

Data Science

Lecture 9-2: Image Data Processing (Convolutional Neural Network)



Lecturer: Yen-Chia Hsu

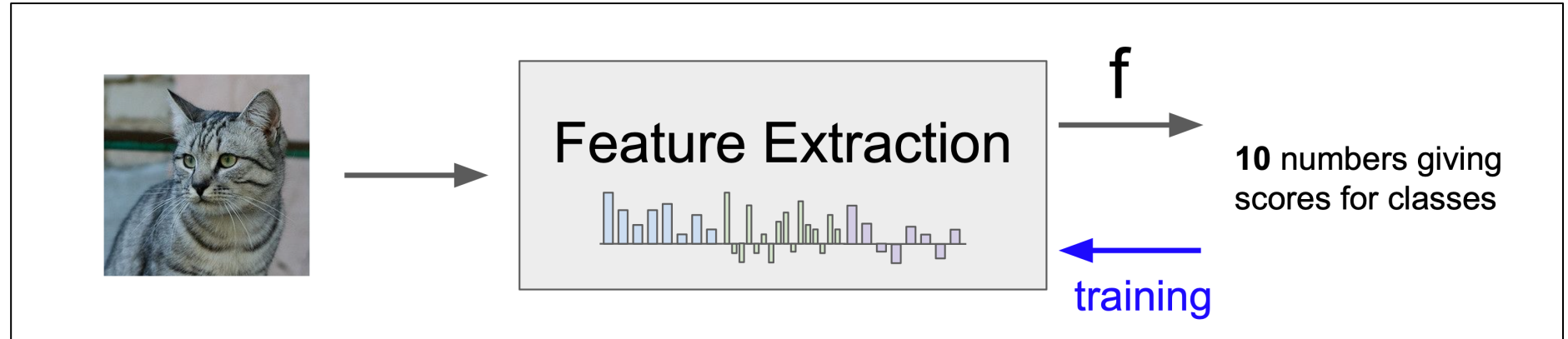
Date: Mar 2025

This lecture covers image processing basics
using Convolutional Neural Network.

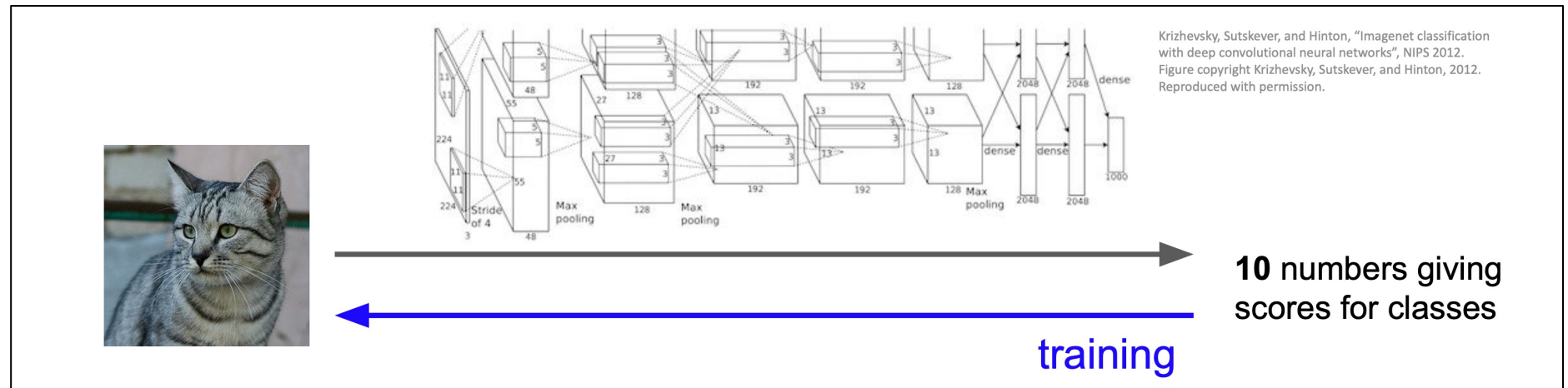
Can we train models to learn convolution kernels/filters and features automatically?

Deep learning allows us to **train a model end-to-end**, which means the inputs are raw pixel values, and the outputs are categories or heatmaps.

Traditional Approach



Deep Learning Approach



On 1998, LeNet was developed to learn convolution kernels/filters to recognize digits (i.e., the MNIST dataset). But due to insufficient computational power and lack of data during that time, deep learning was not popular in Computer Vision.

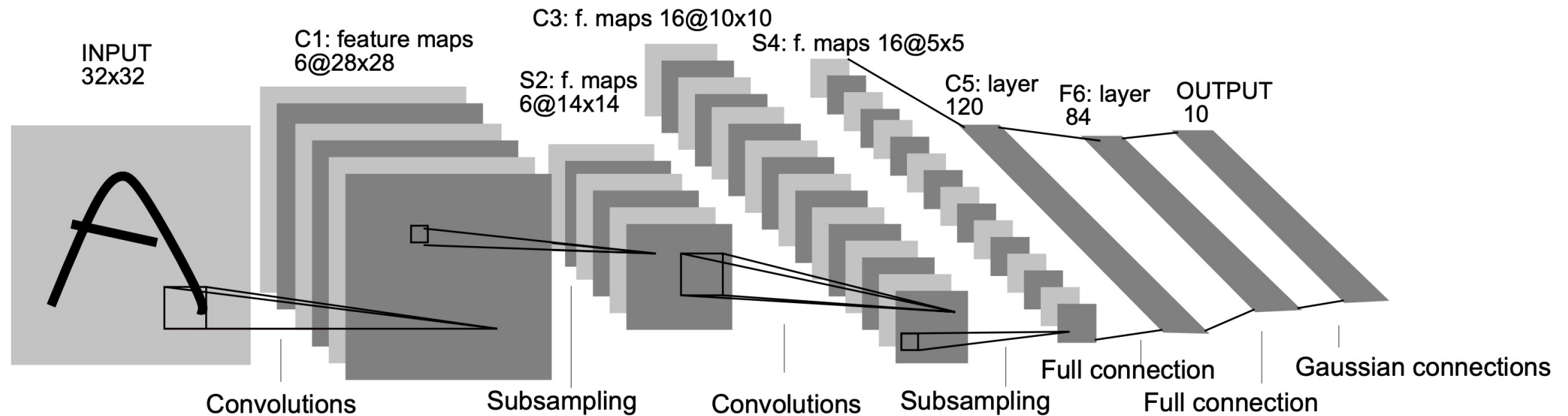
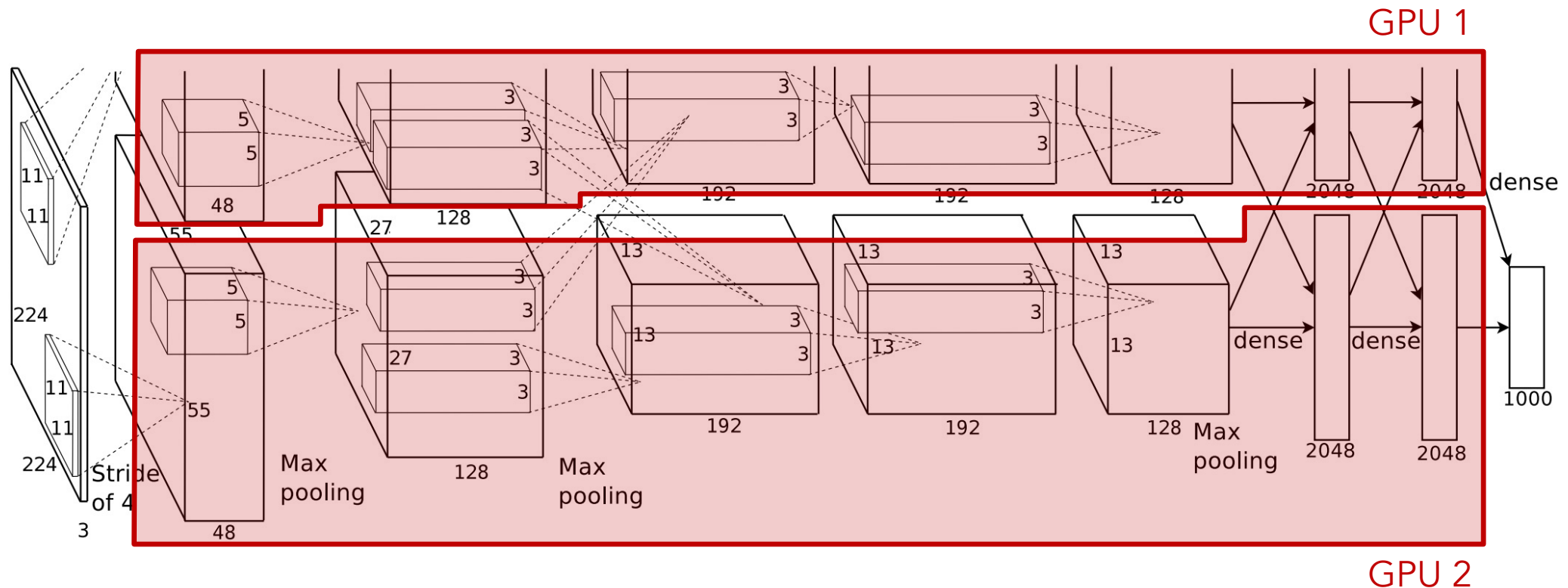
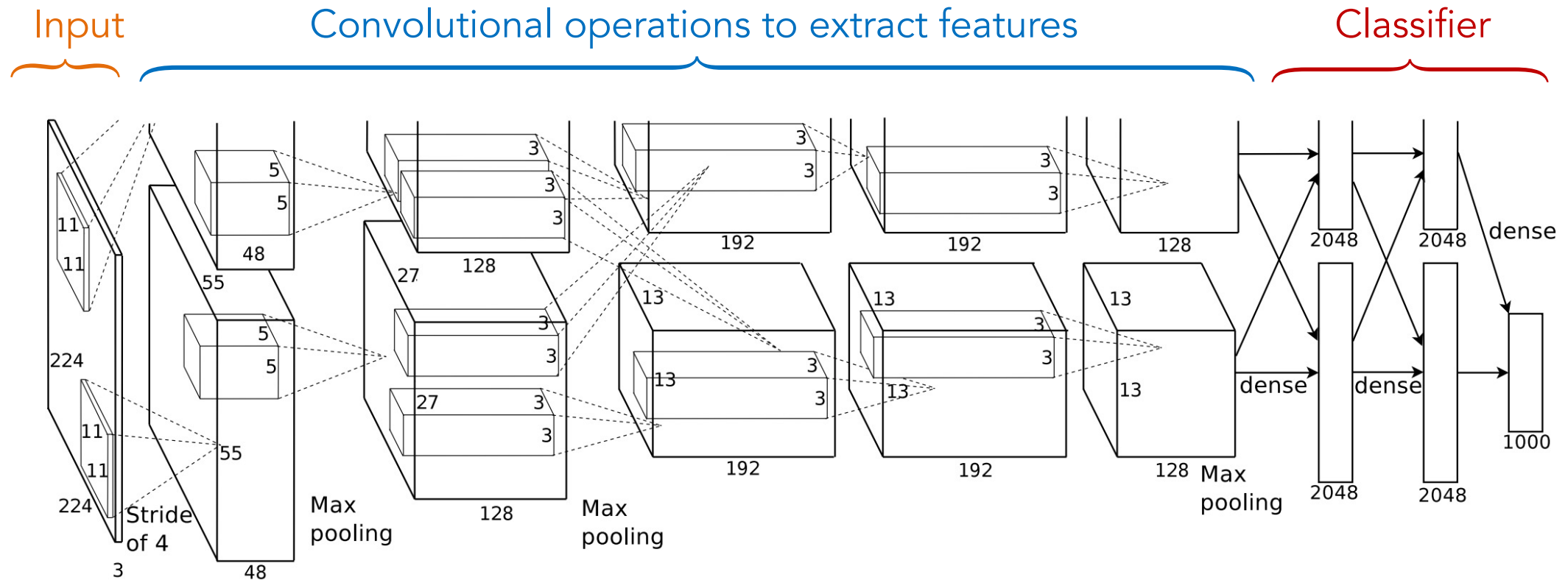


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

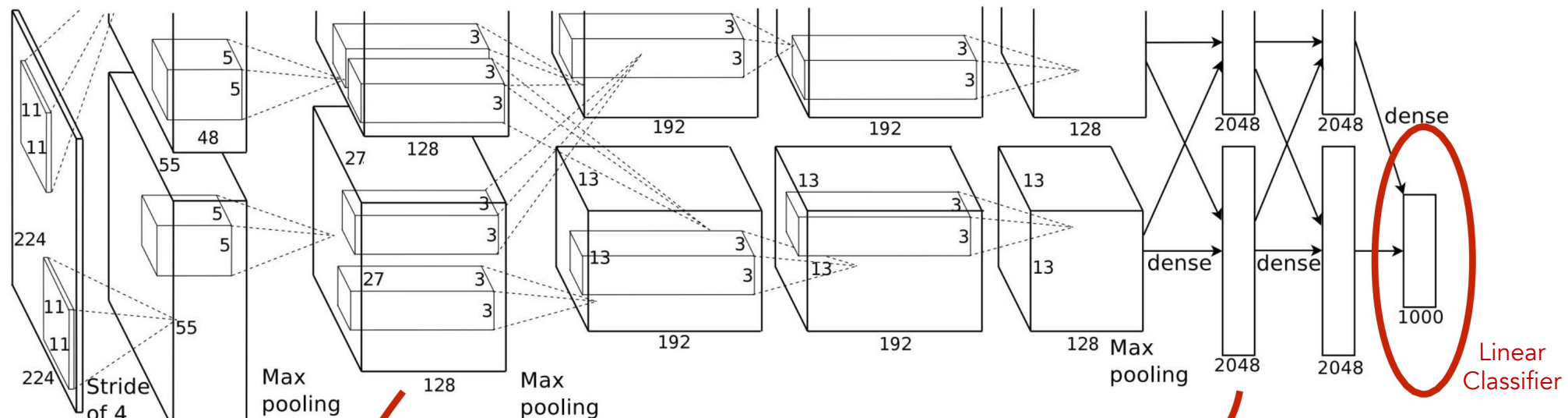
On 2012, a breakthrough paper, AlexNet, showed that we can use **multiple GPUs** to run deep **Convolutional Neural Networks (CNN)** with significantly better performance in image classification (15.3% Top-5 error on ILSVRC2012 challenge, next best was 25.7%).



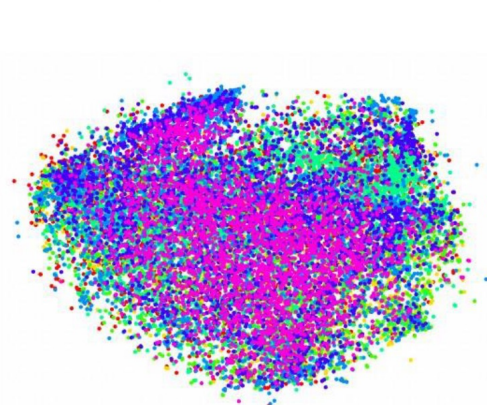
The convolutional parts of the architecture are used to **learn kernels/filters to extract features**. The last layer(s) of most CNNs are just **linear classifiers**.



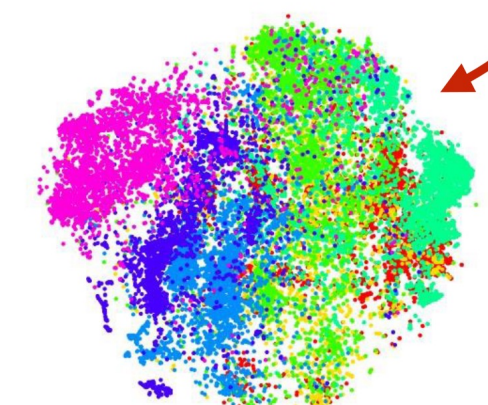
The data should be **linearly separable** by the time it reaches the end of the network.



- structure, construction
- covering
- commodity, trade good, good
- conveyance, transport
- invertebrate
- bird
- hunting dog



(c) DeCAF₁



(d) DeCAF₆

Donahue, J., et al. (2014). [DeCAF: A deep convolutional activation feature for generic visual recognition](#). ICML.

Many learned CNN kernels from the first layer look similar to the hand-crafted kernels.

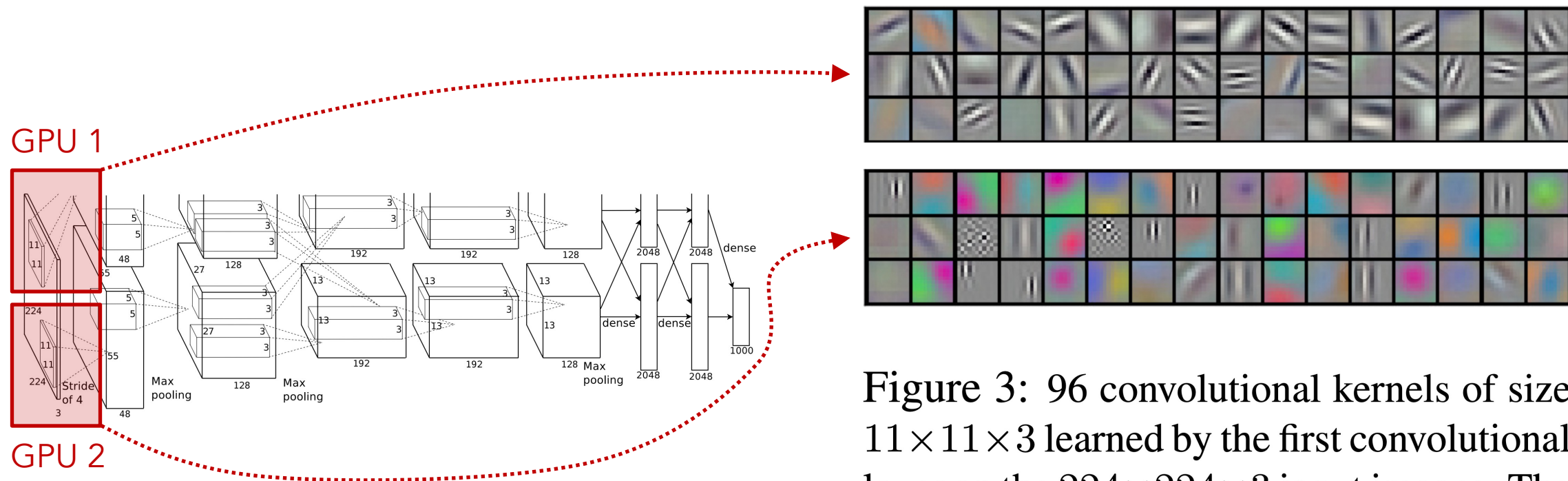
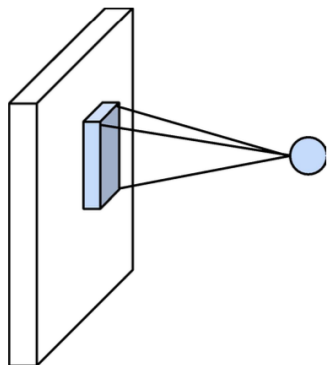


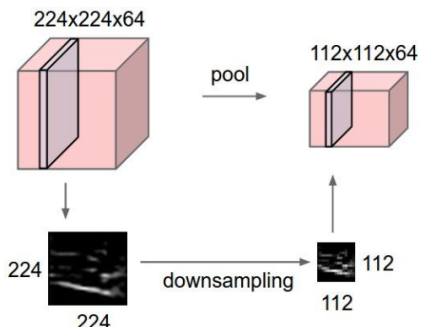
Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

Components of CNN typically involves convolutional layers, activation functions (e.g., ReLU), pooling layers (e.g., max pooling), fully connected layers, and normalization.

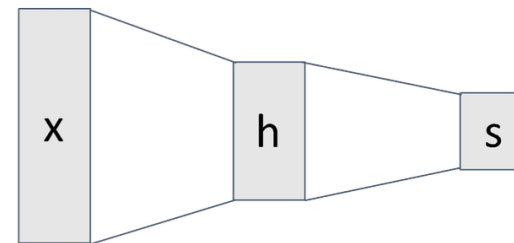
CONV
Convolution Layers



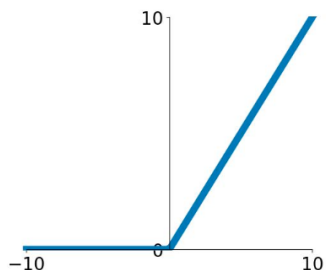
POOL
Pooling Layers



FC
Fully-Connected Layers



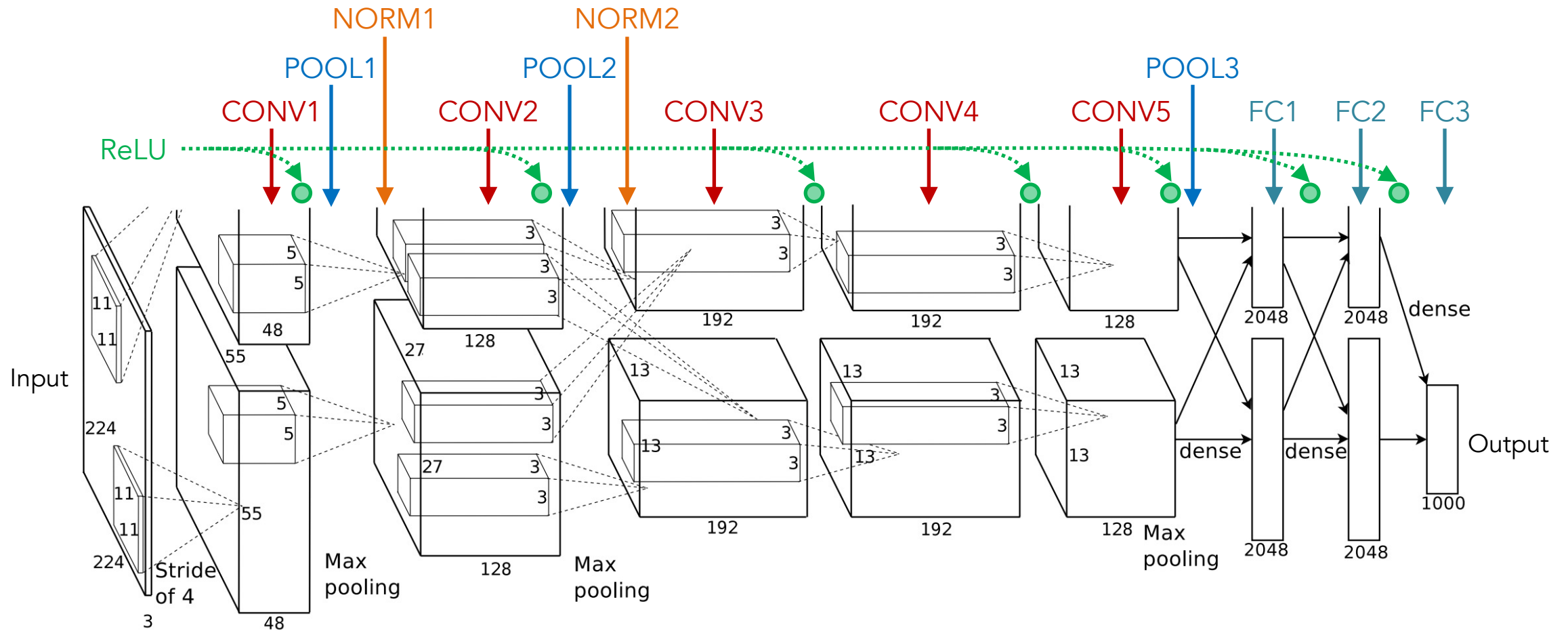
ReLU
Activation Function



NORM
Normalization

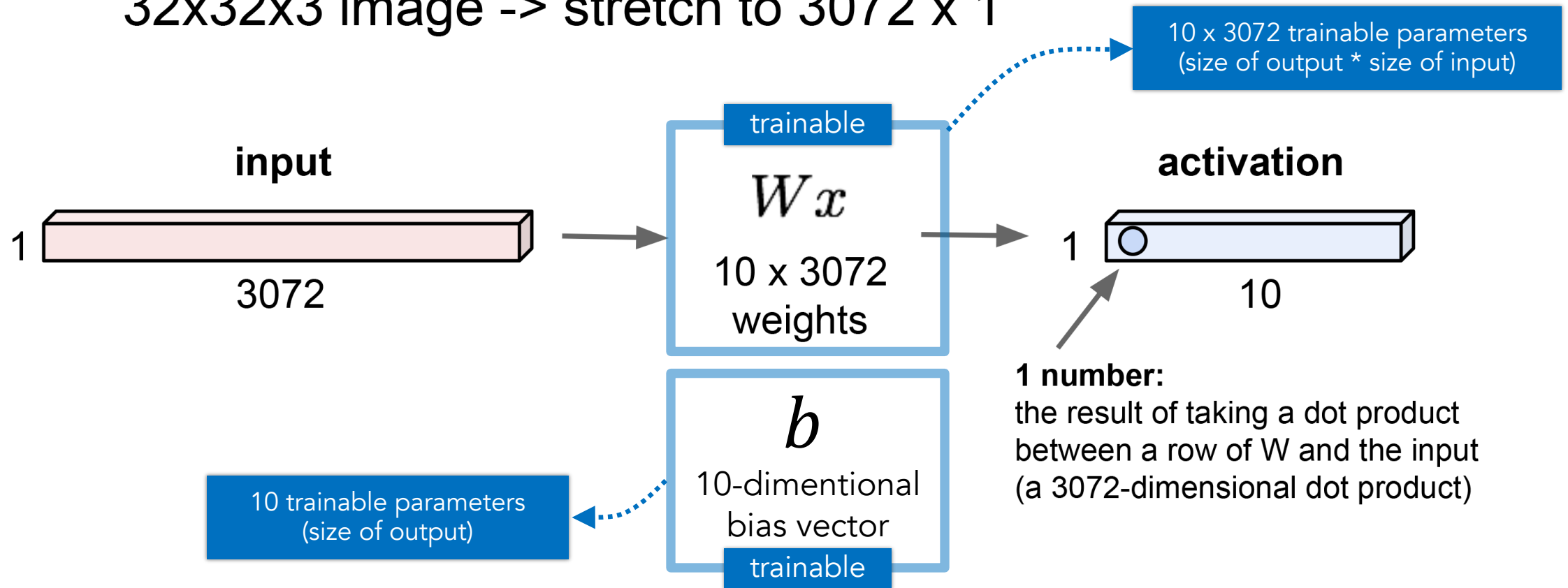
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

There are many ways of combining these CNN components, below is an example.

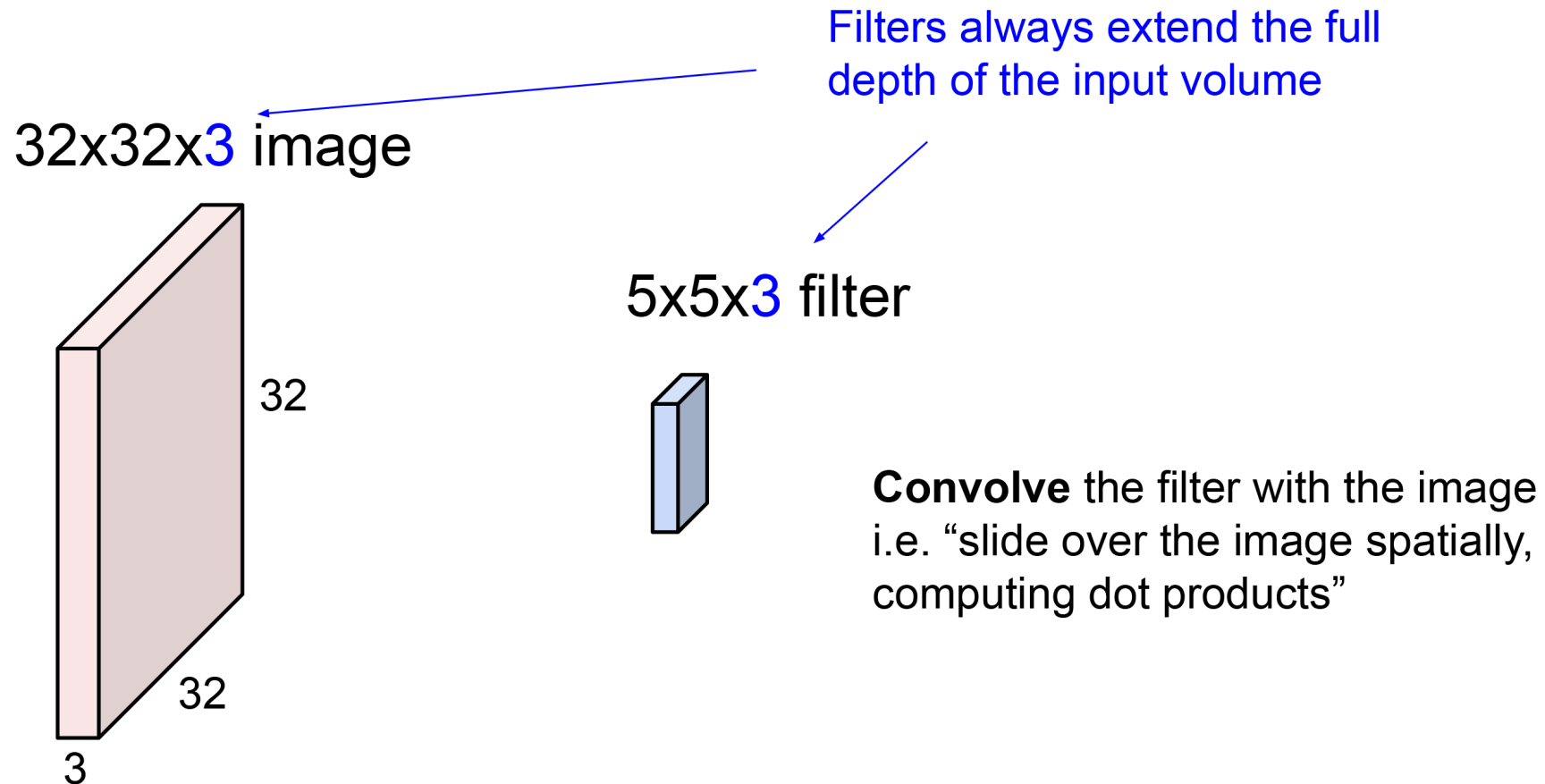


The **fully connected layer** flattens a feature map (image) to a 1-dimensional vector which is then passed to an activation function and goes to the next layer.

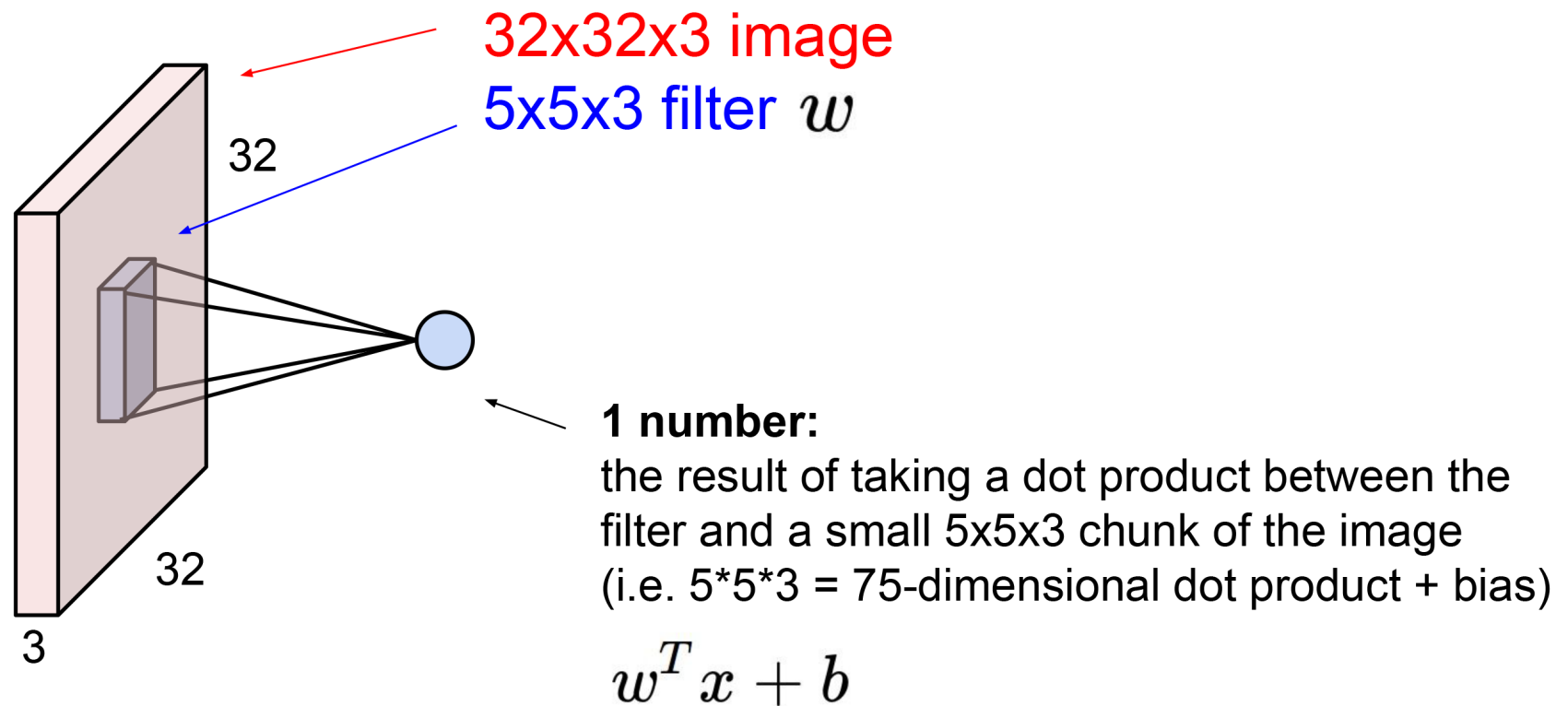
32x32x3 image -> stretch to 3072 x 1



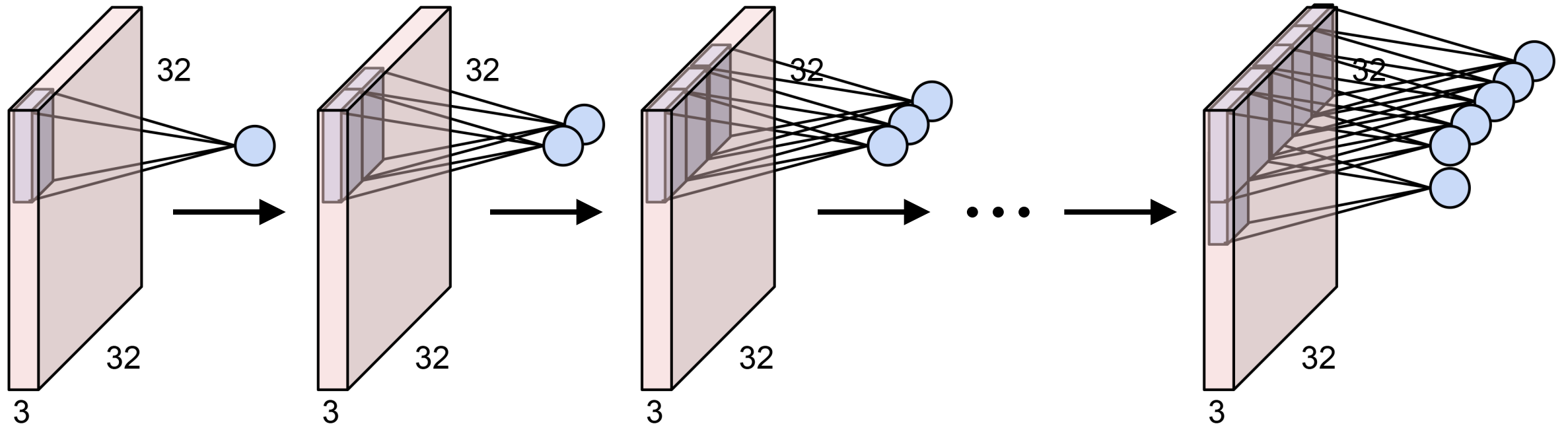
The **convolutional layers** in the CNN perform convolution operations as we discussed previously (i.e., using a box filter to blur an image).



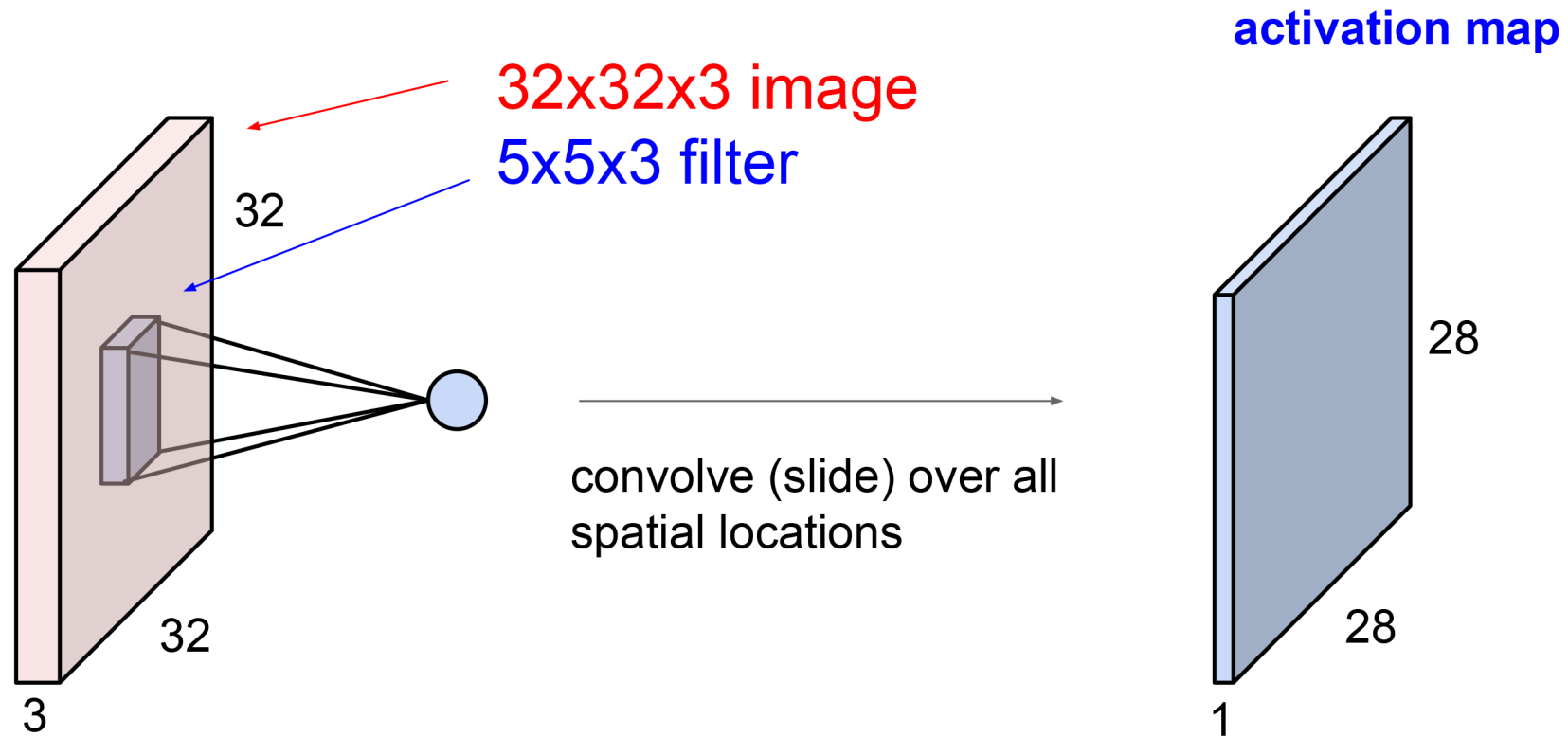
Each step in the convolutional operation produces **one number**, which is the sum of the element-wise multiplication, or it can also be seen as a dot product of two tensors.



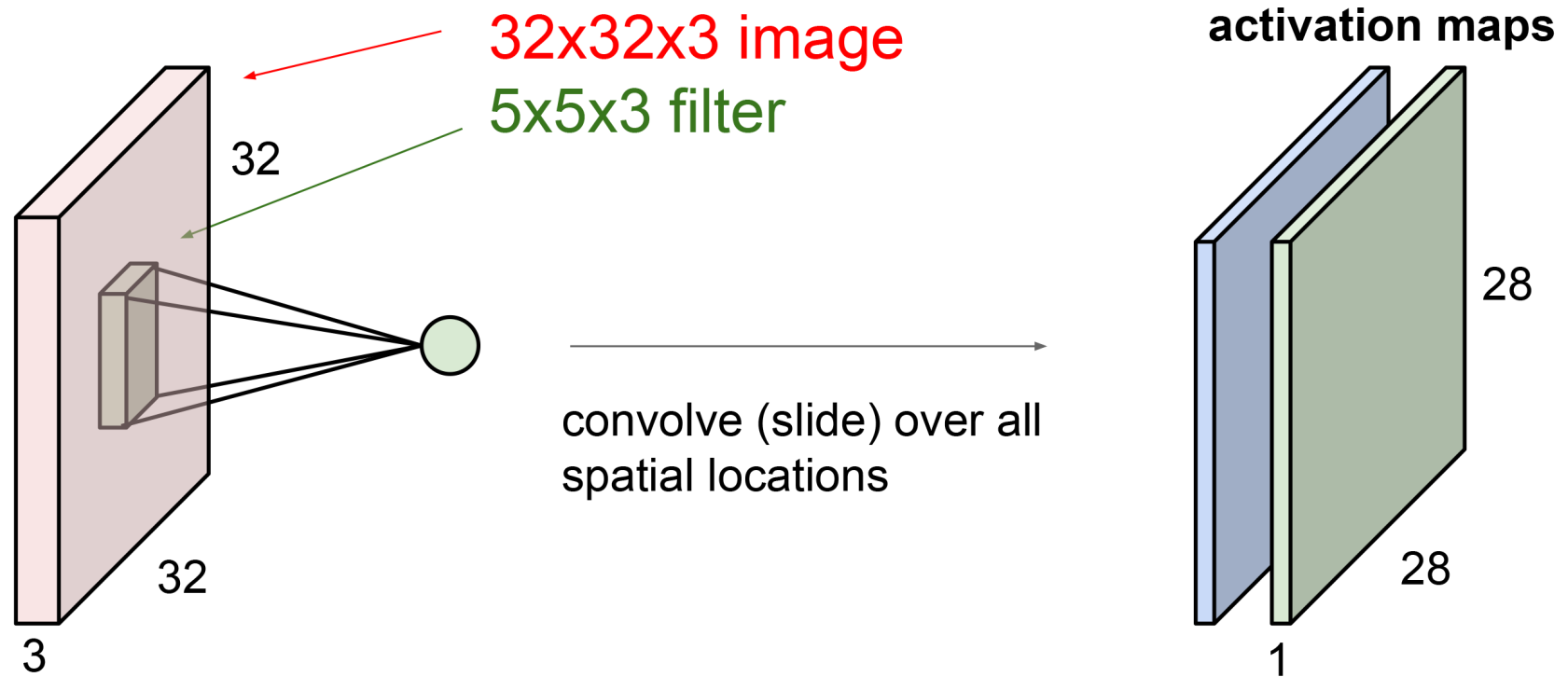
We then repeatedly slide the convolution kernel over the input feature map (or images).
The result is a new matrix of numbers.



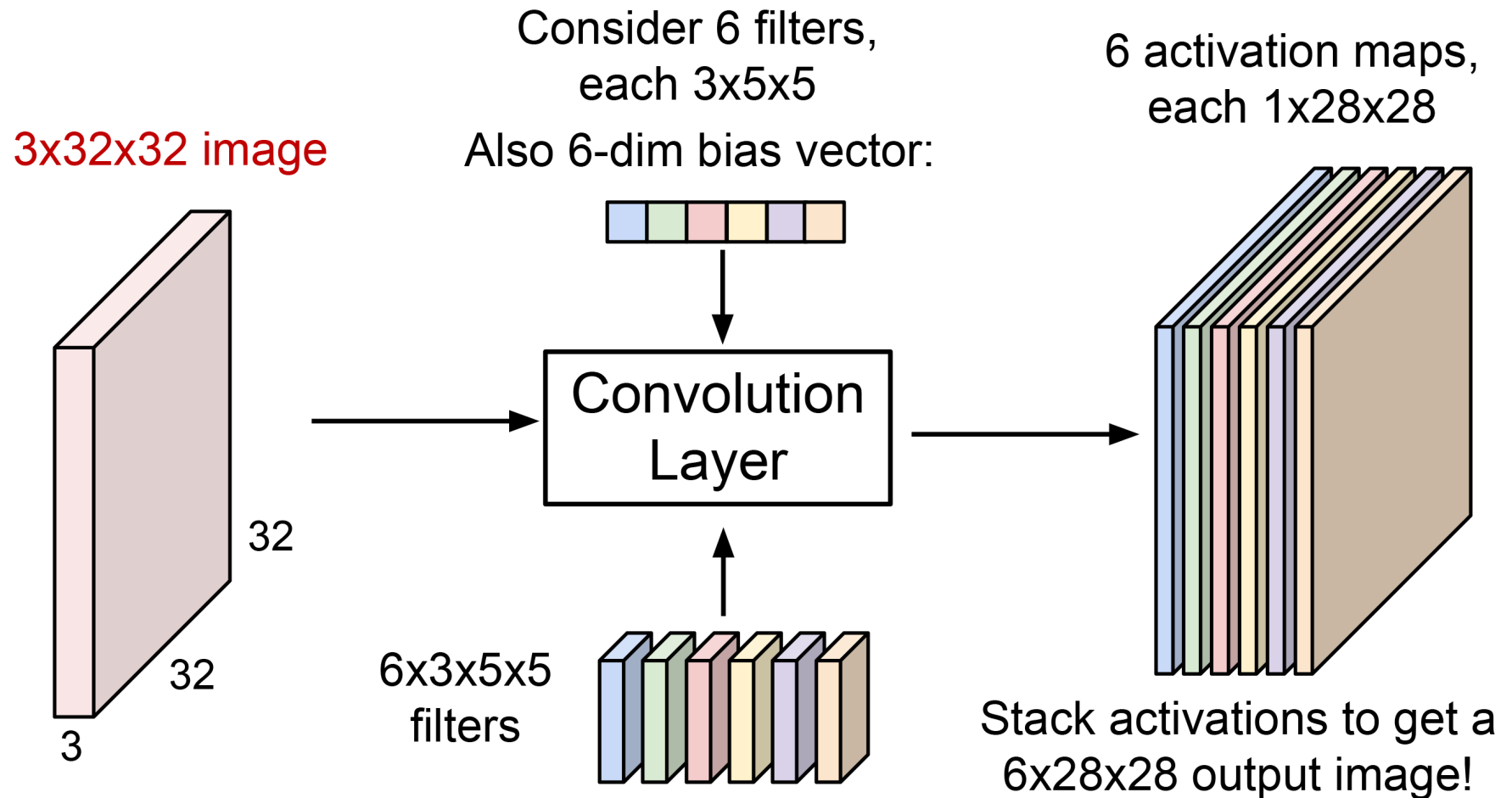
For each kernel, after convolution, we get a **feature map (or activation map)**. The input and output image sizes are different due to how we slide the convolutional kernel.



If we use another convolutional kernel (indicated in green color below) and slide the kernel over the input image again, we obtain **another feature (activation) map**.

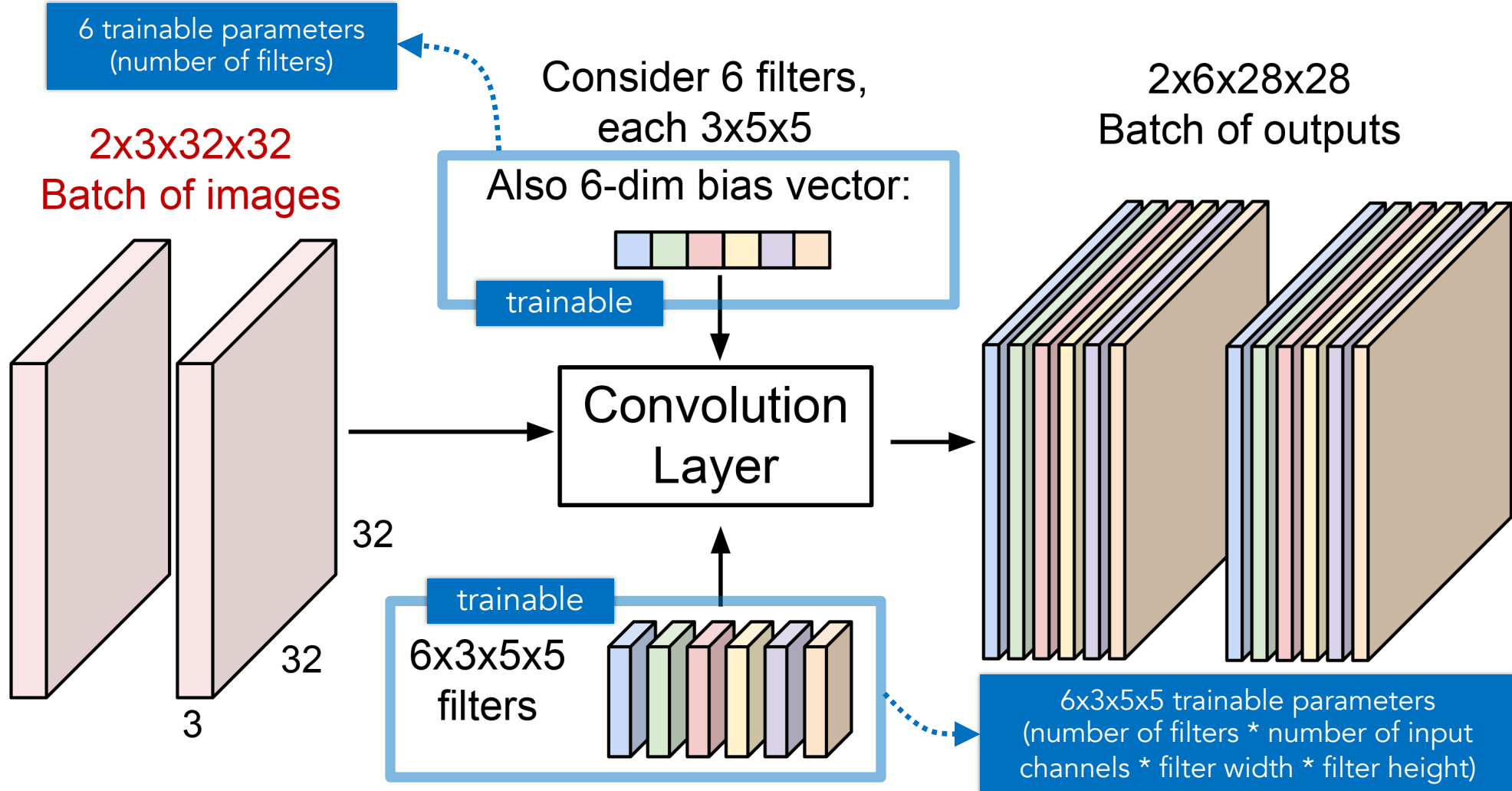


We can repeat this process many times, and we will **get a bunch of feature (activation) maps**. The depth of the feature map depends on the number of filters/kernels.

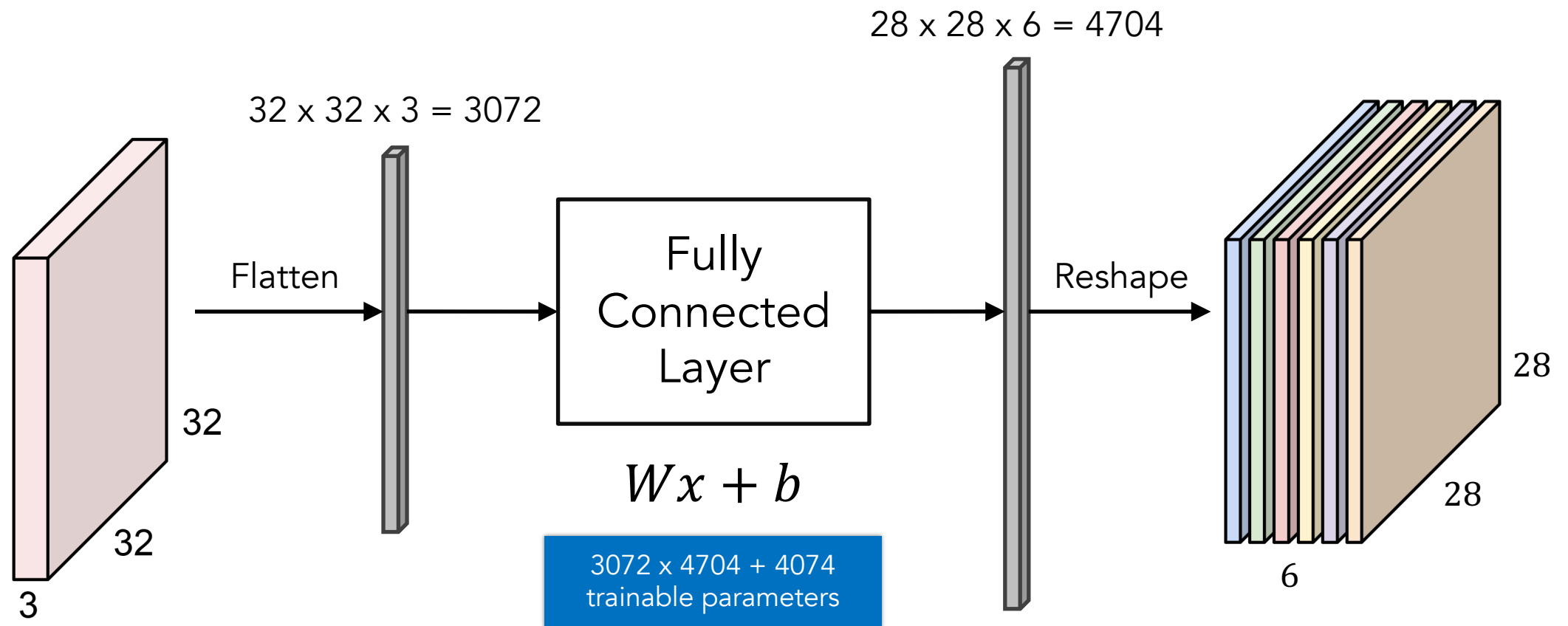


Depending on the size of a mini-batch, we obtain **multiple batches of feature maps**.

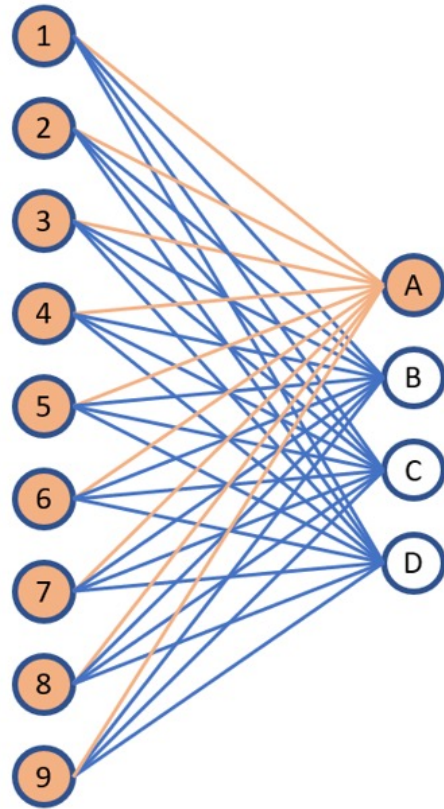
Notice that we use a lot of data to train the bias vector and the kernels/filters.



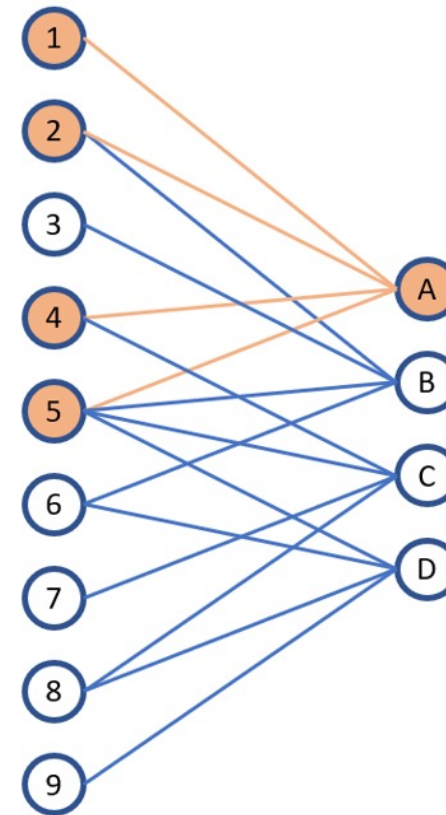
In the previous slide, we have 456 trainable parameters using the convolutional layer. Without convolution, using a fully connected layer gives more than 14M parameters.



So, convolutional layers can also be seen as a way to **reduce the number of trainable parameters** (compared to fully connected layers) by only looking at a local region.

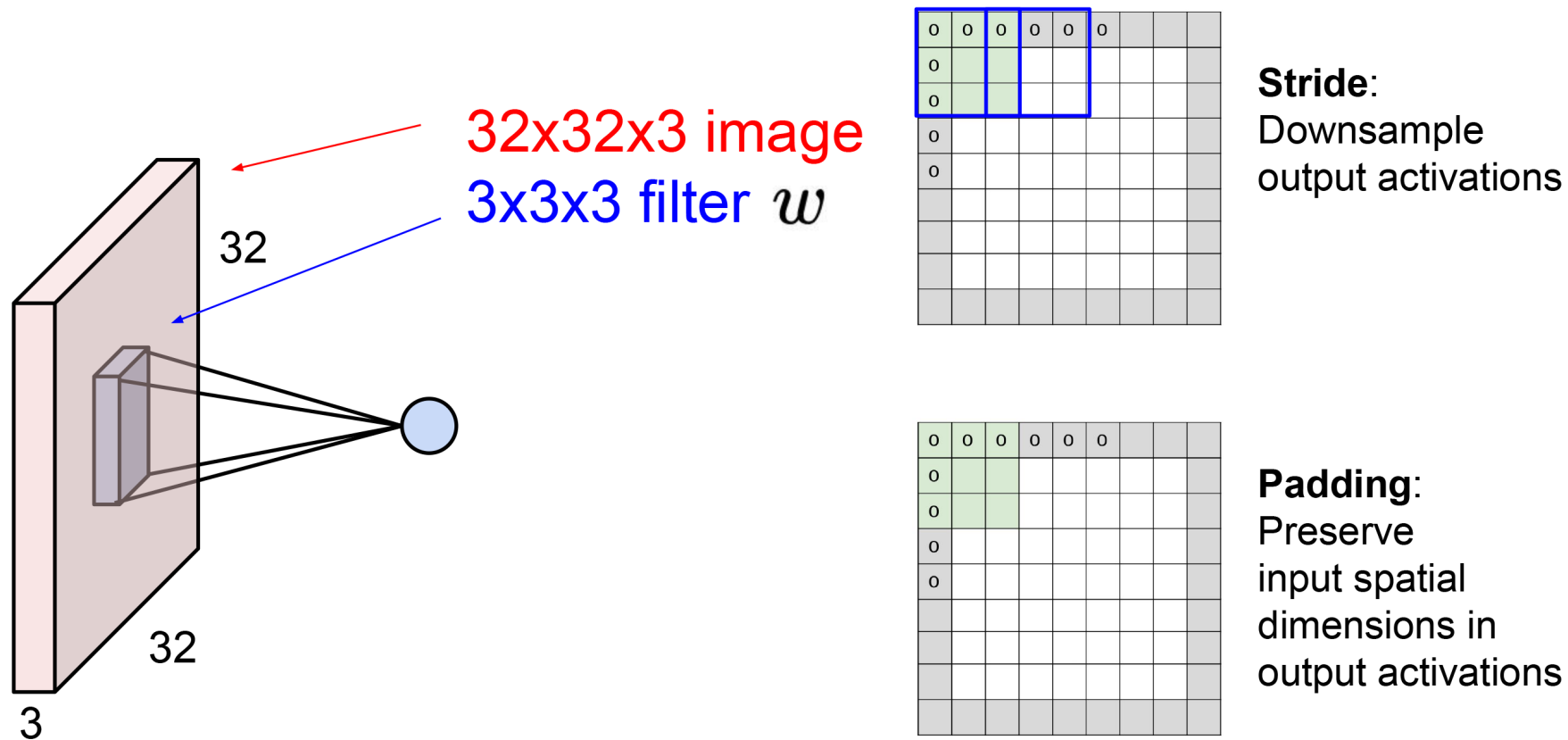


Fully Connected Layer

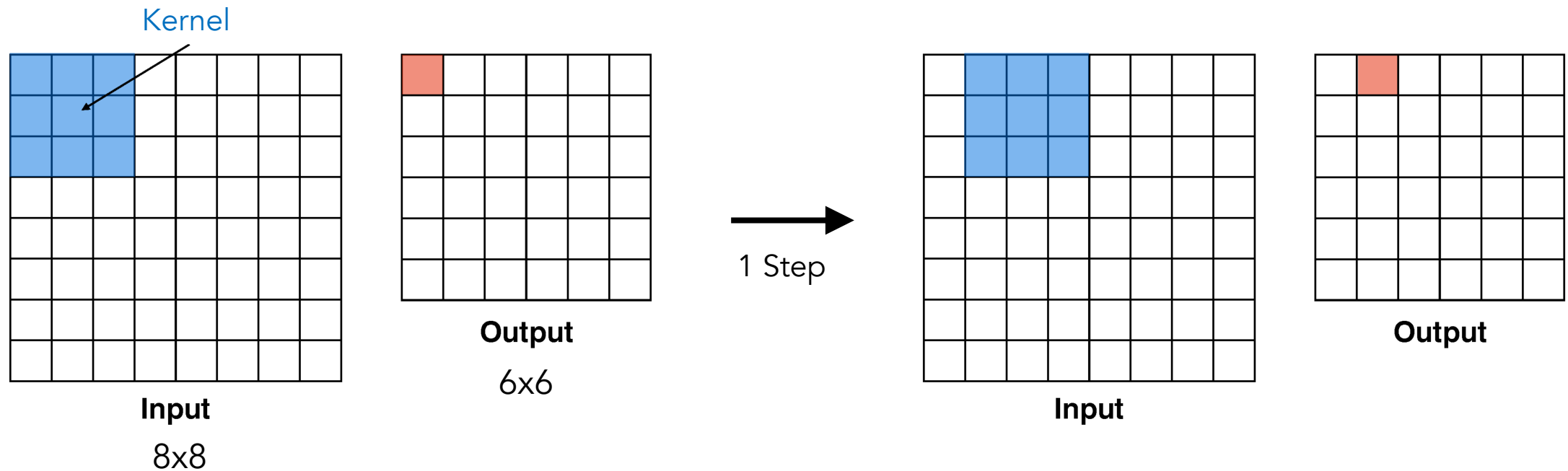


Convolutional Layer

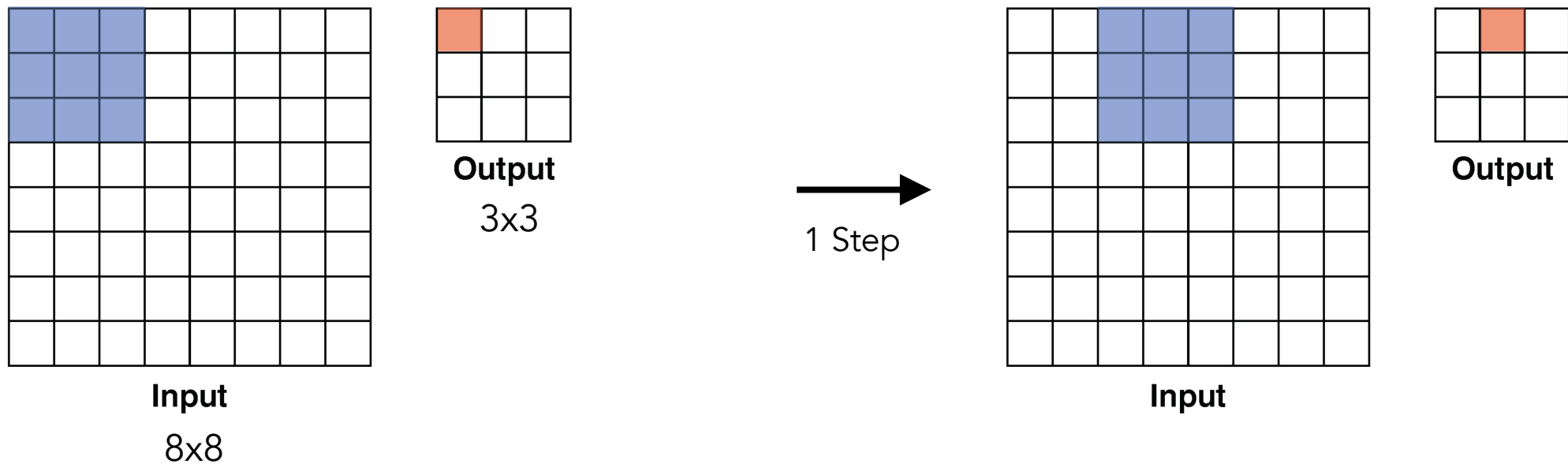
Convolution operations consider **stride** and **padding**. Stride means the number of steps when moving the filter. Padding means adding zeros around the input feature map.



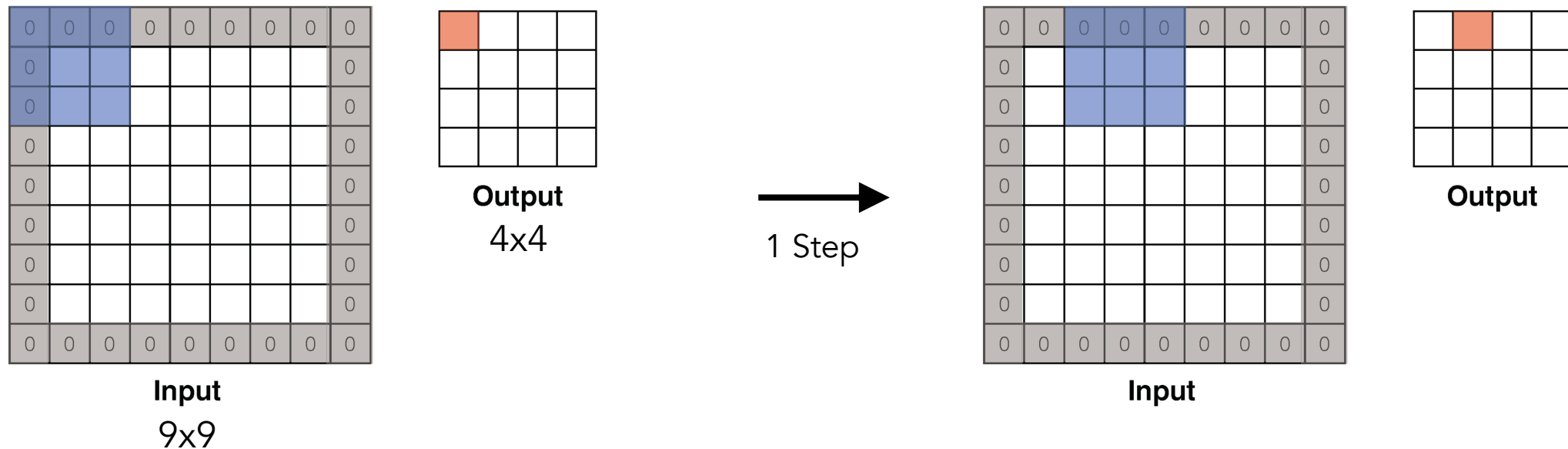
Below is a typical example of **no padding with stride 1 using kernel size 3**. Notice that the sizes of input (before convolution) and output (after convolution) are different.



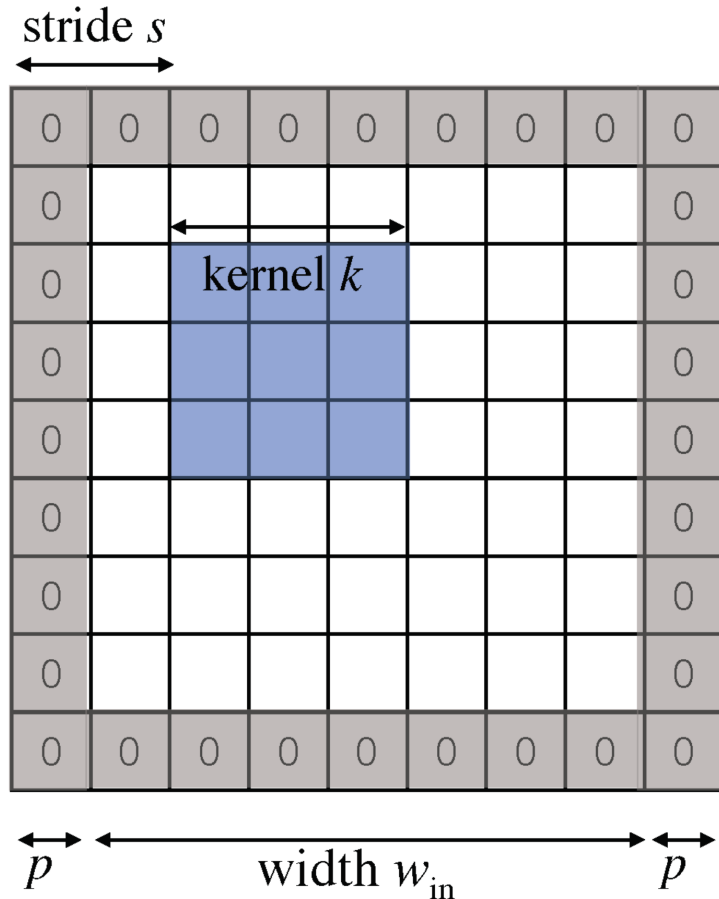
Below is another example of **no padding with stride 2** using **kernel size 3**, which means we slide the kernel 2 pixels (both horizontally and vertically) for each convolutional step.



We can also pad the input with zeros (i.e., adding zeros around the input). Below is an example of padding 1 with stride 2 using kernel size 3.



Below is the formula to calculate the size of output w_{out} after the convolution operation (of input size w_{in}) with different padding p , kernel size k , and stride s .



In general, the output has size:

$$w_{out} = \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1$$

$\lfloor x \rfloor$ is the mathematical floor operation, which gives the largest integer that is less than or equal to x , for example, $\lfloor 5.2 \rfloor = 5$

Example: $k=3, s=1, p=1$

$$\begin{aligned} w_{out} &= \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1 \\ &= \left\lfloor \frac{w_{in} + 2 - 3}{1} \right\rfloor + 1 \\ &= w_{in} \end{aligned}$$

In practice, we usually pick a particular combination of padding and stride to **keep the input and output the same size** (for convenience).

| | | | | | | | | |
|---|---|---|---|---|---|--|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 | | | |
| 0 | | | | | | | | |
| 0 | | | | | | | | |
| 0 | | | | | | | | |
| 0 | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output?

7x7 output!

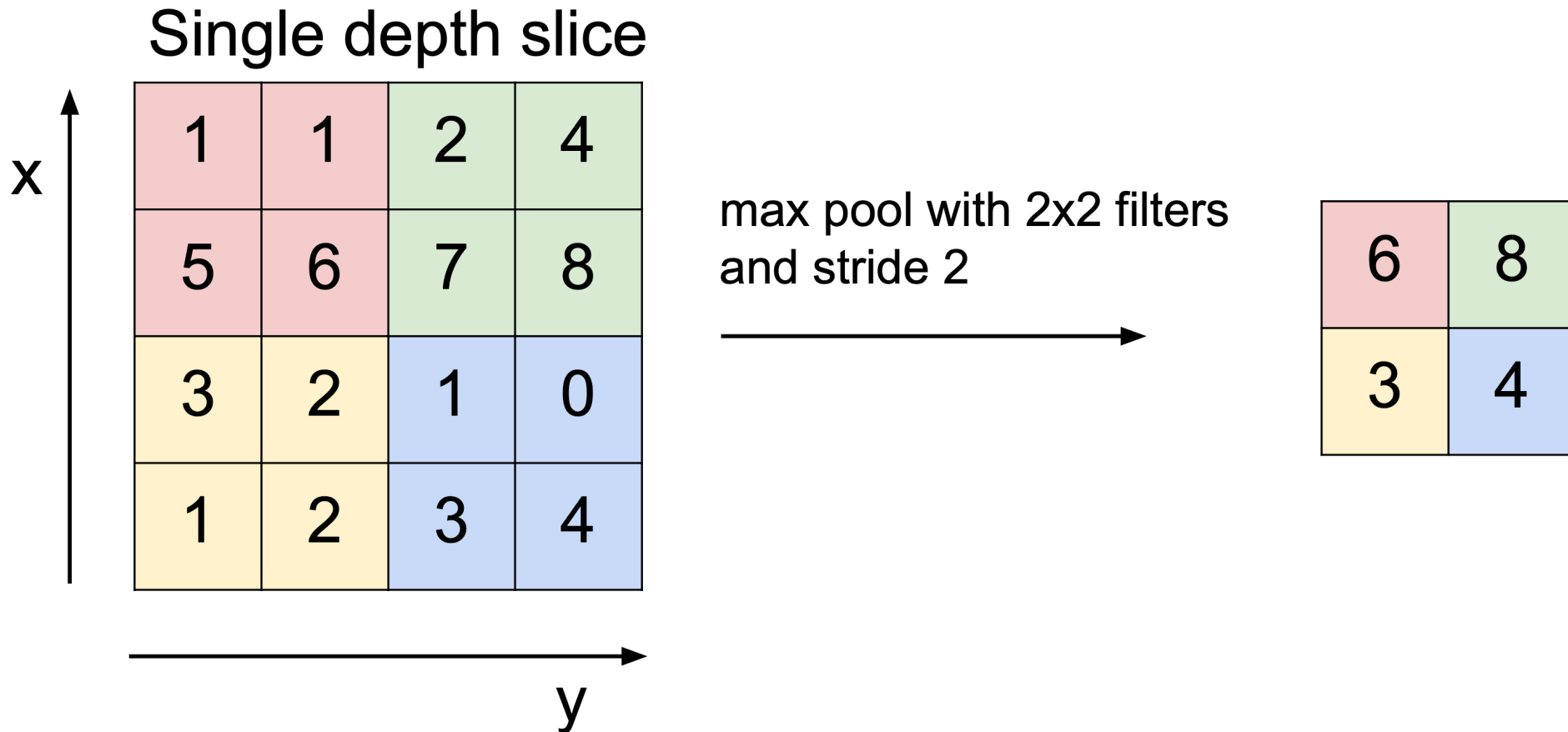
in general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

e.g. $F = 3 \Rightarrow$ zero pad with 1

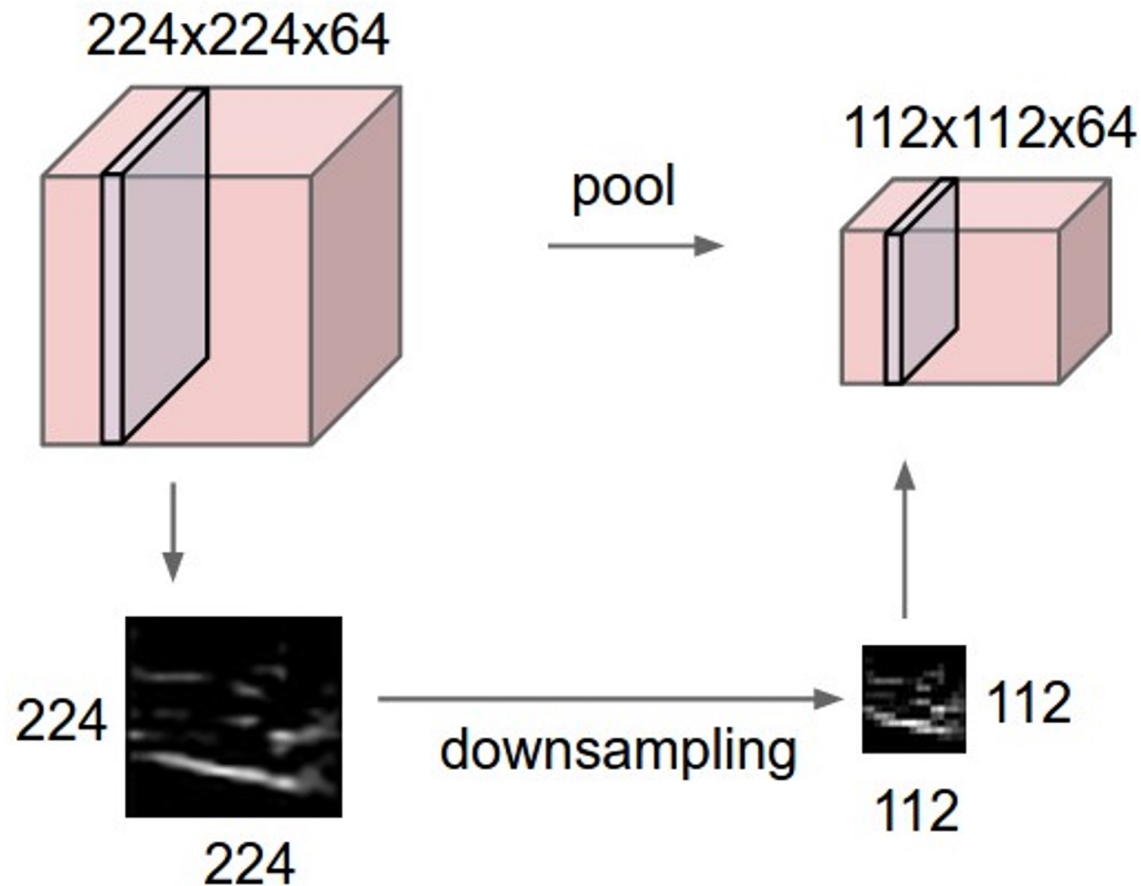
$F = 5 \Rightarrow$ zero pad with 2

$F = 7 \Rightarrow$ zero pad with 3

The **max pooling layer** is designed to have the neural network **pay attention to the most important information** by taking the maximum value in a convolution window.

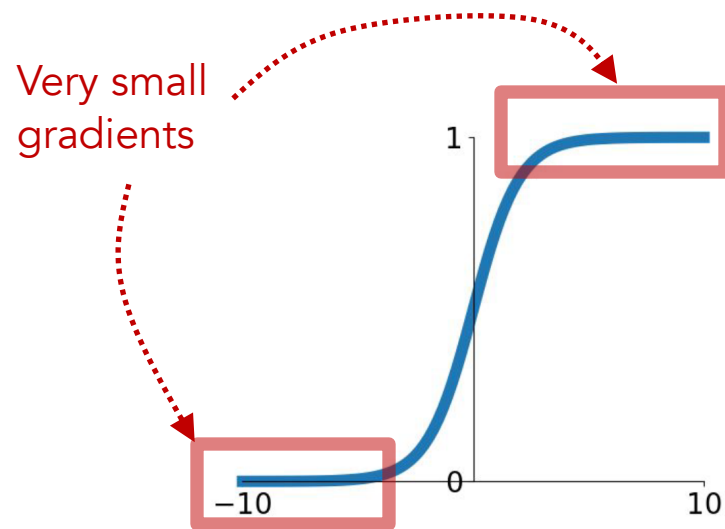


The max pooling layer **reduces the size** of each feature (activation) map independently. Notice that there are **no learnable/trainable parameters** in the max pooling layers.



Hyperparameters:
Kernel Size
Stride
Pooling function

The **activation function** in the neural network is designed to **introduces non-linearity**, such as the sigmoid activation function below.



Sigmoid

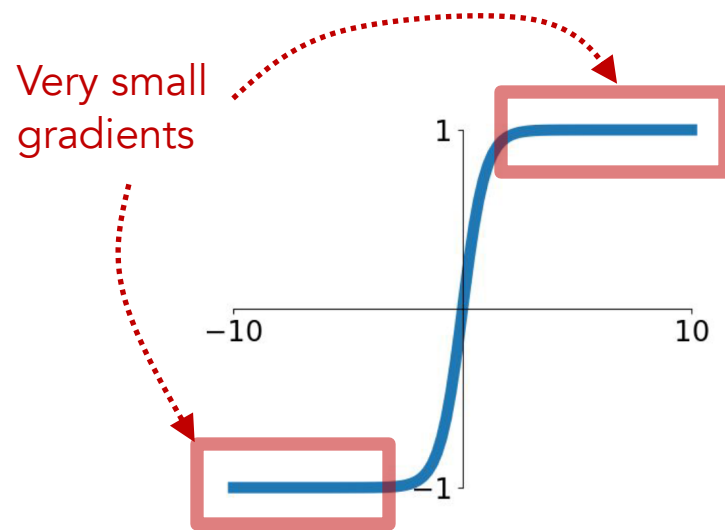
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. $\exp()$ is a bit compute expensive

We can use the tanh function instead of the sigmoid function to mitigate some problems, but the gradients still saturate (which can lead to vanishing gradients).

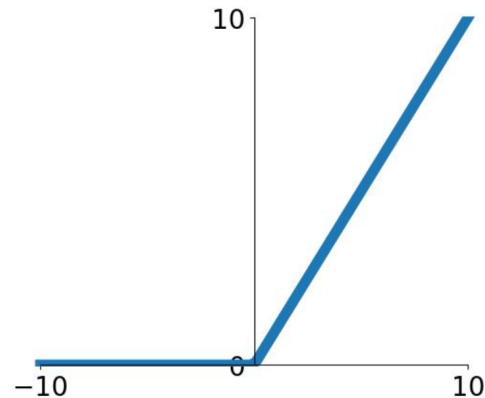


$\tanh(x)$

- Squashes numbers to range $[-1,1]$
- zero centered (nice)
- still kills gradients when saturated :(

Typically, the saturating gradient problem can be fixed by using the ReLU activation.
But the gradient when $x < 0$ is zero, which lead to the dying ReLU problem.

$$f(x) = \max(0, x)$$



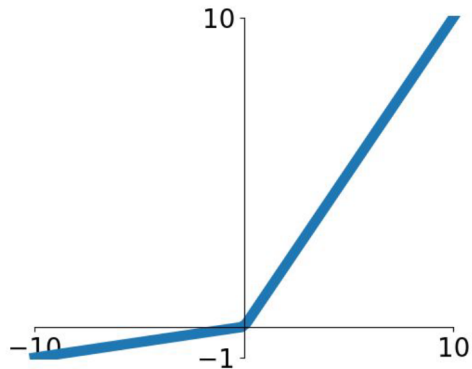
ReLU
(Rectified Linear Unit)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

One way to mitigate the dying ReLU problem is to use a leaky ReLU instead, where the negative value regions still have a slight slope. But in practice people still use ReLU.



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”**.

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

The normalization layer normalizes a certain region (e.g., the blue region below) of the feature maps to zero mean and unit variance. A typical example is Batch Normalization. Notice that there are trainable parameters in Batch Norm (see the paper for details).

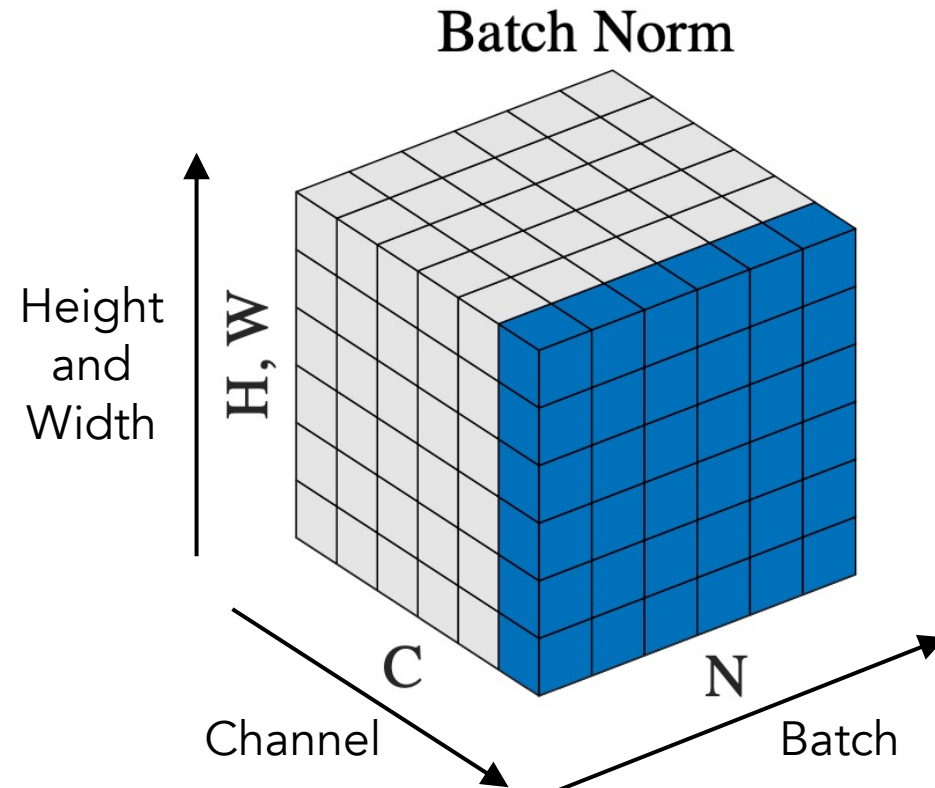
Normalization

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

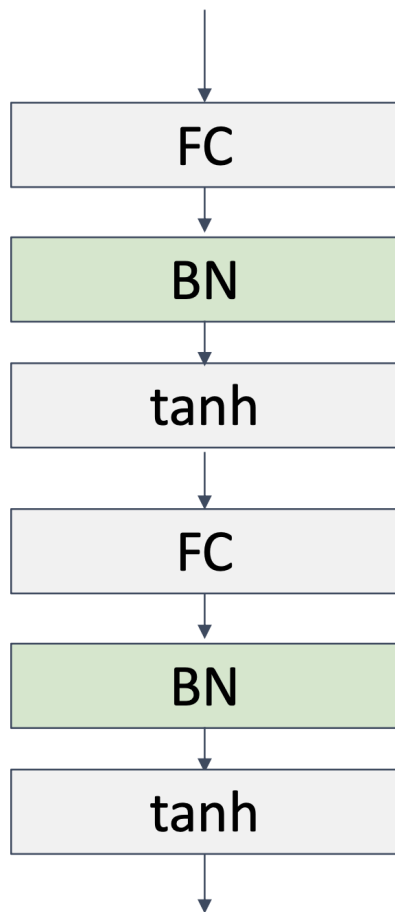
μ_j Mean of the region

σ_j^2 Variance of the region

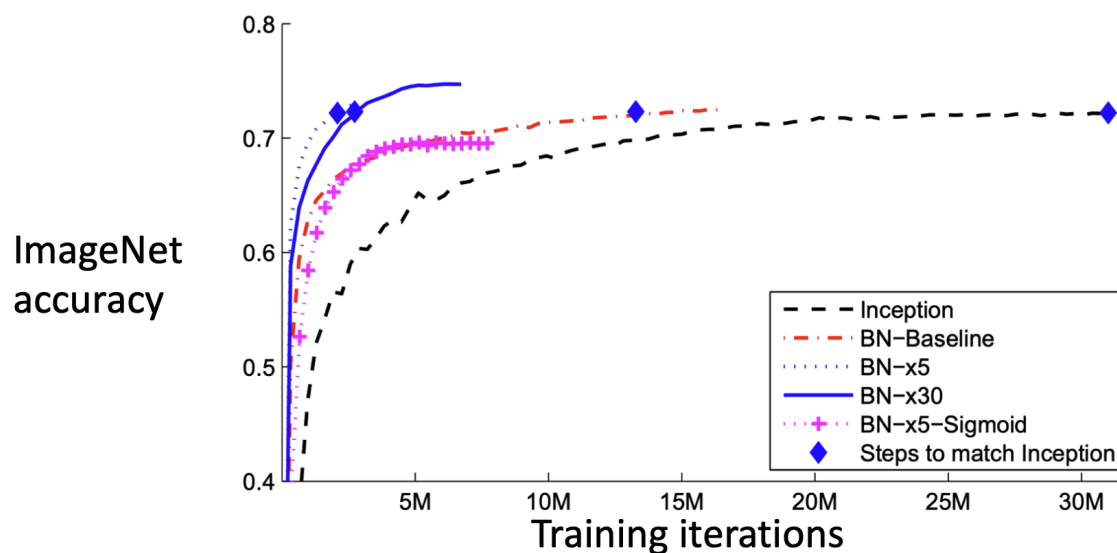
ϵ For numerical stability



Batch Normalization is usually inserted after convolutional (or fully connected) layers and before the activation function (the non-linearity), which has advantages below.

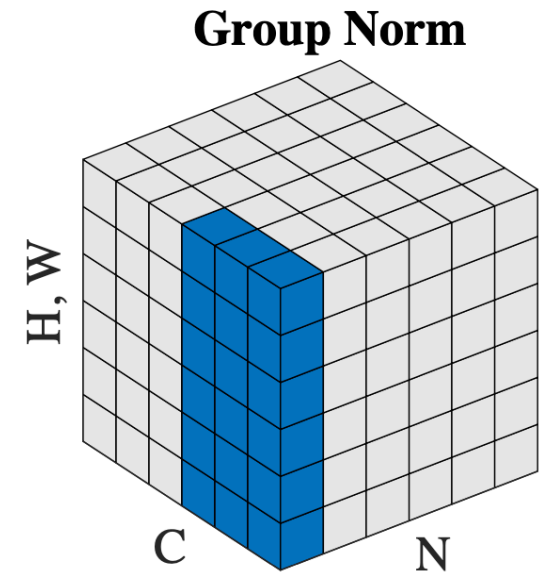
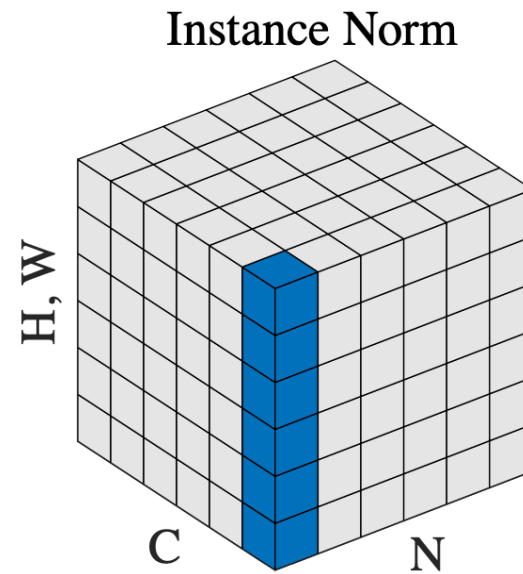
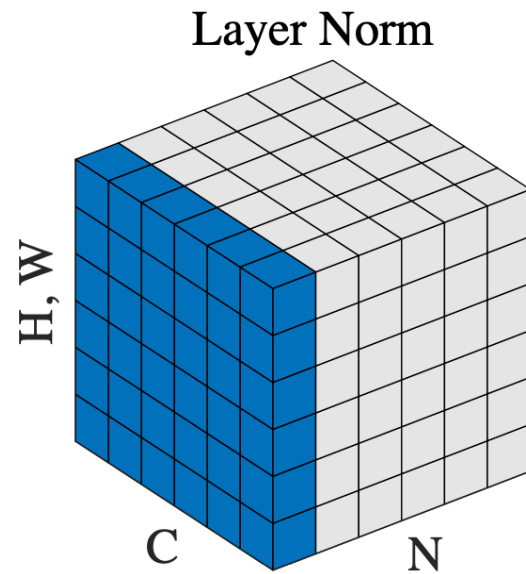
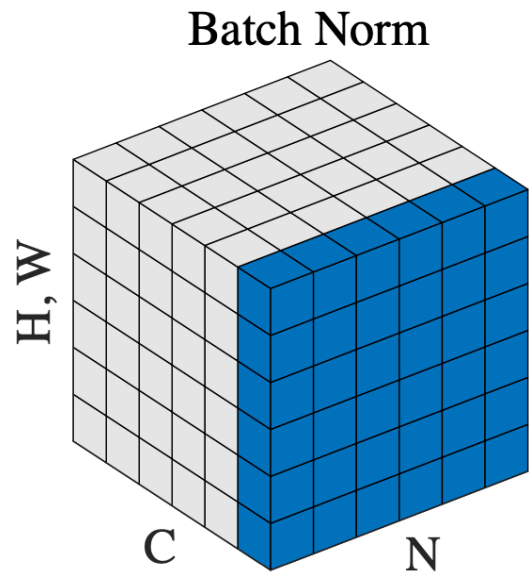


- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training



Ioffe, S., & Szegedy, C. (2015, June). [Batch normalization: Accelerating deep network training by reducing internal covariate shift](#). ICML.

There are many ways for normalization. Each cube below shows a feature map tensor. Pixels in blue are normalized by the same mean and variance that are computed from the values of these blue pixels (C is channel, N is batch, H and W are height and width).



But deep models are harder to train. In fact, **deep models underfit the data**. They perform worse in training and testing than the shallow models.

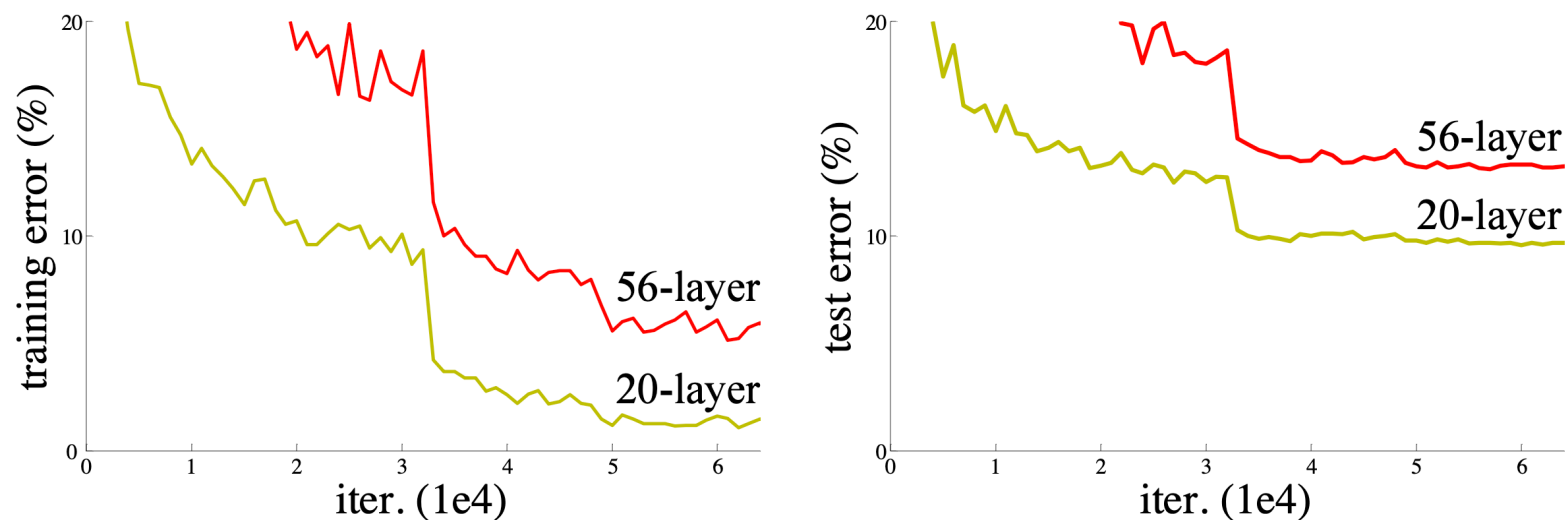
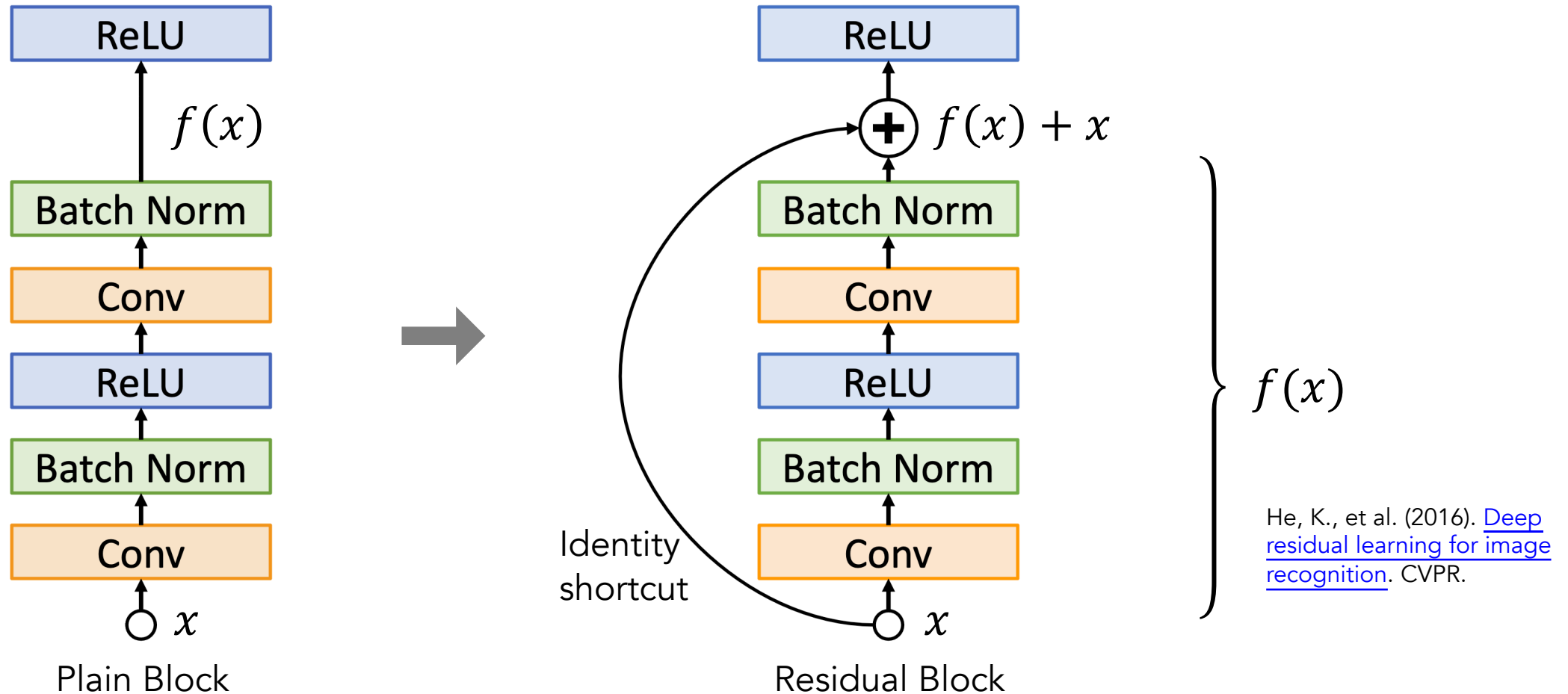
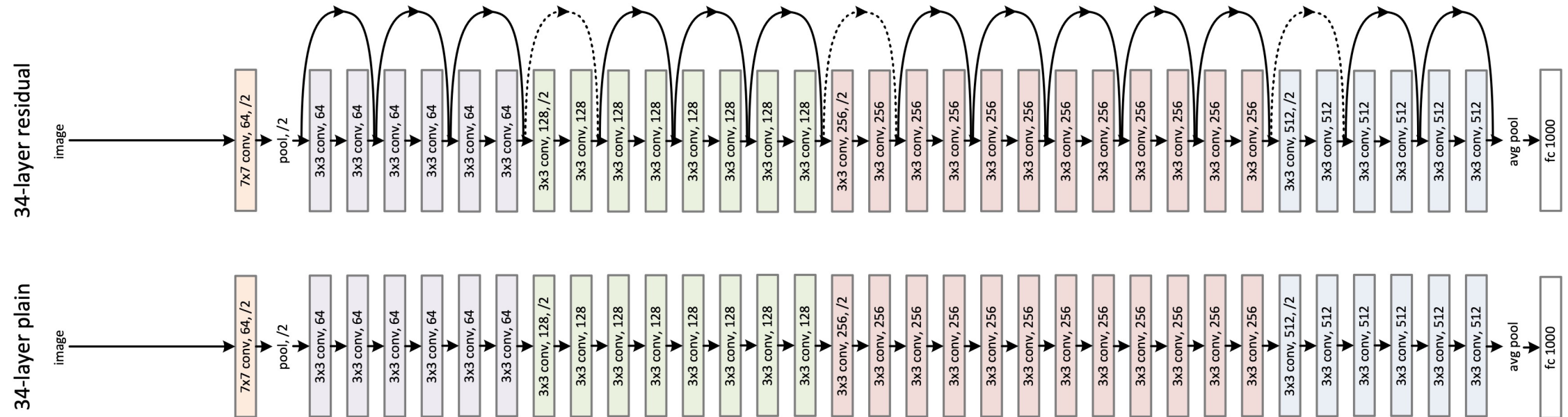


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error.

Intuitively, deep models should perform at least as good as shallow models by copying layers from the shallower model and setting extra layers to identity (i.e., set $f(x) = 0$).



By stacking many residual blocks, we can build the residual network architecture (i.e., ResNet), which is a reasonable baseline for image classification.

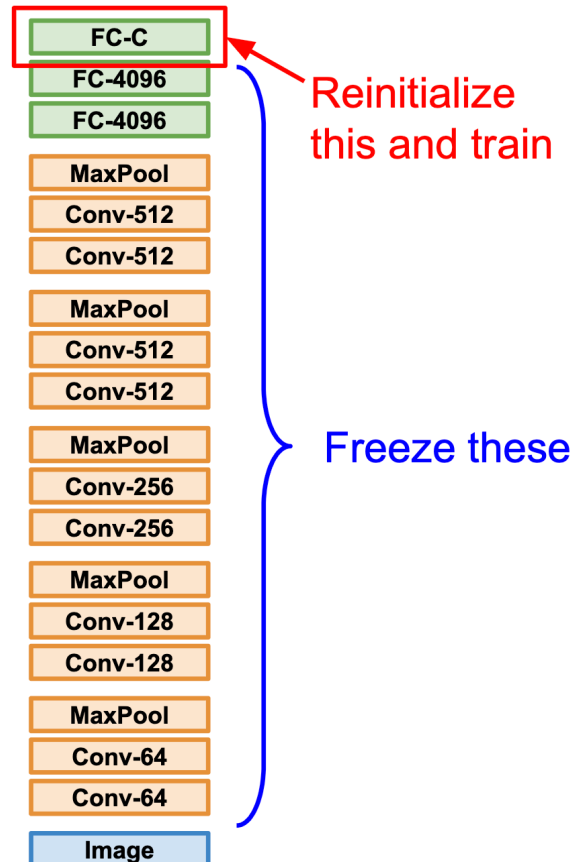


We often use pre-trained weights in similar or other tasks as a starting point (but not from scratch). This idea is called **Transfer Learning**, where we reuse prior knowledge.

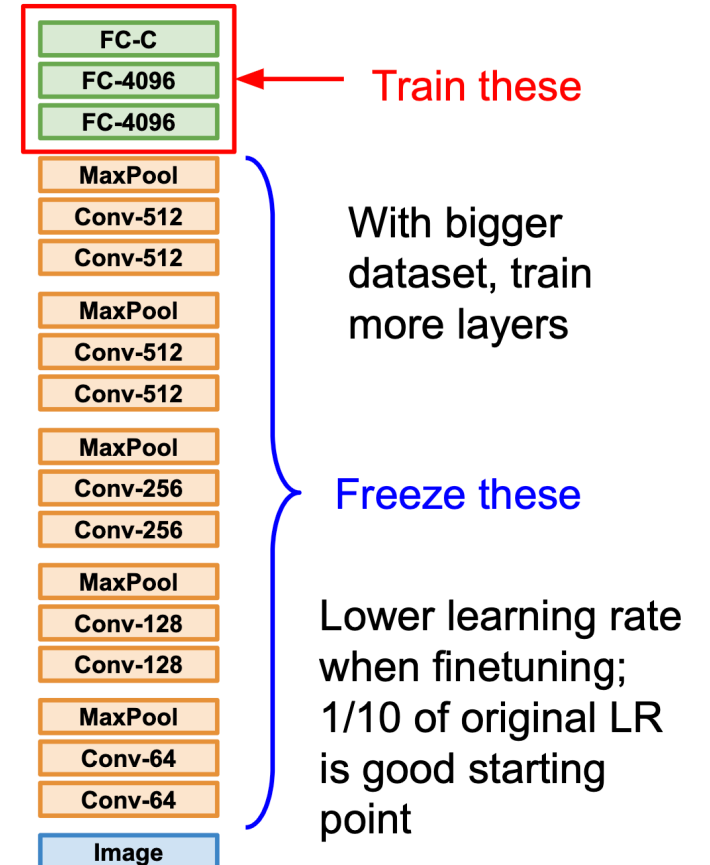
1. Train on Imagenet



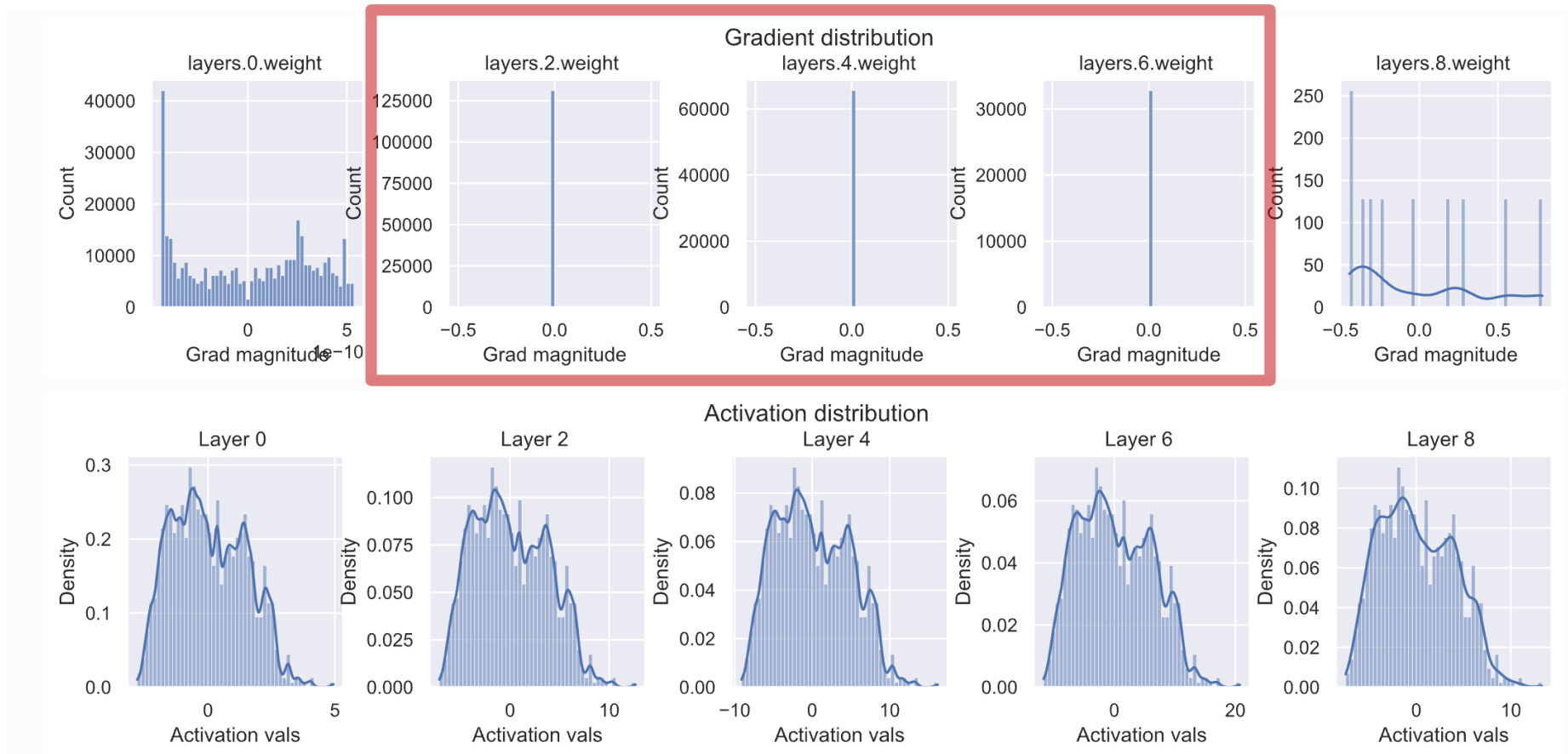
2. Small Dataset (C classes)



3. Bigger dataset

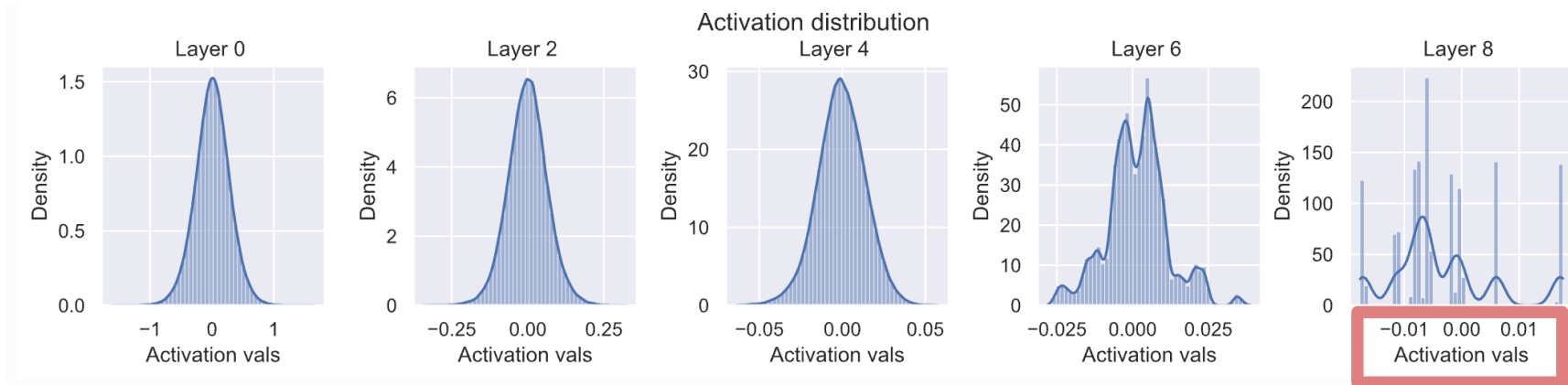


What if we need to train the model from scratch? One (bad) idea is to **initialize the weights with a constant** (or zero). But this will cause layers to have the same gradient (or no gradient when weights are zero) during training.



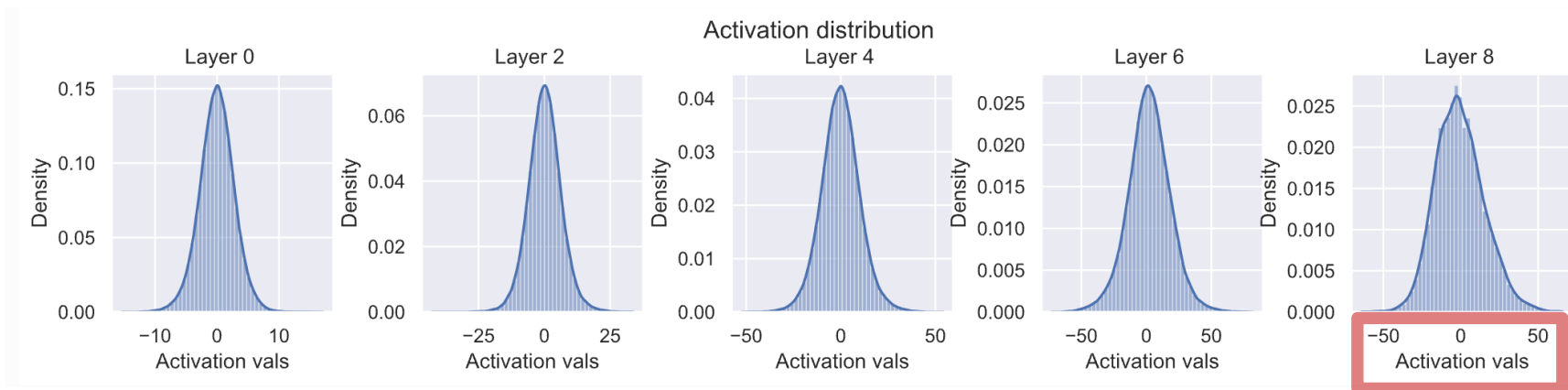
Another idea is to initialize the model weights by randomly sampling from a Gaussian or uniform distribution **with a constant variance**. But this can cause the activation values to vanish or explode in the deeper layers (which means vanishing/exploding gradients).

Smaller variance



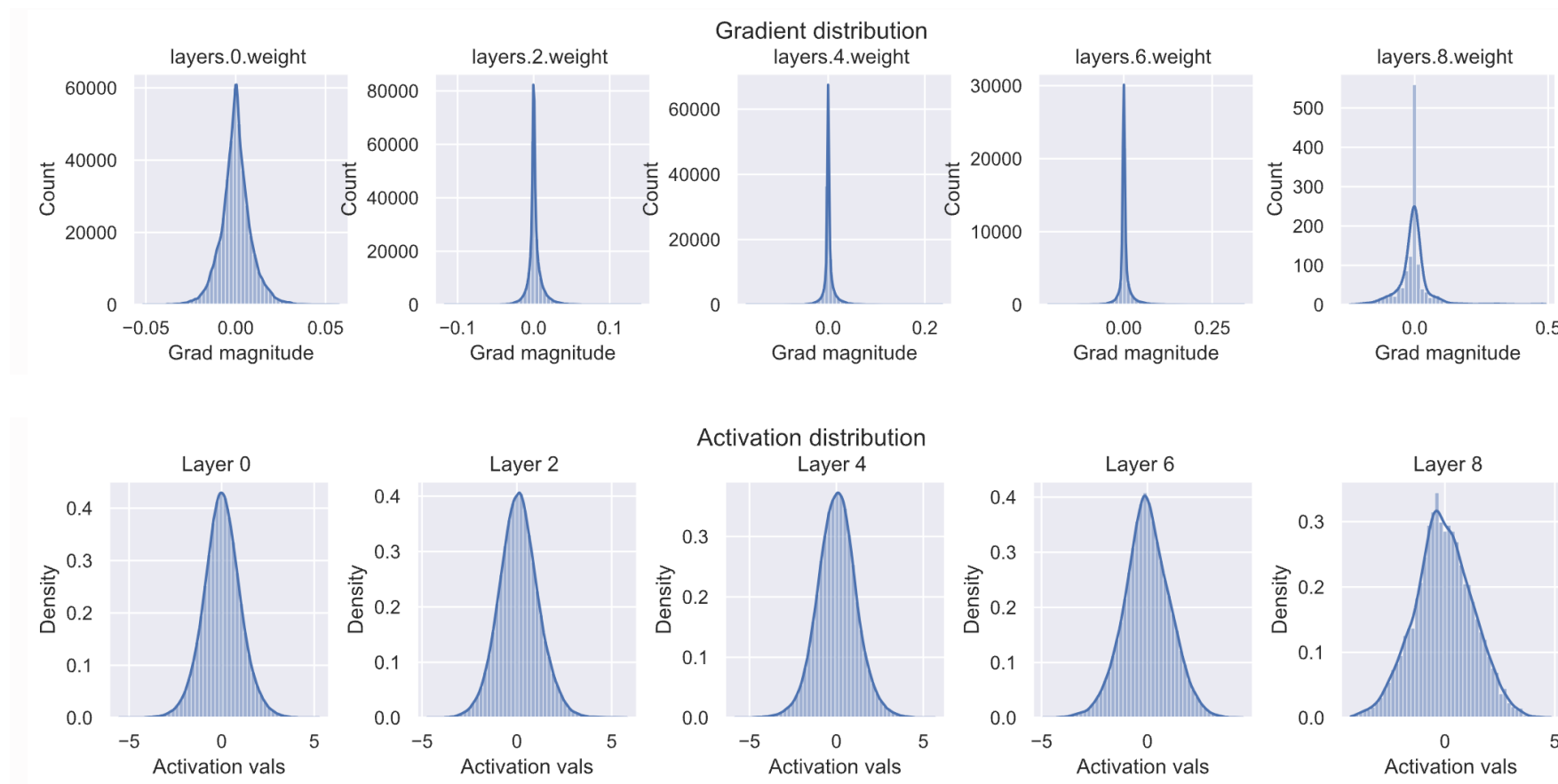
Vanishing values

Larger variance

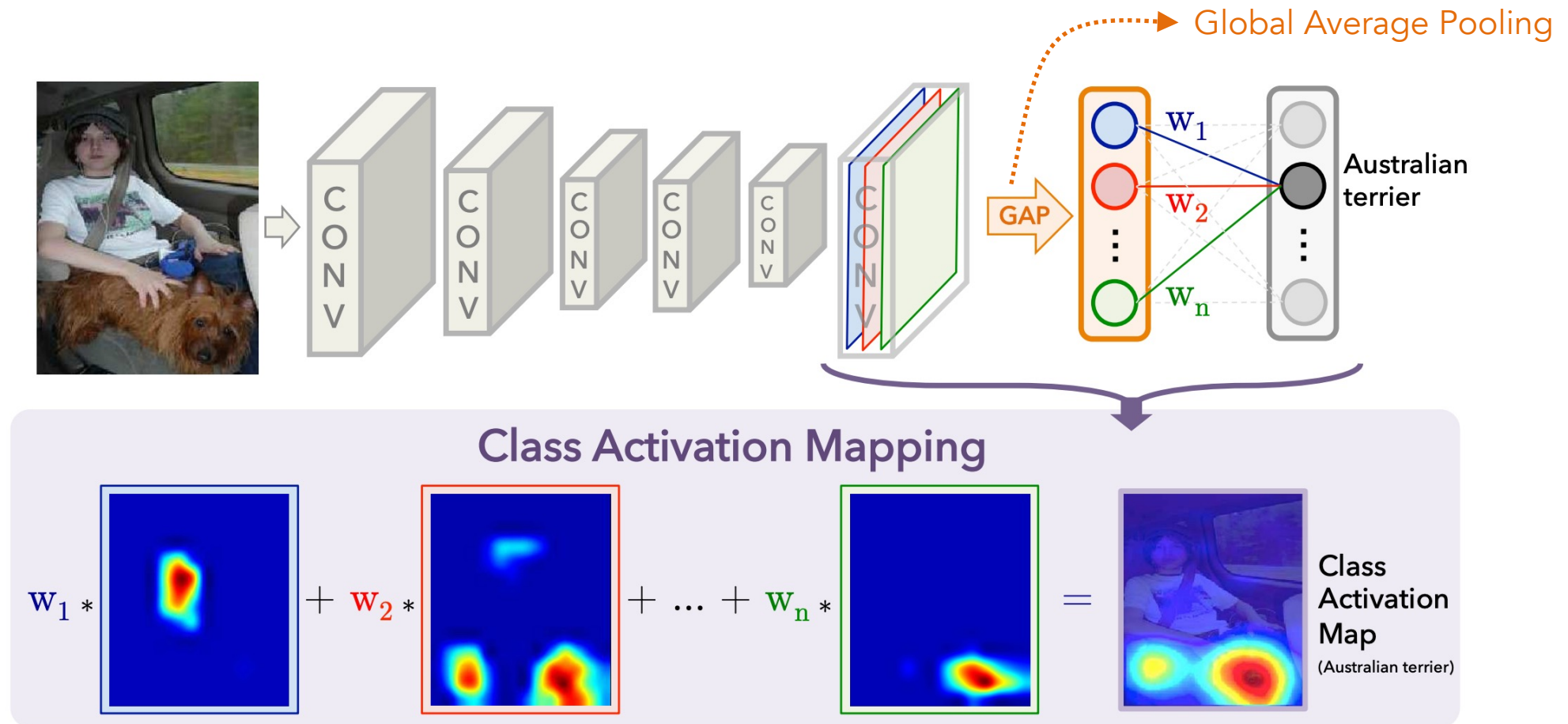


Exploding values

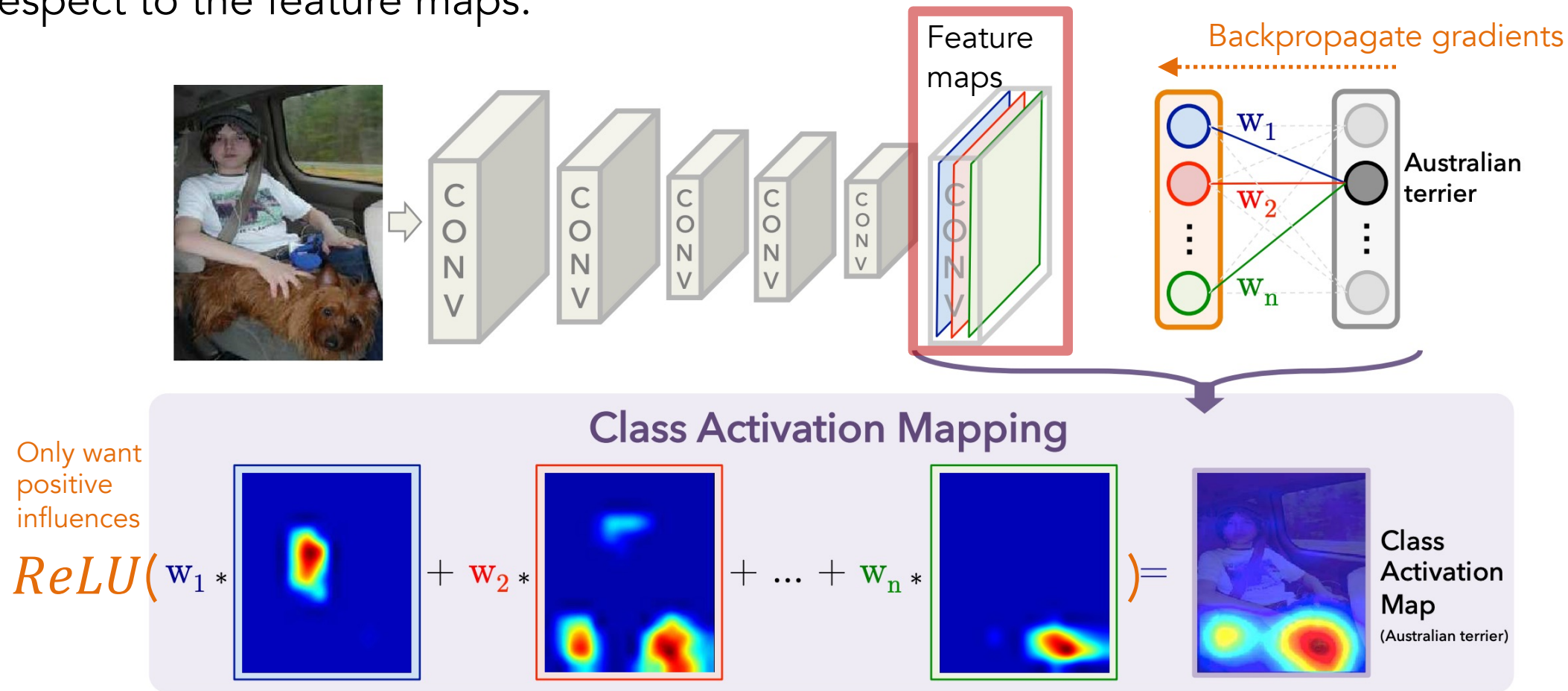
A better idea is to **scale the variance** of the random initialization (Gaussian or uniform distributions) for each layer, which leads to the Xavier (for Tanh activation) and Kaiming (for ReLU activation) initialization. The example below uses the Kaiming method.



We can use the Class Activation Mapping (CAM) method to visualize the areas in an image that contribute most significantly to the model decisions.

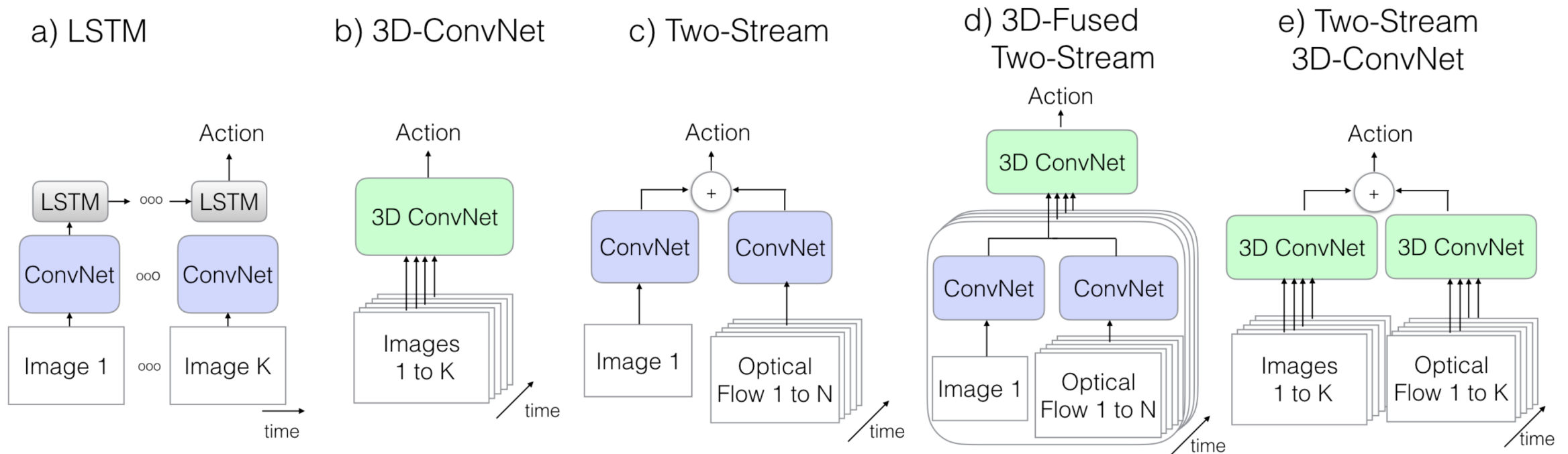


CAM only works for CNNs with a global average pooling layer. For generalization, Grad-CAM computes weights by averaging the gradients of the target class score with respect to the feature maps.

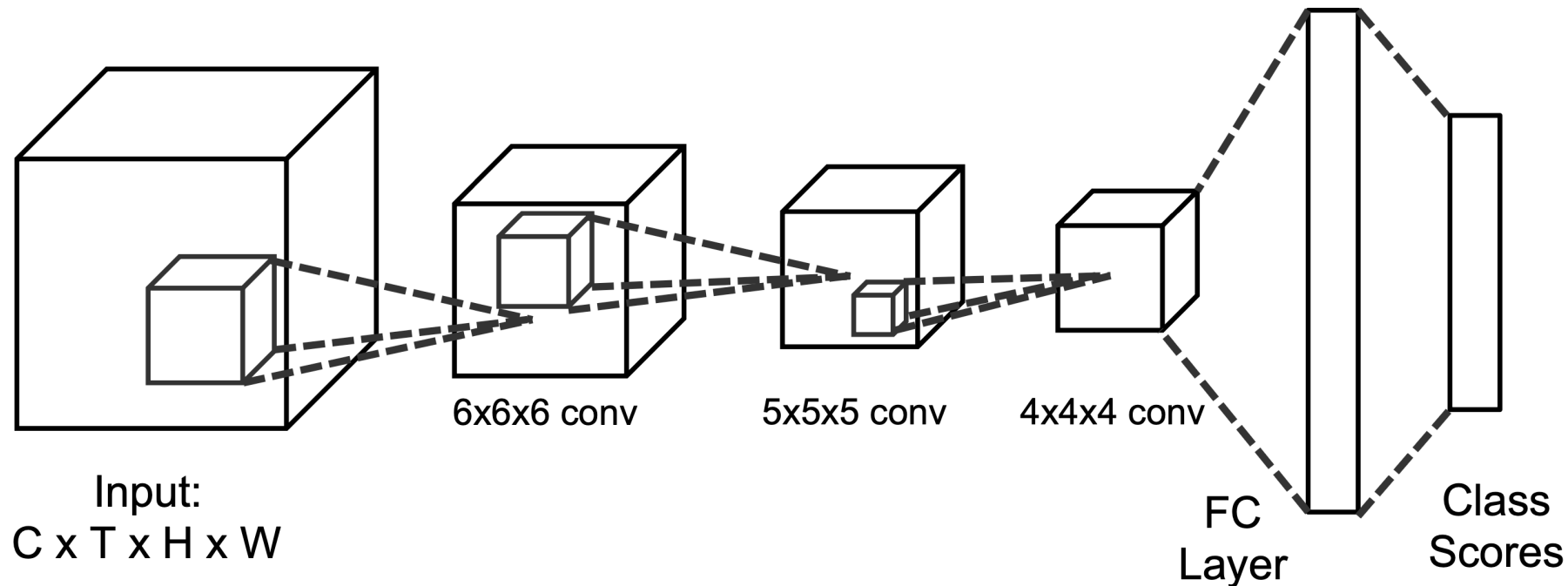


What about processing **videos**, which are series of images over time?

There are many ways for video classification (as shown below), where K means the total number of video frames, N means a subset of neighboring video frames.



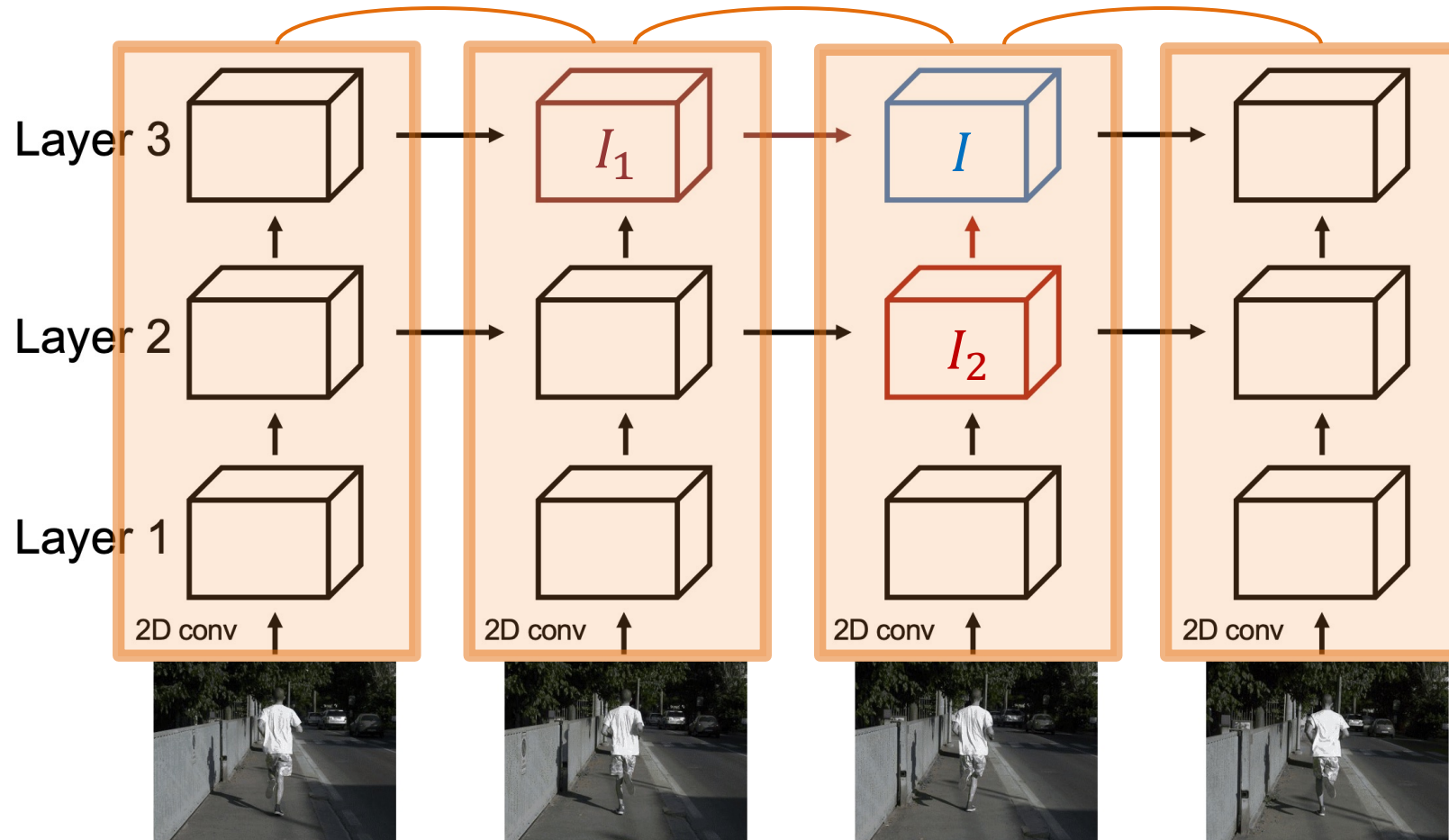
Instead of using 2D convolutional kernels, we can use **3D kernels** to learn information from videos. Videos can be represented as a 4D tensor (channel*time*height*width).



Karpathy, A., et al. (2014). [Large-scale video classification with convolutional neural networks](#). CVPR.

We can also combine CNN and RNN for video classification.

Share model weights (parameters) across time steps, but use different weights at each layer



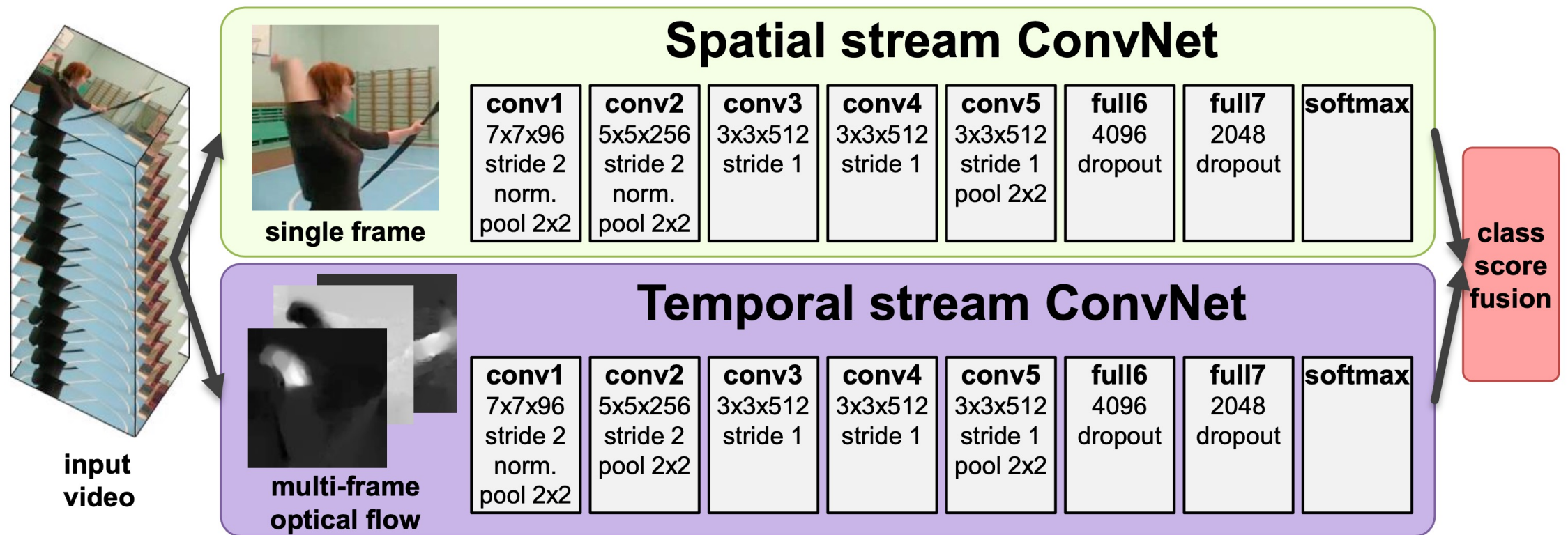
Entire network uses 2D feature maps (channel*height*width)

Each feature map I depends on two inputs:

- I_1 : Same layer, previous timestep
- I_2 : Previous layer, same time step

Ballas, N., et al. (2016). [Delving deeper into convolutional networks for learning video representations](#). ICLR.

We can *separately consider appearance and motion*. The two-stream network below uses CNN on individual video frames for the original image and optical flow.



Optical flow is a computer vision technique that **calculates and highlights local motions** (of objects, surfaces, edges, etc.) in consecutive video frames.

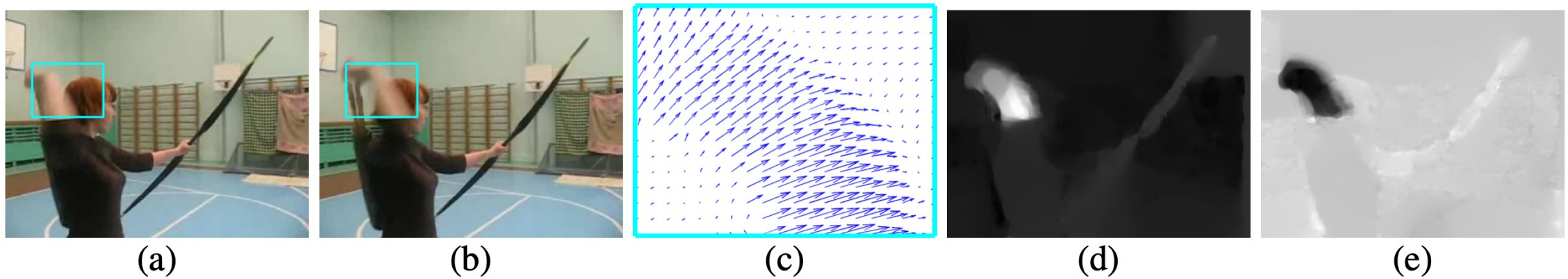


Figure 2: **Optical flow.** (a),(b): a pair of consecutive video frames with the area around a moving hand outlined with a cyan rectangle. (c): a close-up of dense optical flow in the outlined area; (d): horizontal component d^x of the displacement vector field (higher intensity corresponds to positive values, lower intensity to negative values). (e): vertical component d^y . Note how (d) and (e) highlight the moving hand and bow.

Take-Away Messages

- We can use deep learning to train models end-to-end from raw data to the desired output.
- Convolutional Neural Networks can learn image filters/kernels from a lot of data.
- Kernel size, stride, and padding will determine the final output size in convolution operations.
- The pooling layer has no trainable parameters and reduces the size of feature maps independently.
- ReLU is easy to compute, introduces non-linearity, and can mitigate the vanishing gradient problem.
- The normalization layers (e.g., Batch Norm) can make the network easier to train.
- Residual blocks allow very deep networks to learn identity functions that emulate shallow networks.
- For video classification, we can combine CNN with RNN, or we can use 3D convolutional layers.



Questions?