7. 1 Deep CNN

- Although LeNet achieved good results, they didn’t dominate the field
- In 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer CNNs
- Typical computer vision pipelines consisted of manually engineering feature extraction pipelines
- Rather than learn the features, the features were crafted

Classical pipelines

- Obtain an interesting dataset
- Preprocess the dataset with hand-crafted features based on some domain knowledge
- Feed the data through a standard set of feature extractors (SIFT, SURF, …)
- Dump the resulting representation into your favorite classifier (linear model or kernel method)

7. 1 Deep CNN

Learning Representations

- Features themselves ought to be learned (not handy features like SIFT, SURF, HOG)
- Lowest layers might come to detect edges, colors, and textures
- Higher layers in the network represent larger structures, like eyes, noses
- Even higher layers could represent whole object like people, airplanes

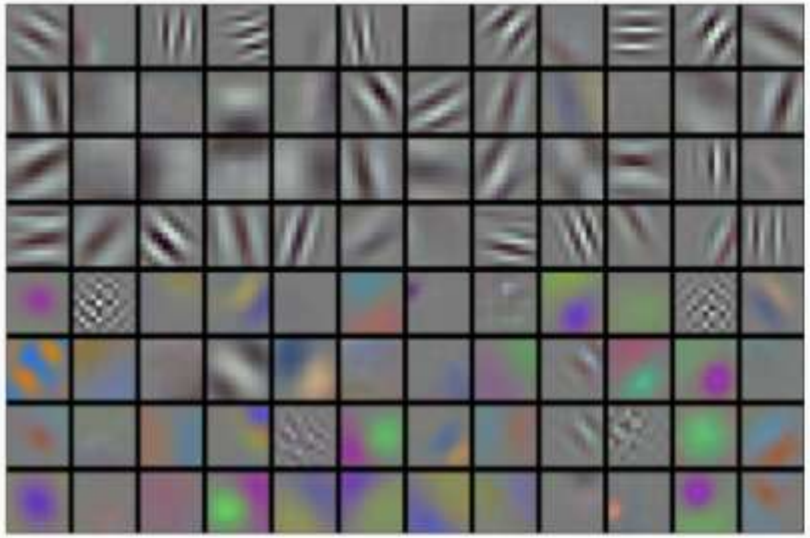


Fig. 7.1.1 Image filters learned by the first layer of AlexNet.

7. 1 Deep CNN

Missing Ingredient: Data

- Deep models with many layers require large amounts of data
- **However,** given limited storage, relative expense of sensors, tighter research budgets, …
- In the 1990s, most research relied on tiny datasets
- UCI collection of dataset, only hundreds or thousands of images

ImageNet dataset

- 1 million examples, 1000 each from 1000 distinct categories of object
- Led by Fei-Fei Li, using Amazon M-Turk crowdsourcing

Missing Ingredient: Hardware

- GPUs consist of 100 ~ 1000 small processing elements compared to CPUs (4 ~ 64)
- Game changer in making deep learning feasible, efficient for computing convolutional layer

7. 1 Deep CNN

AlexNet

- 2012 ImageNet winner, 8-layer CNN
- Break the previous paradigm in computer vision
- AlexNet and LeNet ar very similar
  - Much deeper than LeNet
  - Used ReLU instead of sigmoid

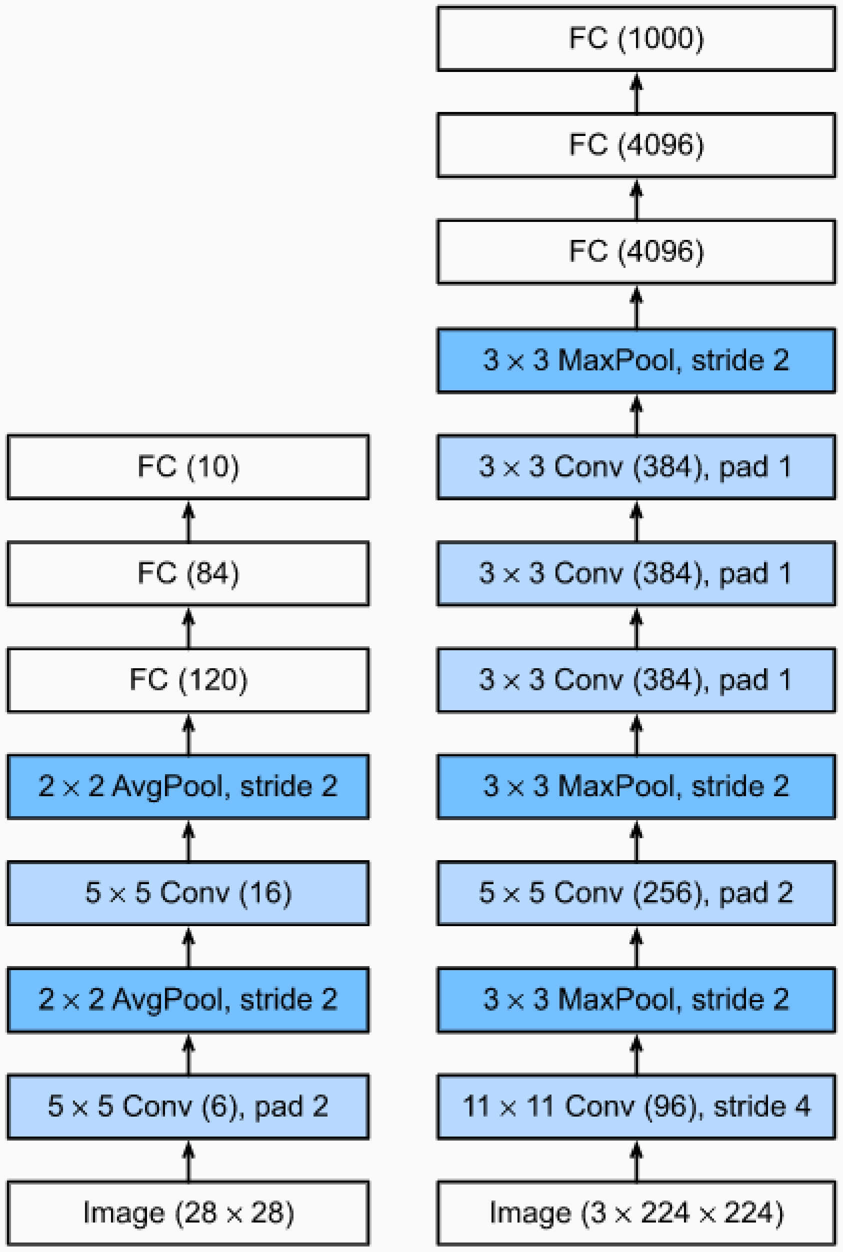


Fig. 7.1.2 From LeNet (left) to AlexNet (right).

7. 2 Networks Using Blocks (VGG)

Basic building block of class CNNs

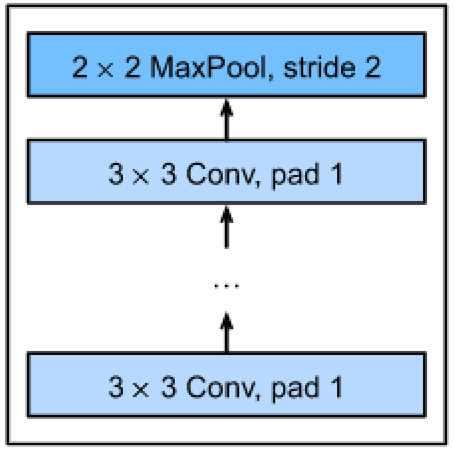
- A **convolutional layer** with padding
- A **nonlinearity** such as a ReLU
- A **pooling layer** such as a max pooling layer

VGG Blocks

- 3 x 3 kernels with padding of 1
- 2 x 2 max pooling with stride of 2

```
def vgg_block(num_convs, num_channels):
    blk = tf.keras.models.Sequential()
    for _ in range(num_convs):
        blk.add(tf.keras.layers.Conv2D(num_channels,kernel_size=3,
                                    padding='same',activation='relu'))
    blk.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
    return blk
```

VGG block



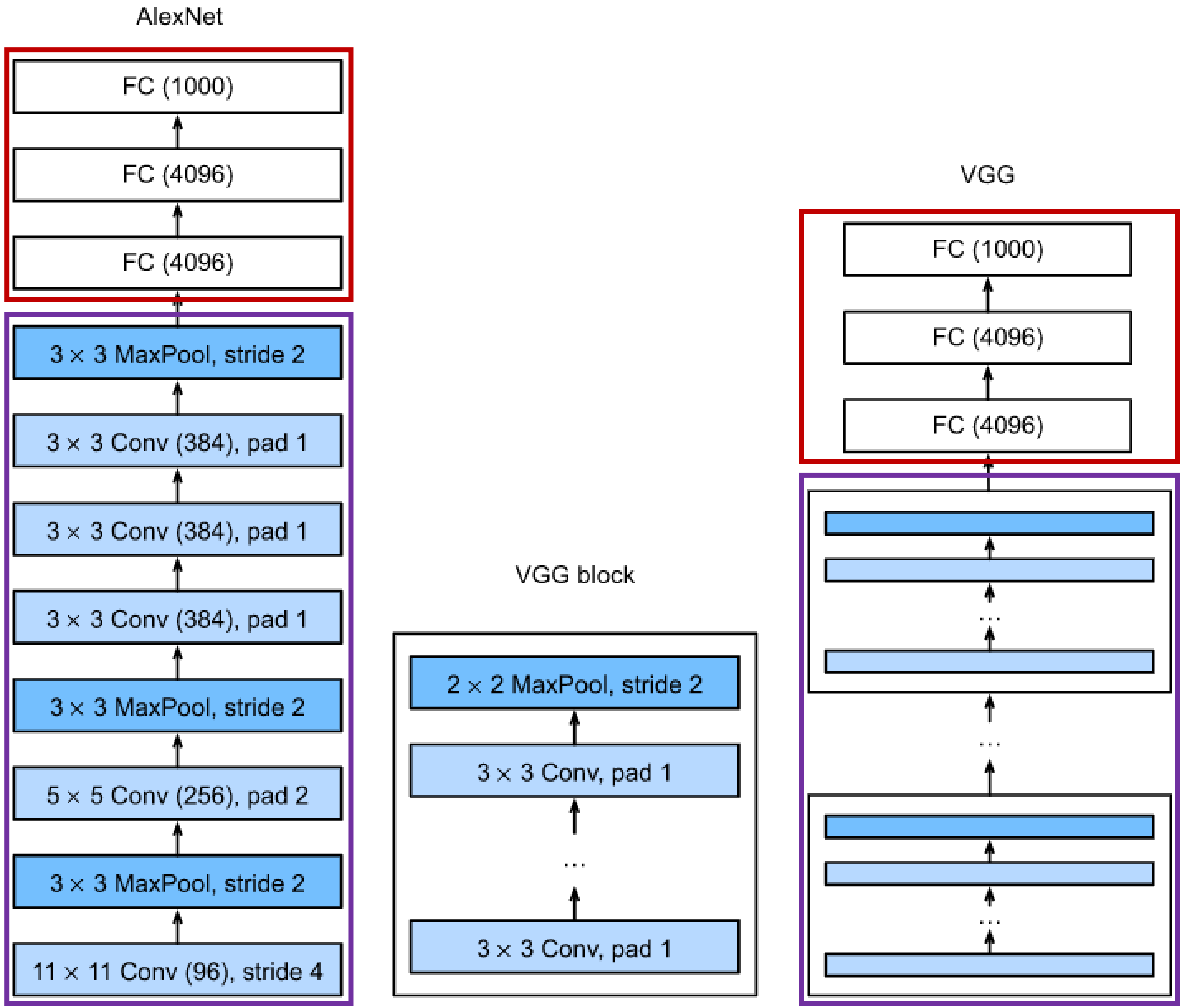
7. 2 Networks Using Blocks (VGG)

VGG Network

- Conv Arch (8 CNN) + 3 FC -> VGG-11
  - (1, 64)
  - (1, 128)
  - (2, 256)
  - (2, 256)
  - (2, 512)

```python
def vgg(conv_arch):
    net = tf.keras.models.Sequential()
    # The convulational part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully-connected part
    net.add(tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10)]))
    return net

net = vgg(conv_arch)
```



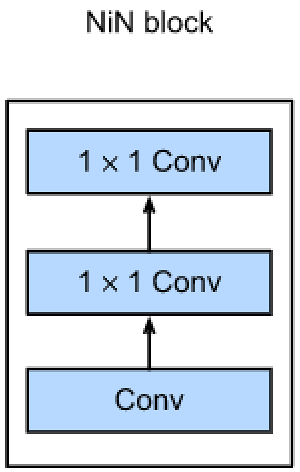
7. 3 Network in Network (NiN)

NiN Blocks

- Use an MLP on the channels for each pixel separately
- I/O of CNN -> (example, channel, height, weight)
- I/O of FC -> (example, feature)
- The idea behind NiN is to apply a fully-connected layer at each pixel location

```
from d2l import tensorflow as d2l
import tensorflow as tf

def nin_block(num_channels, kernel_size, strides, padding):
    return tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(num_channels, kernel_size, strides=strides,
                               padding=padding, activation='relu'),
        tf.keras.layers.Conv2D(num_channels, kernel_size=1,
                               activation='relu'),
        tf.keras.layers.Conv2D(num_channels, kernel_size=1,
                               activation='relu')])
```

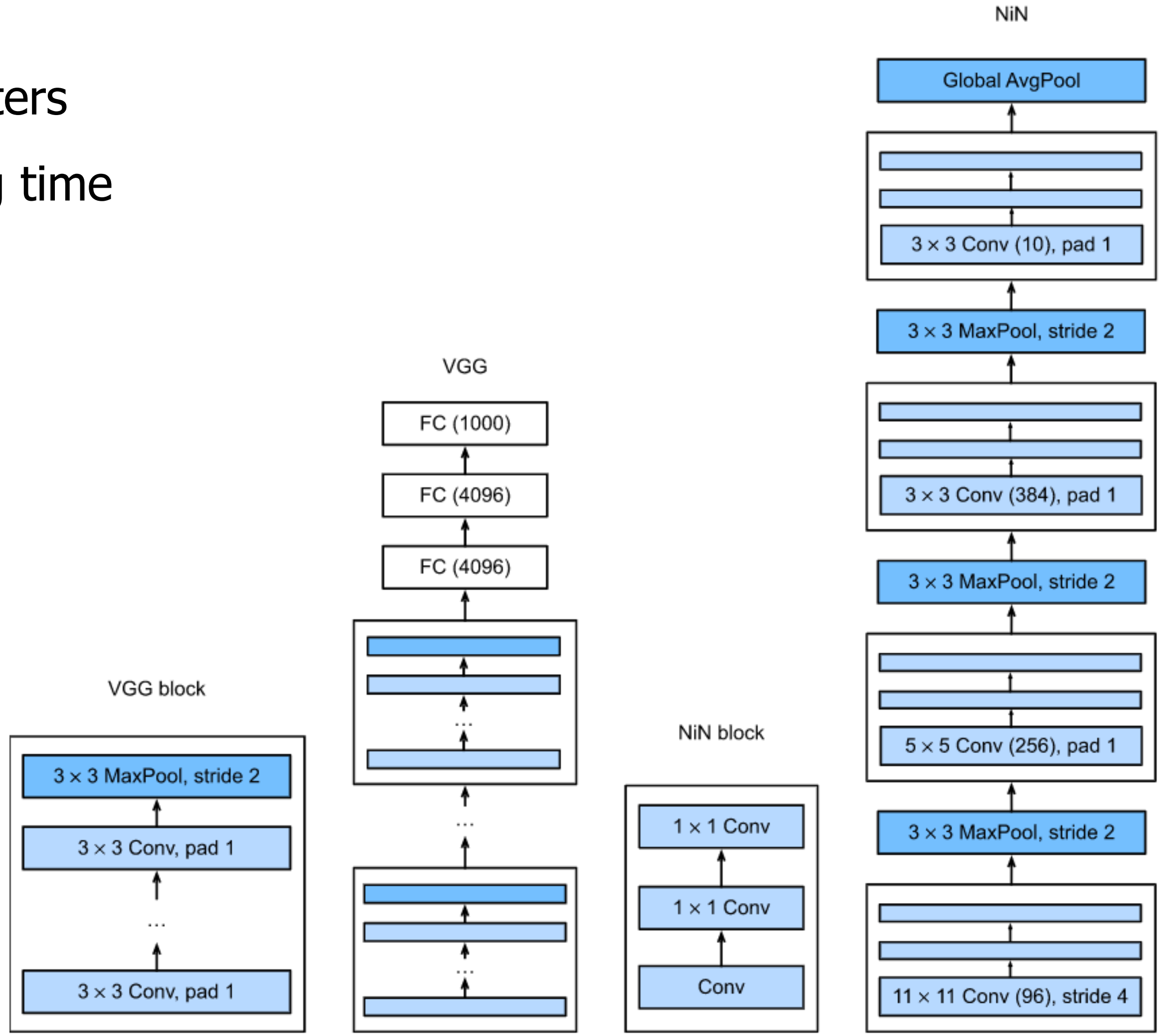


7. 3 Network in Network (NiN)

NiN Model

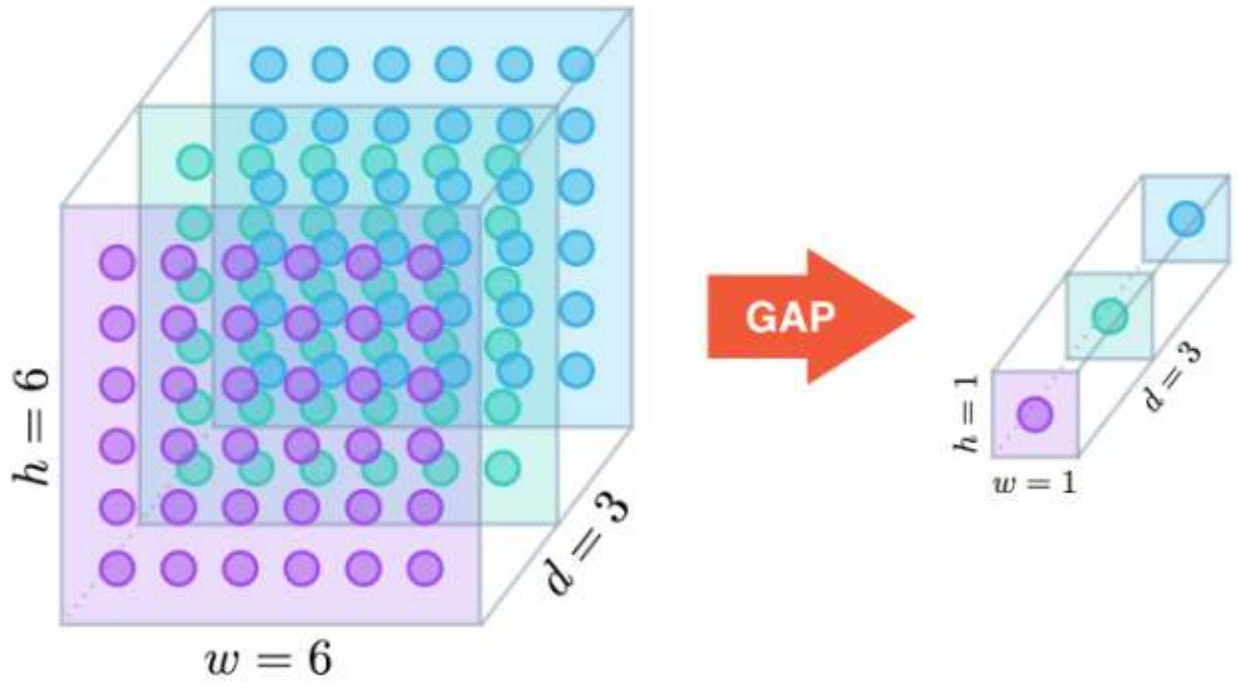
- Avoids fully-connected layers, this can reduce overfitting
- NiN's design reduce the number of required model parameters
- However, in practice, this design requires increased training time

```
def net():
    return tf.keras.models.Sequential([
        nin_block(96, kernel_size=11, strides=4, padding='valid'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding='same'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding='same'),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
        tf.keras.layers.Dropout(0.5),
        # There are 10 label classes
        nin_block(10, kernel_size=3, strides=1, padding='same'),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Reshape((1, 1, 10)),
        # Transform the four-dimensional output into two-dimensional output
        # with a shape of (batch size, 10)
        tf.keras.layers.Flatten(),
        ])
```



7. 3 Network in Network (NiN)

Global Average Pooling



Global Average Pooling

https://ganghee-lee.tistory.com/39

7. 4 Networks with Parallel Concatenations

Inception Blocks in GoogLeNet

- 2014 ImageNet winner, combining **NiN** and **repeated blocks** idea
- One focus of the paper was which sized convolution kernels are best
- From 1x1 ~ 11x11 usage, they found variously sized kernels are working well
- Inception(“we need to go deeper”)

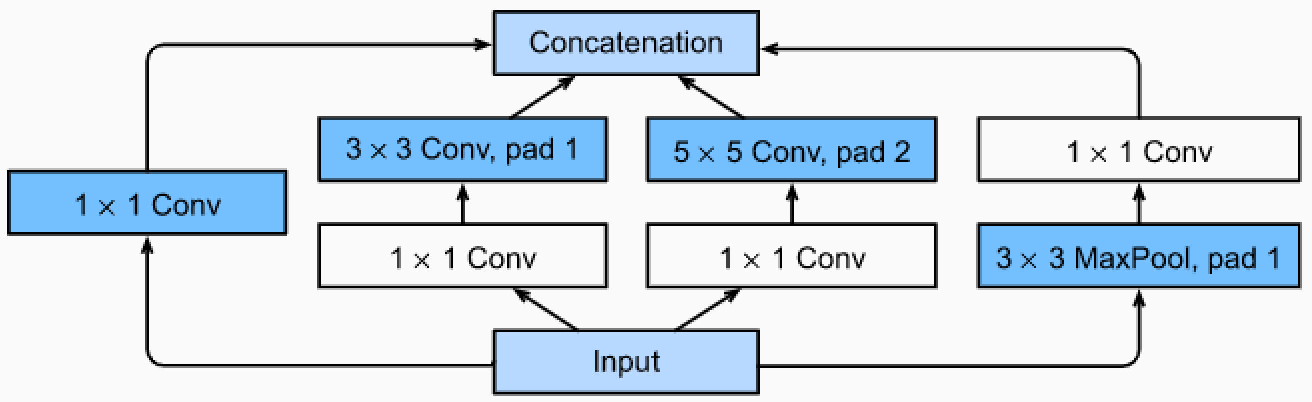


Fig. 7.4.1 Structure of the Inception block.

7. 4 Networks with Parallel Concatenations

Inception Blocks in GoogLeNet

```
class Inception(tf.keras.Model):
    # `c1`--`c4` are the number of output channels for each path
    def __init__(self, c1, c2, c3, c4):
        super().__init__()
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = tf.keras.layers.Conv2D(c1, 1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = tf.keras.layers.Conv2D(c2[0], 1, activation='relu')
        self.p2_2 = tf.keras.layers.Conv2D(c2[1], 3, padding='same',
                                           activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = tf.keras.layers.Conv2D(c3[0], 1, activation='relu')
        self.p3_2 = tf.keras.layers.Conv2D(c3[1], 5, padding='same',
                                           activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = tf.keras.layers.MaxPool2D(3, 1, padding='same')
        self.p4_2 = tf.keras.layers.Conv2D(c4, 1, activation='relu')
```

```
def call(self, x):
    p1 = self.p1_1(x)
    p2 = self.p2_2(self.p2_1(x))
    p3 = self.p3_2(self.p3_1(x))
    p4 = self.p4_2(self.p4_1(x))
    # Concatenate the outputs on the channel dimension
    return tf.keras.layers.Concatenate()([p1, p2, p3, p4])
```

- To gain intuition, they explore the image in a variety of filter size
- Also we can allocate different amount of parameters for different filters

7. 4 Networks with Parallel Concatenations

GoogLeNet Model

- Total of 9 inception blocks and global average pooling
- Max pooling between inception blocks reduces the dimensionality
- First module is similar to AlexNet and LeNet
- Stack of blocks is inherited from VGG

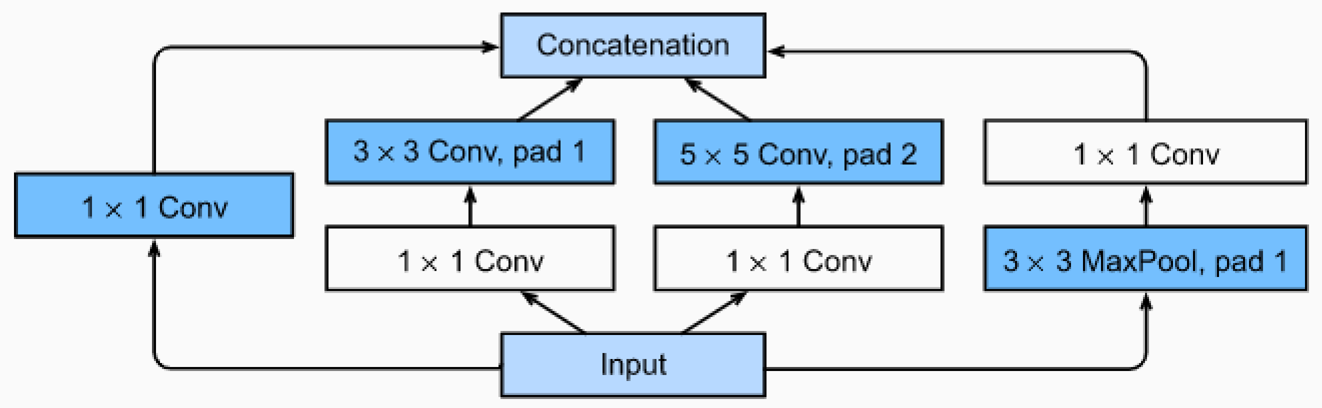


Fig. 7.4.1 Structure of the Inception block.