STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching

Lecture 14

Introduction to CNNs Part 2 -- CNN Architectures

Lecture Overview

1. Padding (control output size in addition to stride)
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

Controlling output size in addition to stride

1. **Padding**
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

Padding

output size

input size

padding pixels per side

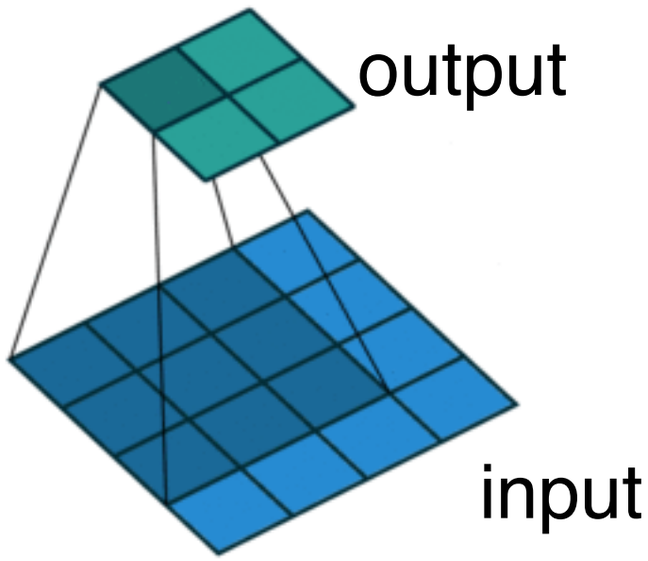
$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1$$

kernel size

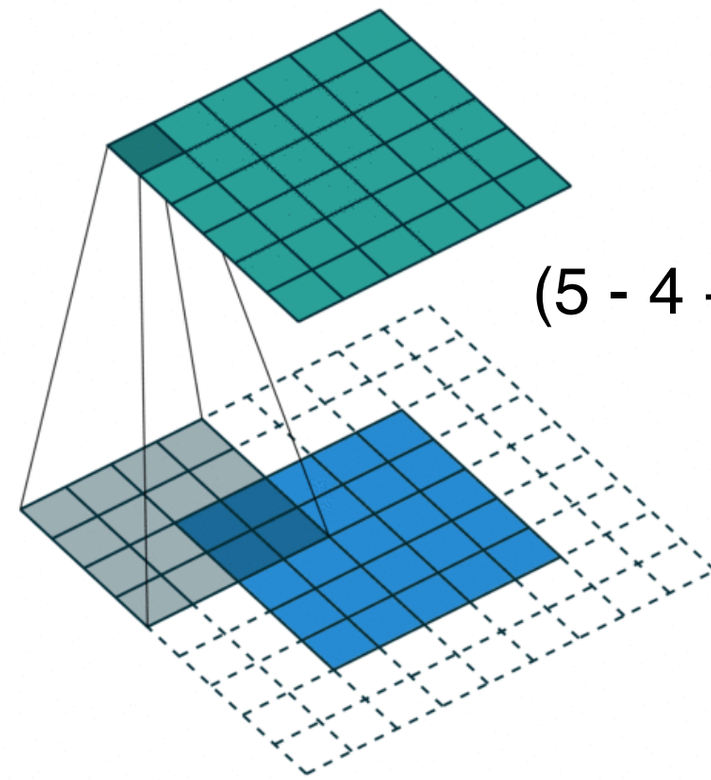
"floor" function

stride

(4 - 3 + 2*0)/1 + 1 = 2

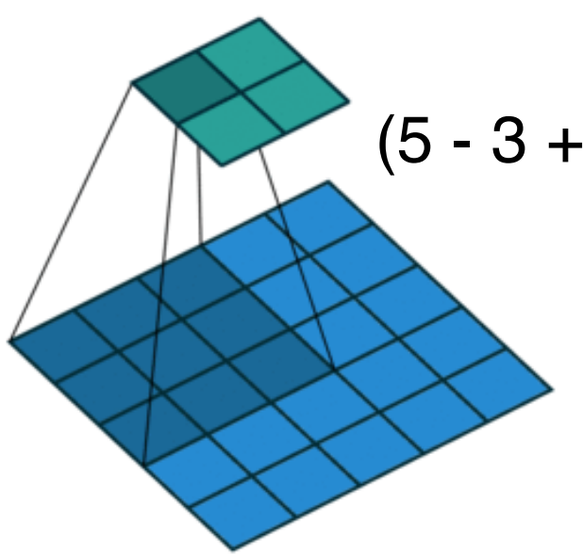


No padding, stride=1



(5 - 4 + 2*2)/1 + 1 = 6

padding=2, stride=1



(5 - 3 + 2*0)/2 + 1 = 2

No padding, stride=2

Highly recommended:

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

Padding Jargon

"valid" convolution: no padding (feature map may shrink)

"same" convolution: padding such that the output size is equal to the input size

Common kernel size conventions:
3x3, 5x5, 7x7 (sometimes 1x1 in later layers to reduce channels)

Padding

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

Assume you want to use a convolutional operation with stride 1 and maintain the input dimensions in the output feature map:

How much padding do you need for "same" convolution?

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

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

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

Padding

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

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

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

Probably explains why common kernel size conventions are 3x3, 5x5, 7x7 (sometimes 1x1 in later layers to reduce channels)

Familiar Concepts Now in 2D

1. Padding
2. **Spatial Dropout and BatchNorm**
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

Spatial Dropout -- Dropout2D

- Problem with regular dropout and CNNs: Adjacent pixels are likely highly correlated (thus, may not help with reducing the "dependency" much as originally intended by dropout)

- Hence, it may be better to drop entire feature maps

Idea comes from

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.

Spatial Dropout -- Dropout2D

- Dropout2d will drop full feature maps (channels)

```
import torch

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

```
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 .

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

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

BatchNorm 2D

BatchNorm1d Inputs are rank-2 tensors: [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 Inputs are rank-4 tensors: [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 .

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

In BatchNorm2d, the mean and standard deviation are computed for N*H*W, i.e., over the channel dimension

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

BatchNorm 2D

In BatchNorm2d, the mean and standard deviation are computed for N*H*W, i.e., over the channel dimension

[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()
```

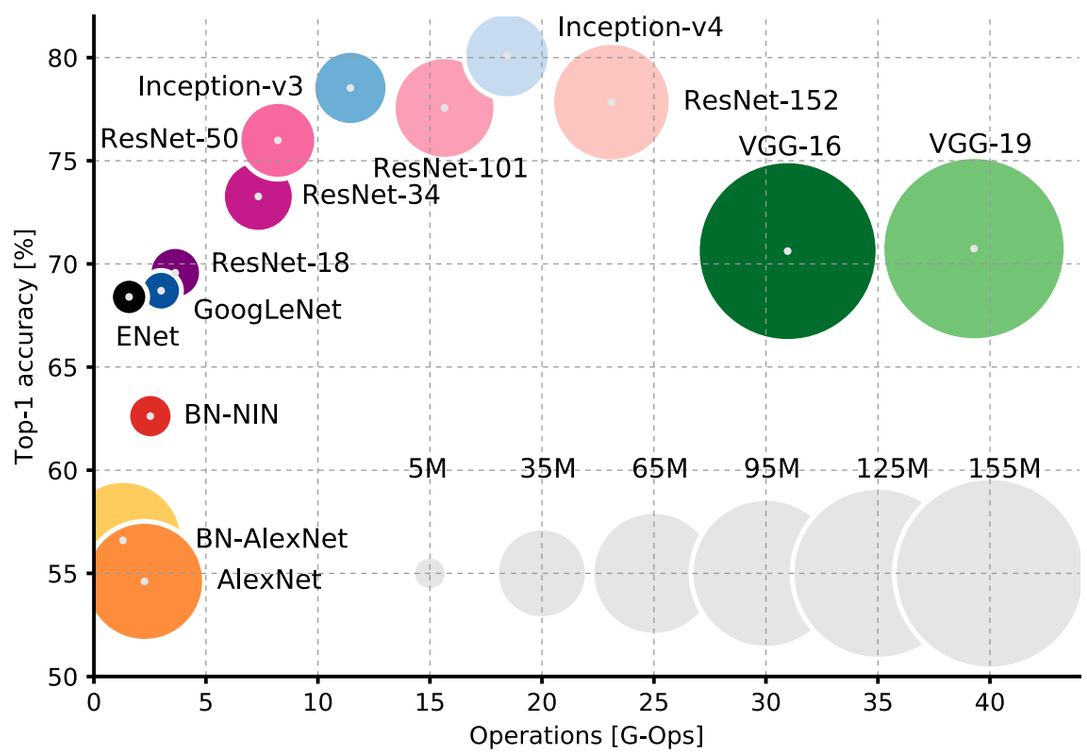
[3]:
```
torch.Size([192])
```

Common Architectures Revisited

1. Padding
2. Spatial Dropout and BatchNorm
3. **Common architectures**
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

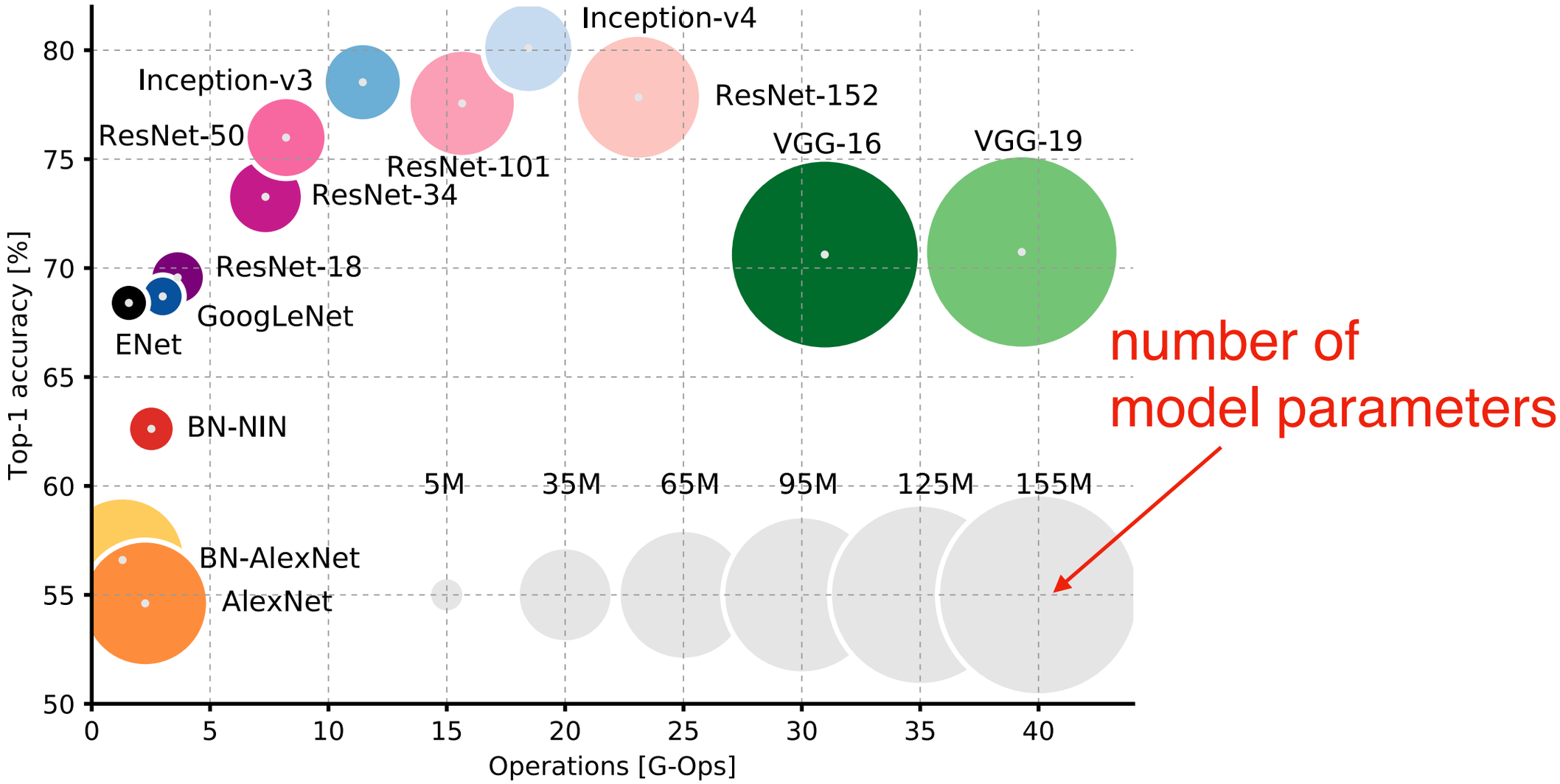
Common Architectures Revisited

We will discuss some additional common CNN architectures since the field evolved quite a bit since 2012 ...



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

Common Architectures Revisited

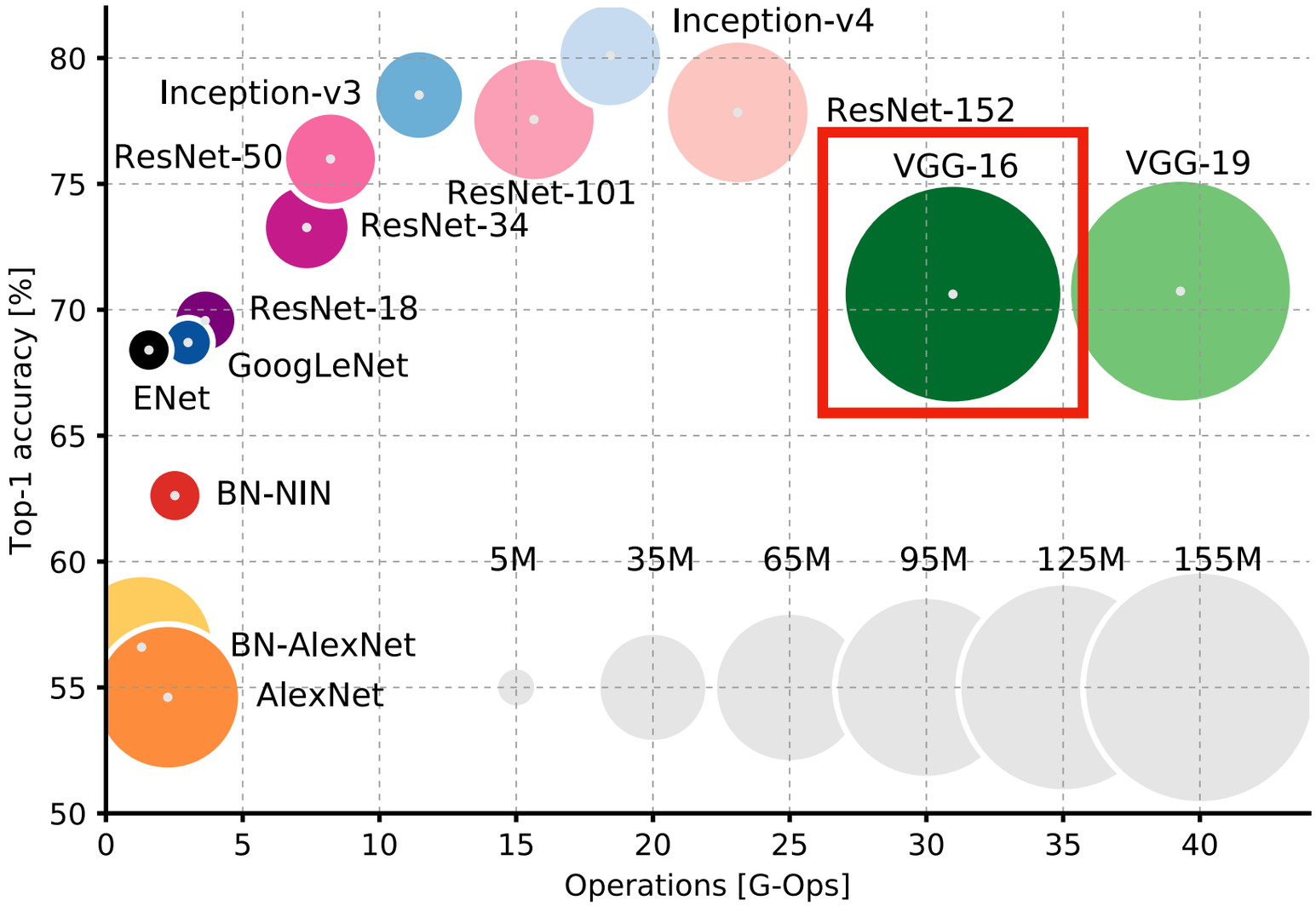


Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

Adding More Layers

1. Padding
2. Spatial Dropout and BatchNorm
3. Common architectures
   - **3.1 VGG16 (simple, deep CNN)**
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

Common Architectures Revisited



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

VGG-16

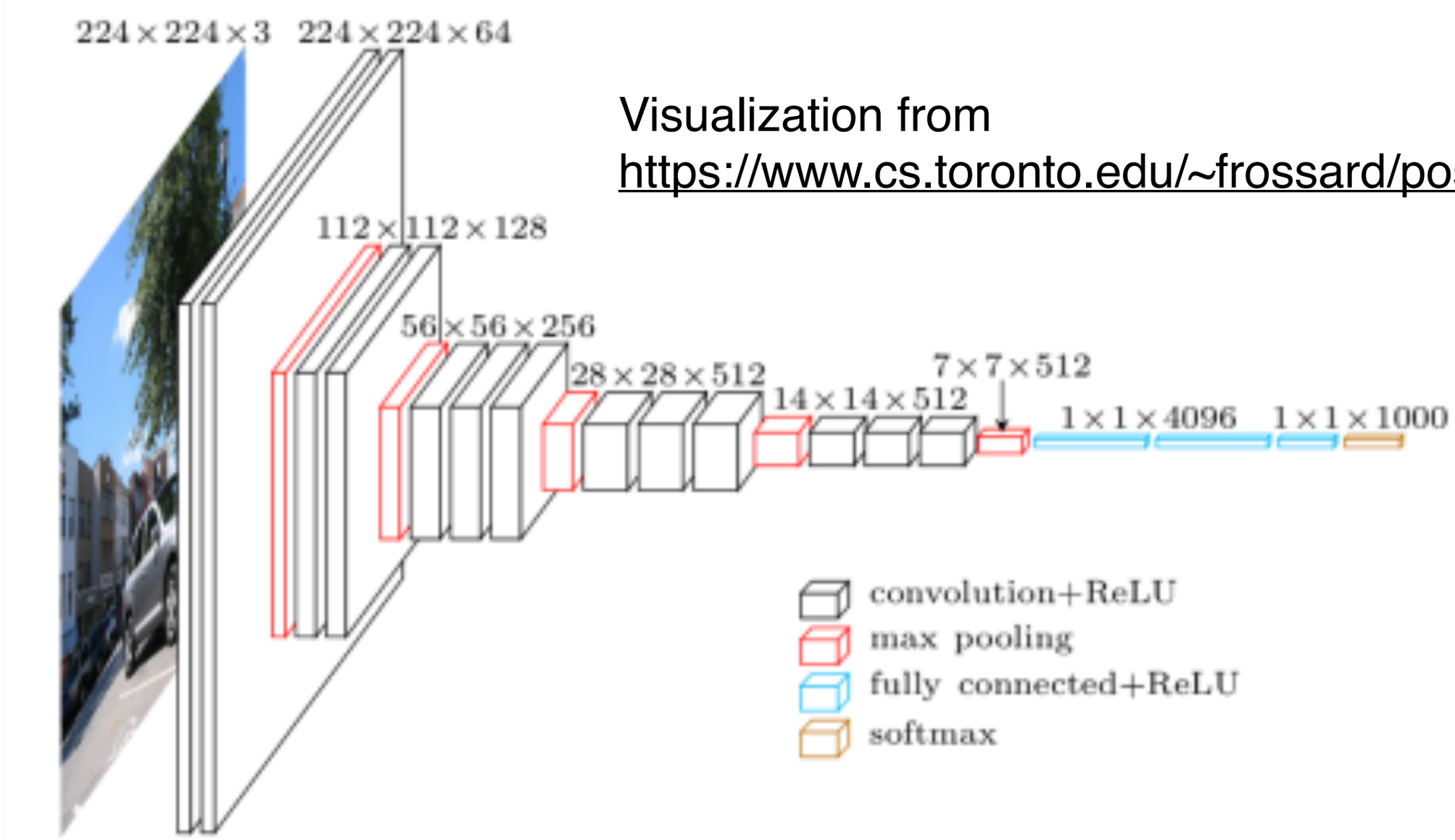
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	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					

<u>Advantages:</u>
very simple architecture,
3x3 convs, stride=1,
"same" padding, 2x2 max pooling

<u>Disadvantage:</u>
very large number of parameters and slow
(see previous slide)

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

VGG-16



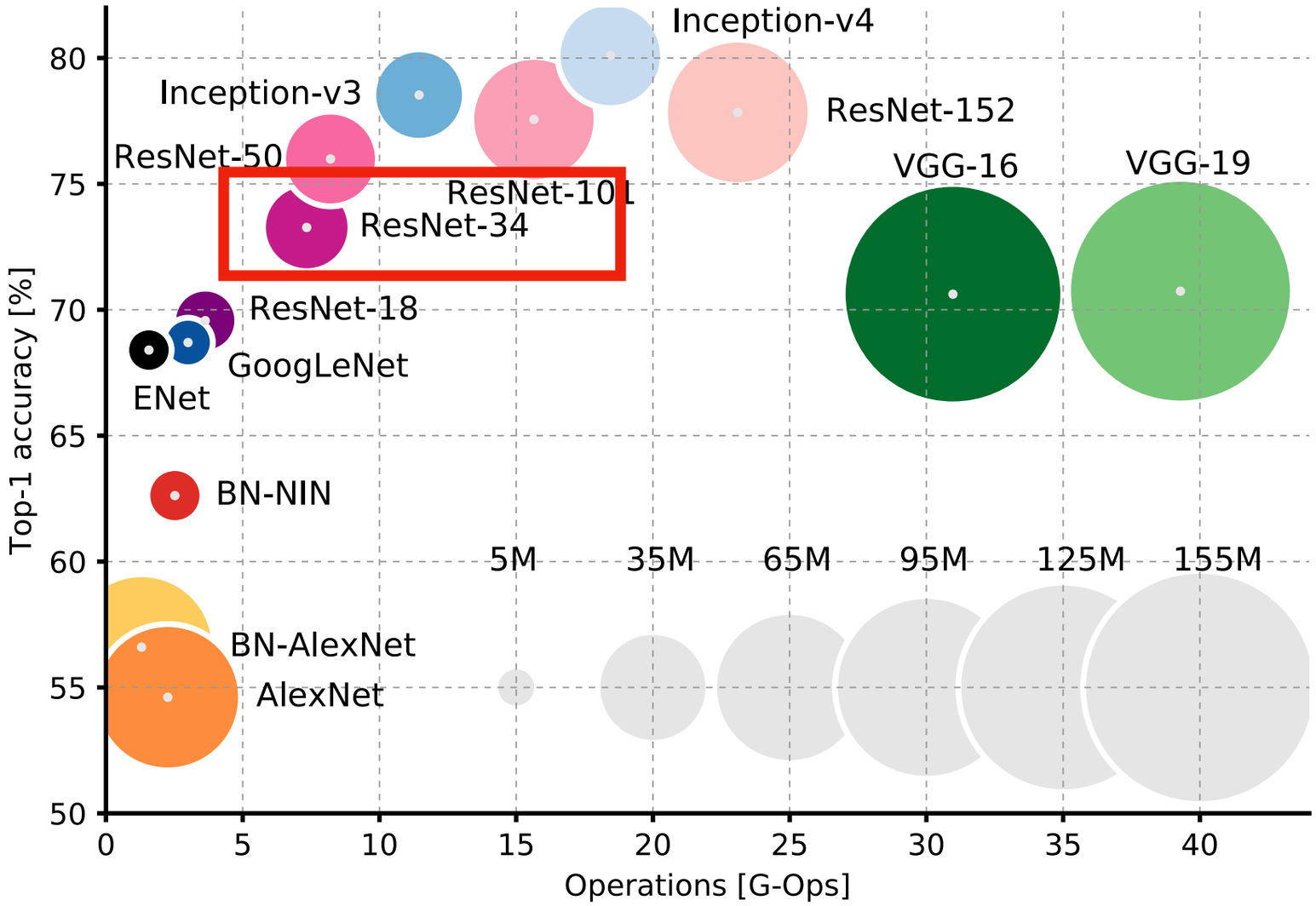
Visualization from
https://www.cs.toronto.edu/~frossard/post/vgg16/

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

Can We Add More Layers? CNNs with a Simple Trick

1. Padding
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - **3.2 ResNet and skip connections**
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. Transfer learning

Common Architectures Revisited



Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *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.

With their simple trick of allowing skip connections (the possibility to learn identity functions and skip layers that are not useful), ResNets allow us to to implement very, very deep architectures

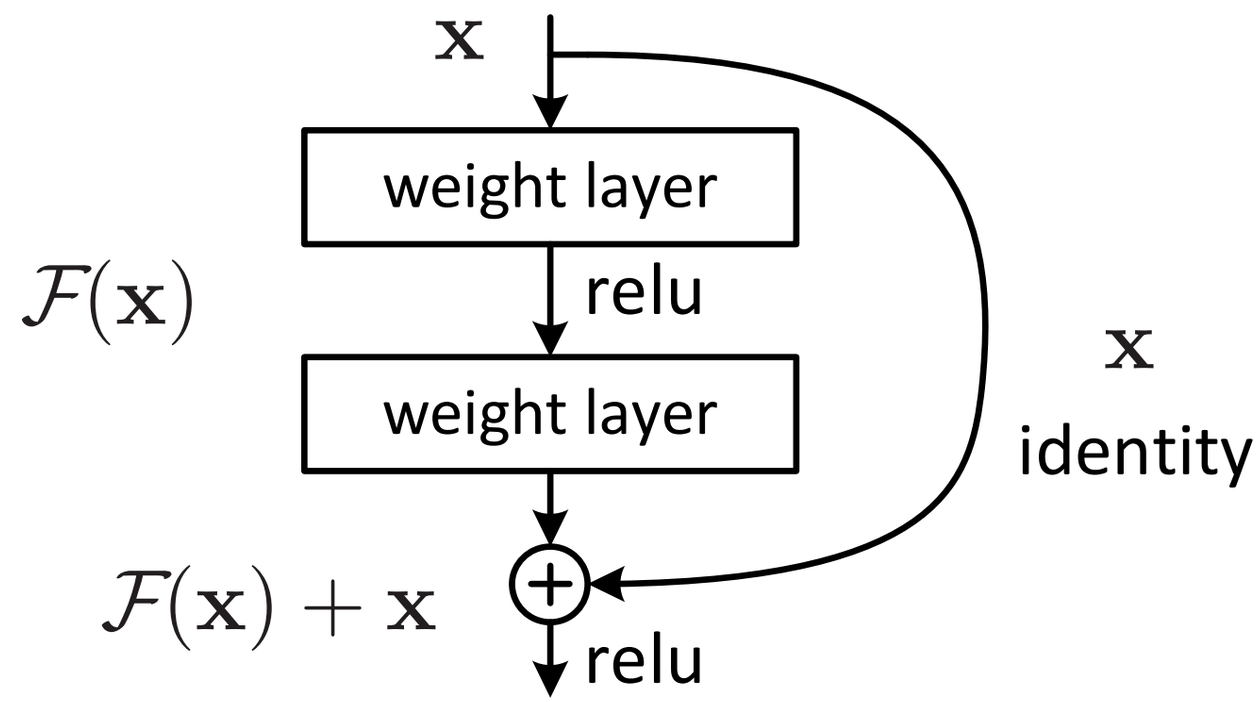
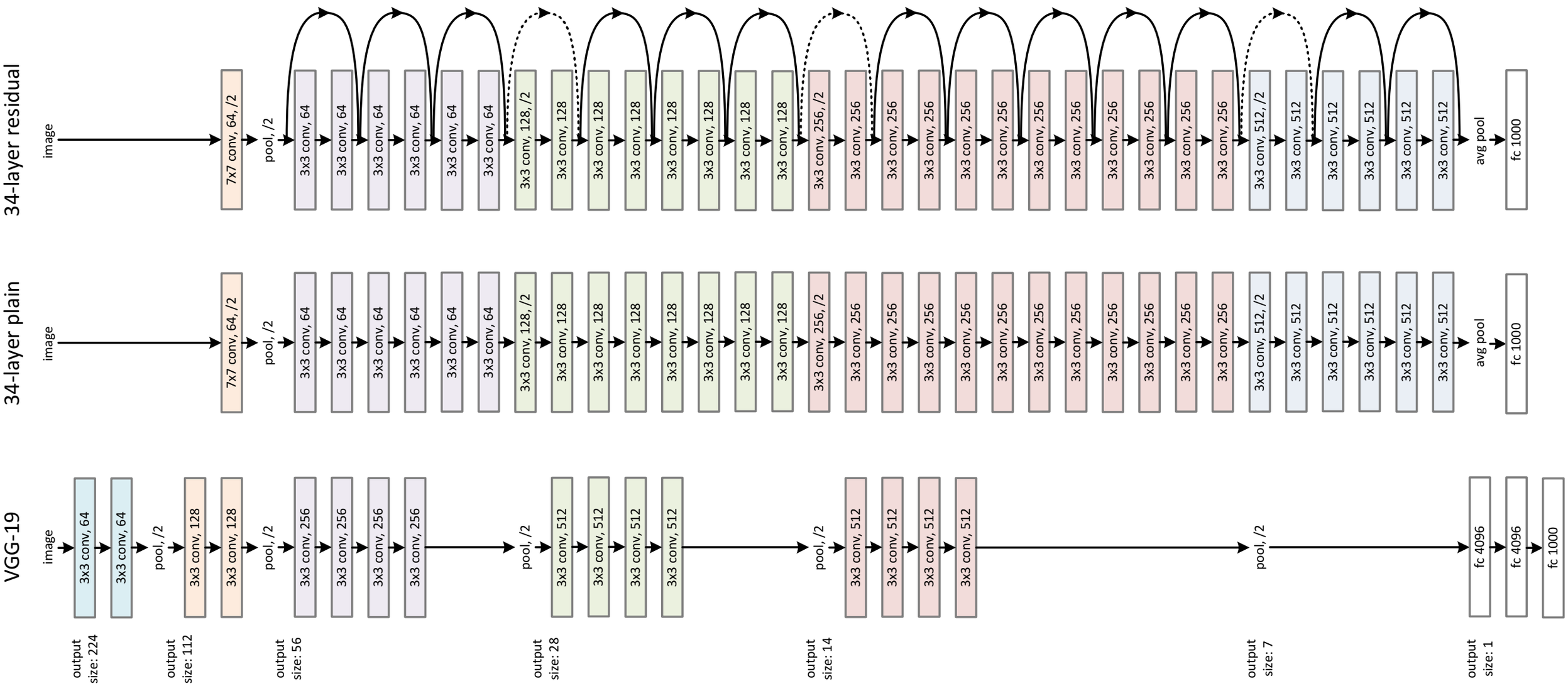


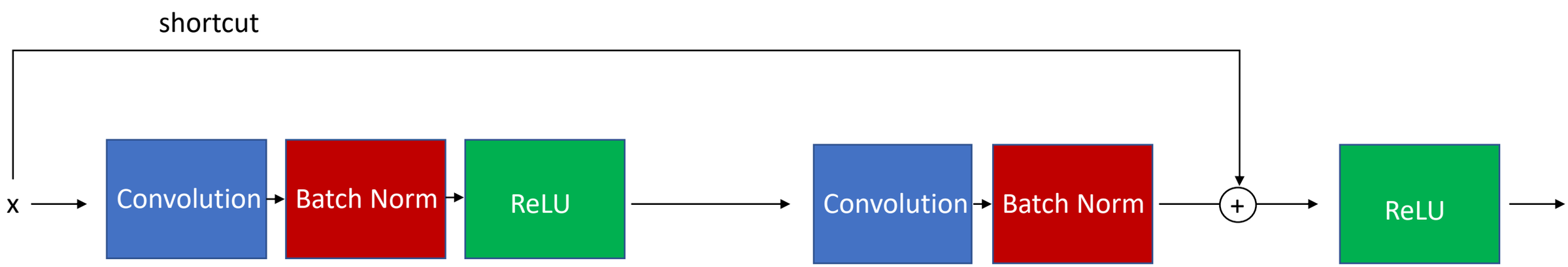
Figure 2. Residual learning: a building block.

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.

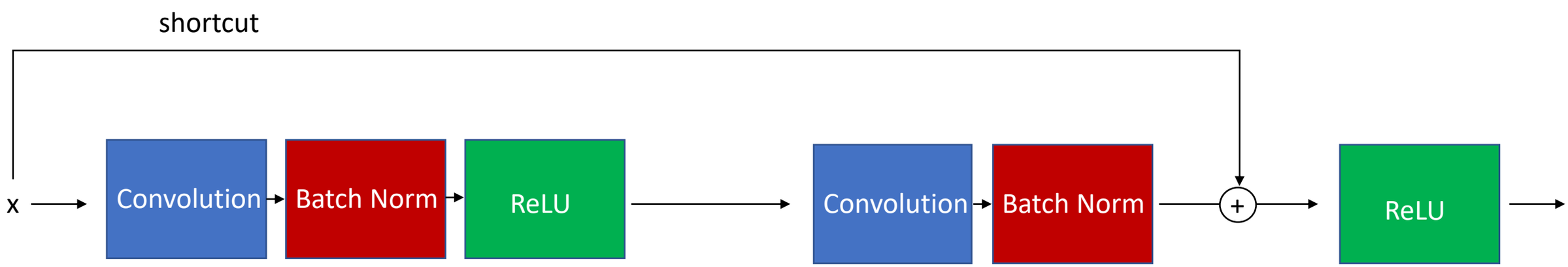


ResNets



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

ResNets



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

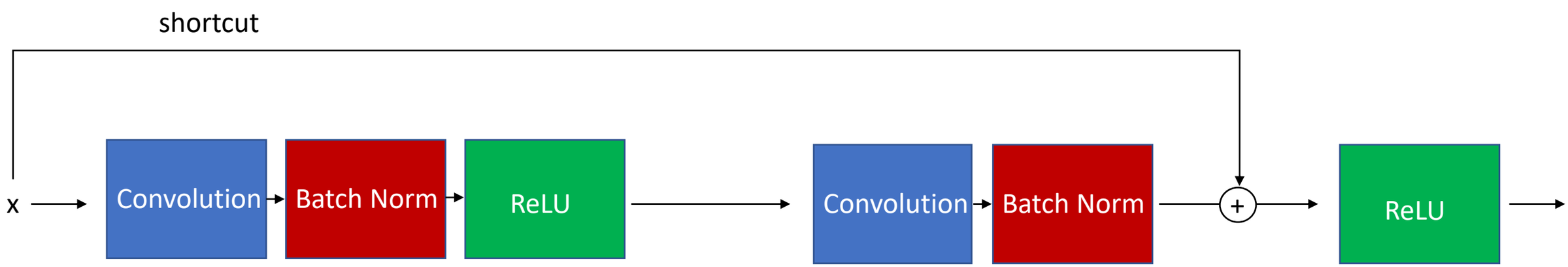
$$= \sigma\left(a^{(l+1)}W^{(l+2)} + b^{(l+2)} + a^{(l)}\right)$$

If all weights and the bias are zero, then

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

due to ReLU

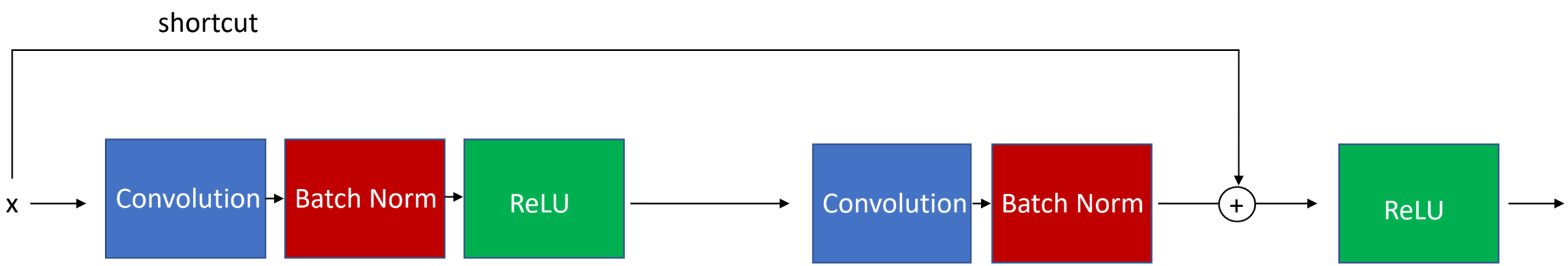
ResNets

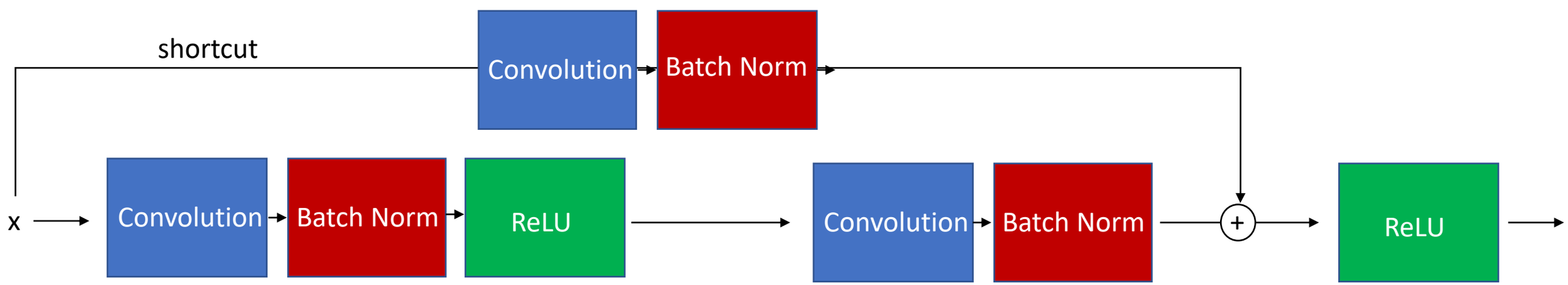


$$a^{(l+2)} = \sigma\left(z^{(l+2)} + a^{(l)}\right)$$

We assume these have the same dimension (e.g., via "same" convolution)

ResNets





alternative residual blocks with skip connections such that the input passed via the shortcut is resized to dimensions of the main path's output

Simplifying CNNs

1. Padding
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. **Replacing max-pooling with convolutional layers**
5. Convolutional instead of fully connected layers
6. Transfer learning

"All-Convolutional Network"

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

Key Idea: Replace Maxpooling by strided convolutions (i.e., conv layers with stride=2)

We can think of "strided convolutions" as learnable pooling

Global Average Pooling in Last Layer

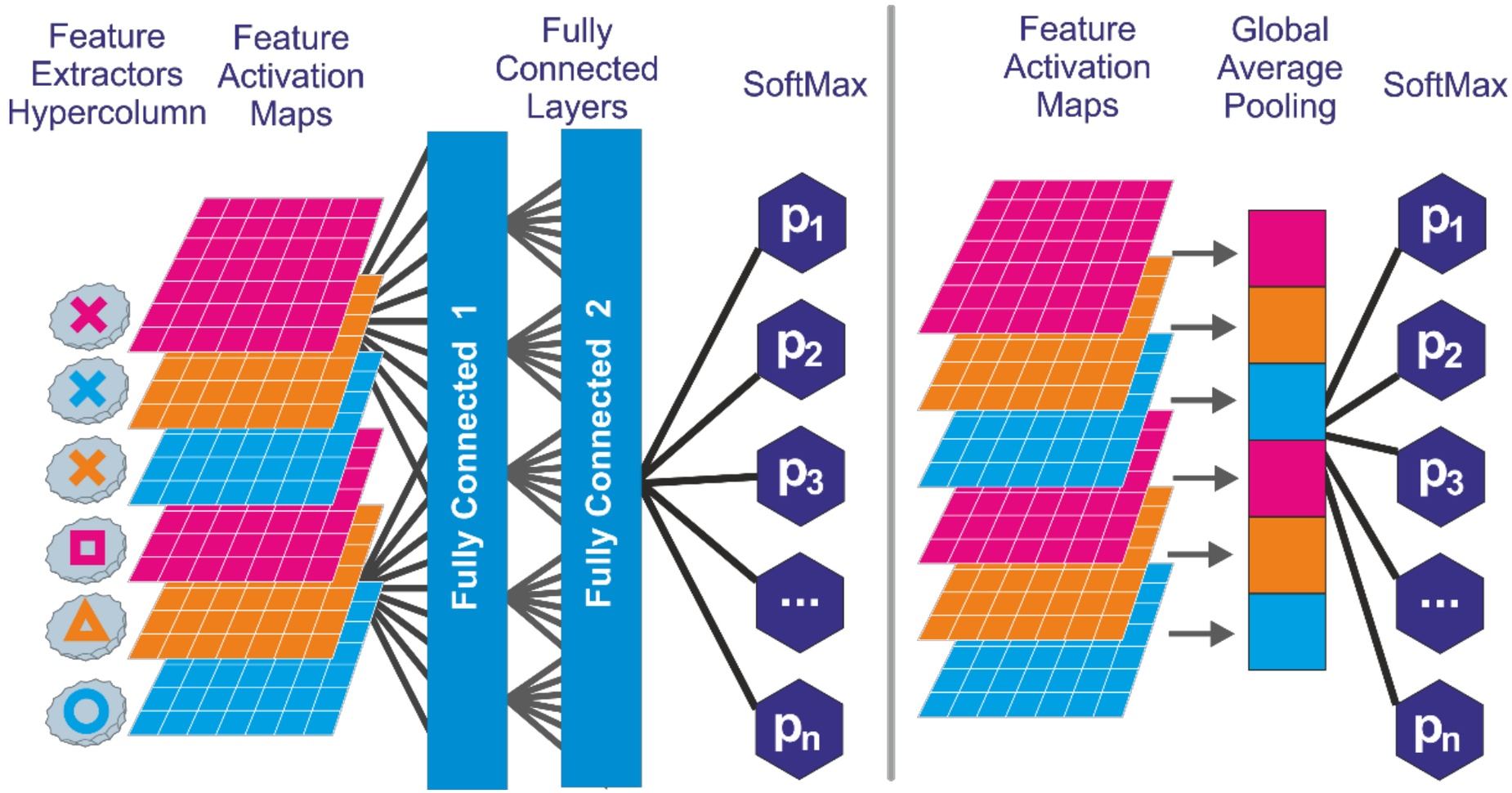


Figure 16: Global average pooling layer replacing the fully connected layers. The output layer implements a Softmax operation with $p_1, p_2, \ \dots \ , p_n$ the predicted probabilities for each class.

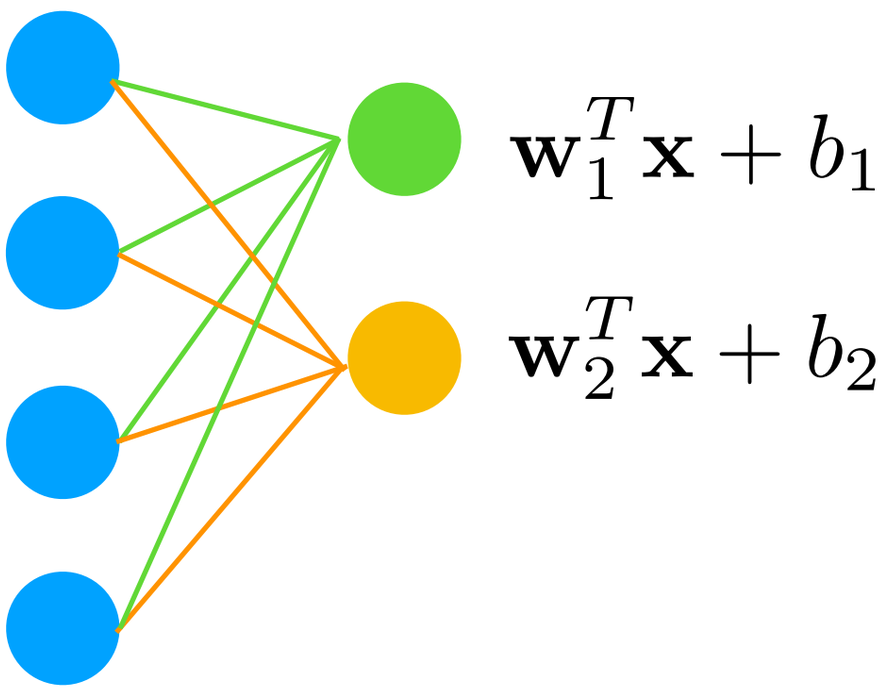
Figure Source: Singh, Anshuman Vikram. "Content-based image retrieval using deep learning." (2015).

Code Example

Simplifying CNNs Part 2

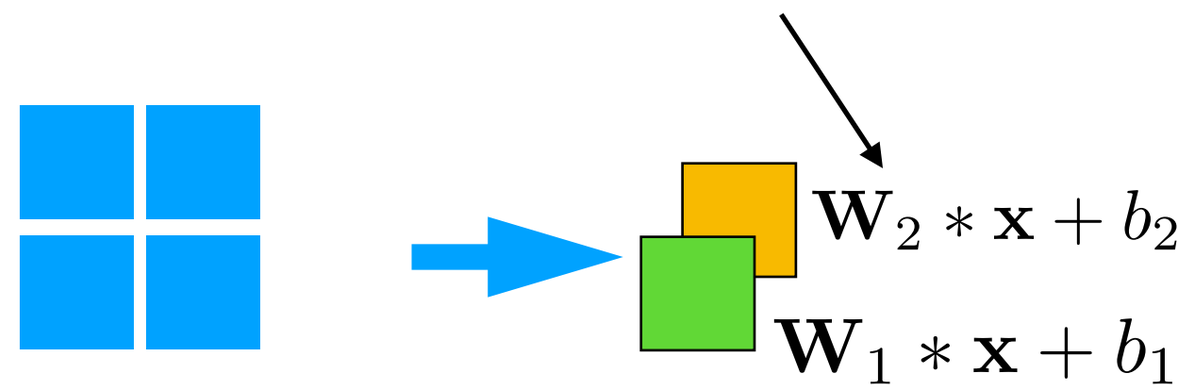
1. Padding
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. **Convolutional instead of fully connected layers**
6. Transfer learning

It is Possible to Replace Fully Connected Layers by Convolutional Layers



Fully connected layer

remember, these also involve dot products between the receptive fields and kernels



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

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

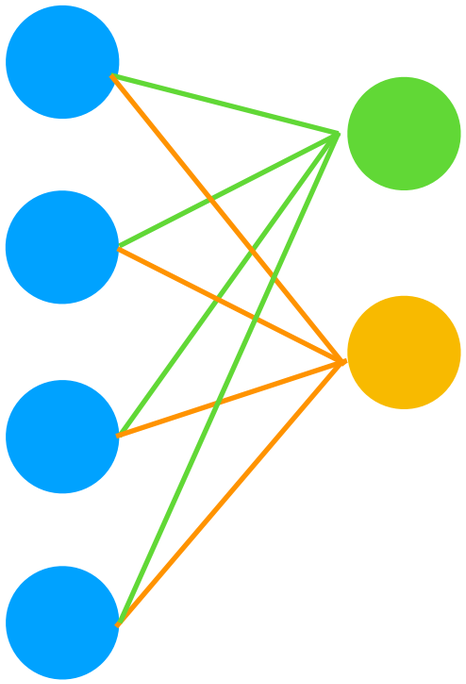
```python
import torch
```

Assume we have a 2x2 input image:

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

inputs.shape
```

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

NCHW



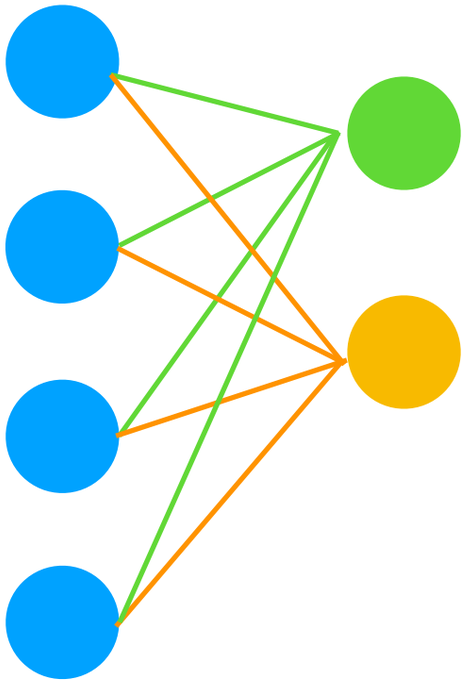
$$\mathbf{w}_1^T\mathbf{x} + b_1$$

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

```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
```

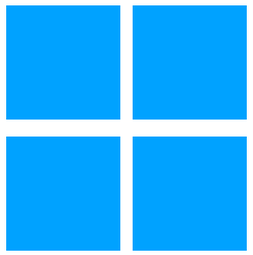
```python
torch.relu(fc(inputs.view(-1, 4)))
```

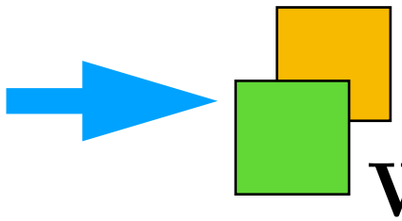
```
tensor([[14.9000, 19.0000]], grad_fn=<ReluBackward0>)
```



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

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





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

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

```
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())
```

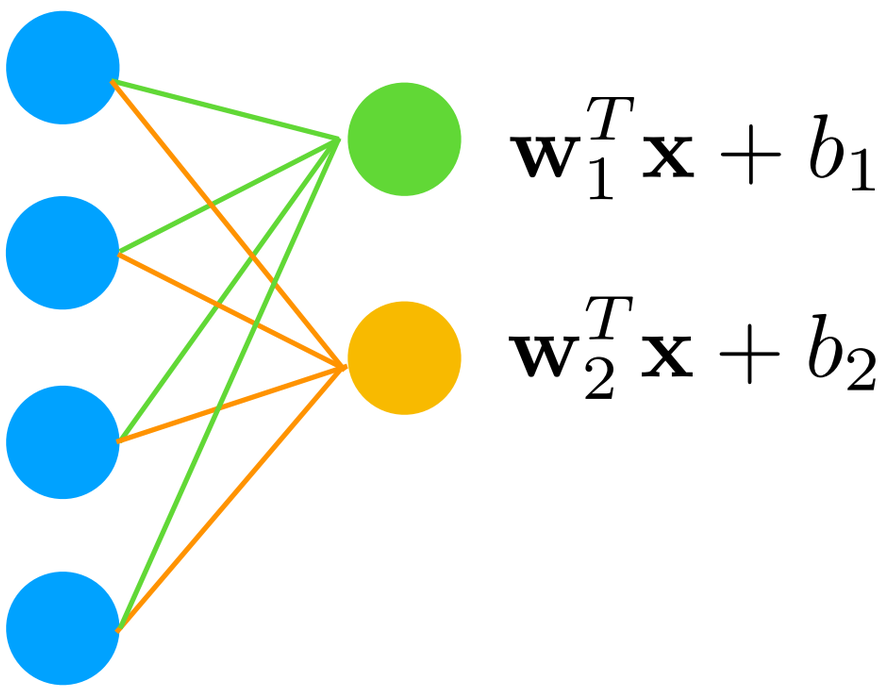
```
torch.Size([2, 1, 2, 2])
torch.Size([2])
```

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

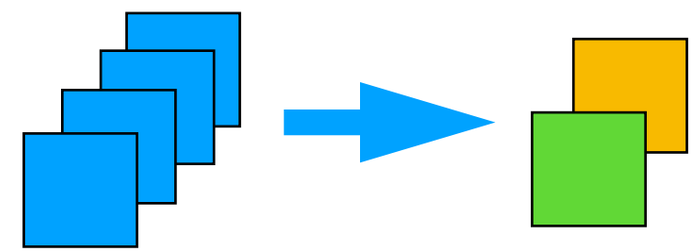
```
torch.relu(conv(inputs))
```

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

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

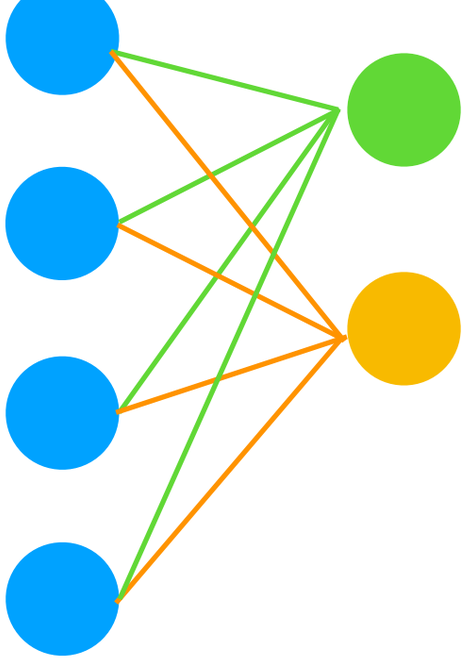
It is Possible to Replace Fully Connected Layers by Convolutional Layers



Fully connected layer

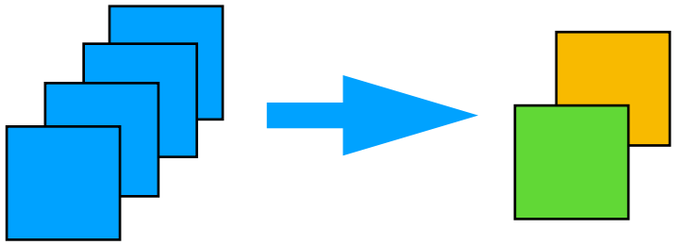


Or, we can concatenate the inputs into 1x1 images with 4 channels and then use 2 kernels
(remember, each kernel then also has 4 channels)



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

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



```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])
```

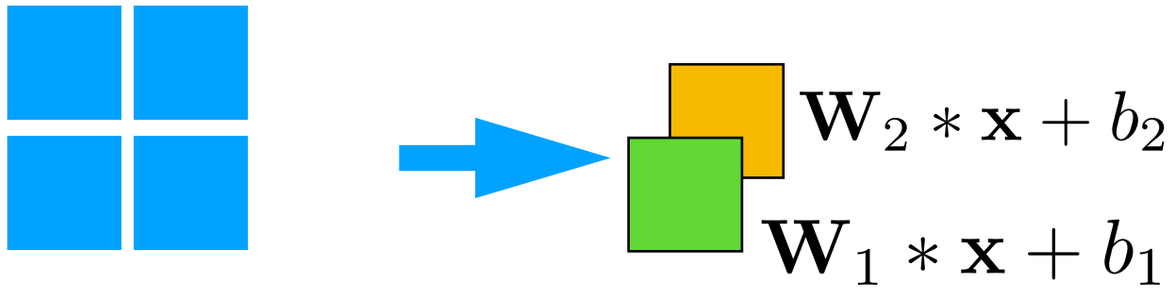
```python
fc = torch.nn.Linear(4, 2)

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)))
```

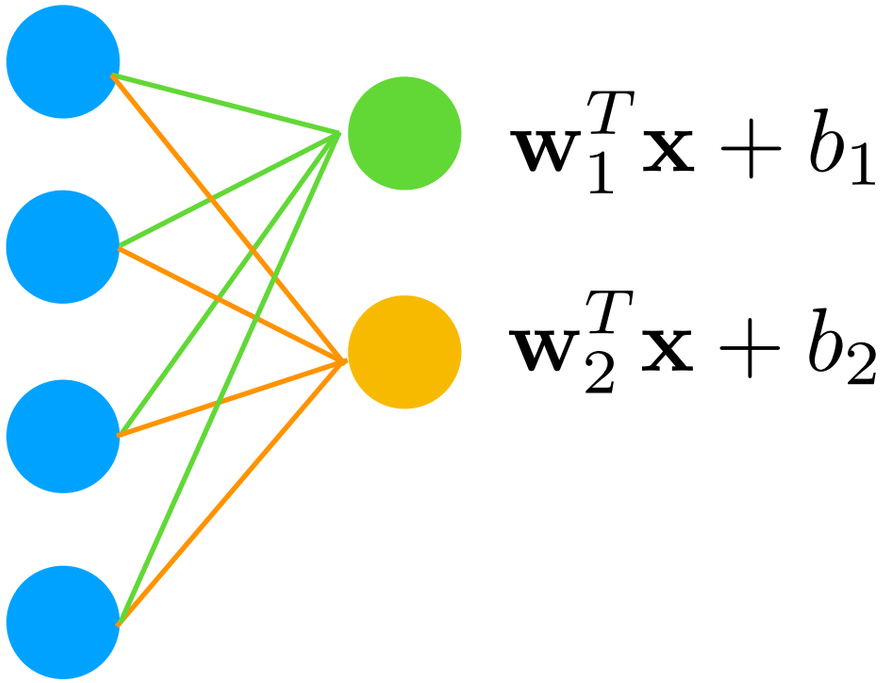
```
tensor([[14.9000, 19.0000]], grad_fn=<ReluBackward0>)
```

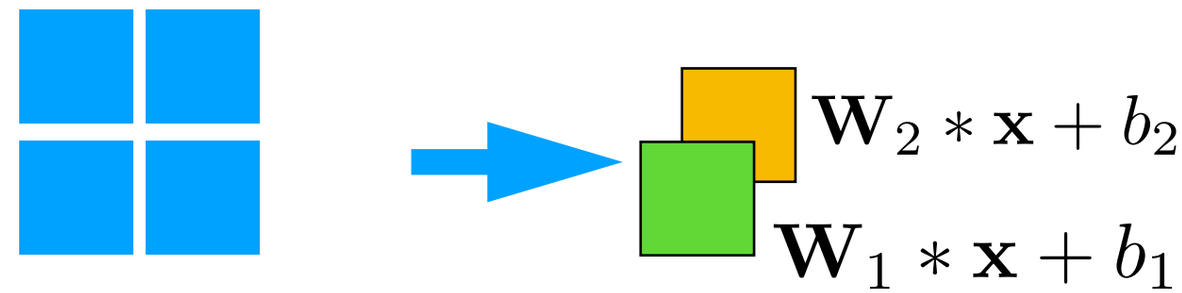
```python
conv = torch.nn.Conv2d(in_channels=4,
                       out_channels=2,
                       kernel_size=(1, 1))

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()
```

Can You Teach an Old Dog New Tricks?

1. Padding
2. Spatial Dropout and BatchNorm
3. Common architectures
   - 3.1 VGG16 (simple, deep CNN)
   - 3.2 ResNet and skip connections
4. Replacing Max-Pooling with convolutional layers
5. Convolutional instead of fully connected layers
6. **Transfer learning**

Transfer Learning

- A technique that may be useful for your class projects
- Key idea:
  - ✦ Feature extraction layers may be generally useful
  - ✦ Use a pre-trained model (e.g., pre-trained on ImageNet)
  - ✦ Freeze the weights: Only train last layer (or last few layers)
- Related approach: Fine-tuning, train a pre-trained network on your smaller dataset

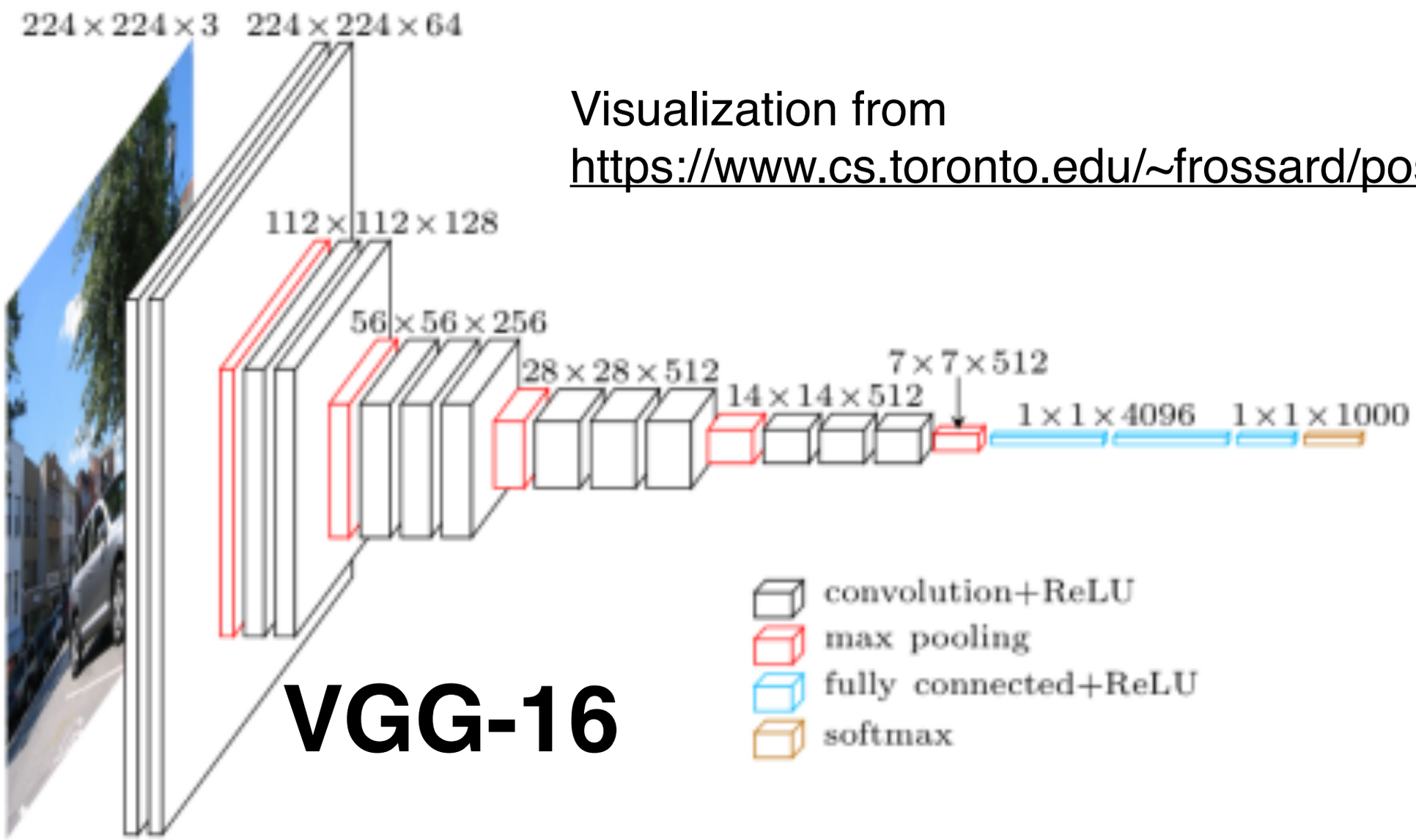
Which Layers to Replace & Train?

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).

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.

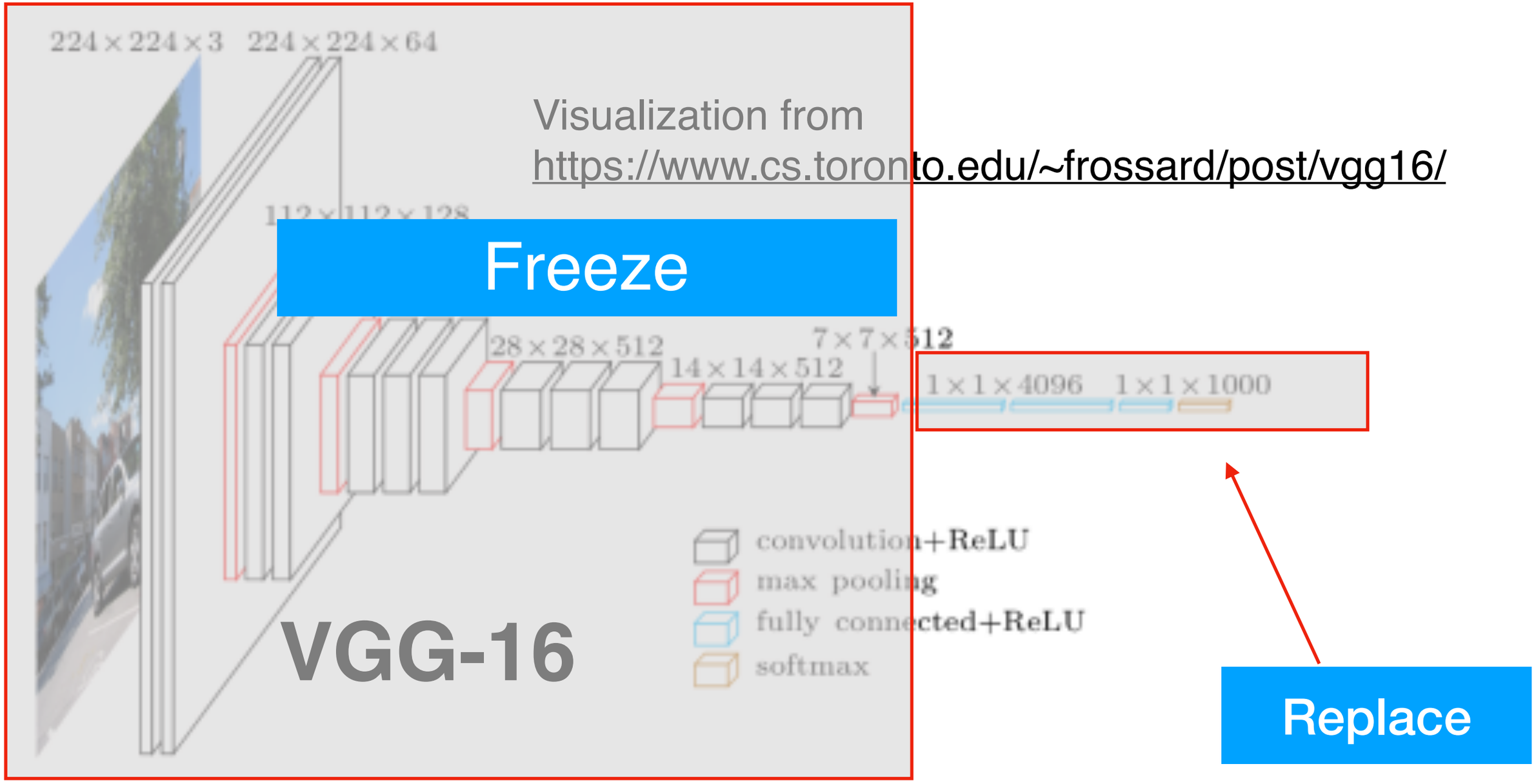
Transfer Learning



Visualization from
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).

Transfer Learning



Visualization from 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).

Transfer Learning

https://pytorch.org/vision/stable/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

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])
```

Transfer Learning Code Example