1.CNN Design Patterns:

Pattern 1: Feature extraction and classification—Convolutional nets (ConvNets) are typically composed of two parts: the feature extraction part, which consists of a series of convolutional layers; and the classification part, which consists of a series of fully connected layers (figure 1a).

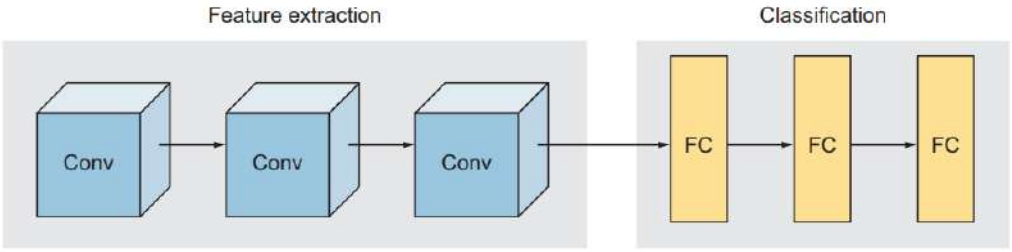


Fig 1a: Convolutional nets generally include feature extraction and classification.

Pattern 2: Image depth increases, and dimensions decrease—The input data at each layer is an image. With each layer, we apply a new convolutional layer over a new image. This pushes us to think of an image in a more generic way. First, you see that each image is a 3D object that has a height, width, and depth. Depth is referred to as the color channel, where depth is 1 for grayscale images and 3 for color images. In the later layers, the images still have depth, but they are not colors per se: they are feature maps that represent the features extracted from the previous layers. That's why the depth increases as we go deeper through the network layers. In figure 1b, the depth of an image is equal to 96; this represents the number of feature maps in the layer. So, that's one pattern you will always see: the image depth increases, and the dimensions decrease.

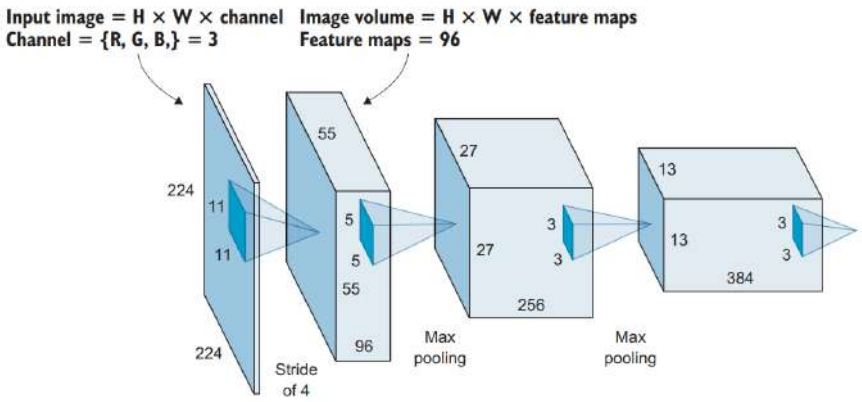


Fig1b: Image depth increases, and the dimensions decrease.

Pattern 3: Fully connected layers—All fully connected layers in a network either have the same number of hidden units or decrease at each layer. It is rare to find a network where the number of units in the fully connected layers increases at each layer. Research has found that keeping the number of units constant doesn't hurt the neural network, so it may be a good approach if you want to limit the number of choices you have to make when designing your network. This way, all you have to do is to pick a number of units per layer and apply that to all your fully connected layers.

2. LeNet-5:

The LeNet-5 architecture is composed of five weight layers that is three convolutional, pooling, and two fully connected layers.

i) LeNet Architecture:

The architecture of LeNet-5 is shown in figure 1c:

INPUT IMAGE ⇒ C1 ⇒ TANH ⇒ S2 ⇒ C3 ⇒ TANH ⇒ S4 ⇒ C5 ⇒ TANH ⇒ FC6 ⇒ SOFTMAX7

where C is a convolutional layer, S is a subsampling or pooling layer, and FC is a fully connected layer.

ii)LeNet-5 implementation in Keras :

To implement LeNet-5 in Keras, the main takeaways for building the LeNet-5 network:

- ***Number of filters in each convolutional layer***—As you can see in figure 1c the depth (number of filters) of each convolutional layer is as follows: C1 has 6, C3 has 16, C5 has 120 layers.
- ***Kernel size of each convolutional layer***—The paper specifies that the kernel_size is 5 × 5.
- ***Subsampling (pooling) layers—***A subsampling (pooling) layer is added after each convolutional layer. The receptive field of each unit is a 2 × 2 area (for example, pool_size is 2). Note that the LeNet-5 creators used average pooling, which computes the average value of its inputs, instead of the max pooling layer that we used in our earlier projects, which passes the maximum value of its inputs. You can try both if you are interested, to see the difference. For this experiment, we are going to follow the paper's architecture.
- ***Activation function***—As mentioned before, the creators of LeNet-5 used the tanh activation function for the hidden layers because symmetric functions are believed to yield faster convergence compared to sigmoid functions (figure 1d).

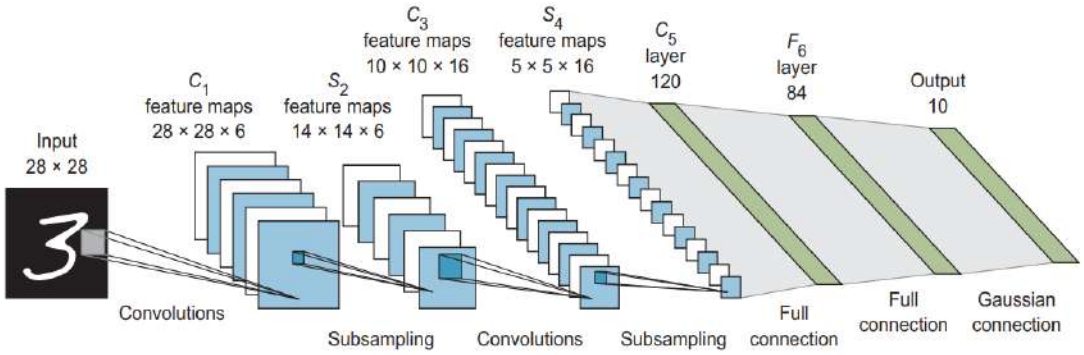


Figure 1c: LeNet architecture

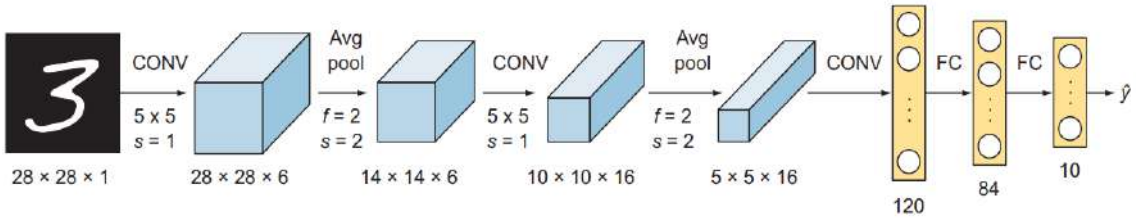


Figure 1d The LeNet architecture consists of convolutional kernels of size 5 × 5; pooling layers; an activation function (tanh); and three fully connected layers with 120, 84, and 10 neurons, respectively.

```
from keras.models import Sequential
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense

model = Sequential()

# C1 Convolutional Layer
model.add(Conv2D(filters = 6, kernel_size = 5, strides = 1, activation = 'tanh',
                 input_shape = (28,28,1), padding = 'same'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size = 2, strides = 2, padding = 'valid'))

# C3 Convolutional Layer
model.add(Conv2D(filters = 16, kernel_size = 5, strides = 1,activation = 'tanh',
                 padding = 'valid'))

# S4 Pooling Layer
model.add(AveragePooling2D(pool_size = 2, strides = 2, padding = 'valid'))

# C5 Convolutional Layer
model.add(Conv2D(filters = 120, kernel_size = 5, strides = 1,activation = 'tanh',
                 padding = 'valid'))

model.add(Flatten())

# FC6 Fully Connected Layer
model.add(Dense(units = 84, activation = 'tanh'))

# FC7 Output layer with softmax activation
model.add(Dense(units = 10, activation = 'softmax'))

model.summary()
```

Imports the Keras model and layers

Instantiates an empty sequential model

Flattens the CNN output to feed it fully connected layers

Prints the model summary (figure 5.5)

LeNet-5 is a small neural network by today's standards. It has 61,706 parameters, compared to millions of parameters in more modern network.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 6)         156
_________________________________________________________________
average_pooling2d_1 (Average (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 16)        2416
_________________________________________________________________
average_pooling2d_2 (Average (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 1, 120)         48120
_________________________________________________________________
flatten_1 (Flatten)          (None, 120)               0
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

Figure 1e: LeNet-5 model summary

LeNet hyperparameters tuning:

LeCun and his team used scheduled decay learning where the value of the learning rate was decreased using the following schedule: 0.0005 for the first two epochs, 0.0002 for the next three epochs, 0.00005 for the next four, and then 0.00001 thereafter. The authors trained their network for 20 epochs. Let's build a lr_schedule function with this schedule. The method takes an integer epoch number as an argument and returns the learning rate (lr):

```
def lr_schedule(epoch):
    if epoch <= 2:
        lr = 5e-4
    elif epoch > 2 and epoch <= 5:
        lr = 2e-4
    elif epoch > 5 and epoch <= 9:
        lr = 5e-5
    else:
        lr = 1e-5
    return lr
```

lr is 0.0005 for the first two epochs, 0.0002 for the next three epochs (3 to 5), 0.00005 for the next four (6 to 9), then 0.00001 thereafter (more than 9).

We use the lr_schedule function in the following code snippet to compile the model:

```
from keras.callbacks import ModelCheckpoint, LearningRateScheduler

lr_scheduler = LearningRateScheduler(lr_schedule)
checkpoint = ModelCheckpoint(filepath='path_to_save_file/file.hdf5',
                             monitor='val_acc',

                             verbose=1,
                             save_best_only=True)

callbacks = [checkpoint, lr_reducer]

model.compile(loss='categorical_crossentropy', optimizer='sgd',
              metrics=['accuracy'])
```

Now start the network training for 20 epochs, as mentioned in the paper:

```
hist = model.fit(X_train, y_train, batch_size=32, epochs=20,
          validation_data=(X_test, y_test), callbacks=callbacks,
          verbose=2, shuffle=True)
```

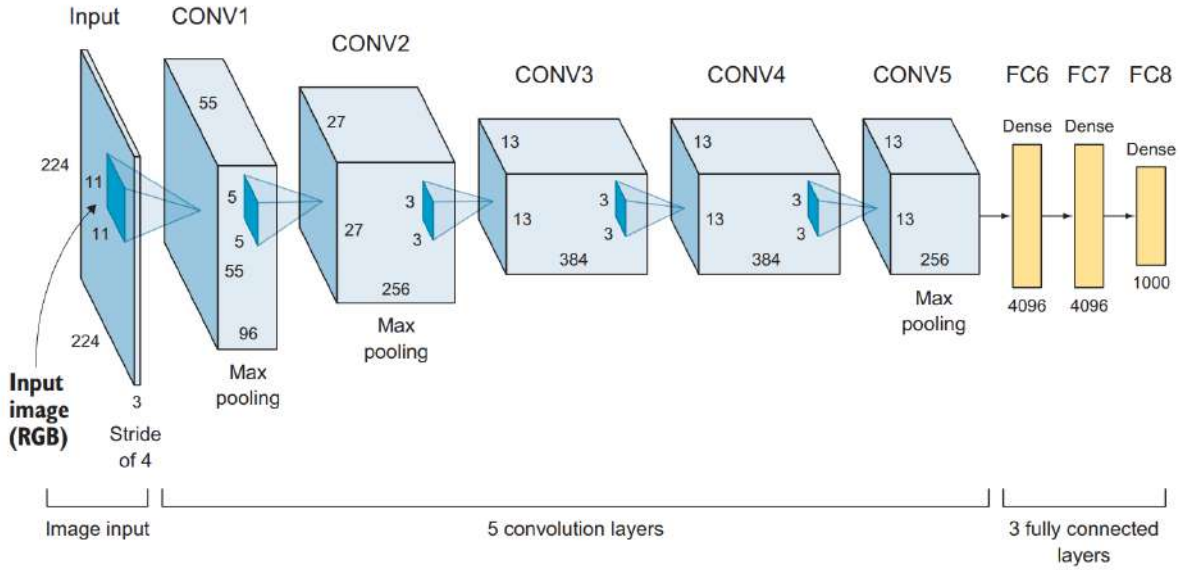
LeNet performance on the MNIST dataset When you train LeNet-5 on the MNIST dataset, you will get above 99% accuracy (see the code notebook with the book's code). Try to re-run this experiment with the ReLU activation function in the hidden layers, and observe the difference in the network performance.

3. Alex Net: (introduced in 2012)

LeNet performs very well on the MNIST dataset. But it turns out that the MNIST dataset is very simple because it contains grayscale images (1 channel) and classifies into only 10 classes, which makes it an easier challenge. The main motivation behind AlexNet was to build a deeper network that can learn more complex functions.

AlexNet (figure below) is a neural network trained on 1.2 million high-resolution images into 1,000 different classes of the ImageNet dataset. AlexNet has a lot of similarities to LeNet but is much deeper (more hidden layers) and bigger (more filters per layer). They have similar building blocks: a series of convolutional and pooling layers stacked on top of each other followed by fully connected layers and a softmax. We've seen that LeNet has around 61,000 parameters, whereas AlexNet has about 60 million parameters and 650,000 neurons, which gives it a larger learning capacity to understand more complex features. Alex Net has achieved a remarkable performance for image classification.

i) AlexNet architecture:



The architecture consists of:

- Convolutional layers with the following kernel sizes: 11 × 11, 5 × 5, and 3 × 3
- Max pooling layers for images downsampling
- Dropout layers to avoid overfitting
- Unlike LeNet, ReLU activation functions in the hidden layers and a softmax activation in the output layer AlexNet 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.

The architecture can be represented in text as follows:

INPUT IMAGE ⇒ CONV1 ⇒ POOL2 ⇒ CONV3 ⇒ POOL4 ⇒ CONV5 ⇒ CONV6 ⇒ CONV7 ⇒ POOL8 ⇒ FC9 ⇒ FC10 ⇒ SOFTMAX7

ii)Alex Net Features:

The features used are new approaches to convolutional neural networks:

- **ReLU Nonlinearity**. AlexNet uses Rectified Linear Units (ReLU) instead of the tanh function, which was standard at the time. ReLU's advantage is in training time; a CNN using ReLU was able to reach a 25% error on the CIFAR-10 dataset six times faster than a CNN using tanh.
- **Multiple GPUs.** Back in the day, GPUs were still rolling around with 3 gigabytes of memory (nowadays those kinds of memory would be rookie numbers). This was especially bad because the training set had 1.2 million images. AlexNet
- allows for multi-GPU training by putting half of the model's neurons on one GPU and the other half on another GPU. Not only does this mean that a bigger model can be trained, but it also cuts down on the training time.
- **Overlapping Pooling.** CNNs traditionally "pool" outputs of neighboring groups of neurons with no overlapping. However, when the authors introduced overlap, they saw a reduction in error by about 0.5% and found that models with overlapping pooling generally find it harder to overfit.

The Overfitting Problem. AlexNet had 60 million parameters, a major issue in terms of overfitting. Two methods were employed to reduce overfitting:

- **Data Augmentation.** The authors used label-preserving transformation to make their data more varied. Specifically, they generated image translations and horizontal reflections, which increased the training set by a factor of 2048. They also performed Principal Component Analysis (PCA) on the RGB pixel values to change the intensities of RGB channels, which reduced the top-1 error rate by more than 1%.
- **Dropout.** This technique consists of "turning off" neurons with a predetermined probability (e.g. 50%). This means that every iteration uses a different sample of the model's parameters, which forces each neuron to have more robust features that can be used with other random neurons. However, dropout also increases the training time needed for the model's convergence.

The Results. On the 2010 version of the ImageNet competition, the best model achieved 47.1% top-1 error and 28.2% top-5 error. AlexNet vastly outpaced this with a 37.5% top-1 error and a 17.0% top-5 error. AlexNet is able to recognize off-center objects and most of its top five classes for each image are reasonable. AlexNet won the 2012 ImageNet competition with a top-5 error rate of 15.3%, compared to the second place top-5 error rate of 26.2%.

iii)AlexNet implementation in Keras:

As shown in figure 5.7, the network contains eight weight layers: the first five are convolutional, and the remaining three are fully connected. The output of the last fully connected layer is fed to a 1000-way softmax that produces a distribution over the 1,000 class labels.

AlexNet input starts with 227 × 227 × 3 images. If you read the paper, you will notice that it refers to a dimensions volume of 224 × 224 × 3 for the input images. But the numbers make sense only for 227 × 227 × 3 images (figure 5.7).

The layers are stacked together as follows:

- CONV1—The authors used a large kernel size (11). They also used a large stride (4), which makes the input dimensions shrink by roughly a factor 4 (from 227 × 227 to 55 × 55). We calculate the dimensions of the output as follows:

((227-11) /4) + 1 = 55 and the depth is the number of filters in the convolutional layer (96). The output dimensions are 55 × 55 × 96.

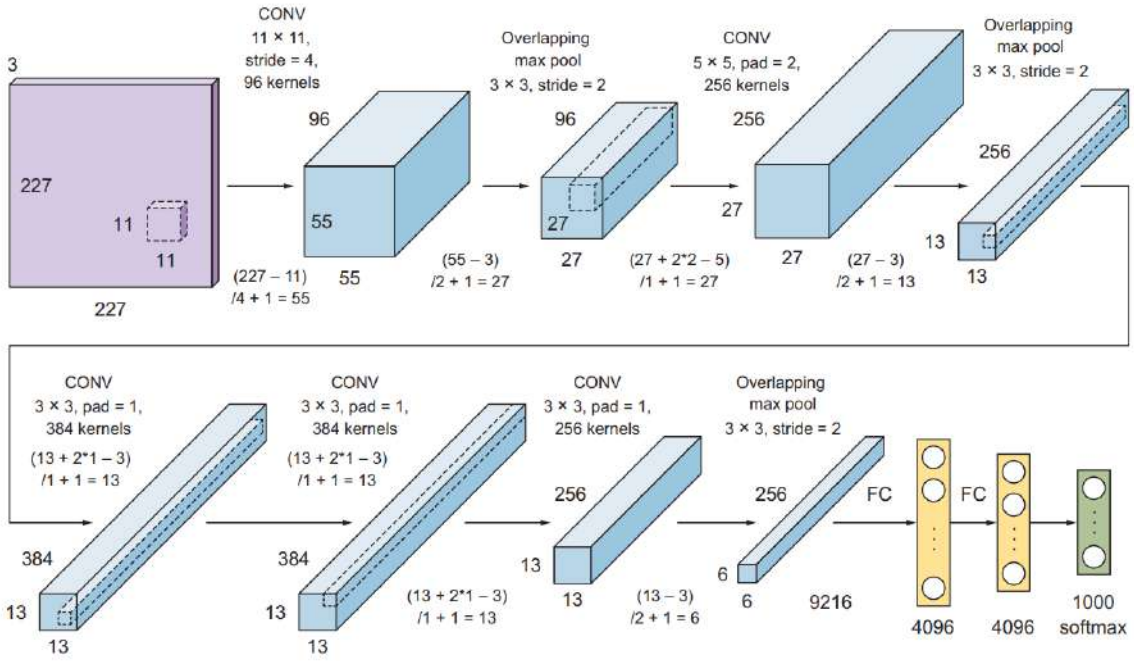


Figure 5.7 AlexNet contains eight weight layers

- POOL with a filter size of 3 × 3—This reduces the dimensions from 55 × 55 to 27 × 27: + 1 = 27 The pooling layer doesn't change the depth of the volume. The output dimensions are 27 × 27 × 96. Similarly, we can calculate the output dimensions of the remaining layers:
- CONV2—Kernel size = 5, depth = 256, and stride = 1
- POOL—Size = 3 × 3, which downsamples its input dimensions from 27 × 27 to 13 × 13
- CONV3—Kernel size = 3, depth = 384, and stride = 1
- CONV4—Kernel size = 3, depth = 384, and stride = 1
- CONV5—Kernel size = 3, depth = 256, and stride = 1
- POOL—Size = 3 × 3, which downsamples its input from 13 × 13 to 6 × 6
- Flatten layer—Flattens the dimension volume 6 × 6 × 256 to 1 × 9,216
- FC with 4,096 neurons
- FC with 4,096 neurons
- Softmax layer with 1,000 neurons

Both LeNet and AlexNet have many hyperparameters to tune. The authors of those networks had to go through many experiments to set the kernel size, strides, and padding for each layer, which makes the networks harder to understand and manage. VGGNet (explained next) solves this problem with a very simple, uniform architecture.

```
from keras.models import Sequential
from keras.regularizers import l2
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense,
    Activation,MaxPool2D, BatchNormalization, Dropout

model = Sequential()
# 1st layer (CONV + pool + batchnorm)
model.add(Conv2D(filters= 96, kernel_size= (11,11), strides=(4,4),
     padding='valid',
                input_shape = (227,227,3)))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2)))
model.add(BatchNormalization())

# 2nd layer (CONV + pool + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same',
                kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2), padding='valid'))
model.add(BatchNormalization())

# layer 3 (CONV + batchnorm)
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
     kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 4 (CONV + batchnorm)
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
                kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 5 (CONV + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same',
                kernel_regularizer=l2(0.0005)))

model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2), padding='valid'))

model.add(Flatten())

# layer 6 (Dense layer + dropout)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 7 (Dense layers)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 8 (softmax output layer)
model.add(Dense(units = 1000, activation = 'softmax'))

model.summary()
```

Imports the Keras model, layers, and regularizers

Instantiates an empty sequential model

The activation function can be added on its own layer or within the Conv2D function as we did in previous implementations.

Note that the AlexNet authors did not add a pooling layer here.

Similar to layer 3

Flattens the CNN output to feed it fully connected layers

Prints the model summary

When you print the model summary, you will see that the number of total parameters is 62 million:

```
----------------------------------------
Total params: 62,383, 848
Trainable params: 62,381, 096
Non-trainable params: 2,752
```

iv)AlexNet Hyperparameter Tuning:

AlexNet was trained for 90 epochs, which took 6 days on two Nvidia Geforce GTX 580 GPUs simultaneously. This is why you will see that the network is split into two pipelines, with an initial learning rate of 0.01 with a momentum of 0.9. The lr is then divided by 10 when the validation error stops improving:

Sets the SGD optimizer with lr of 0.01 and momentum of 0.9

Reduces the learning rate by 0.1 when the validation error plateaus

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=np.sqrt(0.1))

optimizer = keras.optimizers.sgd(lr = 0.01, momentum = 0.9)

model.compile(loss='categorical_crossentropy', optimizer=optimizer,
              metrics=['accuracy'])
```

Compiles the model

```
model.fit(X_train, y_train, batch_size=128, epochs=90,
          validation_data=(X_test, y_test), verbose=2, callbacks=[reduce_lr])
```

Trains the model and calls the reduce_lr value using callbacks in the training method

AlexNet significantly outperformed all the models. It achieved test error rate of 15.3%, compared to 26.2%.

4. VGGNet :

VGGNet was developed in 2014 by the Visual Geometry Group at Oxford University (hence the name VGG). VGGNet, also known as VGG16, consists of 16 weight layers: 13 convolutional layers and 3 fully connected layers.

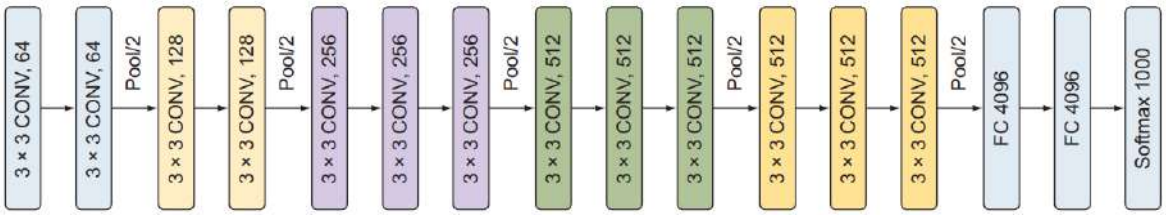
i)features of VGGNet:

VGGNet's is a simple architecture containing uniform components (convolutional and pooling layers). It improves on AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layers, respectively) with multiple 3 × 3 pool-size filters one after another.

The architecture is composed of a series of uniform convolutional building blocks followed by a unified pooling layer, where:

- All convolutional layers are 3 × 3 kernel-sized filters with a strides value of 1 and a padding value of same.
- All pooling layers have a 2 × 2 pool size and a strides value of 2.

This is followed by the traditional classifier, which is composed of fully connected layers and a softmax, as depicted in figure 5.8



i) VGGNet configurations

Simonyan and Zisserman created several configurations for the VGGNet architecture, as shown in figure 5.9. All of the configurations follow the same generic design. Configurations D and E are the most commonly used and are called VGG16 and VGG19, referring to the number of weight layers. Each block contains a series of 3 × 3 convolutional layers with similar hyperparameter configuration, followed by a 2 × 2 pooling layer.

The Table below lists the number of learning parameters (in millions) for each configuration. VGG16 yields ~138 million parameters; VGG19, which is a deeper version VGGNet, has more than 144 million parameters. VGG16 is more commonly used because it performs almost as well as VGG19 but with fewer parameters.

Network	A, A-LRN	B	C	D	E
No. of parameters	133	133	134	138	144

Table 5.1 VGGNet architecture parameters (in millions)

ConvNet configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
Input (224 x 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					

Figure 5.9 VGGNet architecture configurations

ii) Implementation of VGG16 in keras :

Configurations D (VGG16) and E (VGG19) are the most commonly used configurations because they are deeper networks that can learn more complex functions. The configuration D has 16 weight layers. VGG19 (configuration E) can be similarly implemented by adding a fourth convolutional layer to the third, fourth, and fifth blocks as you can see in figure 5.9. The VGGNet uses regularization techniques to avoid overfitting:

- L2 regularization with weight decay of 5 × 10–4. For simplicity, this is not added to the implementation that follows.
- Dropout regularization for the first two fully connected layers, with a dropout ratio set to 0.5.

```
model = Sequential()
```

Instantiates an empty sequential model

```
# block #1
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same', input_shape=(224,224, 3)))
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #2
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #3
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #4
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #5
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1),
     activation='relu',
                  padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #6 (classifier)
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))

model.summary()
```

Prints the model summary

When you print the model summary, you will see that the number of total parameters is ~138 million:

```
_________________________________________________________________
Total params: 138,357, 544
Trainable params: 138,357, 544
Non-trainable params: 0
```

iii) VGG Net Hyperparameter Tuning:

VGGNet used the same hyperparameters tuning similar to that of AlexNet. The training of VGG is carried out using mini-batch gradient descent with momentum of 0.9. The learning rate is initially set to 0.01 and then decreased by a factor of 10 when the validation set accuracy stops improving.

The VGGNet performance VGG16 achieved a top-5 with a error rate of 8.1% on the ImageNet dataset compared to 15.3% achieved by AlexNet. VGG19 did even better: it was able to achieve a top-5 error rate of ~7.4%. It is worth noting that in spite of the larger number of parameters and the greater depth of VGGNet compared to AlexNet, VGGNet required fewer epochs to converge due to the implicit regularization imposed by greater depth and smaller convolutional filter sizes.

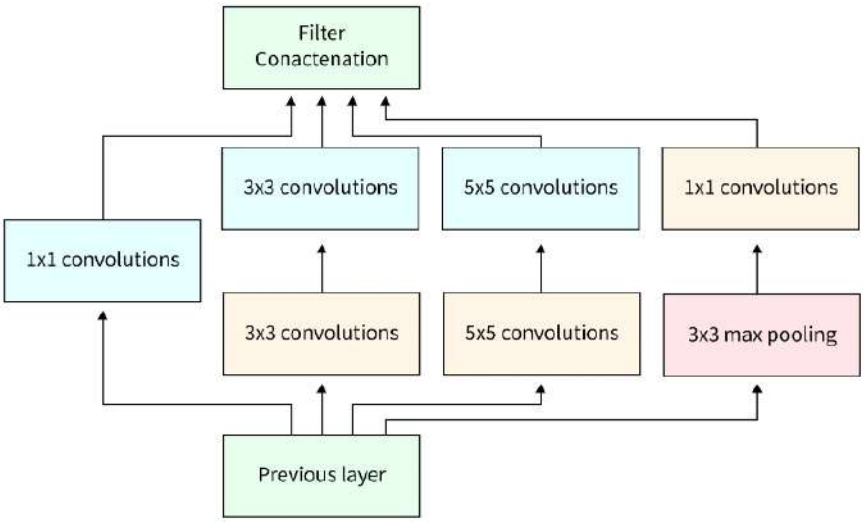
5. Inception and GoogLeNet:

InceptionNet is a **convolutional neural network (CNN)** architecture that Google developed to improve upon the performance of previous CNNs on the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** benchmark. It uses "inception modules" that apply a combination of **1x1, 3x3, and 5x5** convolutions on the input data and utilizes auxiliary classifiers to improve performance. InceptionNet won the 2014 ILSVRC competition and has been used in various applications, including image classification, object detection, and image segmentation.

InceptionNet is a convolutional neural network architecture developed by **Google** in **2014**. It is known for using inception modules, blocks of layers that learn a combination of local and global features from the input data. InceptionNet was designed to be more efficient and faster to train than other deep convolutional neural networks. It has been used in image classification, object detection, and face recognition and has been the basis for popular neural network architectures such as **Inception-v4 and Inception-ResNet**.

i)What is InceptionNet

InceptionNet is a convolutional neural network (CNN) designed for image classification tasks and developed for the ImageNet Large Scale Visual Recognition Challenge. InceptionNet is known for using inception modules, blocks of layers designed to learn a combination of local and global features from the input data. These modules are composed of smaller convolutional and pooling layers, which are combined to allow the network to learn spatial and temporal features from the input data. InceptionNet was designed to train more efficiently and faster than other deep CNNs. It has been widely used in various applications, including image classification, object detection, and face recognition.

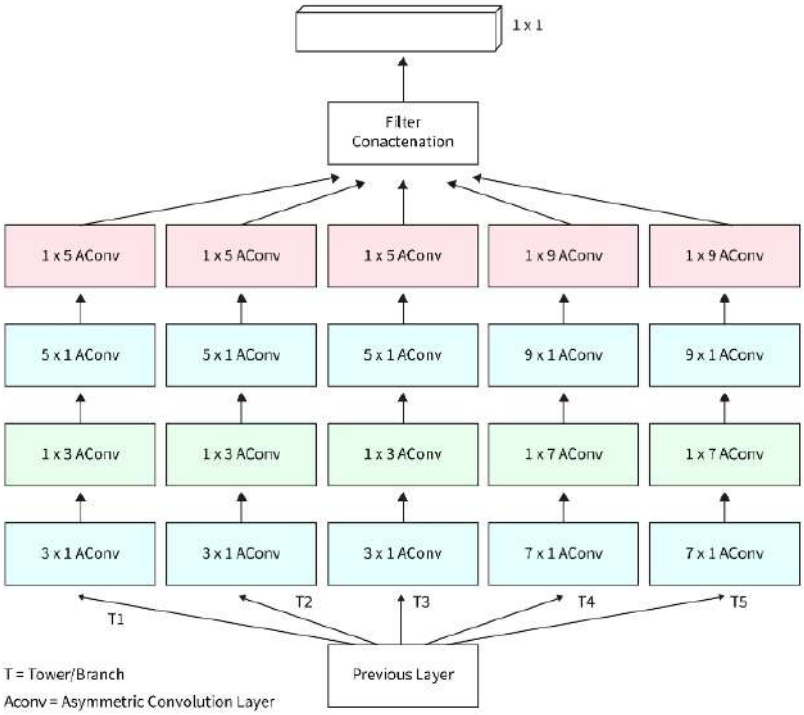


ii)Inception Blocks

Conventional convolutional neural networks typically use convolutional and pooling layers to extract features from the input data. However, these networks are limited in capturing local and global features, as they typically focus on either one or the other. The inception blocks in the InceptionNet architecture are intended to solve the problem of learning a combination of local and global features from the input data.

Inception blocks address this problem using a modular design that allows the network to learn a variety of feature maps at different scales. These feature maps are then concatenated together to form a more comprehensive representation of the input data. This allows the network to capture a wide range of features, including both low-level and high-level features, which can be useful for tasks such as image classification.

By using inception blocks, the InceptionNet architecture can learn a more comprehensive set of features from the input data, which can improve the network's performance on tasks such as image classification.

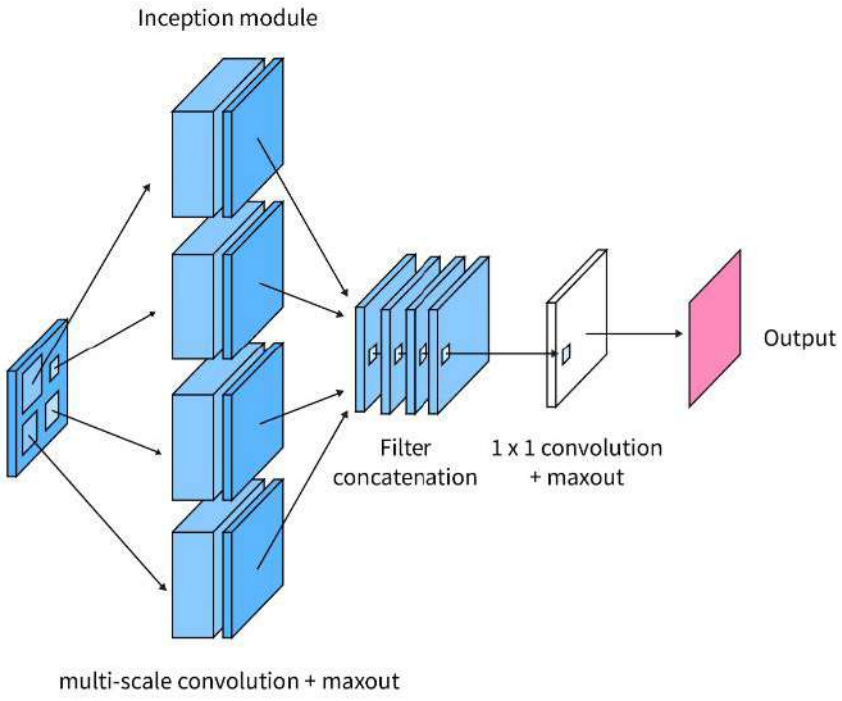


iii)Inception Modules

Inception modules are a key feature of the InceptionNet convolutional neural network architecture. They are blocks of layers designed to learn a combination of local and global features from the input data. Inception modules comprise a series of smaller convolutional and pooling layers, which are combined to allow the network to learn spatial and temporal features from the input data.

The **idea behind the inception module** is to learn a variety of feature maps at different scales and then concatenate them together to form a more comprehensive representation of the input data. This allows the network to capture a wide range of low-level and high-level features, which can be useful for tasks such as image classification.

Inception modules can be added to the network at various points, depending on the desired complexity level and the input data size. We can also modify them by changing the number and size of the convolutional and pooling layers and the type of **nonlinear activation function** used.



iv)How does an Inception Module Function Work?

- An Inception Module is a building block used in the Inception network architecture for CNNs. the Inception module improves feature extraction, improving the network's performance.
- It improves performance by allowing multiple parallel convolutional filters to be applied to the input data.
- The basic structure of an Inception Module is a combination of multiple convolutional filters of different sizes applied in parallel to the input data.
- The filters may have different kernel sizes (e.g. 3x3, 5x5) and/or different strides (e.g. 1x1, 2x2).
- **Output of each filter** is concatenated together to form a single output feature map.

- Inception Module also includes a max pooling layer, which takes the maximum value from a set of non-overlapping regions of the input data.
- This reduces the spatial dimensionality of the data and allows for translation invariance.
- The **use of multiple parallel filters** and max pooling layers allows the Inception Module to extract features at different scales and resolutions, improving the network's ability to recognize patterns in the input data.

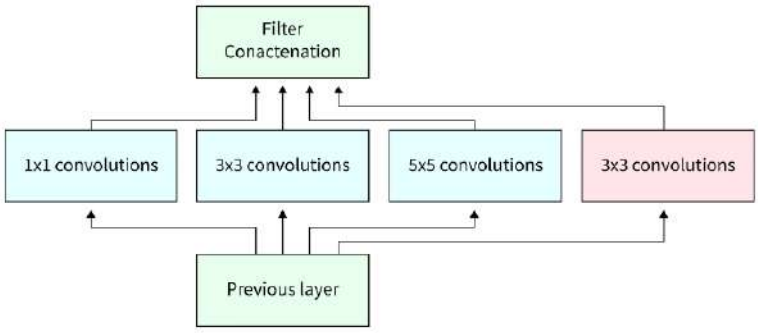
v)Why 1 X 1 Convolutions are Less Expensive?

- 1x1 convolutions are less computationally expensive than larger convolutional filters because they involve fewer parameters.
- Since the kernel size is 1x1, it only has one set of weights, much less than the number required for larger convolutional filters.
- 1x1 convolutions also require less memory to store the weights and less computation to perform the convolution.
- These smaller kernels are more efficient as they are applied to lower-dimensional feature maps, which reduces the number of operations and memory required.
- Using 1x1 convolutions also allows for dimensionality reduction, which can help to reduce the number of parameters in the network and improve performance.

Different Inception Versions

Inception v1

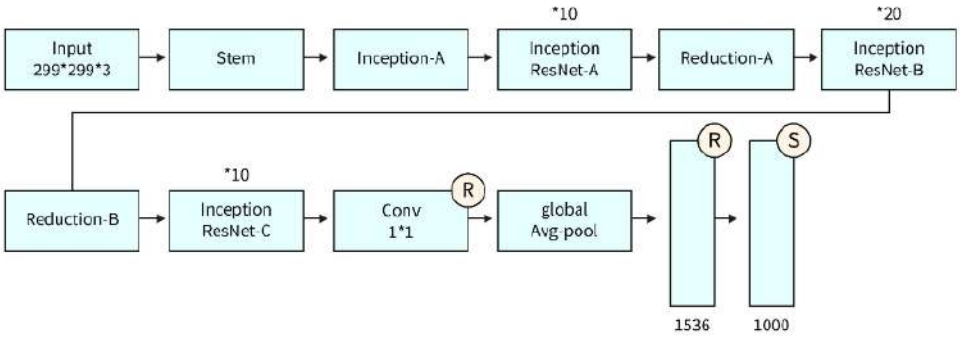
- Inception v1, also known as GoogLeNet, was the first version of the Inception network architecture.
- It was introduced in 2014 by Google and designed to improve the performance of CNNs on the ImageNet dataset.
- It uses a modular architecture, where the network comprises multiple Inception Modules stacked together.
- Each module contains multiple parallel convolutional filters of different sizes, which are applied to the input data and concatenated to form a single output feature map.
- Inception v1 includes a total of 9 Inception Modules, with max-pooling layers at different scales.
- It includes a global average pooling layer and a fully connected layer for classification.
- It achieved state-of-the-art performance on the ImageNet dataset at the time of its release.
- It was a very deep and complex network. It introduced the idea of using multiple parallel convolutional filters and showed how to reduce the computational cost using 1x1 convolution.



Inception Module, Naive Version

Inception v2

- Inception v2 is an improved version of the Inception network architecture introduced in 2015 by Google.
- It builds upon the original Inception v1 architecture and aims to improve the performance of CNNs further.
- Inception v2 uses a similar modular architecture, where the network comprises multiple Inception Modules stacked together.
- It uses a new Inception Module, called the Inception-ResNet Module, which combines the benefits of both Inception and Residual networks.
- These Inception-ResNet Modules allow for a deeper network with fewer parameters and better performance.
- Inception v2 also uses a batch normalization layer after each convolutional layer, which helps improve the network's stability and performance.
- Inception v2 achieved state-of-the-art performance on several image classification benchmarks, and its architecture has been used as a basis for many subsequent CNNs.
- Inception v2 improved the Inception architecture by introducing Inception-ResNet modules, which allow for deeper networks with fewer parameters, and batch normalization layers which improved the stability and performance of the network.



Inception-Resnet v1 and v2

Inception-Resnet v1 and v2 are deep convolutional neural network architectures developed by Google. They are based on the Inception architecture and incorporate residual connections, which bypass one or more layers. This allows the network to learn the residual mapping between the input and the output, making it easier to train deeper networks.

Feature	Inception-ResNet v1	Inception-Resnet v2
Network Depth	Deep	Shallow
Complexity	High	Low
Residual Connections	No	Yes
Feature Map Sizes	Large	Small
Training Time	Long	Short
Accuracy	Good	Better
Resource Requirements	High	Low

Inception-Resnet v1 and v2 have been widely used for **image classification** tasks and other computer vision tasks.

- **InceptionNet** is a convolutional neural network (CNN) architecture developed by **Google** in 2014. It is known for its use of inception modules, which learn a combination of local and global features from input data
- InceptionNet was designed to be more **efficient and faster** to train than other **deep CNNs**
- It has been widely used in image classification, object detection, and face recognition
- InceptionNet has also been the basis for other popular neural network architectures, such as Inception-v4 and Inception-ResNet.

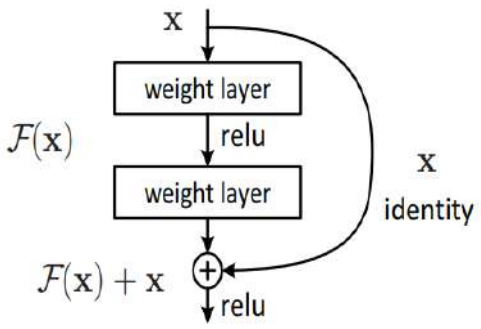
6) ResNet :

The Residual Neural Network (ResNet) was developed in 2015 by a group from the Microsoft Research team. They introduced a novel residual module architecture with skip connections. The network also features heavy batch normalization for the hidden layers. This technique allowed the team to train very deep neural networks with 50, 101, and 152 weight layers while still having lower complexity than smaller networks like VGGNet (19 layers). ResNet was able to achieve a top-5 error rate of 3.57% in the ILSVRC 2015 competition, which beat the performance of all prior ConvNets.

Residual Network solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Blocks. In this network, we use a technique called skip connections. The skip connection connects activations of a layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together.

The approach behind this network is instead of layers learning the underlying mapping, we allow the network to fit the residual mapping. So, instead of say H(x), initial mapping, let the network fit,

F(x) := H(x) - x which gives H(x) := F(x) + x.



Architecture residual blocks:

- Residual blocks contain two paths: the shortcut path and the main path.
- The main path consists of three convolutional layers, and we add a batch normalization layer to them:
  - 1 × 1 convolutional layer
  - 3 × 3 convolutional layer
  - 1 × 1 convolutional layer
- There are two ways to implement the shortcut path:
  - Regular shortcut—Add the input dimensions to the main path.
  - Reduce shortcut—Add a convolutional layer in the shortcut path before merging with the main path.

Implementation of ResNet:

When we are implementing the ResNet network, we will use both regular and reduce shortcuts. This will be clearer when you see the full implementation. But for now, we will implement bottleneck_residual_block function that takes a reduce Boolean argument. When reduce is True, this means we want to use the reduce shortcut; otherwise, it will implement the regular shortcut. The bottleneck_residual_block function takes the following arguments:

- X—Input tensor of shape (number of samples, height, width, channel)
- f—Integer specifying the shape of the middle convolutional layer's window for the main path
- filters—Python list of integers defining the number of filters in the convolutional layers of the main path
- reduce—Boolean: True identifies the reduction layer
- s—Integer (strides)

The function returns X: the output of the residual block, which is a tensor of shape (height, width, channel). The function is as follows:

```
def bottleneck_residual_block(X, kernel_size, filters, reduce=False, s=2):
    F1, F2, F3 = filters

    X_shortcut = X

    if reduce:
      X_shortcut = Conv2D(filters = F3, kernel_size = (1, 1), strides =
(s,s))(X_shortcut)
      X_shortcut = BatchNormalization(axis = 3)(X_shortcut)

    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (s,s), padding =
'valid')(X)
      X = BatchNormalization(axis = 3)(X)
      X = Activation('relu')(X)

else:
    # First component of main path
      X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding =
'valid')(X)
      X = BatchNormalization(axis = 3)(X)
      X = Activation('relu')(X)
```

Unpacks the tuple to retrieve the filters of each convolutional layer

Condition if reduce is True

Saves the input value to use it later to add back to the main path

To reduce the spatial size, applies a 1 × 1 convolutional layer to the shortcut path. To do that, we need both convolutional layers to have similar strides.

If reduce, sets the strides of the first convolutional layer to be similar to the shortcut strides.

```
    # Second component of main path
    X = Conv2D(filters = F2, kernel_size = kernel_size, strides = (1,1), padding =
      'same')(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)

    # Third component of main path
    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding =
      'valid')(X)
    X = BatchNormalization(axis = 3)(X)

    # Final step
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

Adds the shortcut value to main path and passes it through a ReLU activation

Implement Resnet50:

The total number of weight layers inside the ResNet50 network as follows:

- Stage 1: 7 × 7 convolutional layer
- Stage 2: 3 residual blocks, each containing [1 × 1 convolutional layer + 3 × 3 convolutional layer + 1 × 1 convolutional layer] = 9 convolutional layers
- Stage 3: 4 residual blocks = total of 12 convolutional layers
- Stage 4: 6 residual blocks = total of 18 convolutional layers
- Stage 5: 3 residual blocks = total of 9 convolutional layers
- Fully connected softmax layer

When we sum all these layers together, we get a total of 50 weight layers that describe the architecture of ResNet50. Building a ResNet50 function that takes input_shape and classes as arguments and outputs the model:

```
def ResNet50(input_shape, classes):
    X_input = Input(input_shape)

    # Stage 1
    X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1')(X_input)
    X = BatchNormalization(axis=3, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    # Stage 2
    X = bottleneck_residual_block(X, 3, [64, 64, 256], reduce=True, s=1)
    X = bottleneck_residual_block(X, 3, [64, 64, 256])
    X = bottleneck_residual_block(X, 3, [64, 64, 256])

    # Stage 3
    X = bottleneck_residual_block(X, 3, [128, 128, 512], reduce=True, s=2)
    X = bottleneck_residual_block(X, 3, [128, 128, 512])
    X = bottleneck_residual_block(X, 3, [128, 128, 512])
    X = bottleneck_residual_block(X, 3, [128, 128, 512])
```

Defines the input as a tensor with shape input_shape

```
# Stage 4
X = bottleneck_residual_block(X, 3, [256, 256, 1024], reduce=True, s=2)
X = bottleneck_residual_block(X, 3, [256, 256, 1024])
X = bottleneck_residual_block(X, 3, [256, 256, 1024])
X = bottleneck_residual_block(X, 3, [256, 256, 1024])
X = bottleneck_residual_block(X, 3, [256, 256, 1024])
X = bottleneck_residual_block(X, 3, [256, 256, 1024])

# Stage 5
X = bottleneck_residual_block(X, 3, [512, 512, 2048], reduce=True, s=2)
X = bottleneck_residual_block(X, 3, [512, 512, 2048])
X = bottleneck_residual_block(X, 3, [512, 512, 2048])

# AVGPOOL
X = AveragePooling2D((1,1))(X)

# output layer
X = Flatten()(X)
X = Dense(classes, activation='softmax', name='fc' + str(classes))(X)

model = Model(inputs = X_input, outputs = X, name='ResNet50')

return model
```

Creates the model

ResNet hyperparameters Tuning:

The training is carried out using mini-batch GD with momentum of 0.9, learning rate 0.1 and then decreased it by a factor of 10 when the validation error stopped improving. They also used L2 regularization with a weight decay of 0.0001 they used batch normalization right after each convolutional and before activation to speed up training:

```
from keras.callbacks import ReduceLROnPlateau

epochs = 200
batch_size = 256

reduce_lr= ReduceLROnPlateau(monitor='val_loss',factor=np.sqrt(0.1),
    patience=5, min_lr=0.5e-6)

model.compile(loss='categorical_crossentropy', optimizer=SGD,
     metrics=['accuracy'])

model.fit(X_train, Y_train, batch_size=batch_size, validation_data=(X_test,
     Y_test),
          epochs=epochs, callbacks=[reduce_lr])
```

Sets the training parameters

min_lr is the lower bound on the learning rate, and factor is the factor by which the learning rate will be reduced.

Compiles the model

Trains the model, calling the reduce_lr value using callbacks in the training method

The performance of ResNet models on CIFAR Dataset is benchmarked based on their results in the ILSVRC competition. ResNet-152 won first place in the 2015 classification competition with a top-5 error rate of 4.49% with a single model and 3.57% using an ensemble of models. This was much better than all the other networks, such as GoogLeNet (Inception), which achieved a top-5 error rate of 6.67%. ResNet also won first place in many object detection and image localization challenges. More importantly, the residual blocks concept in ResNet opened the door to new possibilities for efficiently training super-deep neural networks with hundreds of layers.

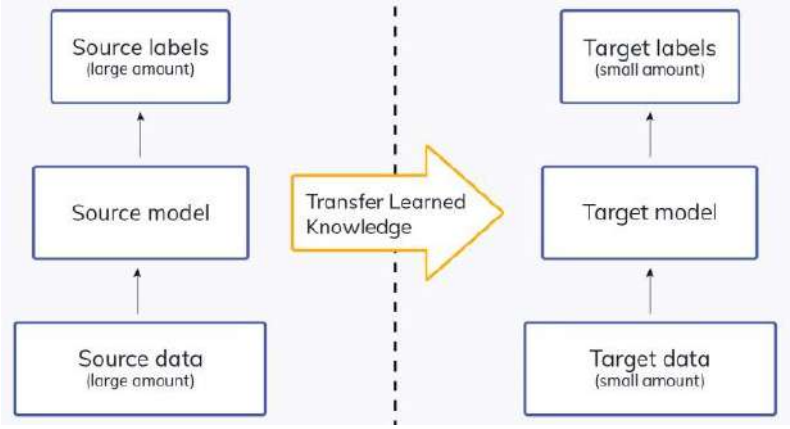
9. Transfer Learning:

Transfer learning is the transfer of the knowledge (feature maps) that the network has acquired from one task, where we have a large amount of data, to a new task where data is not abundantly available. It is generally used where a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest.

To train an image classifier that will achieve image classification accuracy near to or above the human level, we'll need massive amounts of data, large compute power, and lots of time on our hands.

Knowing that this would be a problem for people with little-to-no resources, researchers built state-of-the-art models that were trained on large image datasets like ImageNet, MS COCO, Open Images, and so on, and then shared their models with the general public for reuse.

- Transfer learning is usually the go-to approach when starting a classification and object detection project, especially when you don't have a lot of training data.
- Transfer learning migrates the knowledge learned from the source dataset to the target dataset, to save training time and computational cost.



- The neural network learns the features in your dataset step by step in increasing levels of complexity. The deeper you go through the network layers, the more image-specific the features that are learned.
- Early layers in the network learn low-level features like lines, blobs, and edges. The output of the first layer becomes input to the second layer, which produces higher-level features. The next layer assembles the output of the previous layer into parts of familiar objects, and a subsequent layer detects the objects.
- The three main transfer learning approaches are using a pretrained network as a classifier, using a pretrained network as a feature extractor, and fine-tuning.
- Using a pretrained network as a classifier means using the network directly to classify new images without freezing layers or applying model training.
- Using a pretrained network as a feature extractor means freezing the classifier part of the network and retraining the new classifier.

- Fine-tuning means freezing a few of the network layers that are used for feature extraction, and jointly training both the non-frozen layers and the newly added classifier layers of the pretrained model.
- The transferability of features from one network to another is a function of the size of the target data and the domain similarity between the source and target data.
- Generally, fine-tuning parameters use a smaller learning rate, while training the output layer from scratch can use a larger learning rate.

The advantage of pre-trained models is that they are generic enough for use in other real-world applications. For example:

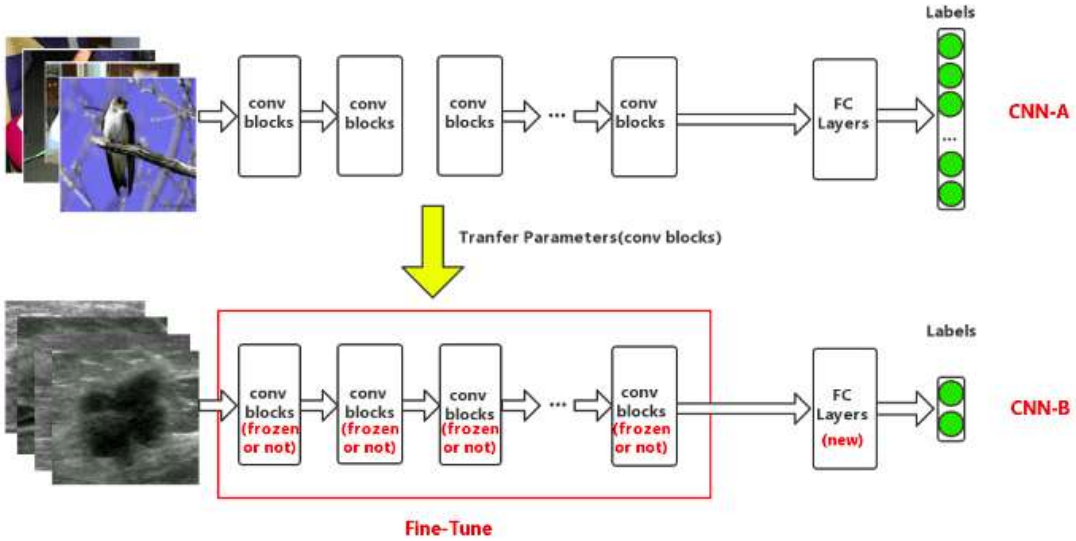
- models trained on the ImageNet can be used in real-world image classification problems. This is because the dataset contains over 1000 classes. Let's say you are an insect researcher. You can use these models and fine-tune them to classify insects.
- classifying text requires knowledge of word representations in some vector space. You can train vector representations yourself. The challenge here is that you might not have enough data to train the embeddings. Furthermore, training will take a long time. In this case, you can use a pre-trained word embedding like GloVe to hasten your development process.

What is the difference between transfer learning and fine-tuning?

Transfer learning migrates the knowledge learned from the source dataset to the target dataset, to save training time and computational cost.

- Fine-tuning means freezing a few of the network layers that are used for feature extraction, and jointly training both the non-frozen layers and the newly added classifier layers of the pretrained model.

Fine-tuning is an optional step in transfer learning. Fine-tuning will usually improve the performance of the model. However, since you have to retrain the entire model, you'll likely overfit.



Overfitting is avoidable. Just retrain the model or part of it using a low learning rate. This is important because it prevents significant updates to the gradient. These updates result in poor

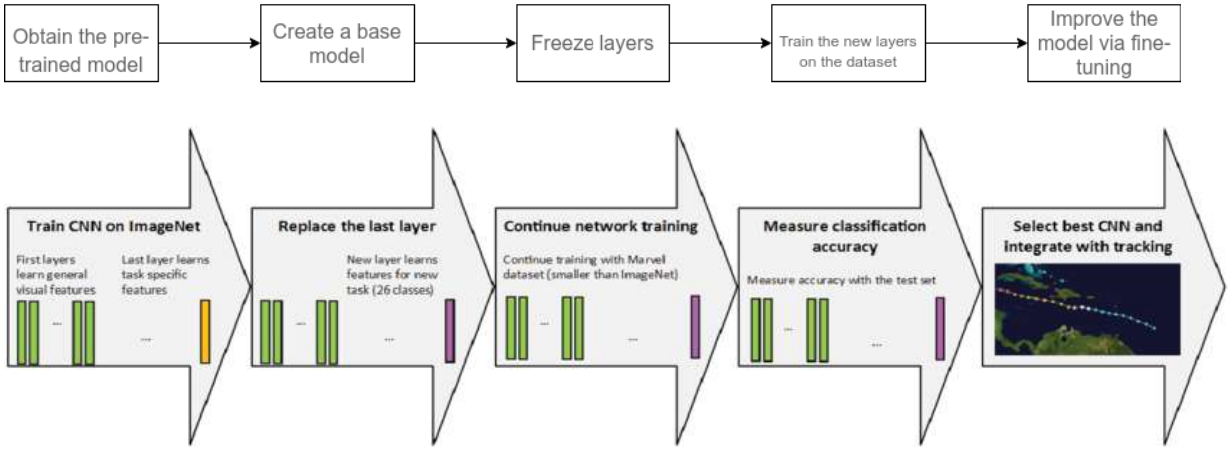
performance. Using a callback to stop the training process when the model has stopped improving is also helpful.

Why use transfer learning?

Assuming you have 100 images of cats and 100 dogs and want to build a model to classify the images. How would you train a model using this small dataset? You can train your model from scratch, but it will most likely overfit horribly. Enter transfer learning. Generally speaking, there are two big reasons why you want to use transfer learning:

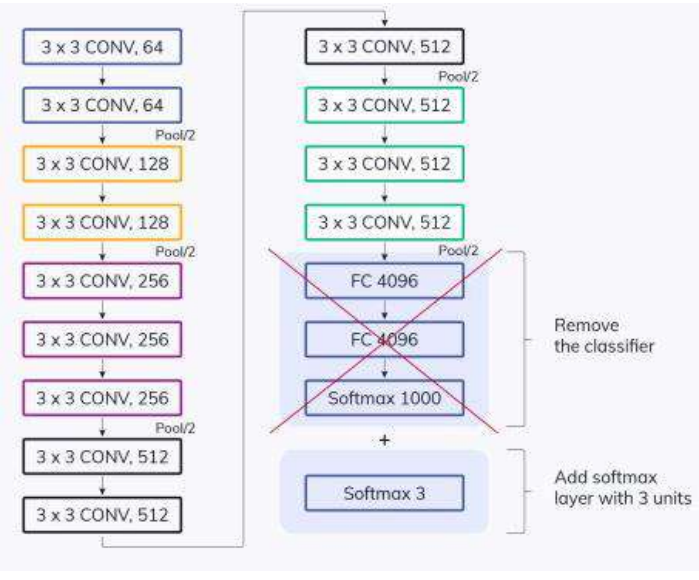
- training models with high accuracy requires a lot of data. For example, the ImageNet dataset contains over 1 million images. In the real world, you are unlikely to have such a large dataset.
- assuming that you had that kind of dataset, you might still not have the resources required to train a model on such a large dataset. Hence transfer learning makes a lot of sense if you don't have the compute resources needed to train models on huge datasets.
- even if you had the compute resources at your disposal, you still have to wait for days or weeks to train such a model. Therefore using a pre-trained model will save you precious time.

Implementation of Transfer Learning in these six general steps.



Step1 : Obtain the pre-trained model

The first step is to get the pre-trained model that you would like to use for your problem. The various sources of pre-trained models are covered in a separate section.



Step2: Create a base model

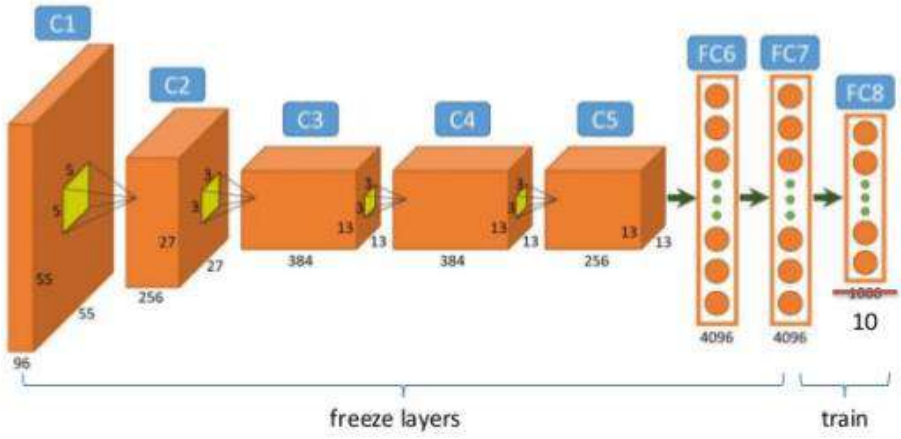
Usually, the first step is to instantiate the base model using one of the architectures such as ResNet or Xception. You can also optionally download the pre-trained weights. If you don't download the weights, you will have to use the architecture to train your model from

scratch. Recall that the base model will usually have more units in the final output layer than you require. When creating the base model, you, therefore, have to remove the final output layer. Later on, you will add a final output layer that is compatible with your problem.

Step3 : Freeze layers so they don't change during training

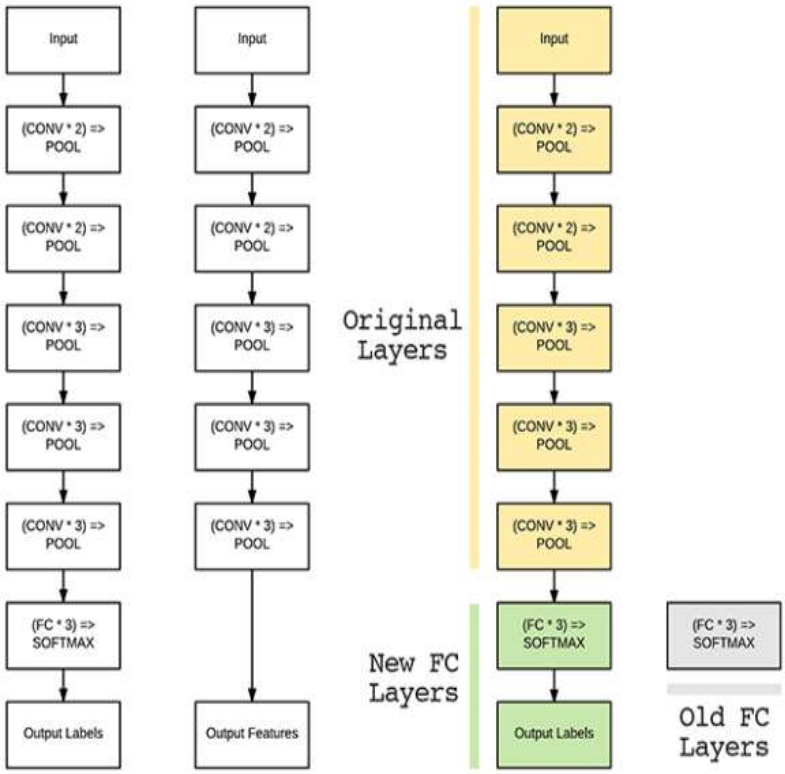
Freezing the layers from the pre-trained model is vital. This is because you don't want the weights in those layers to be re-initialized. If they are, then you will lose all the learning that has already taken place. This will be no different from training the model from scratch.

base_model.trainable = False



Step4 : Add new trainable layers

The next step is to add new trainable layers that will turn old features into predictions on the new dataset. This is important because the pre-trained model is loaded without the final output layer.



Step5: Train the new layers on the dataset

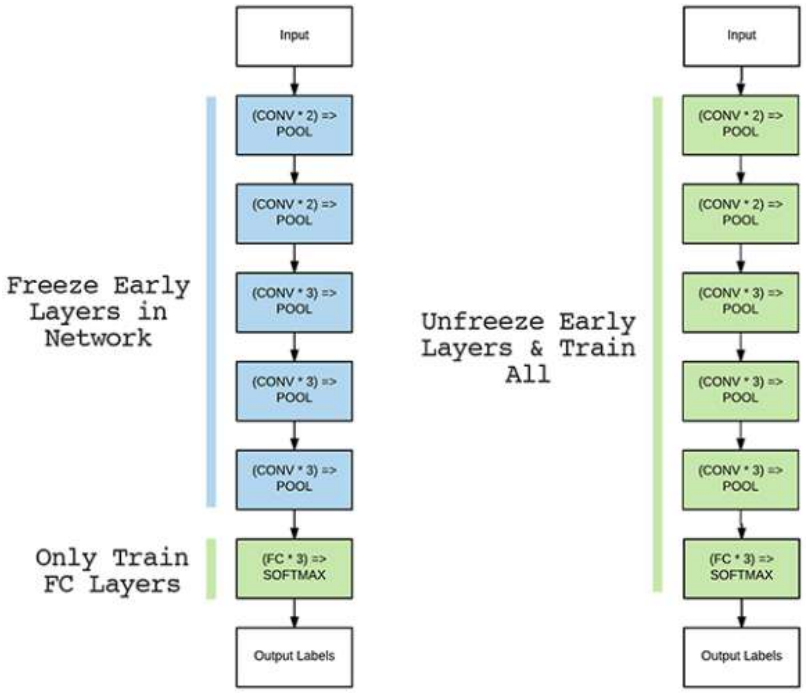
Remember that the pre-trained model's final output will most likely be different from the output that you want for your model. For example, pre-trained models trained on the ImageNet dataset will output 1000 classes. However, your model might just have two classes. In this case, you have to train the model with a new output layer in place.

Therefore, you will add some new dense layers as you please, but most importantly, a final dense layer with units corresponding to the number of outputs expected by your model.

Step6 : Improve the model via fine-tuning

Once you have done the previous step, you will have a model that can make predictions on your dataset. Optionally, you can improve its performance through fine-tuning. Fine-tuning is done by unfreezing the base model or part of it and training the entire model again on the whole dataset at a very low learning rate. The low learning rate will increase the performance of the model on the new dataset while preventing overfitting.

The learning rate has to be low because the model is quite large while the dataset is small. This is a recipe for overfitting, hence the low learning rate. Recompile the model once you have made these changes so that they can take effect. This is because the behavior of a model is frozen whenever you call the compile function. That means that you have to call the compile function again whenever you want to change the model's behavior. The next step will be to train the model again while monitoring it via callbacks to ensure it does not overfit.



10. A Pretrained Network as a Feature Extractor:

The process to use a pretrained model as a feature extractor is well established:

Step1 : Import the necessary libraries.

Step2 : Preprocess the data to make it ready for the neural network.

Step3: Load pretrained weights from the VGG16 network trained on a large dataset.

Step4 : Freeze all the weights in the convolutional layers (feature extraction part). Remember, the layers to freeze are adjusted depending on the similarity of the new task to the original dataset. In our case, we observed that ImageNet has a lot of dog and cat images, so the network has already been trained to extract the detailed features of our target object.

Step5 : Replace the fully connected layers of the network with a custom classifier. You can add as many fully connected layers as you see fit, and each can have as many hidden units as you want. For simple problems like this, we will just add one hidden layer with 64 units. You can observe the results and tune up if the model is underfitting or down if the model is overfitting. For the softmax layer, the number of units must be set equal to the number of classes (two units, in our case).

Step6 : Compile the network, and run the training process on the new data of cats and dogs to optimize the model for the smaller dataset.

Step7 : Evaluate the model.

Now, let's go through these steps and implement this project:

1. Import the necessary libraries:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.applications import imagenet_utils
from keras.applications import vgg16
from keras.applications import mobilenet
from keras.optimizers import Adam, SGD
from keras.metrics import categorical_crossentropy
from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.models import Model
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Preprocess the data to make it ready for the neural network. Keras has an ImageDataGenerator class that allows us to easily perform image augmentation**.**

In this example, we use ImageDataGenerator to generate our image tensors, but for simplicity, we will not implement image augmentation. The ImageDataGenerator class has a method called flow_from_directory() that is used to read images from folders containing images. This method expects your data directory to be structured as in figure 6.18**.** to use flow_ from_directory(). Now, load the data into train_path, valid_path, and test _path variables, and then generate the train, valid, and test batches:

```
train_path  = 'data/train'
valid_path  = 'data/valid'
test_path   = 'data/test'

train_batches = ImageDataGenerator().flow_from_directory(train_path,
                                                         target_size=(224,224),
                                                         batch_size=10)

valid_batches = ImageDataGenerator().flow_from_directory(valid_path,
                                                         target_size=(224,224),
                                                         batch_size=30)

test_batches = ImageDataGenerator().flow_from_directory(test_path,
                                                        target_size=(224,224),
                                                        batch_size=50,
                                                        shuffle=False)
```

ImageDataGenerator generates batches of tensor image data with real-time data augmentation. The data will be looped over (in batches). In this example, we won't be doing any image augmentation.

3. Load in pretrained weights from the VGG16 network trained on a large dataset. Similar to the examples in this chapter, we download the VGG16 network from Keras and download its weights after they are pretrained on the ImageNet dataset. Remember that we want to remove the classifier part from this network, so we set the parameter include_top=False:

```
base_model = vgg16.VGG16(weights = "imagenet", include_top=False,
                         input_shape = (224,224, 3))
```

4 Freeze all the weights in the convolutional layers (feature extraction part). We freeze the convolutional layers from the base_model created in the previous step and use that as a feature extractor, and then add a classifier on top of it in the next step:

```
for layer in base_model.layers:
    layer.trainable = False
```

Iterates through layers and locks them to make them non-trainable with this code

5 Add the new classifier, and build the new model. We add a few layers on top of the base model. In this example, we add one fully connected layer with 64 hidden units and a softmax with 2 hidden units. We also add batch norm and dropout layers to avoid overfitting:

Uses the get_layer method to save the last layer of the network. Then saves the output of the last layer to be the input of the next layer.

```
last_layer = base_model.get_layer('block5_pool')
last_output = last_layer.output

x = Flatten()(last_output)
```

Flattens the classifier input, which is output of the last layer of the VGG16 model

```
x = Dense(64, activation='relu', name='FC_2')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(2, activation='softmax', name='softmax')(x)
```

Adds one fully connected layer that has 64 units and batchnorm, dropout, and softmax layers

Instantiates a new_model using Keras's Model class

```
new_model = Model(inputs=base_model.input, outputs=x)
new_model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0

```
block4_conv1 (Conv2D)          (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)          (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)          (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)     (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)          (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)     (None, 7, 7, 512)         0
_________________________________________________________________
flatten_1 (Flatten)            (None, 25088)             0
_________________________________________________________________
FC_2 (Dense)                   (None, 64)                1605696
_________________________________________________________________
batch_normalization_1 (Batch   (None, 64)                256
_________________________________________________________________
dropout_1 (Dropout)            (None, 64)                0
_________________________________________________________________
softmax (Dense)                (None, 2)                 130
=================================================================
Total params: 16,320,770
Trainable params: 1,605,954
Non-trainable params: 14,714,816
_________________________________________________________________
```

6 Compile the model and run the training process:

```
new_model.compile(Adam(lr=0.0001), loss='categorical_crossentropy',
                  metrics=['accuracy'])

new_model.fit_generator(train_batches, steps_per_epoch=4,
                        validation_data=valid_batches, validation_steps=2,
                        epochs=20, verbose=2)
```

When you run the previous code snippet, the verbose training is printed after each epoch as follows:

```
Epoch 1/20
 - 28s - loss: 1.0070 - acc: 0.6083 - val_loss: 0.5944 - val_acc: 0.6833
Epoch 2/20
 - 25s - loss: 0.4728 - acc: 0.7754 - val_loss: 0.3313 - val_acc: 0.8605
Epoch 3/20
 - 30s - loss: 0.1177 - acc: 0.9750 - val_loss: 0.2449 - val_acc: 0.8167
Epoch 4/20
 - 25s - loss: 0.1640 - acc: 0.9444 - val_loss: 0.3354 - val_acc: 0.8372
Epoch 5/20
 - 29s - loss: 0.0545 - acc: 1.0000 - val_loss: 0.2392 - val_acc: 0.8333
Epoch 6/20
```

```
 - 25s - loss: 0.0941 - acc: 0.9505 - val_loss: 0.2019 - val_acc: 0.9070
Epoch 7/20
 - 28s - loss: 0.0269 - acc: 1.0000 - val_loss: 0.1707 - val_acc: 0.9000
Epoch 8/20
 - 26s - loss: 0.0349 - acc: 0.9917 - val_loss: 0.2489 - val_acc: 0.8140
Epoch 9/20
 - 28s - loss: 0.0435 - acc: 0.9891 - val_loss: 0.1634 - val_acc: 0.9000
Epoch 10/20
 - 26s - loss: 0.0349 - acc: 0.9833 - val_loss: 0.2375 - val_acc: 0.8140
Epoch 11/20
 - 28s - loss: 0.0288 - acc: 1.0000 - val_loss: 0.1859 - val_acc: 0.9000
Epoch 12/20
 - 29s - loss: 0.0234 - acc: 0.9917 - val_loss: 0.1879 - val_acc: 0.8372
Epoch 13/20
 - 32s - loss: 0.0241 - acc: 1.0000 - val_loss: 0.2513 - val_acc: 0.8500
Epoch 14/20
 - 29s - loss: 0.0120 - acc: 1.0000 - val_loss: 0.0900 - val_acc: 0.9302
Epoch 15/20
 - 36s - loss: 0.0189 - acc: 1.0000 - val_loss: 0.1888 - val_acc: 0.9000
Epoch 16/20
 - 30s - loss: 0.0142 - acc: 1.0000 - val_loss: 0.1672 - val_acc: 0.8605
Epoch 17/20
 - 29s - loss: 0.0160 - acc: 0.9917 - val_loss: 0.1752 - val_acc: 0.8667
Epoch 18/20
 - 25s - loss: 0.0126 - acc: 1.0000 - val_loss: 0.1823 - val_acc: 0.9070
Epoch 19/20
 - 29s - loss: 0.0165 - acc: 1.0000 - val_loss: 0.1789 - val_acc: 0.8833
Epoch 20/20
 - 25s - loss: 0.0112 - acc: 1.0000 - val_loss: 0.1743 - val_acc: 0.8837
```

the model was trained very quickly using regular CPU computing power. Each epoch took approximately 25 to 29 seconds, which means the model took less than 10 minutes to train for 20 epochs.

7 Evaluate the model. First, let's define the load_dataset() method that we will use to convert our dataset into tensors:

```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np

def load_dataset(path):
    data = load_files(path)
    paths = np.array(data['filenames'])
    targets = np_utils.to_categorical(np.array(data['target']))
    return paths, targets

test_files, test_targets = load_dataset('small_data/test')
```

Then, we create test_tensors to evaluate the model on them:

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
from tqdm import tqdm
```

```
def path_to_tensor(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in
tqdm(img_paths)]
    return np.vstack(list_of_tensors)

test_tensors = preprocess_input(paths_to_tensor(test_files))
```

Converts the PIL.Image.Image type to a 3D tensor with shape (224, 224, 3)

Converts the 3D tensor to a 4D tensor with shape (1, 224, 224, 3) and returns the 4D tensor

Loads an RGB image as PIL.Image.Image type

Now we can run Keras's `evaluate()` method to calculate the model accuracy:

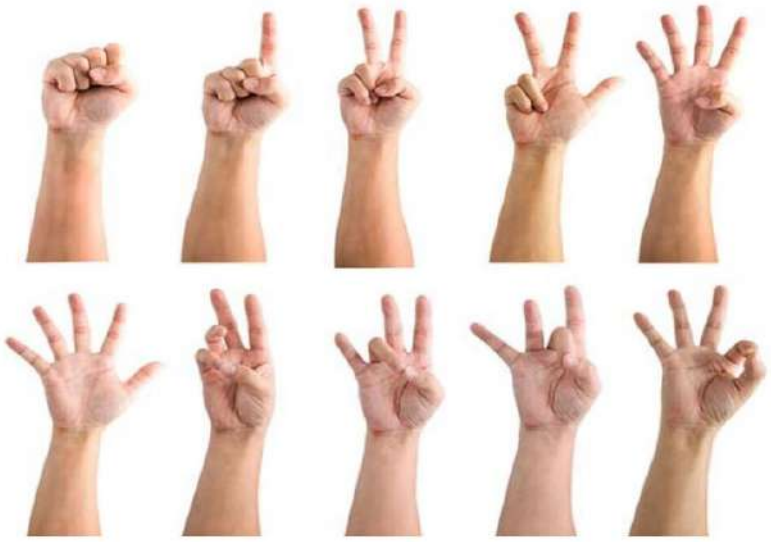
```
print('\nTesting loss: {:.4f}\nTesting accuracy:
  {:.4f}'.format(*new_model.evaluate(test_tensors, test_targets)))

Testing loss: 0.1042
Testing accuracy: 0.9579
```

The model has achieved an accuracy of 95.79% in less than 10 minutes of training.This is very good, given our very small dataset.

3. Sign Language Classifier with Fine-tuning:

To build a sign language classifier that distinguishes 10 classes: the Sign language digits from 0 to 9. Figure 6.19 shows a sample of our dataset.



Dataset:

- Number of classes = 10 (digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9)
- Image size = 100 × 100
- Color space = RGB ϒ 1,712 images in the training set
- 300 images in the validation set
- 50 images in the test set

It is very noticeable how small our dataset is. If you try to train a network from scratch on this very small dataset, you will not achieve good results. On the other hand, we were able to achieve an accuracy higher than 98% by using transfer learning, even though the source and target domains were very different.

Implementation

The VGG16 network trained on the ImageNet dataset. The process to fine-tune a pretrained network is as follows:

1 Import the necessary libraries.

2 Preprocess the data to make it ready for the neural network.

3 Load in pretrained weights from the VGG16 network trained on a large dataset (ImageNet).

4 Freeze part of the feature extractor part.

5 Add the new classifier layers.

6 Compile the network, and run the training process to optimize the model for the smaller dataset.

7 Evaluate the model.

Code: 1 Import the necessary libraries:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.applications import imagenet_utils
from keras.applications import vgg16
from keras.optimizers import Adam, SGD
from keras.metrics import categorical_crossentropy

from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.models import Model
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
%matplotlib inline
```

2 Preprocess the data to make it ready for the neural network. The ImageDataGenerator class from Keras and the flow_from_ directory() method to preprocess our data. The data is already structured for you to directly create your tensors:

```
train_path  = 'dataset/train'
valid_path  = 'dataset/valid'
test_path  = 'dataset/test'

train_batches = ImageDataGenerator().flow_from_directory(train_path,
                                                         target_size=(224,224),
                                                         batch_size=10)

valid_batches = ImageDataGenerator().flow_from_directory(valid_path,
                                                         target_size=(224,224),
                                                         batch_size=30)

test_batches = ImageDataGenerator().flow_from_directory(test_path,
                                                        target_size=(224,224),
                                                        batch_size=50,
                                                        shuffle=False)
```

ImageDataGenerator generates batches of tensor image data with real-time data augmentation. The data will be looped over (in batches). In this example, we won't be doing any image augmentation.

```
Found 1712 images belonging to 10 classes.
Found 300 images belonging to 10 classes.
Found 50 images belonging to 10 classes.
```

3 Load in pretrained weights from the VGG16 network trained on a large dataset (imagenet). We download the VGG16 architecture from the Keras library with imagenet weights. Note

that we use the parameter pooling='avg' here: this basically means global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. We use this as an alternative to the Flatten layer before adding the fully connected layers:

```
base_model = vgg16.VGG16(weights = "imagenet", include_top=False,
                         input_shape = (224,224, 3), pooling='avg')
```

4 Freeze some of the feature extractor part, and fine-tune the rest on our new training data. The level of fine-tuning is usually determined by trial and error. VGG16 has 13 convolutional layers: you can freeze them all or freeze a few of them, depending on how similar your data is to the source data. In the sign language case, the new domain is very different from our domain, so we will start with fine-tuning only the last five layers; if we don't get satisfying results, we can fine-tune more. It turns out that after we trained the new model, we got 98% accuracy, so this was a good level of fine-tuning. network doesn't converge, try fine-tunin more layers.

```
for layer in base_model.layers[:-5]:
    layer.trainable = False

base_model.summary()
```

Iterates through layers and locks them, except for the last five layers

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 512)               0
=================================================================
```

5 Add the new classifier layers, and build the new model:

Adds our new softmax layer with 10 hidden units

Saves the output of base_model to be the input of the next layer

Instantiates a new_model using Keras's Model class

Prints the new_model summary

```
last_output = base_model.output
x = Dense(10, activation='softmax', name='softmax')(last_output)
new_model = Model(inputs=base_model.input, outputs=x)
new_model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
softmax (Dense)              (None, 10)                5130
=================================================================
Total params: 14,719,818
Trainable params: 7,084,554
Non-trainable params: 7,635,264
```

6 Compile the network, and run the training process to optimize the model for the smaller dataset:

```
new_model.compile(Adam(lr=0.0001), loss='categorical_crossentropy',
                  metrics=['accuracy'])

from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='signlanguage.model.hdf5',
                               save_best_only=True)

history = new_model.fit_generator(train_batches, steps_per_epoch=18,
                    validation_data=valid_batches, validation_steps=3,
                    epochs=20, verbose=1, callbacks=[checkpointer])
```

```
Epoch 1/150
18/18 [==============================] - 40s 2s/step - loss: 3.2263 - acc:
   0.1833 - val_loss: 2.0674 - val_acc: 0.1667
Epoch 2/150
18/18 [==============================] - 41s 2s/step - loss: 2.0311 - acc:
   0.1833 - val_loss: 1.7330 - val_acc: 0.3000
Epoch 3/150
18/18 [==============================] - 42s 2s/step - loss: 1.5741 - acc:
   0.4500 - val_loss: 1.5577 - val_acc: 0.4000
Epoch 4/150
18/18 [==============================] - 42s 2s/step - loss: 1.3068 - acc:
   0.5111 - val_loss: 0.9856 - val_acc: 0.7333
Epoch 5/150
18/18 [==============================] - 43s 2s/step - loss: 1.1563 - acc:
   0.6389 - val_loss: 0.7637 - val_acc: 0.7333
Epoch 6/150
18/18 [==============================] - 41s 2s/step - loss: 0.8414 - acc:
   0.6722 - val_loss: 0.7550 - val_acc: 0.8000
Epoch 7/150
18/18 [==============================] - 41s 2s/step - loss: 0.5982 - acc:
   0.8444 - val_loss: 0.7910 - val_acc: 0.6667
Epoch 8/150
18/18 [==============================] - 41s 2s/step - loss: 0.3804 - acc:
   0.8722 - val_loss: 0.7376 - val_acc: 0.8667
Epoch 9/150
18/18 [==============================] - 41s 2s/step - loss: 0.5048 - acc:
   0.8222 - val_loss: 0.2677 - val_acc: 0.9000


Epoch 10/150
18/18 [==============================] - 39s 2s/step - loss: 0.2383 - acc:
   0.9276 - val_loss: 0.2844 - val_acc: 0.9000
Epoch 11/150
18/18 [==============================] - 41s 2s/step - loss: 0.1163 - acc:
   0.9778 - val_loss: 0.0775 - val_acc: 1.0000
Epoch 12/150
18/18 [==============================] - 41s 2s/step - loss: 0.1377 - acc:
   0.9667 - val_loss: 0.5140 - val_acc: 0.9333
Epoch 13/150
18/18 [==============================] - 41s 2s/step - loss: 0.0955 - acc:
   0.9556 - val_loss: 0.1783 - val_acc: 0.9333
Epoch 14/150
18/18 [==============================] - 41s 2s/step - loss: 0.1785 - acc:
   0.9611 - val_loss: 0.0704 - val_acc: 0.9333
Epoch 15/150
18/18 [==============================] - 41s 2s/step - loss: 0.0533 - acc:
   0.9778 - val_loss: 0.4692 - val_acc: 0.8667
Epoch 16/150
18/18 [==============================] - 41s 2s/step - loss: 0.0809 - acc:
   0.9778 - val_loss: 0.0447 - val_acc: 1.0000
Epoch 17/150
18/18 [==============================] - 41s 2s/step - loss: 0.0834 - acc:
   0.9722 - val_loss: 0.0284 - val_acc: 1.0000
Epoch 18/150
18/18 [==============================] - 41s 2s/step - loss: 0.1022 - acc:
   0.9611 - val_loss: 0.0177 - val_acc: 1.0000
Epoch 19/150
18/18 [==============================] - 41s 2s/step - loss: 0.1134 - acc:
   0.9667 - val_loss: 0.0595 - val_acc: 1.0000
Epoch 20/150
18/18 [==============================] - 39s 2s/step - loss: 0.0676 - acc:
   0.9777 - val_loss: 0.0862 - val_acc: 0.9667
```

Notice the training time of each epoch from the verbose output. The model was trained very quickly using regular CPU computing power. Each epoch took approximately 40 seconds, which means it took the model less than 15 minutes to train for 20 epochs.

7 Evaluate the accuracy of the model. Similar to the previous project, we create a load_dataset() method to create test_targets and test_tensors and then use the evaluate() method from Keras to run inferences on the test images and get the model accuracy:

```
print('\nTesting loss: {:.4f}\nTesting accuracy:
   {:.4f}'.format(*new_model.evaluate(test_tensors, test_targets)))
```

```
Testing loss: 0.0574
Testing accuracy: 0.9800
```

A deeper level of evaluating your model involves creating a confusion matrix Which describe the performance of a classification model, to provide a deeper understanding of how the model performed on the test dataset.

Now, let's build the confusion matrix for the sign language classifier:

```
from sklearn.metrics import confusion_matrix
import numpy as np

cm_labels = ['0','1','2','3','4','5','6','7','8','9']

cm = confusion_matrix(np.argmax(test_targets, axis=1),
                      np.argmax(new_model.predict(test_tensors), axis=1))
plt.imshow(cm, cmap=plt.cm.Blues)
plt.colorbar()
indexes = np.arange(len(cm_labels))
for i in indexes:
    for j in indexes:
        plt.text(j, i, cm[i, j])
plt.xticks(indexes, cm_labels, rotation=90)
plt.xlabel('Predicted label')
plt.yticks(indexes, cm_labels)
plt.ylabel('True label')
plt.title('Confusion matrix')
plt.show()
```

The model successfully made the correct predictions for all the test images except the image with true label = 8. In that case, the model mistakenly classified an image of number 8 as number = 7.