CNN for 2D Object Classification

Lecture 2

Hyung-Sin Kim

SNU Graduate School of Data Science

Object Classification

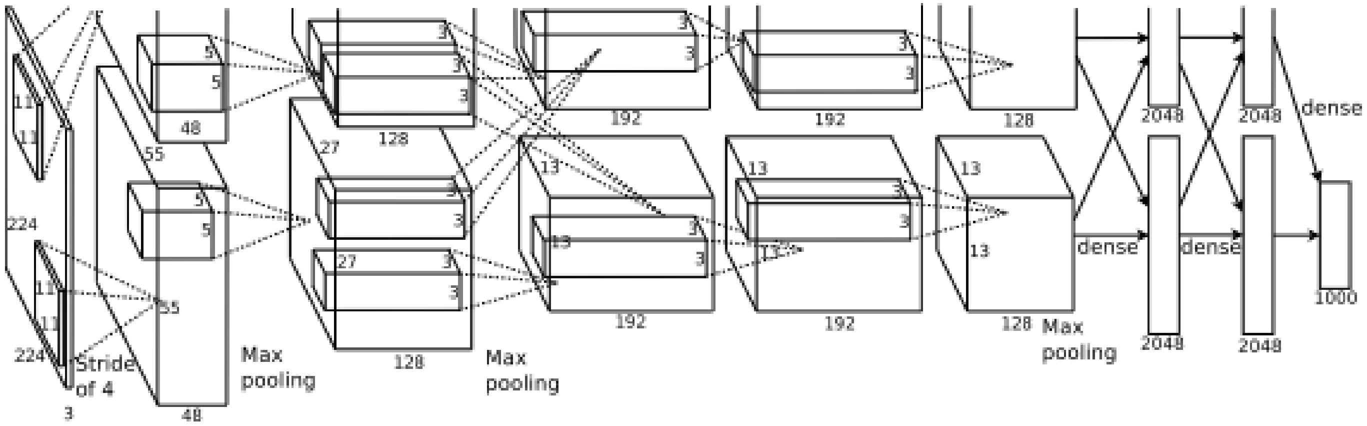
- Assumption: This image has a **single** object.
- Question: What is the object?

$$\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \text{Dog?} \\ \text{Fish?} \\ \\ \text{Cat?} \end{array}$$

SVM (simple) approaches were superior to deep neural network (DNN) approaches until ...

AlexNet [NIPS’12]

- Convolutional Neural Network (CNN) works very well on ImageNet!
  - http://www.image-net.org/challenges/LSVRC/
- DNN started receiving significant attention!
- Cited more than 86k times…



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called “dropout” that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

1 Introduction

Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Until recently, datasets of labeled images were relatively small — on the order of tens of thousands of images (e.g., NORB [16], Caltech-101/256 [8, 9], and CIFAR-10/100 [12]). Simple recognition tasks can be solved quite well with datasets of this size, especially if they are augmented with label-preserving transformations. For example, the current-best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance [4]. But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets. And indeed, the shortcomings of small image datasets have been widely recognized (e.g., Pinto et al. [21]), but it has only recently become possible to collect labeled datasets with millions of images. The new larger datasets include LabelMe [23], which consists of hundreds of thousands of fully-segmented images, and ImageNet [6], which consists of over 15 million labeled high-resolution images in over 22,000 categories.

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don’t have. Convolutional neural networks (CNNs) constitute one such class of models [16, 11, 13, 18, 15, 22, 26]. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feedforward neural networks with similarly-sized layers, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-best performance is likely to be only slightly worse.

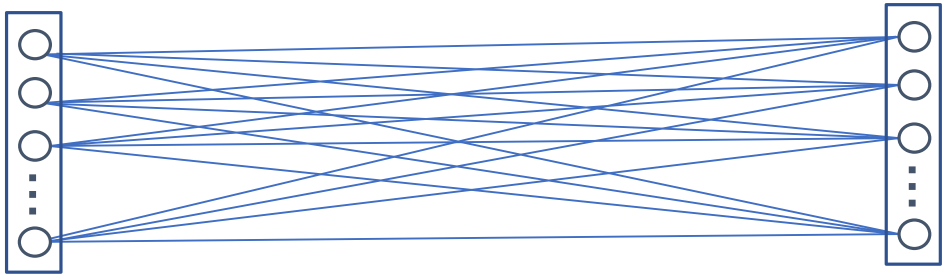
1

By the way, what is CNN?

CNN Summary – Fully Connected Layer

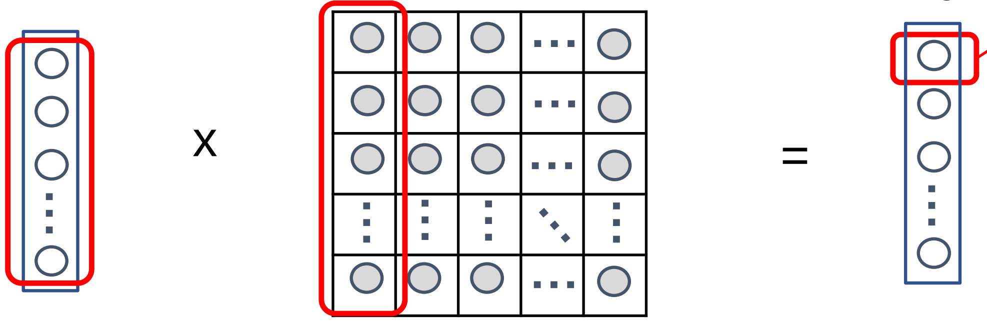
- Fully connected layer (What we have seen so far)
  - Input x: nx1 vector
  - Output y: mx1 vector

Every node is connected to each other (nxm edges!)



Learnable weight vector **w** (36x16)

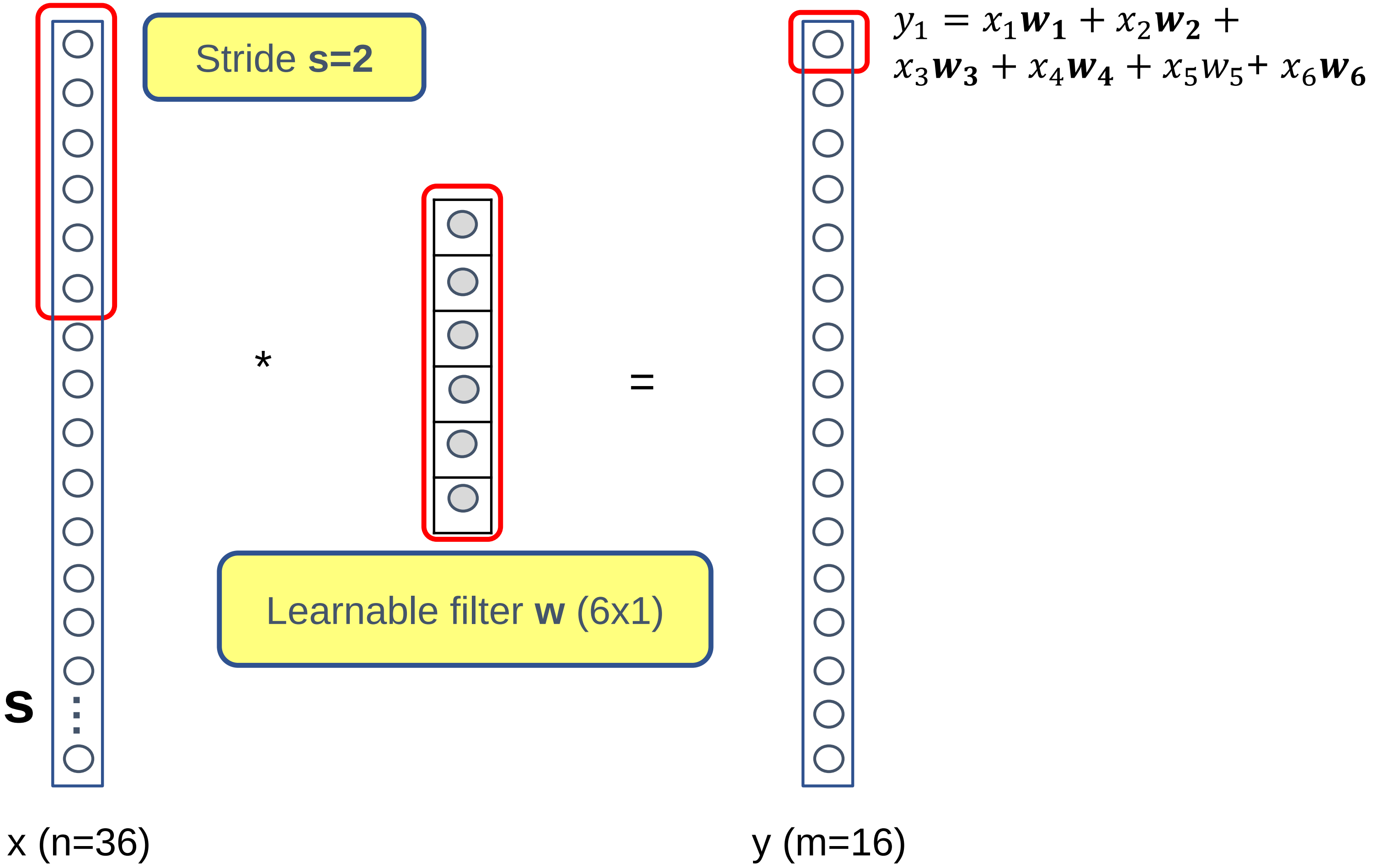
$$y_1 = x_1\boldsymbol{w}_{1,1} + x_2\boldsymbol{w}_{2,1} + x_3\boldsymbol{w}_{3,1} + \cdots + x_{36}\boldsymbol{w}_{36,1}$$



  - $\boldsymbol{Y} = \boldsymbol{W}^T\boldsymbol{X}$
  - Weight: **nxm** vector

CNN Summary – Convolutional Layer

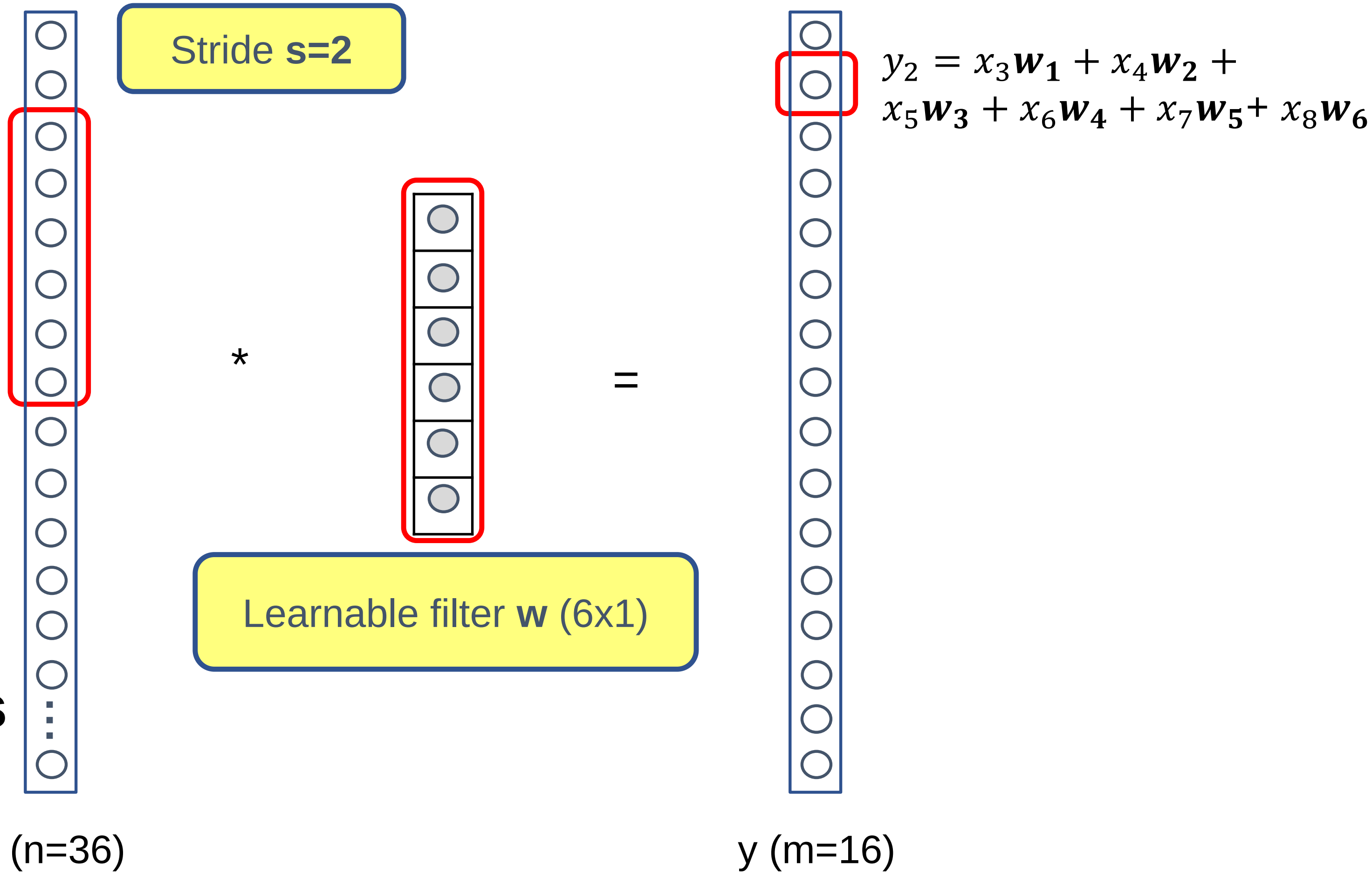
- Convolutional layer (1D)
  - Input x: nx1 vector
  - Output y: mx1 vector



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **f**x1 filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

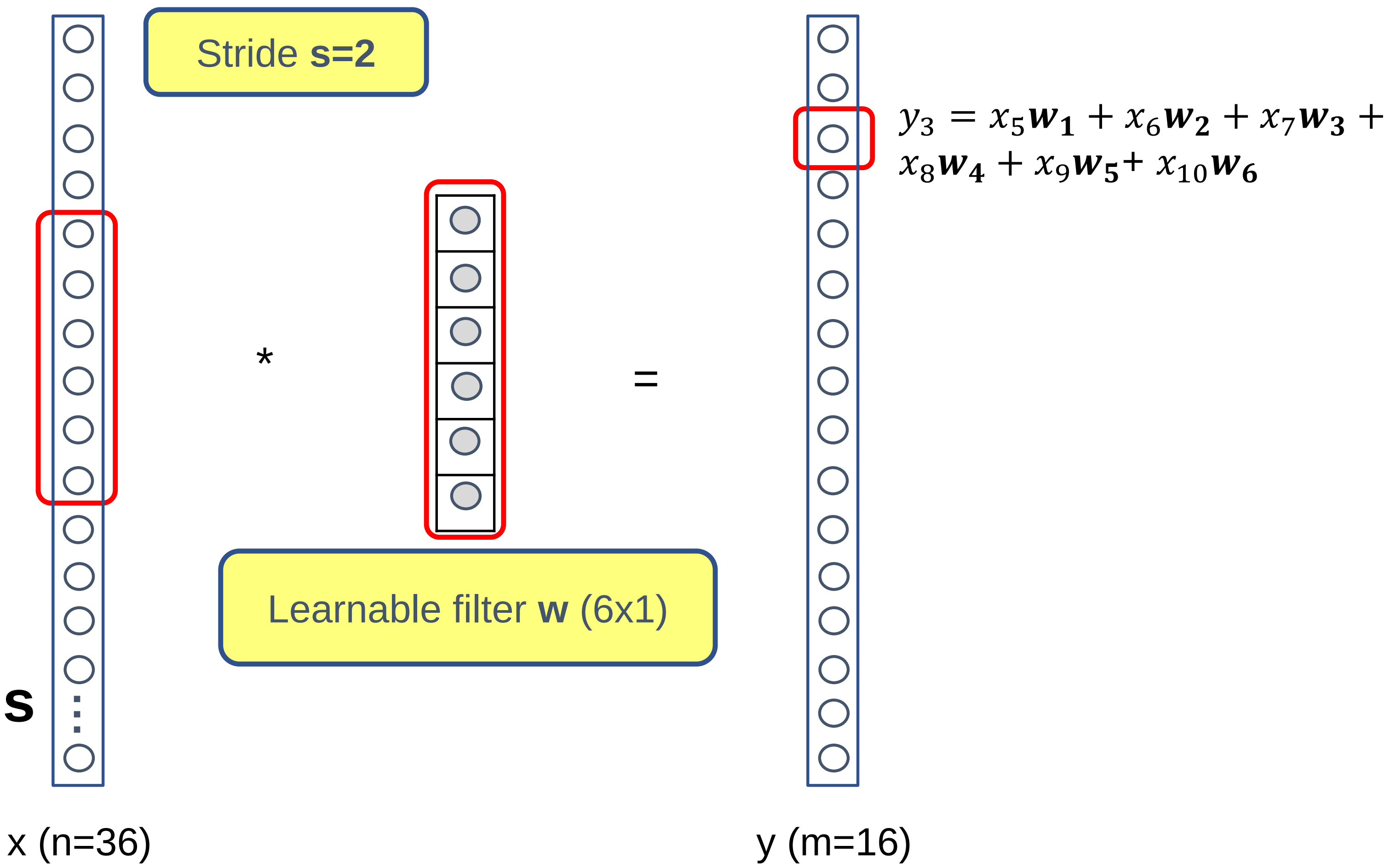
- Convolutional layer (1D)
  - Input x: nx1 vector
  - Output y: mx1 vector



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **f**x1 filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

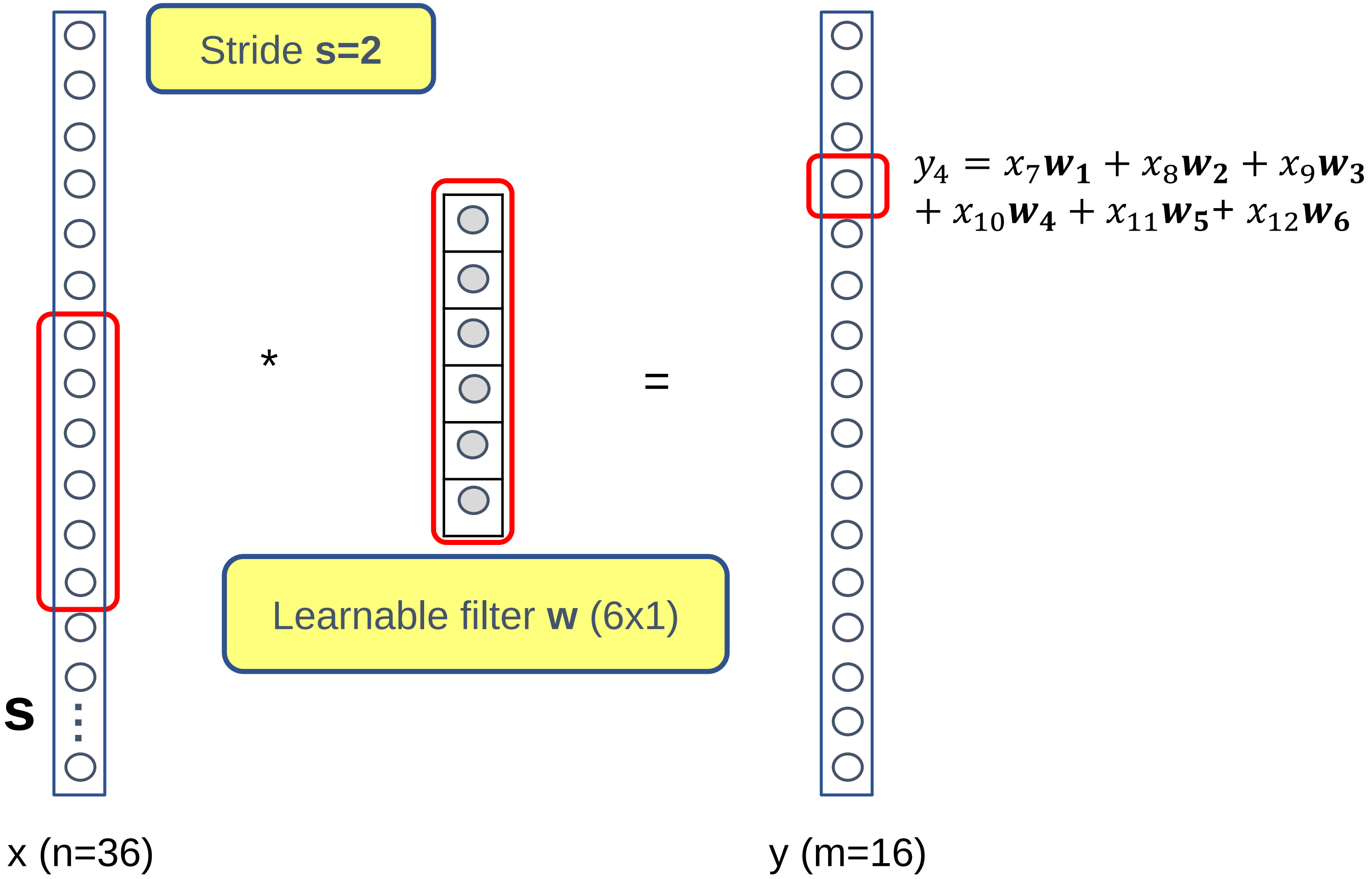
- Convolutional layer (1D)
  - Input x: nx1 vector
  - Output y: mx1 vector



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **f**x1 filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

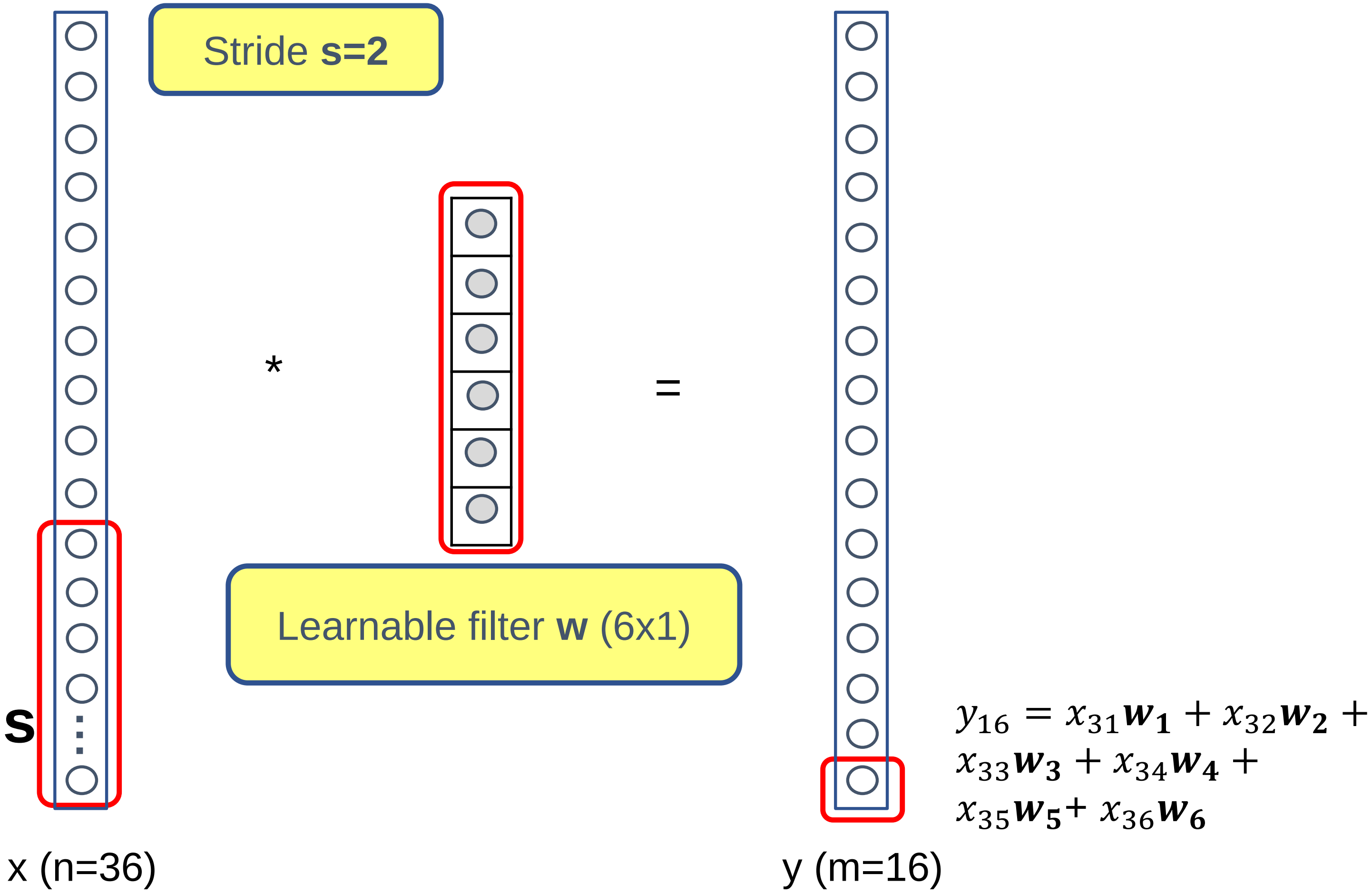
- Convolutional layer (1D)
  - Input x: nx1 vector
  - Output y: mx1 vector



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **f**x1 filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

- Convolutional layer (1D)
  - Input x: nx1 vector
  - Output y: mx1 vector

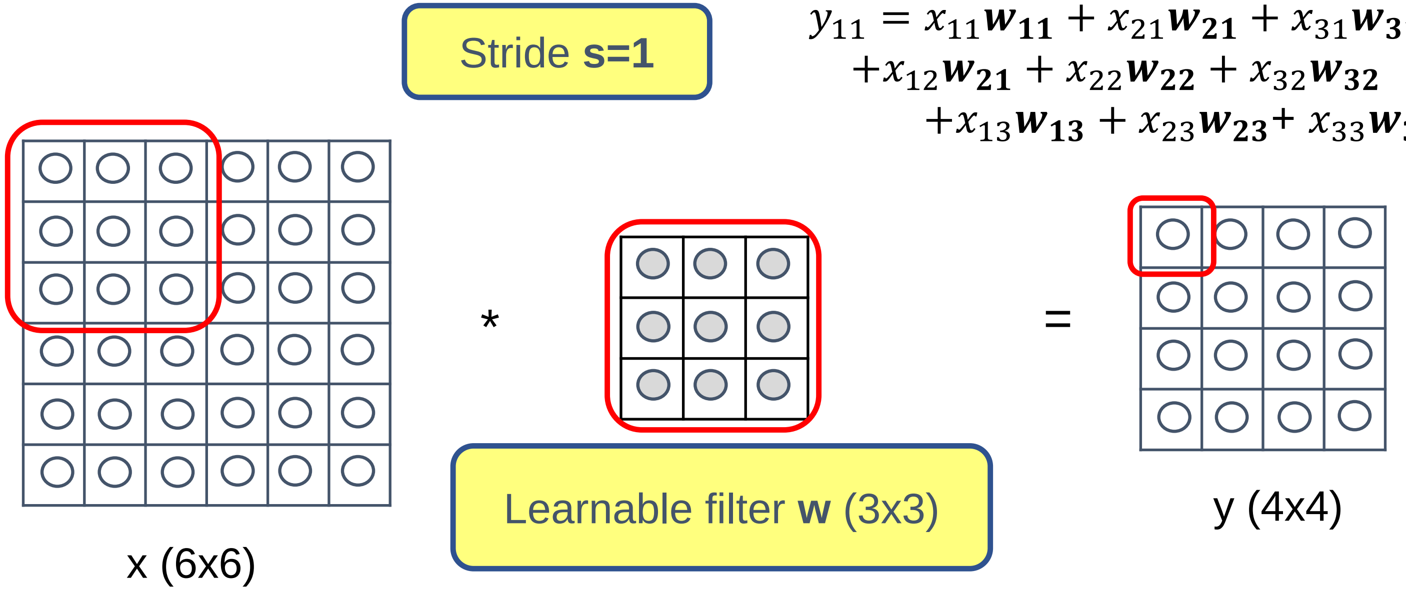


  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **f**x1 filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

- Convolutional layer (2D)
  - Input x: nxn vector
  - Output y: mxm vector

$$y_{11} = x_{11}\mathbf{w}_{11} + x_{21}\mathbf{w}_{21} + x_{31}\mathbf{w}_{31} + x_{12}\mathbf{w}_{21} + x_{22}\mathbf{w}_{22} + x_{32}\mathbf{w}_{32} + x_{13}\mathbf{w}_{13} + x_{23}\mathbf{w}_{23} + x_{33}\mathbf{w}_{33}$$

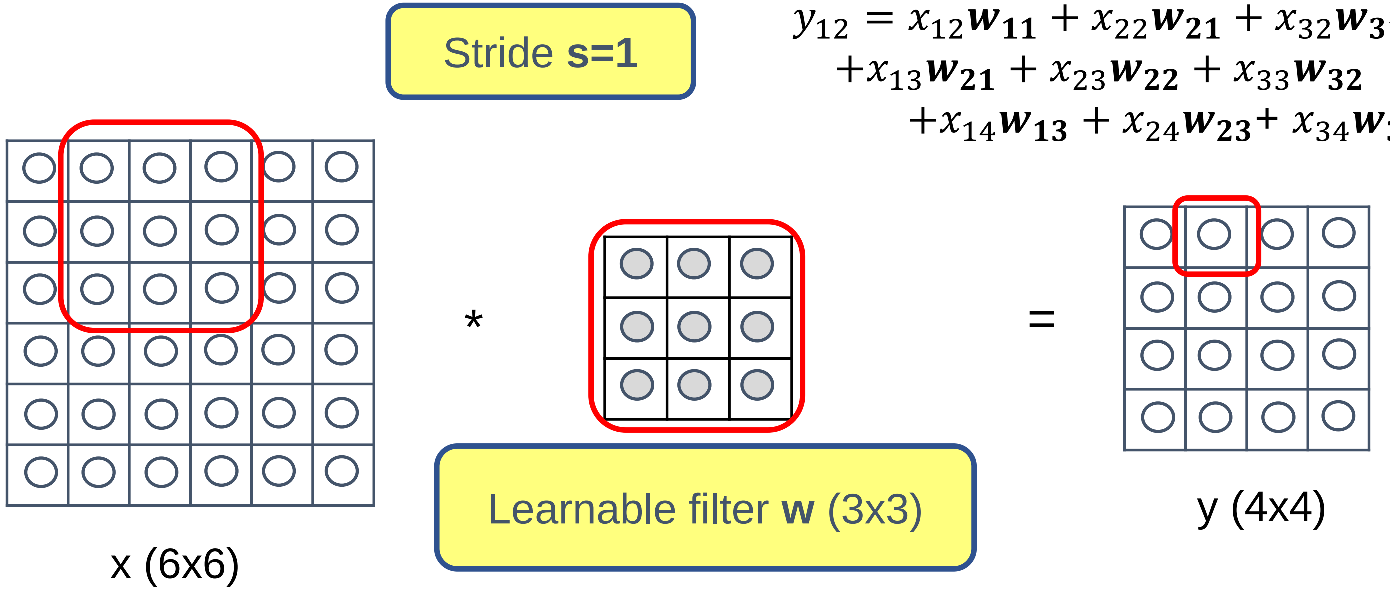


  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **fxf** filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

- Convolutional layer (2D)
  - Input x: nxn vector
  - Output y: mxm vector

$$y_{12} = x_{12}\boldsymbol{w_{11}} + x_{22}\boldsymbol{w_{21}} + x_{32}\boldsymbol{w_{31}}$$
$$+x_{13}\boldsymbol{w_{21}} + x_{23}\boldsymbol{w_{22}} + x_{33}\boldsymbol{w_{32}}$$
$$+x_{14}\boldsymbol{w_{13}} + x_{24}\boldsymbol{w_{23}} + x_{34}\boldsymbol{w_{33}}$$



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **fxf** filter (kernel)
  - Stride: hopping distance **s**

CNN Summary – Convolutional Layer

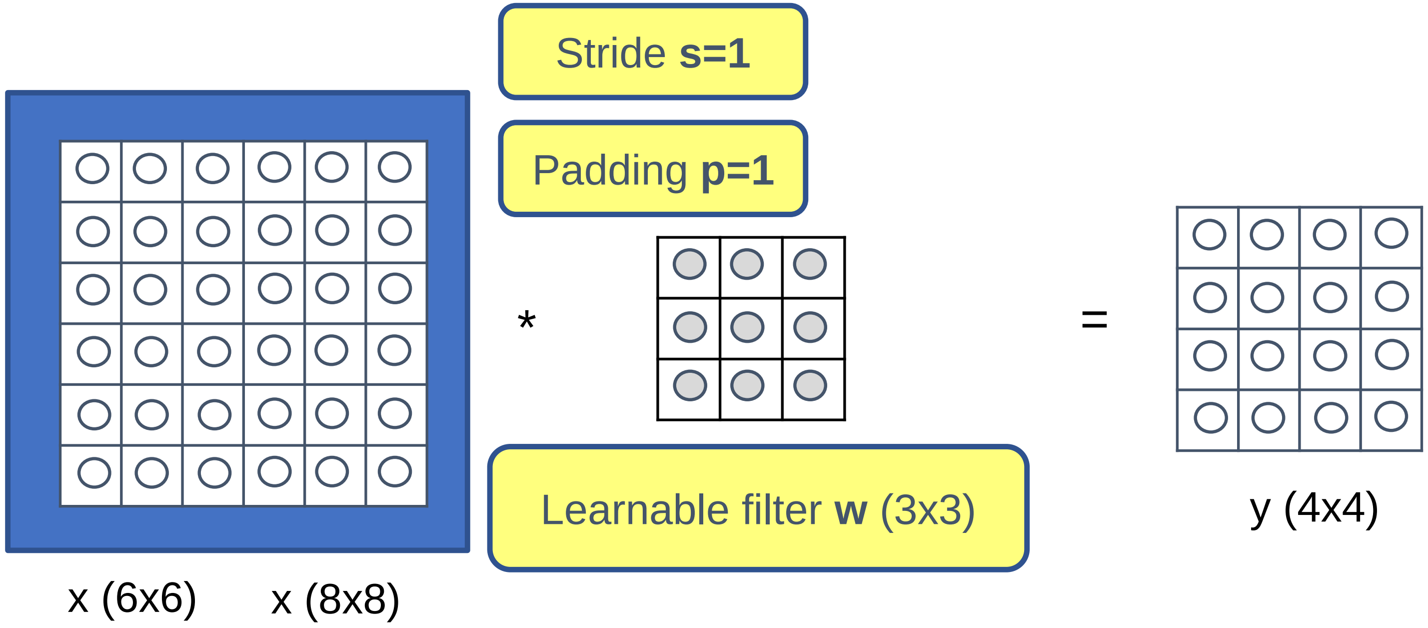
- Convolutional layer (2D)
  - Intuition how it works

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \underbrace{\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}}_{\text{Edge detector}} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

Filters can be used to extract specific features

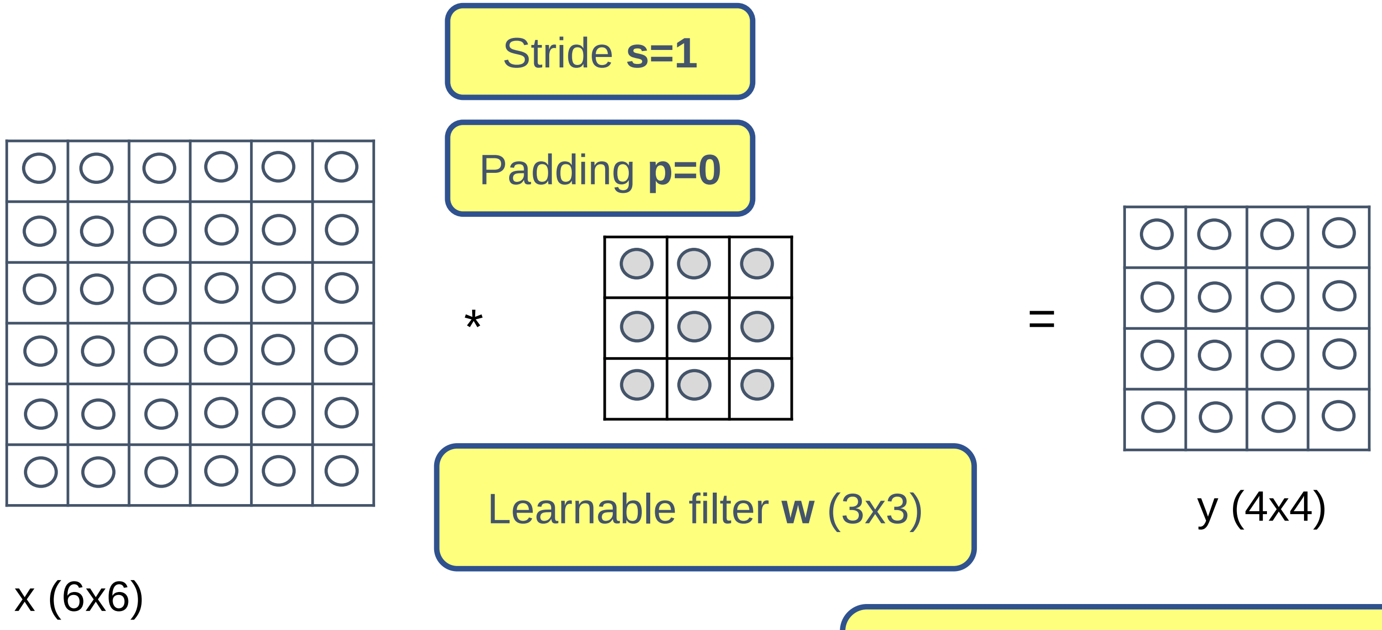
CNN Summary – Convolutional Layer

- Convolutional layer (2D)
  - Input x: nxn vector
  - Output y: mxm vector
  - $Y = X * W$
  - Weight: **fxf** filter (kernel)
  - Stride: hopping distance **s**
  - Padding (**p**): adding extra values around the edge
    - Effect: (1) Avoid shrinking image too fast, (2) Avoid throwing away info from the edge
    - Type: (1) Valid Conv (no padding), (2) Same Conv (padding to maintain output size)



CNN Summary – Convolutional Layer

- Convolutional layer (2D)
  - Input x: nxn vector
  - Output y: mxm vector



  - $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$
  - Weight: **fxf** filter (kernel)
  - Stride: hopping distance **s**
  - Padding (**p**): adding extra values around the edge
    - Effect: (1) Avoid shrinking image too fast, (2) Avoid throwing away info from the edge
    - Type: (1) Valid Conv (no padding), (2) Same Conv (padding to maintain output size)

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \text{x} \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

CNN Summary – Pooling Layer

- Pooling layer
  - Filter (fxf)
    - Max or average pooling
    - Pooling layer has **nothing learnable**
  - Stride (s)

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

Stride **s=2**

Then… why is CNN better than MLP for computer vision?

Why CNN?

- CNN

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6x6

$*$

1	0	-1
1	0	-1
1	0	-1

3x3 =
9 parameters

$=$

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

4x4

Parameter sharing
(less memory)

- Fully connected layer

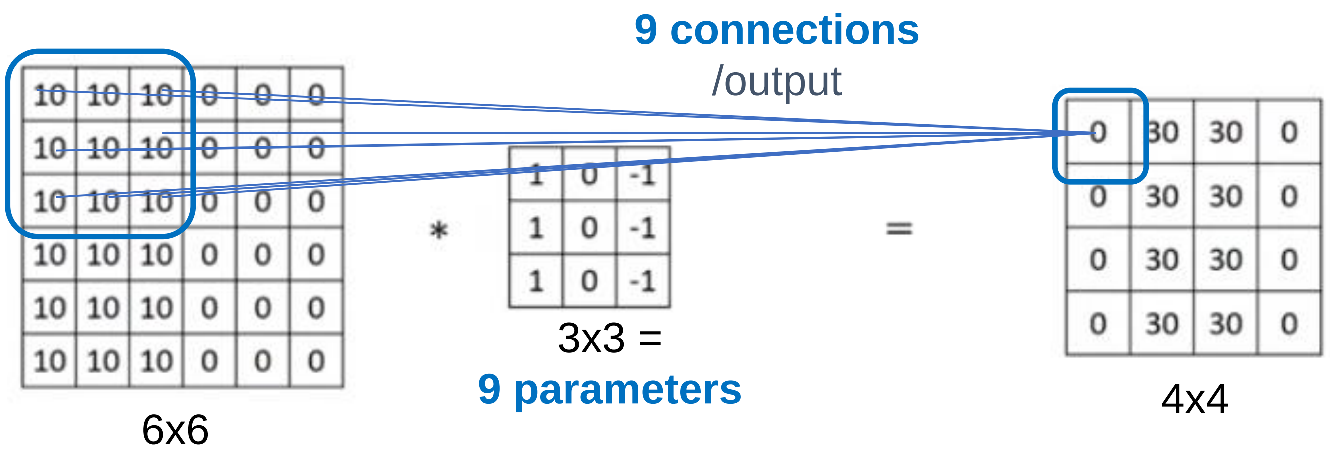
36

36x16 = **576 parameters**

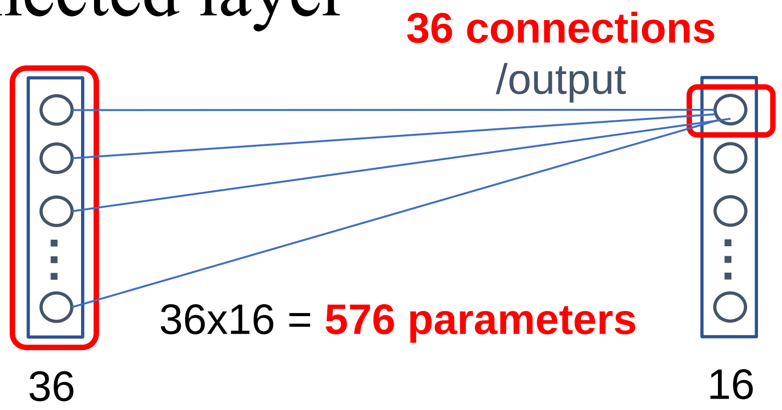
16

Why CNN?

- CNN



- Fully connected layer



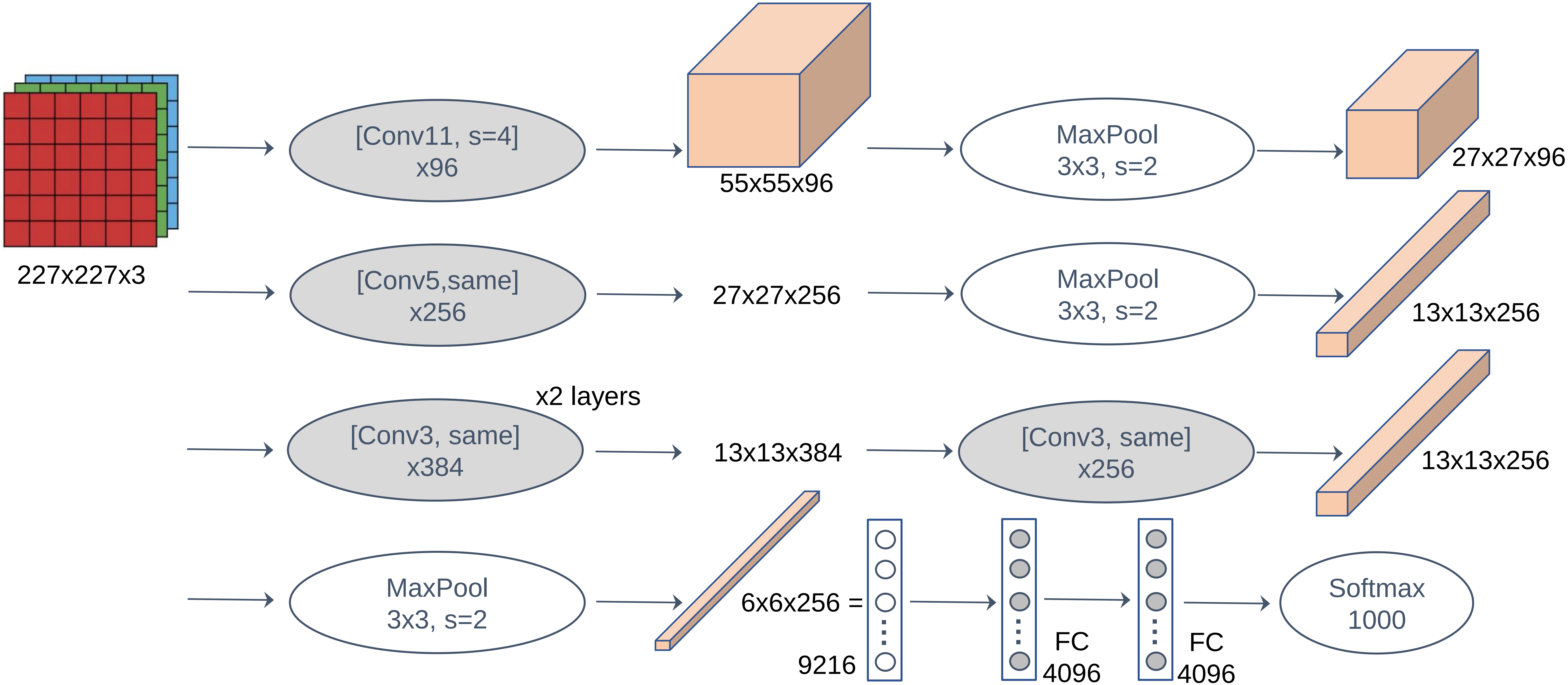
Parameter sharing (less memory)

Sparsity of connections (less computation)

Regularization (less overfitting)

Now, let's go back to AlexNet.

AlexNet [NIPS’12] - Architecture



AlexNet [NIPS'12] – Pros and Cons

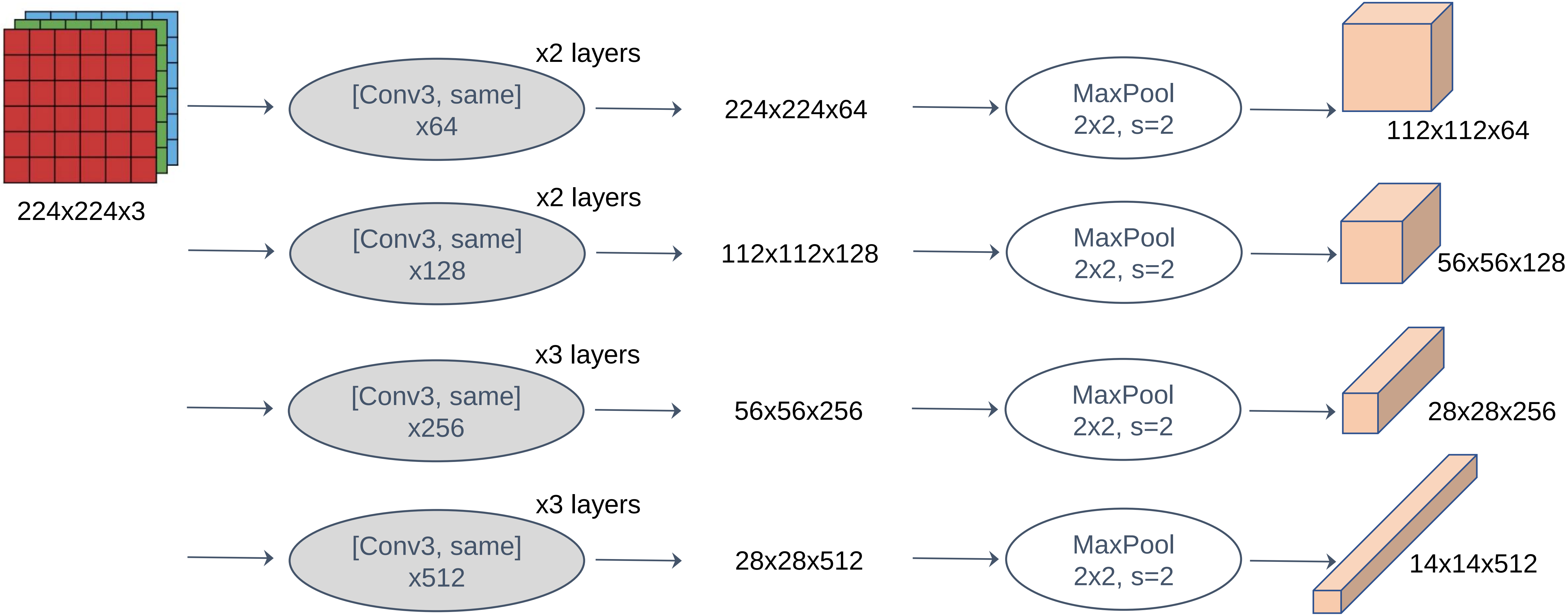
- Similar to LeNet (or LeNet-5) but has **1000x more** parameters (~60M)
  - More parameters can be trained well due to more data and computation power
- ReLu activation instead of sigmoid

- Complex architecture
  - Various filter sizes: 3x3, 5x5, 11x11
  - Various strides: 1, 2, 4

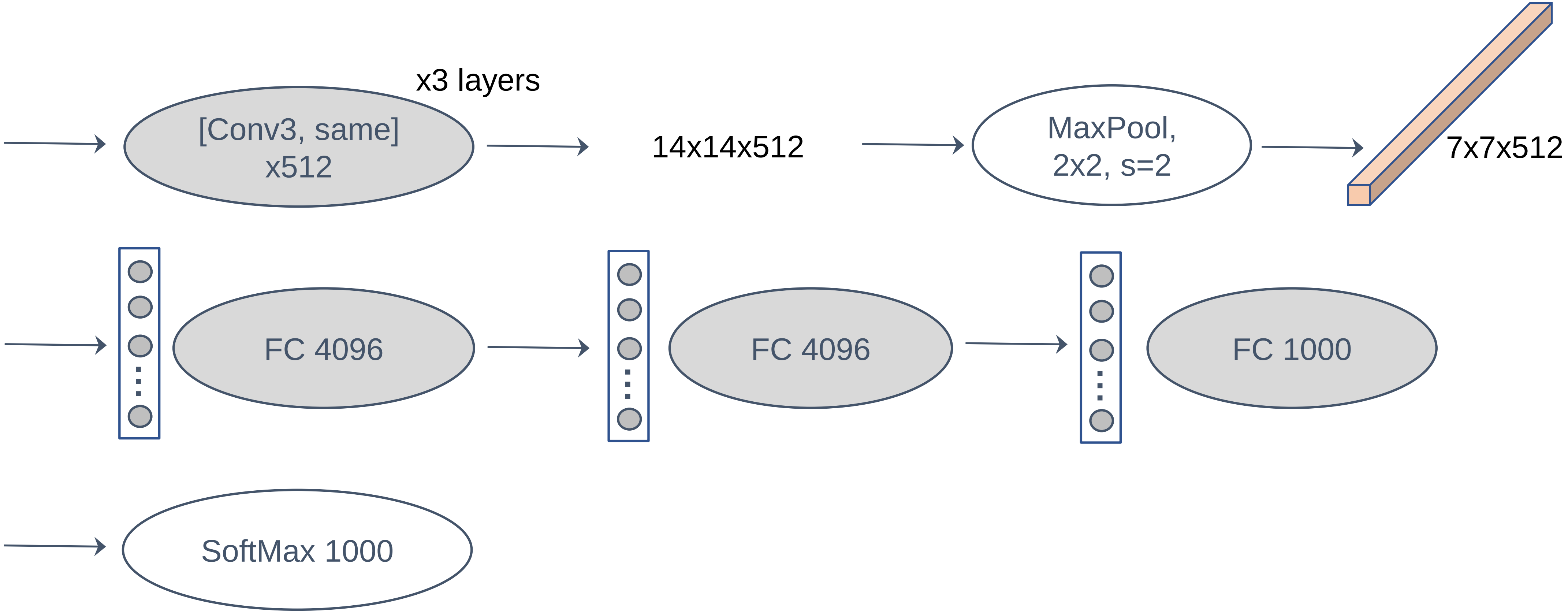
VGG-16 [ICLR'15]

- Simple and uniform architecture!
  - One type convolution layer: 3x3, s=1
  - One type pooling layer: max pooling, 2x2, s=2
  - Doubling channels

VGG-16 [ICLR'15] - Architecture



VGG-16 [ICLR'15] - Architecture



VGG-16 [ICLR'15]

- Simple and uniform architecture!
  - One type convolution layer: 3x3, s=1
  - One type pooling layer: max pooling, 2x2, s=2
  - Doubling channels

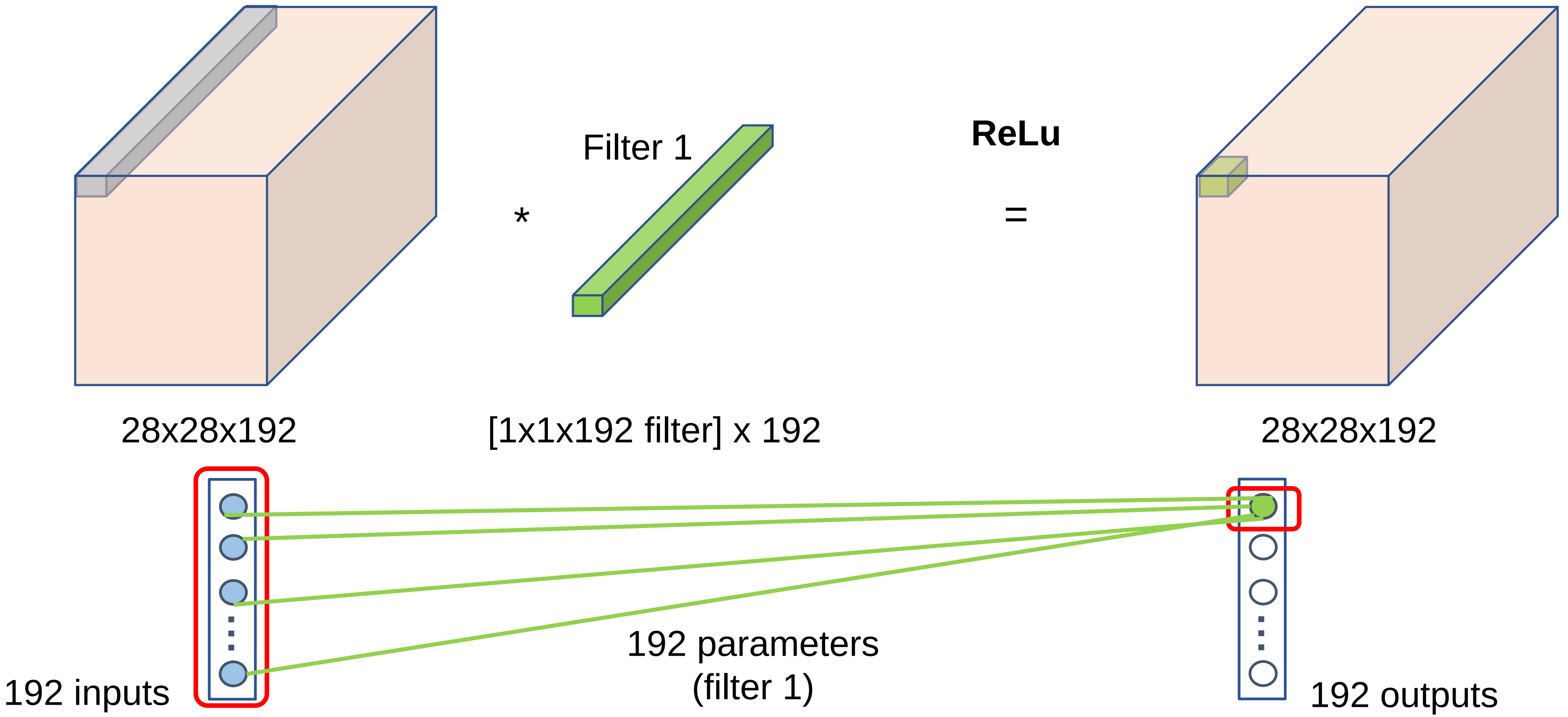
- So many parameters (16 layers, ~138M parameters)

Inception [CVPR'15]

- Problems
  - Computation increases with # of layers
  - CNN design is complex
    - You should determine layer type (conv or pooling), filter size, and stride…
- Don't worry, we will do these all at each layer, with **efficient computation**!
  - Then, we can go deeper conveniently

1x1 Convolution

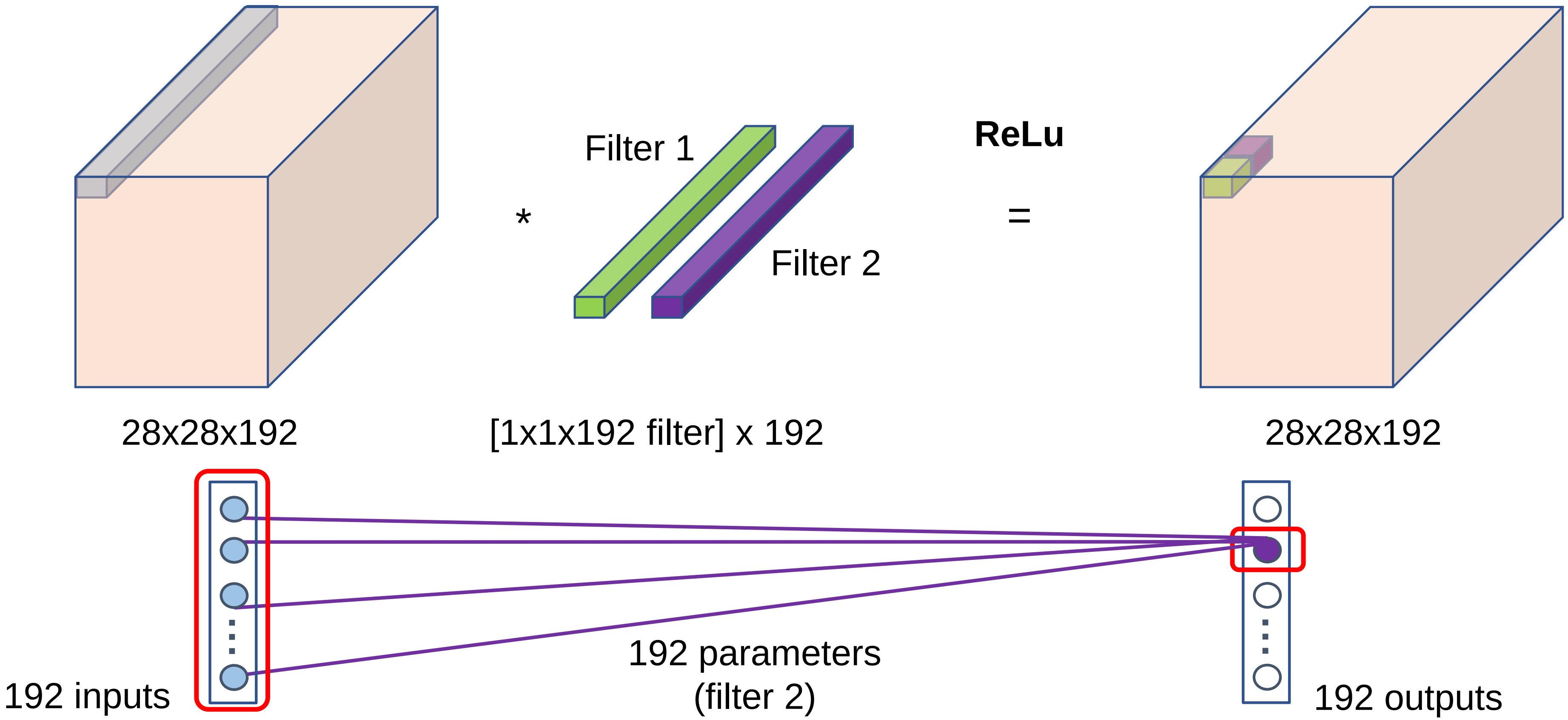
- Cross-channel fully connected layer



1x1 Convolution

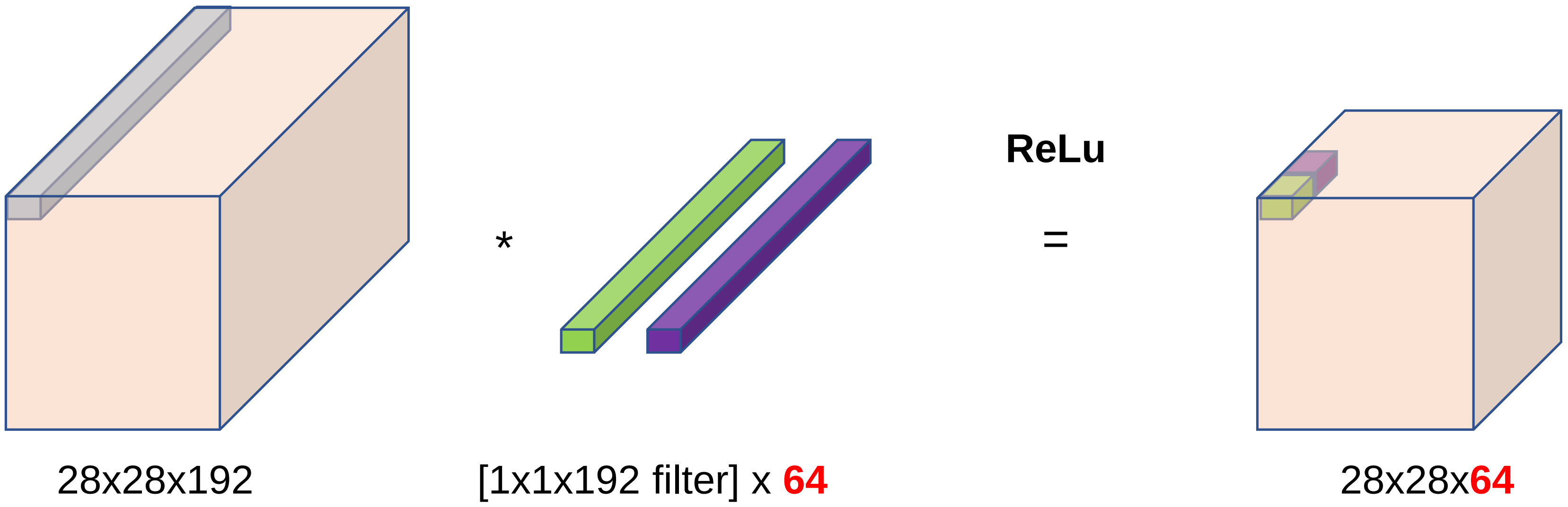
- Cross-channel fully connected layer

Adding nonlinearity without changing output dimension



1x1 Convolution

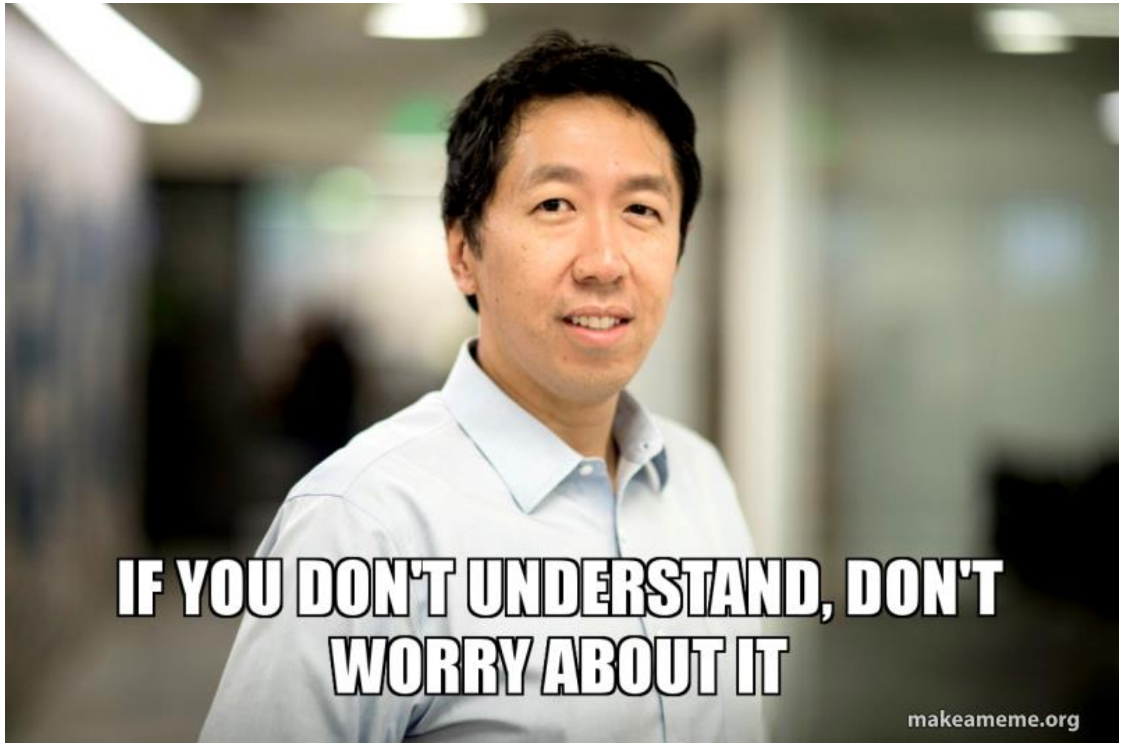
- What if we change the number of filters?



Changing # of channels while maintaining width and height

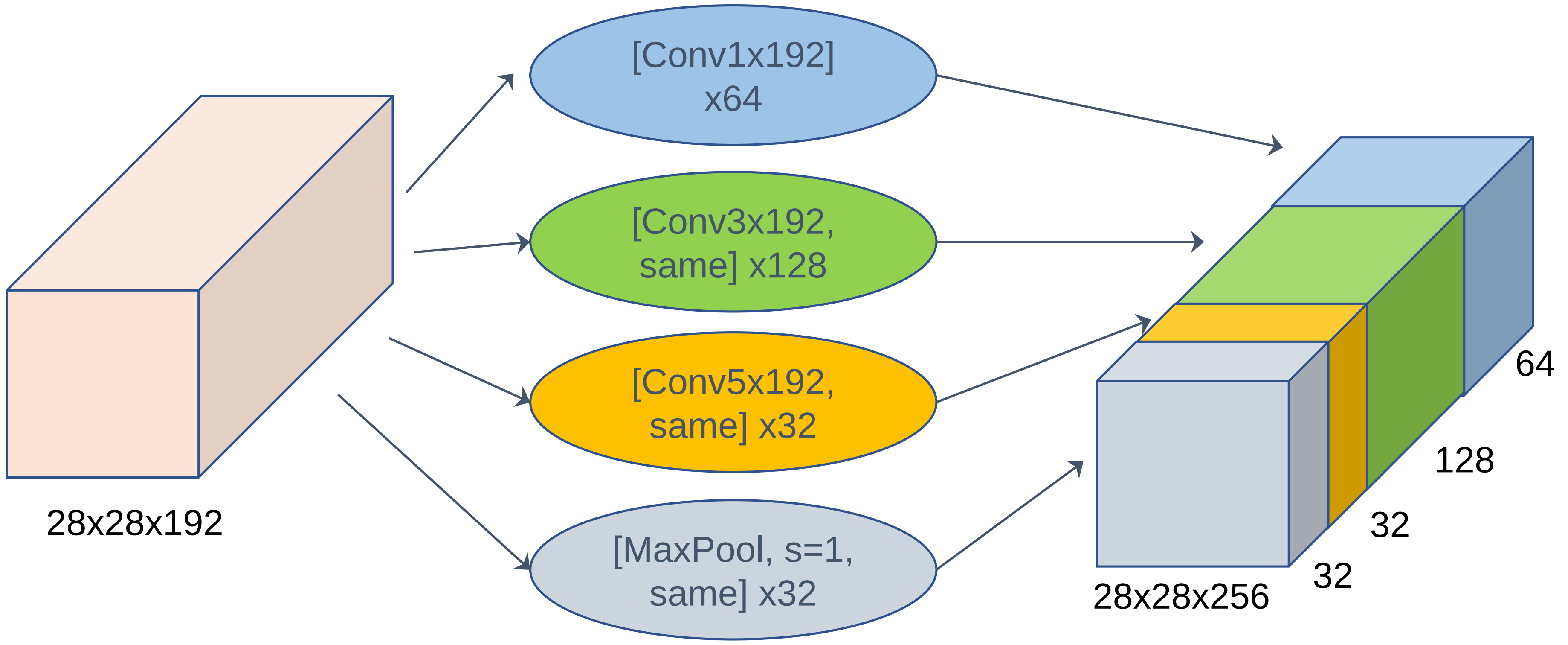
1x1 Convolution

- Finally, 1x1 conv is actually **faster** than normal conv even when the number of operations is same, because it does not require **memory reordering**



Inception [CVPR’15]

- Inception module – naïve (impractical) version
  - Mix various filters in a single layer and let them trained automatically
  - Maintain width and height by using padding



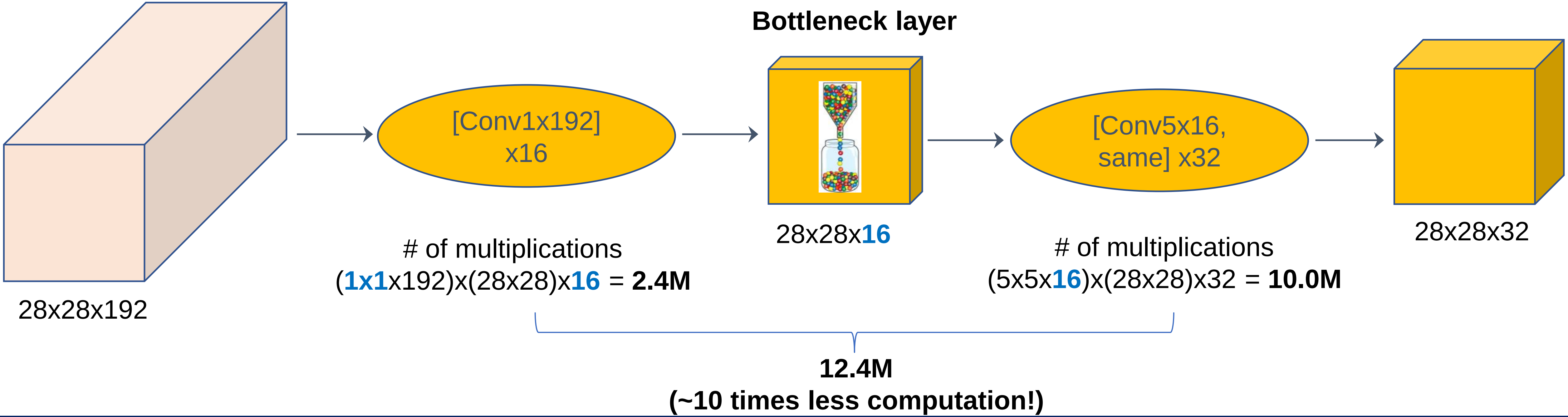
Inception [CVPR'15]

- Inception module – Reducing computation cost
  - Passing a conv filter is quite expensive when the input (or output) has **many channels**

\# of multiplications
(5x5x**192**)x(28x28)x32 = **120.4M**

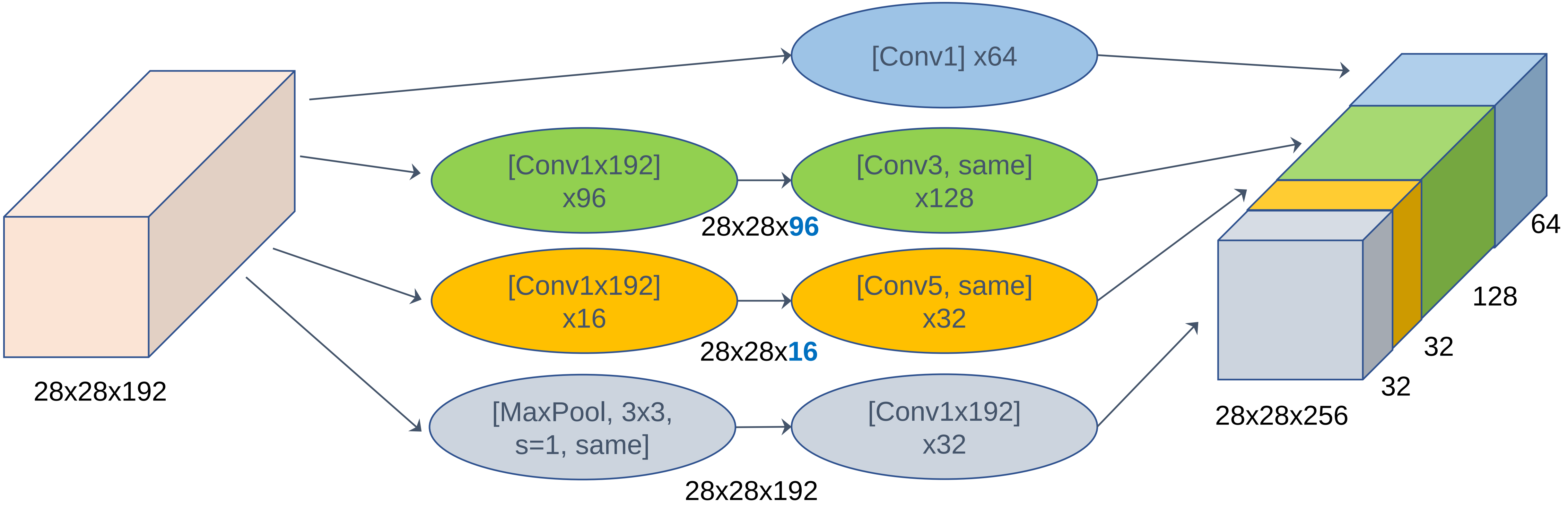
Inception [CVPR'15]

- Inception module – Reducing computation cost
  - Passing a conv filter is quite expensive when the input (or output) has **many channels**
  - Reduce # of channels by **inserting a 1x1 Conv layer**, making an intermediate **bottleneck** layer
  - Then, increase # of channels again by using the original filter size



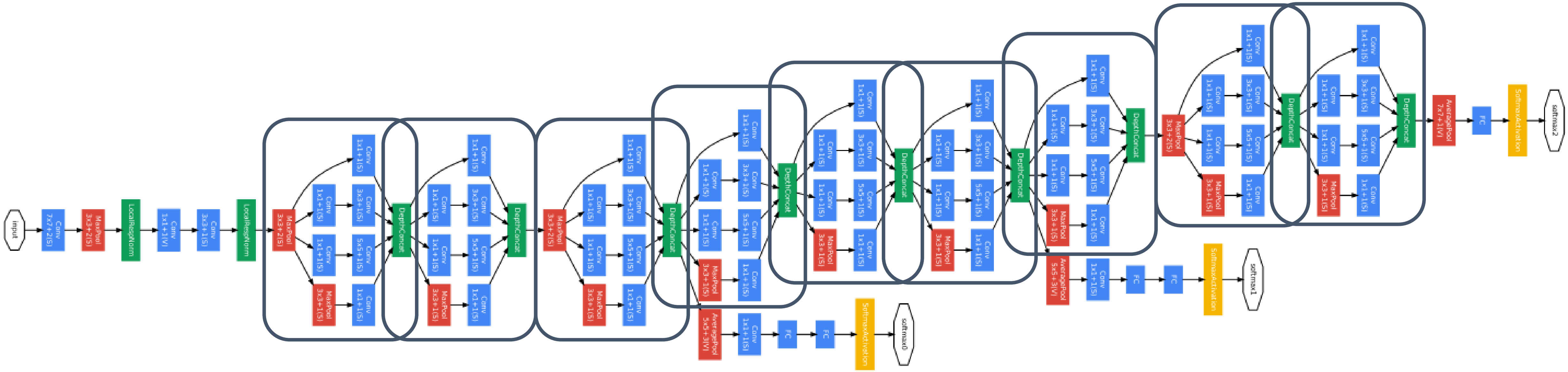
Inception [CVPR'15]

- Inception module – with less computational overhead



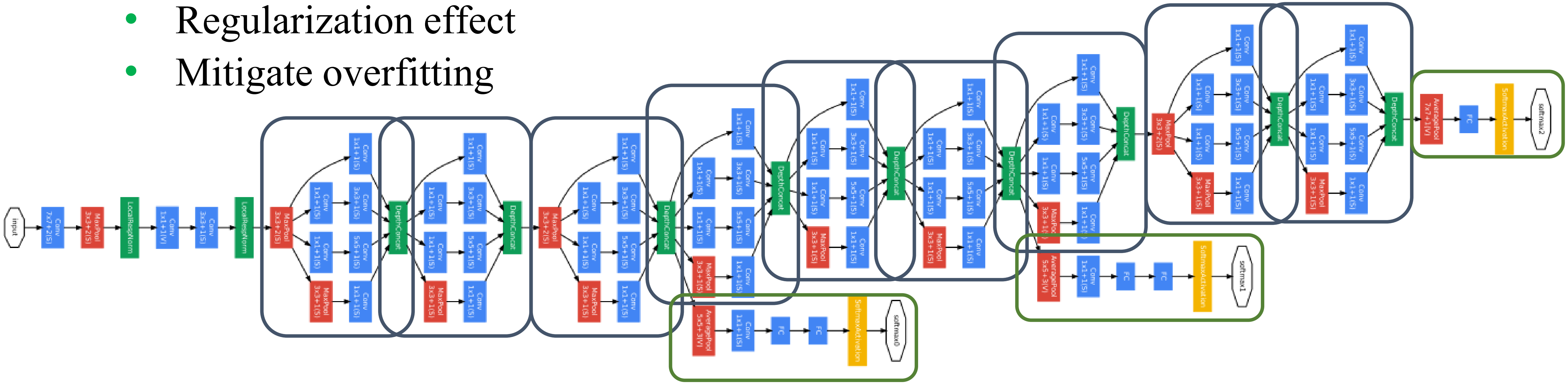
Inception [CVPR'15]

- GoogLeNet – a **22-layer** DNN using inception modules



Inception [CVPR'15]

- GoogLeNet – a **22-layer** DNN using inception modules
- Extract results in the middle of network and include them in the loss function
  - Regularization effect
  - Mitigate overfitting



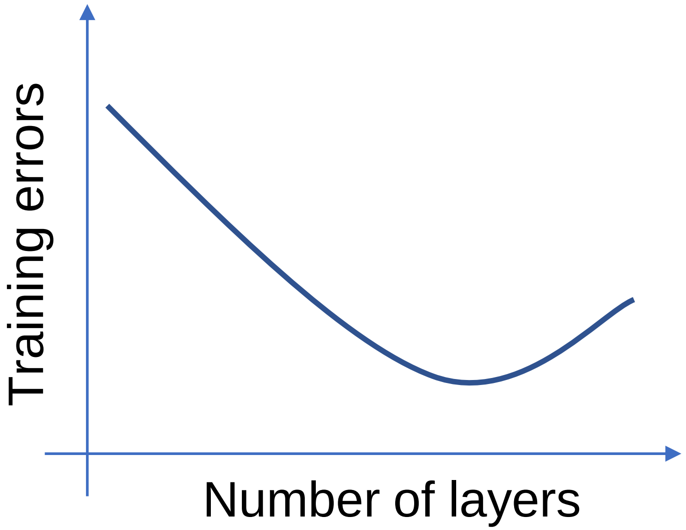
Inception [CVPR'15]

- Inception shows that, even though inserting a bottleneck layer could lose lots of information, doing it is actually **NOT** that harmful

- Implication
  - CNN is more efficient than MLP
    - Parameter sharing and sparsity of connections
  - However, this architecture is also designed by **human intuition**
    - Yes, it works, but there has been no mathematical/empirical verification showing if it is "THE" optimal architecture
  - A significant amount of CNN computation might still be **unnecessary**
  - We might be able to build a lighter DNN without sacrificing accuracy

ResNet [CVPR'16] - Motivation

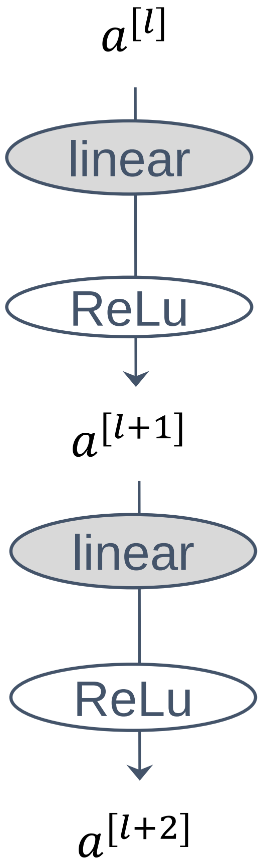
- You know what? I want to train an even ***deeper and heavier*** neural network
- Hmm… but when a model is too deep, performance is degraded



How can we continuously decrease training errors as the number of layers increases?

ResNet [CVPR'16] – Core Element

- Residual connection



$z^{[l+1]} = w^{[l+1]}a^{[l]} + b^{[l+1]}$

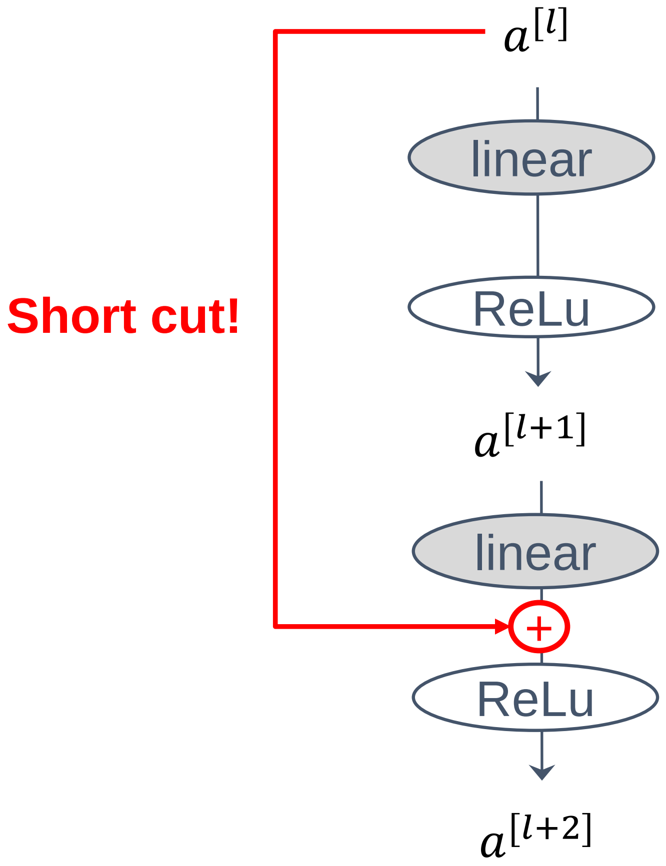
$a^{[l+1]} = g(z^{[l+1]})$

$z^{[l+2]} = w^{[l+2]}a^{[l+1]} + b^{[l+2]}$

$a^{[l+2]} = g(z^{[l+2]})$

ResNet [CVPR’16] – Core Element

- Residual connection



$$z^{[l+1]} = w^{[l+1]}a^{[l]} + b^{[l+1]}$$

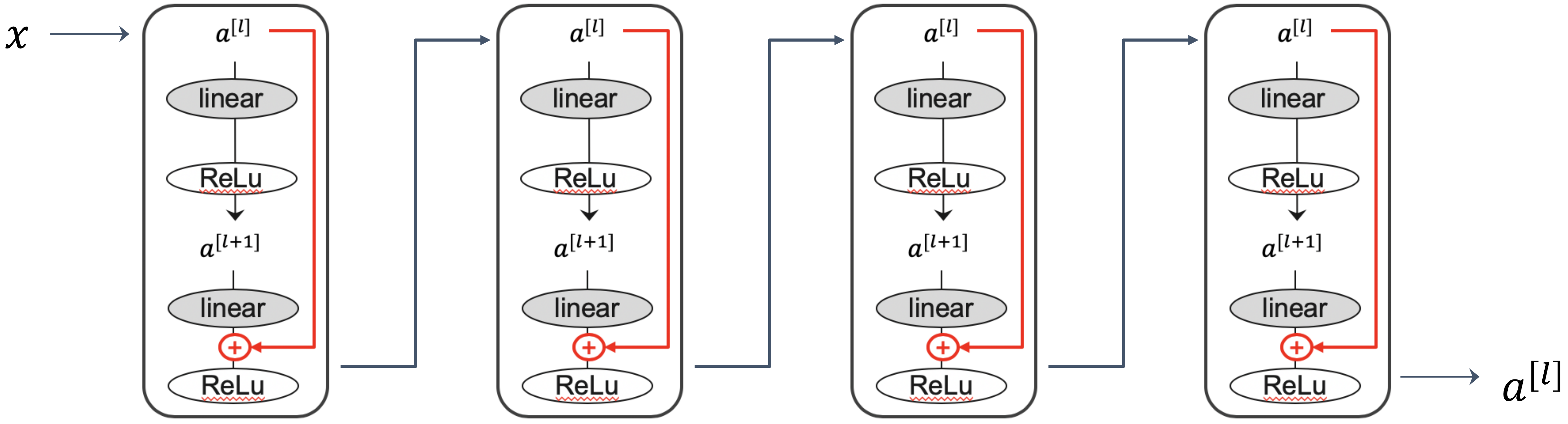
$$a^{[l+1]} = g(z^{[l+1]})$$

$$z^{[l+2]} = w^{[l+2]}a^{[l+1]} + b^{[l+2]}$$

$$a^{[l+2]} = g(z^{[l+2]} + \boldsymbol{a^{[l]}})$$

ResNet [CVPR'16]

- ResNet: DNN architecture with residual connection



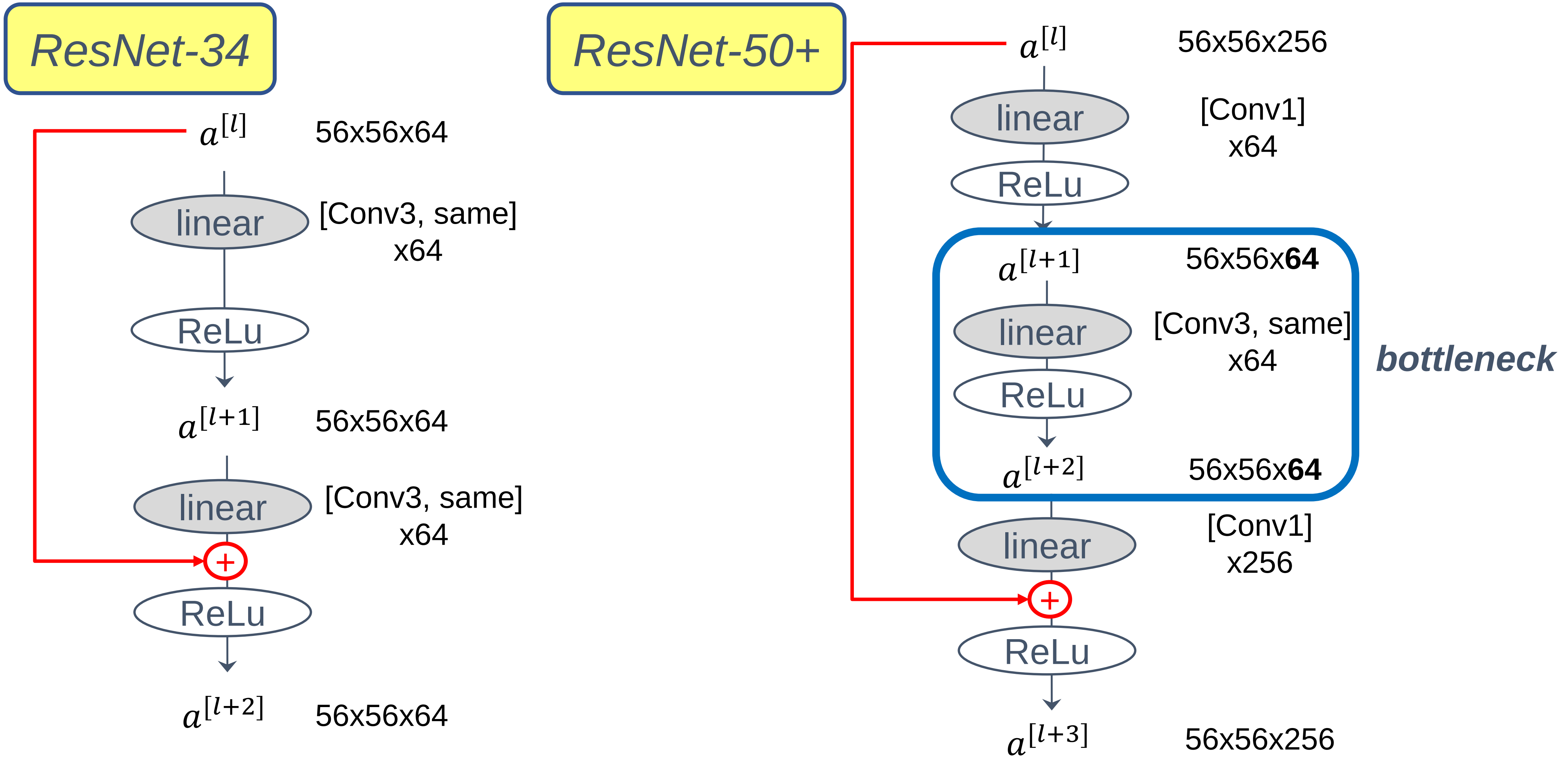
ResNet [CVPR'16] – Why does it work?

- Why is the residual connection good for training a deep network?
  - In a negative case, weights of a new layer can be trained as the identity function, which does not degrade performance at least
    - $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$
      $= g(w^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$
      $= g(a^{[l]})$ when $w^{[l+2]} = 0$, $b^{[l+2]} = 0$
      $= a^{[l]}$
  - In a fortunate case, weights of a new layer can be trained in a positive way, improving performance
  - Less risk, but more opportunity as a DNN goes deeper **(>100 layers)**

But how about training overhead for a super deep DNN?

*We have the **bottleneck** approach!*

ResNet [CVPR’16] – Bottleneck Architecture



Thanks!