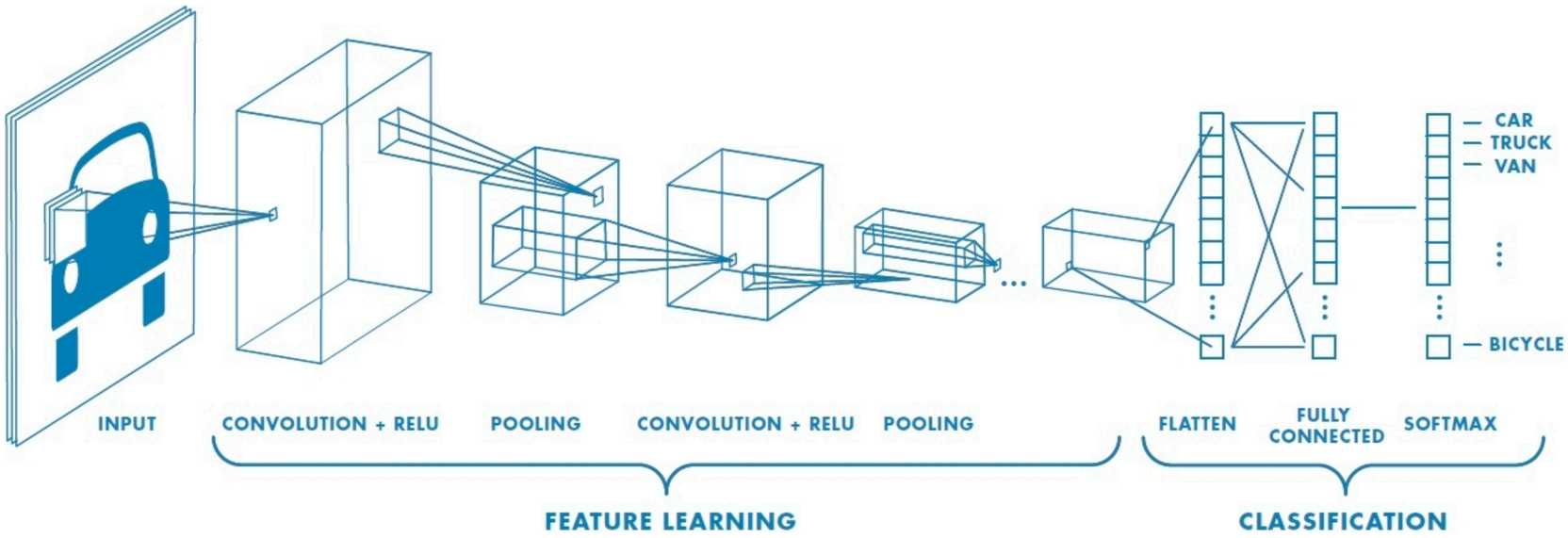
Convolutional Neural Networks

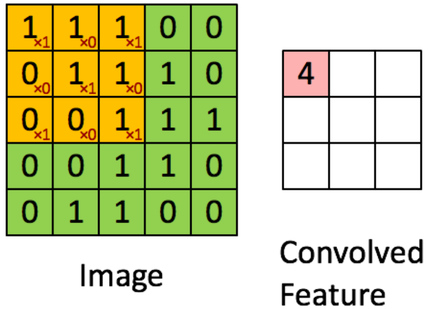
Rafał Nowak April 8, 2025

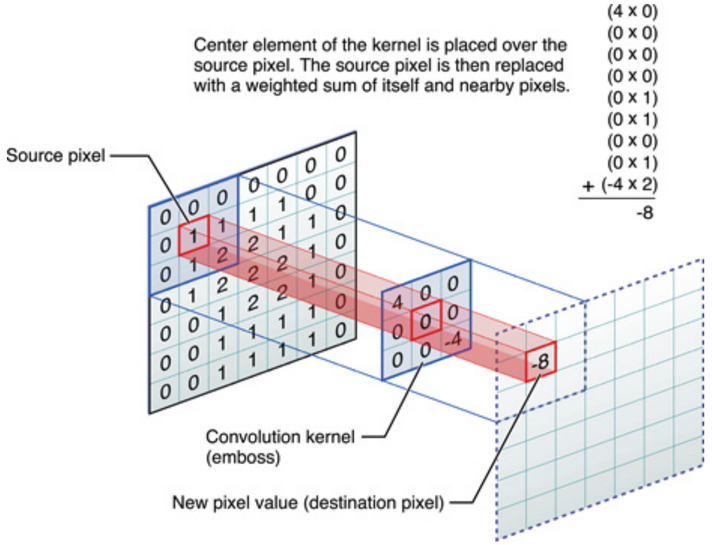
Convolutional Neural Networks

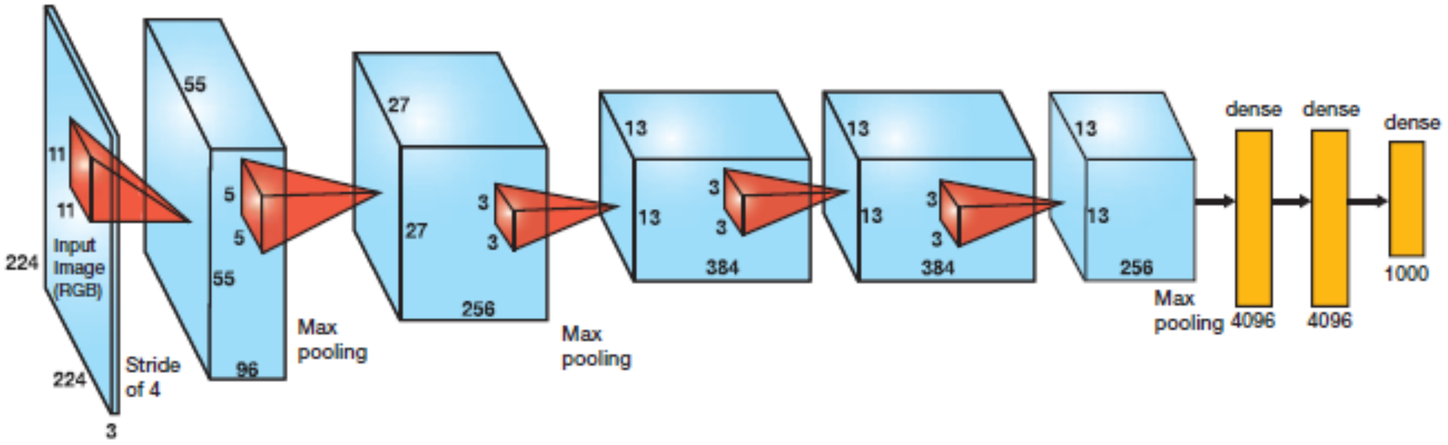


https://www.mathworks.com/discovery/convolutional-neural-network.html

Convolutional Neural Networks





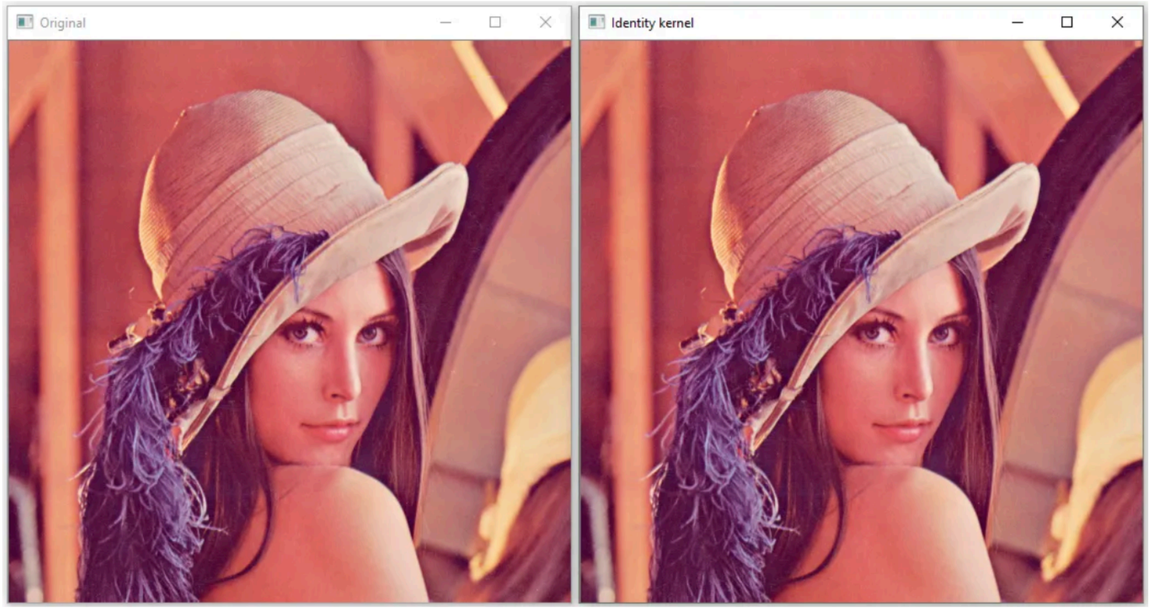


Convolutional Neural Networks

Identity kernel

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Identity Kernel



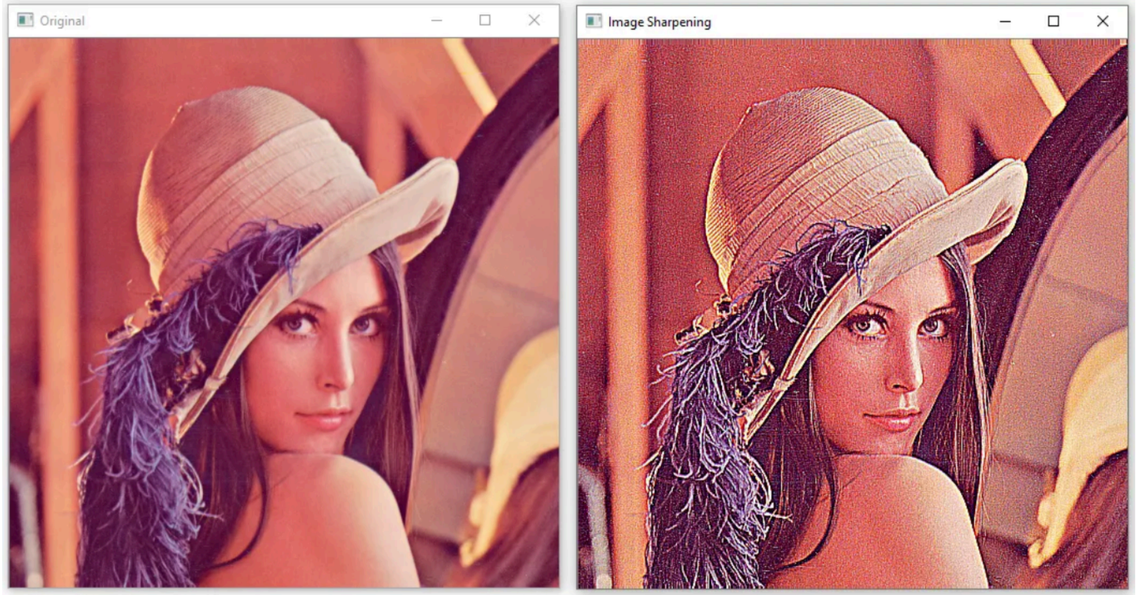
Original Image(Left) and Image after applying Identity Filter of size 3×3(Right)

Convolutional Neural Networks

Sharpen kernel

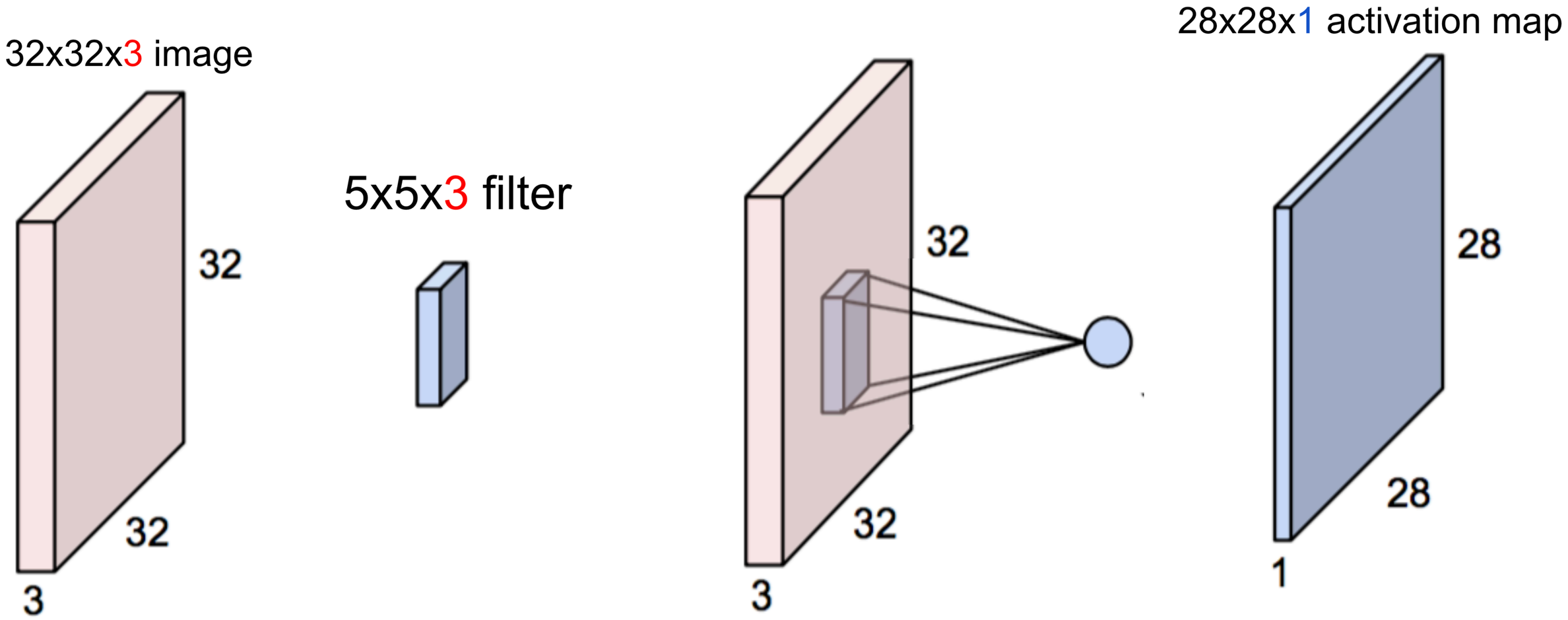
```
[[-1 -1 -1]
 [-1  9 -1]
 [-1 -1 -1]]
```

Sharpen Kernel



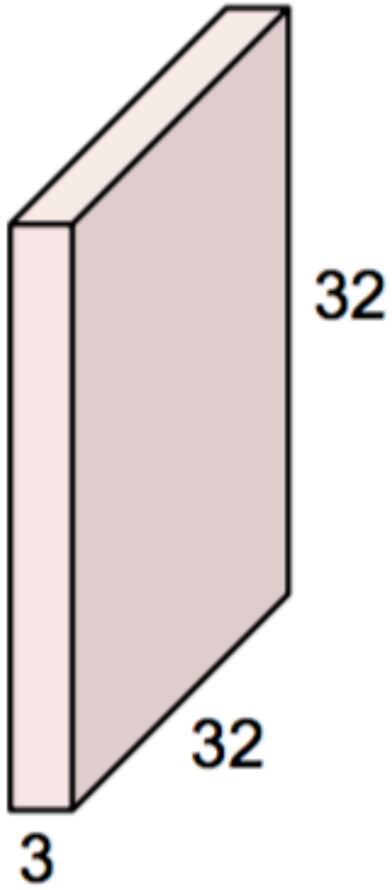
Original Image(Left) and Image after applying Sharpen Filter of size 3×3 (Right)

Convolutional Neural Networks

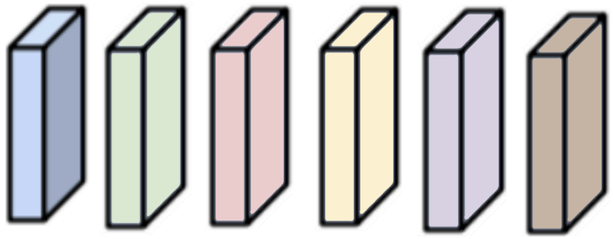


CNN Filters

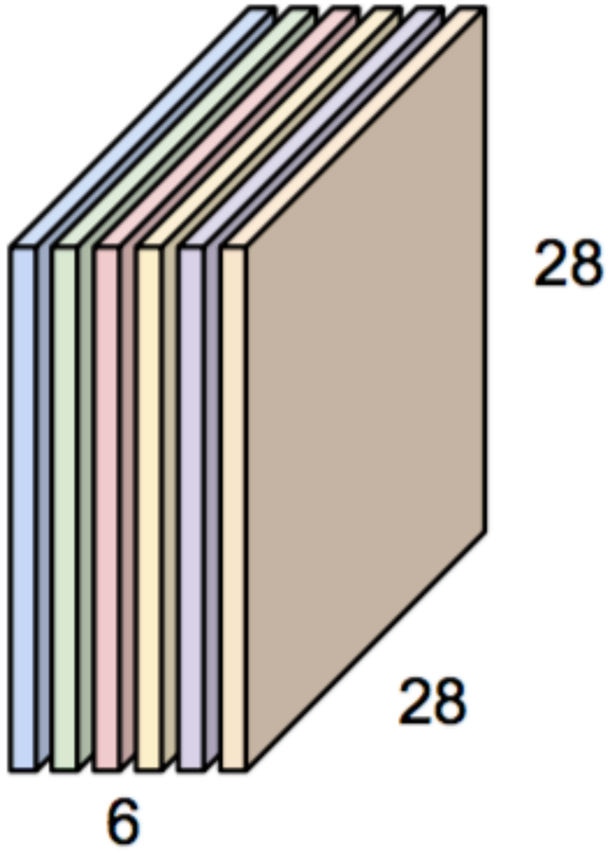
32x32x3 image



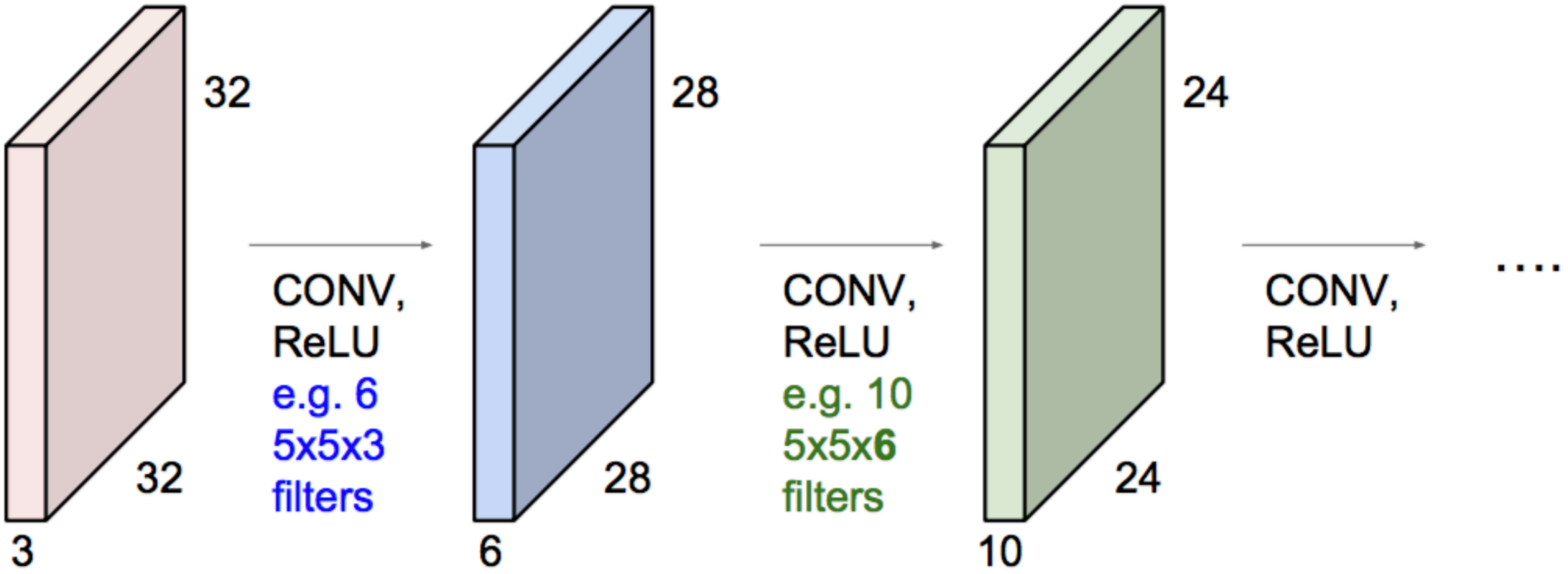
6 filters 5x5x3



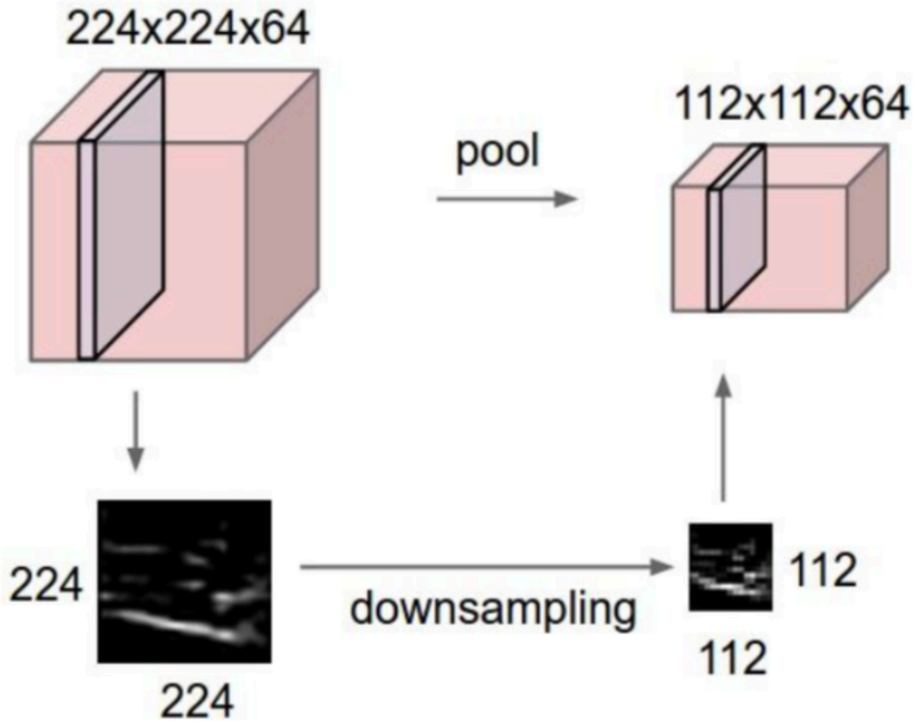
28x28x6 activation map



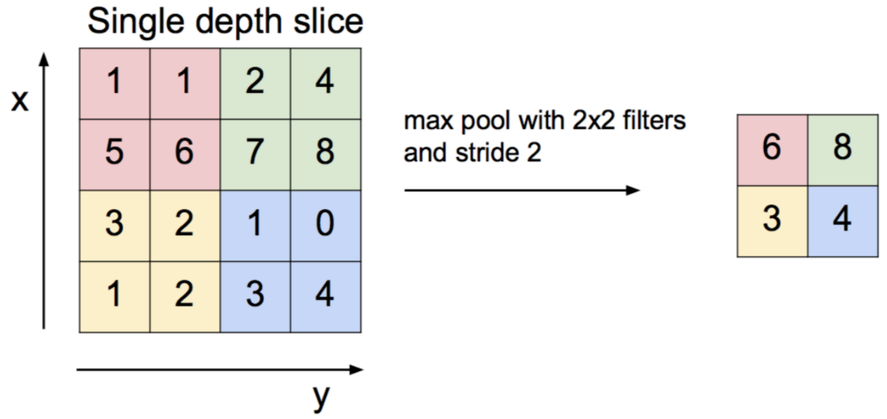
CNN: Activation



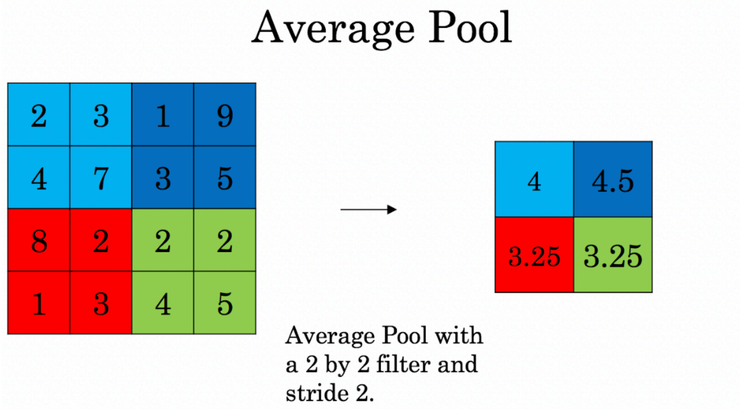
CNN: Pooling



- MAX pooling



- AVG pooling



Le-Net 1998

- Y. LeCun et al., Gradient-based learning applied to document recognition, 1998
- Average pooling
- Sigmoid + tanh
- Dense Top Layers
- MNIST (60k) dataset

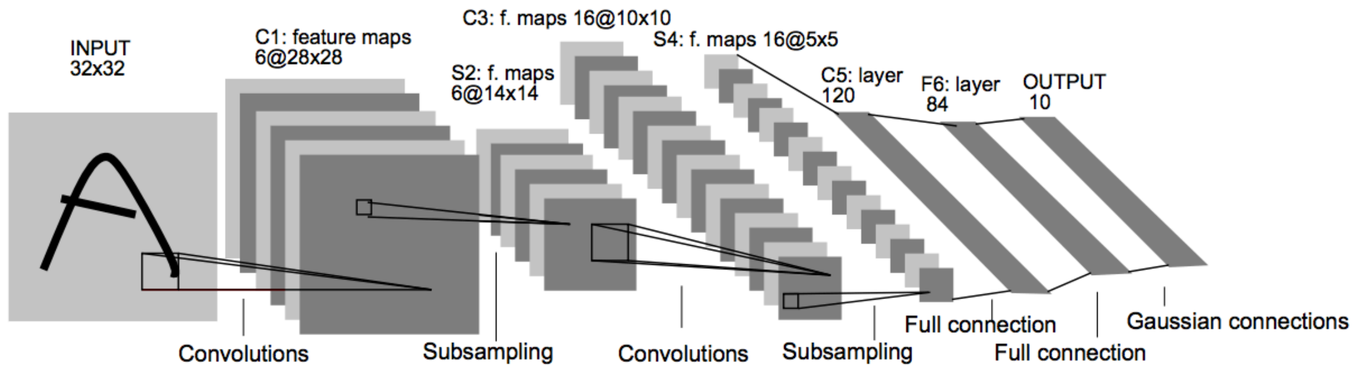


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

torch.nn.Conv2D

CONV2D

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Variables

- **weight** (*Tensor*) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

Examples

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

Some notes on CNNs

- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations
- The parameters of the network can be learned through regular back propagation
- Like a regular MLP, individual layers may either increase or decrease the span of the representation learned
- The models can be easily modified to include invariance to other transforms
  - Although these tend to be computationally painful.

PyTorch: switching to GPU

```
1  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
2  device
```

```
device(type='cuda')
```

```
class MyAwesomeNeuralNetwork(nn.Module):
    # your model here

model = MyAwesomeNeuralNetwork()
model.to(device)

epochs = 10
for epoch in range(epochs):
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        # backpropagation code here

        # evaluation
        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
        # ...
```

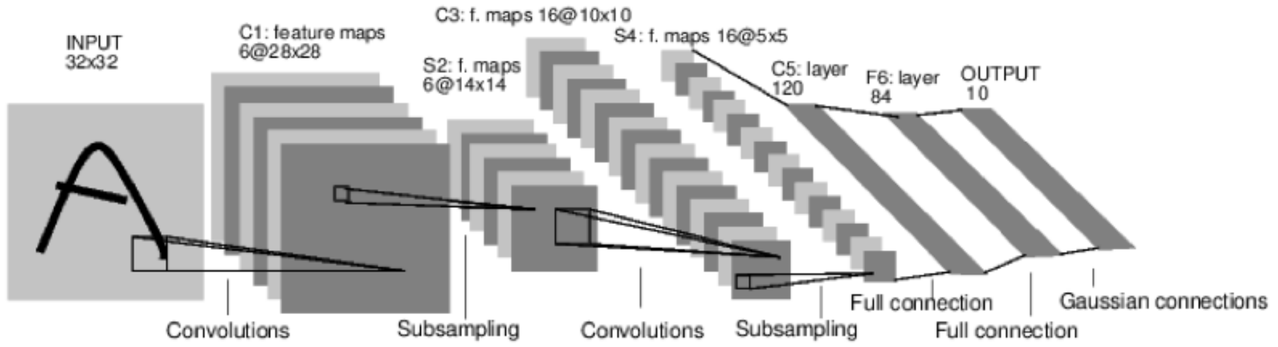
see https://towardsdatascience.com/pytorch-switching-to-the-gpu-a7c0b21e8a99

Some CNN architectures

- LeNet-5
- AlexNet
- VGGNet
- GoogLeNet
- ResNet
- and some others like: ResNeXt, DenseNet, SqueezeNet, MobileNet, etc

LeNet-5 (1998)

- Y. LeCun et al., Gradient-based learning applied to document recognition, 1998
- Average pooling
- Sigmoid + tanh - activation function
- Dense layers at the top (end)
- Trained on MNIST (60K images), CPU
- It consists of **7 layers**:
  - 3 convolutional layers
  - 2 subsampling layers
  - 2 fully connected layers

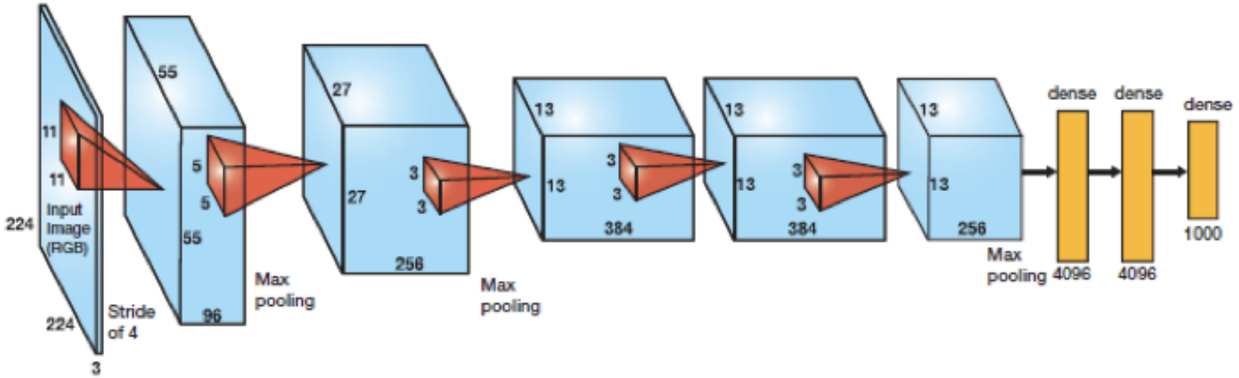


ImageNet Large Scale Visual Recognition Challenge

ILSVRC

- The ImageNet Large Scale V annual competition where research teams comp
- The competition was starte
- The competition is held at th ern Recognition (CVPR)
- The competition is hosted b
- The competition is sponsore
- IEEE, https://ieeexplore.ieee
- see https://cs.stanford.edu/people/karpathy/cnnembed/ for nice visualizations

AlexNet (2012)



- Krizhevsky, A., et al., ImageNet Classification with Deep Convolutional Neural Networks, NeurIPS, 2012
- Improved LeNet
- More layers: 7 hidden, 650k neurons, 60M parameters
- Dataset: 1M images , 1K classes
- GPU (50x faster than CPU); training time 6 days (2 GPU)
- DropOut
- winner of ILSVRC 2012; top-5 error = 16.4% (next result was 26.2%)

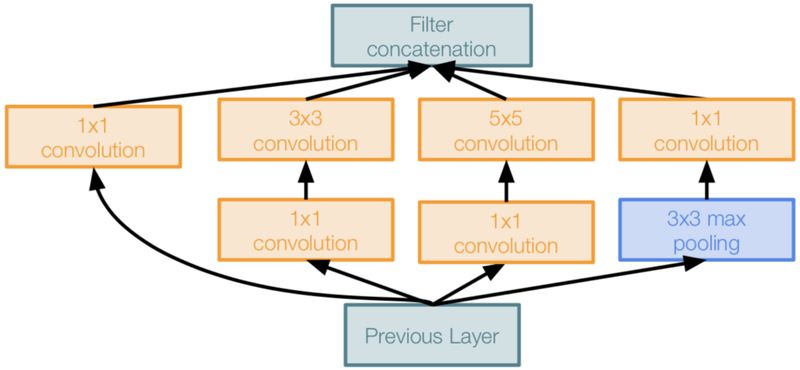
VGG (2014)

- K. Simonyan, Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR, 2015
- VGG16, VGG19 - first **deep** ConvNets
- VGG16 - 138M parameters
- training time: 2-3 weeks (4 GPU)
- ILSVRC 2014; top-5 error 7.3%

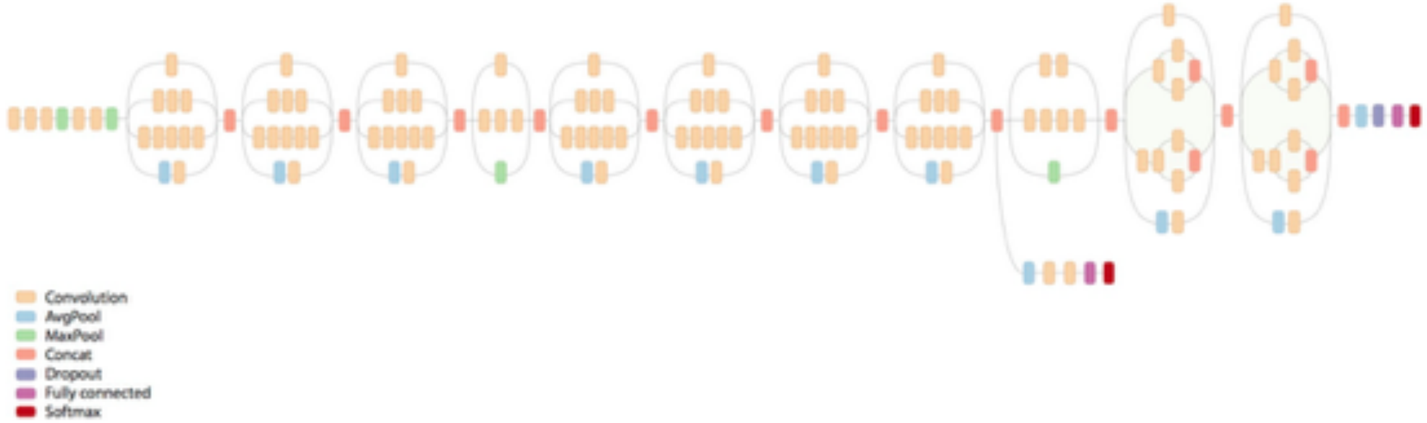
GoogleNet / Inception (2014)

“We need to go deeper”

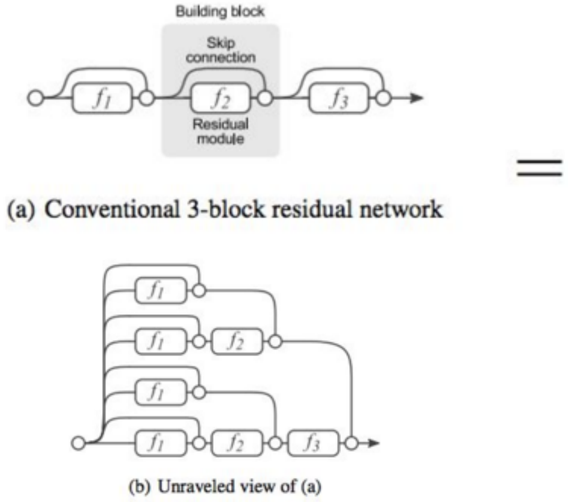
- Szegedy, C., et al., Going Deeper with Convolutions
- Inception V1
- winner of ILSVRC 2014 (top-5 error: 6.67%)
- 22 layers**, 25M parameters**
- training time: 2 weeks (8 GPU)
- **BatchNormalization**, **RMSProp**
- Inception V3
- ILSVRC 2014, top-5 error: 5.1%



Inception module with dimension reduction



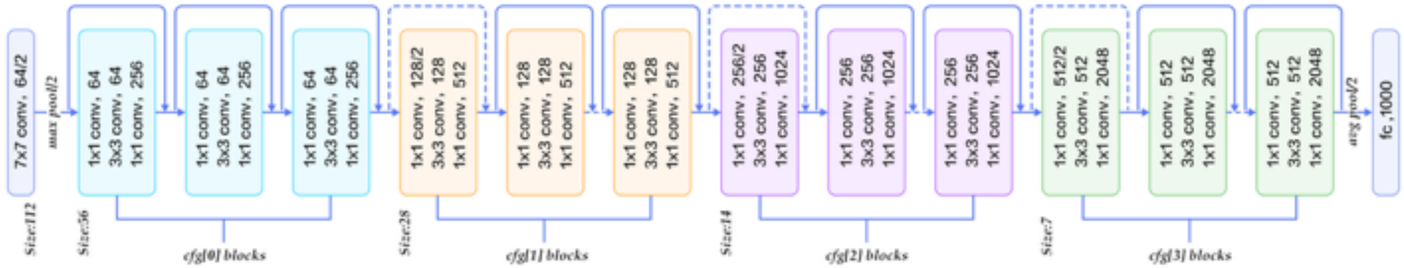
ResNet (2015)



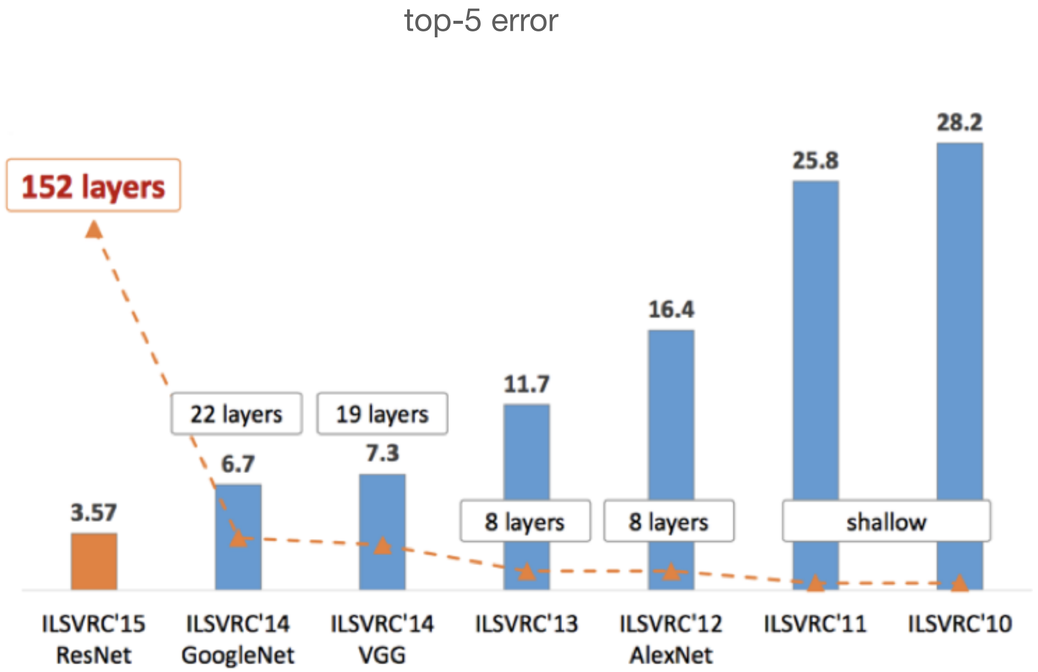
Andreas Veit, Michael Wilber, Serge Belongie, Residual Networks Behave Like Ensembles of Relatively Shallow Networks, arxiv 2016

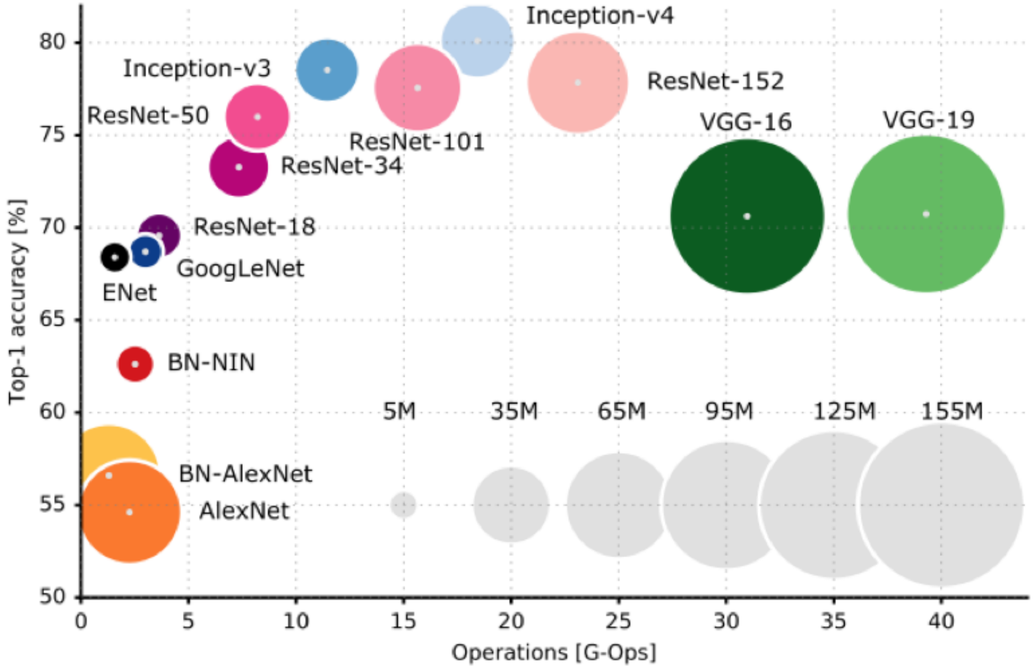
- ResNet152 (152 layers, 60M parameters)
- He, K., Zhang, X., Ren, S., & Sun, J., Deep Residual Learning for Image Recognition (pp. 770–778). Presented at the 2016 IEEE Conference on Computer Vision and Pattern Recognition (**CVPR**), 2016
- winner of ILSVRC 2015 - top-5 error:
  - 4.5% - single model
  - 3.6% - ensembled model
- innovative “**resblocks**” (shortcuts)
- training time: 2-3 weeks (8 GPU)

ResNet50



ILSVRC - results





torchvision

models

- torchvision.models provides a number of **pre-trained** models
- These models have been trained on the ImageNet dataset
- The models can be used for classification, segmentation, detection, etc.
- The models can be used for transfer learning
- see https://pytorch.org/vision/stable/models.html

Classification

The following classification models are available, with or without pre-trained weights:

- AlexNet
- ConvNeXt
- DenseNet
- EfficientNet
- EfficientNetV2
- GoogLeNet
- Inception V3
- MaxVit
- MNASNet
- MobileNet V2
- MobileNet V3
- RegNet
- ResNet
- ResNeXt
- ShuffleNet V2
- SqueezeNet
- SwinTransformer
- VGG
- VisionTransformer
- Wide ResNet

Table of all available classification weights

Accuracies are reported on ImageNet-1K using single crops:

Weight	Acc@1	Acc@5	Params	GFLOPS	Recipe
AlexNet_Weights.IMAGENET1K_V1	56.522	79.066	61.1M	0.71	link
ConvNeXt_Base_Weights.IMAGENET1K_V1	84.062	96.87	88.6M	15.36	link
ConvNeXt_Large_Weights.IMAGENET1K_V1	84.414	96.976	197.8M	34.36	link
ConvNeXt_Small_Weights.IMAGENET1K_V1	83.616	96.65	50.2M	8.68	link
ConvNeXt_Tiny_Weights.IMAGENET1K_V1	82.52	96.146	28.6M	4.46	link
DenseNet121_Weights.IMAGENET1K_V1	74.434	91.972	8.0M	2.83	link
DenseNet161_Weights.IMAGENET1K_V1	77.138	93.56	28.7M	7.73	link
DenseNet169_Weights.IMAGENET1K_V1	75.6	92.806	14.1M	3.36	link
DenseNet201_Weights.IMAGENET1K_V1	76.896	93.37	20.0M	4.29	link
EfficientNet_B0_Weights.IMAGENET1K_V1	77.692	93.532	5.3M	0.39	link
EfficientNet_B1_Weights.IMAGENET1K_V1	78.642	94.186	7.8M	0.69	link
EfficientNet_B1_Weights.IMAGENET1K_V2	79.838	94.934	7.8M	0.69	link
EfficientNet_B2_Weights.IMAGENET1K_V1	80.608	95.31	9.1M	1.09	link
EfficientNet_B3_Weights.IMAGENET1K_V1	82.008	96.054	12.2M	1.83	link
EfficientNet_B4_Weights.IMAGENET1K_V1	83.384	96.594	19.3M	4.39	link
EfficientNet_B5_Weights.IMAGENET1K_V1	83.444	96.628	30.4M	10.27	link
EfficientNet_B6_Weights.IMAGENET1K_V1	84.008	96.916	43.0M	19.07	link
EfficientNet_B7_Weights.IMAGENET1K_V1	84.122	96.908	66.3M	37.75	link
EfficientNet_V2_L_Weights.IMAGENET1K_V1	85.808	97.788	118.5M	56.08	link
EfficientNet_V2_M_Weights.IMAGENET1K_V1	85.112	97.156	54.1M	24.58	link
EfficientNet_V2_S_Weights.IMAGENET1K_V1	84.228	96.878	21.5M	8.37	link
GoogLeNet_Weights.IMAGENET1K_V1	69.778	89.53	6.6M	1.5	link
Inception_V3_Weights.IMAGENET1K_V1	77.294	93.45	27.2M	5.71	link
MNASNet0_5_Weights.IMAGENET1K_V1	67.734	87.49	2.2M	0.1	link
MNASNet0_75_Weights.IMAGENET1K_V1	71.18	90.496	3.2M	0.21	link
MNASNet1_0_Weights.IMAGENET1K_V1	73.456	91.51	4.4M	0.31	link
MNASNet1_3_Weights.IMAGENET1K_V1	76.506	93.522	6.3M	0.53	link
MaxVit_T_Weights.IMAGENET1K_V1	83.7	96.722	30.9M	5.56	link
MobileNet_V2_Weights.IMAGENET1K_V1	71.878	90.286	3.5M	0.3	link
MobileNet_V2_Weights.IMAGENET1K_V2	72.154	90.822	3.5M	0.3	link
MobileNet_V3_Large_Weights.IMAGENET1K_V1	74.042	91.34	5.5M	0.22	link
MobileNet_V3_Large_Weights.IMAGENET1K_V2	75.274	92.566	5.5M	0.22	link

CNNs

Overview & Best Practices

- What Are CNNs For?
    - Primarily used for image and video analysis, also effective in:
    - Object detection
    - Facial recognition
    - Image classification & segmentation
    - Medical imaging
- Work by learning spatial hierarchies through filters (kernels)

CNNs

Choosing the Kernel Size

- Common sizes: 3x3, 5x5, 7x7 (for very early layers)
- Best Practice: Use stacked 3×3 filters instead of a single large one
  - Stacked 3×3s preserve more non-linearity and reduce parameters
- Larger kernels can be used for aggressive downsampling or to capture more global context

CNNs

Choosing the Kernel Size

- Kernel count (filters per layer): start with 16–32, double in deeper layers
- Use stride 1 and padding=`same` to preserve spatial dimensions early on
- End with a Global Average Pooling + Dense layer for classification

CNNs

How Many Layers

- Depends on the task complexity:
  - Simple tasks: 3–10 layers
  - State-of-the-art models (ResNet, VGG, etc.): 20–100+ layers
- Deep networks use residual connections (ResNets) to avoid vanishing gradients

CNNs

Best Practices

- Start small: Begin with a shallow network, then scale
- ReLU activation is a standard; consider Leaky ReLU or ELU if dead neurons arise
- Use Batch Normalization to stabilize and accelerate training
- Dropout helps with overfitting (0.25–0.5 commonly used)
- Pooling layers (MaxPooling) reduce spatial size and computation
- Always monitor overfitting — use validation loss and early stopping
- Use data augmentation for image tasks (flipping, cropping, rotation)

Feature selector and classifier

Example: AlexNet

- ALEXNET

torchvision.models.alexnet(*, *weights: Optional[AlexNet_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → AlexNet [SOURCE]

AlexNet model architecture from One weird trick for parallelizing convolutional neural networks.

• NOTE

AlexNet was originally introduced in the ImageNet Classification with Deep Convolutional Neural Networks paper. Our implementation is based instead on the "One weird trick" paper above.

Parameters:

- **weights** (`AlexNet_Weights`, optional) – The pretrained weights to use. See `AlexNet_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.squeezenet.AlexNet` base class. Please refer to the source code for more details about this class.

see https://github.com/pytorch/vision/blob/main/torchvision/models/alexnet.py

```python
class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000, dropout: float = 0.5) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Feature selector and classifier

Example: VGG

VGG16

torchvision.models.vgg16(*, *weights: Optional[VGG16_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → VGG [SOURCE]

VGG-16 from Very Deep Convolutional Networks for Large-Scale Image Recognition.

Parameters:

- **weights** (`VGG16_Weights`, optional) – The pretrained weights to use. See `VGG16_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.vgg.VGG` base class. Please refer to the source code for more details about this class.

VGG19

torchvision.models.vgg19(*, *weights: Optional[VGG19_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → VGG [SOURCE]

VGG-19 from Very Deep Convolutional Networks for Large-Scale Image Recognition.

Parameters:

- **weights** (`VGG19_Weights`, optional) – The pretrained weights to use. See `VGG19_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.vgg.VGG` base class. Please refer to the source code for more details about this class.

```python
class VGG(nn.Module):
    def __init__(
        self, features: nn.Module, num_classes: int = 1000, init_weights: bool = True, dropout:
float = 0.5
    ) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
                    if m.bias is not None:
                        nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.BatchNorm2d):
                    nn.init.constant_(m.weight, 1)
                    nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, 0, 0.01)
                    nn.init.constant_(m.bias, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```