**Advanced CNN architectures**

We will start with LeNet, developed in 1998, which performed fairly well at recognizing handwritten characters. You will see how CNN architectures have evolved since then to deeper CNNs like AlexNet and VGGNet, and beyond to more advanced and super-deep networks like Inception and ResNet, developed in 2014 and 2015, respectively.
For each CNN architecture, you will learn the following:

- *Novel features*—We will explore the novel features that distinguish these networks from others and what specific problems their creators were trying to solve.
- *Network architecture*—We will cover the architecture and components of each network and see how they come together to form the end-to-end network.
- *Network code implementation*—We will walk step-by-step through the network implementations using the Keras deep learning (DL) library. The goal of this section is for you to learn how to read research papers and implement new architectures as the need arises.
- *Setting up learning hyperparameters*—After you implement a network architecture, you need to set up the hyperparameters of the learning algorithms that you learned in chapter 4 (optimizer, learning rate, weight decay, and so on). We will implement the learning hyperparameters as presented in the original research paper of each network. In this section, you will see how performance evolved from one network to another over the years.
- *Network performance*—Finally, you will see how each network performed on benchmark datasets like MNIST and ImageNet, as represented in their research papers.

1. **CNN design patterns:**

- ***Pattern 1: Feature extraction and classification***—Convolutional nets 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 below). This is pretty much always the case with ConvNets, starting from LeNet and AlexNet to the very recent CNNs that have come out in the past few years, like Inception and ResNet.

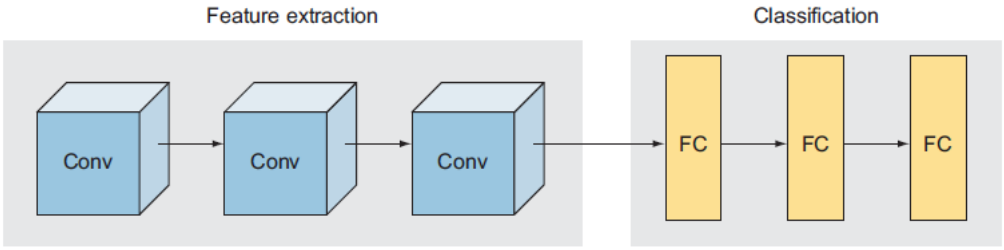



Figure 5.1 Convolutional nets generally include feature extraction and classification.

- ***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 below, 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.

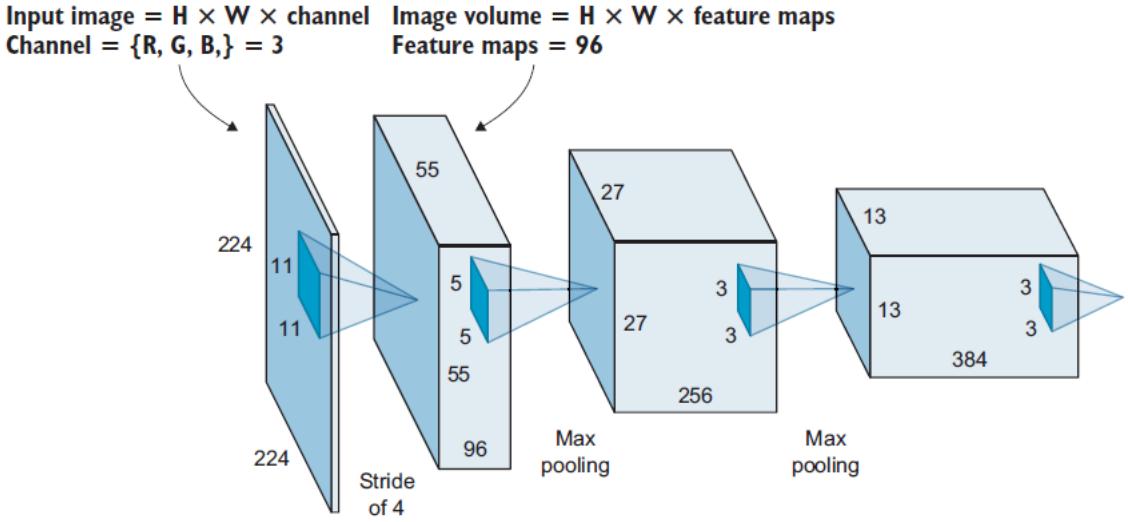
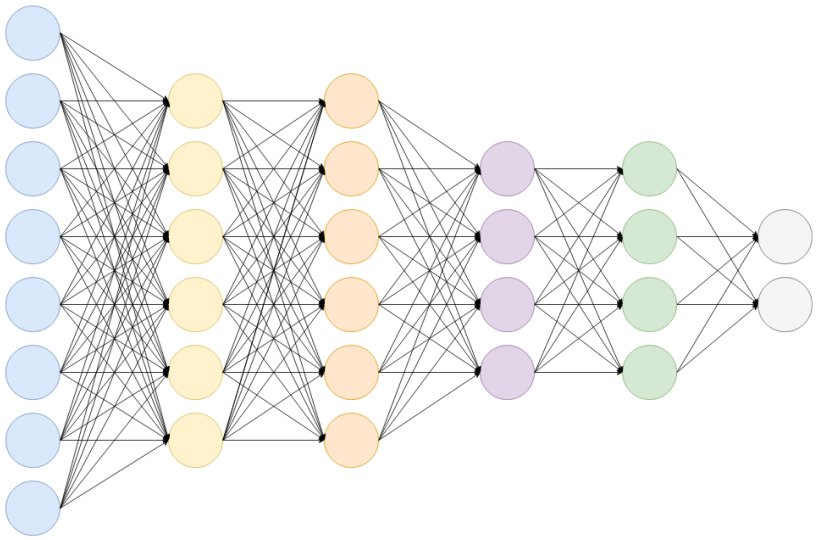



Figure 5.2 Image depth increases, and the dimensions decrease.

- ***Pattern 3: Fully connected layers***—Typically, all fully connected layers in a network either have the same number of hidden units or decrease at each layer. 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.



***i). LeNet-5:***

In 1998, Lecun et al. introduced a pioneering CNN called *LeNet-5*. The architecture is composed of five weight layers, and hence the name LeNet-5: three convolutional layers and two fully connected layers. We refer to the convolutional and fully connected layers as *weight layers* because they contain trainable weights as opposed to pooling layers that don't contain any weights.

- ***LeNet architecture:***

The architecture of LeNet-5 is shown in figure below:

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.

Notice that Yann LeCun and his team used tanh as an activation function instead of the currently state-of-the-art ReLU. This is because in 1998, ReLU had not yet been used in the context of DL, and it was more common to use tanh or sigmoid as an activation function in the hidden layers.

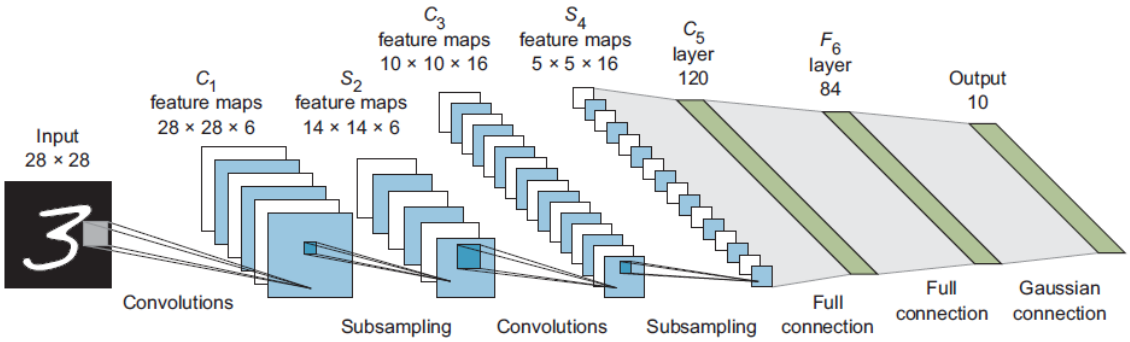



Figure 5.3 LeNet architecture

***LeNet-5 implementation in Keras***

Here are the main takeaways for building the LeNet-5 network:

- *Number of filters in each convolutional layer*—As you can see in figure 5.3, 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 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.
- *Activation function*—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 below).

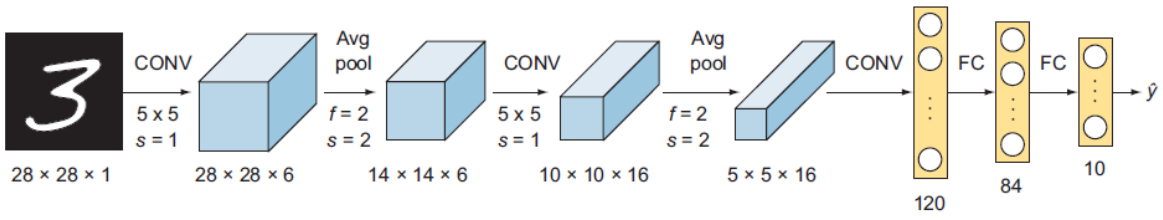



Figure 5.4 The LeNet architecture consists of convolutional kernels of size 5 × 5; pooling layers; an activation function (tanh); and three fully connected layers with 120, 84, and 10 neurons, respectively.

In LeNet-5, the **tanh (hyperbolic tangent)** activation function is used for the following reasons:

**1. Non-linearity:**

- Like other activation functions, tanh introduces non-linearity to the network. This is important because convolution and pooling are linear operations, and without non-linearity, the network would behave like a linear model, which limits its ability to model complex patterns in data. tanh ensures that the network can learn and approximate complex functions.

**2. Zero-centered output:**

- Unlike the sigmoid function, which outputs values in the range [0,1], the tanh function outputs values in the range [−1,1]. This means that the output of neurons can be both negative and positive. Having zero-centered activations helps to balance the network's weight updates during backpropagation, allowing faster convergence and better optimization, as the gradients tend to oscillate less compared to sigmoid activations.

Now let's put that in code to build the LeNet-5 architecture:

```
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 networks.

| 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 5.5 LeNet-5 model summary**

**LeNet-5** architecture, which is a convolutional neural network (CNN) commonly used for image classification tasks such as handwritten digit recognition (e.g., MNIST dataset). Here's a breakdown of each layer:

1. **conv2d_1 (Conv2D)**:
   - Type: 2D Convolutional layer.
   - Output Shape: (None, 28, 28, 6) – The output is a 28x28 feature map with 6 filters.
   - Param #: 156 – The number of learnable parameters. It comes from (filter width × filter height × input channels + bias) × number of filters = (5×5×1+1)×6 = 156.
2. **average_pooling2d_1 (AveragePooling2D)**:
   - Type: Average Pooling layer.
   - Output Shape: (None, 14, 14, 6) – The spatial size is reduced to 14x14 after pooling.
   - Param #: 0 – No learnable parameters, as pooling layers only reduce the spatial dimensions.
3. **conv2d_2 (Conv2D)**:
   - Type: 2D Convolutional layer.
   - Output Shape: (None, 10, 10, 16) – Produces a 10x10 feature map with 16 filters.
   - Param #: 2416 – Calculated as (5×5×6+1)×16 = 2416.
4. **average_pooling2d_2 (AveragePooling2D)**:
   - Type: Average Pooling layer.
   - Output Shape: (None, 5, 5, 16) – Further reduces the spatial size to 5x5.
   - Param #: 0 – No learnable parameters, as it's a pooling operation.
5. **conv2d_3 (Conv2D)**:
   - Type: 2D Convolutional layer.
   - Output Shape: (None, 1, 1, 120) – Produces a single (1x1) feature map with 120 filters.

- Param #: 48120 – Calculated as (5×5×16+1)×120 = 48120.

6. **flatten_1 (Flatten)**:
   - Type: Flatten layer.
   - Output Shape: (None, 120) – Flattens the previous 1x1x120 output into a single 120-dimensional vector.
   - Param #: 0 – No learnable parameters, as it's a reshaping operation.
7. **dense_1 (Dense)**:
   - Type: Fully connected (dense) layer.
   - Output Shape: (None, 84) – Reduces the 120 input units to 84 units.
   - Param #: 10164 – Calculated as (120×84 + 84) = 10164.
8. **dense_2 (Dense)**:
   - Type: Fully connected (dense) layer.
   - Output Shape: (None, 10) – The output layer with 10 units, typically corresponding to 10 classes.
   - Param #: 850 – Calculated as (84×10 + 10) = 850.

**Summary:**

- **Total params**: 61,706 – The total number of learnable parameters in the model.
- **Trainable params**: 61,706 – All parameters are trainable.
- **Non-trainable params**: 0 – No non-trainable parameters.

***Setting up the learning hyperparameters:***

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. In the paper, 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).

### 1. ***AlexNet:***

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 Alex- Net was to build a deeper network that can learn more complex functions. AlexNet (figure 5.6) was the winner of the ILSVRC image classification competition in 2012. Krizhevsky et al. created the neural network architecture and trained it on 1.2 million high-resolution images into 1,000 different classes of the ImageNet dataset. AlexNet was state of the art at its time because it was the first real "deep" network that opened the door for the CV community to seriously consider convolutional networks in their applications. We will explain deeper networks later in this chapter, like VGGNet and ResNet, but it is good to see how ConvNets evolved and the main drawbacks of AlexNet that were the main motivation for the later networks.

As you can see in figure 5.6, 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. This allowed AlexNet to achieve remarkable performance in the ILSVRC image classification competition in 2012.

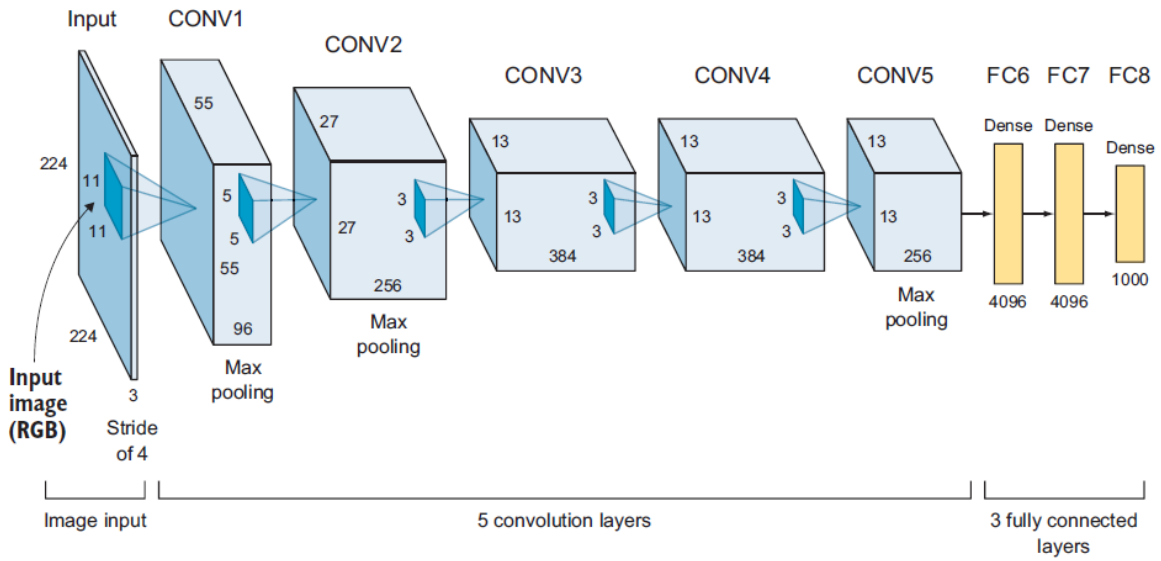



Figure 5.6 AlexNet architecture

***AlexNet architecture:***

The architecture is pretty straightforward. It 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

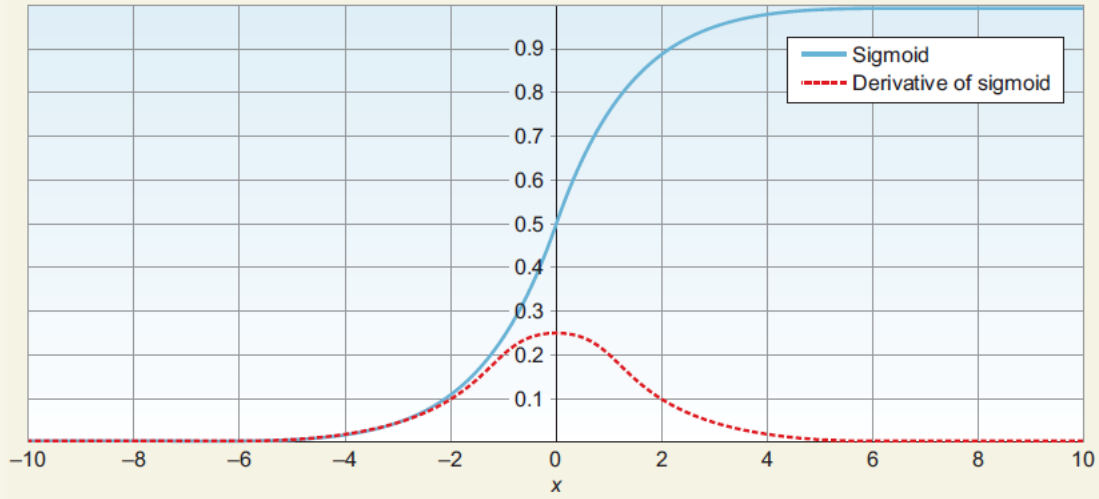
***Novel features of AlexNet:***

1. RELU ACTIVATION FUNCTION:

AlexNet uses ReLu for the nonlinear part instead of the tanh and sigmoid functions that were the earlier standard for traditional neural networks (like LeNet). ReLu was used in the hidden layers of the AlexNet architecture because it trains much faster. This is because the derivative of the sigmoid function becomes very small in the saturating region, and therefore the updates applied to the weights almost vanish. This phenomenon is called the *vanishing gradient problem*. ReLU is represented by this equation:

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

The vanishing gradient problem Certain activation functions, like the sigmoid function, squish a large input space into a small input space between 0 and 1 (–1 to 1 for tanh activations). Therefore, a large change in the input of the sigmoid function causes a small change in the output. As a result, the derivative becomes very small:




**The vanishing gradient problem: a large change in the input of the sigmoid function causes a negligible change in the output.**

2. DROPOUT LAYER

AlexNet also addresses the over-fitting problem by using drop-out layers where a connection is dropped during training with a probability of p=0.5. Although this avoids the network from over-fitting by helping it escape from bad local minima, the number of iterations required for convergence is doubled too.

Dropout layers are used to prevent the neural network from overfitting. The neurons that are "dropped out" do not contribute to the forward pass and do not participate in backpropagation. This means every time an input is presented, the neural network samples a different architecture, but all of these architectures share the same weights. This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. Therefore, the neuron is forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Krizhevsky et al. used dropout with a probability of 0.5 in the two fully connected layers.

3. DATA AUGMENTATION

One popular and very effective approach to avoid overfitting is to artificially enlarge the dataset using label-preserving transformations. This happens by generating new instances of the training images with transformations like image rotation, flipping, scaling, and many more.

4. LOCAL RESPONSE NORMALIZATION

AlexNet uses local response normalization. It is different from the batch normalization technique. Normalization helps to speed up convergence. Nowadays, batch normalization is used instead of

local response normalization.

5. WEIGHT REGULARIZATION

A network with large network weights can be a sign of an unstable network where small changes in the input can lead to large changes in the output. This can be a sign that the network has overfit the training dataset and will likely perform poorly when making predictions on new data.
A solution to this problem is to update the learning algorithm to encourage the network to keep the weights small. This is called weight regularization and it can be used as a general technique to reduce overfitting of the training dataset and improve the generalization of the model.

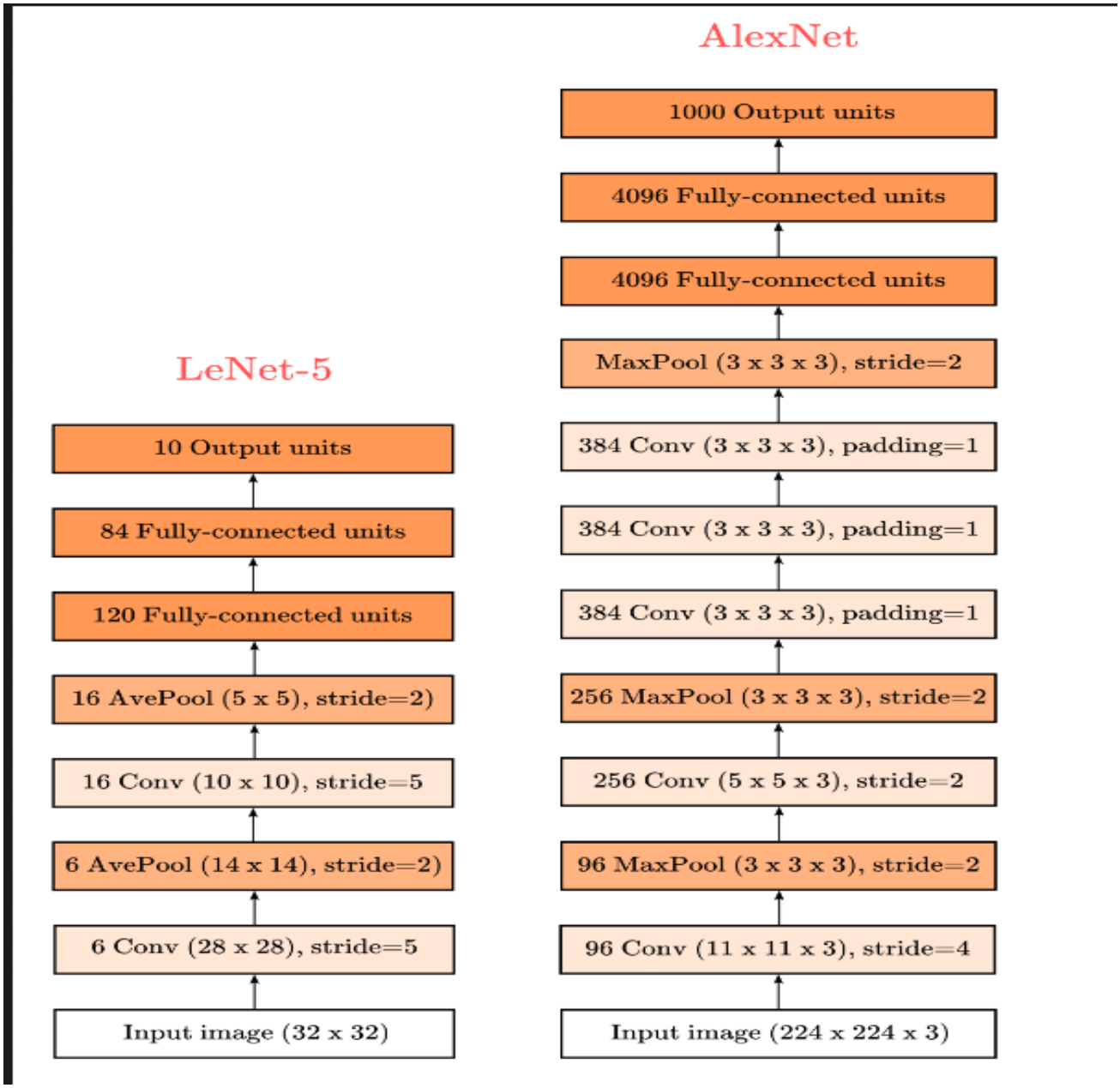
Krizhevsky et al. used a weight decay of 0.0005. Weight decay is another term for the L2 regularization. This approach reduces the overfitting of the DL neural network model on training data to allow the network to generalize better on new data:

```
model.add(Conv2D(32, (3,3), kernel_regularizer=l2(λ)))
```

The lambda (λ) value is a weight decay hyperparameter that you can tune. If you still see overfitting, you can reduce it by increasing the lambda value. In this case, Krizhevsky and his team found that a small decay value of 0.0005 was good enough for the model to learn.

6. TRAINING ON MULTIPLE GPUS

Krizhevsky et al. used a GTX 580 GPU with only 3 GB of memory. It was state-of-the-art at the time but not large enough to train the 1.2 million training examples in the dataset.




***2. VGGNet:***

VGGNet was developed in 2014 by the Visual Geometry Group at Oxford University (hence the name VGG).3 The building components are exactly the same as those in LeNet and AlexNet, except that VGGNet is an even deeper network with more convolutional, pooling, and dense layers. Other than that, no new components are introduced here. VGGNet, also known as VGG16, consists of 16 weight layers: 13 convolutional layers and 3 fully connected layers. Its uniform architecture makes it appealing in the DL community because it is very easy to understand.

The major shortcoming of too many hyper-parameters of AlexNet was solved by VGG Net by replacing large kernel-sized filters (11 and 5 in the first and second convolution layer, respectively)

with multiple 3×3 kernel-sized filters one after another. The architecture developed by Simonyan and Zisserman was the 1st runner up of the Visual Recognition Challenge of 2014. The architecture consist of 3*3 Convolutional filters, 2*2 Max Pooling layer with a stride of 1, keeping the padding same to preserve the dimension. In total, there are 16 layers in the network where the input image is RGB format with dimension of 224*224*3, followed by 5 pairs of Convolution(filters: 64, 128, 256,512,512) and Max Pooling. The output of these layers is fed into three fully connected layers and a softmax function in the output layer. In total there are 138 Million parameters in VGG Net.

***Novel features of VGGNet:***

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.

Simonyan and Zisserman decided to use a smaller 3 × 3 kernel to allow the network to extract finer-level features of the image compared to AlexNet's large kernels (11 × 11 and 5 × 5). The idea is that with a given convolutional receptive field, multiple stacked smaller kernels is better than a larger kernel because having multiple nonlinear layers increases the depth of the network; this enables it to learn more complex features at a lower cost because it has fewer learning parameters.

This unified configuration of the convolutional and pooling components simplifies the neural network architecture, which makes it very easy to understand and implement. The VGGNet architecture is developed by stacking 3 × 3 convolutional layers with 2 × 2 pooling layers inserted after several convolutional layers. This is followed by the traditional classifier, which is composed of fully connected layers and a softmax, as depicted in figure 5.8.

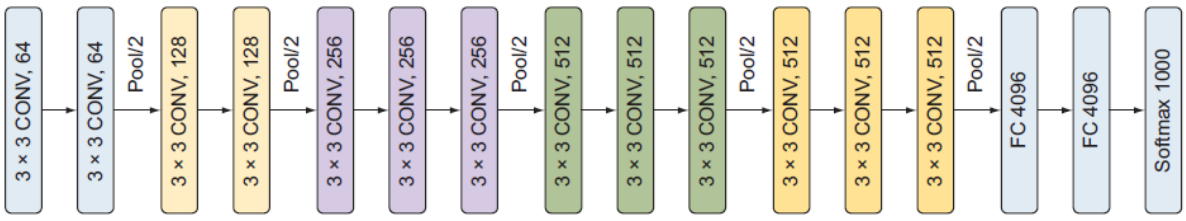



Figure 5.8 VGGNet-16 architecture

***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.

| 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<br>LRN | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>conv3-256<br>conv3-256 |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512<br>conv3-512 |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>conv3-512<br>conv3-512 |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |

**Figure 5.9 VGGNet architecture configurations**

| VGG16 - Structural Details | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Input Image | | | output | | | Layer | Stride | Kernel | | in | out | Param |
| 1 | 224 | 224 | 3 | 224 | 224 | 64 | conv3-64 | 1 | 3 | 3 | 3 | 64 | 1792 |
| 2 | 224 | 224 | 64 | 224 | 224 | 64 | conv3064 | 1 | 3 | 3 | 64 | 64 | 36928 |
| | 224 | 224 | 64 | 112 | 112 | 64 | maxpool | 2 | 2 | 2 | 64 | 64 | 0 |
| 3 | 112 | 112 | 64 | 112 | 112 | 128 | conv3-128 | 1 | 3 | 3 | 64 | 128 | 73856 |
| 4 | 112 | 112 | 128 | 112 | 112 | 128 | conv3-128 | 1 | 3 | 3 | 128 | 128 | 147584 |
| | 112 | 112 | 128 | 56 | 56 | 128 | maxpool | 2 | 2 | 2 | 128 | 128 | 65664 |
| 5 | 56 | 56 | 128 | 56 | 56 | 256 | conv3-256 | 1 | 3 | 3 | 128 | 256 | 295168 |
| 6 | 56 | 56 | 256 | 56 | 56 | 256 | conv3-256 | 1 | 3 | 3 | 256 | 256 | 590080 |
| 7 | 56 | 56 | 256 | 56 | 56 | 256 | conv3-256 | 1 | 3 | 3 | 256 | 256 | 590080 |
| | 56 | 56 | 256 | 28 | 28 | 256 | maxpool | 2 | 2 | 2 | 256 | 256 | 0 |
| 8 | 28 | 28 | 256 | 28 | 28 | 512 | conv3-512 | 1 | 3 | 3 | 256 | 512 | 1180160 |
| 9 | 28 | 28 | 512 | 28 | 28 | 512 | conv3-512 | 1 | 3 | 3 | 512 | 512 | 2359808 |
| 10 | 28 | 28 | 512 | 28 | 28 | 512 | conv3-512 | 1 | 3 | 3 | 512 | 512 | 2359808 |
| | 28 | 28 | 512 | 14 | 14 | 512 | maxpool | 2 | 2 | 2 | 512 | 512 | 0 |
| 11 | 14 | 14 | 512 | 14 | 14 | 512 | conv3-512 | 1 | 3 | 3 | 512 | 512 | 2359808 |
| 12 | 14 | 14 | 512 | 14 | 14 | 512 | conv3-512 | 1 | 3 | 3 | 512 | 512 | 2359808 |
| 13 | 14 | 14 | 512 | 14 | 14 | 512 | conv3-512 | 1 | 3 | 3 | 512 | 512 | 2359808 |
| | 14 | 14 | 512 | 7 | 7 | 512 | maxpool | 2 | 2 | 2 | 512 | 512 | 0 |
| 14 | 1 | 1 | 25088 | 1 | 1 | 4096 | fc | | 1 | 1 | 25088 | 4096 | 102764544 |
| 15 | 1 | 1 | 4096 | 1 | 1 | 4096 | fc | | 1 | 1 | 4096 | 4096 | 16781312 |
| 16 | 1 | 1 | 4096 | 1 | 1 | 1000 | fc | | 1 | 1 | 4096 | 1000 | 4097000 |
| Total | | | | | | | | | | | | | 138,423,208 |

Table 5.1 lists the number of learning parameters (in millions) for each configuration. VGG16 yields ~138 million parameters; VGG19, which is a deeper version of VGGNet, has more than 144 million parameters. VGG16 is more commonly used because it performs almost as well as VGG19 but with fewer parameters.

**Table 5.1 VGGNet architecture parameters (in millions)**

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

***Learning hyperparameters:***

Simonyan and Zisserman followed a training procedure similar to that of AlexNet: the training 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.

***VGGNet performance:***

VGG16 achieved a top-5 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.

## 1. Inception and GoogLeNet

The Inception network came to the world in 2014 when a group of researchers at Google published their paper, "Going Deeper with Convolutions." The main hallmark of this architecture is building a deeper neural network while improving the utilization of the computing resources inside the network. One particular incarnation of the Inception network is called GoogLeNet and was used in the team's submission for ILSVRC 2014. It uses a network 22 layers deep (deeper than VGGNet) while reducing the number of parameters by 12 times (from ~138 million to ~13 million) and achieving significantly more accurate results. The network used a CNN inspired by the classical networks (AlexNet and VGGNet) but implemented a novel element dubbed as the inception module.

- **Inception (GoogleNet)** (2014, Christian Szegedy et al.)

### 1 Novel features of Inception:

Szegedy et al. took a different approach when designing their network architecture. As we've seen in the previous networks, there are some architectural decisions that you need to make for each layer when you are designing a network, such as these:

- The kernel size of the convolutional layer—We've seen in previous architectures that the kernel size varies: 1 × 1, 3 × 3, 5 × 5, and, in some cases, 11 × 11 (as in Alex- Net). When designing the convolutional layer, we find ourselves trying to pick and tune the kernel size of each layer that fits our dataset. Smaller kernels capture finer details of the image, whereas bigger filters will leave out minute details.
- When to use the pooling layer—AlexNet uses pooling layers every one or two convolutional layers to downsize spatial features. VGGNet applies pooling after every two, three, or four convolutional layers as the network gets deeper. Configuring the kernel size and positioning the pool layers are decisions we make mostly by trial and error and experiment with to get the optimal results. Inception says, "Instead of choosing a desired filter size in a convolutional layer and deciding where to place the pooling layers, let's apply all of them all together in one block and call it the inception module." That is, rather than stacking layers on top of each other as in classical architectures, Szegedy and his team suggest that we create an inception module consisting of several convolutional layers with different kernel sizes. The architecture is then developed by stacking the inception modules on top of each other. Figure below shows how classical convolutional networks are architected versus the Inception network.

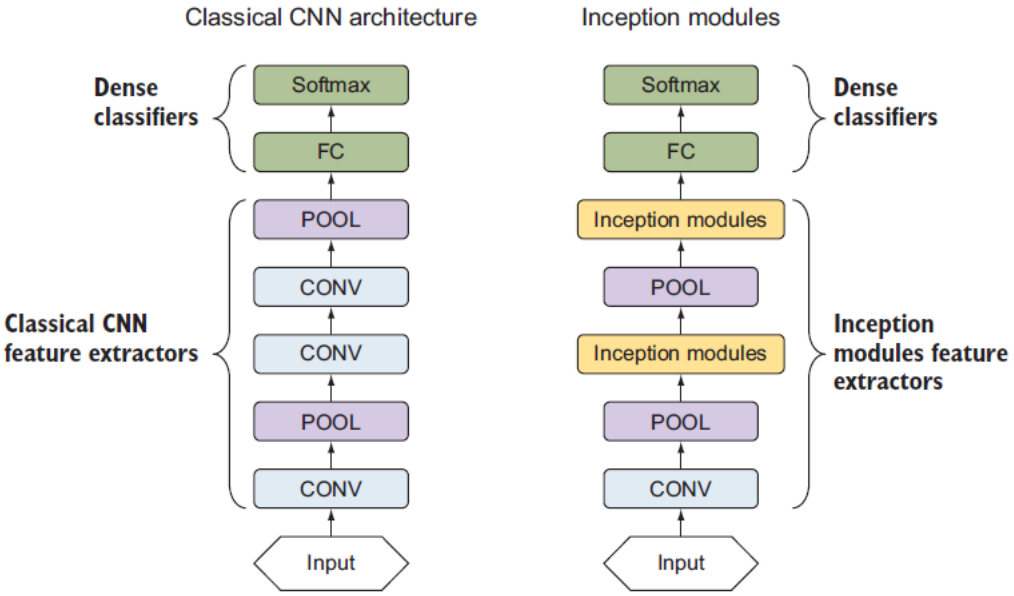



Figure 5.10 Classical convolutional networks vs. the Inception network

From the diagram, you can observe the following:

- In classical architectures like LeNet, AlexNet, and VGGNet, we stack convolutional and pooling layers on top of each other to build the feature extractors. At the end, we add the dense fully connected layers to build the classifier.
- In the Inception architecture, we start with a convolutional layer and a pooling layer, stack the inception modules and pooling layers to build the feature extractors, and then add the regular dense classifier layers.

**2 Inception module: Naive version**

The inception module is a combination of four layers:

- 1 × 1 convolutional layer
- 3 × 3 convolutional layer
- 5 × 5 convolutional layer
- 3 × 3 max-pooling layer

The outputs of these layers are concatenated into a single output volume forming the input of the next stage. The naive representation of the inception module is shown in figure 5.11.

Let's follow along with this example:

1. Suppose we have an input dimensional volume from the previous layer of size 32 × 32 × 200.
2. We feed this input to four convolutions simultaneously:

– 1 × 1 convolutional layer with depth = 64 and padding = same. The output of this kernel = 32 × 32 × 64.

– 3 × 3 convolutional layer with depth = 128 and padding = same. Output = 32 ×32 × 128.

– 5 × 5 convolutional layer with depth = 32 and padding = same. Output = 32 ×32 × 32.

– 3 × 3 max-pooling layer with padding = same and strides = 1. Output = 32 ×32 × 32.

3. We concatenate the depth of the four outputs to create one output volume of dimensions 32 × 32 × 256.

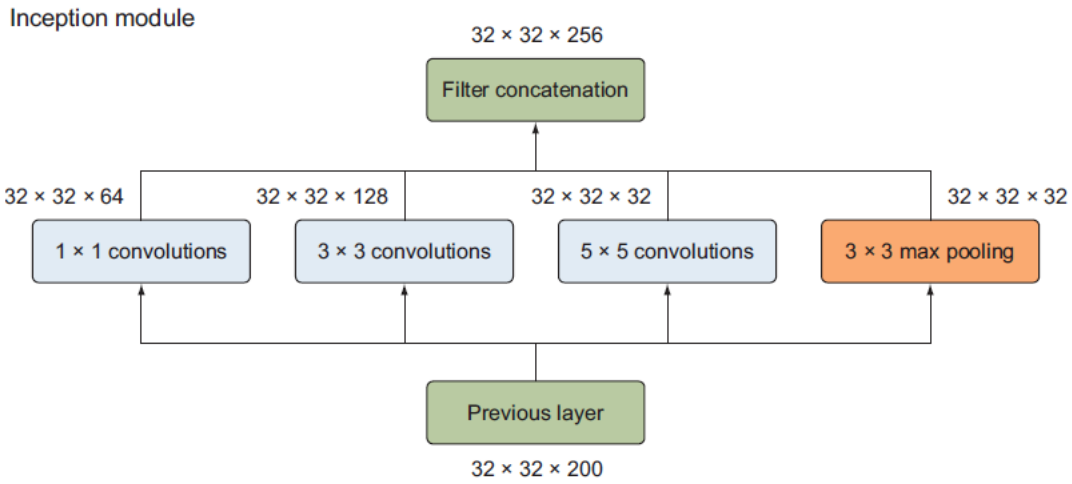



**Figure 5.11 Naive representation of an inception module**

Now we have an inception module that takes an input volume of 32 × 32 × 200 and outputs a volume of 32 × 32 × 256.

## 3 Inception module with dimensionality reduction

The naive representation of the inception module that we just saw has a big computational cost problem that comes with processing larger filters like the 5 × 5 convolutional layer. To get a better sense of the compute problem with the naive representation, let's calculate the number of operations that will be performed for the 5 × 5 convolutional layer in the previous example.

The input volume with dimensions of 32 × 32 × 200 will be fed to the 5 × 5 convolutional layer of 32 filters with dimensions = 5 × 5 × 32. This means the total number of multiplications that the computer needs to compute is 32 × 32 × 200 multiplied by 5 × 5 × 32, which is more than 163 million operations. While we can perform this many operations with modern computers, this is still pretty expensive. This is when the dimensionality reduction layers can be very useful.

### DIMENSIONALITY REDUCTION LAYER (1 × 1 CONVOLUTIONAL LAYER)

The 1 × 1 convolutional layer can reduce the operational cost of 163 million operations to about a tenth of that. That is why it is called a reduce layer. The idea here is to add a 1 × 1 convolutional layer before the bigger kernels like the 3 × 3 and 5 × 5 convolutional layers, to reduce their depth, which in turn will reduce the number of operations.

Let's look at an example. Suppose we have an input dimension volume of 32 × 32 ×200. We then add a 1 × 1 convolutional layer with a depth of 16. This reduces the dimension volume from 200 to 16 channels. We can then apply the 5 × 5 convolutional layer on the output, which has much less depth (figure 5.12).

Notice that the 32 × 32 × 200 input is processed through the two convolutional layers and outputs a volume of dimensions 32 × 32 × 32, which is the same as produced without applying the dimensionality reduction layer. But here, instead of processing the 5 × 5 convolutional layer on the entire 200 channels of the input volume, we take this huge volume and shrink its representation to a much smaller intermediate volume that has only 16 channels.

Now, let's look at the computational cost involved in this operation and compare it to the 163 million multiplications that we got before applying the reduce layer:

Computation

= operations in the 1 × 1 convolutional layer + operations in the 5 × 5 convolutional layer = (32 × 32

× 200) multiplied by (1 × 1 × 16 + 32 × 32 × 16) multiplied by (5 × 5 × 32) = 3.2 million + 13.1 million

The total number of multiplications in this operation is 16.3 million, which is a tenth of the 163 million multiplications that we calculated without the reduce layers.

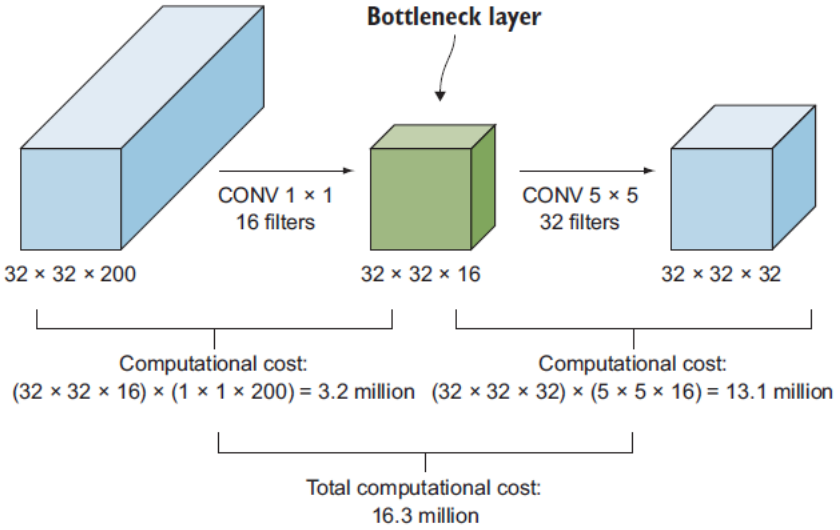
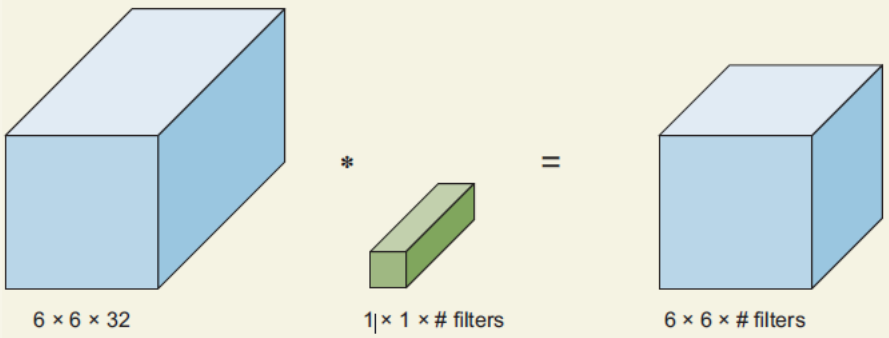



**Figure 5.12 Dimensionality reduction is used to reduce the computational cost by reducing the depth of the layer.**

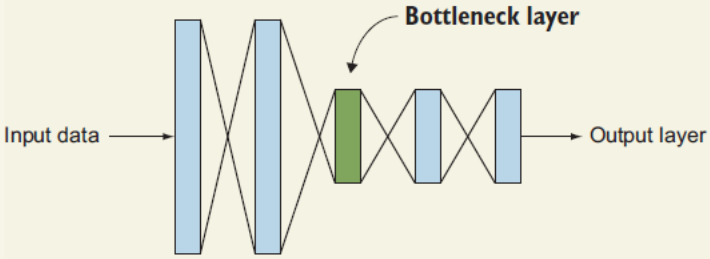
## The 1 × 1 convolutional layer

The idea of the 1 × 1 convolutional layer is that it preserves the spatial dimensions (height and width) of the input volume but changes the number of channels of the volume (depth):




**1 × 1 conv layers preserve the spatial dimensions but change the depth.**

The 1 × 1 convolutional layers are also known as *bottleneck layers* because the bottleneck is the smallest part of the bottle and reduce layers reduce the dimensionality of the network, making it look like a bottleneck:




1 × 1 convolutional layers are called *bottleneck layers*.

## IMPACT OF DIMENSIONALITY REDUCTION ON NETWORK PERFORMANCE

You might be wondering whether shrinking the representation size so dramatically hurts the performance of the neural network. Szegedy et al. ran experiments and found that as long as you implement the reduce layer in moderation, you can shrink the representation size significantly without hurting performance—and save a lot of computations. Now, let's put the reduce layers into action and build a new inception module with dimensionality reduction. To do that, we will keep the same concept of concatenating the four layers from the naive representation. We will add a 1 × 1 convolutional reduce layer before the 3 × 3 and 5 × 5 convolutional layers to reduce their computational cost. We will also add a 1 × 1 convolutional layer after the 3 × 3 max-pooling layer because pooling layers don't reduce the depth for their inputs. So, we will need to apply the reduce layer to their output before we do the concatenation (figure 5.13).

Inception module with dimensionality reduction

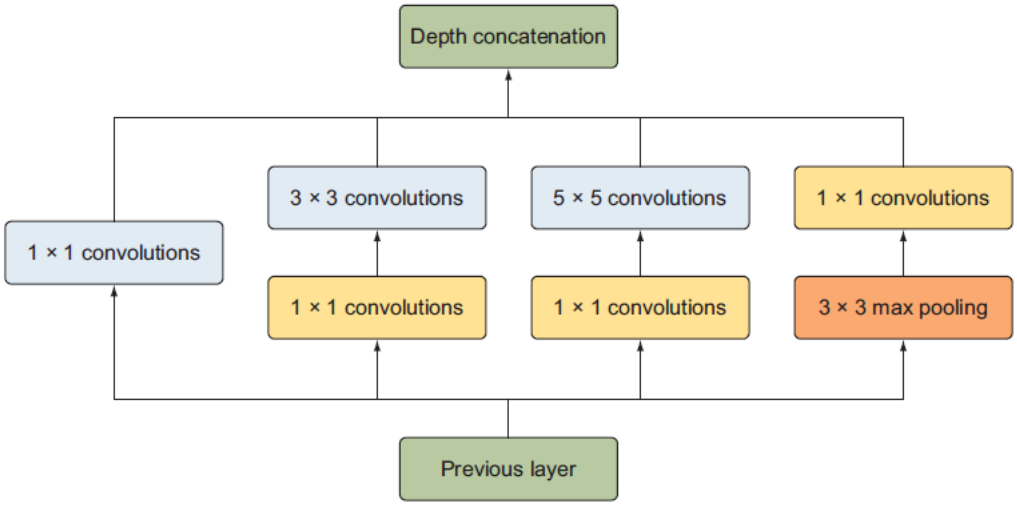



Figure 5.13 Building an inception module with dimensionality reduction

We add dimensionality reduction prior to bigger convolutional layers to allow for increasing the number of units at each stage significantly without an uncontrolled blowup in computational complexity at later stages. Furthermore, the design follows the practical intuition that visual information should be processed at various scales and then aggregated so that the next stage can abstract features from the different scales simultaneously.

## RECAP OF INCEPTION MODULES

To summarize, if you are building a layer of a neural network and you don't want to have to decide what filter size to use in the convolutional layers or when to add pooling layers, the inception module lets you use them all and concatenate the depth of all the outputs. This is called the naive

representation of the inception module.

We then run into the problem of computational cost that comes with using large filters. Here, we use a 1 × 1 convolutional layer called the reduce layer that reduces the computational cost significantly. We add reduce layers before the 3 × 3 and 5 × 5 convolutional layers and after the max-pooling layer to create an inception module with dimensionality reduction.

**Inception and GoogLeNet 223**

We then run into the problem of computational cost that comes with using large filters. Here, we use a 1 × 1 convolutional layer called the reduce layer that reduces the computational cost significantly. We add reduce layers before the 3 × 3 and 5 × 5 convolutional layers and after the max-pooling layer to create an inception module with dimensionality reduction.

## 4 Inception architecture

Now that we understand the components of the inception module, we are ready to build the Inception network architecture. We use the dimension reduction representation of the inception module, stack inception modules on top of each other, and add a 3 × 3 pooling layer in between for downsampling, as shown in figure below.

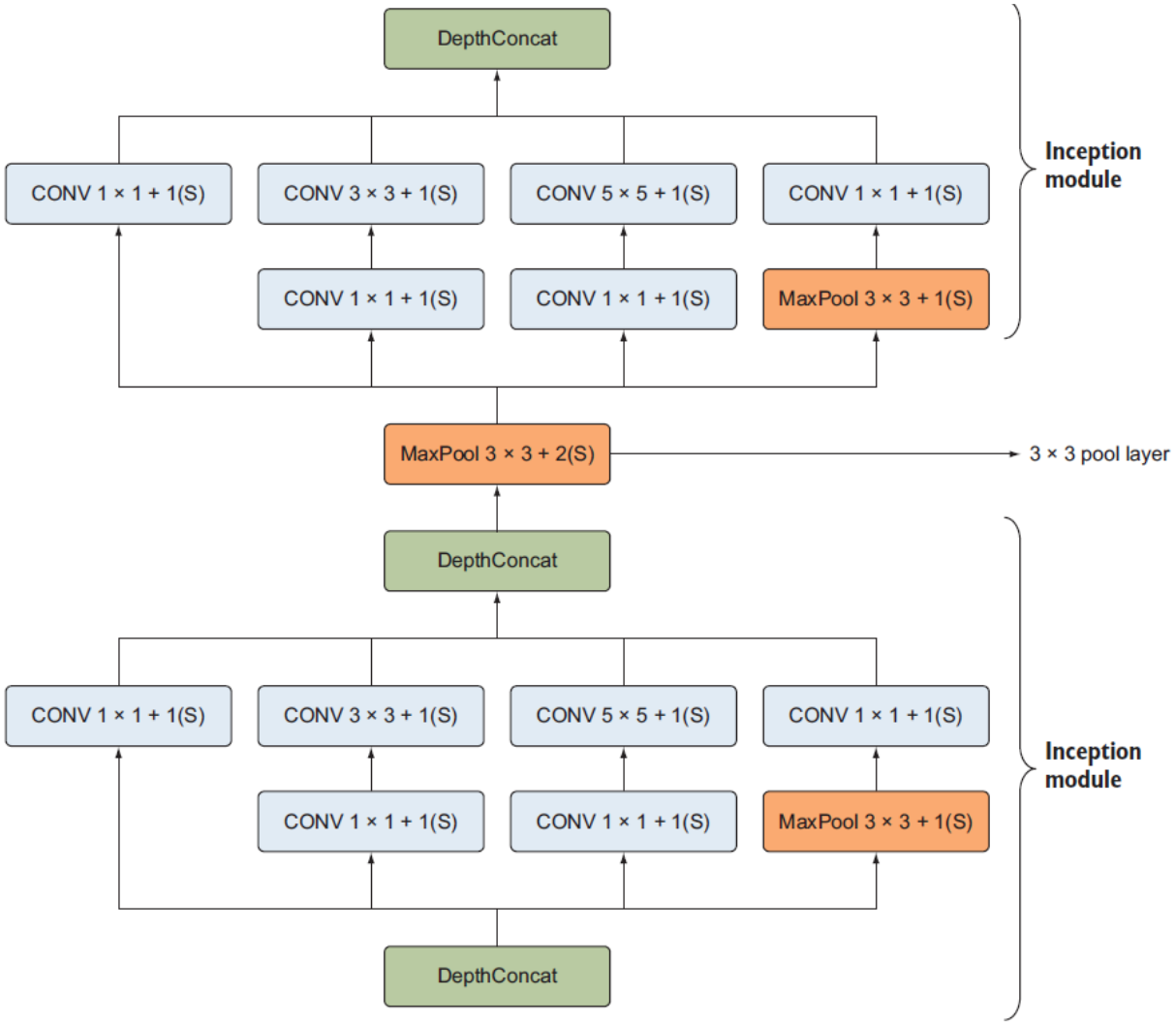



Figure 5.14 We build the inception network by adding a stack of inception modules on top of each other.

We can stack as many inception modules as we want to build a very deep convolutional network. In

the original paper, the team built a specific incarnation of the inception module and called it GoogLeNet. They used this network in their submission for the ILSVRC 2014 competition. The GoogLeNet architecture is shown in figure below.

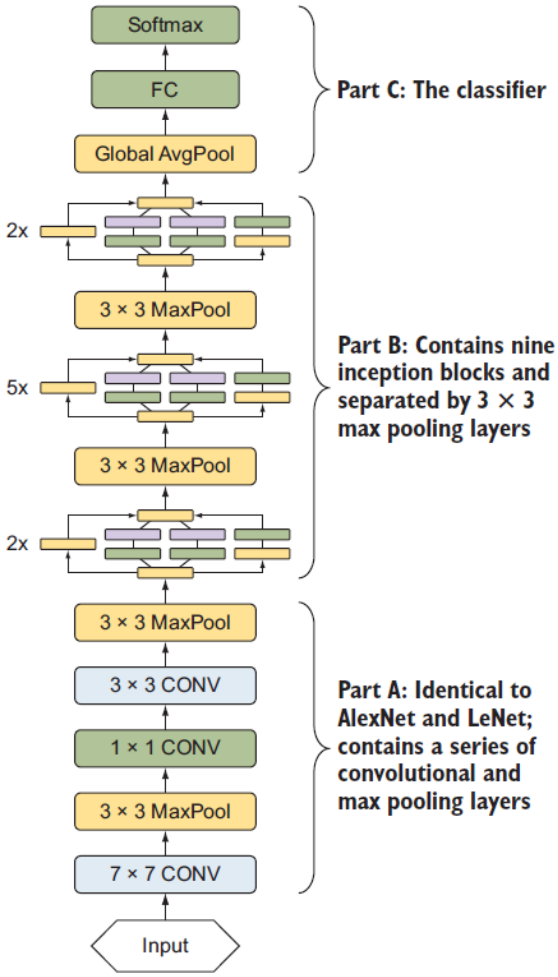



Figure 5.15 The full GoogLeNet model consists of three parts: the first part has the classical CNN architecture like AlexNet and LeNet, the second part is a stack of inceptions modules and pooling layers, and the third part is the traditional fully connected classifiers.

As you can see, GoogLeNet uses a stack of a total of nine inception modules and a max pooling layer every several blocks to reduce dimensionality. To simplify this implementation, we are going to break down the GoogLeNet architecture into three parts:

Part A—Identical to the AlexNet and LeNet architectures; contains a series of convolutional and pooling layers.

Part B—Contains nine inception modules stacked as follows: two inception modules + pooling layer + five inception modules + pooling layer + five inception modules.

Part C—The classifier part of the network, consisting of the fully connected and softmax layers.

The Inception Module is characterized by several key features that differentiate it from traditional CNN layers:

- **Multi-level Feature Extraction**
- **Dimensionality Reduction**
- **Pooling**
- **Concatenation**

**Advantages of the Inception Module:**

The Inception Module offers several advantages over traditional CNN architectures:

- **Efficiency:** By implementing filters of multiple sizes, the module efficiently uses computing resources to extract relevant features without the need for deeper or wider networks.
- **Reduced Overfitting:** The architecture's complexity and depth help in learning more robust features, which can reduce overfitting, especially when combined with other regularization techniques.
- **Improved Performance:** Networks with Inception Modules have shown improved performance on various benchmark datasets for image recognition and classification tasks.

# ResNet

The Residual Neural Network (ResNet) was developed in 2015 by a group from the Microsoft Research team.5 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.

## 1 Novel features of ResNet

Looking at how neural network architectures evolved from LeNet, AlexNet, VGGNet, and Inception, you might have noticed that the deeper the network, the larger its learning capacity, and the better it extracts features from images. This mainly happens because very deep networks are able to represent very complex functions, which allows the network to learn features at many different levels of abstraction, from edges (at the lower layers) to very complex features (at the deeper layers).
Earlier in this chapter, we saw deep neural networks like VGGNet-19 (19 layers) and GoogLeNet (22 layers). Both performed very well in the ImageNet challenge. But can we build even deeper networks? We learned from chapter 4 that one downside of adding too many layers is that doing so makes the network more prone to overfit the training data. This is not a major problem because we can use regularization techniques like dropout, L2 regularization, and batch normalization to avoid overfitting. So, if we can take care of the overfitting problem, wouldn't we want to build networks that are 50, 100, or even 150 layers deep? The answer is yes. We definitely should try to build very deep neural networks. We need to fix just one other problem, to unblock the capability of building super-deep networks: a phenomenon called vanishing gradients.

### Vanishing and exploding gradients:

The problem with very deep networks is that the signal required to change the weights becomes very small at earlier layers. To understand why, let's consider the gradient descent process explained in chapter 2. As the network backpropagates the gradient of the error from the final layer back to the first layer, it is multiplied by the weight matrix at each step; thus the gradient can decrease exponentially quickly to zero, leading to a vanishing gradient phenomenon that prevents the earlier layers from learning. As a result, the network's performance gets saturated or even starts to degrade rapidly.
In other cases, the gradient grows exponentially quickly and "explodes" to take very large values. This phenomenon is called exploding gradients.

To solve the vanishing gradient problem, He et al. created a shortcut that allows the gradient to be directly backpropagated to earlier layers. These shortcuts are called skip connections: they are used to flow information from earlier layers in the network to later layers, creating an alternate shortcut path for the gradient to flow through. Another important benefit of the skip connections is that they allow the model to learn an identity function, which ensures that the layer will perform at least as well as the previous layer (figure below).

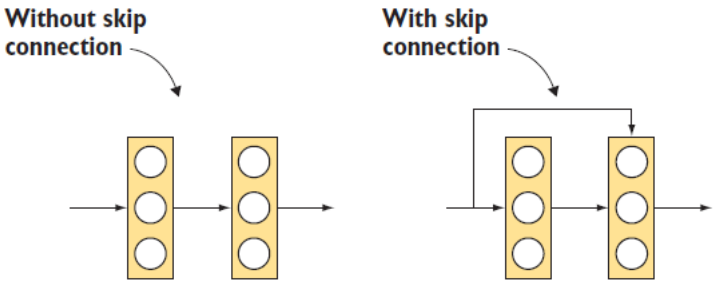



**Figure 5.19 Traditional network without skip connections (left); network with a skip connection (right).**

At left in figure above is the traditional stacking of convolutional layers one after the other. On the right, we still stack convolutional layers as before, but we also add the original input to the output of the convolutional block. This is a skip connection. We then add both signals: skip connection + main path.

Note that the shortcut arrow points to the end of the second convolutional layer— not after it.

The reason is that we add both paths before we apply the ReLU activation function of this layer. As you can see in figure 5.20, the X signal is passed along the shortcut path and then added to the main path, f(x). Then, we apply the ReLU activation to f(x) + x to produce the output signal: relu(f(x) + x).

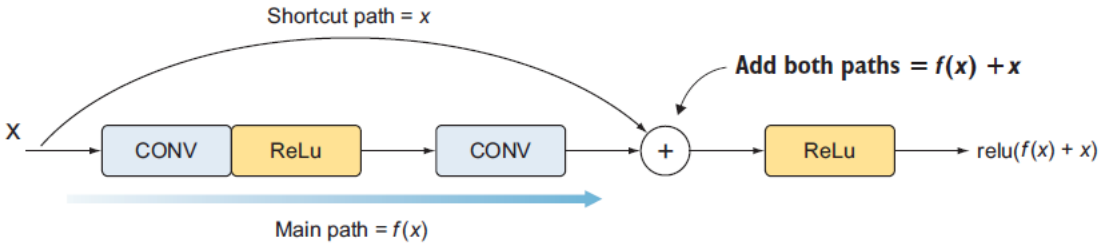



**Figure 5.20 Adding the paths and applying the ReLU activation function to solve the vanishing gradient problem that usually comes with very deep networks**

This combination of the skip connection and convolutional layers is called a *residual block*. Similar to the Inception network, ResNet is composed of a series of these residual block building blocks that are stacked on top of each other (figure below).

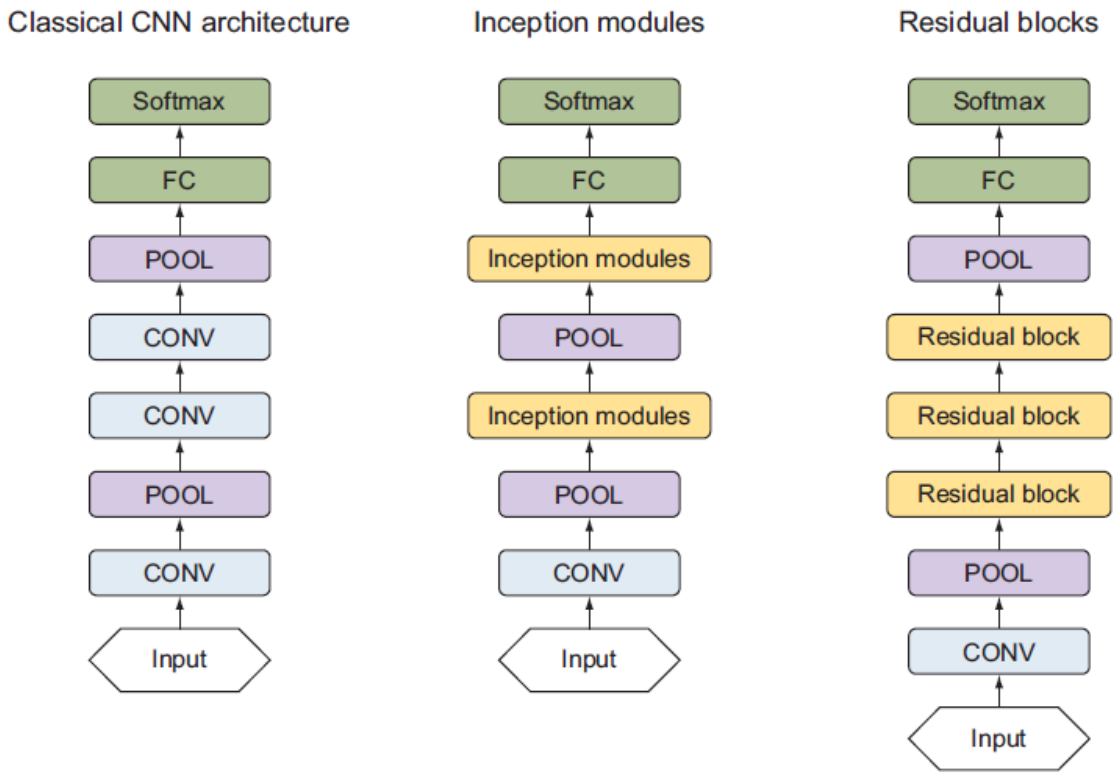



**Figure 5.21 Classical CNN architecture (left). The Inception network consists of a set of inception modules (middle). The residual network consists of a set of residual blocks (right).**

From the figure, you can observe the following:

 *Feature extractors*—To build the feature extractor part of ResNet, we start with a convolutional layer and a pooling layer and then stack residual blocks on top of each other to build the network. When we are designing our ResNet network, we can add as many residual blocks as we want to build even deeper networks.

 *Classifiers*—The classification part is still the same as we learned for other networks: fully connected layers followed by a softmax.

Now that you know what a skip connection is and you are familiar with the high-level architecture of ResNet, let's unpack residual blocks to understand how they work.

### *5.6.2 Residual blocks*

A residual module consists of two branches:

 *Shortcut path* (figure 5.22)—Connects the input to an addition of the second branch.

 *Main path*—A series of convolutions and activations. The main path consists of three convolutional layers with ReLU activations. We also add batch normalization to each convolutional layer to reduce overfitting and speed up training.

The main path architecture looks like this: [CONV ⇒ BN ⇒ ReLU] × 3.

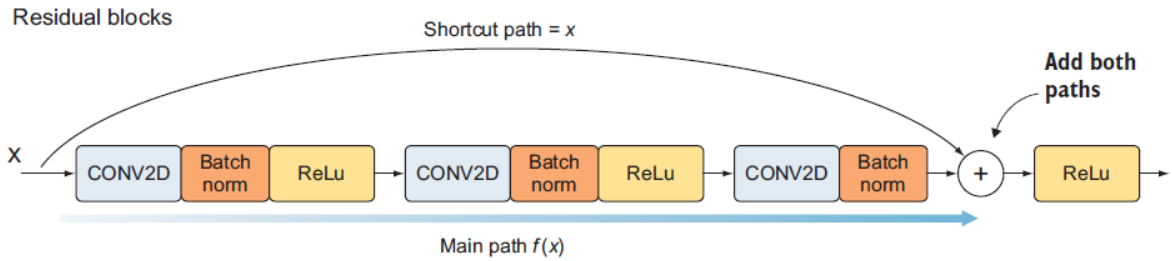



**Figure 5.22 The output of the main path is added to the input value through the shortcut before they are fed to the ReLU function.**

Similar to what we explained earlier, the shortcut path is added to the main path right before the activation function of the last convolutional layer. Then we apply the ReLU function after adding the two paths.

Notice that there are no pooling layers in the residual block. Instead, He et al. decided to do dimension downsampling using bottleneck 1 × 1 convolutional layers, similar to the Inception network. So, each residual block starts with a 1 × 1 convolutional layer to downsample the input dimension volume, and a 3 × 3 convolutional layer and another 1 × 1 convolutional layer to downsample the output. This is a good technique to keep control of the volume dimensions across many layers. This configuration is called a *bottleneck residual block*.

When we are stacking residual blocks on top of each other, the volume dimensions change from one block to another. And as you might recall from the matrices introduction in chapter 2, to be able to perform matrix addition operations, the matrices should have similar dimensions. To fix this problem, we need to downsample the shortcut path as well, before merging both paths. We do that by adding a bottleneck layer (1 × 1 convolutional layer + batch normalization) to the shortcut path, as shown in figure 5.23. This is called the *reduce shortcut*.

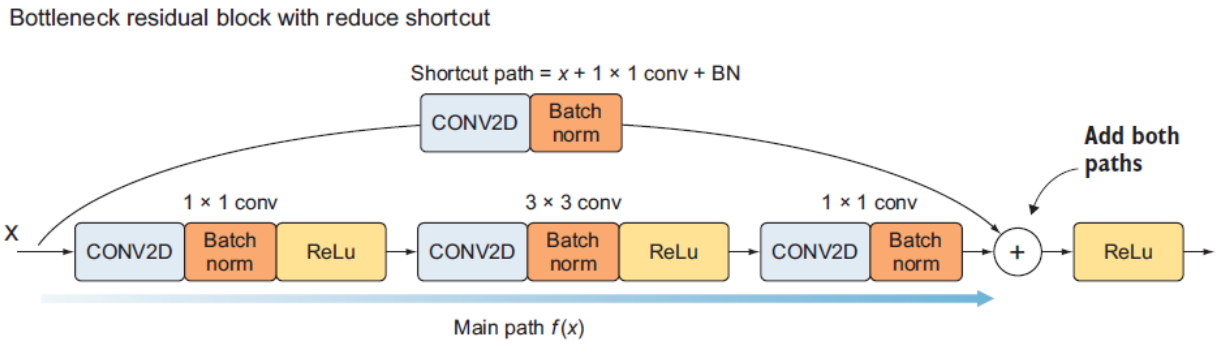



**Figure 5.23 To reduce the input dimensionality, we add a bottleneck layer (1 × 1 convolutional layer + batch normalization) to the shortcut path. This is called the *reduce shortcut*.**

let's recap the discussion of 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.

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)

## Transfer learning-1

Humans possess an extraordinary ability to transfer knowledge. When faced with a new problem or challenge, we naturally tap into our reservoir of past experiences and apply relevant expertise to resolve it. It's like having a powerful weapon that allows us to navigate tasks with ease and efficiency. Thus, if you know how to ride a bicycle and want to learn to ride a motorbike, your experience with a bicycle will help you handle tasks like balancing and steering, making the learning process smoother compared to starting from scratch.

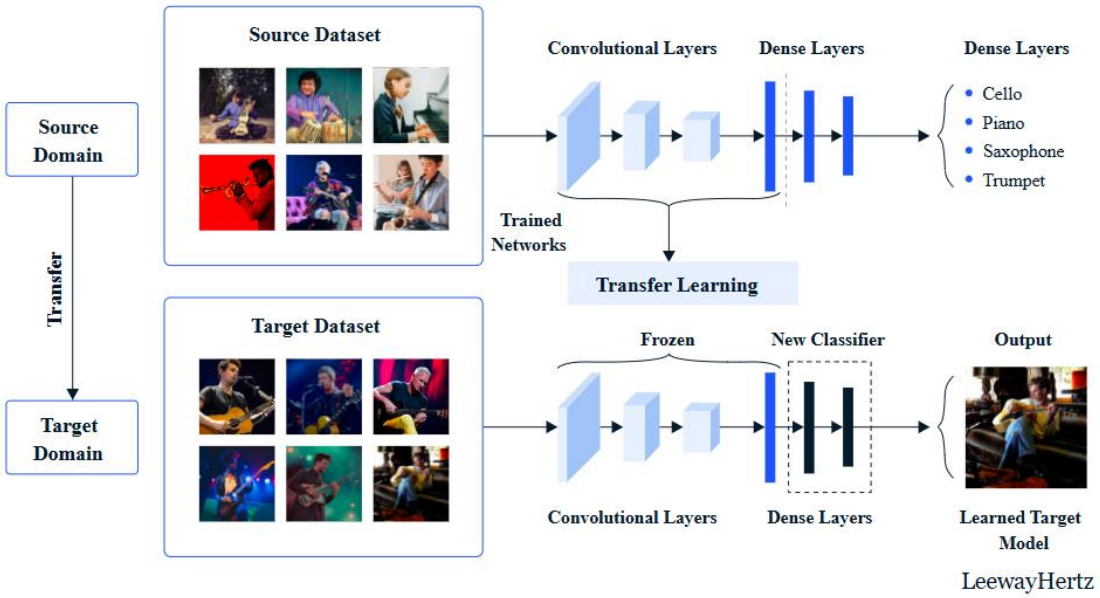
The idea of transfer learning in machine learning was born out of inspiration from the human learning approach. Transfer learning is essentially a technique where knowledge gained from training a model on one task or domain is utilized to improve the performance of the model on a different task or domain. It's, however, worth noting that this approach proves beneficial when the second task shares similarities with the first task or when limited data is available for the second task.

The fundamental idea behind transfer learning is centered around using a deep learning network that has already been trained on a related problem. This pre-trained network is initially trained on a similar task or domain, allowing it to learn general features and patterns from the data. Initializing the network with these learned weights and features provides a starting point for a new task, where the network can be fine-tuned or further trained using a smaller dataset specific to the new problem. By utilizing this network as a foundation, the duration of training required for the new yet related problem can be significantly reduced.

**How does transfer learning work?**

Before delving into the operational mechanics of transfer learning, it is essential to understand the fundamentals of deep learning. A deep neural network comprises numerous weights that connect the layers of neurons within the network. These weights, usually real values, undergo adjustment during the training process and are utilized to apply inputs, including those from intermediate layers, to feed-forward an output classification. The fundamental concept of transfer learning begins with a pre-initialized deep learning network trained on a similar problem. By leveraging this network, the training duration for the new but related problem can be significantly reduced while achieving comparable performance.

When employing transfer learning, we encounter the concept of layer freezing. A layer, whether a Convolutional Neural Network (CNN) layer, hidden layer, or a subset of layers, is considered frozen when it is no longer trainable, meaning its weights remain unchanged during the training process. Conversely, unfrozen layers undergo regular training updates.




In the image, layers are "frozen" during the transfer learning process to prevent the weights of the pre-trained convolutional layers from being updated.

Transfer learning: the transfer of knowledge from a pretrained network in one domain to your own problem in a different domain.
Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.
It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.

- 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.

## 1. What problems does transfer learning solve?

As the name implies, transfer learning means transferring what a neural network has learned from being trained on a specific dataset to another related problem (figure 6.1). Transfer learning is currently very popular in the field of DL because it enables you to train deep neural networks with comparatively little data in a short training time. The importance of transfer learning comes from the fact that in most real-world problems, we typically do not have millions of labeled images to train such complex models.

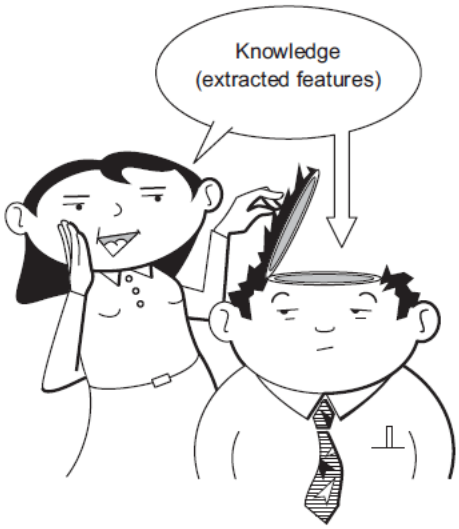



**Figure 6.1 Transfer learning is the transfer of the knowledge that the network has acquired from one task to a new task. In the context of neural networks, the acquired knowledge is the extracted features.**

The idea is pretty straightforward. First we train a deep neural network on a very large amount of data. During the training process, the network extracts a large number of useful features that can be used to detect objects in this dataset. We then transfer these extracted features (*feature maps*) to a new network and train this new network on our new dataset to solve a different problem. Transfer learning is a great way to shortcut the process of collecting and training huge amounts of data simply by reusing the model weights from pretrained models that were developed for standard CV benchmark datasets, such as the ImageNet image-recognition tasks. Top-performing models can be downloaded and used directly, or integrated into a new model for your own CV problems.

The question is, why would we want to use transfer learning? Why don't we just train a neural network directly on our new dataset to solve our problem? To answer this question, we first need to know the main problems that transfer learning solves.

We'll discuss those now; then I'll go into the details of how transfer learning works and the different approaches to apply it. Deep neural networks are immensely data-hungry and rely on huge amounts of labeled data to achieve high performance. In practice, very few people train an entire convolutional network from scratch. This is due to two main problems:

- *Data problem*—Training a network from scratch requires a lot of data in order to get decent results, which is not feasible in most cases. It is relatively rare to have a dataset of sufficient size to solve your problem. It is also very expensive to acquire and label data: this is mostly a manual process done by humans capturing images and labeling them one by one, which makes it a nontrivial task.
- *Computation problem*—Even if you are able to acquire hundreds of thousands of images for your problem, it is computationally very expensive to train a deep neural network on millions of images because doing so usually requires weeks of training on multiple GPUs. Also keep in mind that training a neural network is an iterative process. So, even if you happen to have the computing power required to train a complex neural network, spending weeks experimenting with different hyperparameters in each training iteration until you finally reach satisfactory results will make the project very costly.

Additionally, an important benefit of using transfer learning is that it helps the model generalize its learnings and avoid overfitting. When you apply a DL model in the wild, it is faced with countless conditions it may never have seen before and does not know how to deal with; each client has its own preferences and generates data that is different from the data used for training. The model is asked to perform well on many tasks that are related to but not exactly similar to the task it was trained for.

For example, when you deploy a car classifier model to production, people usually have different camera types, each with its own image quality and resolution. Also, images can be taken during

different weather conditions. These image nuances vary from one user to another. To train the model on all these different cases, you either have to account for every case and acquire a lot of images to train the network on, or try to build a more robust model that is better at generalizing to new use cases. This is what transfer learning does. Since it is not realistic to account for all the cases the model may face in the wild, transfer learning can help us deal with novel scenarios. It is necessary for production-scale use of DL that goes beyond tasks and domains where labeled data is plentiful. Transferring features extracted from another network that has seen millions of images will make our model less prone to overfit and help it generalize better when faced with novel scenarios. You will be able to fully grasp this concept when we explain how transfer learning works in the following sections.

## 2. What is transfer learning?

Armed with the understanding of the problems that transfer learning solves, let's look at its formal definition. *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.

As we discussed earlier, 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. I'm sure most of us don't have all these things. 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. This means you should never have to train an image classifier from scratch again, unless you have an exceptionally large dataset and a very large computation budget to train everything from scratch by yourself. Even if that is the case, you might be better off using transfer learning to fine-tune the pretrained network on your large dataset. Later in this chapter, we will discuss the different transfer learning approaches, and you will understand what fine-tuning means and why it is better to use transfer learning even when you have a large dataset. We will also talk briefly about some of the popular datasets mentioned here.

*In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task.*

—Jason Yosinski et al.

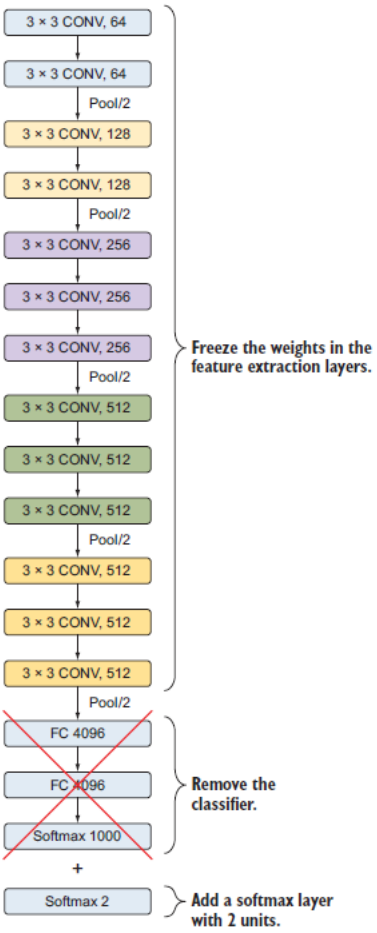



Figure 6.2 Example of applying transfer learning to a VGG16 network. We freeze the feature extraction part of the network and remove the classifier part. Then we add our new classifier softmax layer with two hidden units.

### 3. How transfer learning works

So far, we learned what the transfer learning technique is and the main problems it solves. We also saw an example of how to take a pretrained network that was trained on ImageNet and transfer its learnings to our specific task. Now, let's see *why transfer learning works*, what is really being transferred from one problem to another, and how a network that is trained on one dataset can perform well on a different, possibly unrelated, dataset.

The following quick questions are reminders from previous chapters to get us to the core of what is happening in transfer learning:

a. What is really being learned by the network during training? The short answer is: *feature maps*.

b. How are these features learned? During the backpropagation process, the weights are updated until we get to the *optimized weights* that minimize the error function.

c. What is the relationship between features and weights? A feature map is the result of passing the weights filter on the input image during the convolution process (figure 6.4).

d. What is really being transferred from one network to another? To transfer features, we download the *optimized weights* of the pretrained network. These weights are then reused as the starting point for the training process and retrained to adapt to the new problem.

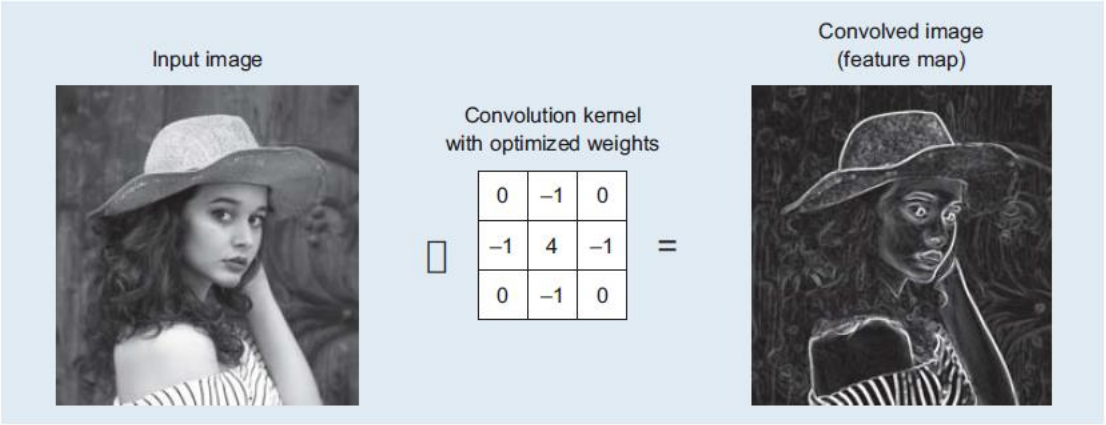



Figure 6.4 Example of generating a feature map by applying a convolutional kernel to the input image

**what we mean when we say *pretrained network:***

When we're training a convolutional neural network, the network extracts features from an image in the form of feature maps: outputs of each layer in a neural network after applying the weights filter. They are representations of the features that exist in the training set. They are called feature maps because they map where a certain kind of feature is found in the image. CNNs look for features such as straight lines, edges, and even objects. Whenever they spot these features, they report them to the feature map. Each weight filter is looking for something different that is reflected in the feature maps: one filter could be looking for straight lines, another for curves, and so on (figure 6.5).

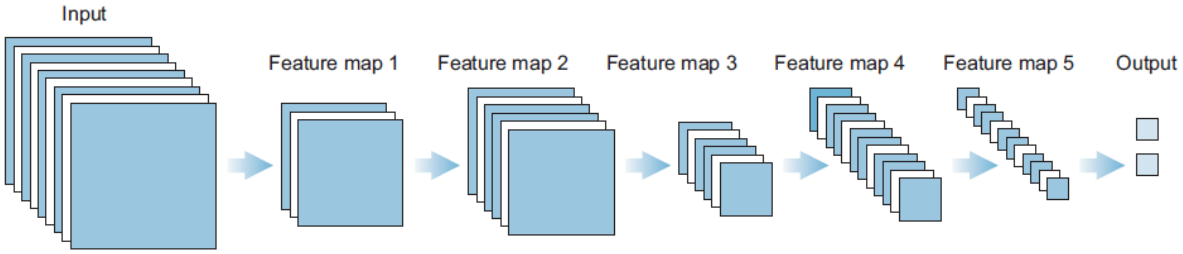



Figure 6.5 The network extracts features from an image in the form of feature maps. They are representations of the features that exist in the training set after applying the weight filters.

Now, recall that neural networks iteratively update their weights during the training cycle of feedforward and backpropagation. We say the network has been *trained* when we go through a series of training iterations and hyperparameter tuning until the network yields satisfactory results. When training is complete, we output two main items: the network architecture and the trained weights. So, when we say that we are going to use a *pretrained network*, we mean that we will download the network architecture together with the weights.

During training, the model learns only the features that exist in this training dataset. But when we download large models (like Inception) that have been trained on huge numbers of datasets (like ImageNet), all the features that have already been extracted from these large datasets are now available for us to use. I find that really exciting because these pretrained models have spotted other features that weren't in our dataset and will help us build better convolutional networks.

In vision problems, there's a huge amount of stuff for neural networks to learn about the training dataset. There are low-level features like edges, corners, round shapes, curvy shapes, and blobs; and then there are mid- and higher-level features like eyes, circles, squares, and wheels. There are many details in the images that CNNs can pick up on—but if we have only 1,000 images or even 25,000 images in our training dataset, this may not be enough data for the model to learn all those things. By using a pretrained network, we can basically download all this knowledge into our neural network to

give it a huge and much faster start with even higher performance levels.

***How do neural networks learn features?***

A neural network learns the features in a dataset step by step in increasing levels of complexity, one layer after another. These are called *feature maps*. The deeper you go through the network layers, the more image-specific features are learned. In figure 6.6, the first layer detects low-level features such as edges and curves. The output of the first layer becomes input to the second layer, which produces higher-level features like semicircles and squares. The next layer assembles the output of the previous layer into parts of familiar objects, and a subsequent layer detects the objects. As we go through more layers, the network yields an *activation map* that represents more complex features. As we go deeper into the network, the filters begin to be more responsive to a larger region of the pixel space. Higher-level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

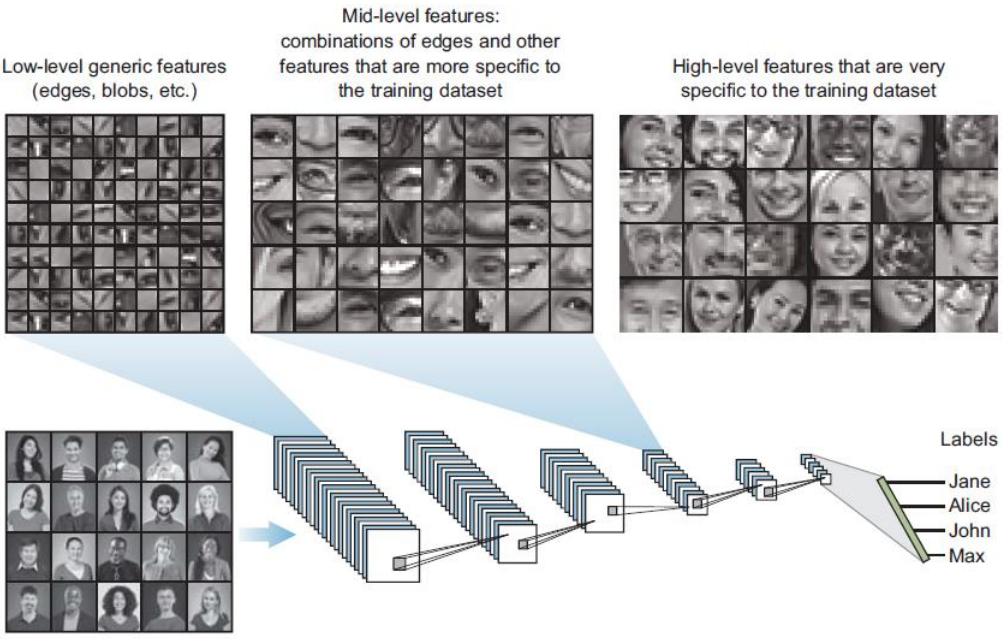



**Figure 6.6 An example of how CNNs detect low-level generic features at the early layers of the network. The deeper you go through the network layers, the more image-specific features are learned.**

Consider the example in figure 6.6. Suppose we are building a model that detects human faces. We notice that the network learns low-level features like lines, edges, and blobs in the first layer. These low-level features appear not to be specific to a particular dataset or task; they are general features that are applicable to many datasets and tasks. The mid-level layers assemble those lines to be able to recognize shapes, corners, and circles. Notice that the extracted features start to get a little more specific to our task (human faces): mid-level features contain combinations of shapes that form objects in the human face like eyes and noses. As we go deeper through the network, we notice that features eventually transition from general to specific and, by the last layer of the network, form high-level features that are very specific to our task. We start seeing parts of human faces that distinguish one person from another.

Now, let's take this example and compare the feature maps extracted from four models that are trained to classify faces, cars, elephants, and chairs (see figure 6.7). Notice that the earlier layers' features are very similar for all the models. They represent low-level features like edges, lines, and blobs. This means models that are trained on one task capture similar relations in the data types in the earlier layers of the network and can easily be reused for different problems in other domains. The deeper we go into the network, the more specific the features, until the network overfits its training data and it becomes harder to generalize to different tasks. The lower-level features are almost always transferable from one task to another because they contain generic information like the structure and nature of how images look. Transferring information like lines, dots, curves, and small parts of objects is very valuable for the network to learn faster and with less data on the new

task.

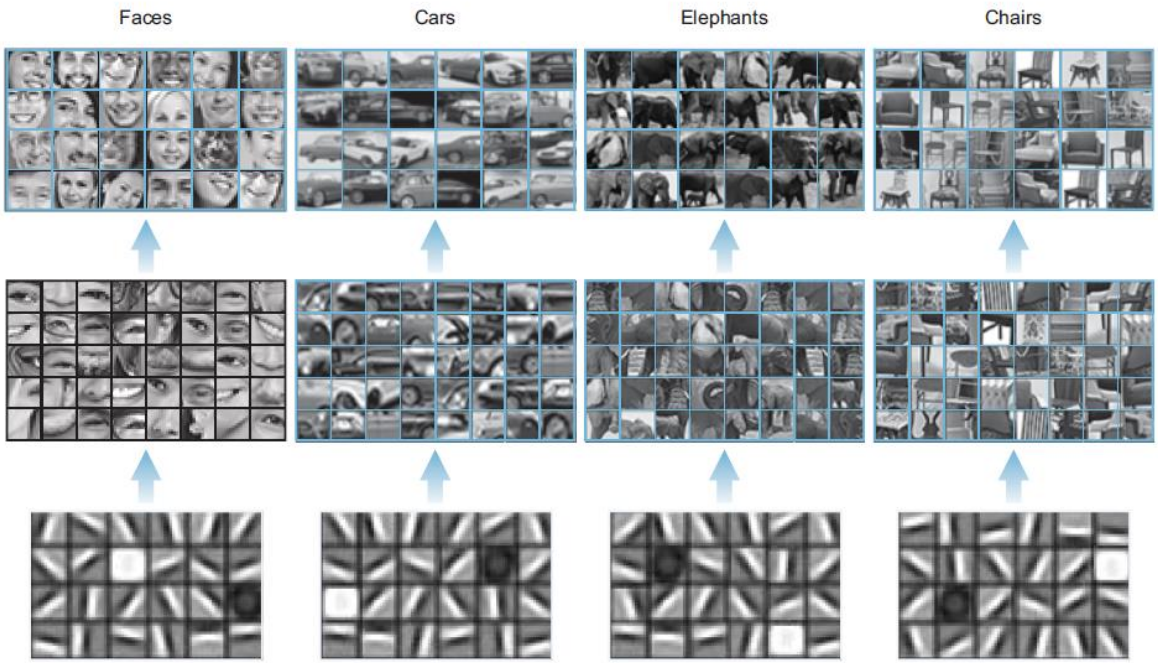



Figure 6.7 Feature maps extracted from four models that are trained to classify faces, cars, elephants, and chairs

***Transferability of features extracted at later layers:***

The transferability of features that are extracted at later layers depends on the similarity of the original and new datasets. The idea is that all images must have shapes and edges, so the early layers are usually transferable between different domains. We can only identify differences between objects when we start extracting higher-level features: say, the nose on a face or the tires on a car. Only then can we say, "Okay, this is a person, because it has a nose. And this is a car, because it has tires." Based on the similarity of the source and target domains, we can decide whether to transfer only the low-level features from the source domain, or the high-level features, or somewhere in between. This is motivated by the observation that the later layers of the network become progressively more specific to the details of the classes contained in the original dataset, as we are going to discuss in the next section.

DEFINITIONS

- The *source domain* is the original dataset that the pretrained network is trained on.
- The *target domain* is the new dataset that we want to train the network on.

## Transfer learning approaches

There are three major transfer learning approaches: pretrained network as a classifier, pretrained network as a feature extractor, and fine-tuning. Each approach can be effective and save significant time in developing and training a deep CNN model. It may not be clear which use of a pretrained model may yield the best results on your new CV task, so some experimentation may be required. In this section, we will explain these three scenarios and give examples of how to implement them.

### *1.1.Using a pretrained network as a classifier*

Using a pretrained network as a classifier doesn't involve freezing any layers or doing extra model training. Instead, we just take a network that was trained on a similar problem and deploy it directly to our task. The pretrained model is used directly to classify new images with no changes applied to it and no extra training. All we do is download the network architecture and its pretrained weights and then run the predictions directly on our new data. In this case, we are saying that the domain of our new problem is very similar to the one that the pretrained network was trained on, and it is ready to be deployed. In the dog breed example, we could have used a VGG16 network that was trained on an ImageNet dataset directly to run predictions. ImageNet already contains a lot of dog images, so a significant portion of the representational power of the pretrained network may be devoted to features that are specific to differentiating between dog breeds.

Let's see how to use a pretrained network as a classifier. In this example, we will use a VGG16 network that was pretrained on the ImageNet dataset to classify the image of the German Shepherd dog in figure 6.8.

Figure 6.8 A sample image of a German Shepherd that we will use to run predictions

### *1.2.Using a pretrained network as a feature extractor*

This approach is similar to the dog breed example that we implemented earlier in this chapter: we take a pretrained CNN on ImageNet, freeze its feature extraction part, remove the classifier part, and add our own new, dense classifier layers. In figure 6.9, we use a pretrained VGG16 network, freeze the weights in all 13 convolutional layers, and replace the old classifier with a new one to be trained from scratch. We usually go with this scenario when our new task is similar to the original dataset that the pretrained network was trained on. Since the ImageNet dataset has a lot of dog and cat examples, the feature maps that the network has learned contain a lot of dog and cat features that are very applicable to our new task. This means we can use the high-level features that were extracted from the ImageNet dataset in this new task.

To do that, we freeze all the layers from the pretrained network and only train the classifier part that we just added on the new dataset. This approach is called using a pretrained network as a feature extractor because we freeze the feature extractor part to transfer all the learned feature maps to our new problem. We only add a new classifier, which will be trained from scratch, on top of the pretrained model so that we can repurpose the previously learned feature maps for our dataset.

We remove the classification part of the pretrained network because it is often very specific to the original classification task, and subsequently it is specific to the set of classes on which the model was trained. For example, ImageNet has 1,000 classes. The classifier part has been trained to overfit the training data to classify them into 1,000 classes. But in our new problem, let's say cats versus dogs, we have only two classes. So, it is a lot more effective to train a new classifier from scratch to overfit these two classes.

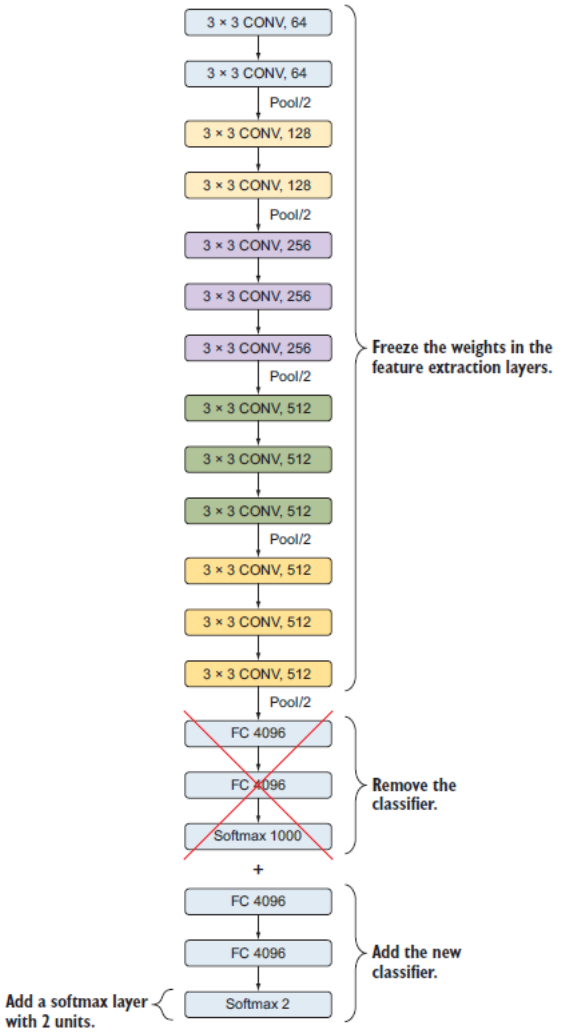



Figure 6.9 Load a pretrained VGG16 network, remove the classifier, and add your own classifier.

### *1.3.Fine-tuning*

So far, we've seen two basic approaches of using a pretrained network in transfer learning: using a pretrained network as a classifier or as a feature extractor. We generally use these approaches when the target domain is somewhat similar to the source domain. But what if the target domain is different from the source domain? What if it is *very* different? Can we still use transfer learning? Yes. Transfer learning works great even when the domains are very different. We just need to extract the *correct* feature maps from the source domain and *fine-tune* them to fit the target domain.

In figure 6.10, we show the different approaches of transferring knowledge from a pretrained network. If you are downloading the entire network with no changes and just running predictions, then you are using the network as a classifier. If you are freezing the convolutional layers only, then you are using the pretrained network as a feature extractor and transferring all of its high-level

feature maps to your domain.

The formal definition of *fine-tuning* is 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. It is called fine-tuning because when we retrain the feature extraction layers, we fine-tune the higher-order feature representations to make them more relevant for the new task dataset.

In more practical terms, if we freeze features maps 1 and 2 in figure 6.10, the new network will take feature maps 2 as its input and will start learning from this point to adapt the features of the later layers to the new dataset. This saves the network the time that it would have spent learning feature maps 1 and 2.

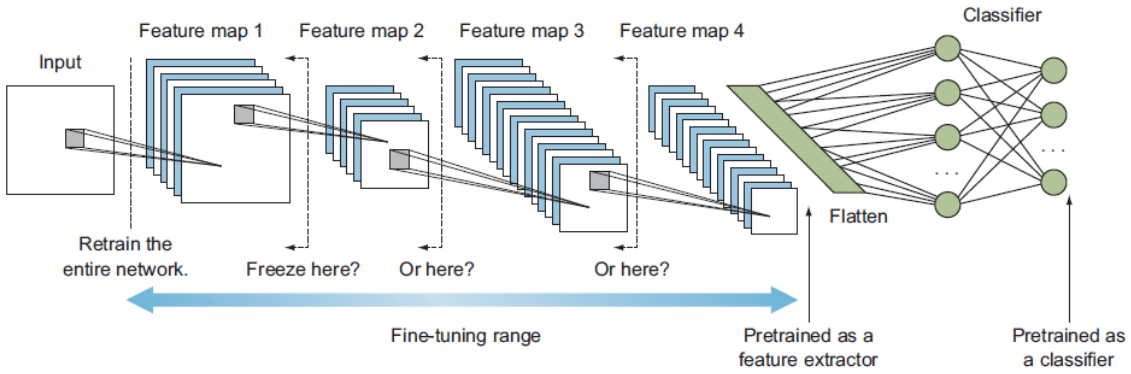



**Figure 6.10 The network learns features through its layers. In transfer learning, we make a decision to freeze specific layers of a pretrained network to preserve the learned features. For example, if we freeze the network at feature maps of layer 3, we preserve what it has learned in layers 1, 2, and 3.**

In this real-world example, transfer learning with fine-tuning helps classify medical images for pneumonia detection, where the pretrained model offers a powerful feature extractor, and fine-tuning adjusts the model for the specific task of medical image classification. This approach is widely used in fields like healthcare, where large labeled datasets are rare, but pretrained models can be adapted for specific diagnostic tasks.

WHY IS FINE-TUNING BETTER THAN TRAINING FROM SCRATCH?

When we train a network from scratch, we usually randomly initialize the weights and apply a gradient descent optimizer to find the best set of weights that optimizes our error function (as discussed in chapter 2). Since these weights start with random values, there is no guarantee that they will begin with values that are close to the desired optimal values. And if the initialized value is far from the optimal value, the optimizer will take a long time to converge. This is when fine-tuning can be very useful. The pretrained network's weights have been already optimized to learn from its dataset. Thus, when we use this network in our problem, we start with the weight values that it ended with. So, the network converges much faster than if it had to randomly initialize the weights. We are basically *fine-tuning* the already-optimized weights to fit our new problem instead of training the entire network from scratch with random weights. Even if we decide to retrain the entire pretrained network, starting with the trained weights will converge faster than having to train the network from scratch with randomly initialized weights.

USING A SMALLER LEARNING RATE WHEN FINE-TUNING

It's common to use a smaller learning rate for ConvNet weights that are being finetuned, in comparison to the (randomly initialized) weights for the new linear classifier that computes the class scores of a new dataset. This is because we expect that the ConvNet weights are relatively good, so we don't want to distort them too quickly and too much (especially while the new classifier above them is being trained from random initialization).

**2. Choosing the appropriate level of transfer learning**

Recall that early convolutional layers extract generic features and become more specific to the training data the deeper we go through the network. With that said, we can choose the level of detail for feature extraction from an existing pretrained model. For example, if a new task is quite different from the source domain of the pretrained network (for example, different from ImageNet), then perhaps the output of the pretrained model after the first few layers would be appropriate. If a new task is similar to the source domain, then perhaps the output from layers much deeper in the model can be used, or even the output of the fully connected layer prior to the softmax layer.

As mentioned earlier, choosing the appropriate level for transfer learning is a function of two important factors:

- *Size of the target dataset (small or large)*—When we have a small dataset, the network probably won't learn much from training more layers, so it will tend to overfit the new data. In this case, we most likely want to do less fine-tuning and rely more on the source dataset.
- *Domain similarity of the source and target datasets*—How similar is our new problem to the domain of the original dataset? For example, if your problem is to classify cars and boats, ImageNet could be a good option because it contains a lot of images of similar features. On the other hand, if your problem is to classify lung cancer on X-ray images, this is a completely different domain that will likely require a lot of fine-tuning.

These two factors lead to the four major scenarios:

i. The target dataset is *small* and *similar* to the source dataset.
ii. The target dataset is *large* and *similar* to the source dataset.
iii. The target dataset is *small* and *very different* from the source dataset.
iv. The target dataset is *large* and *very different* from the source dataset.

Let's discuss these scenarios one by one to learn the common rules of thumb for navigating our options.

***Scenario 1: Target dataset is small and similar to the source dataset***

Since the original dataset is similar to our new dataset, we can expect that the higherlevel features in the pretrained ConvNet are relevant to our dataset as well. Then it might be best to freeze the feature extraction part of the network and only retrain the classifier.

Another reason it might not be a good idea to fine-tune the network is that our new dataset is small. If we fine-tune the feature extraction layers on a small dataset, that will force the network to overfit to our data. This is not good because, by definition, a small dataset doesn't have enough information to cover all possible features of its objects, which makes it fail to generalize to new, previously unseen, data. So in this case, the more fine-tuning we do, the more the network is prone to overfit the new data.

For example, suppose all the images in our new dataset contain dogs in a specific weather environment—snow, for example. If we fine-tuned on this dataset, we would force the new network to pick up features like snow and a white background as dogspecific features and make it fail to classify dogs in other weather conditions. Thus the general rule of thumb is: if you have a small amount of data, be careful of overfitting when you fine-tune your pretrained network.

***Scenario 2: Target dataset is large and similar to the source dataset***

Since both domains are similar, we can freeze the feature extraction part and retrain the classifier, similar to what we did in scenario 1. But since we have more data in the new domain, we can get a performance boost from fine-tuning through all or part of the pretrained network with more confidence that we won't overfit. Fine-tuning through the entire network is not really needed because the higher-level features are related (since the datasets are similar). So a good start is to freeze approximately 60–80% of the pretrained network and retrain the rest on the new data.

***Scenario 3: Target dataset is small and different from the source dataset***

Since the dataset is different, it might not be best to freeze the higher-level features of the pretrained network, because they contain more dataset-specific features. Instead, it would work better to retrain layers from somewhere earlier in the network—or to not freeze any layers and fine-tune the entire network. However, since you have a small dataset, fine-tuning the entire network on the dataset might not be a good idea, because doing so will make it prone to overfitting. A midway solution will work better in this case. A good start is to freeze approximately the first third or half of the retrained network. After all, the early layers contain very generic feature maps that will be useful for your dataset even if it is very different.

***Scenario 4: Target dataset is large and different from the source dataset***

Since the new dataset is large, you might be tempted to just train the entire network from scratch and not use transfer learning at all. However, in practice, it is often still very beneficial to initialize weights from a pretrained model, as we discussed earlier. Doing so makes the model converge faster. In this case, we have a large dataset that provides us with the confidence to fine-tune through the entire network without having to worry about overfitting.

***Recap of the transfer learning scenarios***

We've explored the two main factors that help us define which transfer learning approach to use (size of our data and similarity between the source and target datasets). These two factors give us the four major scenarios defined in table 6.1. Figure 6.11 summarizes the guidelines for the appropriate fine-tuning level to use in each of the scenarios.

**Table 6.1 Transfer learning scenarios**

| Scenario | Size of the target data | Similarity of the original and new datasets | Approach |
|---|---|---|---|
| 1 | Small | Similar | Pretrained network as a feature extractor |
| 2 | Large | Similar | Fine-tune through the full network |
| 3 | Small | Very different | Fine-tune from activations earlier in the network |
| 4 | Large | Very different | Fine-tune through the entire network |

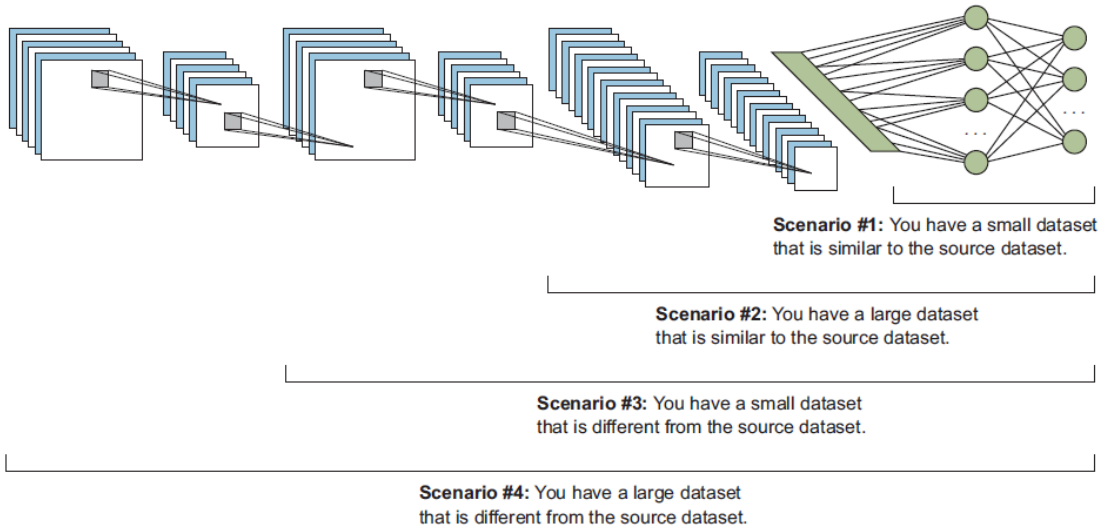



Figure 6.11 Guidelines for the appropriate fine-tuning level to use in each of the four scenarios

### *3. Open source datasets*

The CV research community has been pretty good about posting datasets on the internet.
So, when you hear names like ImageNet, MS COCO, Open Images, MNIST, CIFAR, and many others, these are datasets that people have posted online and that a lot of computer researchers have used as benchmarks to train their algorithms and get state-of-the-art results.

**Transfer Learning Advantages**

- Helps solve complex real-world problems with several constraints
- Tackle problems like having little or almost no labeled data availability
- Ease of transferring knowledge from one model to another based on domains and tasks
- Provides a path towards achieving Artificial General Intelligence some day in the future!

1. Design and implement to classify 32x32 images using MLP using tensorflow/keras and check theaccuracy.

Here CIFAR-10 dataset is used,

CIFAR-10 consists of 60,000 32x32 color images in 10 categories, with 6,000 images per category (airplane, car, bird, cat, deer, dog, frog, horse, ship, truck).

1. **Importing Libraries**

```
import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to_categorical
```

- **numpy**: Used for numerical operations.
- **Keras**: Provides modules to define models, layers, and utilities like loading datasets.
- **Flatten, Dense**: These are layers of the MLP.
- **cifar10**: Preloaded CIFAR-10 dataset.
- **to_categorical**: Converts class labels into one-hot encoding.

2. **Loading and Preprocessing the CIFAR-10 Dataset**

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train = x_train.astype('float32') / 255.0

x_test = x_test.astype('float32') / 255.0

 y_train = to_categorical(y_train, 10)

y_test = to_categorical(y_test, 10)
```

- CIFAR-10 is loaded into training and testing sets.
- Images are normalized by scaling pixel values to the range [0, 1].
- Labels are converted to one-hot encoding for classification.

**3. Building the MLP Model**

```
model = Sequential()

model.add(Flatten(input_shape=(32, 32, 3)))  # Flatten the input

model.add(Dense(512, activation='relu'))      # First hidden layer

model.add(Dense(256, activation='relu'))      # Second hidden layer

model.add(Dense(10, activation='softmax'))    # Output layer
```

- **Flatten Layer**: Converts the 32x32x3 image input into a 1D vector of size 3072 (32 * 32 * 3).

- **Dense Layers**: The model contains two fully connected layers (512 and 256 neurons) with ReLU activation, followed by an output layer with 10 neurons (for the 10 classes in CIFAR-10), using softmax activation to output a probability distribution.

---

### 4. Compiling the Model

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

- **Optimizer**: Adam optimizer is used for adjusting weights during training.
- **Loss Function**: Categorical cross-entropy is used because this is a multiclass classification problem.
- **Metrics**: Accuracy is used to measure performance during training.

---

### 5. Training the Model

history=model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

- **Epochs**: The model is trained for 10 epochs (full passes through the dataset).
- **Batch Size**: 32 samples are processed before updating the model's weights.
- **Validation Split**: 20% of the training data is used for validation to evaluate the model's performance on unseen data after each epoch.

**OUTPUT**:

```
Epoch 1/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 29s 22ms/step - accuracy: 0.2779 - loss: 2.0356 - val_accuracy: 0.3753 - val_loss: 1.7626
Epoch 2/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 30s 24ms/step - accuracy: 0.3787 - loss: 1.7169 - val_accuracy: 0.3776 - val_loss: 1.7254
Epoch 3/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 25s 20ms/step - accuracy: 0.4165 - loss: 1.6356 - val_accuracy: 0.4237 - val_loss: 1.6232
Epoch 4/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 26s 20ms/step - accuracy: 0.4350 - loss: 1.5715 - val_accuracy: 0.4239 - val_loss: 1.5936
Epoch 5/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 31s 24ms/step - accuracy: 0.4497 - loss: 1.5266 - val_accuracy: 0.4261 - val_loss: 1.5925
Epoch 6/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 28s 22ms/step - accuracy: 0.4619 - loss: 1.4988 - val_accuracy: 0.4320 - val_loss: 1.6152
Epoch 7/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 28s 22ms/step - accuracy: 0.4728 - loss: 1.4657 - val_accuracy: 0.4402 - val_loss: 1.5804
Epoch 8/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 28s 22ms/step - accuracy: 0.4816 - loss: 1.4453 - val_accuracy: 0.4791 - val_loss: 1.4981
Epoch 9/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 38s 30ms/step - accuracy: 0.4901 - loss: 1.4115 - val_accuracy: 0.4609 - val_loss: 1.5510
Epoch 10/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 39s 31ms/step - accuracy: 0.4987 - loss: 1.4057 - val_accuracy: 0.4758 - val_loss: 1.5307
```

Output logs show metrics like training accuracy (accuracy) and loss (loss), as well as validation accuracy (val_accuracy) and loss (val_loss).

---

### 6. Evaluating the Model

test_loss, test_accuracy = model.evaluate(x_test, y_test)

print(f'Test accuracy: {test_accuracy:.4f}')

After training, the model is evaluated on the test set to check how well it generalizes to completely unseen data.

---

**Model Summary**

```
model.summary()
```

```
Model: "sequential"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| flatten (Flatten) | (None, 3072) | 0 |
| dense (Dense) | (None, 512) | 1,573,376 |
| dense_1 (Dense) | (None, 256) | 131,328 |
| dense_2 (Dense) | (None, 10) | 2,570 |

```
Total params: 1,707,274 (6.51 MB)
Trainable params: 1,707,274 (6.51 MB)
Non-trainable params: 0 (0.00 B)
```

The model has 1,707,274 parameters in total, with layers and parameter counts shown in the model summary.

---

**7. Plot training & validation accuracy**

```
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='train_accuracy')

plt.plot(history.history['val_accuracy'], label='val_accuracy')

plt.title('Training and Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.show()
```

---

**Design and implement a CNN model to classify multi category tiff images with tensorflow /keras and check the accuracy. Check whether your model is overfit / underfit / perfect fit andapply the techniques to avoid overfit and underfit like regulizers, dropouts etc.**

To create a Convolutional Neural Network (CNN) model to classify multi-category .tiff images, we can use TensorFlow and Keras. Here's a step-by-step guide with code and explanations.

TIFF stands for Tagged Image File Format. TIFF files are commonly used in photo editing, graphic design, and publishing.

You can convert jpeg dataset to .tiff format if necessary.

To convert the CIFAR-10 dataset to .tiff images, you can use the PIL library to save each image from the dataset in .tiff format. Here's a step-by-step guide using Python.

**Steps to Convert CIFAR-10 to .tiff Format**

1. **Load CIFAR-10 Dataset**: CIFAR-10 is available in Keras, so we'll load it using tf.keras.datasets.
2. **Loop Through the Dataset**: Iterate over each image and label in the dataset.
3. **Save Each Image as .tiff**: Use the PIL.Image module to save each image in .tiff format.

   PIL stands for Python Imaging Library, and it's the original library that enabled Python to deal with images.

```
import os

import numpy as np

from tensorflow.keras.datasets import cifar10

from PIL import Image


# Create directories to save the images

train_dir = 'cifar10_tiff/train'

test_dir = 'cifar10_tiff/test'

os.makedirs(train_dir, exist_ok=True)

os.makedirs(test_dir, exist_ok=True)


# Load CIFAR-10 dataset

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
# Define class labels for CIFAR-10
class_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Helper function to save images in .tiff format
def save_images(images, labels, directory):
    for i, (image_array, label) in enumerate(zip(images, labels)):
        # Convert numpy array to PIL image
        image = Image.fromarray(image_array)

        # Define the label name
        label_name = class_labels[int(label)]

        # Create a subdirectory for each class
        label_dir = os.path.join(directory, label_name)
        os.makedirs(label_dir, exist_ok=True)

        # Save the image in .tiff format
        image_path = os.path.join(label_dir, f"{label_name}_{i}.tiff")
        image.save(image_path, format='TIFF')

# Save training and test images as .tiff
save_images(x_train, y_train, train_dir)
save_images(x_test, y_test, test_dir)

print("Images have been successfully saved as .tiff files.")
```

**Explanation of Code**

1. **Directory Setup**: cifar10_tiff/train and cifar10_tiff/test directories are created to store the converted images.
2. **Loading CIFAR-10**: cifar10.load_data() loads the training and test data.

3. **Saving Images**:
    - Image.fromarray(image_array): Converts the NumPy array (from CIFAR-10) into a PIL image.
    - image.save(image_path, format='TIFF'): Saves the image as .tiff in the specified directory.
4. **Class-based Organization**: Each class has its own subdirectory (e.g., train/airplane), organizing the data for easier use later.

After running this code, you will have all CIFAR-10 images saved in .tiff format, organized by class.

After converting the CIFAR-10 dataset images to .tiff format, you can load these .tiff images and use a CNN to classify them. Here's a complete code that demonstrates this:

**Step 1: Set Up Libraries and Import Modules**

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import os
```

**Step 2: Define Data Directories and Data Generator**

Assuming that the .tiff images are saved in folders like cifar10_tiff/train and cifar10_tiff/test, with subdirectories for each class (e.g., airplane, automobile, etc.).

```
# Set paths

train_dir = 'cifar10_tiff/train'

test_dir = 'cifar10_tiff/test'

# Define ImageDataGenerator with data augmentation for the training set

train_datagen = ImageDataGenerator(

    rescale=1.0/255.0,

    rotation_range=15,

    width_shift_range=0.1,

    height_shift_range=0.1,

    horizontal_flip=True,

    validation_split=0.2  # Split 20% of training data for validation
```

```
)

# Define ImageDataGenerator for the test set (only rescaling)
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Load training data
train_data = train_datagen.flow_from_directory(
    directory=train_dir,
    target_size=(32, 32),  # CIFAR-10 images are 32x32 pixels
    batch_size=32,
    class_mode='categorical',
    subset='training')

# Load validation data
validation_data = train_datagen.flow_from_directory(
    directory=train_dir,
    target_size=(32, 32),
    batch_size=32,
    class_mode='categorical',
    subset='validation')

# Load test data
test_data = test_datagen.flow_from_directory(
    directory=test_dir,
    target_size=(32, 32),
    batch_size=32,
    class_mode='categorical',
    shuffle=False)
```

**Step 3: Build the CNN Model**

Define a CNN model suitable for classifying the CIFAR-10 images.

```
model = Sequential([Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
   MaxPooling2D((2, 2)),Dropout(0.25),Conv2D(64, (3, 3), activation='relu'),
   MaxPooling2D((2, 2)),Dropout(0.25),Conv2D(128, (3, 3), activation='relu'),
   MaxPooling2D((2, 2)),Dropout(0.25),Flatten(),
   Dense(256, activation='relu'), Dropout(0.5),
   Dense(10, activation='softmax')  # 10 classes for CIFAR-10])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

**Step 4: Train the Model with Early Stopping**

Train the model on the .tiff images, using early stopping to avoid overfitting.

```
from tensorflow.keras.callbacks import EarlyStopping
# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = model.fit(train_data, validation_data=validation_data,epochs=50,batch_size=32,
callbacks=[early_stopping])
```

- **train_data**: The model will use batches of 32 images from the training data (as defined in the data generator).
- **validation_data**: After each epoch, the model will evaluate its performance on the validation data.
- **epochs=50**: The model will go through the entire dataset 50 times unless early stopping stops it earlier.
- **early_stopping**: If the validation loss or accuracy stops improving for a set number of epochs, training will stop early to prevent overfitting.

**Step 5: Evaluate the Model**

Check model performance on the test dataset and plot accuracy and loss.

```
# Evaluate on test data
test_loss, test_accuracy = model.evaluate(test_data)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

```
# Plot training history
import matplotlib.pyplot as plt

# Plot accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

plt.show()
```

**Step 6: Make Predictions (Optional)**

You can use the trained model to make predictions on individual images or batches of images.

```
# Get predictions
predictions = model.predict(test_data)
predicted_classes = tf.argmax(predictions, axis=1)
```

```
# Actual classes from the test data generator
true_classes = test_data.classes


# Accuracy by comparing predicted and actual classes
accuracy = np.mean(predicted_classes == true_classes)
print(f"Prediction accuracy on test set: {accuracy * 100:.2f}%")
```

**Explanation of Code**

1. **Data Generators**: The ImageDataGenerator instances perform data augmentation and split data into training, validation, and test sets.
2. **Model Definition**: The CNN model uses three convolutional layers with max pooling and dropout to prevent overfitting.
3. **Early Stopping**: Early stopping halts training if validation loss doesn't improve for a specified number of epochs.
4. **Evaluation and Visualization**: Plot accuracy and loss to monitor training progress and check for overfitting or underfitting.