Fundamentals of Convolutional Neural Networks(CNN)

Prof. Dr. Shamim Akhter

Professor, Dept. of CSE

Ahsanullah University of Science and Technology

Kernels and Filters

- A main component of CNN is filter
  - which is a square matrix that has nK × nK dimension, where nK is an integer and is usually a small number, like 3 or 5.
  - Sometimes filters are also called kernels.
  - Kernels are used to do sharpening, blurring, embossing, and so on during image processing.

Example: Four different filters

The following kernel will allow the detection of horizontal edges

$$\mathfrak{I}_H = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

The following kernel will allow the detection of vertical edges

$$\mathfrak{I}_V = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

The following kernel will allow the detection of edges when luminosity changes drastically

$$\mathfrak{I}_L = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

The following kernel will blur edges in an image

$$\mathfrak{I}_B = -\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Convolution

Let's see how it works. Consider two tensors, both with dimensions 3×3. The convolution operation is done by applying the following formula:

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} * \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix} = \sum_{i=1}^{9} a_i k_i$$

In this case, the result is merely the sum of each element, a_i, multiplied by the respective element, k_i. In more typical matrix formalism, this formula could be written with a double sum as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} k_{ij}$$

In advanced applications, the images may even have higher resolution. To understand how to apply convolution when we have matrices with different dimensions, let's consider a matrix A that is 4×4

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

And a Kernel K that we will take for this example to be 3×3

$$K = \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix}$$

$$A = \begin{pmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} & \boldsymbol{a_3} & a_4 \\ \boldsymbol{a_5} & \boldsymbol{a_6} & \boldsymbol{a_7} & a_8 \\ \boldsymbol{a_9} & \boldsymbol{a_{10}} & \boldsymbol{a_{11}} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix} \quad B_1 = A_1 * K = a_1k_1 + a_2k_2 + a_3k_3 + k_4a_5 + k_5a_5 + k_6a_7 + k_7a_9 + k_8a_{10} + k_9a_{11}$$

$$A=\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix} \qquad B_2 = A_2 * K = a_2k_1 + a_3k_2 + a_4k_3 + a_6k_4 + a_7k_5 + a_8k_6 + a_{10}k_7 + a_{11}k_8 + a_{12}k_9$$

$$A_3=\begin{pmatrix} a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \\ a_{13} & a_{14} & a_{15} \end{pmatrix} \qquad B_3 = A_3 * K = a_5k_1 + a_6k_2 + a_7k_3 + a_9k_4 + a_{10}k_5 + a_{11}k_6 + a_{13}k_7 + a_{14}k_8 + a_{15}k_9$$

$$A_4=\begin{pmatrix} a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \\ a_{14} & a_{15} & a_{16} \end{pmatrix} \qquad B_4 = A_4 * K = a_6k_1 + a_7k_2 + a_8k_3 + a_{10}k_4 + a_{11}k_5 + a_{12}k_6 + a_{14}k_7 + a_{15}k_8 + a_{16}k_9$$

$$B=\begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

In the previous process, we moved our 3 × 3 region always one column to the right and one row down. The number of rows and columns, in this example 1, is called the stride and is often indicated with s. Stride s = 2 means simply that we shift our 3 × 3 region two columns to the right and two rows down at each step.

In a more formal definition, convolution with stride s in the neural network context is a process that takes a tensor A of dimensions $n_A \times n_A$ and a kernel K of dimensions $n_K \times n_K$ and gives as output a matrix B of dimensions $n_B \times n_B$ with

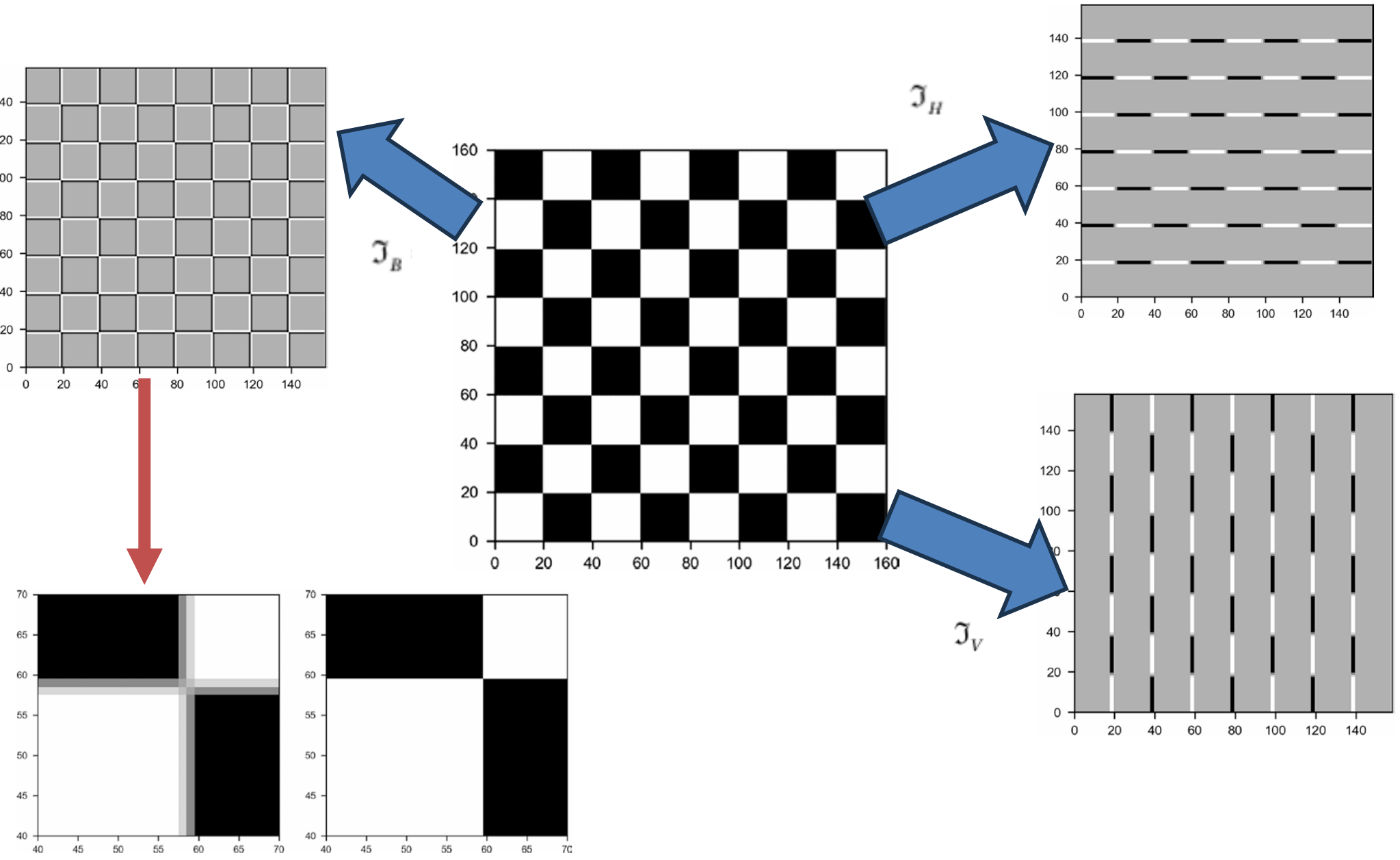
$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor$$

we chose $s = 3$, and since we have $n_A = 5$ and $n_K = 3$, B will be a scalar as a result.

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor = \left\lfloor \frac{5-3}{3} + 1 \right\rfloor = \left\lfloor \frac{5}{3} \right\rfloor = 1$$

Now let's apply convolution to this image with the different kernels with stride $s = 1$.

Using the kernel, $\mathfrak{I}_H$ will detect the horizontal edges. This can be



Pooling: max pooling

- Pooling is the second operation in CNNs.

To perform max pooling, we need to define a region of size $n_K \times n_K$, analogous to what we did for convolution. Let's consider $n_K = 2$. What we need to do is start on the top-left corner of our matrix A and select a $n_K \times n_K$ region, in our case 2×2 from A. Here we would select

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix} \qquad \begin{pmatrix} a_1 & a_2 \\ a_5 & a_6 \end{pmatrix} \qquad B_1 = \max_{i=1,2,5,6} a_i$$

$$\begin{pmatrix} a_3 & a_4 \\ a_7 & a_8 \end{pmatrix} \qquad B_2 = \max_{i=3,4,7,8} a_i$$

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Figure 3-11. *A visualization of pooling with stride $s = 2$*

For example, applying max-pooling to the input A

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix} \Rightarrow B = \begin{pmatrix} 5 & 11 \\ 15 & 21 \end{pmatrix}$$

Padding

- Sometimes, when dealing with images, getting a result from a convolution operation with dimensions different from the original image is not optimal. This is when padding is necessary.
- Add rows of pixels on the top and bottom and columns of pixels on the right and left of the final images so the resulting matrices are the same size as the original.

Only as a reference, in case you use padding p (the width of the rows and columns you use as padding), the final dimensions of the matrix B, in case of both convolution and pooling, is given by

$$n_B = \left\lfloor \frac{n_A + 2p - n_K}{s} + 1 \right\rfloor$$

```
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
```

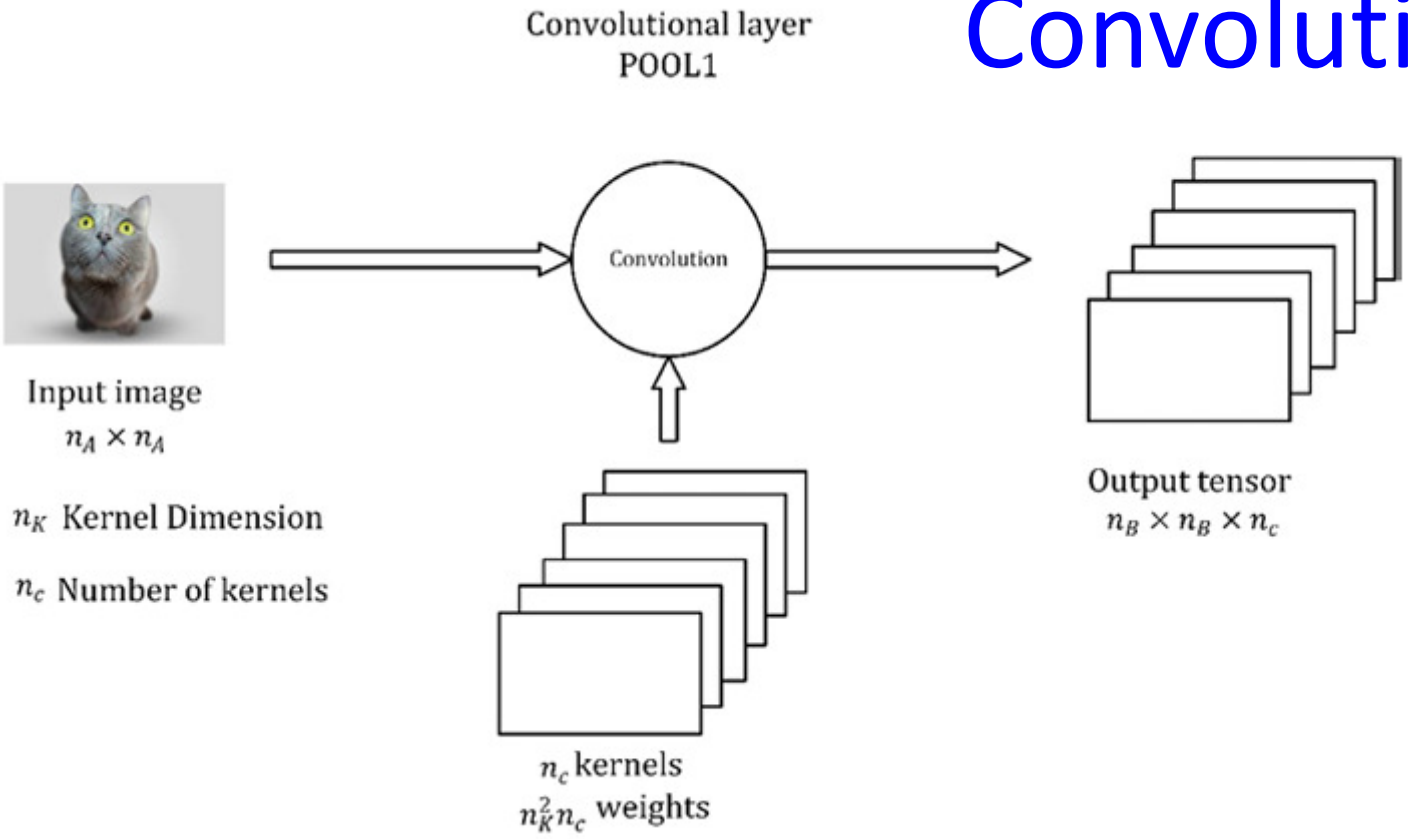
Note When dealing with real images, you always have color images, coded in three channels: RGB. That means that convolution and pooling must be done in three dimensions: width, height, and color channel. This will add a layer of complexity to the algorithms.

Building Blocks of a CNN

- Convolution and pooling operations are used to build the layers in CNNs.

- In CNNs typically you can find the following layers:
  - Convolutional layers
  - Pooling layers
  - Fully connected layers: a layer where neurons are connected to all neurons of previous and subsequent layers.

Convolutional Layers



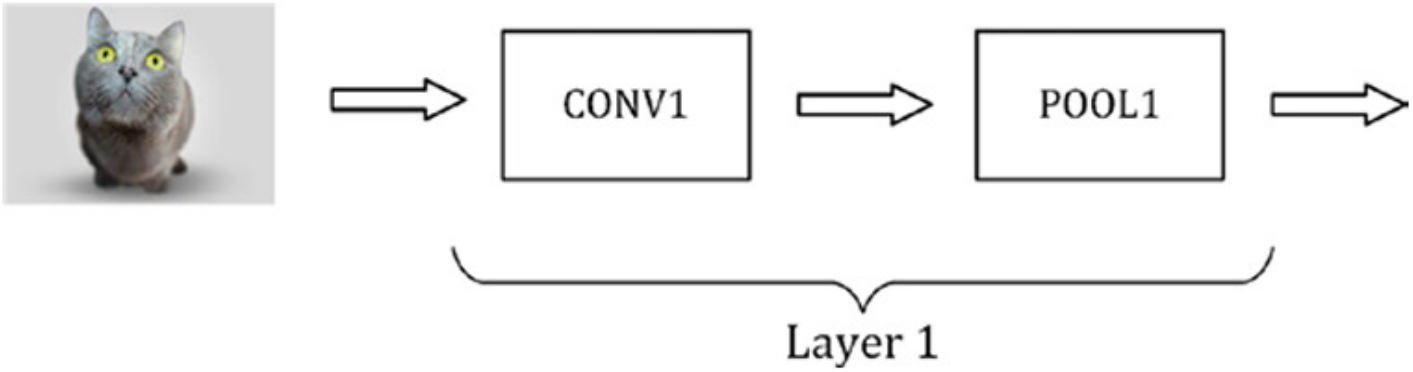
However, what are the weights in this layer?

- The weights, or the parameters that the network learns during the training phase, are the elements of the kernel themselves.
- We have n_c kernels, each of $n_K \times n_K$ dimensions. That means that we have $n_K^2 n_c$ parameters in a convolutional layer.
- since for each filter there is also a bias term that you will need to add.

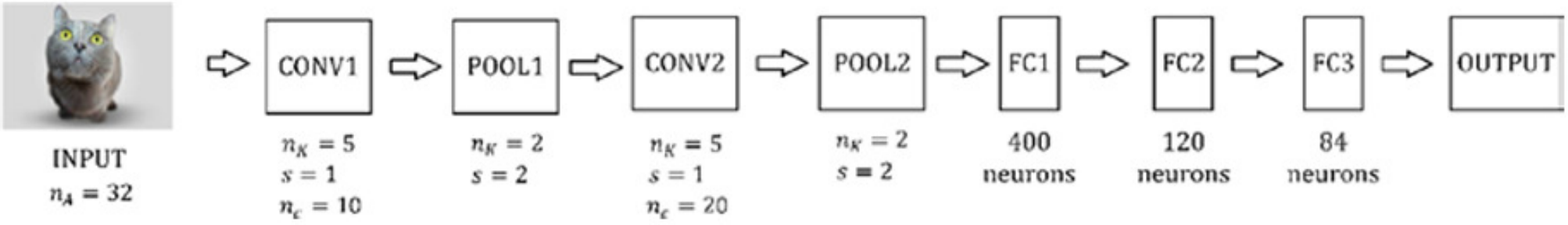
$$n_C \cdot n_K \cdot n_K + n_C$$

Convolution- Pooling layer

Note A pooling layer has no parameter to learn, but it introduces additional hyperparameters: n_K and stride v. Typically, in pooling layers, you don't use any padding, since one of the reasons to use pooling is often to reduce the dimensionality of the tensors.



LeNet-5 with Softmax activation function



Dense/FC Layer

- The weights are the ones you know from traditional feed-forward networks.
- So the number depends on the number of neurons and the number of neurons in the preceding and subsequent layers.

CNN Implementation

```
num_classes = 10
```

Now let's define a function to create and compile our Keras model:

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28),
    activation='relu'))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy',
    optimizer='adam', metrics=['accuracy'])
    return model
```

Why 1 is here?

What is Dropout?

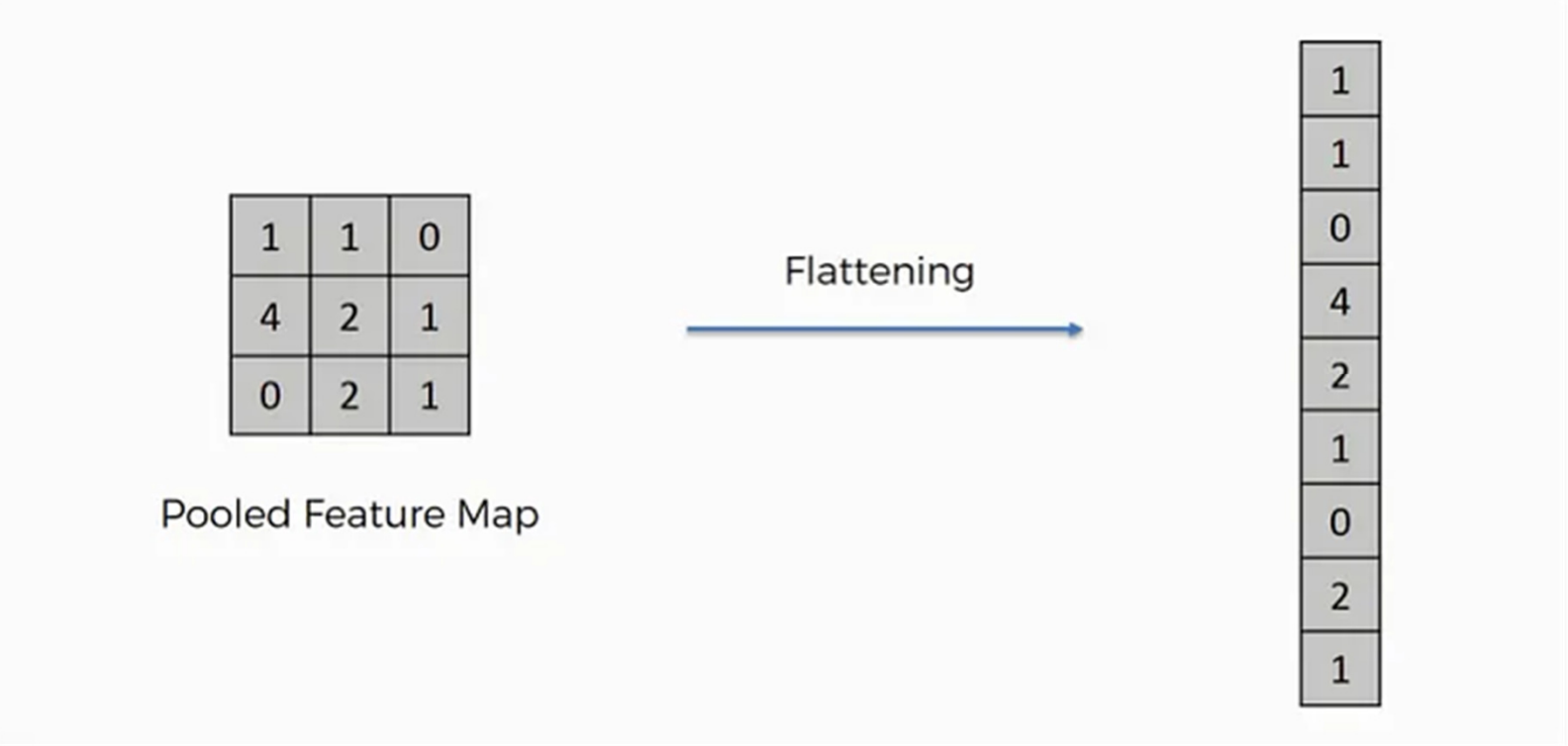
What is Flatten?

Dropout Regularization Techniques

- ***Dropout*** is a technique where randomly selected neurons are ignored during training. They are "dropped out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass, and weight updates are not applied to the neuron on the backward pass.
- Generally, use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A low probability has minimal effect and a high value results in under-learning by the network.

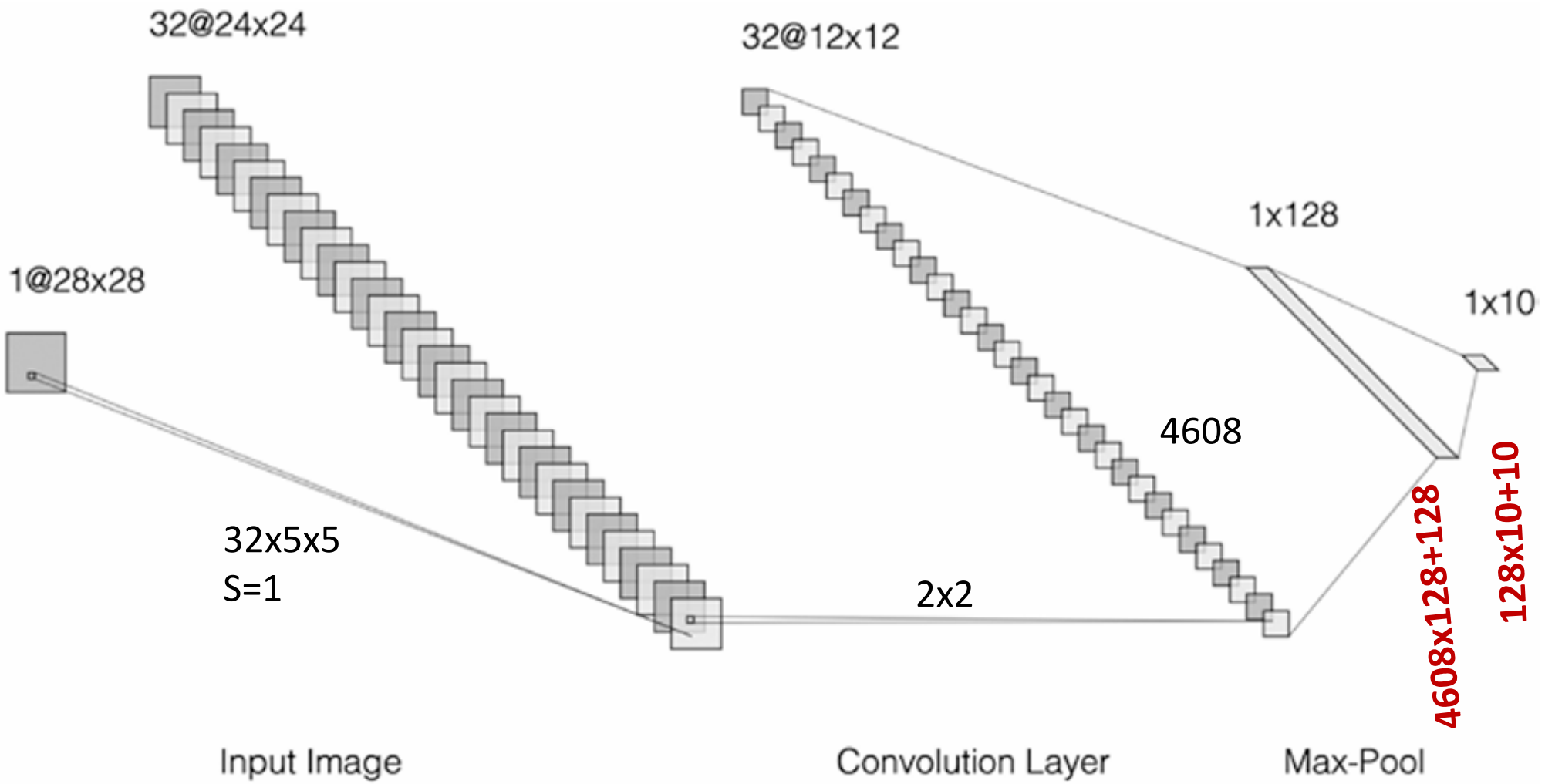
Flatten Layer

- Intuition behind flattening layer is to converts data into 1-dimentional array for feeding next layer. we flatted output of convolutional layer into single long feature vector.



FC and Dense Layer

- A fully-connected layer is a layer that has a connection/edge across every pair of nodes from two node sets. For example, if you want to build a layer with N1 input neurons and N2 output neurons, the number of connections/edges will be N1 X N2, which is also the shape of the weight matrix.
- As the number of connections can be very large (think of connecting thousands of neurons to one another), the layer is going to be highly dense which is why these layers are also called Dense Layer.


32@24x24
32@12x12
1x128
1x10
1@28x28
4608
32x5x5
S=1
2x2
4608x128+128
128x10+10
Input Image
Convolution Layer
Max-Pool

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 24, 24)	832
max_pooling2d_1 (MaxPooling2	(None, 32, 12, 12)	0
dropout_1 (Dropout)	(None, 32, 12, 12)	0
flatten_1 (Flatten)	(None, 4608) 32x12x12	0
dense_1 (Dense)	(None, 128) 4608x128+128	589952
dense_2 (Dense)	(None, 10) 128x10+10	1290

Total params: 592,074

Trainable params: 592,074

Non-trainable params: 0

Effect of Convolution

The first image in the test dataset

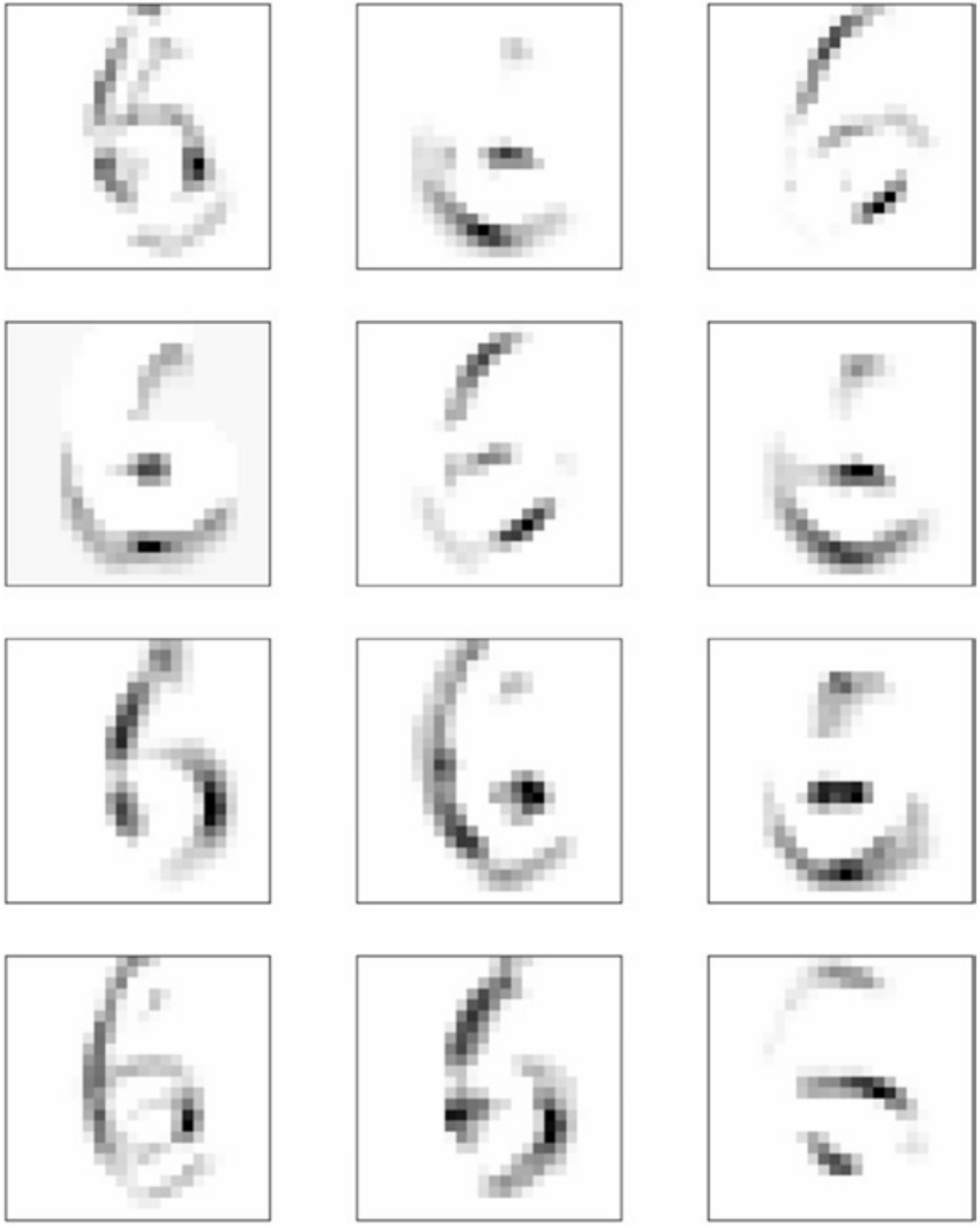


Figure 3-17. *The test image (a 6) convoluted with the first 12 filters learned by the network*

Effect of Max Pooling

Figure 3-19. *The output of the pooling layer when applied to the first 12 tensors coming from the convolutional layer*

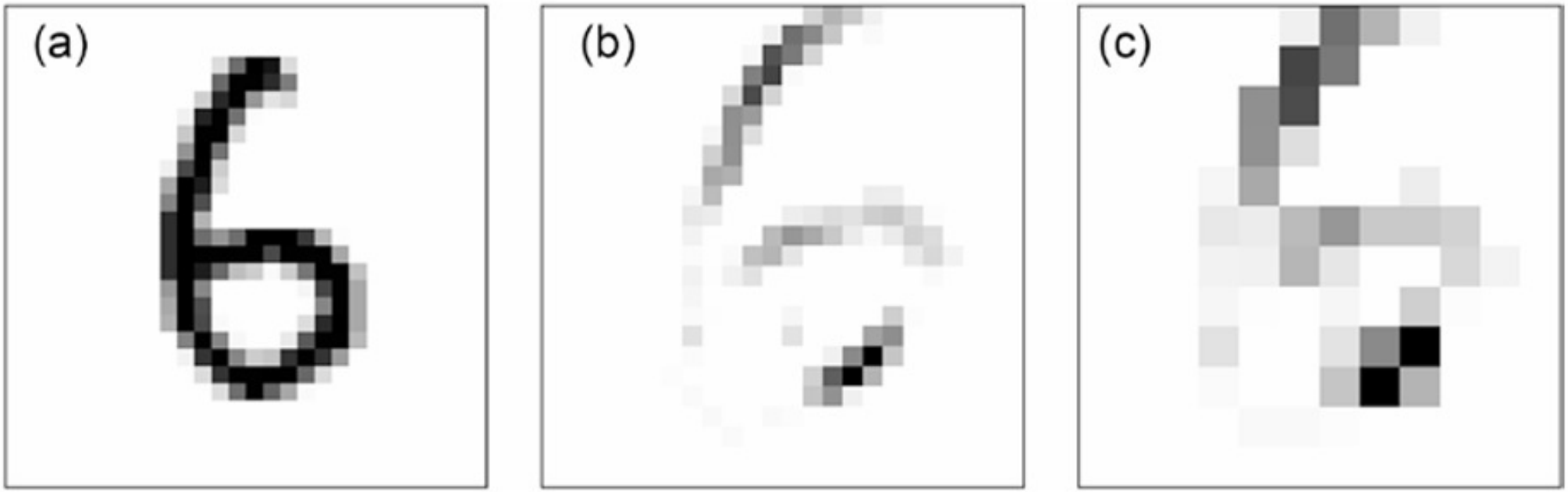


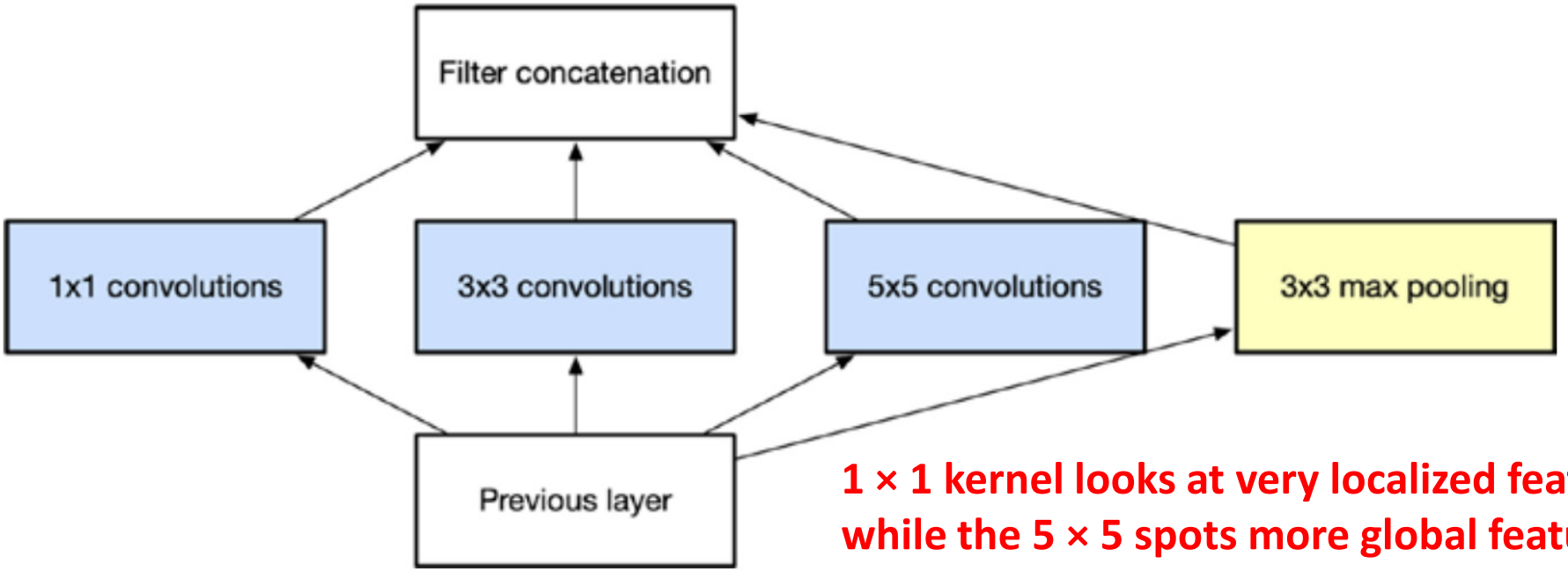
Figure 3-20. *The original test image as in the dataset (in panel a); the image convoluted with the third learned filter (panel b); the image convoluted with the third filter after the max pooling layer (panel c)*

Going Deeper with Convolutions

- Problems of “classical” CNNs
  - It isn’t easy to get the right kernel size. Each image is different. Typically, larger kernels are good for more globally distributed information, and smaller ones for locally distributed information.
  - Deep CNNs are prone to overfitting.
  - Training and inference of networks with many parameters is computationally intensive.

Inception Module: Naïve Version

- Overcome the difficulties of CNN
  - networks are wider instead of deeper
    - perform convolution with multiple-size kernels in parallel, to detect features at different sizes simultaneously, instead of adding convolutional layer after layer sequentially.
    - convolution with 1 × 1, 3 × 3, and 5 × 5 kernels, and even max pooling at the same time in parallel

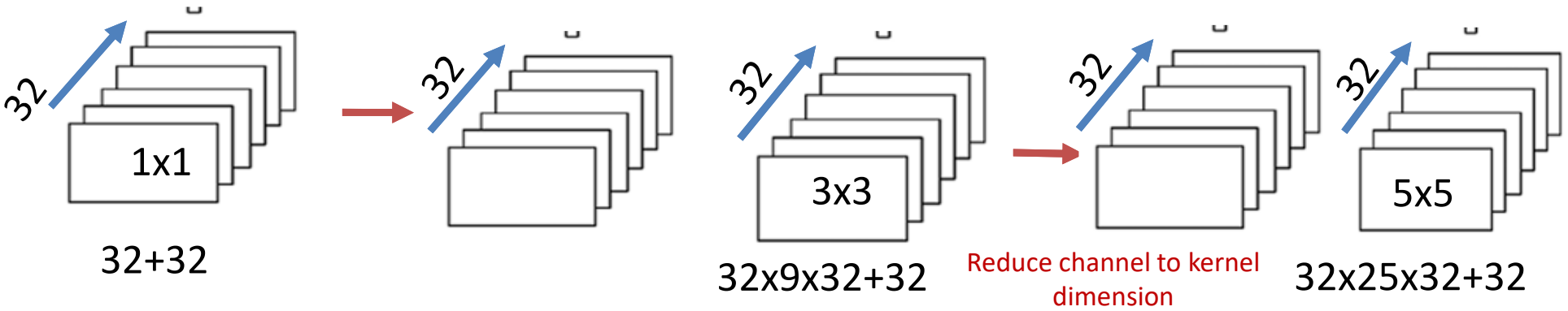


1 × 1 kernel looks at very localized features, while the 5 × 5 spots more global features.

Number of Parameters @ Naïve Inception

- Let's use 32 kernels for all layers.
  - 1 × 1 convolutions: 64 parameters [32+32]
  - 3 × 3 convolutions: 320 parameters [9x32+32]
  - 5 × 5 convolutions: 832 parameters [25x32+32]
  - max-pooling does not have learnable parameters

Models	# of Parameters	
Sequential Processing	64+9248+25632=34,944	Classical
Parallel Processing	64+320+832=1,216	Naïve Inception **30 times faster**

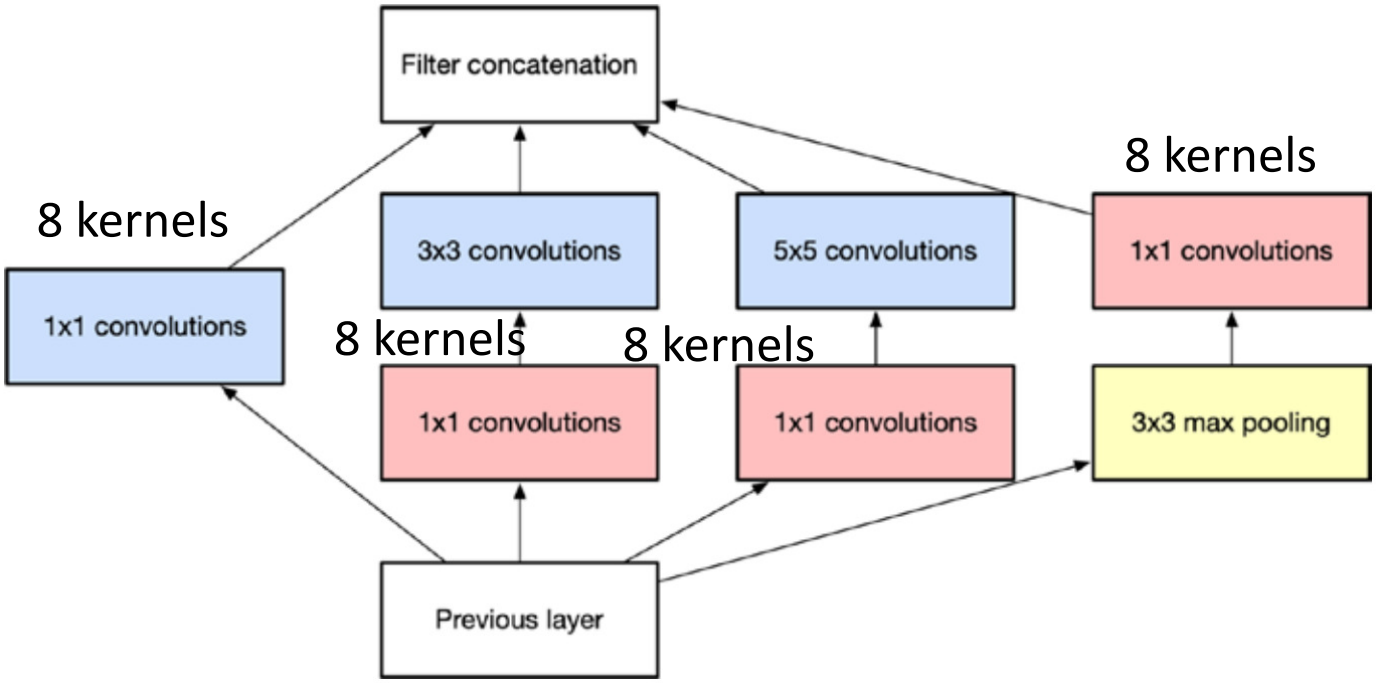


Inception Module: Dimension Reduction

- In the naïve inception module, we get a smaller number of learnable parameters concerning classical CNNs, but we can do even better

- We can use 1 × 1 convolutions at the right places (mainly before the higher dimension convolutions) to reduce dimensions.

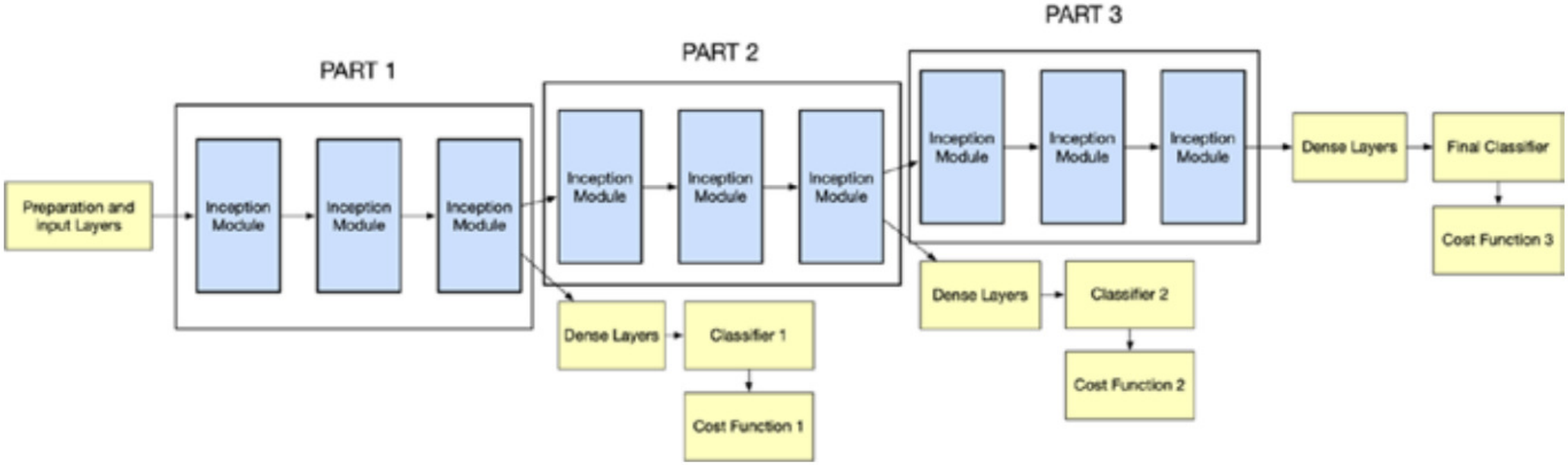
- Suppose that the previous layer is the output of a previous operation and that its output has the dimensions of **256, 28, 28**.



Models	# of Parameters
Naïve Inception	256x1x8+8=2056, 256x9x8+8=18,440, 256x25x8+8=51,208 **Total=71,704**
Parallel Processing	2056, (2056+8x9x8+8=2056+584=2640), (2056+8x25x8+8=2056+1608=3664), 2056 **Total=10,416** [An inception network is simply built by stacking lots of those modules one after the other]

GoogLeNet: Multiple Cost Functions

- Stacks several inception models one after the other
  - the middle layers tend to “die”.



Introduced two intermediate loss functions and then computed the total loss function as a weighted sum of the auxiliary losses, effectively using a total loss

```
Total Loss = Cost Function 1 + 0.3 * (Cost Function 2) + 0.3 *
(Cost Function 3)
```

Of course, the auxiliary losses are used only in training and not during inference.

Pre-Trained Networks

- Pre-trained deep learning models available to use.

```
model = VGG16(weights='imagenet')
```

- If weights is None the weights are randomly initialized. That means that you get the VGG16 architecture and you can train it yourself. But be aware, that it has **roughly 138 million parameters,** so you will need a **really big training dataset.**

- If you use the value imagenet, the weights are the ones obtained by training the network with the imagenet dataset

Transfer Learning

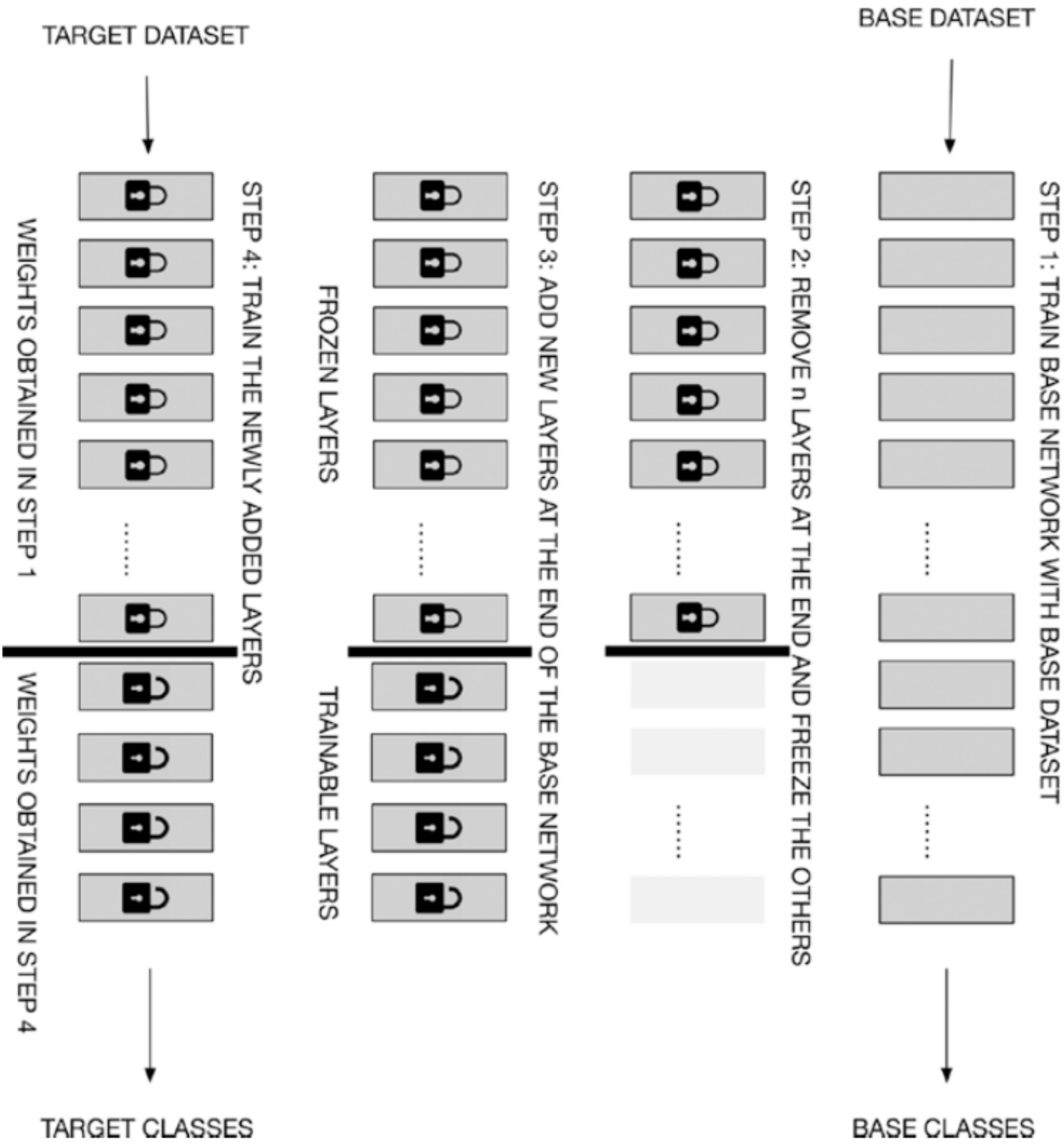
- Transfer learning is a technique where a model trained to solve a specific problem is re-purposed for a new challenge related to the first problem.
- The imagenet dataset can be used to classify dogs' images but should not be used for speech recognition.

- In image recognition with CNN typically,
  - the first layers will learn to detect generic features, and
  - the last layers will be able to detect more specific ones.
  - In a classification problem, the last layer will have N softmax neurons (for classifying N classes), and therefore must learn to be very specific to your problem.

How does Transfer Learning work?

- A network with n_L layers
  - train a base network (or get a pre-trained model) on a big dataset (called a base dataset). The dataset should be problem-specified.
  - the new or target dataset will be much smaller than the previous dataset.
  - train a new network, called a target network, on the target dataset.
    - The target network will typically have the same first n_k (with $n_k < n_L$) layers of our base network.
    - The learnable parameters of the first layers (let's say 1 to n_k, with $n_k < n_L$) are inherited from the base pre-trained network and are not changed during the training of the target network.
    - Only the last and new layers (from layer n_K to n_L) are trained.

Schematic representation of the transfer learning



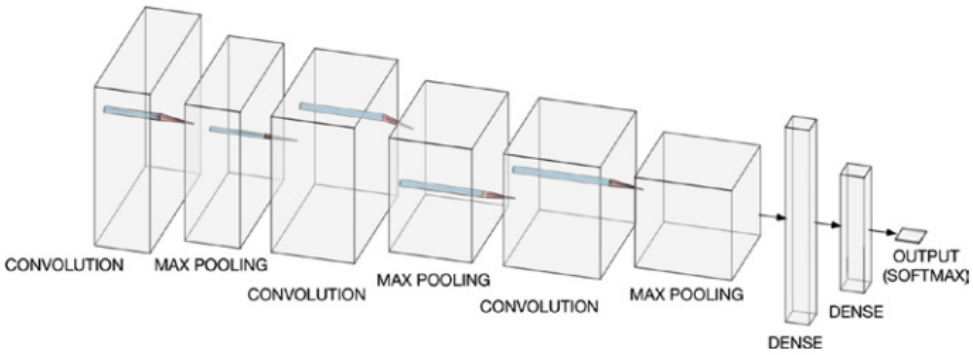
A Dog and Cat Problem

Dogs vs. Cats | Kaggle, 800MB

Training Image (3000, 150, 150, 3), Testing Image (1000, 150, 150, 3)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 148, 148, 16)      448
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 74, 74, 16)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 72, 72, 64)        9280
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 36992)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               18940416
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 19,024,513
Trainable params: 19,024,513
Non-trainable params: 0
```

Naïve CNN Approach




```
batch_size = 30
num_classes = 2
epochs = 2
input_shape = (150, 150, 3)
model.fit(x=train_imgs_scaled, y=train_labels_enc,
                    validation_data=(validation_imgs_scaled,
                    validation_labels_enc),
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1)
```

With two epochs, 69% validation accuracy and 70% training accuracy.

A Dog and Cat Problem

Transfer Learning Approach 1

```
base_model=vgg16.VGG16(include_top=False, weights='imagenet')

from tensorflow.keras.layers import
Dense,GlobalAveragePooling2D
x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x)
preds=Dense(1,activation='softmax')(x)
model=Model(inputs=base_model.input,outputs=preds)
```

```
Total params: 15,242,050
Trainable params: 15,242,050
```

All the 22 layers are trainable at the moment. To be able to really do transfer learning, we need to freeze all layers of the VGG16 base network.

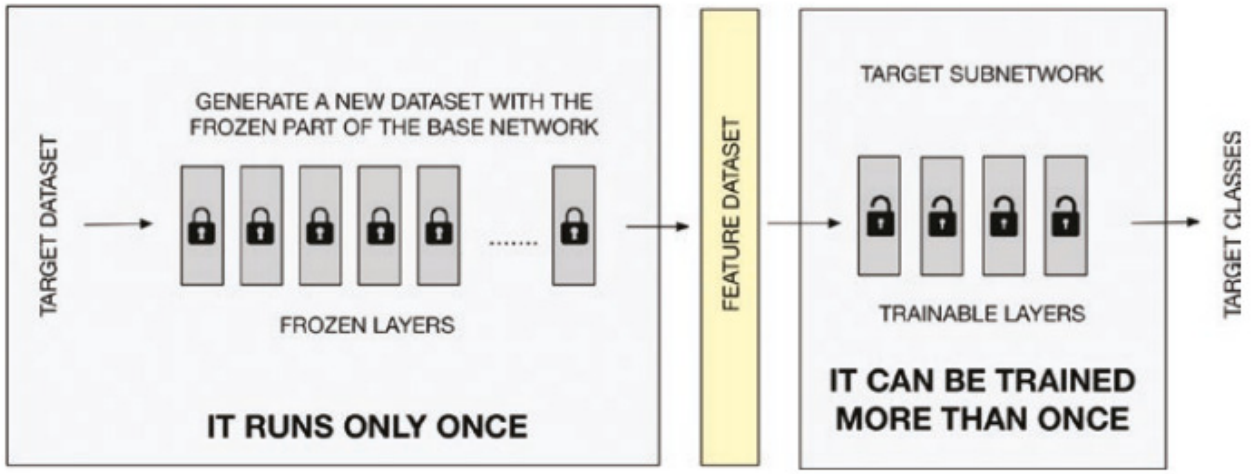
To do that we can do the following:

```
for layer in model.layers[:20]:
    layer.trainable=False
for layer in model.layers[20:]:
    layer.trainable=True
```

The result will be an astounding 88% in two epochs.
An incredibly better result than before!

A Dog and Cat Problem

Transfer Learning Approach 2



90% accuracy in a few seconds.

One epoch takes only six seconds, in comparison to the 4.5 minutes in Approach1

100 epochs??

```
vgg = vgg16.VGG16(include_top=False, weights='imagenet',
                                     input_shape=input_shape)

output = vgg.layers[-1].output
output = keras.layers.Flatten()(output)
vgg_model = Model(vgg.input, output)

vgg_model.trainable = False
for layer in vgg_model.layers:
    layer.trainable = False

def get_ features(model, input_imgs):
    features = model.predict(input_imgs, verbose=0)
    return features

train_features_vgg = get_features(vgg_model, train_imgs_scaled)
validation_features_vgg = get_features(vgg_model, validation_
imgs_scaled)
```

```
input_shape = vgg_model.output_shape[1]

model = Sequential()
model.add(InputLayer(input_shape=(input_shape,)))
model.add(Dense(512, activation='relu', input_dim=input_shape))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.Adam(lr =1e-4),
              metrics=['accuracy'])
```