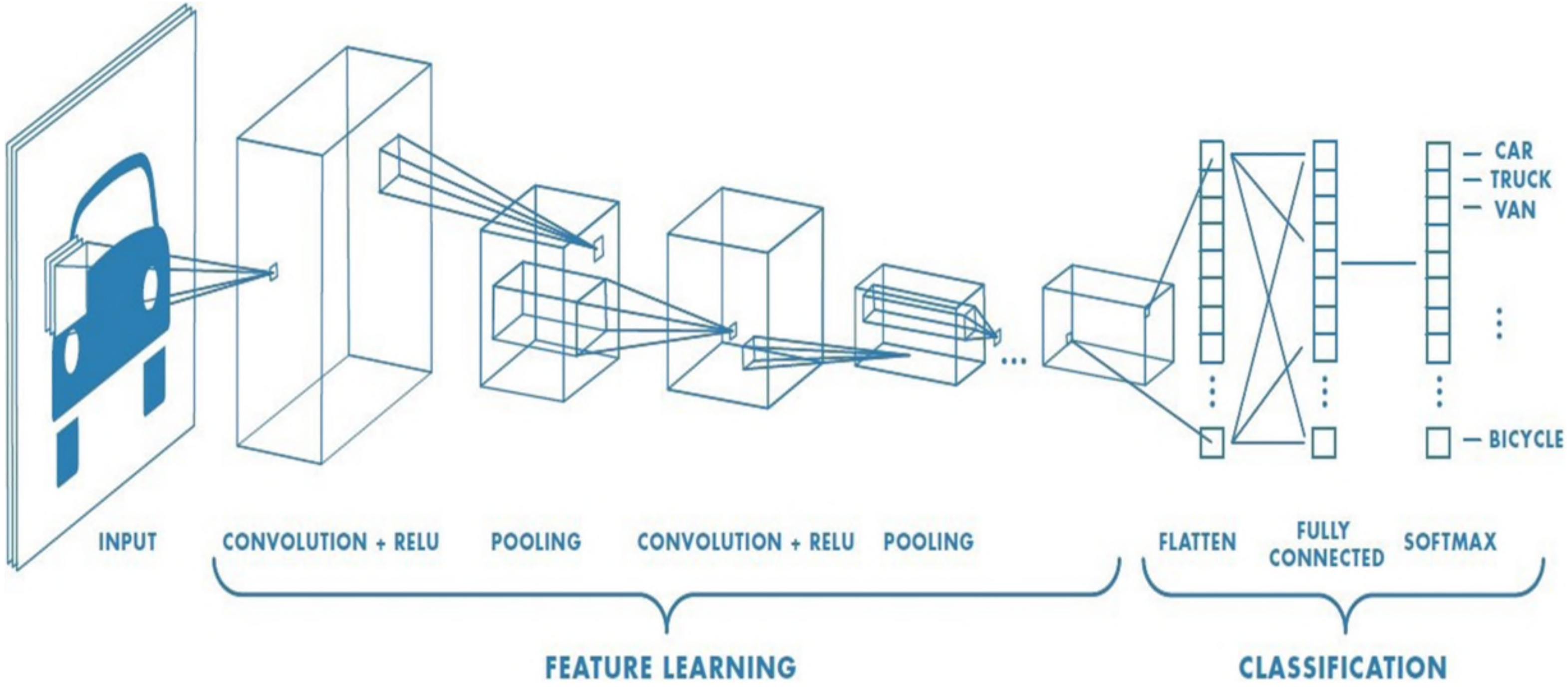
6. Convolutional Neural Networks and Transfer Learning for Image Classification



6.1 Introduction to Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are very similar to other neural networks we have studied in the previous parts. They all are made up of neurons with learnable weights and biases. Each neuron receives inputs, performs a linear operation of the inputs and then a non-linearity operation as the output.

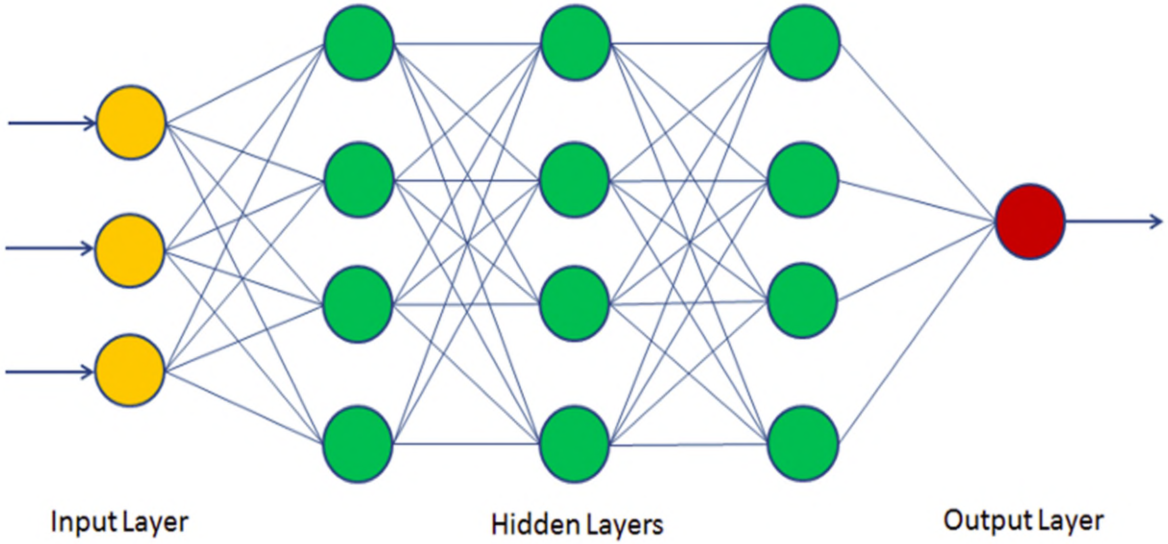
What are the changes in CNNs?

CNN make the explicit assumption that the inputs are images (or text, but we focus on images in this part), which allows us to encode certain properties into the architecture. These allow more efficient implementation and vast reduction of the number of parameters in the network.

Architecture Overview

Conventional neural networks

As introduced in previous parts, a neural network receives an input (a single vector) and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer. The last layer is the output layer whose output is the overall response of the neural network to the input.



(i) Conventional neural networks do not scale well to full images

In the part of MLP neural network, we introduced one application example, that is the recognition of handwritten digits, where each digit is a grey scale image of size 28×28 pixels. Therefore, each digit image is converted into a 784-dimensional vector and therefore 784 neurons in the input layer of the MLP neural network are needed.

For colour images, for example, of size 32×32×3 (32 wide, 32 high, 3 color channels), then the vector dimension will be 32×32×3=3072. Thus, a single fully-connected neuron in the first hidden layer of the MLP neural network will have 3072 weights.

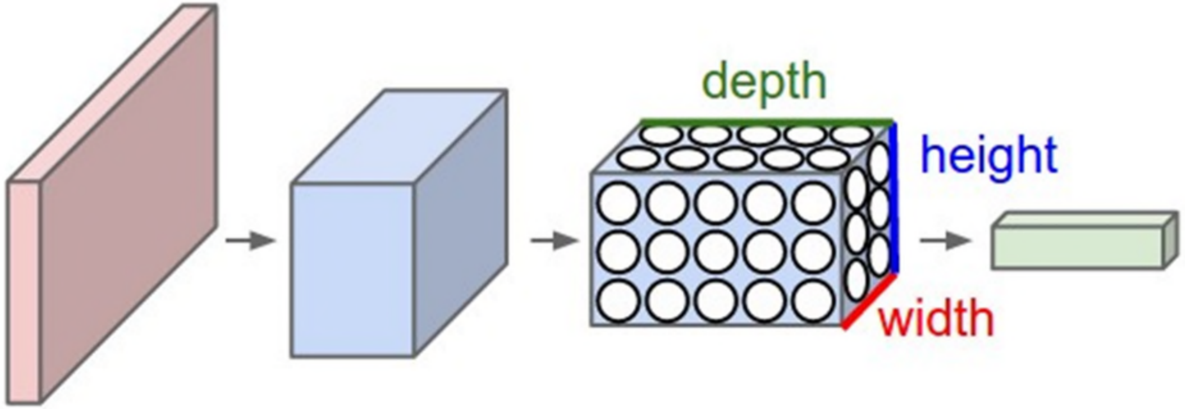
This number still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200×200×3, would make each neuron in the first hidden layer have 200×200×3 = 120,000 weights.

(2) Conventional neural networks can easily lead to overfitting

We certainly will have more than one neurons in the hidden layer, and thus the number of weights (parameters) would add up quickly, and the huge number of parameters would quickly lead to ***overfitting***.

Convolutional neural networks (CNN)

Convolutional neural networks take the advantage of the fact that the input consists of images and constrain the architecture in a more sensible way. The input and the neurons of CNN are arranged in 3 dimensions: *width, height, depth*.



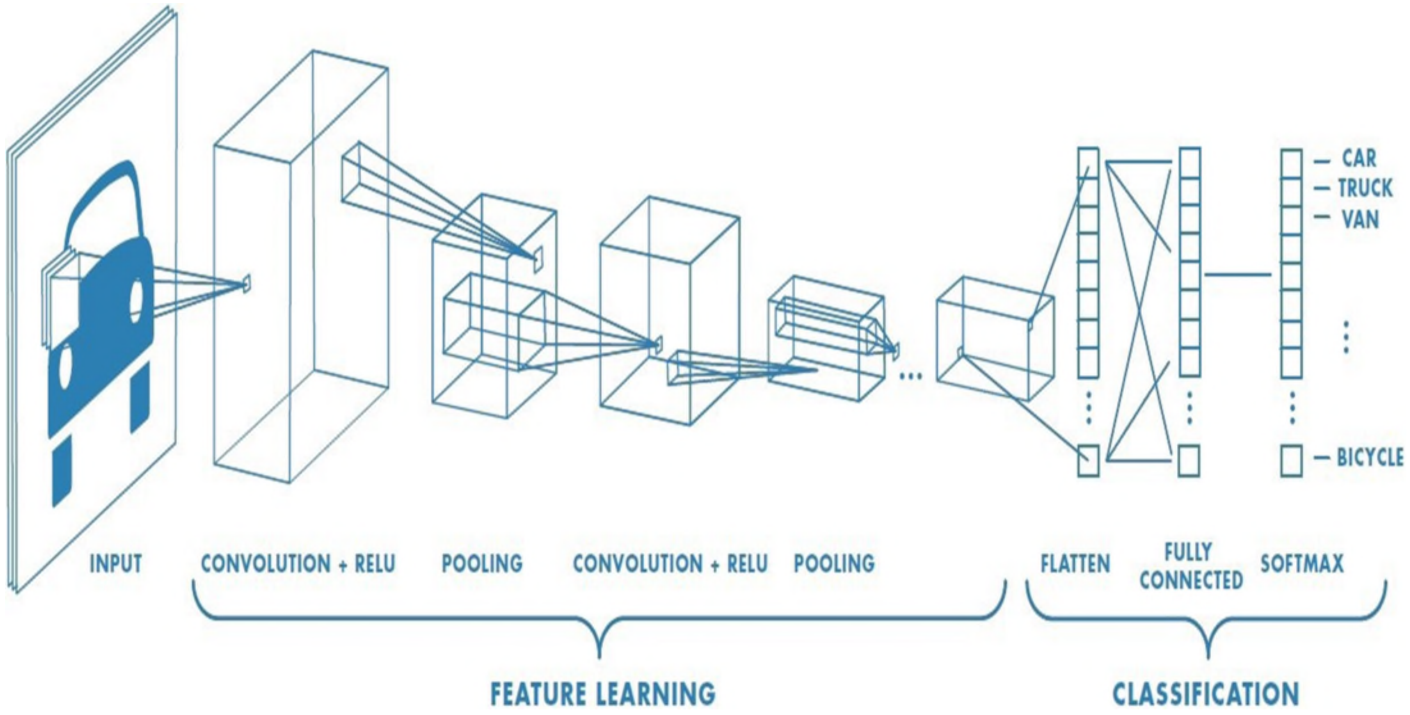
Note that the word *depth* here refers to the third dimension of the image volume, not the depth of a full CNN. For example, if the input image is a colour image with RGB (red, green and blue) three channels, then the volume has dimensions 32×32×3 (width, height, depth respectively). If the image is a grey-scale image, the depth is 1.

As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.

6.2 Layers used to build CNN

As can be seen, a simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of input to another. Three main types of layers are used to build CNN architecture, including

(i) Convolutional Layer
(ii) Pooling Layer
(iii) Fully-Connected Layer (exactly as seen in conventional neural networks).



The architecture of a typical convolutional neural network

(i) Convolutional Layer

The Convolution layer is the core building block of a CNN. This layer consists of a set of learnable filters. Every filter is small spatially (along width and height) but extends through the full depth of the input volume. For example, a typical filter in the first convolutional layer of a CNN might have size of 5×5×3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional output (activation map) that gives the responses of that filter at every spatial position.

Each convolutional layer has a set of filters, and each of them will produce a separate 2-dimensional activation map. All these activation maps along the depth dimension are stacked to produce the output volume.

The convolution is much like the following scenario.

In the dark night, we want to find what are on a sheet of bubble wrap using a flashlight. Assuming that your flashlight shines an area of 5-bubble × 5-bubble. To look at the entire sheet, you would slide your flashlight across each 5×5 square until you have seen all the bubbles.

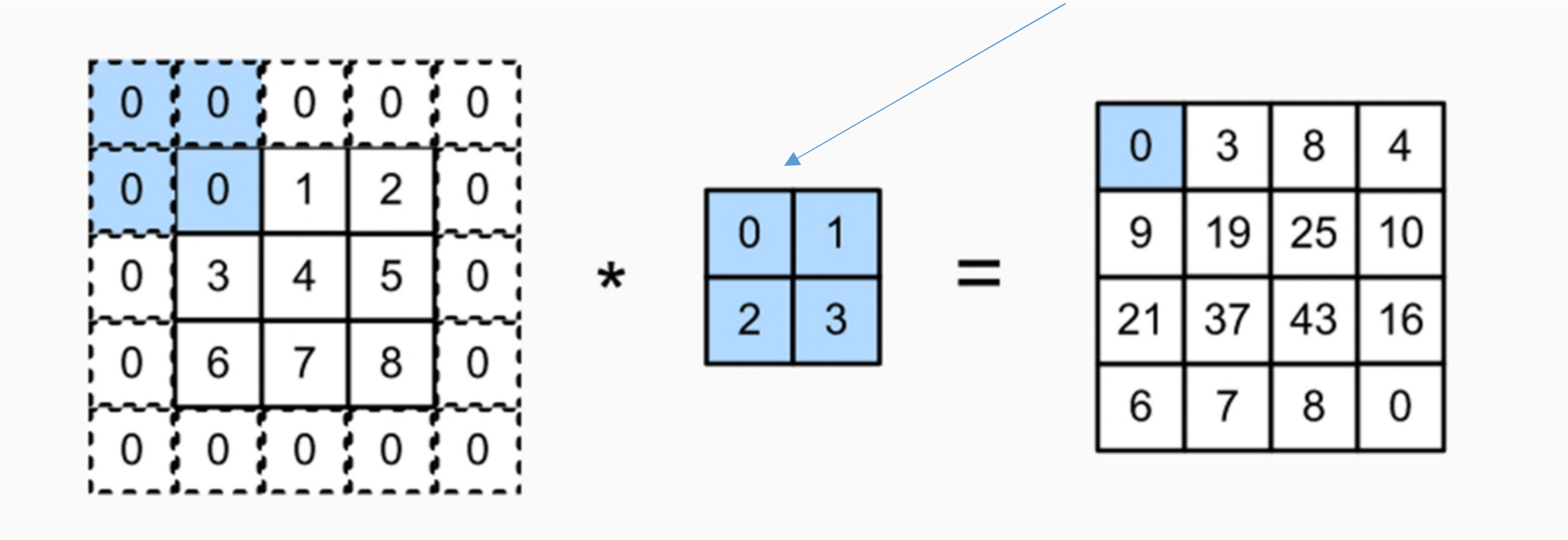
The light from the flashlight here is your *filter* and the region you are sliding over is the *receptive field*. The light sliding across the receptive fields is your flashlight *convolving*. Your filter is an array of numbers (also called weights or parameters).

The depth of the filter has to be the same as the depth of the input, so if we were looking at a color image, the depth would be 3. That makes the dimensions of this filter 5×5×3.

In each position, the filter multiplies the values in the filter with the original values in the pixel. This is element wise multiplication. The multiplications are summed up, creating a single number. The array you end up with is called a *feature map* or an *activation map*.

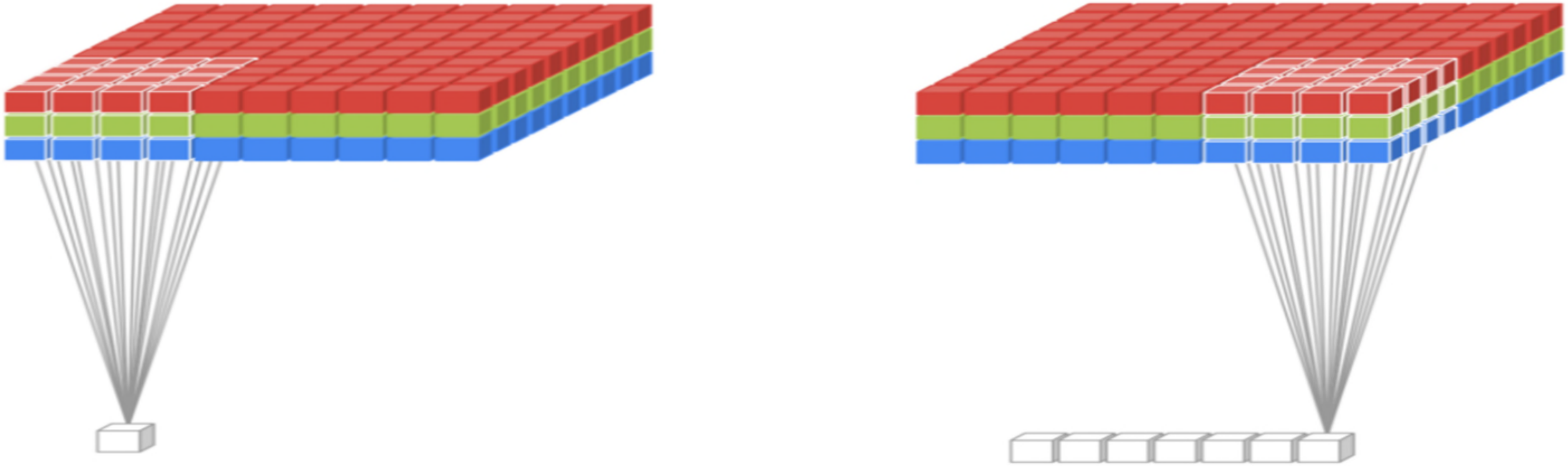
Convolution in 2D

Filter (or kernel, or window) size is 2×2



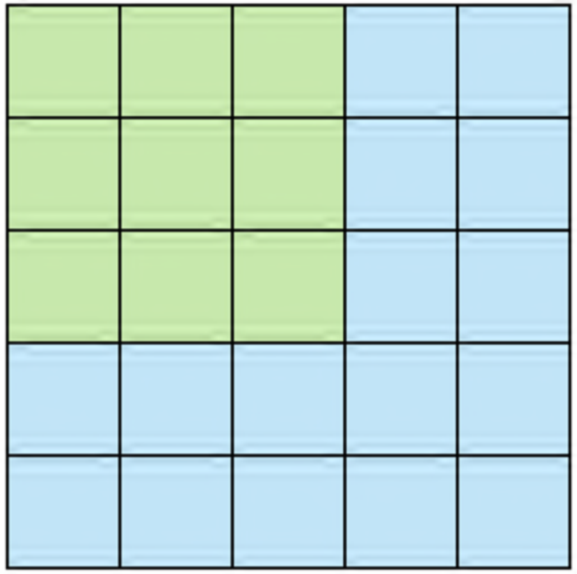
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$$

Convolution in 3D

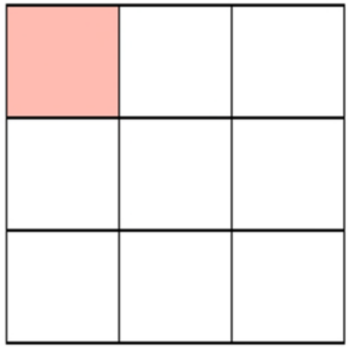


Stride

Stride is the number of pixels shifts over the input image (matrix, 2D or 3D). When the stride is 1, then we move the filters 1 pixel at a time. If the stride is set to 2, then we move the filters 2 pixels at a time and so on. The below figure shows convolution working with a stride of 1

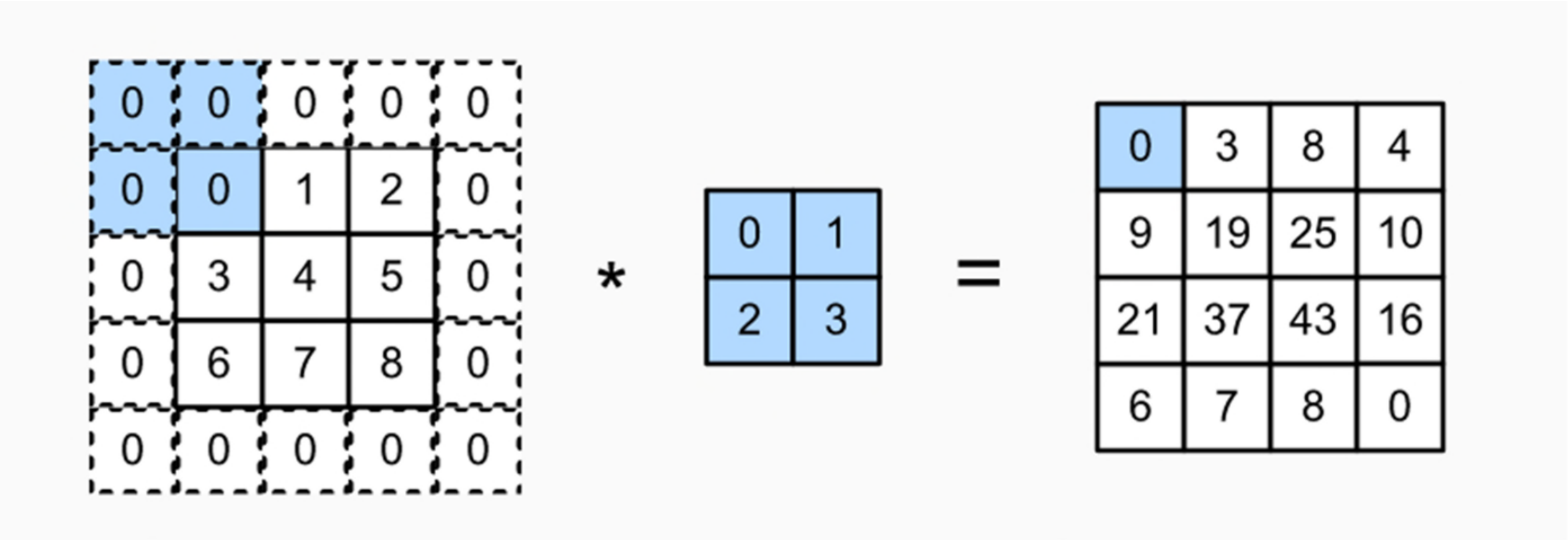


Stride 1



Feature Map

We can observe that the size of output is smaller than the input after convolution operation even if a stride of 1 is used as shown below.



0	0	0	0	0
0	0	1	2	0
0	3	4	5	0
0	6	7	8	0
0	0	0	0	0

*

0	1
2	3

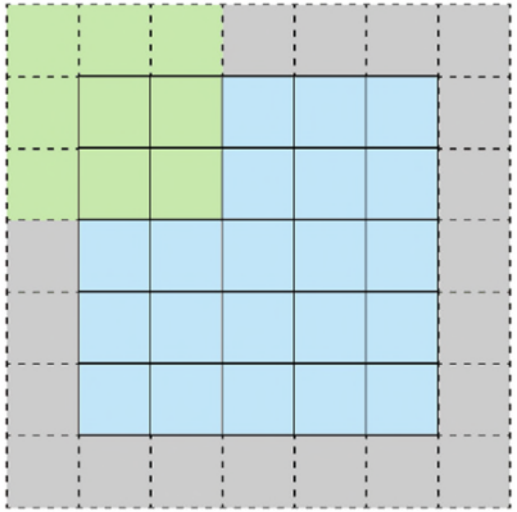
=

0	3	8	4
9	19	25	10
21	37	43	16
6	7	8	0

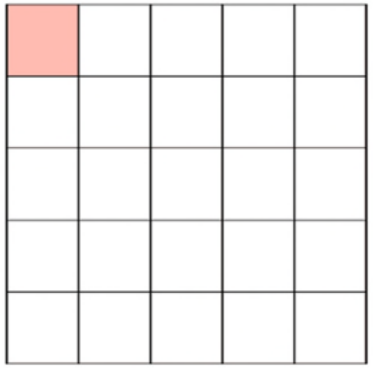
For an image with $I \times I$, the filter window has size of $f \times f$, and the stride is s, then the resulted image size of a convolution will be $[(I - f)/s + 1] \times [(I - f)/s + 1]$. Please note $(I - f)/s$ may not be an integer, for example $s = 2$ in the above example. This problem can be addressed by padding.

Padding

To maintain the dimension of output the same as that of the input , we use padding. Padding is a process of adding zeros to the input matrix symmetrically. As shown in the following example, the grey blocks denote the padding. Assume p rows or columns are added at each side, the resulted image size is $[(I - f + 2p) / s + 1] \times [(I - f + 2p) / s + 1]$.



Stride 1 with Padding



Feature Map

(ii) Pooling Layer

Pooling layers would reduce the number of parameters when the images are too large. Spatial pooling, also called subsampling or downsampling, aims to reduce the dimensionality of each map but retain the important information. Spatial pooling can be of different ways:

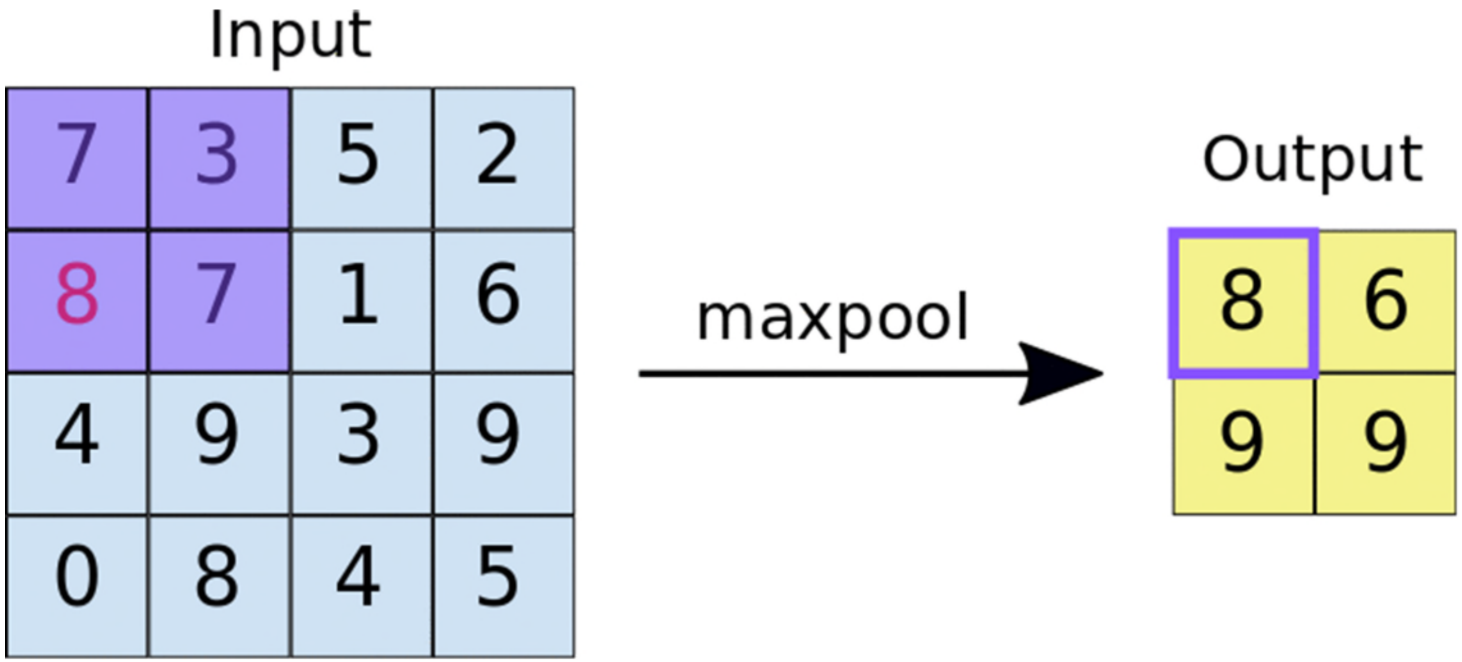
(i) Max pooling
(ii) Average pooling
(iii) Sum pooling

Max pooling

Max pooling takes the largest element in the masked regions of the feature map.

Average and sum pooling

Average pooling and sum pooling takes the average or sum of all elements in the masked regions of the feature map.



Input

7	3	5	2
8	7	1	6
4	9	3	9
0	8	4	5

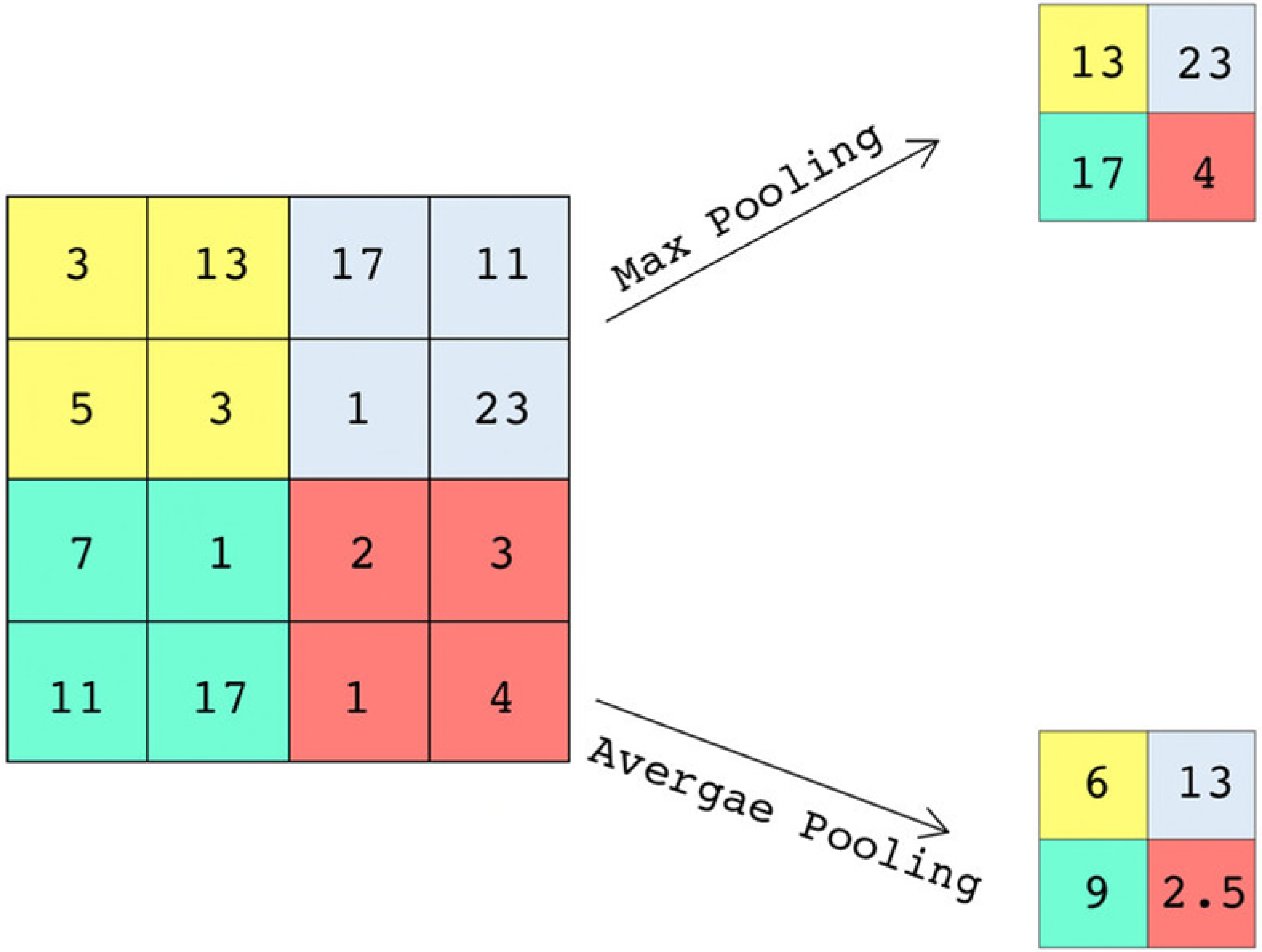
maxpool

Output

8	6
9	9

Obviously, the pooling operation reduces the dimensionality significantly: from 4×4 to 2×2.

The pooling window size here is 2×2. In addition, the pooling window can be overlapping or non-overlapping. The above is non-overlapping.



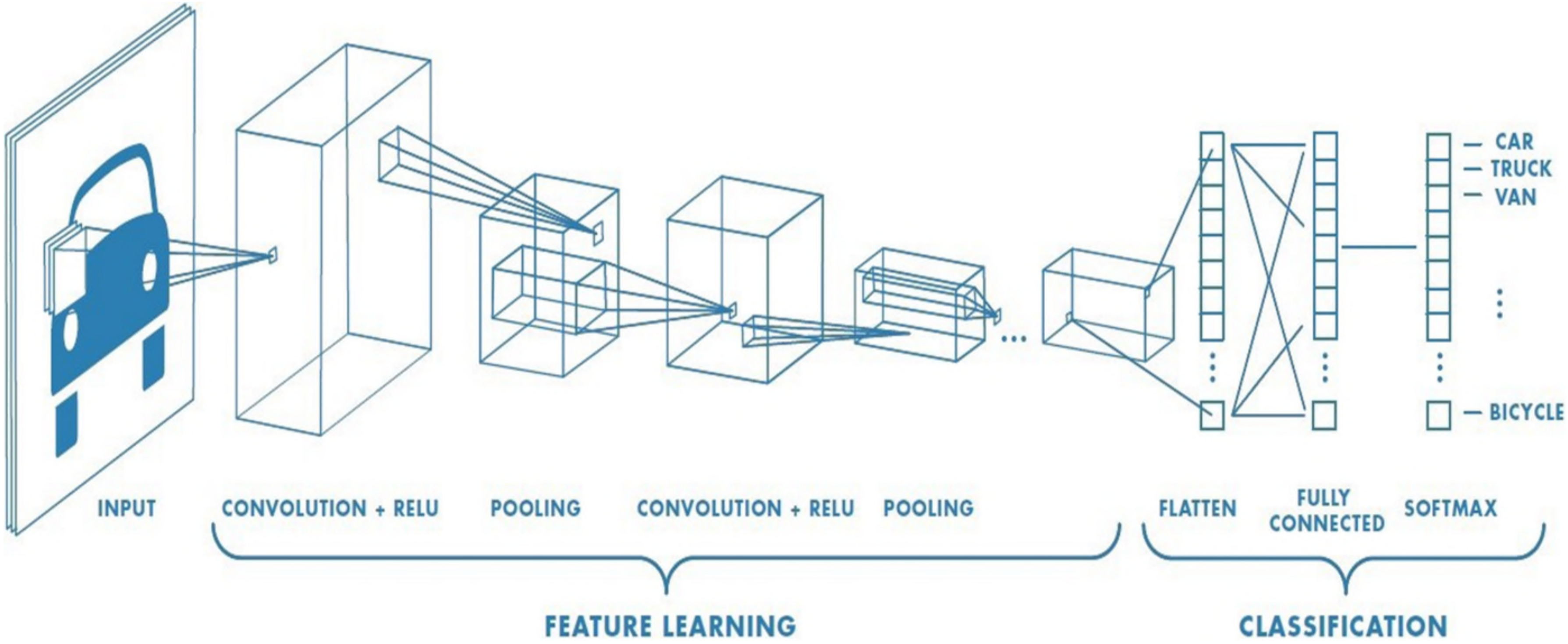
3	13	17	11
5	3	1	23
7	1	2	3
11	17	1	4

13	23
17	4

6	13
9	2.5

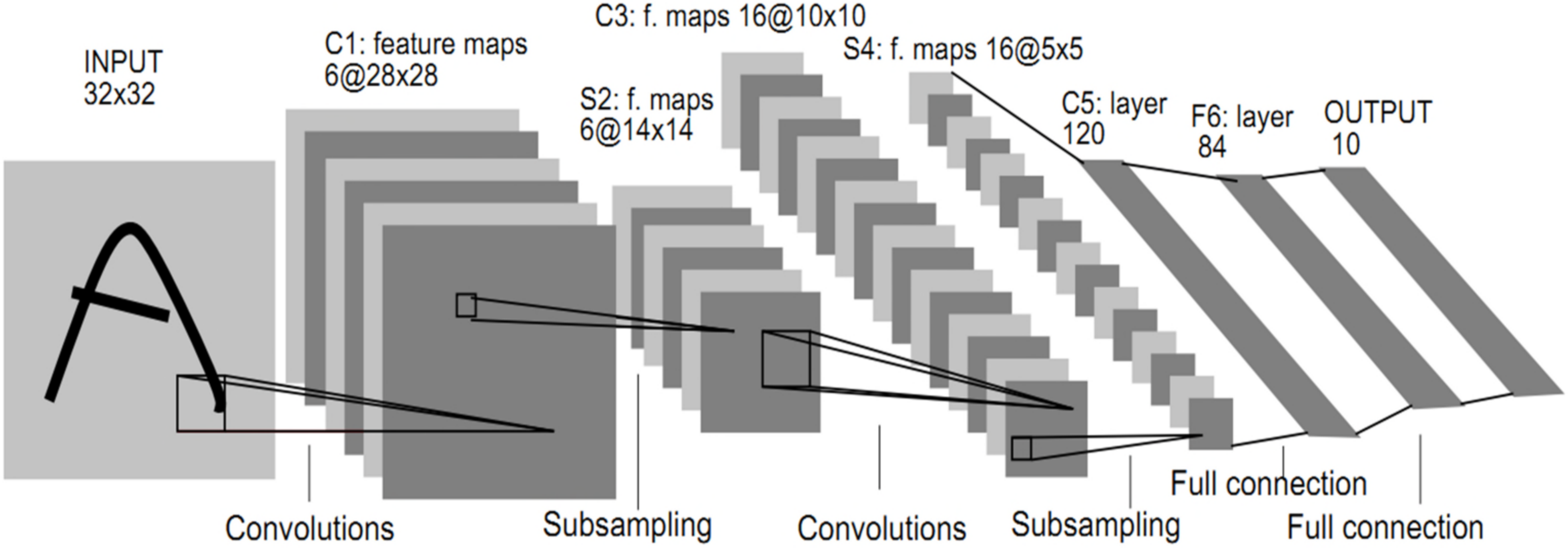
(iii) Fully connected layer

Finally, after several convolutional and max pooling layers, the feature maps are flattened to produce a vector representation of the image features in the fully connected layer. The fully connected layers perform pattern classification as shown below:



6.3 Some Popular CNN Architectures

(i) LeNet---LeCun et al



Let’s understand the architecture of the model. The model contained 7 layers excluding the input layer. Since it is a relatively small architecture, let’s go layer by layer:

Layer 1:

A convolutional layer with filter (kernel) size of 5×5, stride of 1×1 and 6 filters in total. So the input image of size 32×32×1 gives an output of 28×28×6.

Layer 2:

A pooling layer with 2×2 kernel size, stride of 2 × 2 and 6 kernels in total. This pooling layer acted a little differently from what we discussed in previous post. The input values in the receptive were summed up and then were multiplied to a trainable parameter (1 per filter), the result was finally added to a trainable bias (1 per filter). Finally, sigmoid activation was applied to the output. So, the input from previous layer of size 28×28×6 gets sub-sampled to 14×14×6.

Layer 3:

Similar to Layer 1, this layer is a convolutional layer with the same configuration except that it has 16 filters instead of 6. So the input from previous layer of size 14×14×6 gives an output of 10×10×16.

Layer 4:

Again, similar to Layer 2, this layer is a subsampling layer with 16 filters this time around. Remember, the outputs are passed through sigmoid activation function. The input of size 10×10×16 from previous layer gets sub-sampled to 5×5×16.

Layer 5:

This time around we have a convolutional layer with 5×5 kernel size and 120 filters. There is no need to even consider strides as the input size is 5×5×16 so we will get an output of 1×1×120.

Layer 6:

This is a dense layer with 84 neurons. So, the input of 120-dimensional vector is converted to a 84-dimensional vector. The activation function is used here.

Output Layer:

Finally, a dense layer with 10 neurons is used.

(ii) AlexNet --- Krizhevsky et al

AlexNet is a convolutional neural network that has a large impact on application of deep learning to machine vision.

The architecture consists of 5 Convolutional Layers and 3 Fully Connected Layers. These 8 layers combined with some new concepts at that time

(i) Max Pooling

(ii) ReLU activation function

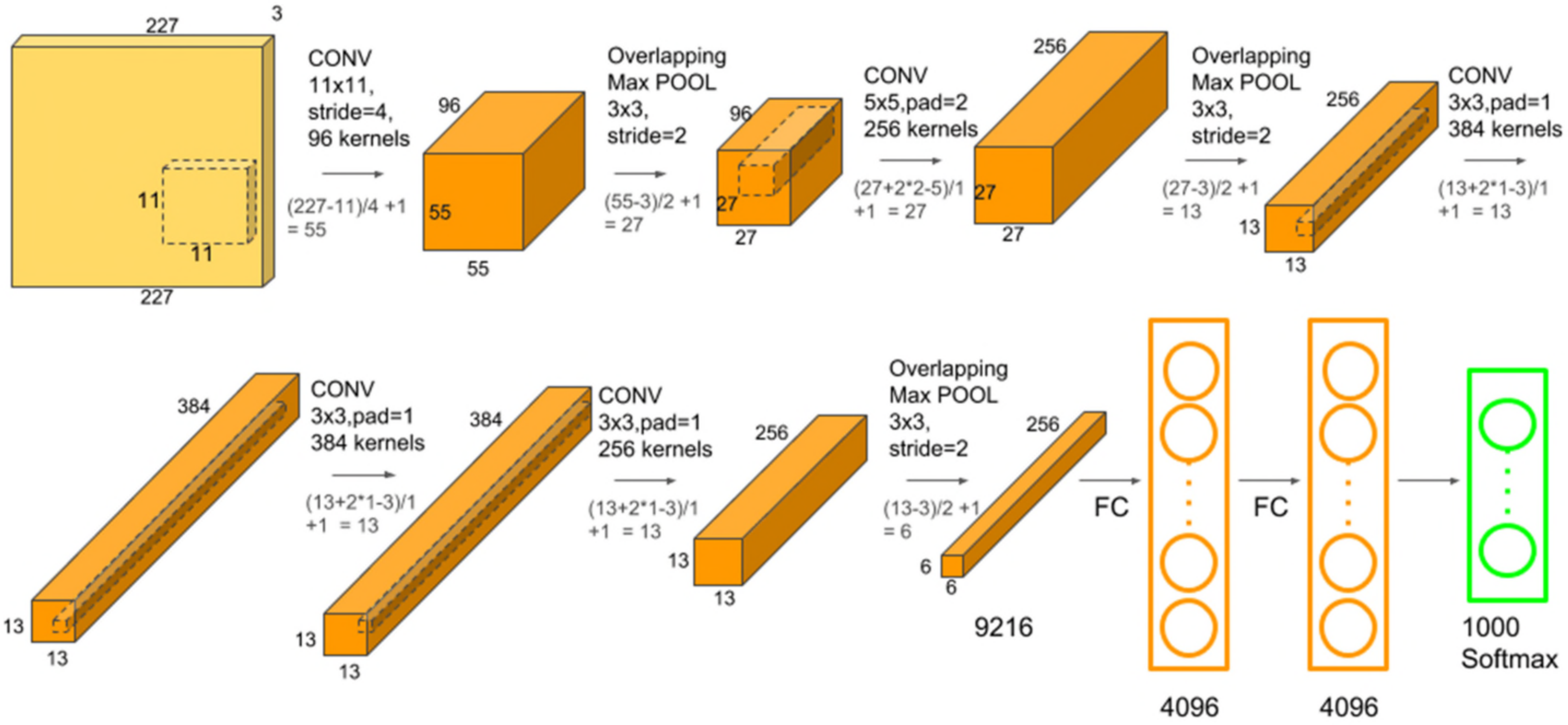
(iii) "Dropout" as the regularization

Network size:

(i) Has 60 million parameters and 650,000 neurons

(ii) Trained on 1.2 million high-resolution images

(iii) Samples from1000 classes


227
3
227
11
11
CONV
11x11,
stride=4,
96 kernels
(227-11)/4 +1
= 55
96
55
55
Overlapping
Max POOL
3x3,
stride=2
(55-3)/2 +1
= 27
96
27
27
CONV
5x5,pad=2
256 kernels
(27+2*2-5)/1
+1 = 27
256
27
27
Overlapping
Max POOL
3x3,
stride=2
(27-3)/2 +1
= 13
256
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
256 kernels
(13+2*1-3)/1
+1 = 13
256
13
13
Overlapping
Max POOL
3x3,
stride=2
(13-3)/2 +1
= 6
256
6
6
9216
FC
4096
FC
4096
1000
Softmax

(iii) VGGNet ---Simonyan et al

VGGNet is a simple form of deep convolutional neural network due to its simplicity of the uniform architecture.

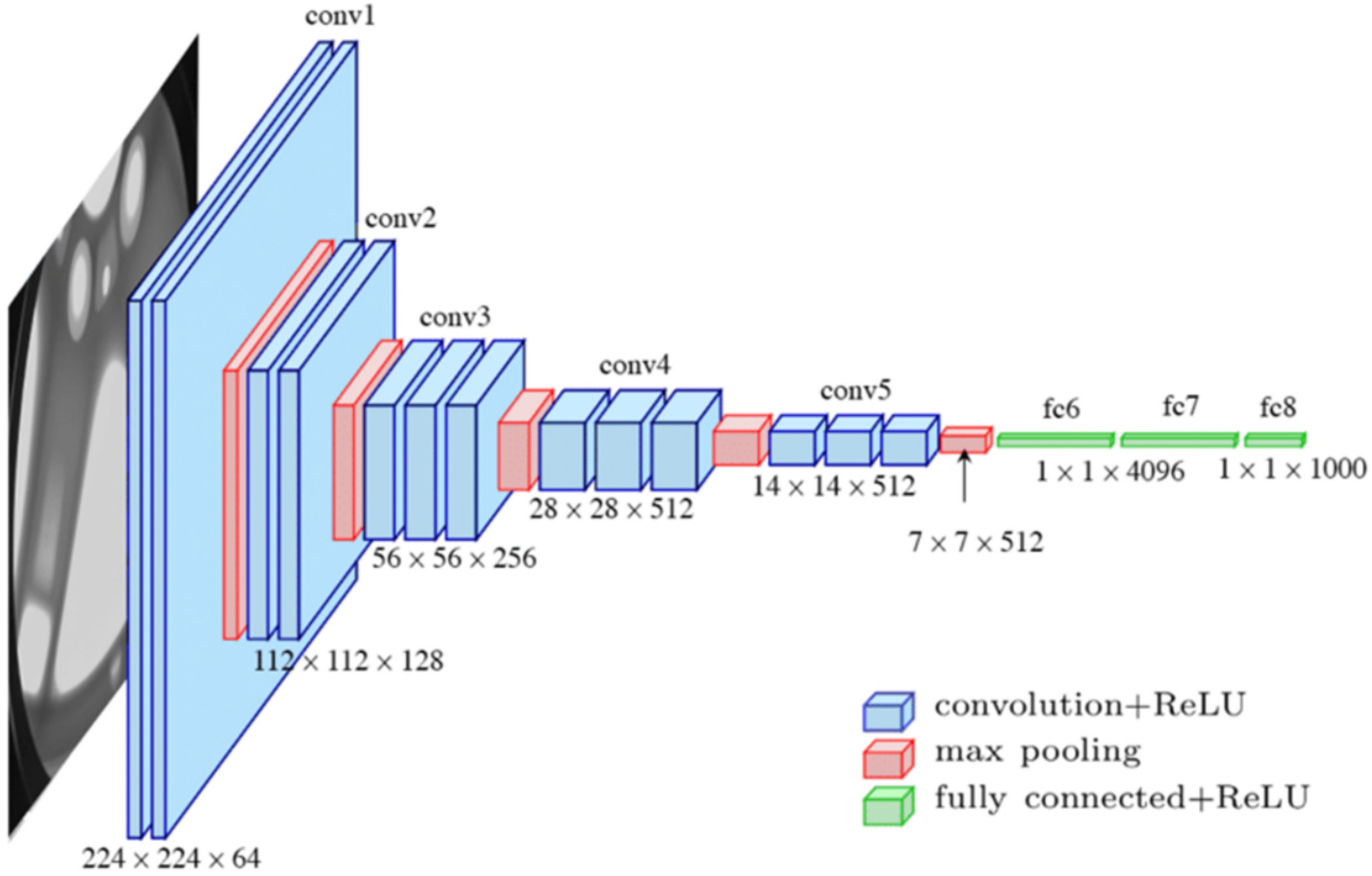
Features

(i) Each Convolutional layer has configuration — kernel size = 3×3, stride = 1×1. The only thing that differs is number of filters.

(ii) Each Max Pooling layer has configuration — windows size = 2×2 and stride = 2×2. Thus, we half the size of the image at every Pooling layer.

(iii) The input is RGB image of 224×224 pixels. So input size is 224×224×3

(iv) Total number of weights is 138 million. Most of these parameters are contributed by fully connected layers.

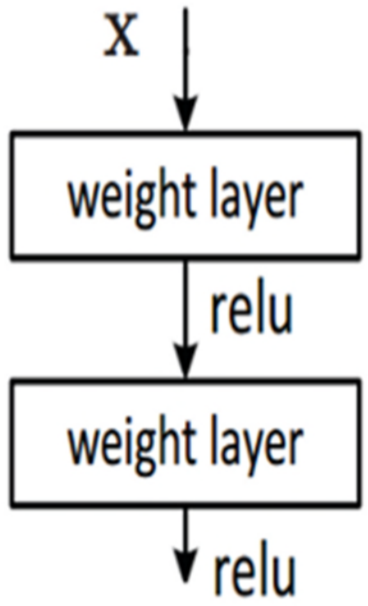

conv1
conv2
conv3
conv4
conv5
fc6
fc7
fc8
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
🟦 convolution+ReLU
🟥 max pooling
🟩 fully connected+ReLU

(iv) ResNet--- Kaiming He et al

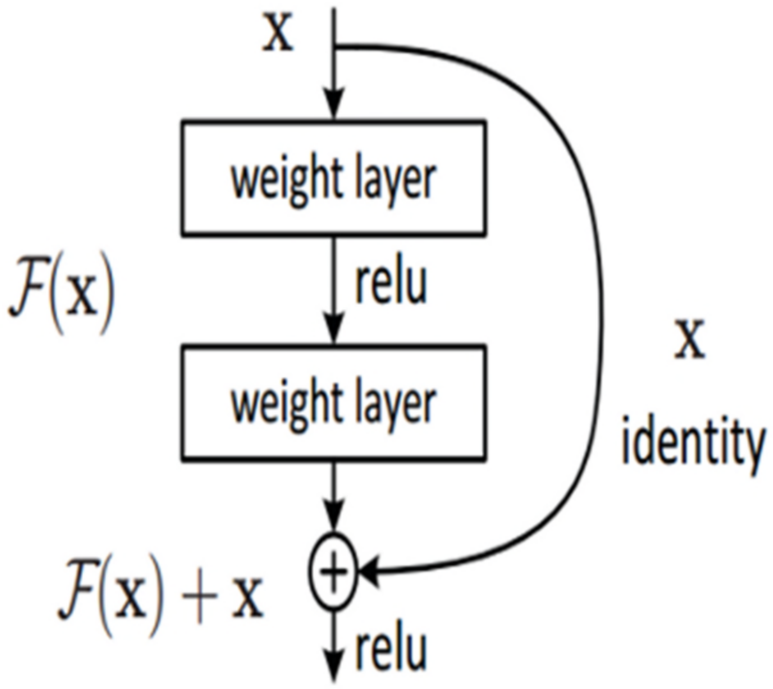
Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper. While AlexNet has only 5 convolutional layers, the VGG network has 19 layers.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers, as shown in the following figure. The ResNet architecture which was introduced by Microsoft, won the ILSVRC (ImageNet Large Scale Visual Recognition Challenge) in 2015.



Conventional method



ResNet method

6.4 Training of CNN

Training of CNN is based on gradient descent optimization, similar to MLP neural networks. But the mini-batch mode is adopted. Under this mode, the gradient is the average over the mini-batch, and is called mini-batch stochastic gradient descent (SGD).

Different CNNs adopt different weight updating rules. Take the AlexNet as an example. The batch size of 128 samples, momentum of 0.9, and weight decay of 0.0005 are introduced into weight updating rule:

$$\Delta\mathbf{w}(n) = 0.9 \times \Delta\mathbf{w}(n-1) - 0.0005 \times \eta \times \mathbf{w}(n-1) - \eta \times \left\langle \frac{\partial J}{\partial \mathbf{w}} |_{\mathbf{w}=\mathbf{w}(n-1)} \right\rangle_{D_n}$$

where $\langle \frac{\partial J}{\partial \mathbf{w}} |_{\mathbf{w}=\mathbf{w}(n-1)} \rangle_{D_n}$ denotes the average over the n-th batch D_n of the derivative of the objective with respect to $\mathbf{w}$, evaluated at $\mathbf{w}(n-1)$.

6.5 Transfer Learning in CNN

Humans have an inherent ability to transfer knowledge across tasks. What we acquire as knowledge while learning about one task, we utilize in the same way to solve related tasks. The more related the tasks, the easier it is for us to transfer, or cross-utilize our knowledge. Some simple examples are:

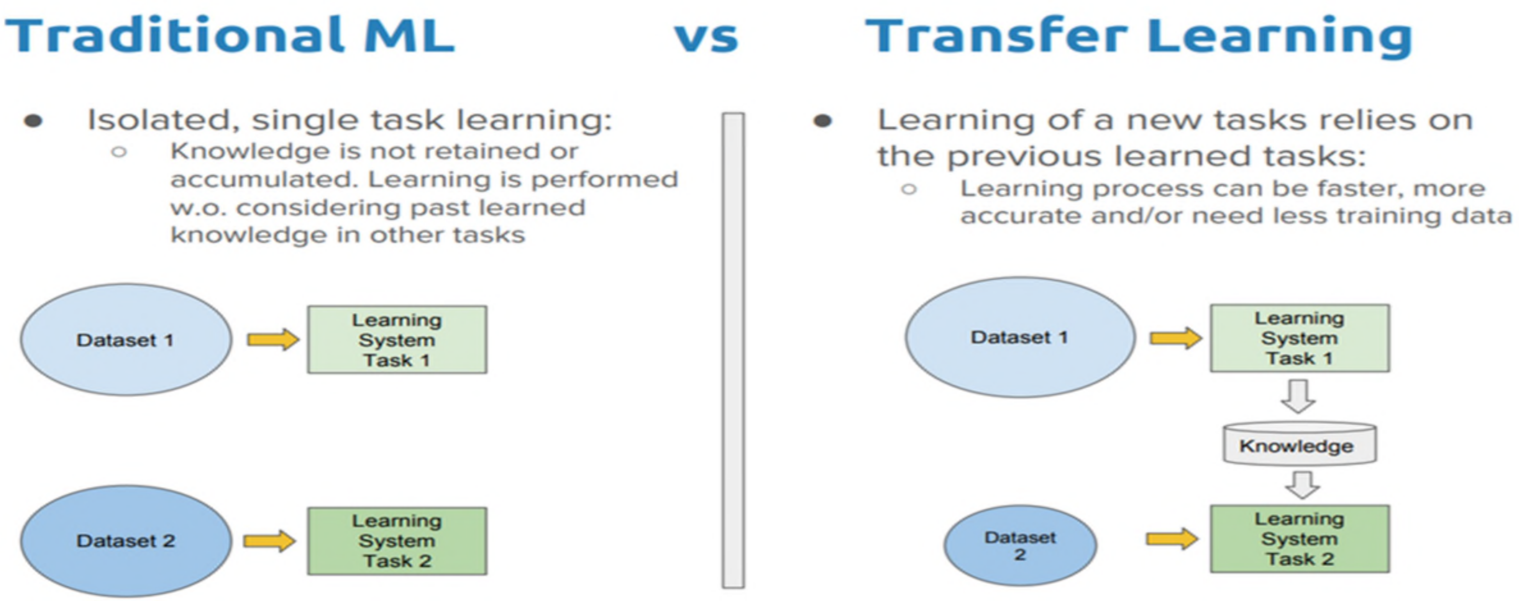
(i) Know how to ride a motorbike ➦ Learn how to ride a car

(ii) Know how to play classic piano ➦ Learn how to play jazz piano

(iii) Know math and statistics ➦ Learn machine learning

In each of the above scenarios, we do not learn everything from scratch when we attempt to learn new aspects or topics. We transfer and leverage our knowledge from what we have learnt in the past!

Conventional machine learning and deep learning algorithms have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones.



Traditional ML vs Transfer Learning

- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks

- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data

Three important questions of transfer learning

To use transfer learning, the following three important questions must be answered:

(i) What to transfer?

This is the first and the most important step in the whole process. We try to seek answers about which part of the knowledge can be transferred from the source to the target in order to improve the performance of the target task. We should try to identify which portion of knowledge is source-specific and what is common between the source and the target.

(ii) When to transfer?

There can be scenarios where transferring knowledge for the sake of it may make matters worse than improving anything (also known as negative transfer). We should aim at utilizing transfer learning to improve target task performance/results and not degrade them. We need to be careful about when to transfer and when not to.

(iii) How to transfer?

Once the *what* and *when* have been answered, we can proceed towards identifying ways of actually transferring the knowledge across domains/tasks. This involves changes to existing algorithms and different techniques.

Three major transfer learning scenarios

It is common to pre-train a CNN on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the CNN either as an initialization or a fixed feature extractor for the task of interest.

The three major transfer learning scenarios are as follows:

(i) CNN as fixed feature extractor

Take a CNN pre-trained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the CNN as a fixed feature extractor for the new dataset.

In an AlexNet, for example, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features *CNN codes*. Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.

(ii) Fine-tuning the CNN

The second strategy is not only to replace and retrain the classifier on top of the CNN on the new dataset, but also to fine-tune the weights of the pre-trained network by continuing the backpropagation.

It is possible to fine-tune all the layers of the CNN, or it is possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a CNN contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the CNN becomes progressively more specific to the details of the classes contained in the original dataset.

In case of ImageNet, for example, which contains many dog breeds, a significant portion of the representational power of the CNN may be devoted to features that are specific to differentiating between dog breeds.

(iii) Pre-trained models

Since modern CNNs take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final CNN checkpoints for the benefit of others who can use the networks for fine-tuning.

When and how to fine-tune?

How do you decide what type of transfer learning you should perform on a new dataset? This is a function of several factors, but the two most important ones are

(i) The size of the new dataset (small or big), and

(ii) Its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different, such as microscope images).

Keep in mind that CNN features are more generic in early layers and more original-dataset-specific in later layers, here are some common rules of thumb for navigating the 4 major scenarios:

(i) New dataset is small and similar to original dataset.

Since the data is small, it is not a good idea to fine-tune the CNN due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the CNN to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.

(ii) New dataset is large and similar to the original dataset

Since we have more data, we can have more confidence that we will not overfit the training data if we try to fine-tune the full network.

(iii) New dataset is small but very different from the original dataset.

Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier from the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

(iv) New dataset is large and very different from the original dataset.

Since the dataset is very large, we may expect that we can afford to train a CNN from scratch. However, in practice it is very often beneficial to initialize with weights from a pre-trained model. In this case, we would have enough data and confidence to fine-tune the entire network.