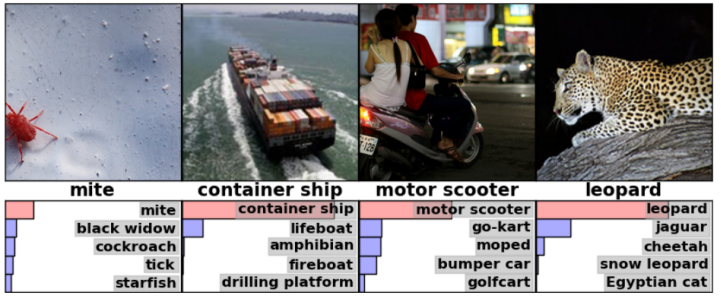
Introduction to Deep Learning

Copernicus Master on Digital Earth

Lecture 6: Convolutional Neural Network (CNN)

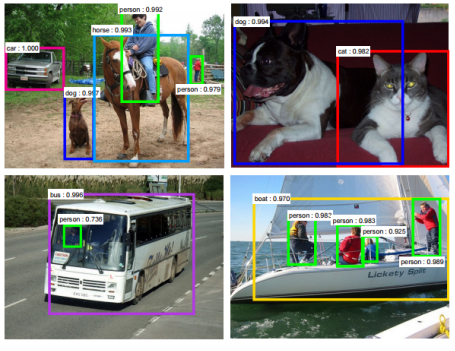
Dr. Charlotte Pelletier
charlotte.pelletier@univ-ubs.fr

Used everywhere for Vision

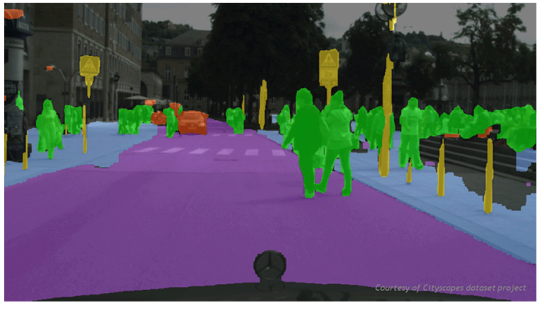


[Krizhevsky 2012]

[Ciresan et al. 2013]



[Faster R-CNN - Ren 2015]



[NVIDIA dev blog]

Many other applications

Speech recognition & speech synthesis

Many other applications

Speech recognition & speech synthesis

Natural Language Processing

Many other applications

Speech recognition & speech synthesis

Natural Language Processing

Protein/DNA binding prediction

Many other applications

Speech recognition & speech synthesis

Natural Language Processing

Protein/DNA binding prediction

Any problem with a spatial (or sequential) structure

Many other applications

Speech recognition & speech synthesis

Natural Language Processing

Protein/DNA binding prediction

Any problem with a spatial (or sequential) structure

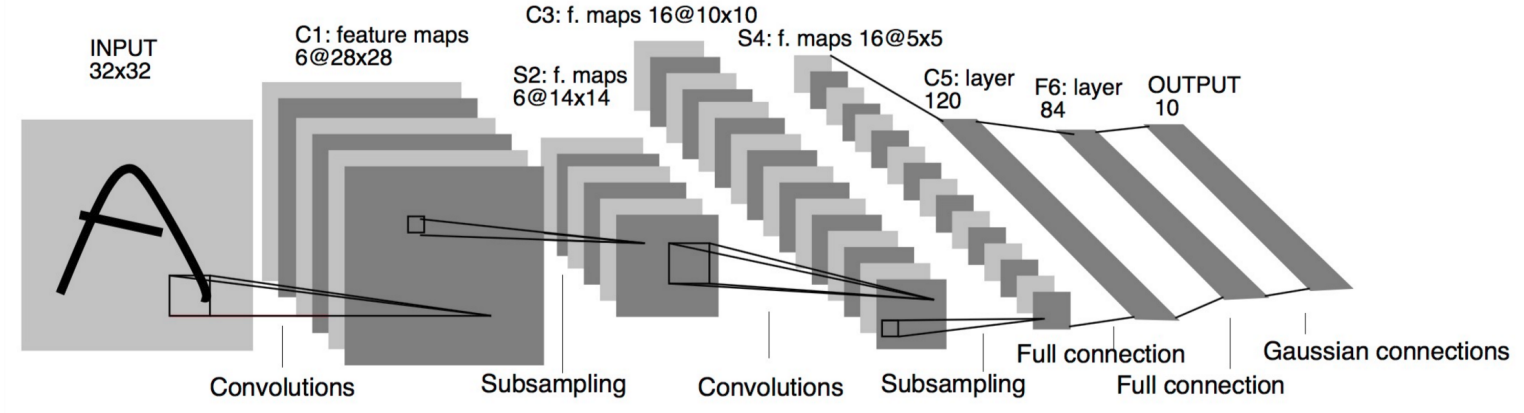
And thus remote sensing data!

ConvNets for image classification

CNN = Convolutional Neural Networks = ConvNet

ConvNets for image classification

CNN = Convolutional Neural Networks = ConvNet



LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition.

Content

Convolutions

Content

Convolutions

CNNs for Image Classification

Content

Convolutions

CNNs for Image Classification

CNN architectures

Convolutions

Motivations

Standard dense layer (aka. fully-connected layer) for an image input:

```
X = torch.rand(28,28,4) # image sat-6
X = X.flatten()
# shape of X is 3136
Z = nn.Linear(X.shape[0], 128)(X)
```

How many trainable parameters in the dense layer?

Motivations

Standard dense layer (aka. fully-connected layer) for an image input:

```
X = torch.rand(28,28,4) # image sat-6
X = X.flatten()
# shape of X is 3136
Z = nn.Linear(X.shape[0], 128)(X)
```

How many trainable parameters in the dense layer?

$28 \times 28 \times 4 \times 128 + 128 = 401,536!$

Motivations

Standard dense layer (aka. fully-connected layer) for an image input:

```
X = torch.rand(28,28,4) # image sat-6
X = X.flatten()
# shape of X is 3136
Z = nn.Linear(X.shape[0], 128)(X)
```

How many trainable parameters in the dense layer?

$$28 \times 28 \times 4 \times 128 + 128 = 401,536!$$

Spatial organization of the input is destroyed by `flatten()`

Motivations

Standard dense layer (aka. fully-connected layer) for an image input:

```
X = torch.rand(28,28,4) # image sat-6
X = X.flatten()
# shape of X is 3136
Z = nn.Linear(X.shape[0], 128)(X)
```

How many trainable parameters in the dense layer?

$$28 \times 28 \times 4 \times 128 + 128 = 401,536!$$

Spatial organization of the input is destroyed by `flatten()`

We never use dense layers directly on large images. The most standard solution is convolution layers.

Multi Layer Perceptron

```
class Sat6MLP(nn.Module):
    def __init__(self, in_size, hidden_units, out_size):
        super(Sat6MLP, self).__init__()

        # Let us now define the linear layers we need:
        self.fc1 = nn.Linear(in_size, hidden_units)
        self.fc2 = nn.Linear(hidden_units, hidden_units)
        self.fc3 = nn.Linear(hidden_units, out_size)

    # We have also to define what is the forward of this module:
    def forward(self, x):
        h1 = nn.functional.relu(self.fc1(x))
        h2 = nn.functional.relu(self.fc2(h1))
        logits = self.fc3(h2)
        return logits
```

Convolutional Neural Network

```
class Sat6ConvNet(nn.Module):
    def __init__(self):
        super(Sat6ConvNet, self).__init__()

        # Let us now define the linear layers we need:
        self.conv1 = nn.Conv2d(1, 32, 5)
        self.conv2 = nn.Conv2d(32, 16, 5)

        self.fc1 = nn.Linear(16*5*5, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 4)

    # We have also to define what is the forward of this module:
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # flatten all dimensions except the batch dimension
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits = self.fc3(x)
        return logits
```

Convolutional Neural Network

```
class Sat6ConvNet(nn.Module):
    def __init__(self):
        super(Sat6ConvNet, self).__init__()

        # Let us now define the linear layers we need:
        self.conv1 = nn.Conv2d(1, 32, 5)
        self.conv2 = nn.Conv2d(32, 16, 5)

        self.fc1 = nn.Linear(16*5*5, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 4)

    # We have also to define what is the forward of this module:
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # flatten all dimensions except the batch dimension
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits = self.fc3(x)
        return logits
```

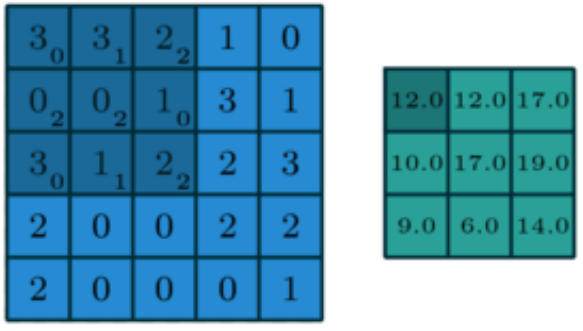
2D spatial organization of features preserved untill `Flatten`.

Example

```
net = Net()
print(net)
input = torch.randn(16, 1, 32, 32) # B x C x H x W
out = net(input)
print(out)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=4, bias=True)
)
torch.Size([16, 4])
```

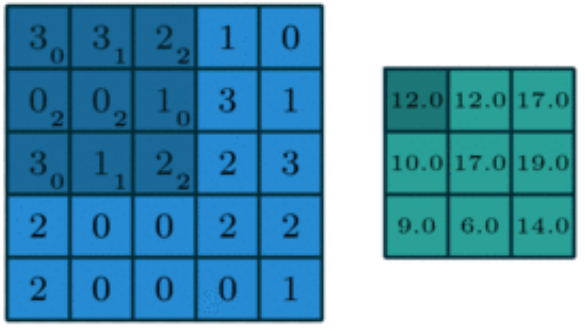
Convolution in a neural network



- x is a 3×3 chunk (dark area) of the image (blue array)
- Each output neuron is parametrized with the 3×3 weight matrix $\mathbf{w}$ (small numbers)

These slides extensively use convolution visualisation by V. Dumoulin available at https://github.com/vdumoulin/conv_arithmetic

Convolution in a neural network



- x is a 3×3 chunk (dark area) of the image (blue array)
- Each output neuron is parametrized with the 3×3 weight matrix $\mathbf{w}$ (small numbers)

The activation obtained by sliding the 3×3 window and computing:

$$y = relu(\mathbf{w}^T x + b)$$

Motivations

Local connectivity

- A neuron depends only on a few local input neurons
- Translation invariance

Motivations

Local connectivity

- A neuron depends only on a few local input neurons
- Translation invariance

Comparison to a fully-connected layer

- Parameter sharing: reduce overfitting
- Make use of spatial structure: strong prior for vision!

Analogy with the animal vision: Hubel & Wiesel, Receptive fields of single neurones in the cat's striate cortex (1959)

Why convolution?

Discrete convolution (actually cross-correlation) between two functions f and g:

$$(f \star g)(x) = \sum_{a+b=x} f(a).g(b) = \sum_{a} f(a).g(x+a)$$

Why convolution?

Discrete convolution (actually cross-correlation) between two functions f and g:

$$(f \star g)(x) = \sum_{a+b=x} f(a).g(b) = \sum_{a} f(a).g(x+a)$$

2D-convolutions (actually 2D cross-correlation):

$$(f \star g)(x,y) = \sum_{n} \sum_{m} f(n,m).g(x+n,y+m)$$

Why convolution?

Discrete convolution (actually cross-correlation) between two functions f and g:

$$(f \star g)(x) = \sum_{a+b=x} f(a).g(b) = \sum_{a} f(a).g(x+a)$$

2D-convolutions (actually 2D cross-correlation):

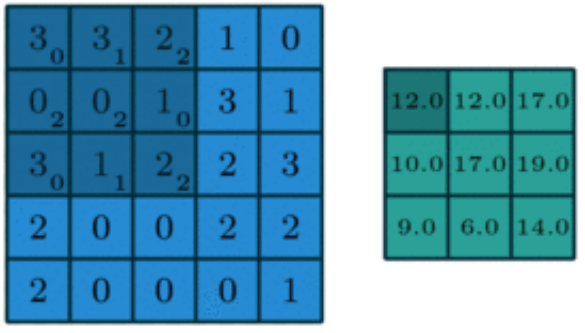
$$(f \star g)(x,y) = \sum_{n} \sum_{m} f(n,m).g(x+n, y+m)$$

f is a convolution kernel or filter applied to the 2-d map g (our image)

Example: convolution image

- Image: im of dimensions 5×5
- Kernel: k of dimensions 3×3

$$(k \star im)(x, y) = \sum_{n=0}^{2} \sum_{m=0}^{2} k(n, m).im(x + n - 1, y + m - 1)$$



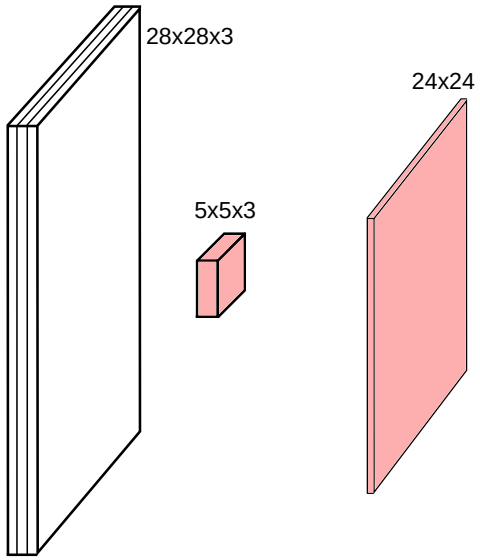
Channels

Colored image = tensor of shape `(channels, height, width)`

Channels

Colored image = tensor of shape `(channels, height, width)`

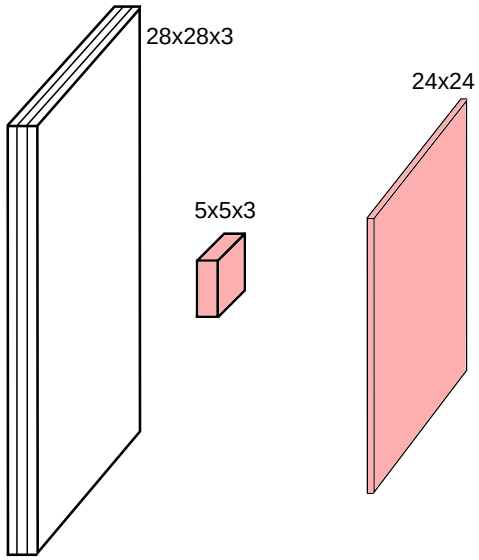
Convolutions are usually computed for each channel and summed:



Channels

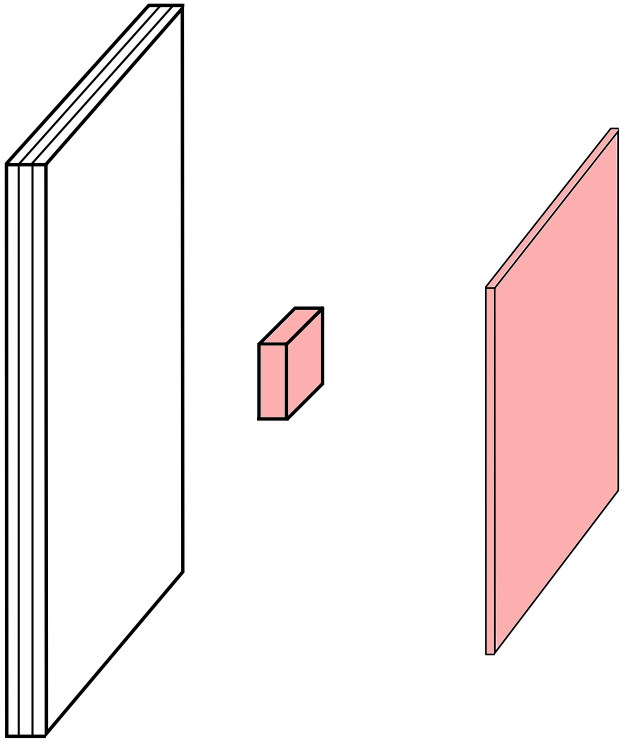
Colored image = tensor of shape `(channels, height, width)`

Convolutions are usually computed for each channel and summed:

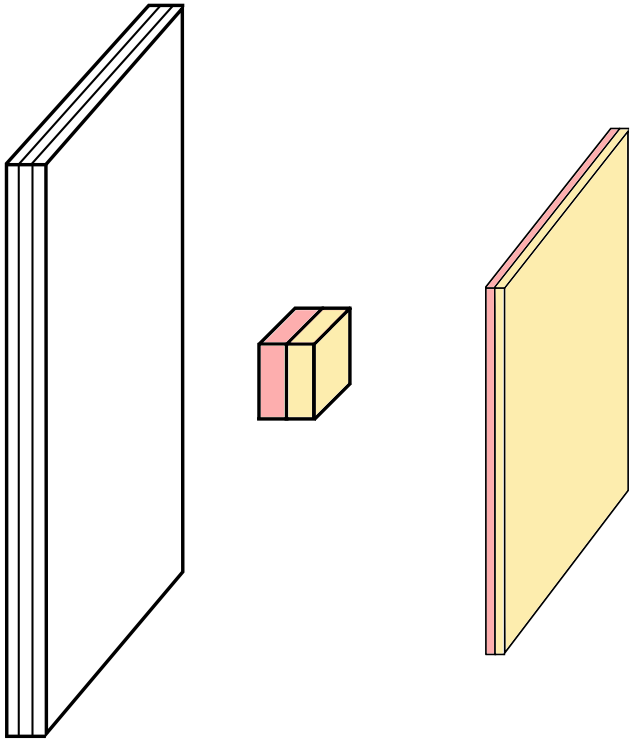


$$(k \star im^{color}) = \sum_{c=0}^{2} k^c \star im^c$$

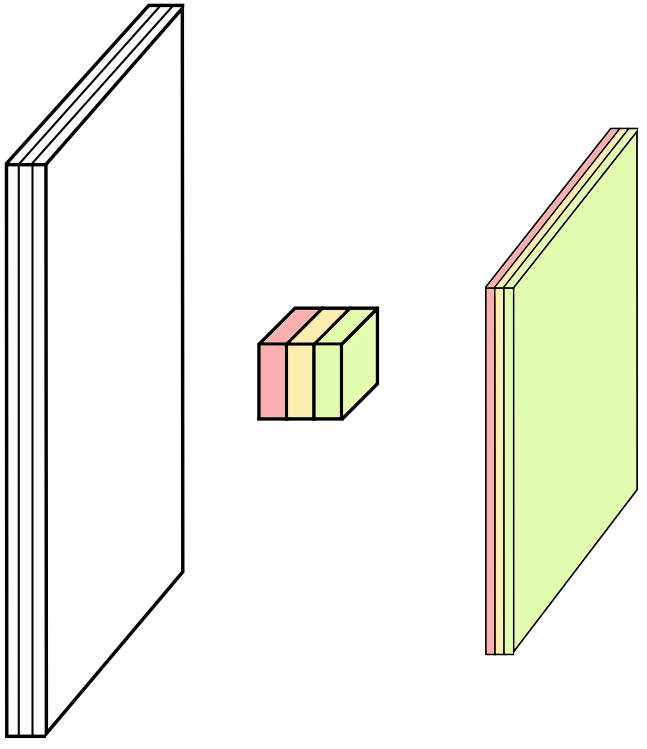
Multiple convolutions



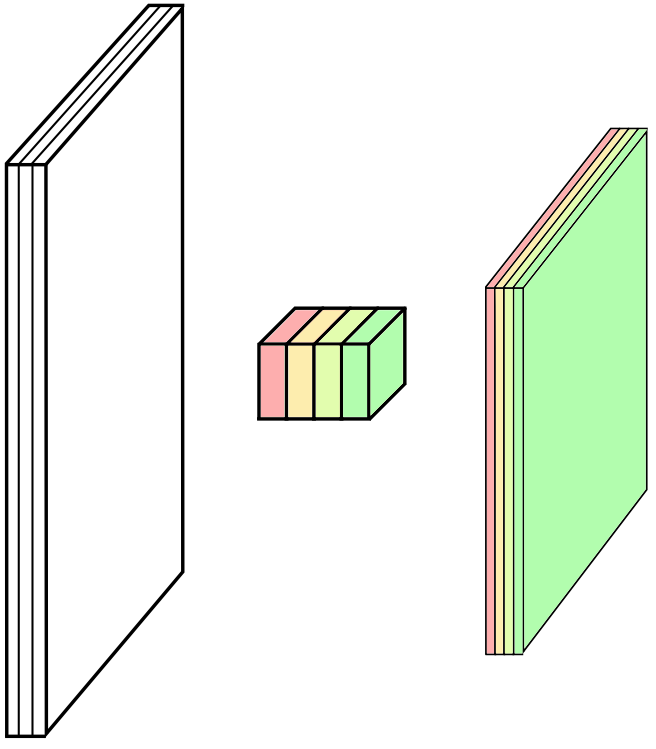
Multiple convolutions



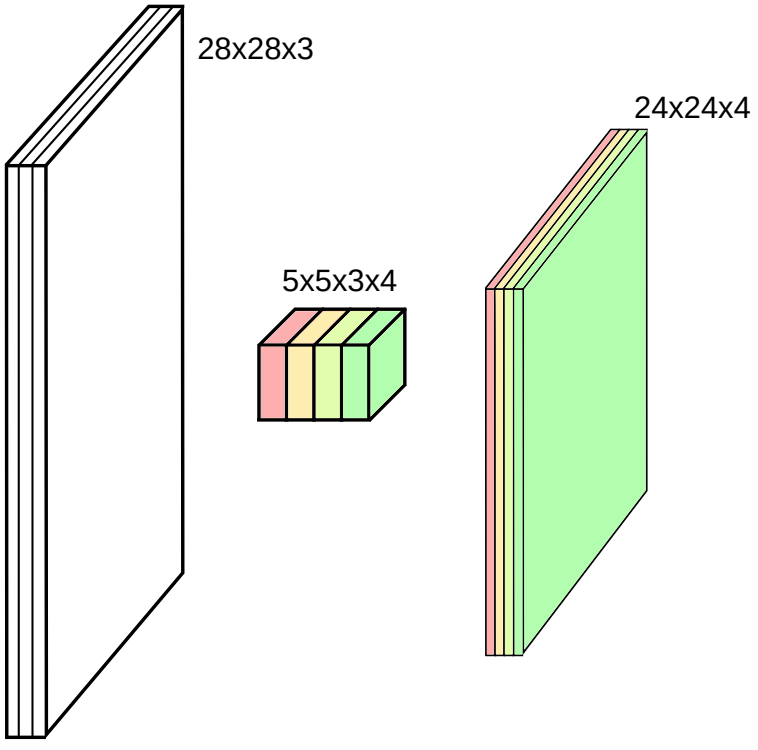
Multiple convolutions



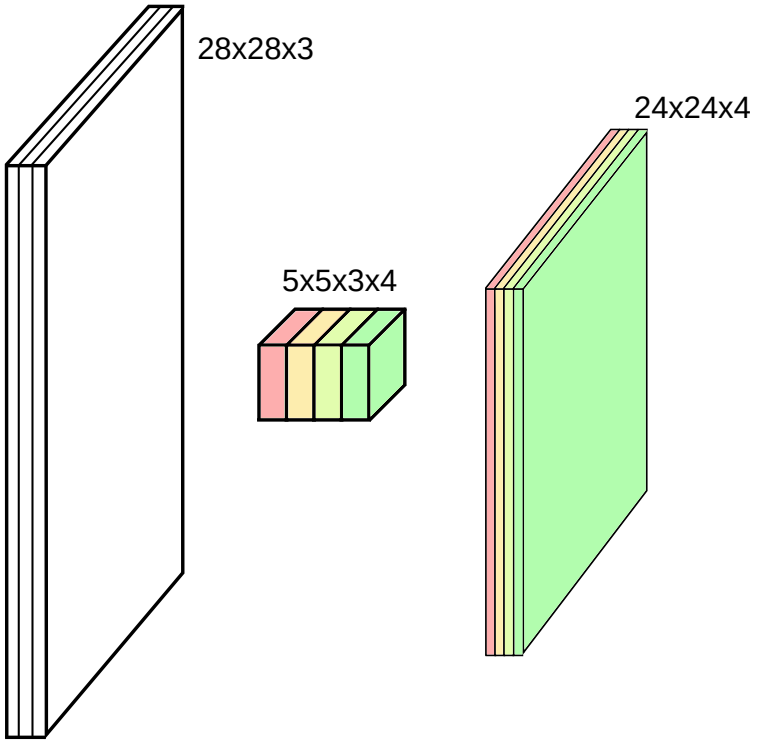
Multiple convolutions



Multiple convolutions



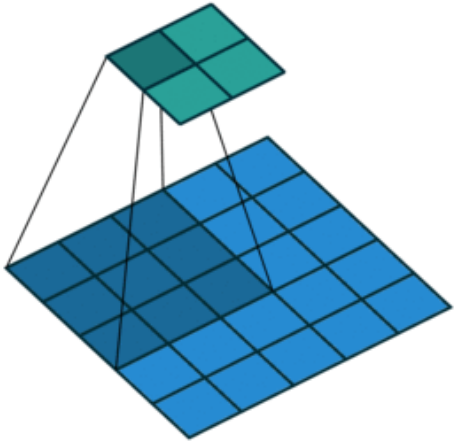
Multiple convolutions



- Kernel size: usually 1, 3, 5, 7, 11
- Output dimension: `(width - kernel_size + 1, height - kernel_size + 1)`

Strides

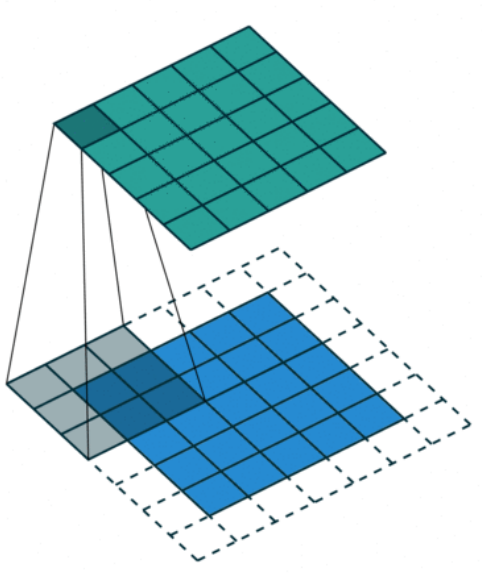
- Strides: increment step size for the convolution operator
- Reduces the size of the output map



Example with kernel size 3×3 and a stride of 2 (input in blue, output in green)

Padding

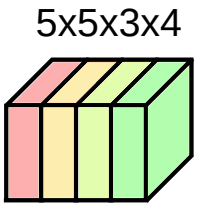
- Padding: artificially fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields
- Usually: fill with 0s



Dealing with shapes

Kernel or filter shape: (C^i, C^o, k, k)

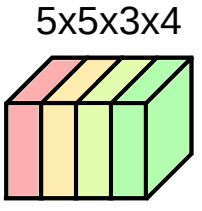
- C^i input channels
- C^o output channels
- $k \times k$ kernel size



Dealing with shapes

Kernel or filter shape: (C^i, C^o, k, k)

- C^i input channels
- C^o output channels
- $k \times k$ kernel size

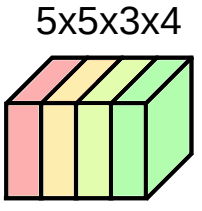


Number of parameters: $(k \times k \times C^i + 1) \times C^o$

Dealing with shapes

Kernel or filter shape: (C^i, C^o, k, k)

- C^i input channels
- C^o output channels
- $k \times k$ kernel size



Number of parameters: $(k \times k \times C^i + 1) \times C^o$

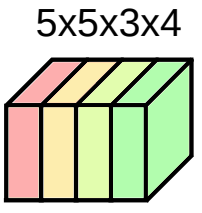
Activations or feature maps shape:

- Input (C^i, W^i, H^i)
- Output (C^o, W^o, H^o)

Dealing with shapes

Kernel or filter shape: (C^i, C^o, k, k)

- C^i input channels
- C^o output channels
- $k \times k$ kernel size



Number of parameters: $(k \times k \times C^i + 1) \times C^o$

Activations or feature maps shape:

- Input (C^i, W^i, H^i)
- Output (C^o, W^o, H^o)

$W^o = \lfloor (W^i - f + 2P)/S + 1 \rfloor$ and $H^o = \lfloor (H^i - f + 2P)/S + 1 \rfloor$

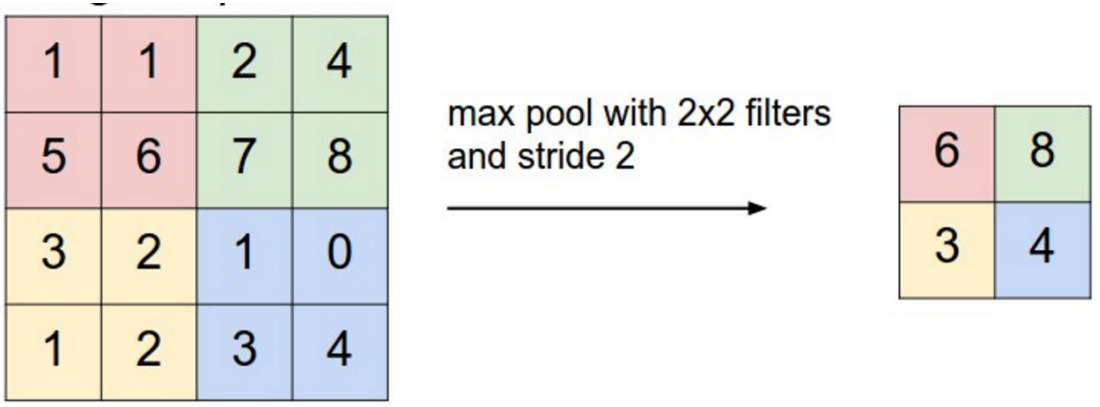
where P is the padding and S the stride.

Pooling

- Spatial dimension reduction
- Local invariance
- No parameters: max or average, for example of 2×2 units

Pooling

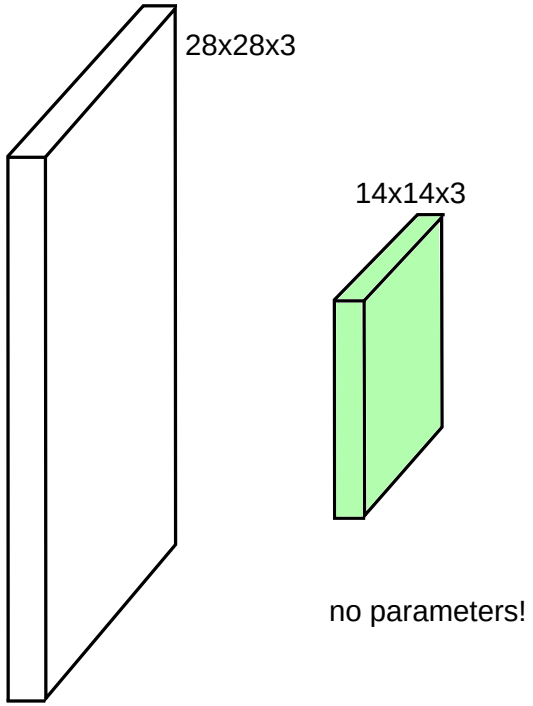
- Spatial dimension reduction
- Local invariance
- No parameters: max or average, for example of 2×2 units



Schematic from Stanford http://cs231n.github.io/convolutional-networks

Pooling

- Spatial dimension reduction
- Local invariance
- No parameters: max or average, for example of 2×2 units



Architectures

Classic ConvNet Architecture

Input

Classic ConvNet Architecture

Input

Conv blocks

- Convolution + activation (relu)
- Convolution + activation (relu)
- ...
- Maxpooling 2x2

Classic ConvNet Architecture

Input

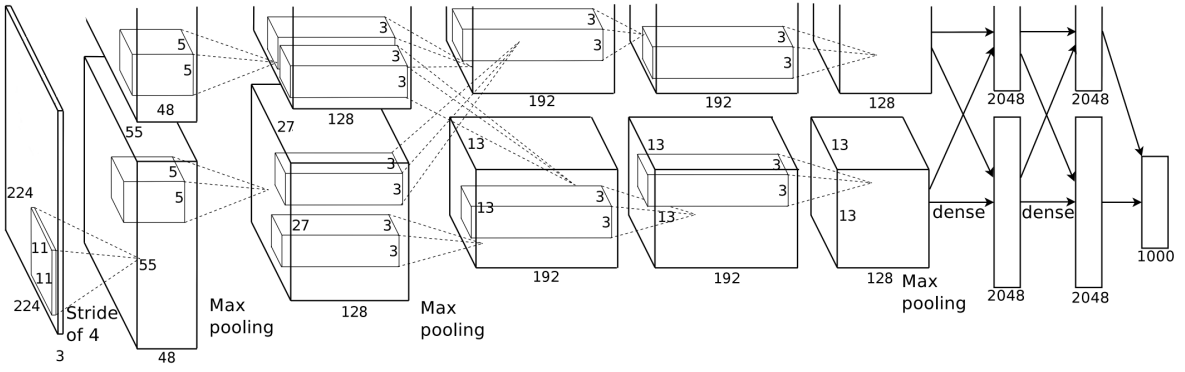
Conv blocks

- Convolution + activation (relu)
- Convolution + activation (relu)
- ...
- Maxpooling 2x2

Output

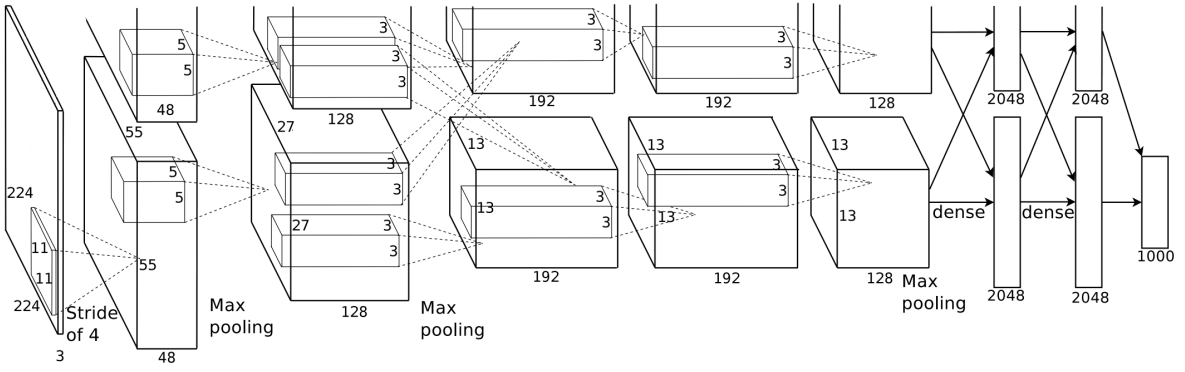
- Fully connected layers
- Softmax

AlexNet



Simplified version of Krizhevsky, Alex, Sutskever, and Hinton. "Imagenet classification with deep convolutional neural networks." NIPS 2012

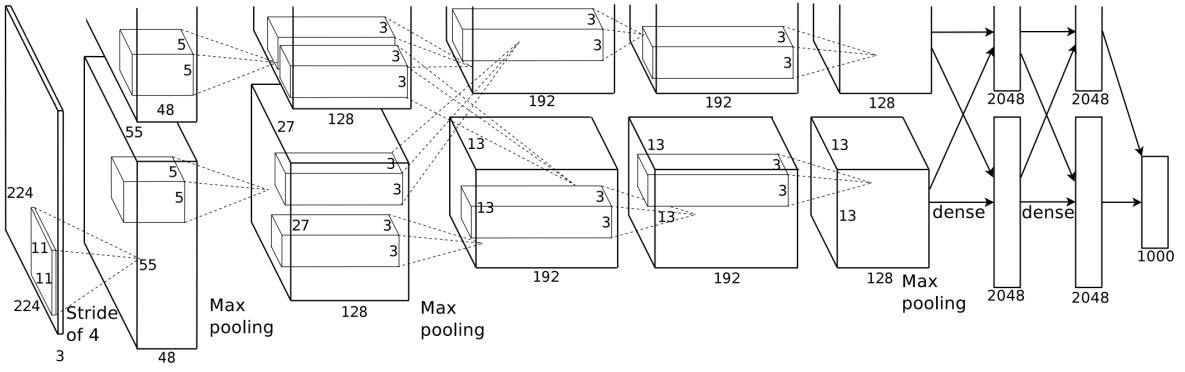
AlexNet



First conv layer: kernel $3 \times 96 \times 11 \times 11$ stride 4

Simplified version of Krizhevsky, Alex, Sutskever, and Hinton. "Imagenet classification with deep convolutional neural networks." NIPS 2012

AlexNet

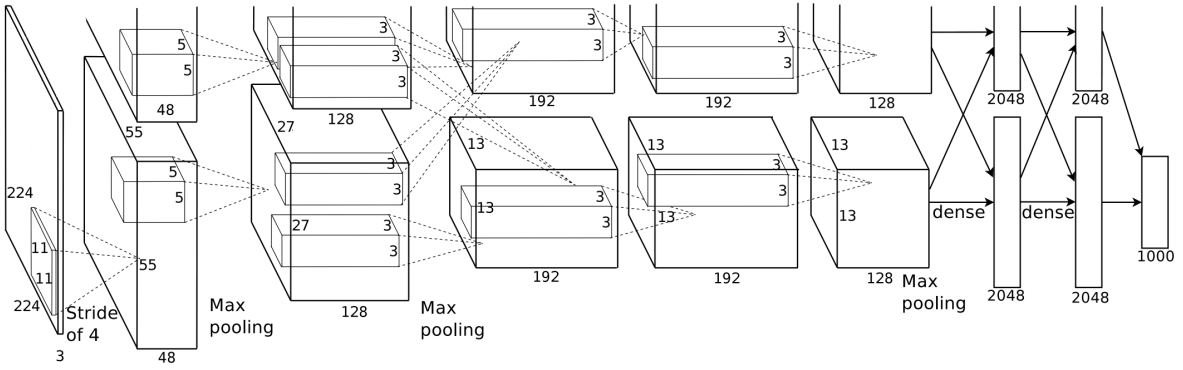


First conv layer: kernel $3 \times 96 \times 11 \times 11$ stride 4

- Kernel shape: `(3,96,11,11)`
- Output shape: `(96,55,55)`
- Number of parameters: `34,944`
- Equivalent MLP parameters: `43.7 x 1e9`

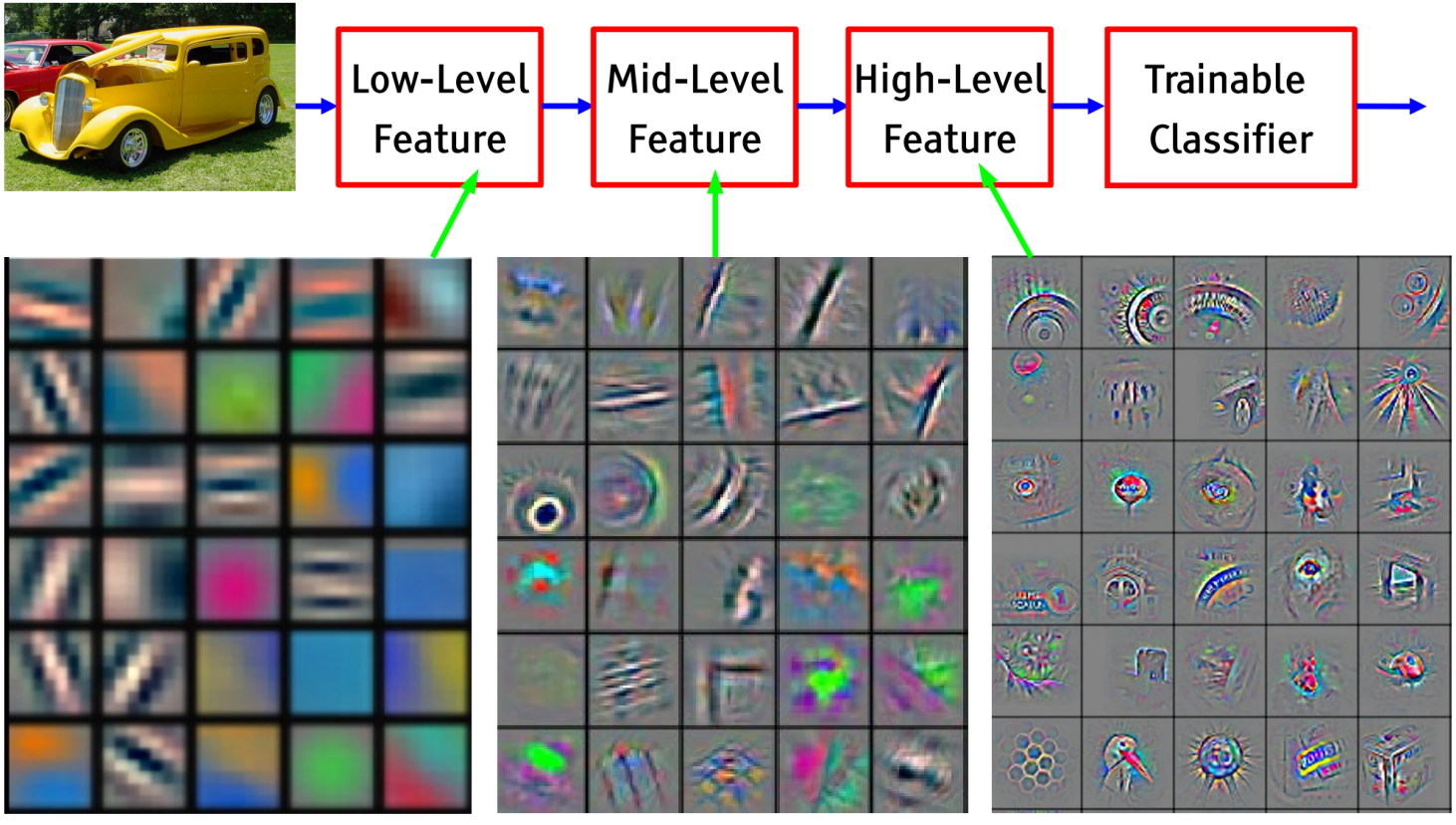
Simplified version of Krizhevsky, Alex, Sutskever, and Hinton. "Imagenet classification with deep convolutional neural networks." NIPS 2012

AlexNet



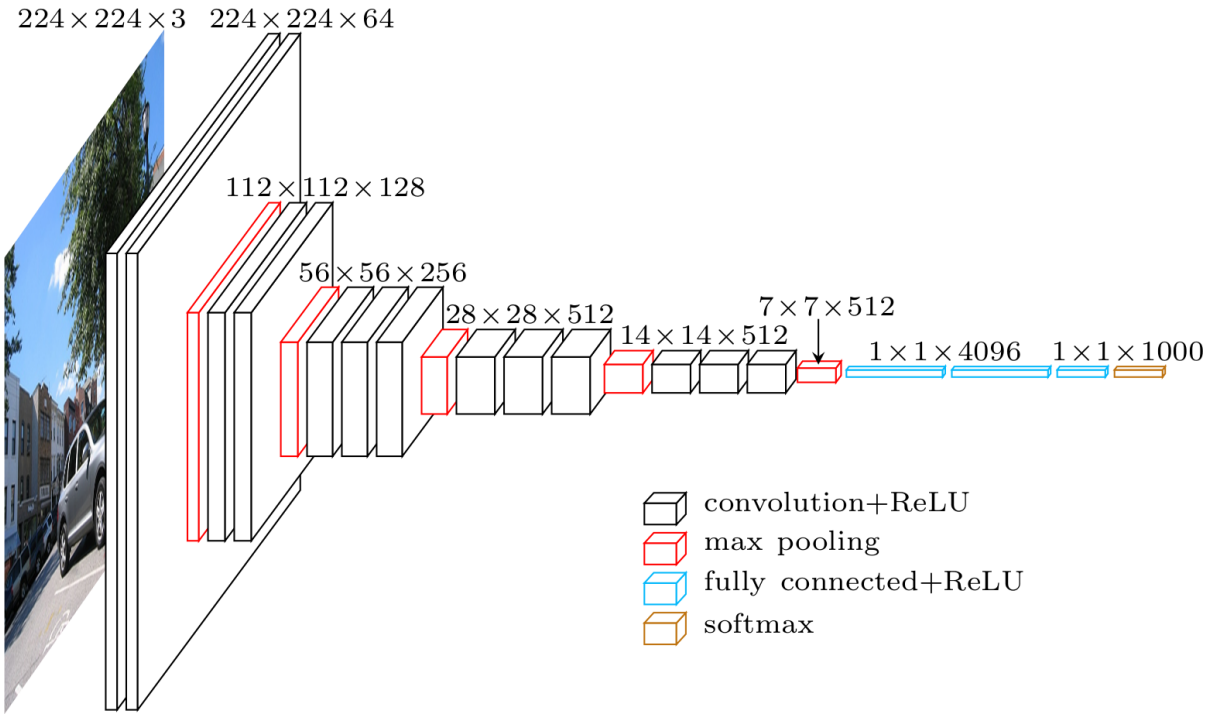

```
INPUT:     [3x227x227]
CONV1:     [96x55x55]   96 11x11 filters at stride 4, pad 0
MAX POOL1: [96x27x27]      3x3   filters at stride 2
CONV2:     [256x27x27] 256 5x5   filters at stride 1, pad 2
MAX POOL2: [256x13x13]     3x3   filters at stride 2
CONV3:     [384x13x13] 384 3x3   filters at stride 1, pad 1
CONV4:     [384x13x13] 384 3x3   filters at stride 1, pad 1
CONV5:     [256x13x13] 256 3x3   filters at stride 1, pad 1
MAX POOL3: [256x6x6]       3x3   filters at stride 2
FC6:       [4096]      4096 neurons
FC7:       [4096]      4096 neurons
FC8:       [1000]      1000 neurons (softmax logits)
```

Hierarchical representation



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

VGG-16



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

VGG in Keras (backend Tensorflow)

```
model.add(Convolution2D(64, 3, 3, activation='relu',input_shape=(3,224,224)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))
```

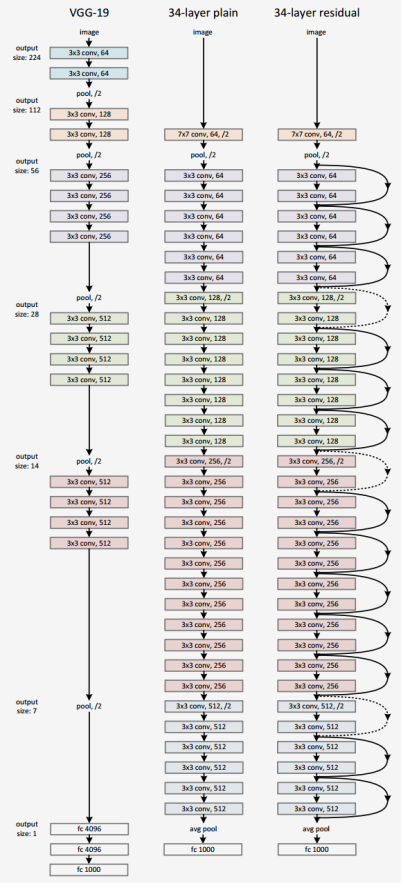
Memory and Parameters

```
           Activation maps          Parameters
INPUT:     [3x224x224]   = 150K     0
CONV3-64:  [64x224x224]  = 3.2M     (3x3x3)x64    =       1,728
CONV3-64:  [64x224x224]  = 3.2M     (3x3x64)x64   =      36,864
POOL2:     [64x112x112]  = 800K     0
CONV3-128: [128x112x112] = 1.6M     (3x3x64)x128  =      73,728
CONV3-128: [128x112x112] = 1.6M     (3x3x128)x128 =     147,456
POOL2:     [128x56x56]   = 400K     0
CONV3-256: [256x56x56]   = 800K     (3x3x128)x256 =     294,912
CONV3-256: [256x56x56]   = 800K     (3x3x256)x256 =     589,824
CONV3-256: [256x56x56]   = 800K     (3x3x256)x256 =     589,824
POOL2:     [256x28x28]   = 200K     0
CONV3-512: [512x28x28]   = 400K     (3x3x256)x512 =   1,179,648
CONV3-512: [512x28x28]   = 400K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x28x28]   = 400K     (3x3x512)x512 =   2,359,296
POOL2:     [512x14x14]   = 100K     0
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
POOL2:     [512x7x7]     =  25K     0
FC:        [4096x1x1]    = 4096     7x7x512x4096  = 102,760,448
FC:        [4096x1x1]    = 4096     4096x4096     =  16,777,216
FC:        [1000x1x1]    = 1000     4096x1000     =   4,096,000

TOTAL activations: 24M x 4 bytes ~=  93MB / image (x2 for backward)
TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)
```

Memory and Parameters

```
           Activation maps          Parameters
INPUT:     [3x224x224]   = 150K     0
*CONV3-64:  [64x224x224]  = 3.2M     (3x3x3)x64    =       1,728
*CONV3-64:  [64x224x224]  = 3.2M     (3x3x64)x64   =      36,864
POOL2:     [64x112x112]  = 800K     0
CONV3-128: [128x112x112] = 1.6M     (3x3x64)x128  =      73,728
CONV3-128: [128x112x112] = 1.6M     (3x3x128)x128 =     147,456
POOL2:     [128x56x56]   = 400K     0
CONV3-256: [256x56x56]   = 800K     (3x3x128)x256 =     294,912
CONV3-256: [256x56x56]   = 800K     (3x3x256)x256 =     589,824
CONV3-256: [256x56x56]   = 800K     (3x3x256)x256 =     589,824
POOL2:     [256x28x28]   = 200K     0
CONV3-512: [512x28x28]   = 400K     (3x3x256)x512 =   1,179,648
CONV3-512: [512x28x28]   = 400K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x28x28]   = 400K     (3x3x512)x512 =   2,359,296
POOL2:     [512x14x14]   = 100K     0
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
CONV3-512: [512x14x14]   = 100K     (3x3x512)x512 =   2,359,296
POOL2:     [512x7x7]     =  25K     0
*FC:        [4096x1x1]    = 4096     7x7x512x4096  = 102,760,448
FC:        [4096x1x1]    = 4096     4096x4096     =  16,777,216
FC:        [1000x1x1]    = 1000     4096x1000     =   4,096,000

TOTAL activations: 24M x 4 bytes ~=  93MB / image (x2 for backward)
TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)
```

ResNet



He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

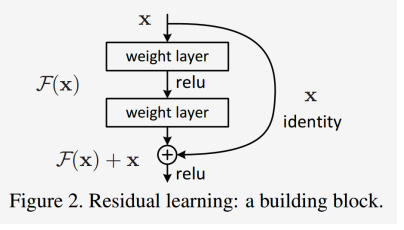


Figure 2. Residual learning: a building block.

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

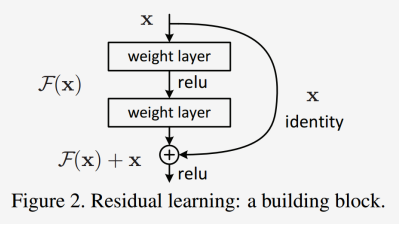


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

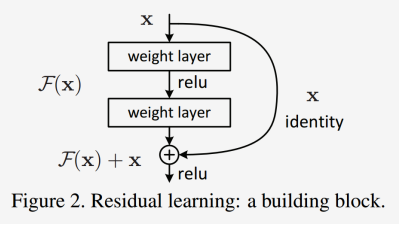


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers
- Good optimization properties

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

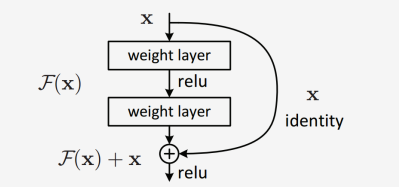


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers
- Good optimization properties
- ResNet50 Compared to VGG:
  - **Superior accuracy in all vision tasks** 5.25% **top-5 error vs 7.1%**

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

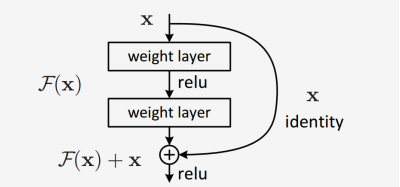


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers
- Good optimization properties
- ResNet50 Compared to VGG:
  - **Superior accuracy in all vision tasks** 5.25% **top-5 error vs 7.1%**
  - **Less parameters** 25M **vs 138M**

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

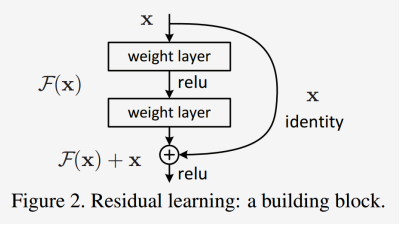


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers
- Good optimization properties
- ResNet50 Compared to VGG:
  - **Superior accuracy in all vision tasks** 5.25% **top-5 error vs 7.1%**
  - **Less parameters** 25M **vs 138M**
  - **Computational complexity** 3.8B Flops **vs 15.3B Flops**

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

ResNet

A block learns the residual w.r.t. identity

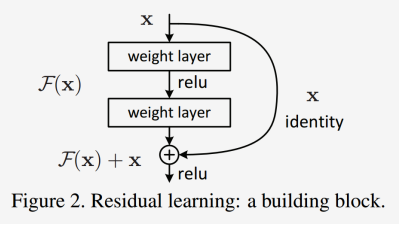
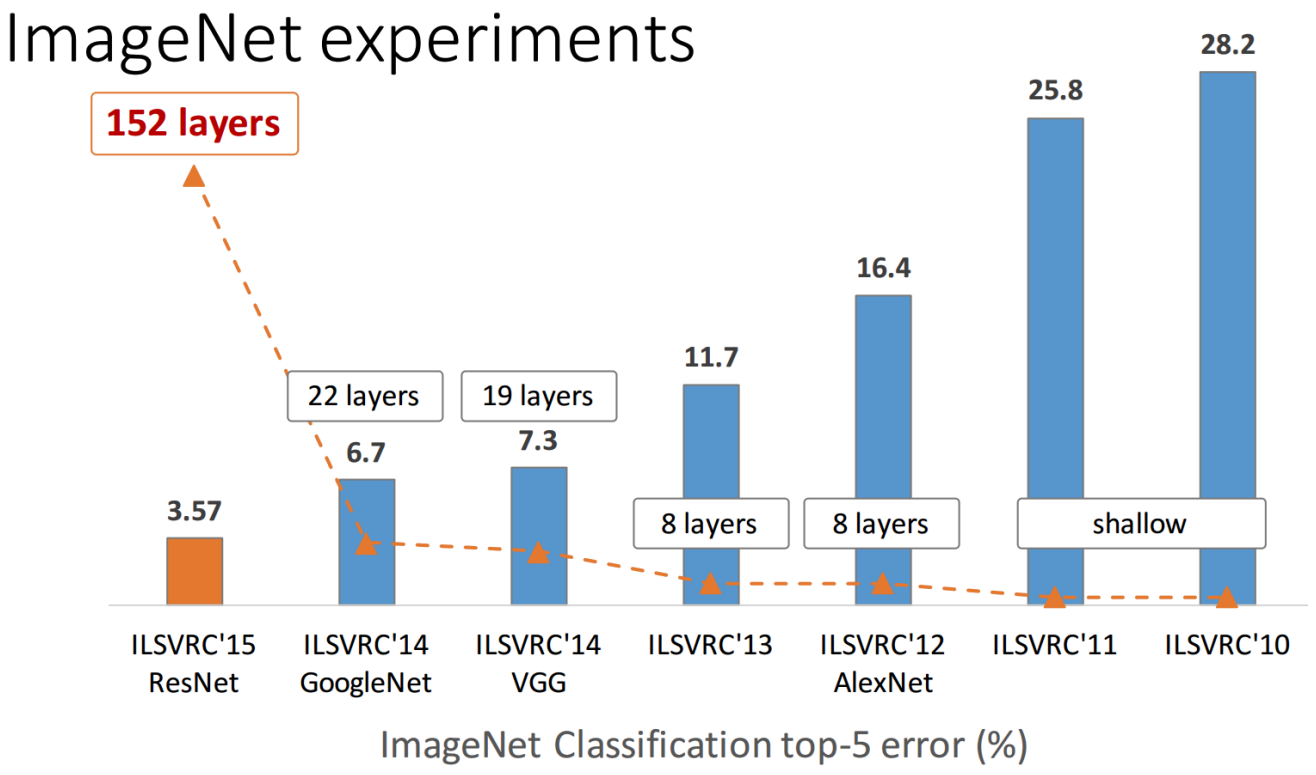


Figure 2. Residual learning: a building block.

- Even deeper models: 34, 50, 101, 152 layers
- Good optimization properties
- ResNet50 Compared to VGG:
  - **Superior accuracy in all vision tasks** 5.25% **top-5 error vs 7.1%**
  - **Less parameters** 25M **vs 138M**
  - **Computational complexity** 3.8B Flops **vs 15.3B Flops**
  - **Fully Convolutional until the last layer**

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.

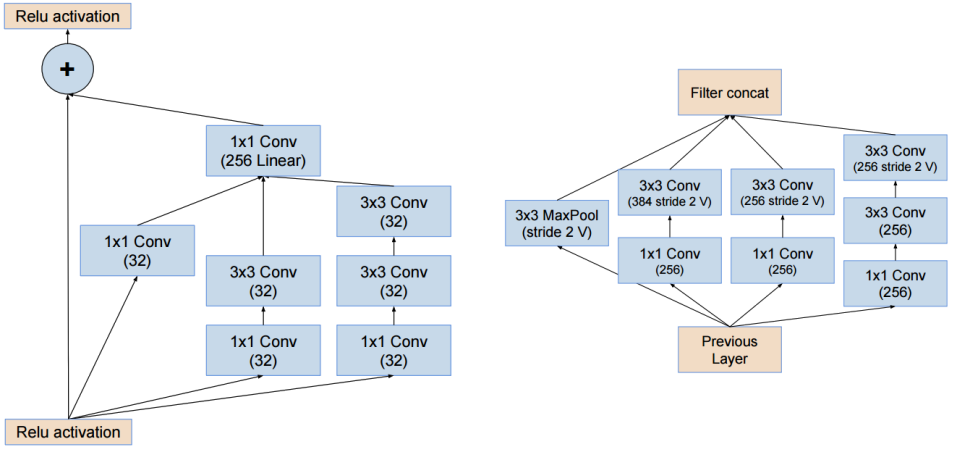
Deeper is better



from Kaiming He slides "Deep residual learning for image recognition." ICML. 2016.

State of the art

- Finding right architectures: Active area or research

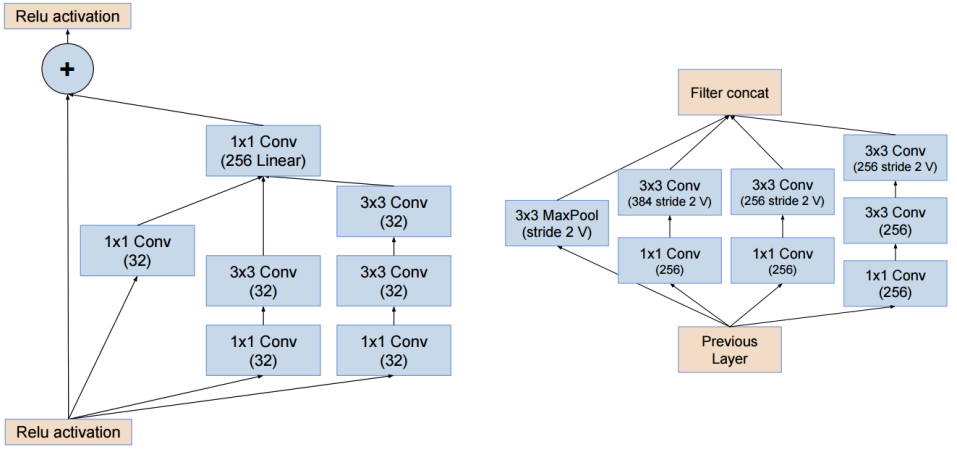


Modular building blocks engineering

from He slides "Deep residual learning for image recognition." ICML. 2016.

State of the art

- Finding right architectures: Active area or research



Modular building blocks engineering

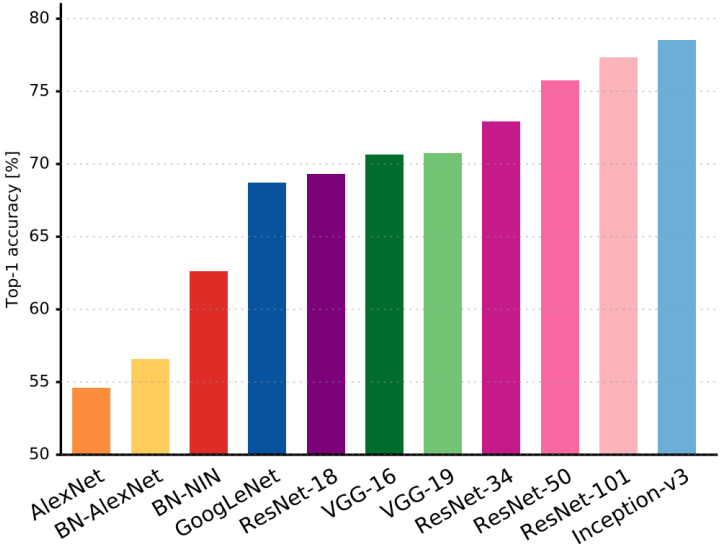
see also DenseNets, Wide ResNets, Fractal ResNets, ResNeXts, Pyramidal ResNets

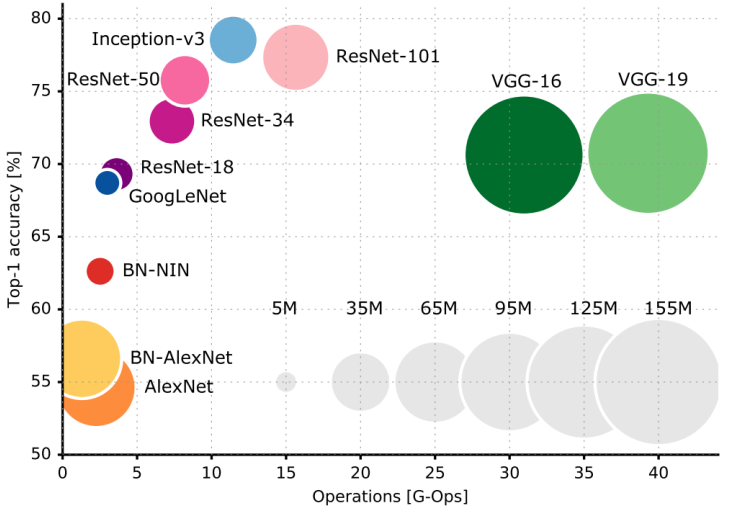
---

from He slides "Deep residual learning for image recognition." ICML. 2016.

State of the art

Top 1-accuracy, performance and size on ImageNet



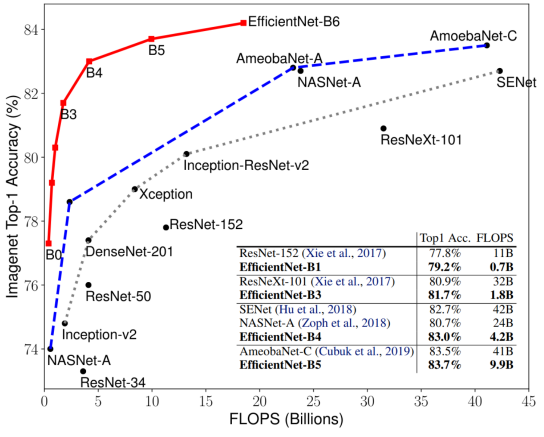


See also: https://paperswithcode.com/sota/image-classification-on-imagenet

---

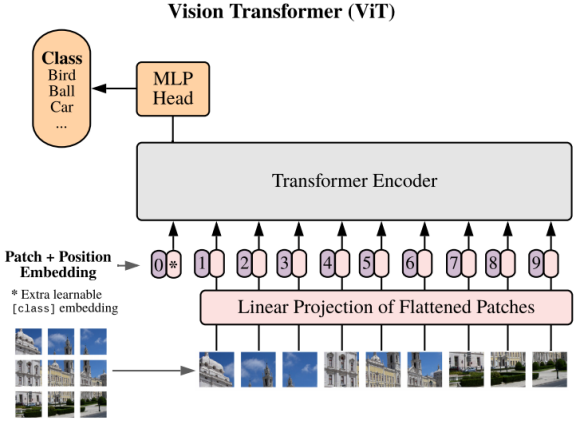
Canziani, Paszke, and Culurciello. "An Analysis of Deep Neural Network Models for Practical Applications." (May 2016).

More ImageNet SOTA



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**79.2%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.7%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**83.0%**	**4.2B**
AmeobaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.7%**	**9.9B**

Figure 5. **FLOPS vs. ImageNet Accuracy –** Similar to Figure 1 except it compares FLOPS rather than model size.



- Mingxing Tan, Quoc V. Le, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks , ICML 2019.
- Irwan Bello, LambdaNetworks: Modeling long-range Interactions without Attention, ICLR 2021
- Dosovitskiy A. et al, An Image is worth 16X16 Words: Transformers for Image Recognition at Scale, ICLR 2021

State of the art

Method	# Params	Extra Data	ImageNet Top-1	ImageNet Top-5	ImageNet-ReaL [6] Precision@1
ResNet-50 [24]	26M	–	76.0	93.0	82.94
ResNet-152 [24]	60M	–	77.8	93.8	84.79
DenseNet-264 [28]	34M	–	77.9	93.9	–
Inception-v3 [62]	24M	–	78.8	94.4	83.58
Xception [11]	23M	–	79.0	94.5	–
Inception-v4 [61]	48M	–	80.0	95.0	–
Inception-resnet-v2 [61]	56M	–	80.1	95.1	–
ResNeXt-101 [78]	84M	–	80.9	95.6	85.18
PolyNet [87]	92M	–	81.3	95.8	–
SENet [27]	146M	–	82.7	96.2	–
NASNet-A [90]	89M	–	82.7	96.2	82.56
AmoebaNet-A [52]	87M	–	82.8	96.1	–
PNASNet [39]	86M	–	82.9	96.2	–
AmoebaNet-C + AutoAugment [12]	155M	–	83.5	96.5	–
GPipe [29]	557M	–	84.3	97.0	–
EfficientNet-B7 [63]	66M	–	85.0	97.2	–
EfficientNet-B7 + FixRes [70]	66M	–	85.3	97.4	–
EfficientNet-L2 [63]	480M	–	85.5	97.5	–
ResNet-50 Billion-scale SSL [79]	26M	3.5B labeled Instagram	81.2	96.0	–
ResNeXt-101 Billion-scale SSL [79]	193M	3.5B labeled Instagram	84.8	–	–
ResNeXt-101 WSL [42]	829M	3.5B labeled Instagram	85.4	97.6	88.19
FixRes ResNeXt-101 WSL [69]	829M	3.5B labeled Instagram	86.4	98.0	89.73
Big Transfer (BiT-L) [33]	928M	300M labeled JFT	87.5	98.5	90.54
Noisy Student (EfficientNet-L2) [77]	480M	300M unlabeled JFT	88.4	98.7	90.55
Noisy Student + FixRes [70]	480M	300M unlabeled JFT	88.5	98.7	–
Vision Transformer (ViT-H) [14]	632M	300M labeled JFT	88.55	–	90.72
EfficientNet-L2-NoisyStudent + SAM [16]	480M	300M unlabeled JFT	88.6	98.6	–
Meta Pseudo Labels (EfficientNet-B6-Wide)	390M	300M unlabeled JFT	90.0	98.7	**91.12**
Meta Pseudo Labels (EfficientNet-L2)	480M	300M unlabeled JFT	**90.2**	**98.8**	91.02

Meta Pseudo Labels, Hieu Pham et al. (Jan 2021)

Pre-trained models

Pre-trained models

- Training a model on ImageNet from scratch takes days or weeks.

Pre-trained models

- Training a model on ImageNet from scratch takes days or weeks.
- Many models trained on ImageNet and their weights are publicly available!

Pre-trained models

- Training a model on ImageNet from scratch takes days or weeks.
- Many models trained on ImageNet and their weights are publicly available!

Transfer learning

- Use pre-trained weights, remove last layers to compute representations of images
- Train a classification model from these features on a new classification task
- The network is used as a generic feature extractor
- Better than handcrafted feature extraction on natural images

Fine-tuning

- Retraining the (some) parameters of the network (given enough data)
- Truncate the last layer(s) of the pre-trained network
- Freeze the remaining layers weights
- Add a (linear) classifier on top and train it for a few epochs
- Then fine-tune the whole network or the few deepest layers
- Use a smaller learning rate when fine tuning

Data Augmentation

Data Augmentation

Data Augmentation

See also: RandAugment and Unsupervised Data Augmentation for Consistency Training.

More during the lab session