Convolutional Neural Networks II

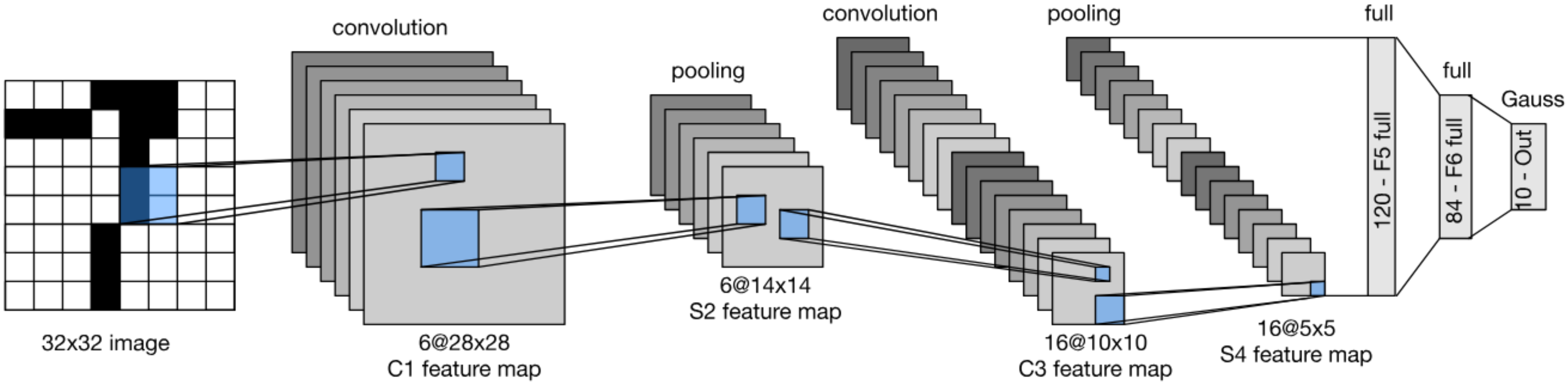
Zachary Lipton & Henry Chai

10701 — October 30th

email: zlipton@cmu.edu

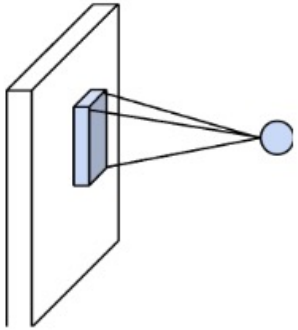
Recap

LeNet Architecture

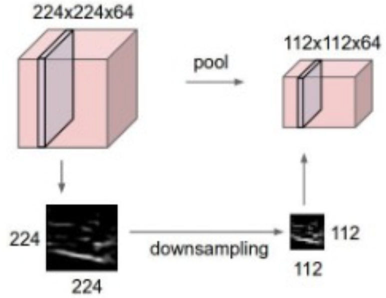


Basic Components of CNN Architectures

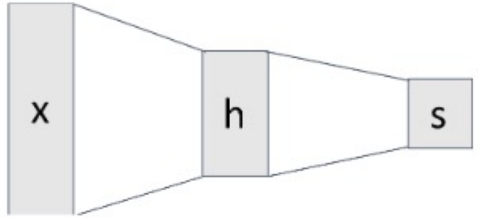
Convolution Layers



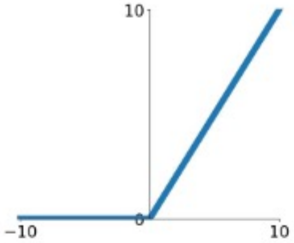
Pooling Layers



Fully-Connected Layers



Activation Function

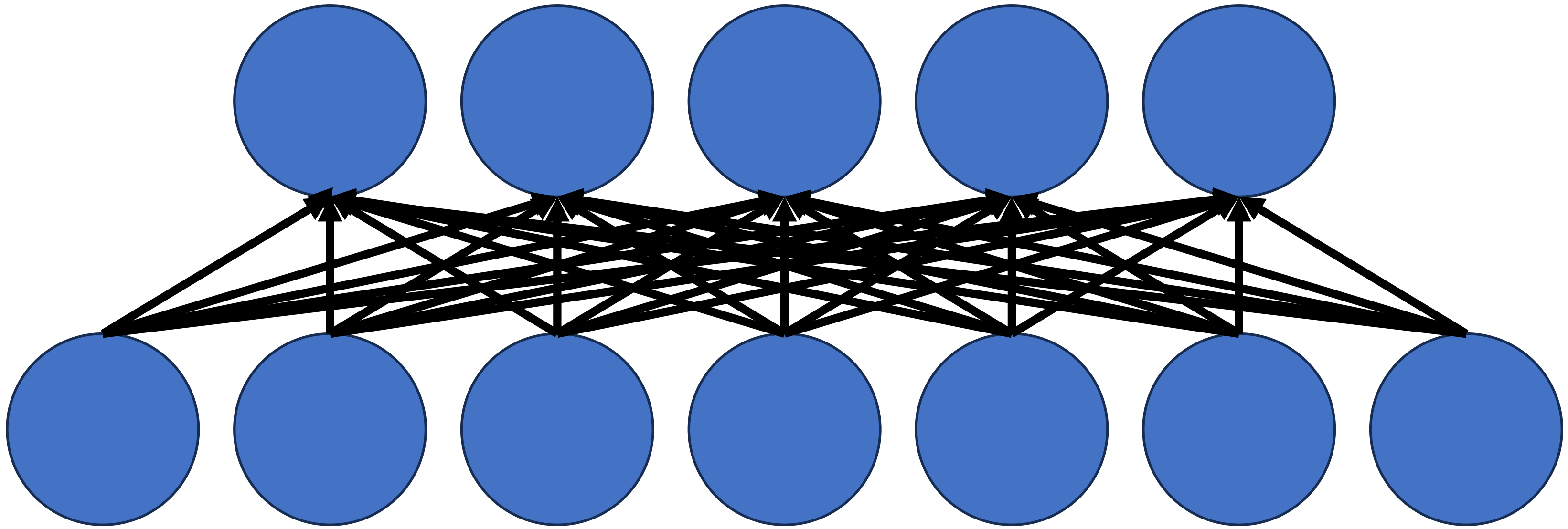


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

slide source: http://cs231n.stanford.edu/slides/2023/lecture_6.pdf

From Fully-Connected to Convolutional (1D)



$$\mathbf{h}' = W\mathbf{h}$$

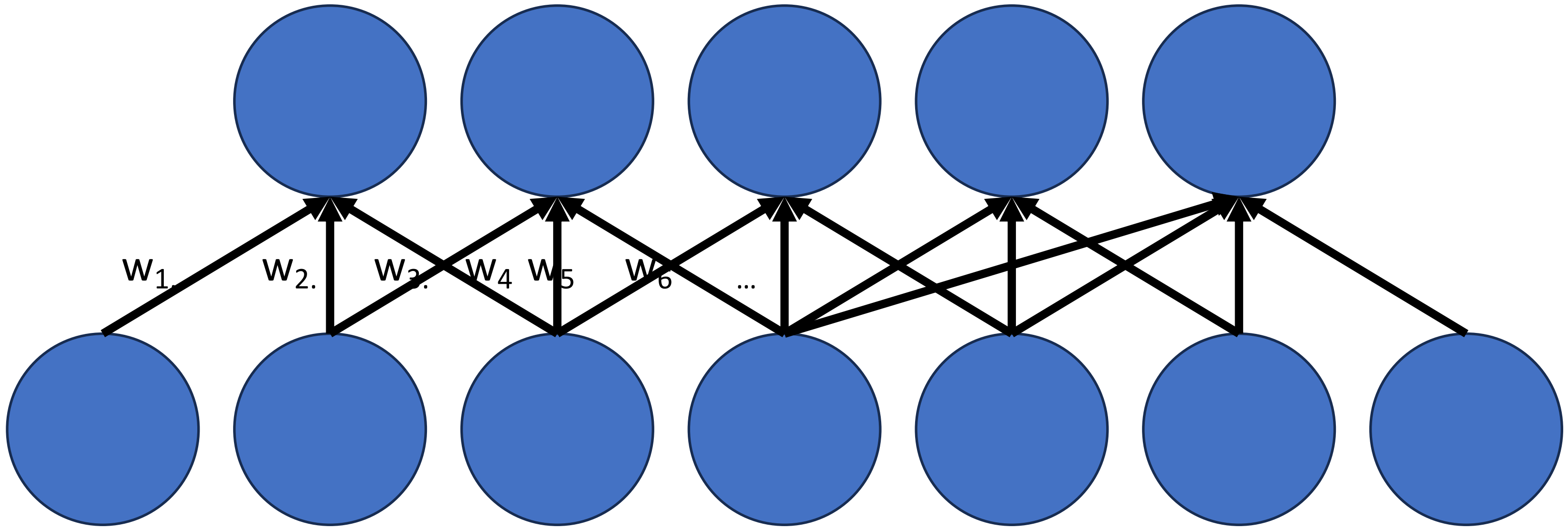
$$\mathbf{h} \in \mathbb{R}^n$$

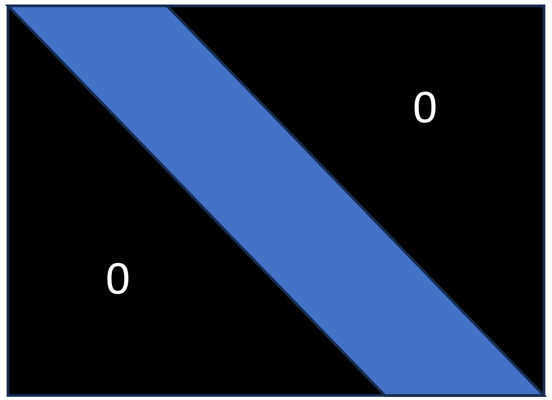
$$\mathbf{h}' \in \mathbb{R}^m$$

$$W \in \mathbb{R}^{m \times n}$$

- Fully connected, m * n params
- No account for spatial structure

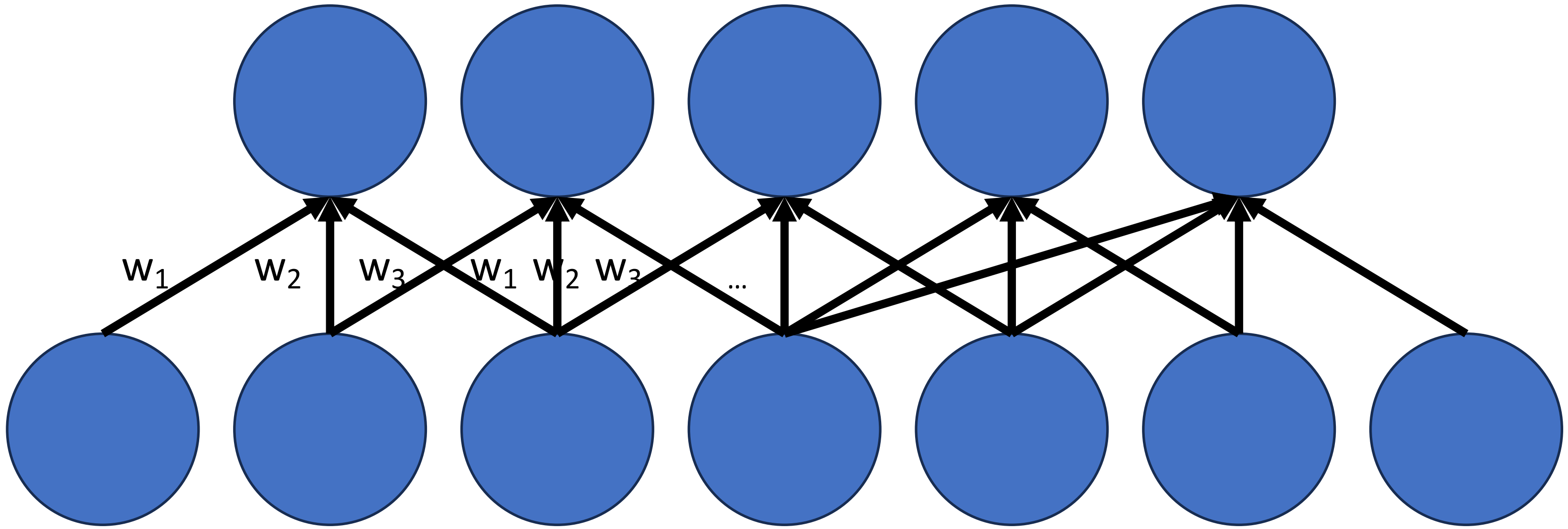
Step 1: Add Locality





- Locally connected, m * 3 params
- Spatial structure accounted for, but no invariance

Step 3: Invariance (via Weight Tying)



- Locally connected, weight tied, 3 params
- Spatial structure AND invariance accounted for

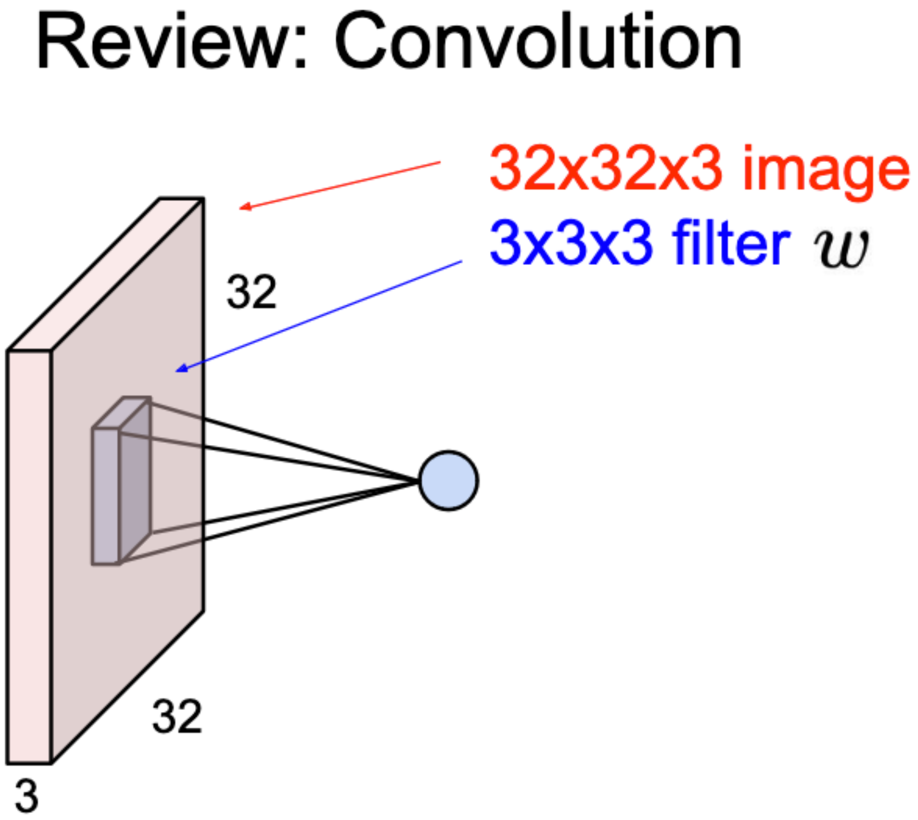
$$\mathbf{h}' = W * \mathbf{h}$$

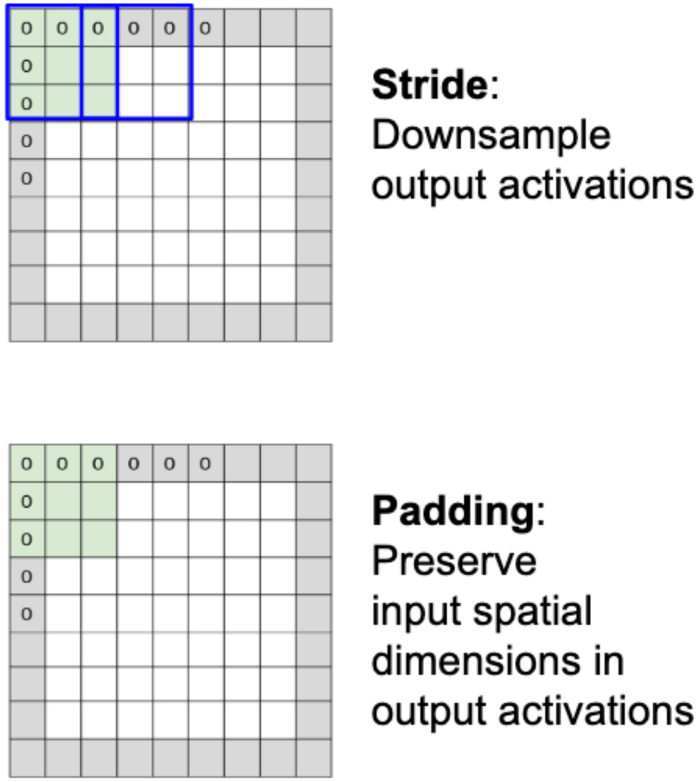
$$\mathbf{h} \in \mathbb{R}^n$$

$$\mathbf{h}' \in \mathbb{R}^m$$

$$W \in \mathbb{R}^3$$

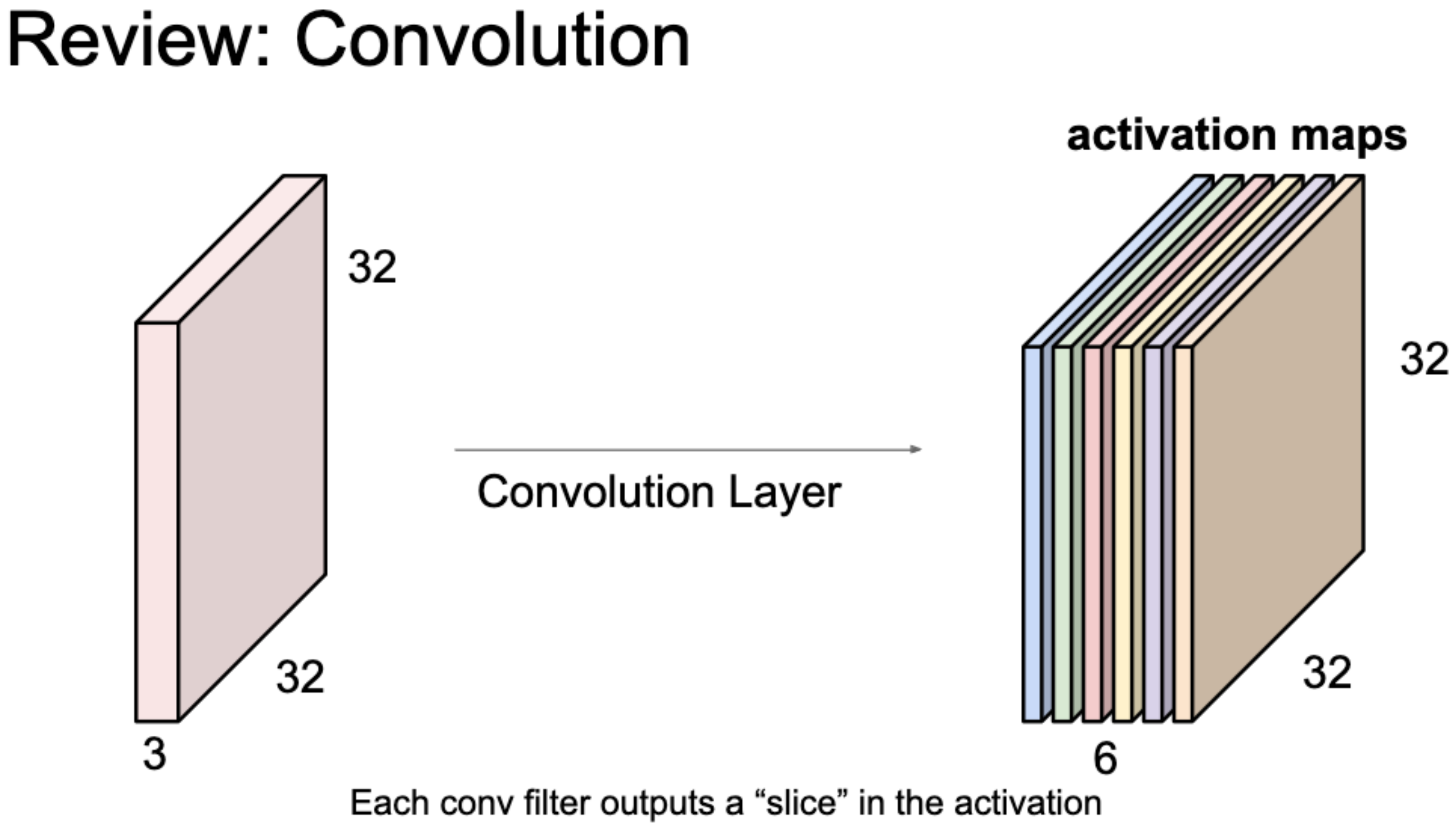
Lifting to 2D Convolutions on Image Input





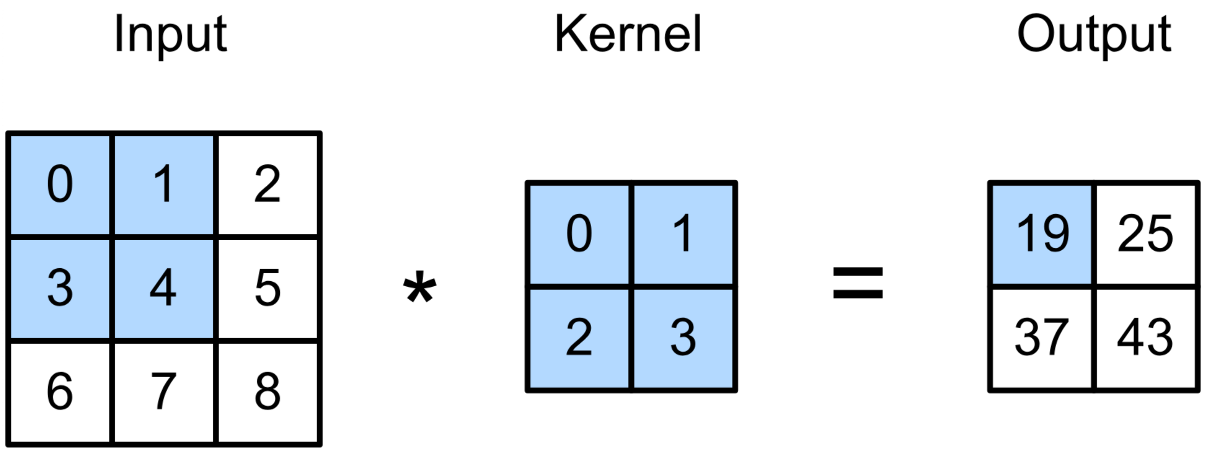
slide source: http://cs231n.stanford.edu/slides/2023/lecture_6.pdf

Multiple Kernels → Multiple Activation Maps

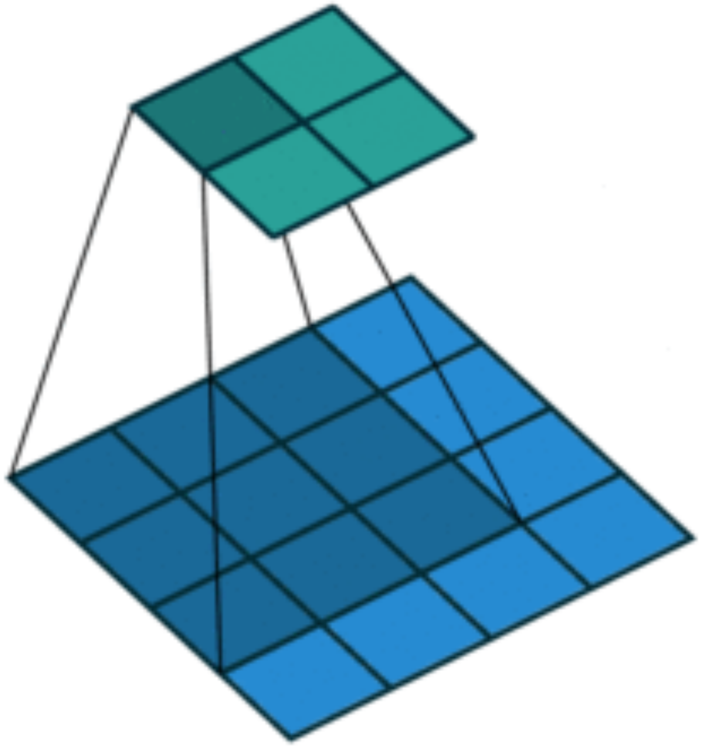


slide source: http://cs231n.stanford.edu/slides/2023/lecture_6.pdf

2-D "Convolution (Cross Correlation)



$$0\times0 + 1\times1 + 3\times2 + 4\times3 = 19,$$
$$1\times0 + 2\times1 + 4\times2 + 5\times3 = 25,$$
$$3\times0 + 4\times1 + 6\times2 + 7\times3 = 37,$$
$$4\times0 + 5\times1 + 7\times2 + 8\times3 = 43.$$



(vdumoulin@ Github)

2-D Convolution Layer

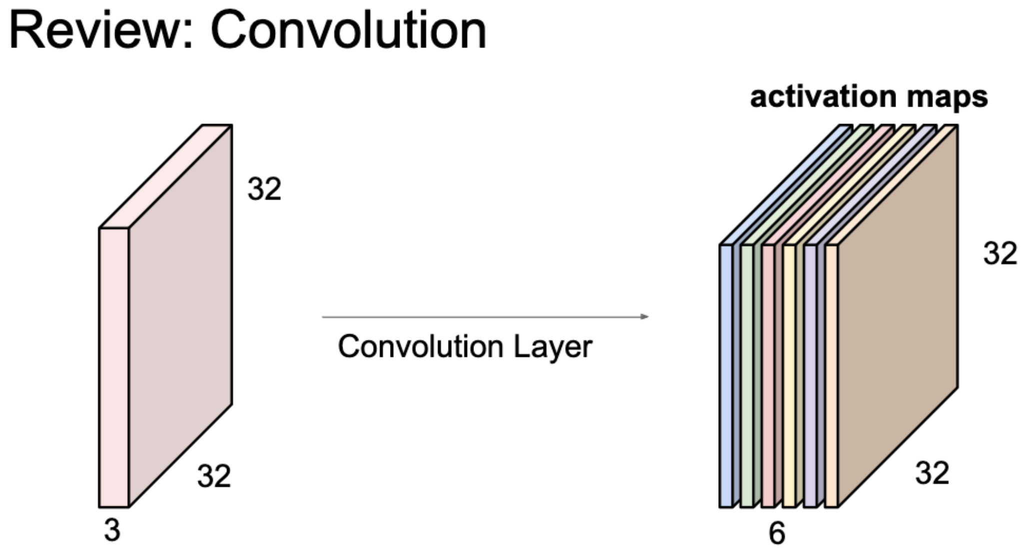
- $\mathbf{X}$: $n_h \times n_w$ input matrix
- $\mathbf{W}$: $k_h \times k_w$ kernel matrix
- b: scalar bias
- $\mathbf{Y}$: $(n_h - k_h + 1) \times (n_w. - k_w + 1)$ output matrix
- **W** and *b* are learnable parameters

$$\mathbf{Y} = \mathbf{X} \star \mathbf{W} + b$$

$$\begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \; * \; \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \; = \; \begin{array}{|c|c|}\hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$$

With Multiple Output Channels

- **X:** input volume (h x w x c_{in})
- **K:** 4d kernel (k_h, k_w, c_{in}, c_{out})
- **b**: one bias per output channel
- **Y:** output volume

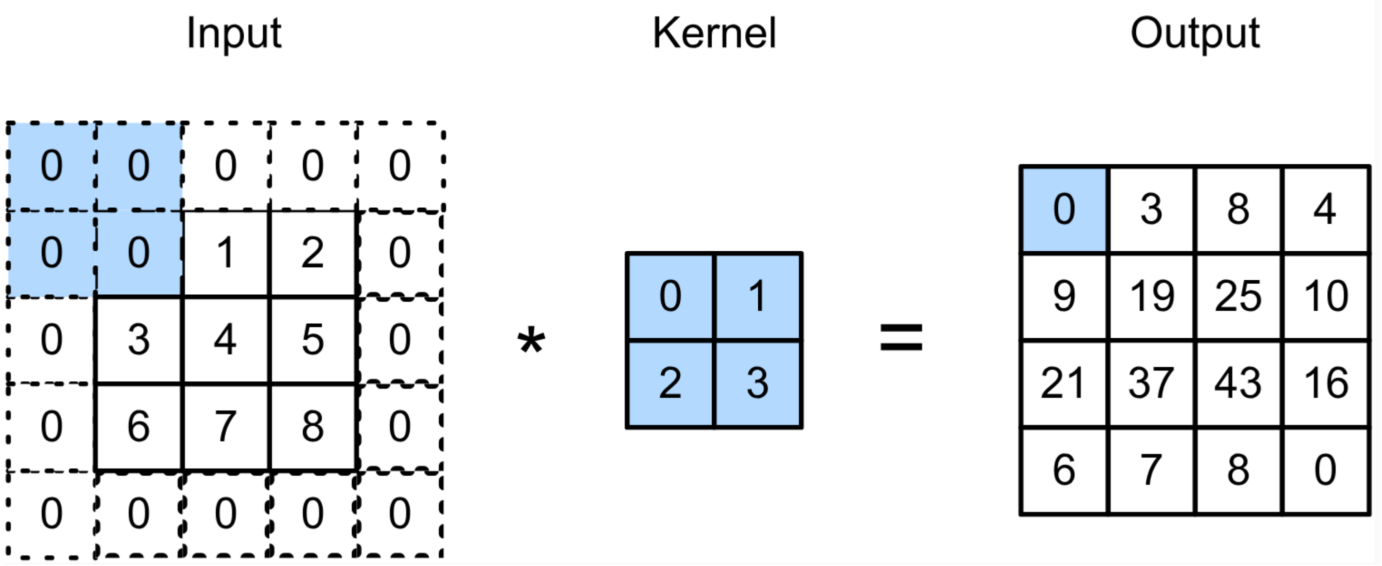


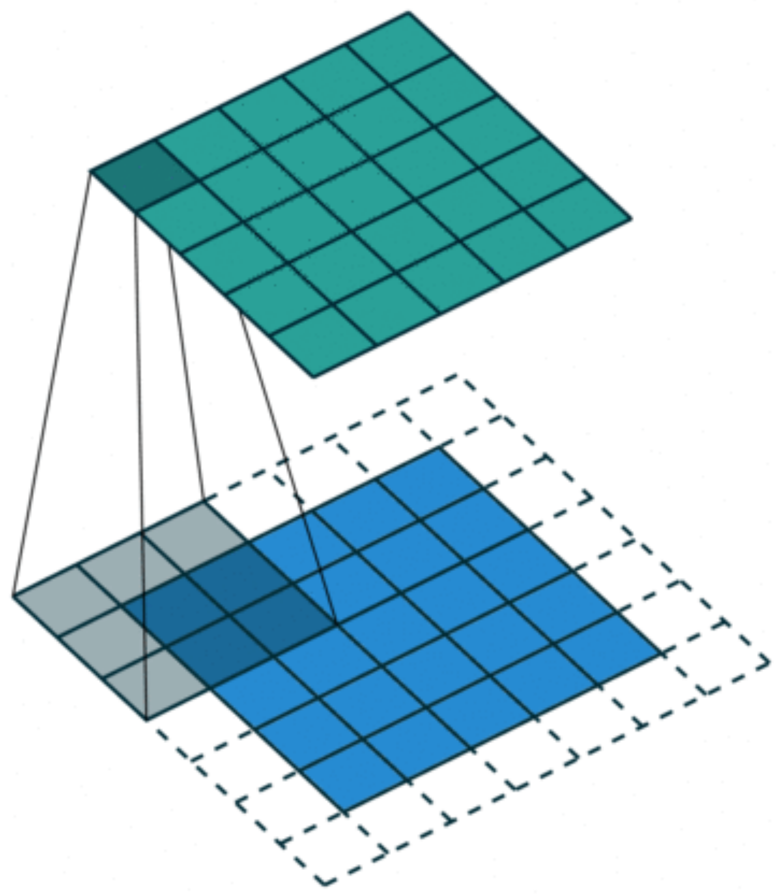
Each conv filter outputs a "slice" in the activation

slide source: http://cs231n.stanford.edu/slides/2023/lecture_6.pdf

Padding

Fills in rows/columns around input (with 0's)

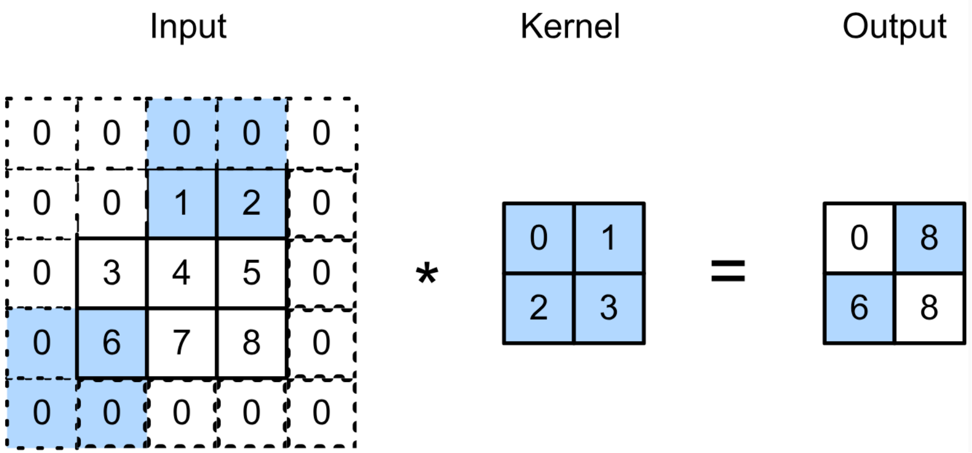




$$0\times0+0\times1+0\times2+0\times3=0$$

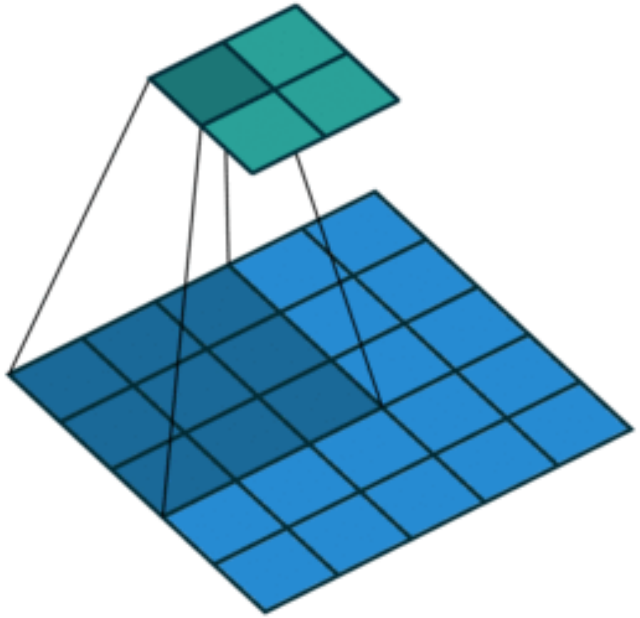
Strides – Skipping Spatial Locations in Conv

- Below:
  stride of 3 for height, 2 for width

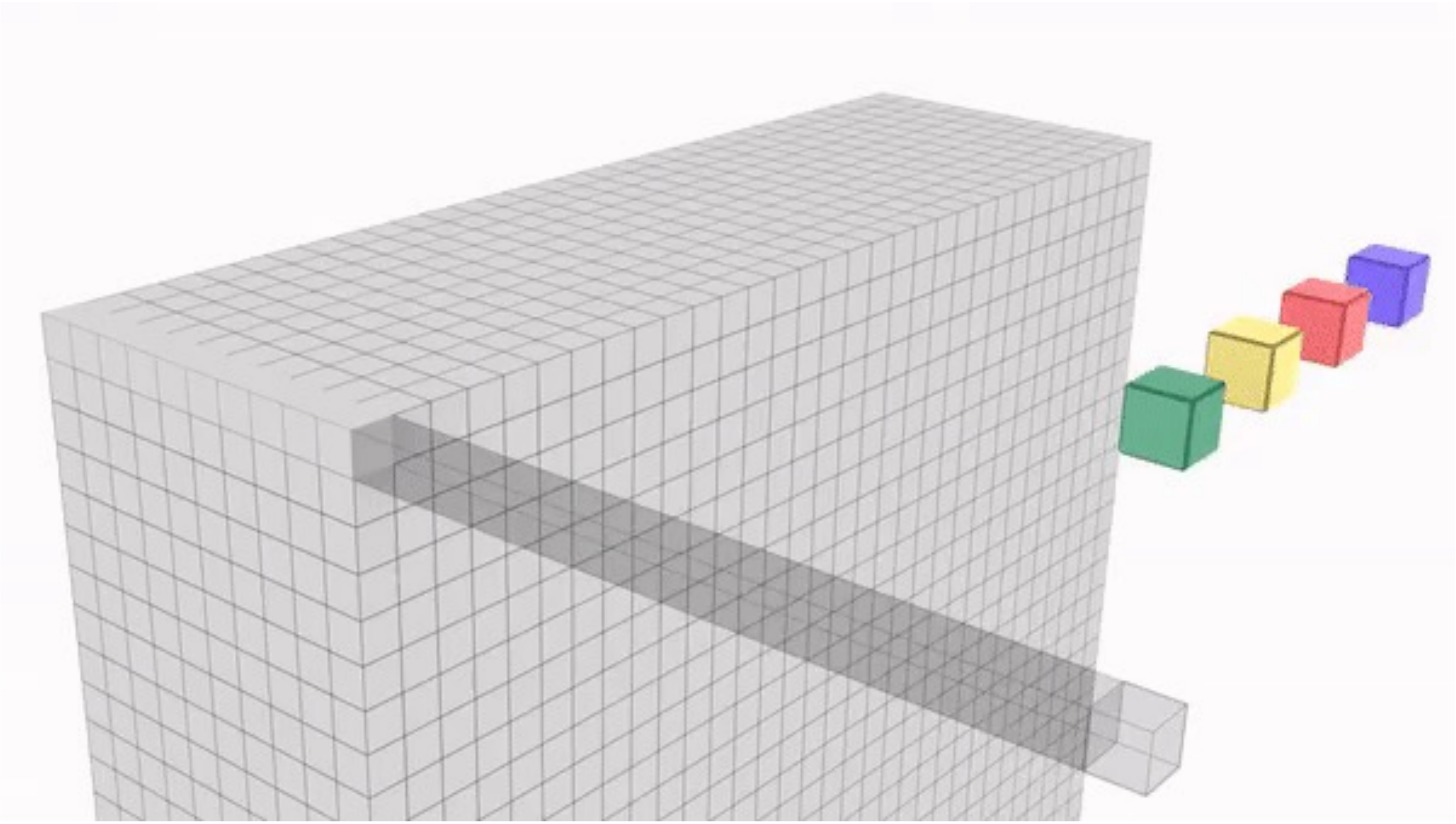


$$0{\times}0 + 0{\times}1 + 1{\times}2 + 2{\times}3 = 8$$
$$0{\times}0 + 6{\times}1 + 0{\times}2 + 0{\times}3 = 6$$

- Below:
  stride of 2 (= for height, width)

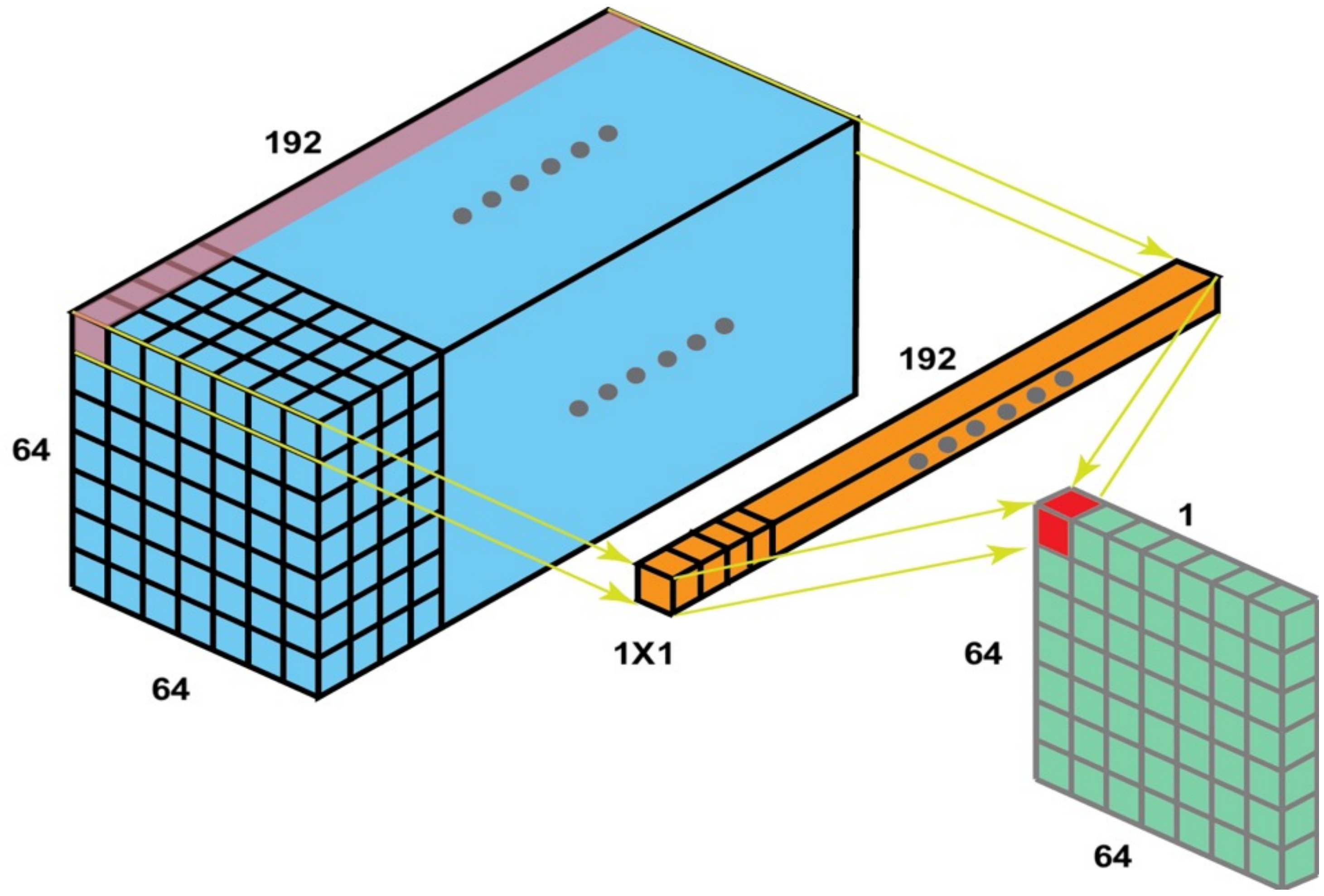


1x1 Point-wise Convolutions



Source: https://medium.com/hitchhikers-guide-to-deep-learning/6-introduction-to-deep-learning-with-computer-vision-3x3-is-a-lie-1x1-convolutions-9edd2baf7fd5

1x1 Convolutions

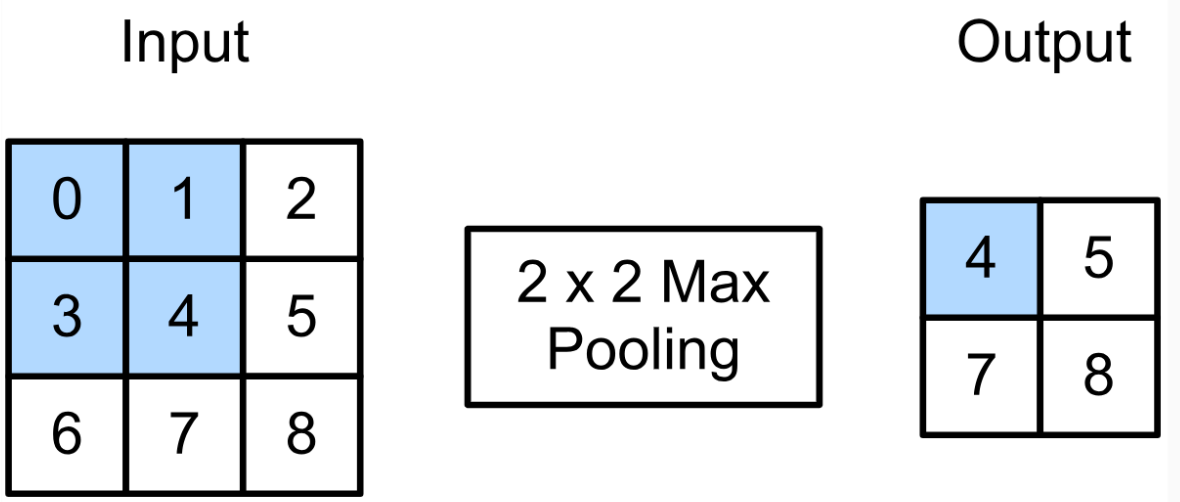


Convolutional Layer Summary

- Linear operations with few non-zero entries, tied weights
- Padding adds rows/columns to input dimension
- Strides skip spatial locations
- Spatial dimensions of output:
  (N – F) / stride + 1
- Convolutional layers can be applied on many dimensions
  - 1D (audio, text)
  - 2D (images, spectrograms)
  - 3D (MRI images, CT scans, video)
  - or even higher dimensions (4D convolutions for 3D video?)

2D Max Pooling

- Returns the maximal value in each sliding window
- Pooling windows of 2x2 most common

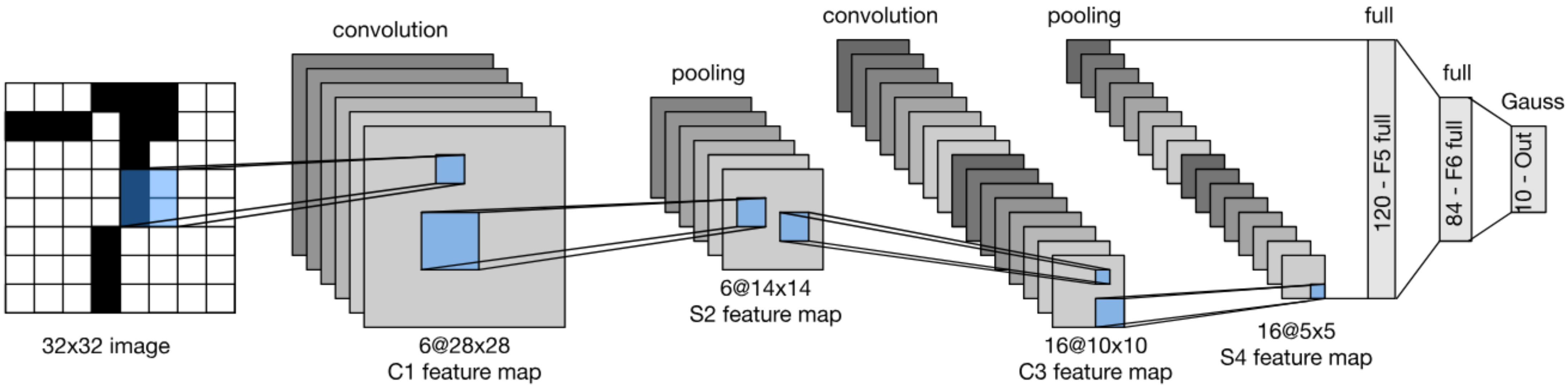


Exercise:

- Input: 100 x 100 x 3
  - →Conv (9x9), 48 filters, ReLU, Padding 0, Stride 1
  - →Conv (9x9), 96 filters, ReLU, Padding 4 (on both sides), Stride 1
  - →Conv (5x5), 192 filters, ReLU, Padding 0, Stride 2
  - →Max Pool (2x2), Stride 2
  - →Conv (1x1), 256 filters
  - → fully connected layer (100 outputs)

- What are the dimensions after each layer?
- What's the total parameter count?

LeNet Architecture

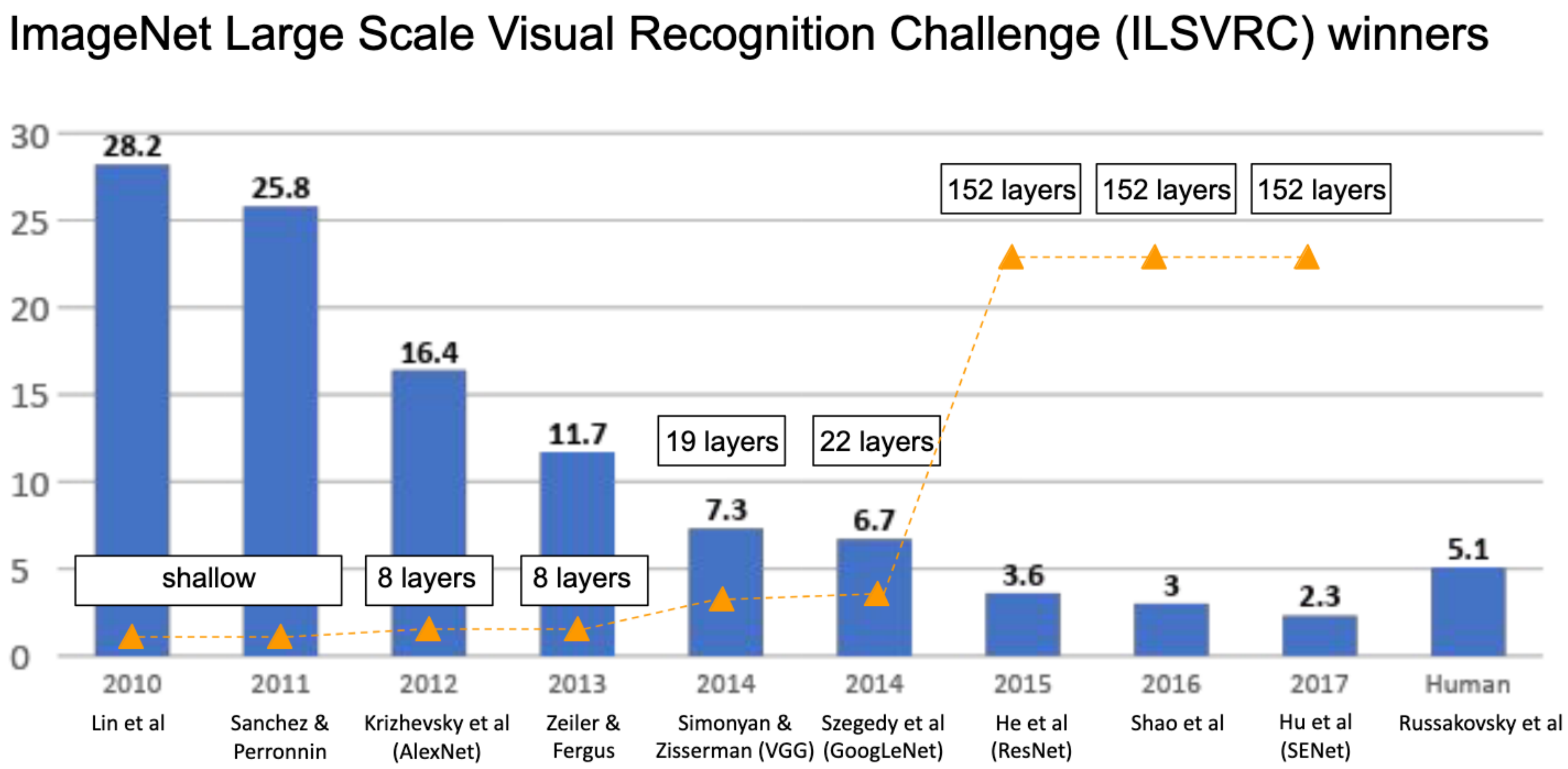


ConvNetJS

- https://cs.stanford.edu/people/karpathy/convnetjs/

Modern CNN Architectures

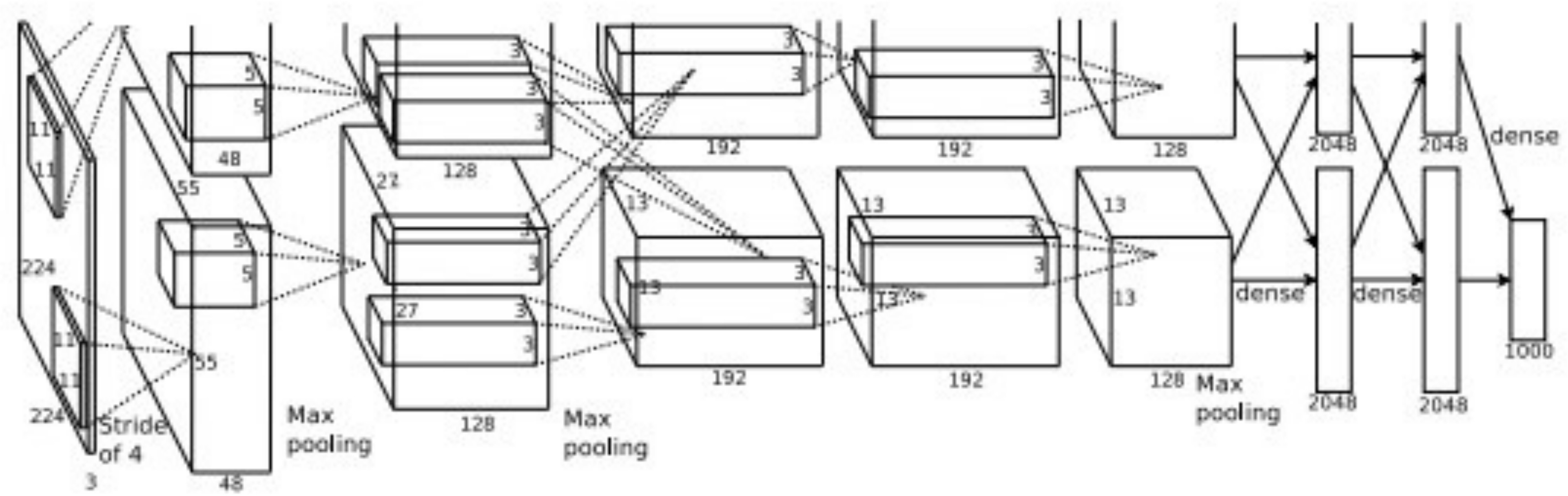
Progress in CNN Architecture via ILSVRC



Enduring Themes of CNN Design

- Convolutions replace FC layers, usually followed by activation fn
- Pooling layers between convolutional layers (dept—preserving)
- Max pooling favored (over mean pooling)
- Trends from input to output:
  - Spatial subsampling
  - Increased channel dimension
- ***Why?***

AlexNet



[“AlexNet” — (Krizhevsky, Sutskever, Hinton 2012)]

AlexNet Architecture

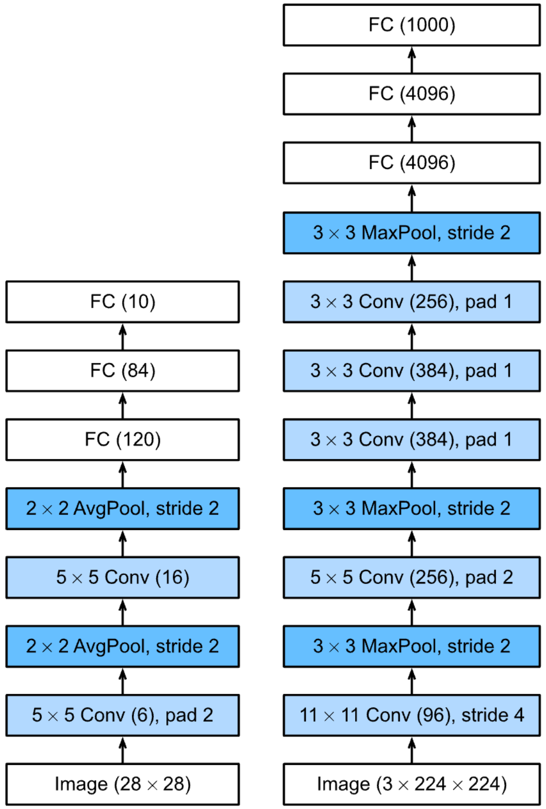
Input: 224 x 224 x 3 images

Architecture:

Conv1 :	(11x11), 96 filters, stride 4, ReLU
Pool1:	(3x3), max, stride 2
Norm1:	local response normalization
Conv2:	(5x5), 256 filters, stride 2, ReLU
Pool2:	(3x3), max, stride 2
Norm2:	local response normalization
Conv3:	(3x3), 384 filters, stride 1, pad 1, ReLU
Conv4:	(3x3), 384 filters, stride 1, pad 1, ReLU
Conv5:	(3x3), 256 filters, stride 1, pad 1, ReLU
Pool3:	(3x3), max, stride 2
FC6:	4096 neurons, ReLU
FC7:	4096 neurons, ReLU
FC8:	1000 neurons, Softmax

Comparison to LeNet

- From 5 → 8 layers
- Input 224x224 (vs 28x28)
- Larger filters
- Higher-dimensional FC layers
- 1000 classes



AlexNet Implementation

PYTORCH MXNET JAX TENSORFLOW

```python
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

Other Training Techniques

- Data augmentation:
  - Translations and horizontal reflections
    (extracting random 224x224 patches from underlying 256x256 images)
  - Altering intensities of the RGB channels
- Dropout:
  - Applied with dropout probability .5 to fully connected layers during training
- Batch Size: 128
- Weight Decay: .0005
- Weight initialization: zero-mean Gaussian w std dev .01
- Optimizer: SGD with momentum of .9
- Schedule: divide learning rate by 10 each time learning stagnated

ZFNet (Zeiler & Fergus ILSVRC winner 2013)

Same general structure as AlexNet, refined hyperparameters

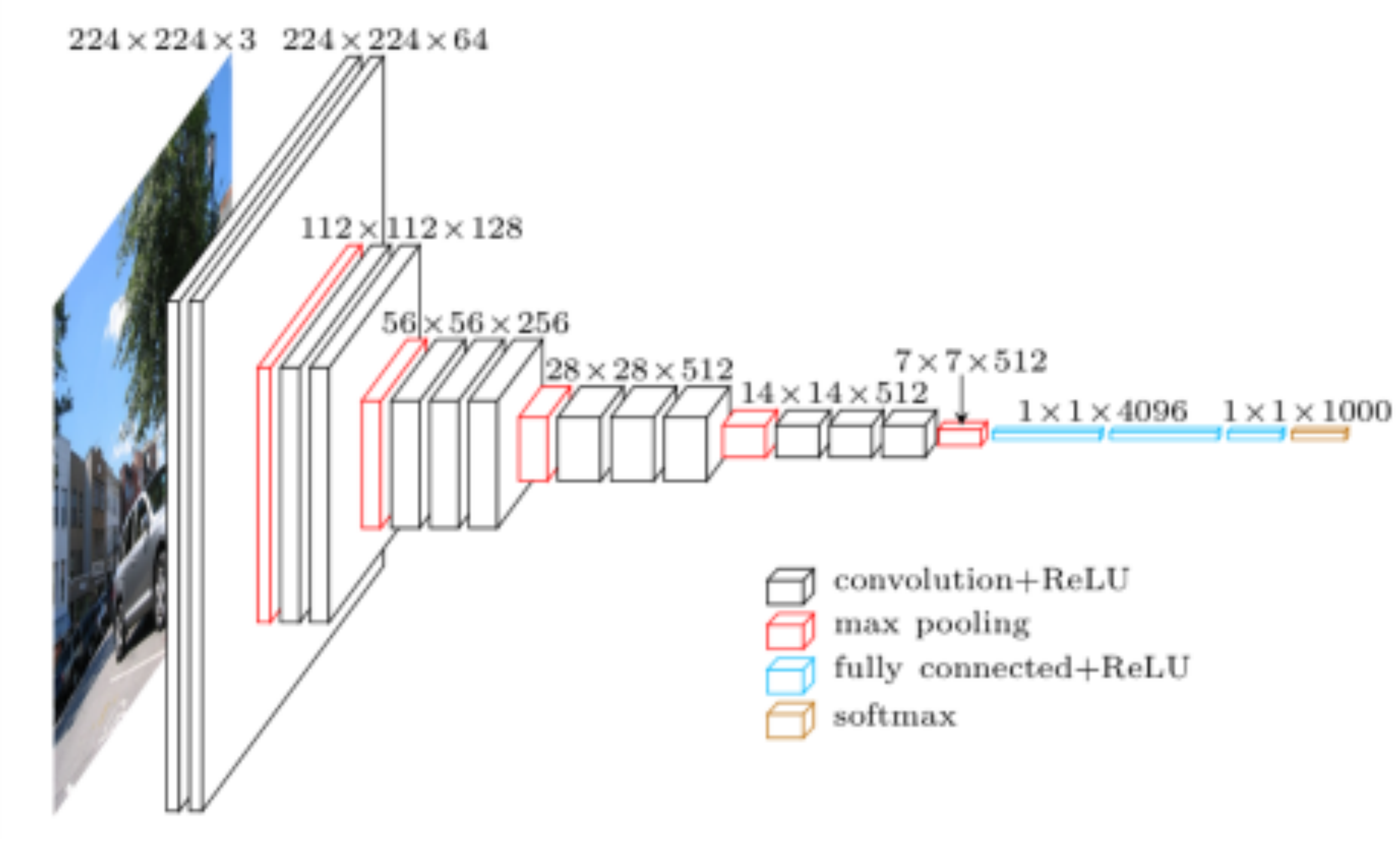
- **Conv1:** (11 x 11), stride 4 → **7x7, stride 2**
- **Conv3,4,5:** filter size (384, 484, 256) → **(512, 1024, 512)**
- **Top5 error:** 16.4% → **11.7%**

Key ideas:

- Visualization techniques to probe learned representations
- Retrain output layer, transfer representation, SOTA on Caltech101, Caltech256

https://arxiv.org/abs/1311.2901

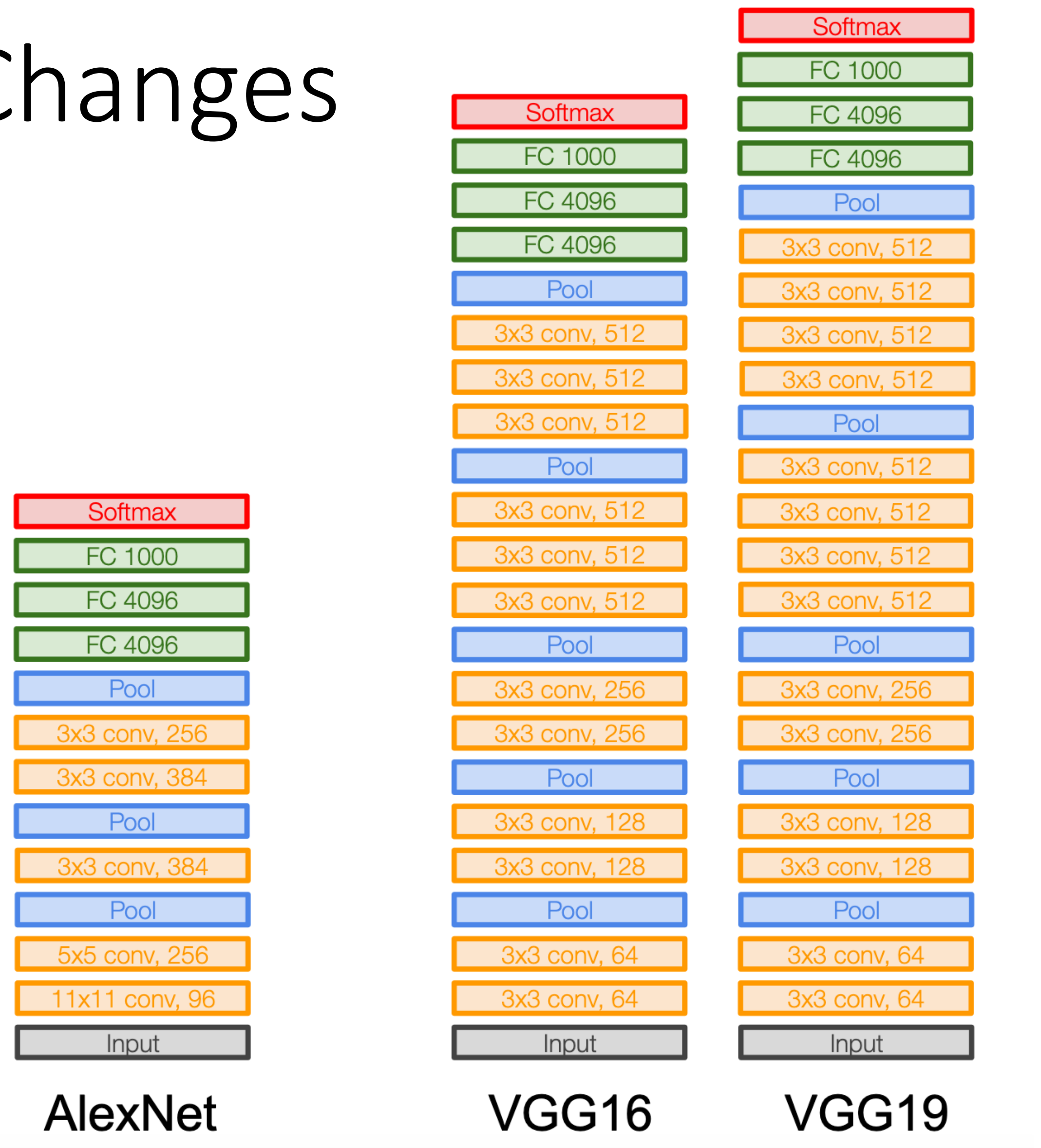
VGG



Simonyan & Zisserman, ICLR 2015

VGG: Key Architectural Changes

- Smaller filters, deeper network (8 layers → 16–19 layers)
- All convs: 3x3 filters, stride 1, pad 1
- All pooling 2x2 max-pool, stride 2
- Adopts motifs, repeated blocks: of 3 conv layers + 1 pool
- Intuition:
  - Stack of 3x3 conv layers has same receptive field as one 7x7 conv layer
  - More depth, more non-linearities, fewer parameters



Repeated Building Blocks

PYTORCH MXNET JAX TENSORFLOW

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

PYTORCH MXNET JAX TENSORFLOW

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

Batch Normalization Intuition

- Loss occurs at last layer
  - Last layers learn quickly
- Data is inserted at bottom layer
  - Bottom layers change — **everything** changes
  - Upper layers need to adjust to inputs of wildly different dynamic range
  - Slow convergence
- Original Intuition: “internal covariate shift” (Ioffe, Szegedy 2015)
- Distributional stability hypotheses refuted by Santurkar et al. 2018, alternative hypothesis for benefits: “smoothen optimization landscape”

What BatchNorm Does

- During training, normalize by batch-wise means & variances

$$x_{i+1} = \gamma \frac{x_i - \hat{\mu}_B}{\hat{\sigma}_B} + \beta$$

- Adds stochastic noise during training:
  - Random shift and scale (depends on minibatch!)
- Benefits tend not to be synergistic with Dropout (most ppl don't mix)
- Ideal minibatch size in range (64 to 256)
- At inference time, normalize using **fixed** batch stats:
  (running average of batch stats computed during training)

Batch Normalization

- Batch norm operation

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}.$$

- Means and variances computed on each minibatch:

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.$$

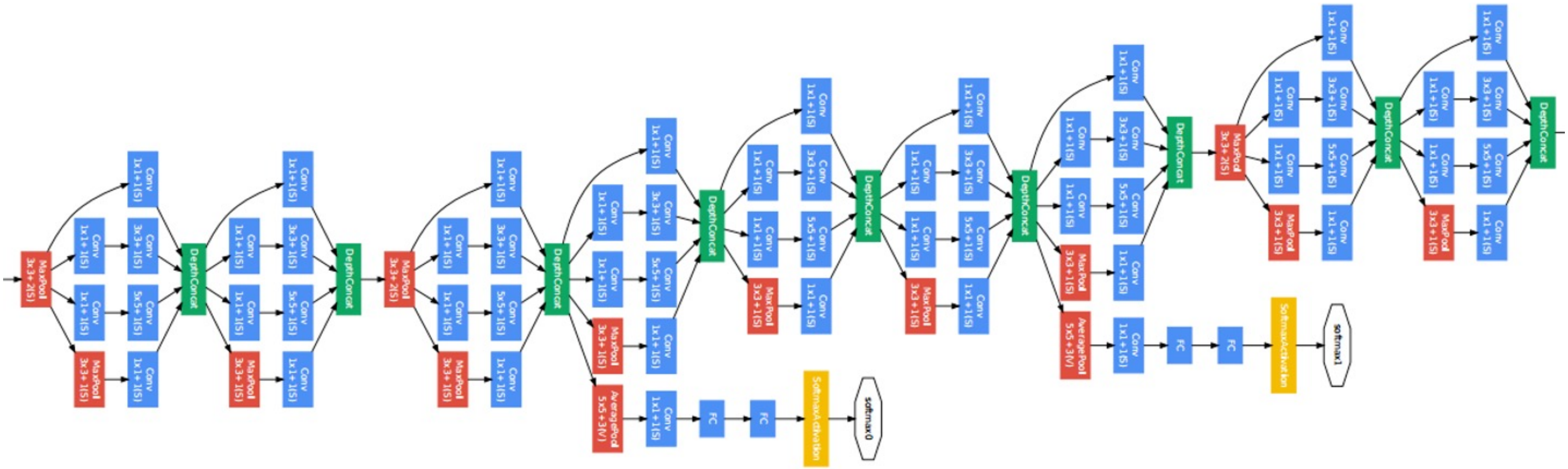
- The **γ** and **β** are learnable parameters (1 per channel), restore the expressive power of the model
- Elements of **γ** initialized at 1, **β** initialized at 0

BatchNorm Implementation

PYTORCH MXNET JAX TENSORFLOW

```python
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use is_grad_enabled to determine whether we are in training mode
    if not torch.is_grad_enabled():
        # In prediction mode, use mean and variance obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # In training mode, the current mean and variance are used
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = (1.0 - momentum) * moving_mean + momentum * mean
        moving_var = (1.0 - momentum) * moving_var + momentum * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.data, moving_var.data
```

GoogLeNet



[“Going Deeper with Convolutions” Szegedy et al. 2014]

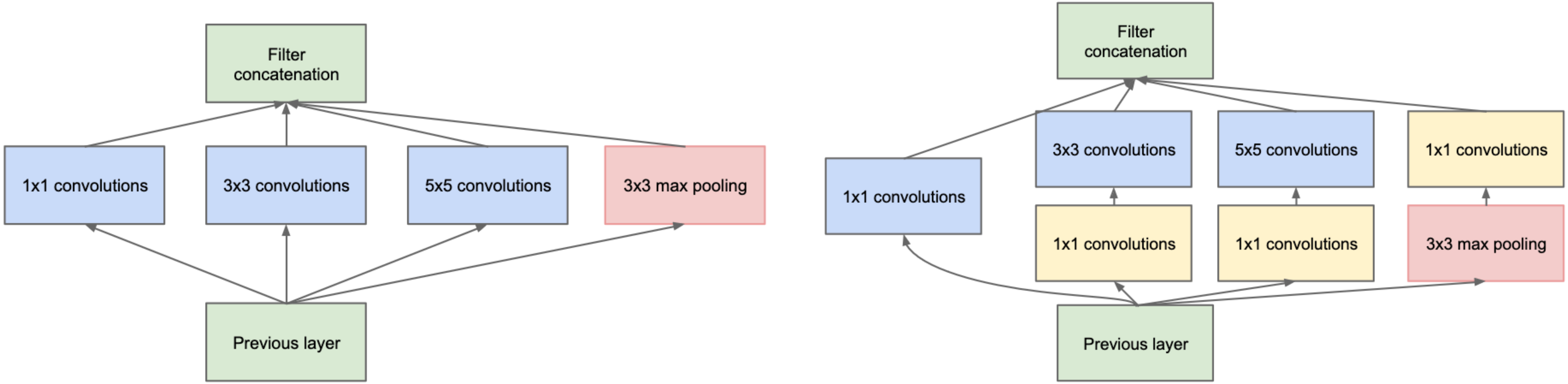
GoogLeNet

- 22 layers
- No fully connected layers
- Global average pooling before classification layer
- Only 5M parameters
- 12x less params than AlexNet
- Brought ILSVRC error down from 11.7% → 6.7%

Inception Blocks

- Problem: how to choose the right dimension for convolutional filter?
- Answer: don't choose, just pick them all!
- Tricks:
  - Blow up depth dimension with high-channel conv filters
  - Reduce it back down via 1x1 conv "bottleneck" layers

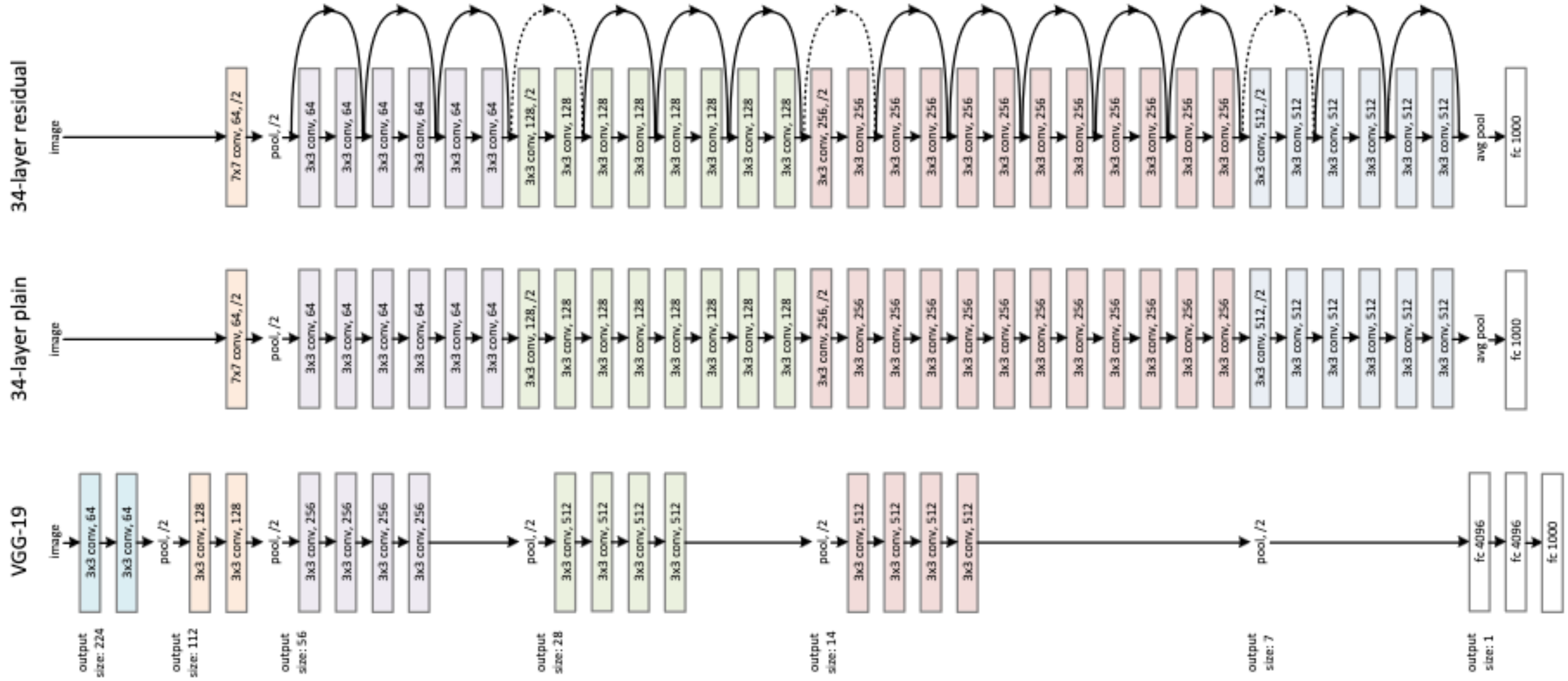
Why add the 1x1 Convolutions?



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

Residual Networks



["Deep Residual Learning for Image Recognition" He et al. 2015]

ResNet

- Inspired a “revolution of depth”
- Brought down error below “human” performance on ILSVRC 3.57%
- Winning networks boasted 152 layers
- Insight: for deep nets, identity function should be easy to learn

Can we go deeper w “vanilla” conv layers?

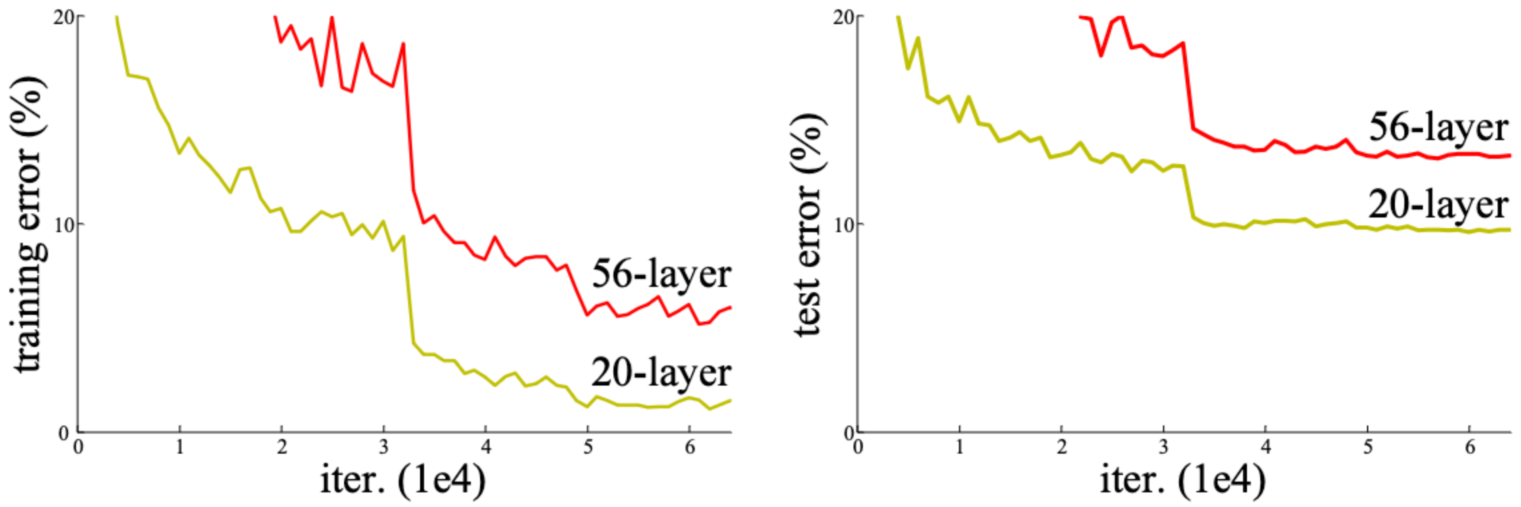
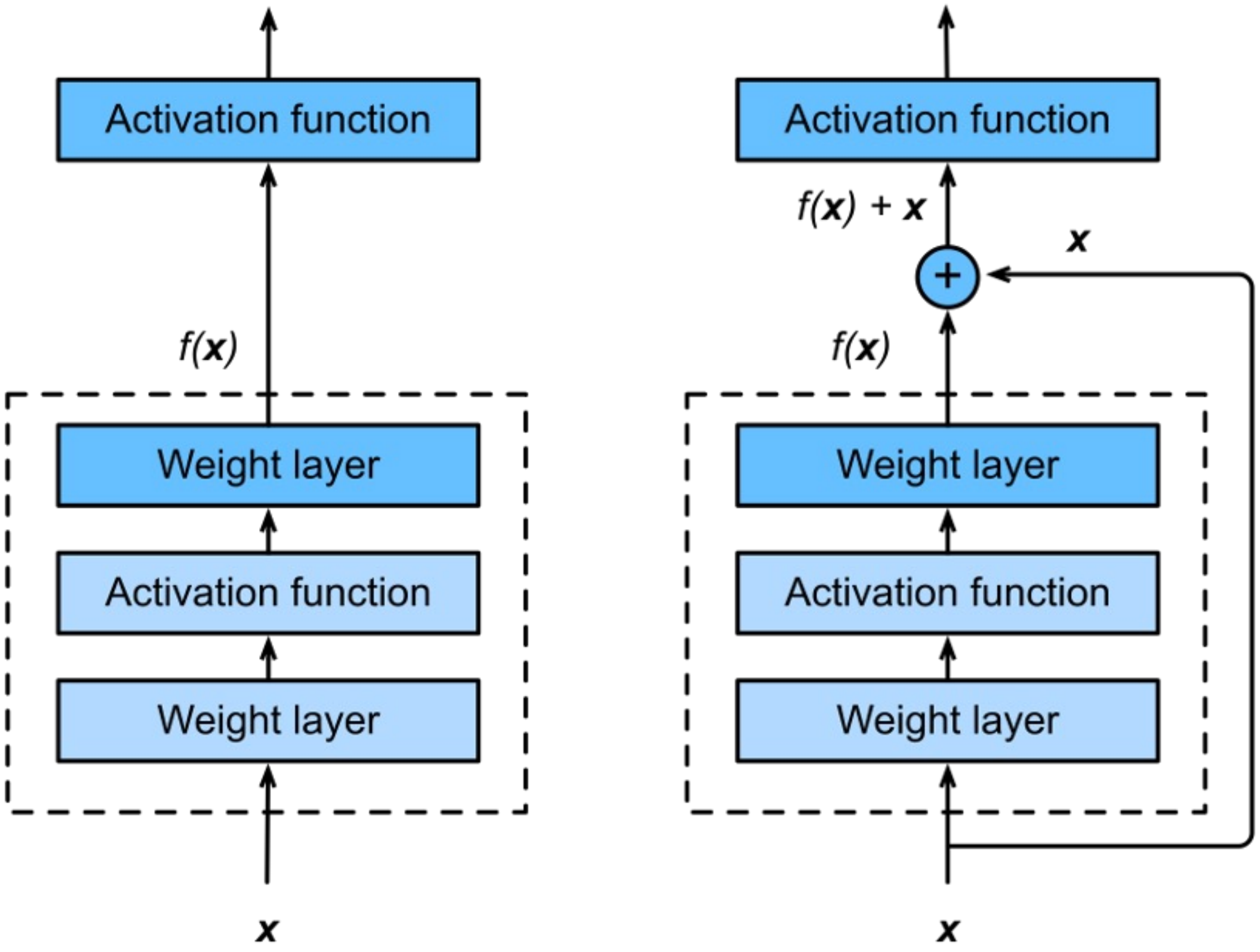
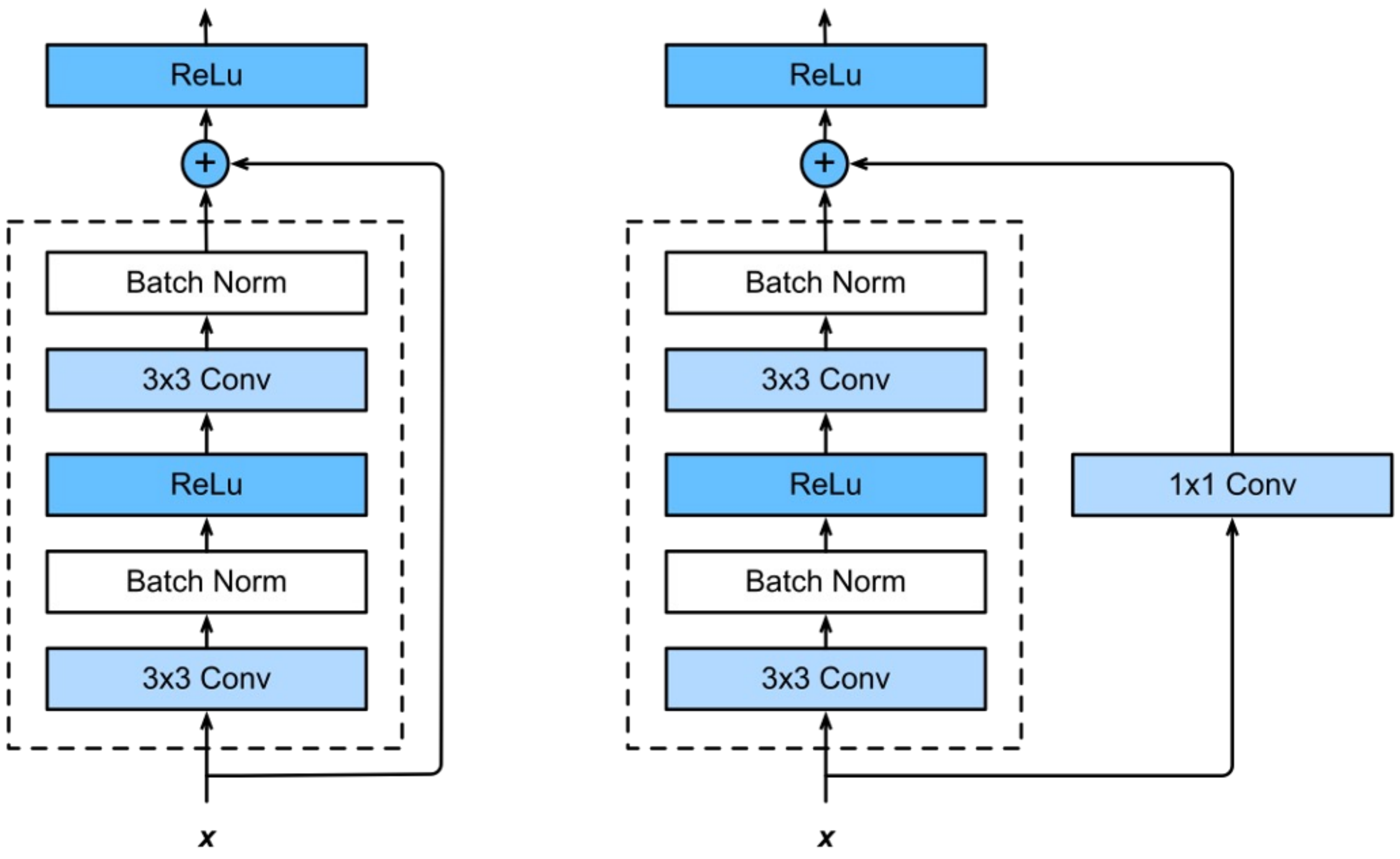


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Idea → Just learn the residual



Residual Layers Applied to Convnet



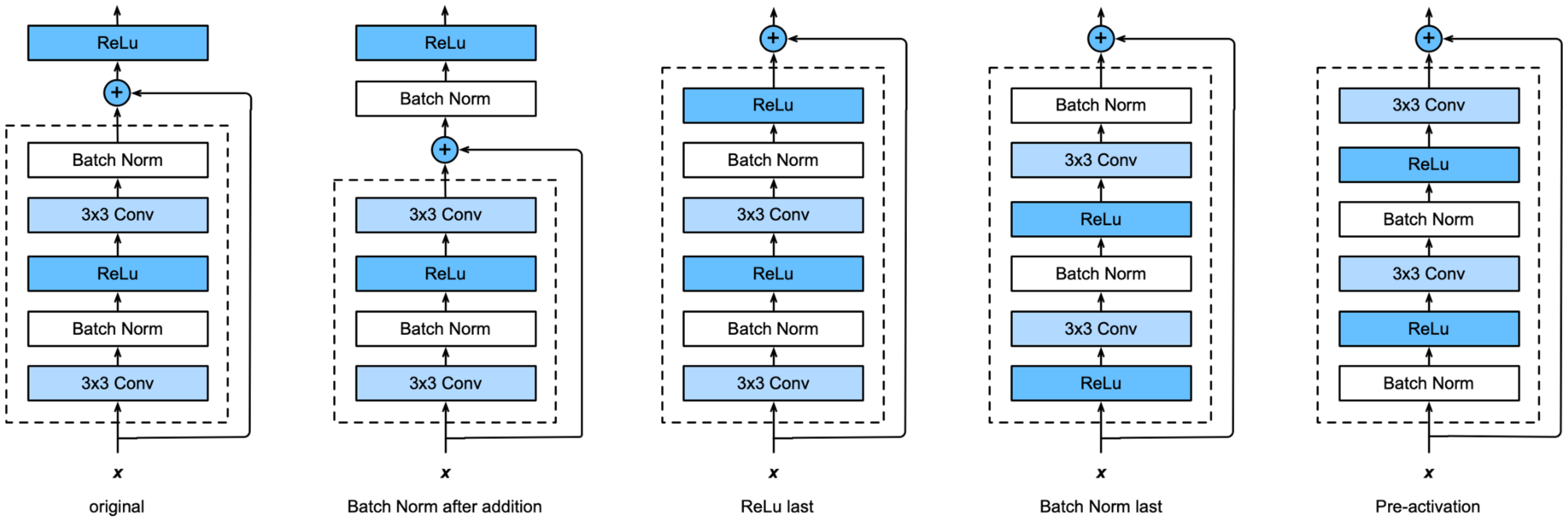
Residual Block Implementation

PYTORCH MXNET JAX TENSORFLOW

```python
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Many Variants of ResNet Blocks



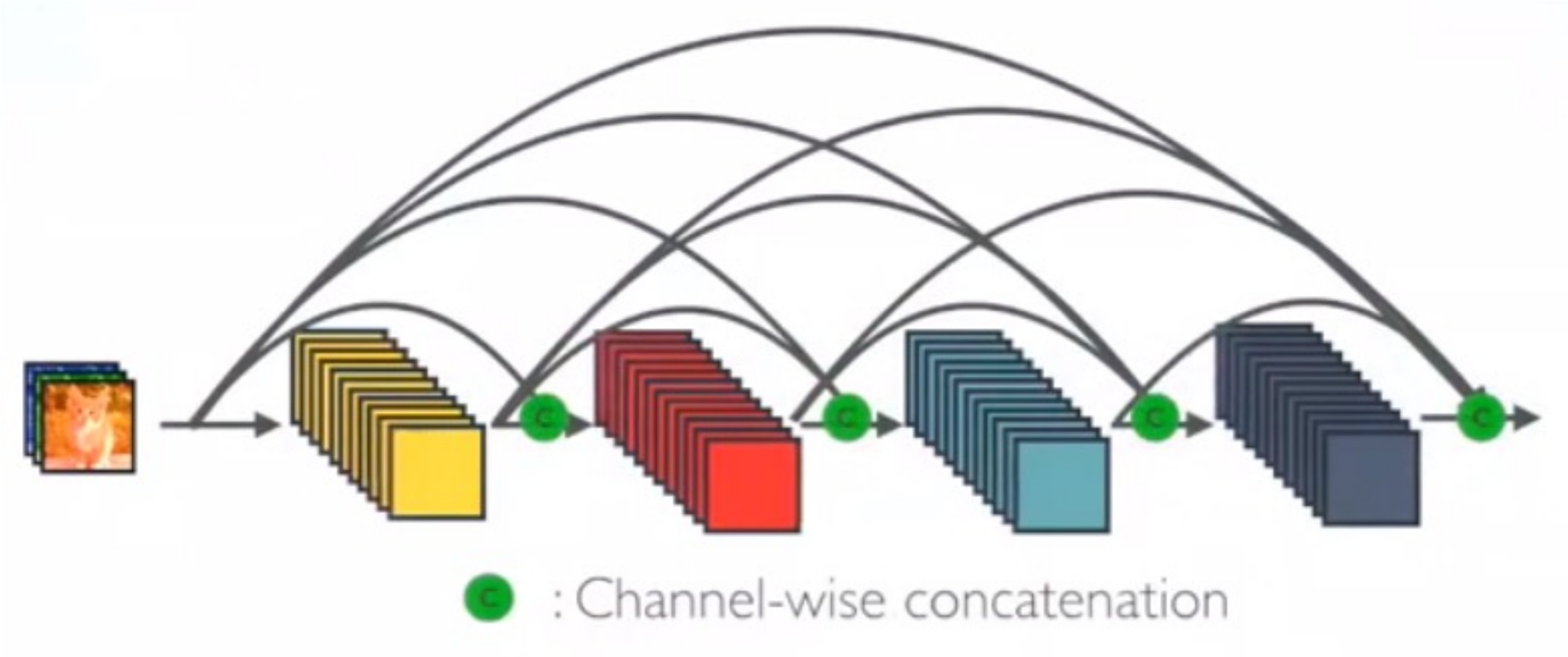
Wide ResNet

Compared to original ResNet

- Blows up number of channels
- Shrinks number of layers
- WideResNet with 50 layers outperformed ResNet with 152
- Explores value of depth vs width

[Wide Residual Networks — Zagoruyko & Komodakis]

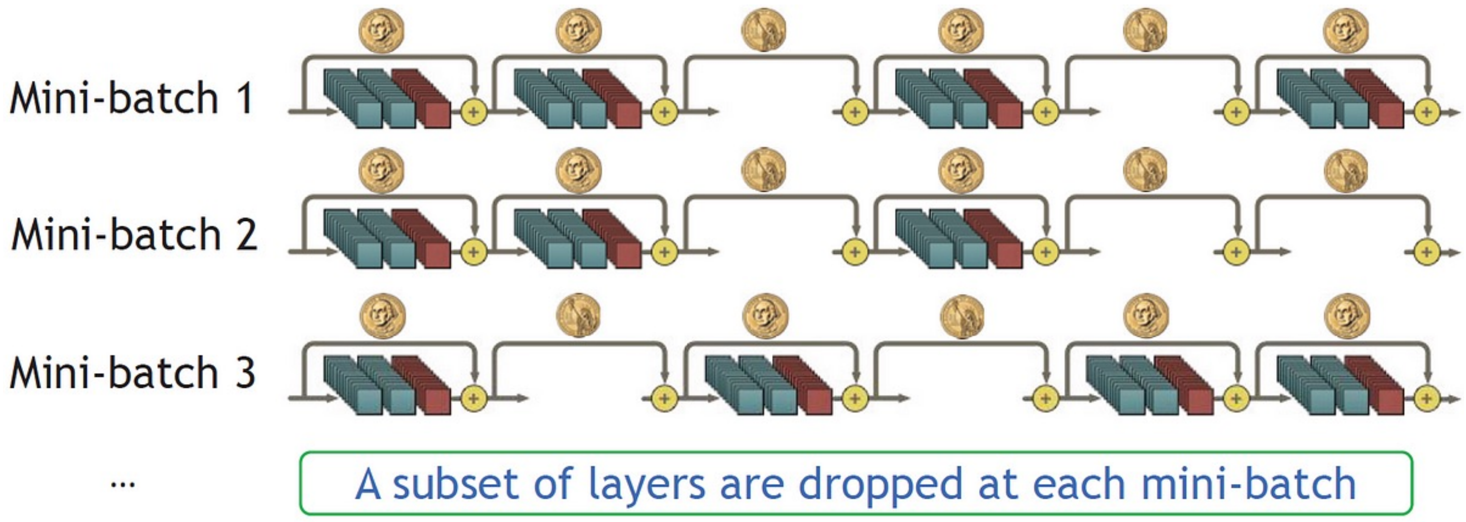
DenseNet



https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803

Stochastic Depth

- Insight: randomly drop out depth
- Make model robust to (in-activity) of any one layer
- Adapts dropout to whole layers



Img src: https://towardsdatascience.com/review-stochastic-depth-image-classification-a4e225807f4a

ResNeXT (Xie et al 2016)

- Combines residual connections with Inception-style grouping