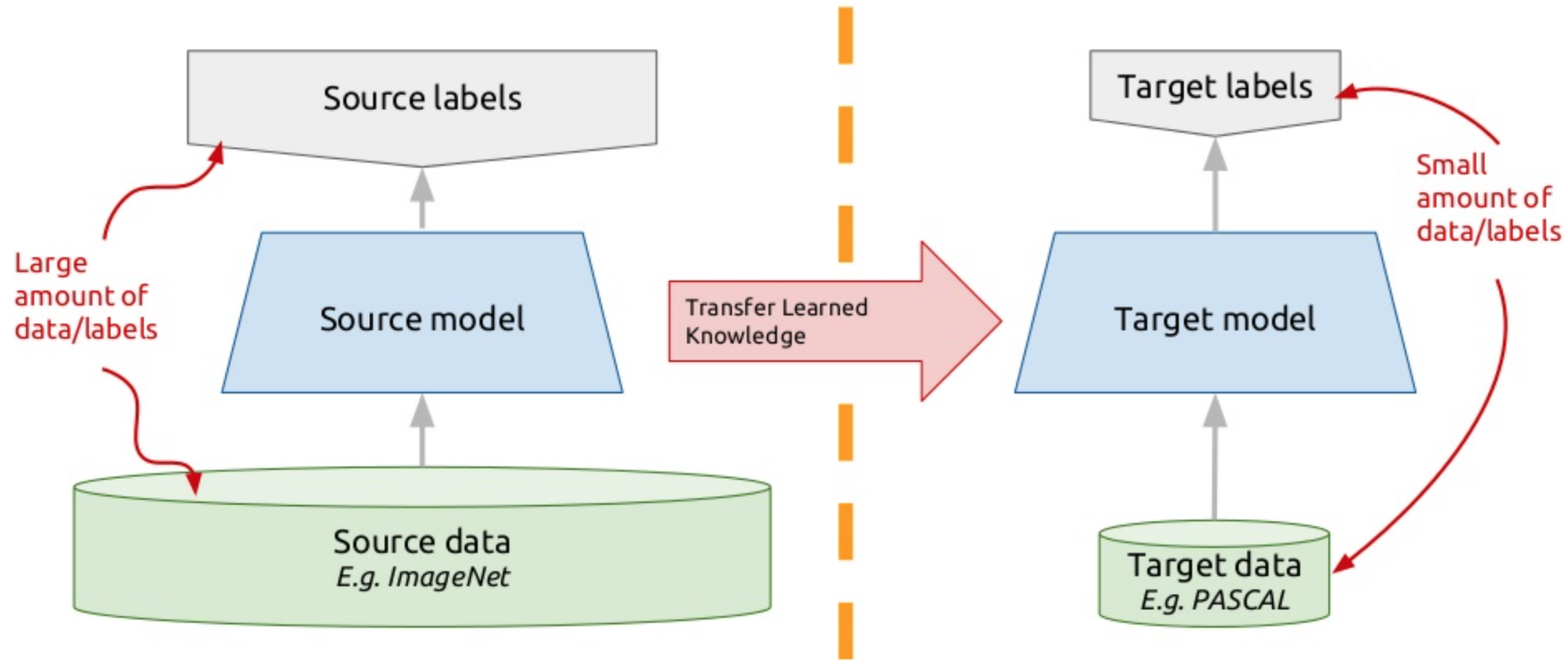
Transfer Learning

Motivazioni:

- Large dataset, tempo di computazione, risorse HW, fine tuning.
- ex. ImageNet è costituito da milioni di immagini
- CPU vs Single GPU vs Multiple GPU



Transfer Learning (2)



Transfer Learning (3)

- Generalizzazione dei modelli
- Complessità dei modelli
- Complessità computazionale
- Sorgenti dati (dati etichettati vs non etichettati)