Introduction to convnets (CNNs)

- A simple convnet example

In [2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # input shape: (1, 28, 28)
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3)

    def forward(self, x):
        x = F.relu(self.conv1(x))      # (batch, 32, 26, 26)
        x = self.pool1(x)              # (batch, 32, 13, 13)
        x = F.relu(self.conv2(x))      # (batch, 64, 11, 11)
        x = self.pool2(x)              # (batch, 64, 5, 5)
        x = F.relu(self.conv3(x))      # (batch, 64, 3, 3)
        return x

# Example usage
model = SimpleCNN()
print(model)
```

```
SimpleCNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
)
```

- CNN takes as input tensors of shape `(image_channels, image_height, image_width)` (not including the batch dimension).

In [4]:
```
from torchsummary import summary

model = SimpleCNN().cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(1, 28, 28), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 26, 26]             320
         MaxPool2d-2          [-1, 32, 13, 13]               0
            Conv2d-3          [-1, 64, 11, 11]          18,496
         MaxPool2d-4            [-1, 64, 5, 5]               0
            Conv2d-5            [-1, 64, 3, 3]          36,928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.28
Params size (MB): 0.21
Estimated Total Size (MB): 0.50
----------------------------------------------------------------
```

- Note that the output of every `Conv2d` and `MaxPooling2d` layer is a 3D tensor of shape `(channels, height, width)`.
- The width and height dimensions tend to shrink as you go deeper in the network.

In [5]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNNWithFC(nn.Module):
    def __init__(self):
        super(SimpleCNNWithFC, self).__init__()
        # feature extractor (conv part)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3)
        # classifier (flatten + dense layers)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(64 * 3 * 3, 64)  # input dim depends on previous conv
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = F.relu(self.conv3(x))
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Example usage
```

```
model = SimpleCNNWithFC()
print(model)
```

```
SimpleCNNWithFC(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=576, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

In [7]:
```
from torchsummary import summary

model = SimpleCNNWithFC().cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(1, 28, 28), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 26, 26]             320
         MaxPool2d-2           [-1, 32, 13, 13]               0
            Conv2d-3           [-1, 64, 11, 11]          18,496
         MaxPool2d-4             [-1, 64, 5, 5]               0
            Conv2d-5             [-1, 64, 3, 3]          36,928
           Flatten-6                  [-1, 576]               0
            Linear-7                   [-1, 64]          36,928
            Linear-8                   [-1, 10]             650
================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.29
Params size (MB): 0.36
Estimated Total Size (MB): 0.65
----------------------------------------------------------------
```

- Training the CNN on the MNIST digits

In [8]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# === 1. Dataset ===
# Automatically downloads and normalizes MNIST
```

```
transform = transforms.Compose([
    transforms.ToTensor(),                # [0,255] → [0,1]
])

train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=
test_dataset  = datasets.MNIST(root="./data", train=False, download=True, transform

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=64, shuffle=False)

# === 2. Model, Loss, Optimizer ===
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = SimpleCNNWithFC().to(device)     # from your previous definition
criterion = nn.CrossEntropyLoss()        # uses logits directly
optimizer = optim.RMSprop(model.parameters(), lr=1e-3)

# === 3. Training Loop ===
EPOCHS = 5

for epoch in range(EPOCHS):
    model.train()
    running_loss, correct, total = 0.0, 0, 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        # forward + backward + optimize
        logits = model(images)
        loss = criterion(logits, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        _, predicted = torch.max(logits, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    train_loss = running_loss / total
    train_acc = correct / total
    print(f"Epoch [{epoch+1}/{EPOCHS}]  Loss: {train_loss:.4f}  Acc: {train_acc:.4f

print("Training complete.")
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 19.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 508kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 4.70MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 4.31MB/s]
Epoch [1/5]  Loss: 0.1919  Acc: 0.9387
Epoch [2/5]  Loss: 0.0571  Acc: 0.9824
Epoch [3/5]  Loss: 0.0388  Acc: 0.9878
Epoch [4/5]  Loss: 0.0300  Acc: 0.9906
Epoch [5/5]  Loss: 0.0230  Acc: 0.9926
Training complete.
```

In [9]:
```
model.eval()
correct, total = 0, 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test accuracy: {100 * correct / total:.2f}%")
```

```
Test accuracy: 98.91%
```

The convolution operation

- The difference between a linear layer and a convolution layer
  - `Linear` layers learn global patterns in their input feature space.
  - Convolution layers learn local patterns: in the case of images, patterns found in small 2D windows of the inputs.
- Convolutions operate over 3D tensors, called *feature maps*, with two spatial axes (*height* and *width*) as well as *depth* axis (also called the *channels* axis).
  - For an RGB image, the dimension of the depth axis is 3.
- The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an *output feature map*.
  - The output feature map is still a 3D tensor with user-specified depth.
  - The different channels in that depth axis stand for *filters*.
- Convolutions are defined by two key parameters.
  - *Size of the patches extracted from the inputs*: Typically, 3*3 or 5*5
  - *Depth of the output feature map*
- A convolution works by sliding the windows of size 3*3 or 5*5 over the 3D input feature map.
  - At every possible location, it extracts the 3D patch of surrounding features, then transforms (via a tensor product with the same learned weight matrix, called the *convolution kernel*) 3D patch into a 1D vector of shape `(output_depth,)`.
  - All of these vectors are then spatially reassembled into a 3D output map of shape `(output_depth, height, width)`.
  - Every spatial location in the output feature map corresponds to the same location in the input feature map.

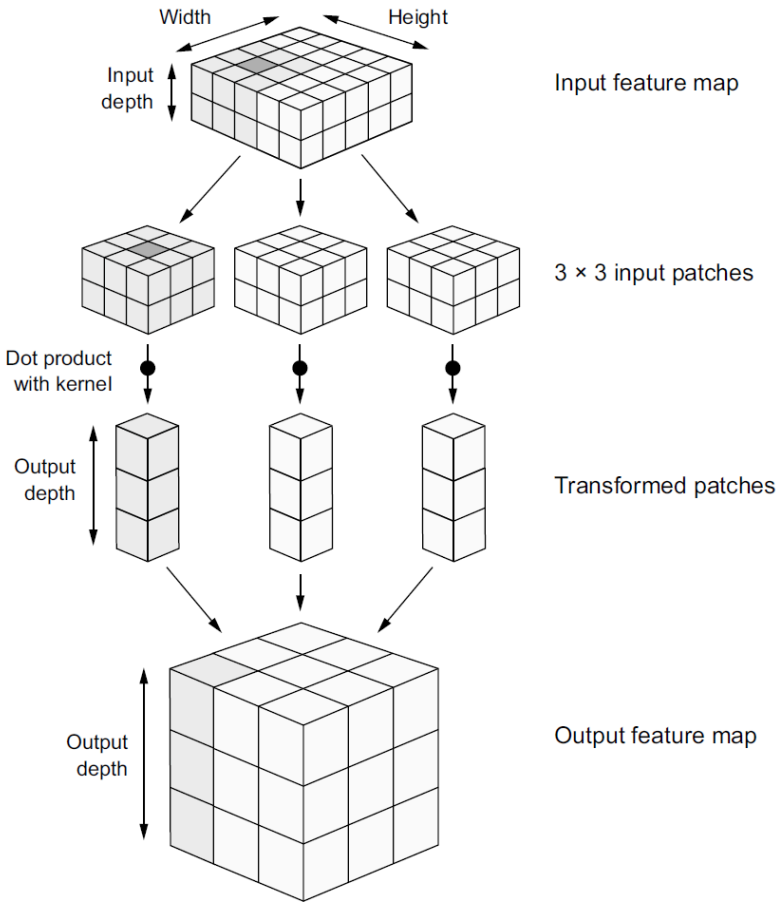


Figure 8.4 How convolution works

- Note that the output width and height may differ from the input width and height.
  - Border effects
  - The use of *strides*

- **Border effects and padding**
  - In `Conv2d` layers, padding is configurable via the `padding` argument, which can be either a string { `valid` , `same` } or an int giving the amount of implicit padding applied on both sides.

- **Convolution strides**
  - The distance between two successive windows is a parameter of the convolution, called its *stride*.
  - It is possible to have *strided convolutions*: convolutions with a stride higher than 1.
  - Using stride 2 means that the width and height of the feature map are downsampled by a factor of 2.
  - To downsample feature maps, we can also use the *max-pooling* operations.

The max-pooling operation

- Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel.
  - It is conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they are transformed via a hardcoded `max` tensor operation.
- Max pooling is usually done with 2*2 windows and stride 2, in order to downsample the feature maps by a factor of 2.
- Why downsample feature maps?
  - The reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows.
- Max pooling is not the only way to downsampling.
  - Strided convolutions, average pooling, etc.

Training a convnet from scratch on a small dataset

- Having to train an image-classification model using very little data is a common situation.
- Here, we will review several strategies to tackle the small dataset problem.
  - Data augmentation
  - Feature extraction with a pretrained network
  - Fine-tuning a pretrained network
- The Dogs vs. Cats dataset (https://www.kaggle.com/c/dogs-vs-cats/data)
  - Download URL: https://drive.google.com/uc?id=1AmgANN-SJmCMtLs6CTsZOyY9_W5DVCMT
  - Medium-resolution color JPEGs
  - 25,000 images of dogs and cats (12,500 from each class)
  - We will use a subset of this dataset.
    - A training set with 1,000 samples of each class
    - A validation set with 500 samples of each class
    - A test set with 500 samples of each class
- **Load the dataset**

In [46]:
```
# mount Google Drive
from google.colab import drive
drive.mount('/content/gdrive')

# unzip
import zipfile, os, shutil

dataset = '/content/gdrive/My Drive/Lectures/deep-learning/datasets/dogs_vs_cats_su
dst_path = '/content/dogs_vs_cats_subset'
dst_file = os.path.join(dst_path, 'dogs_vs_cats_subset.zip')

if not os.path.exists(dst_path):
  os.makedirs(dst_path)

# copy zip file
shutil.copy(dataset, dst_file)

with zipfile.ZipFile(dst_file, 'r') as file:
  file.extractall(dst_path)
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call driv
e.mount("/content/gdrive", force_remount=True).
```

In [2]:
```
!pwd
```

```
/content
```

In [3]:
```
!ls -al
```

```
total 24
drwxr-xr-x 1 root root 4096 Oct 20 03:46 .
drwxr-xr-x 1 root root 4096 Oct 20 03:44 ..
drwxr-xr-x 4 root root 4096 Oct 16 13:41 .config
drwxr-xr-x 4 root root 4096 Oct 20 03:46 dogs_vs_cats_subset
drwx------ 5 root root 4096 Oct 20 03:46 gdrive
drwxr-xr-x 1 root root 4096 Oct 16 13:41 sample_data
```

In [4]:
```
%cd dogs_vs_cats_subset/
```

```
/content/dogs_vs_cats_subset
```

In [5]:
```
!ls -al
```

```
total 88756
drwxr-xr-x 4 root root     4096 Oct 20 03:46 .
drwxr-xr-x 1 root root     4096 Oct 20 03:46 ..
-rw------- 1 root root 90863632 Oct 20 03:46 dogs_vs_cats_subset.zip
drwxr-xr-x 3 root root     4096 Oct 20 03:46 __MACOSX
drwxr-xr-x 5 root root     4096 Oct 20 03:46 subset
```

In [6]:
```
%cd subset
```

```
/content/dogs_vs_cats_subset/subset
```

In [7]:
```
!ls -al
```

```
total 36
drwxr-xr-x 5 root root  4096 Oct 20 03:46 .
drwxr-xr-x 4 root root  4096 Oct 20 03:46 ..
-rw-r--r-- 1 root root 12292 Oct 20 03:46 .DS_Store
drwxr-xr-x 4 root root  4096 Oct 20 03:46 test
drwxr-xr-x 4 root root  4096 Oct 20 03:46 train
drwxr-xr-x 4 root root  4096 Oct 20 03:46 validation
```

In [8]:
```
%cd train
```

```
/content/dogs_vs_cats_subset/subset/train
```

In [9]:
```
!ls -al
```

```
total 92
drwxr-xr-x 4 root root  4096 Oct 20 03:46 .
drwxr-xr-x 5 root root  4096 Oct 20 03:46 ..
drwxr-xr-x 2 root root 36864 Oct 20 03:46 cats
drwxr-xr-x 2 root root 36864 Oct 20 03:46 dogs
-rw-r--r-- 1 root root  8196 Oct 20 03:46 .DS_Store
```

In [47]:
```
train_cats_dir = os.path.join(dst_path, 'subset/train/cats')
train_dogs_dir = os.path.join(dst_path, 'subset/train/dogs')

validation_cats_dir = os.path.join(dst_path, 'subset/validation/cats')
validation_dogs_dir = os.path.join(dst_path, 'subset/validation/dogs')

test_cats_dir = os.path.join(dst_path, 'subset/test/cats')
test_dogs_dir = os.path.join(dst_path, 'subset/test/dogs')

print('total training cat images:', len(os.listdir(train_cats_dir)))
print('total training dog images:', len(os.listdir(train_dogs_dir)))

print('total validation cat images:', len(os.listdir(validation_cats_dir)))
print('total validation dog images:', len(os.listdir(validation_dogs_dir)))

print('total test cat images:', len(os.listdir(test_cats_dir)))
print('total test dog images:', len(os.listdir(test_dogs_dir)))
```

```
total training cat images: 1000
total training dog images: 1000
total validation cat images: 500
total validation dog images: 500
total test cat images: 500
total test dog images: 500
```

- **Building the network**
  - Note that we are dealing with bigger images and a more complex problem than MNIST.
  - We will make the network larger.
  - Here, we start from inputs of size 150*150, and end up with feature maps of size 7*7 just before the `Flatten` layer.
  - The depth of the feature maps progressively increases in the network, whereas the size of the feature maps decreases. This is a pattern you'll see in almost all CNNs.

In [17]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class BinaryCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3,   32, kernel_size=3), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(32,  64, kernel_size=3), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128,128, kernel_size=3), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128*7*7, 512), nn.ReLU(inplace=True),
            nn.Linear(512, 1)  # logits (no sigmoid)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)  # shape: (N, 1)
        return x
```

In [18]:
```python
from torchsummary import summary

model = BinaryCNN().cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(3, 150, 150), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 148, 148]             896
              ReLU-2         [-1, 32, 148, 148]               0
         MaxPool2d-3           [-1, 32, 74, 74]               0
            Conv2d-4           [-1, 64, 72, 72]          18,496
              ReLU-5           [-1, 64, 72, 72]               0
         MaxPool2d-6           [-1, 64, 36, 36]               0
            Conv2d-7          [-1, 128, 34, 34]          73,856
              ReLU-8          [-1, 128, 34, 34]               0
         MaxPool2d-9          [-1, 128, 17, 17]               0
           Conv2d-10          [-1, 128, 15, 15]         147,584
             ReLU-11          [-1, 128, 15, 15]               0
        MaxPool2d-12            [-1, 128, 7, 7]               0
          Flatten-13                 [-1, 6272]               0
           Linear-14                  [-1, 512]       3,211,776
             ReLU-15                  [-1, 512]               0
           Linear-16                    [-1, 1]             513
================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.26
Forward/backward pass size (MB): 20.81
Params size (MB): 13.17
Estimated Total Size (MB): 34.24
----------------------------------------------------------------
```

- **Data preprocessing**
  - Currently, the data is stored on a drive as JPEG files. So we need the following steps:
    - Read the image files from disk.
    - Decode the JPEG content to RGB pixel tensors.
    - Convert them to floating-point tensors.
    - Rescale the pixel values (between 0 and 255) to the `[0,1]` inverval.
  - In PyTorch, these steps are handeled by the `torchvision.datasets.ImageFolder` class together with `torchvision.transforms`, which provides convenient image preprocessing tools.
  - `ImageFolder` automatically assigns labels based on subdirectory names (e.g., `train/cats/`, `train/dogs/`), and `DataLoader` helps create batches of preprocessed tensors for training and validation.

In [19]:
```python
import os
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

```
# Directory paths
train_dir = os.path.join(dst_path, 'subset/train')
validation_dir = os.path.join(dst_path, 'subset/validation')

# Transforms: convert image → tensor and rescale to [0,1]
train_transform = transforms.Compose([
    transforms.Resize((150, 150)),  # same as target_size=(150,150)
    transforms.ToTensor(),          # scales automatically to [0,1]
])

val_transform = transforms.Compose([
    transforms.Resize((150, 150)),
    transforms.ToTensor(),
])

# Datasets
train_dataset = datasets.ImageFolder(train_dir, transform=train_transform)
val_dataset   = datasets.ImageFolder(validation_dir, transform=val_transform)

# DataLoaders (equivalent to flow_from_directory)
train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=20, shuffle=False)

# Optional: check class names and counts
print("Classes:", train_dataset.classes)
print("Number of training batches:", len(train_loader))
print("Number of validation batches:", len(val_loader))
```

```
Classes: ['cats', 'dogs']
Number of training batches: 100
Number of validation batches: 50
```

In [20]:
```
for data_batch, labels_batch in train_loader:
  print('data batch shape:', data_batch.shape)
  print('labels batch shape:', labels_batch.shape)
  break
```

```
data batch shape: torch.Size([20, 3, 150, 150])
labels batch shape: torch.Size([20])
```

In [21]:
```
labels_batch
```

Out[21]:
```
tensor([1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0])
```

In [22]:
```
print(data_batch[0].shape, data_batch[0])
```

```
torch.Size([3, 150, 150]) tensor([[[0.1451, 0.1490, 0.1608,  ..., 0.1373, 0.1490, 0.
1333],
         [0.1529, 0.1569, 0.1608,  ..., 0.1176, 0.1294, 0.1294],
         [0.1725, 0.1608, 0.1647,  ..., 0.0941, 0.1137, 0.1176],
         ...,
         [0.1098, 0.1176, 0.1216,  ..., 0.1804, 0.1882, 0.1882],
         [0.0941, 0.1020, 0.1216,  ..., 0.1804, 0.1922, 0.2000],
         [0.0902, 0.0980, 0.1294,  ..., 0.1804, 0.1922, 0.1961]],

        [[0.1804, 0.1686, 0.1569,  ..., 0.0824, 0.0549, 0.0627],
         [0.1569, 0.1529, 0.1569,  ..., 0.0745, 0.0549, 0.0627],
         [0.1373, 0.1333, 0.1529,  ..., 0.0588, 0.0627, 0.0588],
         ...,
         [0.0941, 0.0941, 0.0784,  ..., 0.0902, 0.1020, 0.1020],
         [0.0784, 0.0745, 0.0745,  ..., 0.0941, 0.1059, 0.1137],
         [0.0745, 0.0706, 0.0863,  ..., 0.0941, 0.1059, 0.1098]],

        [[0.1647, 0.2275, 0.2784,  ..., 0.0627, 0.0118, 0.0118],
         [0.1333, 0.2000, 0.2667,  ..., 0.0588, 0.0235, 0.0235],
         [0.1059, 0.1608, 0.2431,  ..., 0.0549, 0.0471, 0.0314],
         ...,
         [0.0588, 0.0627, 0.0549,  ..., 0.0157, 0.0235, 0.0196],
         [0.0431, 0.0431, 0.0510,  ..., 0.0118, 0.0235, 0.0314],
         [0.0392, 0.0392, 0.0627,  ..., 0.0118, 0.0235, 0.0275]]])
```

- Training the model

In [23]:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Assume model, train_loader, val_loader already defined
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BinaryCNN().to(device)

criterion = nn.BCEWithLogitsLoss()  # since output is a single logit
optimizer = optim.RMSprop(model.parameters(), lr=1e-4)

EPOCHS = 30

for epoch in range(EPOCHS):
    model.train()
    running_loss, total, correct = 0.0, 0, 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.float().to(device).view(-1, 1)

        optimizer.zero_grad()
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        preds = (torch.sigmoid(logits) > 0.5).float()
```

```
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    train_loss = running_loss / total
    train_acc = correct / total

    # ---- Validation ----
    model.eval()
    val_loss, val_correct, val_total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.float().to(device).view(-1,
            logits = model(images)
            loss = criterion(logits, labels)
            val_loss += loss.item() * images.size(0)
            preds = (torch.sigmoid(logits) > 0.5).float()
            val_correct += (preds == labels).sum().item()
            val_total += labels.size(0)

    val_loss /= val_total
    val_acc = val_correct / val_total

    print(f"Epoch [{epoch+1}/{EPOCHS}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
```

```
Epoch [1/30] Train Loss: 0.6974, Train Acc: 0.5090, Val Loss: 0.6924, Val Acc: 0.500
0
Epoch [2/30] Train Loss: 0.6895, Train Acc: 0.5420, Val Loss: 0.6832, Val Acc: 0.542
0
Epoch [3/30] Train Loss: 0.6755, Train Acc: 0.5790, Val Loss: 0.6460, Val Acc: 0.662
0
Epoch [4/30] Train Loss: 0.6524, Train Acc: 0.6290, Val Loss: 0.6326, Val Acc: 0.637
0
Epoch [5/30] Train Loss: 0.6358, Train Acc: 0.6360, Val Loss: 0.6332, Val Acc: 0.639
0
Epoch [6/30] Train Loss: 0.6225, Train Acc: 0.6595, Val Loss: 0.6141, Val Acc: 0.649
0
Epoch [7/30] Train Loss: 0.6069, Train Acc: 0.6830, Val Loss: 0.6344, Val Acc: 0.623
0
Epoch [8/30] Train Loss: 0.5992, Train Acc: 0.6770, Val Loss: 0.6051, Val Acc: 0.668
0
Epoch [9/30] Train Loss: 0.5862, Train Acc: 0.6995, Val Loss: 0.6084, Val Acc: 0.674
0
Epoch [10/30] Train Loss: 0.5714, Train Acc: 0.7135, Val Loss: 0.5879, Val Acc: 0.69
10
Epoch [11/30] Train Loss: 0.5510, Train Acc: 0.7265, Val Loss: 0.5712, Val Acc: 0.69
60
Epoch [12/30] Train Loss: 0.5334, Train Acc: 0.7455, Val Loss: 0.5589, Val Acc: 0.71
80
Epoch [13/30] Train Loss: 0.5107, Train Acc: 0.7505, Val Loss: 0.5529, Val Acc: 0.71
60
Epoch [14/30] Train Loss: 0.4951, Train Acc: 0.7640, Val Loss: 0.5528, Val Acc: 0.72
50
Epoch [15/30] Train Loss: 0.4850, Train Acc: 0.7735, Val Loss: 0.5631, Val Acc: 0.71
80
Epoch [16/30] Train Loss: 0.4637, Train Acc: 0.7815, Val Loss: 0.5455, Val Acc: 0.71
60
Epoch [17/30] Train Loss: 0.4571, Train Acc: 0.7845, Val Loss: 0.5386, Val Acc: 0.73
60
Epoch [18/30] Train Loss: 0.4353, Train Acc: 0.7975, Val Loss: 0.5310, Val Acc: 0.72
90
Epoch [19/30] Train Loss: 0.4320, Train Acc: 0.8070, Val Loss: 0.5571, Val Acc: 0.73
00
Epoch [20/30] Train Loss: 0.4095, Train Acc: 0.8170, Val Loss: 0.5691, Val Acc: 0.72
00
Epoch [21/30] Train Loss: 0.3975, Train Acc: 0.8195, Val Loss: 0.5875, Val Acc: 0.72
20
Epoch [22/30] Train Loss: 0.3826, Train Acc: 0.8330, Val Loss: 0.5596, Val Acc: 0.72
00
Epoch [23/30] Train Loss: 0.3693, Train Acc: 0.8355, Val Loss: 0.6697, Val Acc: 0.69
80
Epoch [24/30] Train Loss: 0.3570, Train Acc: 0.8420, Val Loss: 0.5618, Val Acc: 0.71
90
Epoch [25/30] Train Loss: 0.3326, Train Acc: 0.8515, Val Loss: 0.5744, Val Acc: 0.75
50
Epoch [26/30] Train Loss: 0.3237, Train Acc: 0.8620, Val Loss: 0.5972, Val Acc: 0.72
10
Epoch [27/30] Train Loss: 0.3153, Train Acc: 0.8550, Val Loss: 0.5676, Val Acc: 0.75
50
Epoch [28/30] Train Loss: 0.2929, Train Acc: 0.8775, Val Loss: 0.5721, Val Acc: 0.74
40
```

```
Epoch [29/30] Train Loss: 0.2780, Train Acc: 0.8855, Val Loss: 0.6418, Val Acc: 0.71
30
Epoch [30/30] Train Loss: 0.2657, Train Acc: 0.8950, Val Loss: 0.6125, Val Acc: 0.72
80
```

In [24]:
```
torch.save(model.state_dict(), 'cats_and_dogs_subset_1.pth')
```

- **Data augmentation**
  - Overfitting is caused by having too few samples to learn from, rendering you unable to train a model that can generalize to new data.
  - Data augmentation takes the approach of generating more training data from existing training samples, by *augmenting* the samples via a number of random transformations.
  - In PyTorch, image augmentation is handled with `torchvision.transforms`.

In [48]:
```
from torchvision import transforms

train_transform = transforms.Compose([
    transforms.RandomRotation(40),              # rotation_range=40
    transforms.RandomResizedCrop(150, scale=(0.8, 1.0)),  # approximate zoom/shift
    transforms.RandomAffine(
        degrees=0,
        shear=20,                               # shear_range=0.2 (≈20 degrees)
    ),
    transforms.RandomHorizontalFlip(p=0.5),     # horizontal_flip=True
    transforms.ToTensor(),                      # converts [0,255] → [0,1]
])

# Validation/test set should not be augmented
val_transform = transforms.Compose([
    transforms.Resize((150, 150)),
    transforms.ToTensor(),
])

# --- Datasets ---
train_dataset = datasets.ImageFolder(train_dir, transform=train_transform)
val_dataset   = datasets.ImageFolder(validation_dir,   transform=val_transform)

# --- DataLoaders ---
train_loader = DataLoader(train_dataset, batch_size=20, shuffle=True,  num_workers=
val_loader   = DataLoader(val_dataset,   batch_size=20, shuffle=False, num_workers=

print("Classes:", train_dataset.classes)
print("Train batches:", len(train_loader), "| Val batches:", len(val_loader))
```

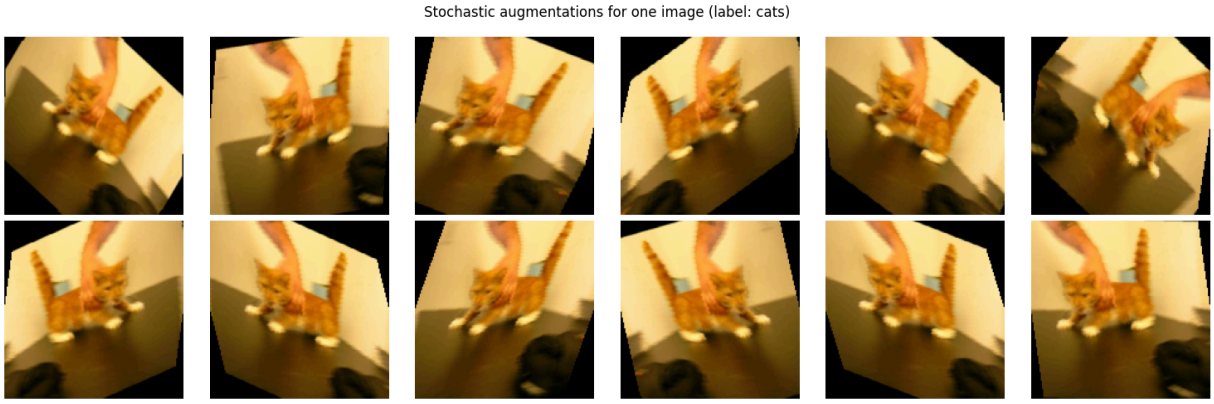
```
Classes: ['cats', 'dogs']
Train batches: 100 | Val batches: 50
```

In [26]:
```
import torch
import matplotlib.pyplot as plt
```

```
import torchvision

def show_augmented_batch(dataloader, class_names, nmax=16):
    model_was_training = False  # just a placeholder if you reuse this in training
    xb, yb = next(iter(dataloader))  # a single augmented batch
    k = min(nmax, xb.size(0))
    grid = torchvision.utils.make_grid(xb[:k], nrow=8, padding=2)  # [C,H,W] -> gri
    plt.figure(figsize=(12, 6))
    plt.imshow(grid.permute(1, 2, 0))  # CHW -> HWC for matplotlib
    # Title: show class names for the first few images
    titles = [class_names[yb[i].item()] for i in range(k)]
    top = " | ".join(titles[:8])
    bottom = " | ".join(titles[8:16]) if k > 8 else ""
    plt.title(top + ("\n" + bottom if bottom else ""))
    plt.axis("off")
    plt.show()

show_augmented_batch(train_loader, train_dataset.classes, nmax=16)
```

In [27]:

```
from PIL import Image
import matplotlib.pyplot as plt

def show_many_augments_of_one(idx=0, times=12):
    # Get the raw file path (ImageFolder stores (path, label) pairs in .samples)
    path, label = train_dataset.samples[idx]
    img = Image.open(path).convert("RGB")

    # Apply the *train* transform multiple times to the same image
    imgs = [train_transform(img) for _ in range(times)]  # tensors in [0,1]

    # Plot
    cols = 6
    rows = (times + cols - 1) // cols
    plt.figure(figsize=(2.5*cols, 2.5*rows))
    for i, t in enumerate(imgs):
        plt.subplot(rows, cols, i+1)
        plt.imshow(t.permute(1, 2, 0))  # CHW -> HWC
        plt.axis("off")
    plt.suptitle(f"Stochastic augmentations for one image (label: {train_dataset.cl
    plt.tight_layout()
    plt.show()

show_many_augments_of_one(idx=0, times=12)
```



In [30]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Model (same structure as earlier)
model = BinaryCNN().to(device)

# Loss & Optimizer (same setting as Keras example)
criterion = nn.BCEWithLogitsLoss()          # because model outputs a single logit
optimizer = optim.RMSprop(model.parameters(), lr=1e-4)

EPOCHS = 100

# Optional: history for plotting later
history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}

best_val_acc = 0.0
best_state = None

for epoch in range(1, EPOCHS + 1):
    # ---------- Train ----------
    model.train()
    running_loss, running_correct, total = 0.0, 0, 0

    for images, labels in train_loader:
        images = images.to(device)
        # ImageFolder returns class indices {0,1}; cast to float and shape (N,1)
        labels = labels.to(device).float().view(-1, 1)

        optimizer.zero_grad()
        logits = model(images)                          # (N,1)
        loss = criterion(logits, labels)                # BCEWithLogitsLoss

        loss.backward()
        optimizer.step()

        # metrics
        with torch.no_grad():
            preds = (torch.sigmoid(logits) >= 0.5).float()
            running_correct += (preds == labels).sum().item()
```

```
            running_loss += loss.item() * images.size(0)
            total += images.size(0)

    train_loss = running_loss / total
    train_acc = running_correct / total

    # ---------- Validation ----------
    model.eval()
    val_loss_sum, val_correct, val_total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device).float().view(-1, 1)

            logits = model(images)
            loss = criterion(logits, labels)
            preds = (torch.sigmoid(logits) >= 0.5).float()

            val_loss_sum += loss.item() * images.size(0)
            val_correct += (preds == labels).sum().item()
            val_total += images.size(0)

    val_loss = val_loss_sum / val_total
    val_acc = val_correct / val_total

    # Record history
    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)

    print(f"Epoch {epoch:02d}/{EPOCHS} | "
          f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} | "
          f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

    # Save best weights by validation accuracy
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().it

# Load best weights (optional but recommended)
if best_state is not None:
    model.load_state_dict(best_state)
    print(f"Loaded best model (val_acc={best_val_acc:.4f}).")
```

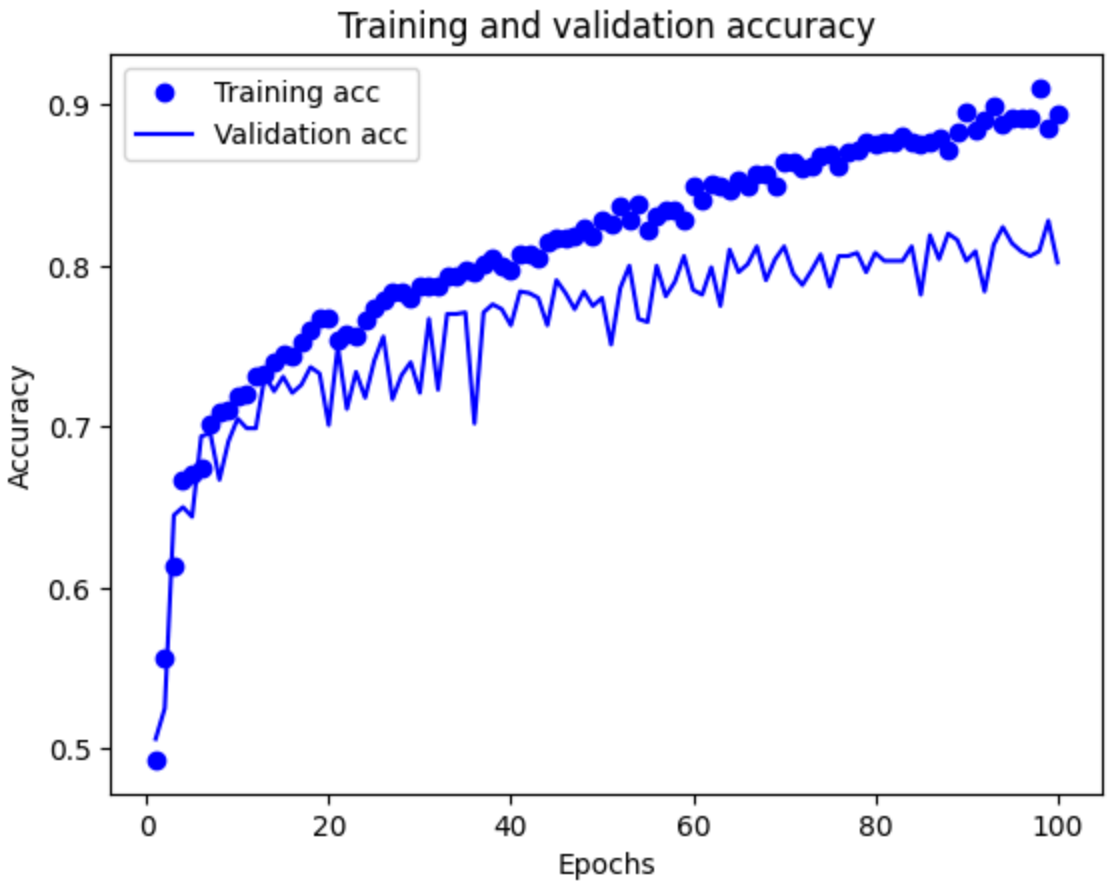
```
Epoch 01/100 | Train Loss: 0.6958 Acc: 0.4930 | Val Loss: 0.6920 Acc: 0.5060
Epoch 02/100 | Train Loss: 0.6830 Acc: 0.5560 | Val Loss: 0.6892 Acc: 0.5250
Epoch 03/100 | Train Loss: 0.6559 Acc: 0.6130 | Val Loss: 0.6374 Acc: 0.6450
Epoch 04/100 | Train Loss: 0.6283 Acc: 0.6665 | Val Loss: 0.6169 Acc: 0.6500
Epoch 05/100 | Train Loss: 0.6163 Acc: 0.6710 | Val Loss: 0.6232 Acc: 0.6440
Epoch 06/100 | Train Loss: 0.6075 Acc: 0.6745 | Val Loss: 0.5878 Acc: 0.6940
Epoch 07/100 | Train Loss: 0.5854 Acc: 0.7015 | Val Loss: 0.5776 Acc: 0.6960
Epoch 08/100 | Train Loss: 0.5697 Acc: 0.7095 | Val Loss: 0.6084 Acc: 0.6670
Epoch 09/100 | Train Loss: 0.5670 Acc: 0.7100 | Val Loss: 0.5734 Acc: 0.6910
Epoch 10/100 | Train Loss: 0.5546 Acc: 0.7190 | Val Loss: 0.5716 Acc: 0.7050
Epoch 11/100 | Train Loss: 0.5461 Acc: 0.7200 | Val Loss: 0.5511 Acc: 0.6990
Epoch 12/100 | Train Loss: 0.5384 Acc: 0.7320 | Val Loss: 0.5505 Acc: 0.6990
Epoch 13/100 | Train Loss: 0.5384 Acc: 0.7325 | Val Loss: 0.5400 Acc: 0.7320
Epoch 14/100 | Train Loss: 0.5352 Acc: 0.7405 | Val Loss: 0.5422 Acc: 0.7220
Epoch 15/100 | Train Loss: 0.5190 Acc: 0.7450 | Val Loss: 0.5276 Acc: 0.7310
Epoch 16/100 | Train Loss: 0.5200 Acc: 0.7435 | Val Loss: 0.5313 Acc: 0.7210
Epoch 17/100 | Train Loss: 0.5170 Acc: 0.7520 | Val Loss: 0.5347 Acc: 0.7260
Epoch 18/100 | Train Loss: 0.5040 Acc: 0.7605 | Val Loss: 0.5149 Acc: 0.7370
Epoch 19/100 | Train Loss: 0.5053 Acc: 0.7680 | Val Loss: 0.5198 Acc: 0.7330
Epoch 20/100 | Train Loss: 0.4953 Acc: 0.7670 | Val Loss: 0.5397 Acc: 0.7010
Epoch 21/100 | Train Loss: 0.5000 Acc: 0.7540 | Val Loss: 0.5055 Acc: 0.7500
Epoch 22/100 | Train Loss: 0.4950 Acc: 0.7575 | Val Loss: 0.5379 Acc: 0.7110
Epoch 23/100 | Train Loss: 0.4946 Acc: 0.7560 | Val Loss: 0.5140 Acc: 0.7340
Epoch 24/100 | Train Loss: 0.4809 Acc: 0.7665 | Val Loss: 0.5385 Acc: 0.7180
Epoch 25/100 | Train Loss: 0.4761 Acc: 0.7735 | Val Loss: 0.5073 Acc: 0.7410
Epoch 26/100 | Train Loss: 0.4704 Acc: 0.7785 | Val Loss: 0.4809 Acc: 0.7560
Epoch 27/100 | Train Loss: 0.4582 Acc: 0.7840 | Val Loss: 0.5349 Acc: 0.7170
Epoch 28/100 | Train Loss: 0.4668 Acc: 0.7840 | Val Loss: 0.5307 Acc: 0.7320
Epoch 29/100 | Train Loss: 0.4626 Acc: 0.7795 | Val Loss: 0.5027 Acc: 0.7400
Epoch 30/100 | Train Loss: 0.4500 Acc: 0.7870 | Val Loss: 0.5499 Acc: 0.7210
Epoch 31/100 | Train Loss: 0.4513 Acc: 0.7870 | Val Loss: 0.4722 Acc: 0.7670
Epoch 32/100 | Train Loss: 0.4500 Acc: 0.7875 | Val Loss: 0.5542 Acc: 0.7230
Epoch 33/100 | Train Loss: 0.4455 Acc: 0.7930 | Val Loss: 0.4710 Acc: 0.7700
Epoch 34/100 | Train Loss: 0.4499 Acc: 0.7930 | Val Loss: 0.4636 Acc: 0.7700
Epoch 35/100 | Train Loss: 0.4431 Acc: 0.7970 | Val Loss: 0.4703 Acc: 0.7710
Epoch 36/100 | Train Loss: 0.4342 Acc: 0.7955 | Val Loss: 0.5595 Acc: 0.7020
Epoch 37/100 | Train Loss: 0.4450 Acc: 0.8015 | Val Loss: 0.4531 Acc: 0.7710
Epoch 38/100 | Train Loss: 0.4269 Acc: 0.8045 | Val Loss: 0.4540 Acc: 0.7760
Epoch 39/100 | Train Loss: 0.4344 Acc: 0.8000 | Val Loss: 0.4632 Acc: 0.7730
Epoch 40/100 | Train Loss: 0.4282 Acc: 0.7970 | Val Loss: 0.4672 Acc: 0.7630
Epoch 41/100 | Train Loss: 0.4169 Acc: 0.8070 | Val Loss: 0.4553 Acc: 0.7840
Epoch 42/100 | Train Loss: 0.4181 Acc: 0.8075 | Val Loss: 0.4567 Acc: 0.7830
Epoch 43/100 | Train Loss: 0.4098 Acc: 0.8045 | Val Loss: 0.4529 Acc: 0.7800
Epoch 44/100 | Train Loss: 0.4050 Acc: 0.8150 | Val Loss: 0.4927 Acc: 0.7630
Epoch 45/100 | Train Loss: 0.4086 Acc: 0.8175 | Val Loss: 0.4373 Acc: 0.7910
Epoch 46/100 | Train Loss: 0.4028 Acc: 0.8170 | Val Loss: 0.4484 Acc: 0.7830
Epoch 47/100 | Train Loss: 0.4014 Acc: 0.8190 | Val Loss: 0.4586 Acc: 0.7730
Epoch 48/100 | Train Loss: 0.3909 Acc: 0.8240 | Val Loss: 0.4452 Acc: 0.7840
Epoch 49/100 | Train Loss: 0.3945 Acc: 0.8190 | Val Loss: 0.4520 Acc: 0.7750
Epoch 50/100 | Train Loss: 0.3859 Acc: 0.8290 | Val Loss: 0.4699 Acc: 0.7800
Epoch 51/100 | Train Loss: 0.3806 Acc: 0.8265 | Val Loss: 0.5150 Acc: 0.7510
Epoch 52/100 | Train Loss: 0.3734 Acc: 0.8370 | Val Loss: 0.4484 Acc: 0.7860
Epoch 53/100 | Train Loss: 0.3743 Acc: 0.8285 | Val Loss: 0.4697 Acc: 0.8000
Epoch 54/100 | Train Loss: 0.3729 Acc: 0.8380 | Val Loss: 0.5074 Acc: 0.7670
Epoch 55/100 | Train Loss: 0.3825 Acc: 0.8225 | Val Loss: 0.5254 Acc: 0.7650
Epoch 56/100 | Train Loss: 0.3738 Acc: 0.8315 | Val Loss: 0.4498 Acc: 0.8000
```

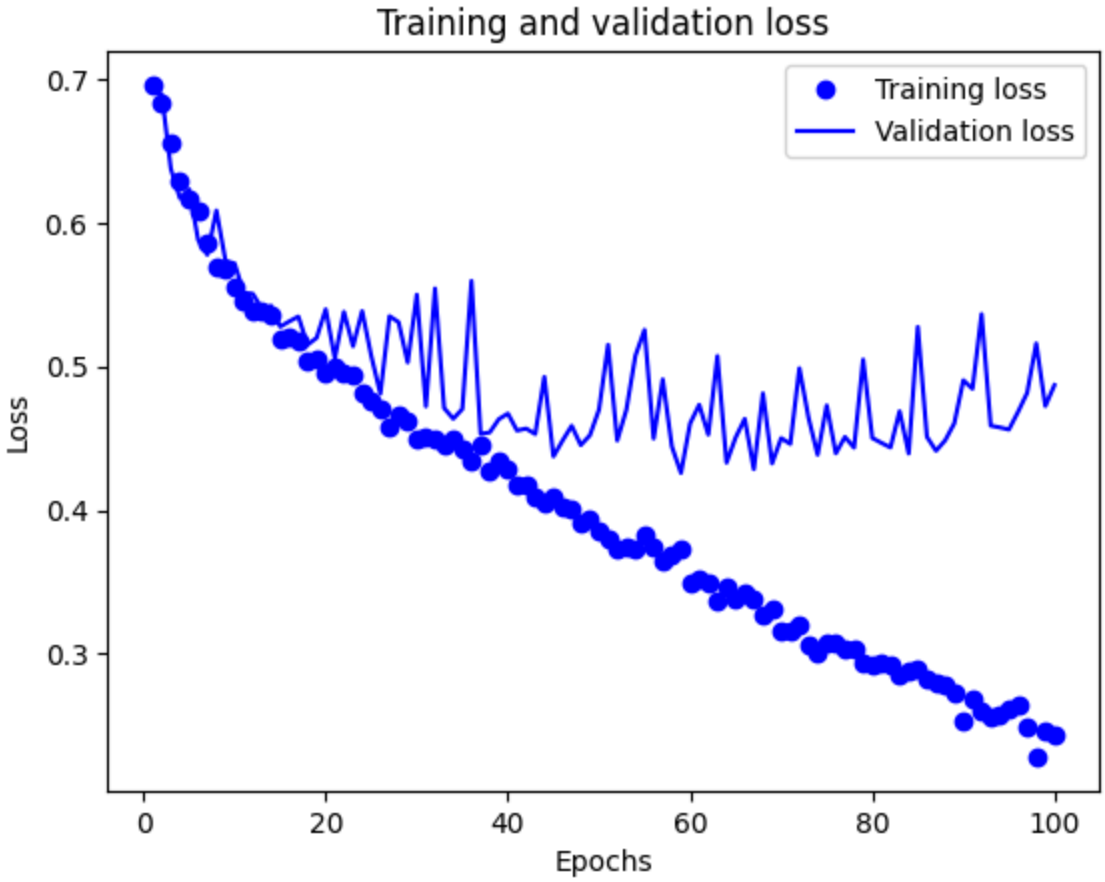
```
Epoch 57/100 | Train Loss: 0.3647 Acc: 0.8345 | Val Loss: 0.4909 Acc: 0.7810
Epoch 58/100 | Train Loss: 0.3688 Acc: 0.8350 | Val Loss: 0.4443 Acc: 0.7900
Epoch 59/100 | Train Loss: 0.3723 Acc: 0.8290 | Val Loss: 0.4258 Acc: 0.8060
Epoch 60/100 | Train Loss: 0.3495 Acc: 0.8490 | Val Loss: 0.4603 Acc: 0.7850
Epoch 61/100 | Train Loss: 0.3518 Acc: 0.8410 | Val Loss: 0.4733 Acc: 0.7820
Epoch 62/100 | Train Loss: 0.3494 Acc: 0.8510 | Val Loss: 0.4525 Acc: 0.7990
Epoch 63/100 | Train Loss: 0.3365 Acc: 0.8500 | Val Loss: 0.5072 Acc: 0.7750
Epoch 64/100 | Train Loss: 0.3460 Acc: 0.8465 | Val Loss: 0.4330 Acc: 0.8100
Epoch 65/100 | Train Loss: 0.3378 Acc: 0.8530 | Val Loss: 0.4505 Acc: 0.7960
Epoch 66/100 | Train Loss: 0.3422 Acc: 0.8490 | Val Loss: 0.4635 Acc: 0.8010
Epoch 67/100 | Train Loss: 0.3381 Acc: 0.8570 | Val Loss: 0.4285 Acc: 0.8120
Epoch 68/100 | Train Loss: 0.3270 Acc: 0.8565 | Val Loss: 0.4812 Acc: 0.7910
Epoch 69/100 | Train Loss: 0.3318 Acc: 0.8495 | Val Loss: 0.4325 Acc: 0.8040
Epoch 70/100 | Train Loss: 0.3153 Acc: 0.8645 | Val Loss: 0.4501 Acc: 0.8120
Epoch 71/100 | Train Loss: 0.3155 Acc: 0.8640 | Val Loss: 0.4460 Acc: 0.7950
Epoch 72/100 | Train Loss: 0.3206 Acc: 0.8605 | Val Loss: 0.4987 Acc: 0.7880
Epoch 73/100 | Train Loss: 0.3066 Acc: 0.8625 | Val Loss: 0.4644 Acc: 0.7970
Epoch 74/100 | Train Loss: 0.3007 Acc: 0.8685 | Val Loss: 0.4385 Acc: 0.8070
Epoch 75/100 | Train Loss: 0.3071 Acc: 0.8690 | Val Loss: 0.4729 Acc: 0.7870
Epoch 76/100 | Train Loss: 0.3074 Acc: 0.8625 | Val Loss: 0.4395 Acc: 0.8060
Epoch 77/100 | Train Loss: 0.3030 Acc: 0.8705 | Val Loss: 0.4509 Acc: 0.8060
Epoch 78/100 | Train Loss: 0.3027 Acc: 0.8725 | Val Loss: 0.4435 Acc: 0.8080
Epoch 79/100 | Train Loss: 0.2944 Acc: 0.8765 | Val Loss: 0.5047 Acc: 0.7960
Epoch 80/100 | Train Loss: 0.2927 Acc: 0.8755 | Val Loss: 0.4502 Acc: 0.8080
Epoch 81/100 | Train Loss: 0.2931 Acc: 0.8770 | Val Loss: 0.4468 Acc: 0.8030
Epoch 82/100 | Train Loss: 0.2924 Acc: 0.8770 | Val Loss: 0.4436 Acc: 0.8030
Epoch 83/100 | Train Loss: 0.2859 Acc: 0.8810 | Val Loss: 0.4688 Acc: 0.8030
Epoch 84/100 | Train Loss: 0.2875 Acc: 0.8765 | Val Loss: 0.4394 Acc: 0.8120
Epoch 85/100 | Train Loss: 0.2899 Acc: 0.8755 | Val Loss: 0.5275 Acc: 0.7820
Epoch 86/100 | Train Loss: 0.2822 Acc: 0.8770 | Val Loss: 0.4510 Acc: 0.8190
Epoch 87/100 | Train Loss: 0.2804 Acc: 0.8795 | Val Loss: 0.4412 Acc: 0.8040
Epoch 88/100 | Train Loss: 0.2787 Acc: 0.8720 | Val Loss: 0.4485 Acc: 0.8200
Epoch 89/100 | Train Loss: 0.2730 Acc: 0.8835 | Val Loss: 0.4603 Acc: 0.8160
Epoch 90/100 | Train Loss: 0.2535 Acc: 0.8960 | Val Loss: 0.4903 Acc: 0.8030
Epoch 91/100 | Train Loss: 0.2690 Acc: 0.8840 | Val Loss: 0.4844 Acc: 0.8090
Epoch 92/100 | Train Loss: 0.2605 Acc: 0.8905 | Val Loss: 0.5364 Acc: 0.7840
Epoch 93/100 | Train Loss: 0.2556 Acc: 0.8990 | Val Loss: 0.4587 Acc: 0.8130
Epoch 94/100 | Train Loss: 0.2579 Acc: 0.8875 | Val Loss: 0.4575 Acc: 0.8240
Epoch 95/100 | Train Loss: 0.2617 Acc: 0.8920 | Val Loss: 0.4560 Acc: 0.8140
Epoch 96/100 | Train Loss: 0.2650 Acc: 0.8920 | Val Loss: 0.4683 Acc: 0.8090
Epoch 97/100 | Train Loss: 0.2495 Acc: 0.8920 | Val Loss: 0.4815 Acc: 0.8060
Epoch 98/100 | Train Loss: 0.2284 Acc: 0.9105 | Val Loss: 0.5159 Acc: 0.8090
Epoch 99/100 | Train Loss: 0.2468 Acc: 0.8860 | Val Loss: 0.4723 Acc: 0.8280
Epoch 100/100 | Train Loss: 0.2439 Acc: 0.8940 | Val Loss: 0.4870 Acc: 0.8020
Loaded best model (val_acc=0.8280).
```

In [31]:
```
torch.save(model.state_dict(), 'cats_and_dogs_subset_1.pth')
```

In [32]:
```
import matplotlib.pyplot as plt

# Suppose you have stored these during training
# e.g. history = {'train_loss': [...], 'val_loss': [...], 'train_acc': [...], 'val_

train_acc = history['train_acc']
val_acc = history['val_acc']
train_loss = history['train_loss']
```

```
val_loss = history['val_loss']

epochs = range(1, len(train_acc) + 1)

# Plot training & validation accuracy
plt.plot(epochs, train_acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.figure()

# Plot training & validation loss
plt.plot(epochs, train_loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```





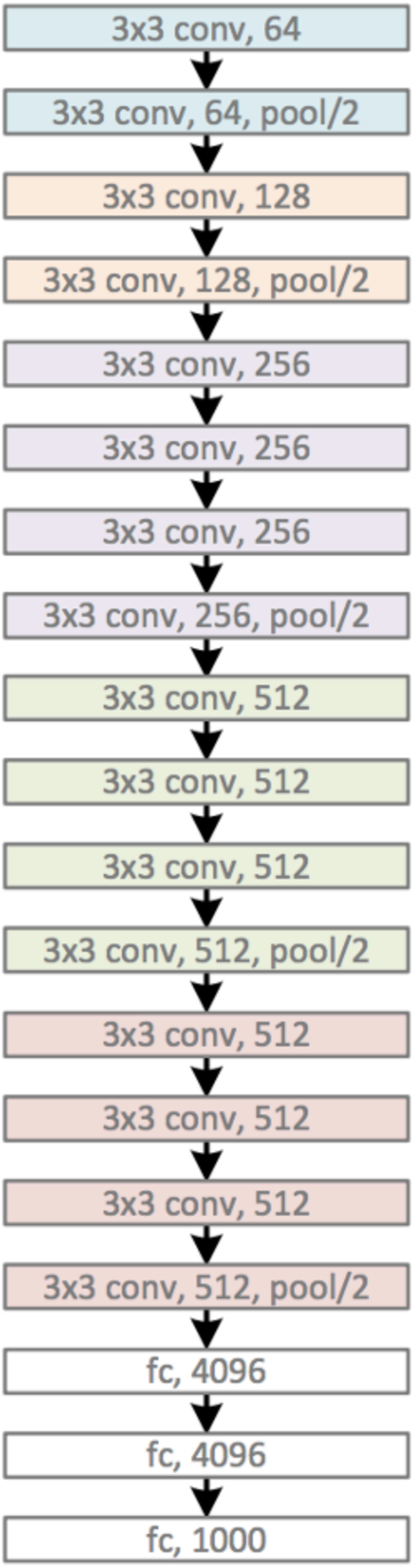
In [ ]:

Using a pretrained convnet

- A common and highly effective approach of deep learning on small image datasets is to use a pretrained network.
    - A *pretrained network* is a saved network that was previously trained on a large dataset.
    - For instance, you might train a network on ImageNet (where classes are mostly animals and everyday objects) and then repurpose this trained network for identifying furniture items in images.
- Such portability of learned features across different problems is a key advantage of deep learning compared to many other approaches.
- Here, let's consider a large CNN trained on the ImageNet dataset (1.4M labeled images and 1,000 different classes).
- We will use the VGGNet architecture, developed by Karen Simonyan and Andrew Zisserman in 2014.

VGGNet

- VGGNet is one of the most influential convolutional neural network architectures in computer vision.
- Key characteristics:
  - Uniform architecture: All convolutional layers use small 3*3 filters and stride 1 and padding 1. This allows the network to capture fine spatial details while keeping the architecture simple.
  - Deep but simple: The depth increases gradually by stacking many convolutional layers followed by max-pooling layers.
  - Parameter efficiency: Although deeper than earlier models, VGGNet maintains a consistent pattern of operations, making it conceptually easy to understand.
  - Variants: The most common versions are VGG16 and VGG19, referring to the total number of weight layers (16 and 19).
- VGG19 architecture


3x3 conv, 64
3x3 conv, 64, pool/2
3x3 conv, 128
3x3 conv, 128, pool/2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256, pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512, pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512, pool/2
fc, 4096
fc, 4096
fc, 1000

Feature extraction

- Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples.

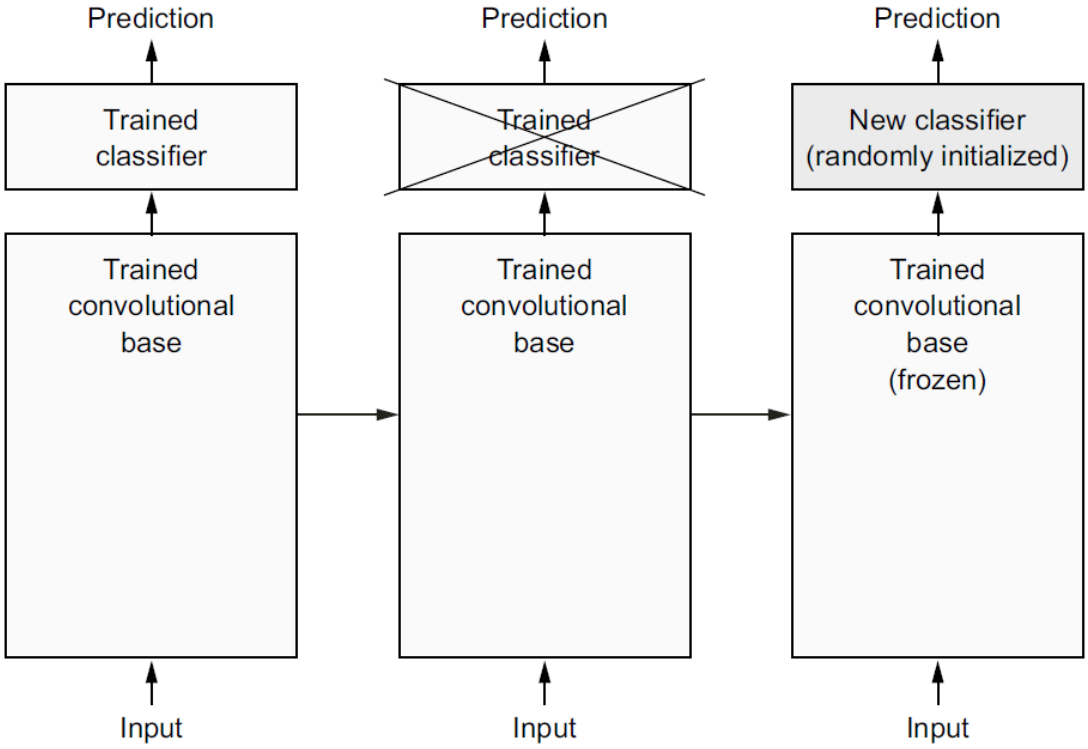


Figure 8.12 Swapping classifiers while keeping the same convolutional base

- Why only reuse the convolutional base? Could we reuse the densely connected classifier as well?
- Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model.
  - Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts (such as “cat ear” or “dog eye”).
- The VGG16 model, along with many other well-known architectures, is available through the `torchvision.models` module in PyTorch.

In [35]:
```
import torch
import torchvision.models as models
```

```
# Load pretrained VGG16 (weights trained on ImageNet)
vgg16 = models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1)

# Remove the top classifier part (fully connected layers)
# This keeps only the convolutional feature extractor
conv_base = vgg16.features

print(conv_base)
```

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

In [36]:

```
from torchsummary import summary

model = conv_base.cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(3, 150, 150), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 150, 150]           1,792
              ReLU-2         [-1, 64, 150, 150]               0
            Conv2d-3         [-1, 64, 150, 150]          36,928
              ReLU-4         [-1, 64, 150, 150]               0
         MaxPool2d-5           [-1, 64, 75, 75]               0
            Conv2d-6          [-1, 128, 75, 75]          73,856
              ReLU-7          [-1, 128, 75, 75]               0
            Conv2d-8          [-1, 128, 75, 75]         147,584
              ReLU-9          [-1, 128, 75, 75]               0
        MaxPool2d-10          [-1, 128, 37, 37]               0
           Conv2d-11          [-1, 256, 37, 37]         295,168
             ReLU-12          [-1, 256, 37, 37]               0
           Conv2d-13          [-1, 256, 37, 37]         590,080
             ReLU-14          [-1, 256, 37, 37]               0
           Conv2d-15          [-1, 256, 37, 37]         590,080
             ReLU-16          [-1, 256, 37, 37]               0
        MaxPool2d-17          [-1, 256, 18, 18]               0
           Conv2d-18          [-1, 512, 18, 18]       1,180,160
             ReLU-19          [-1, 512, 18, 18]               0
           Conv2d-20          [-1, 512, 18, 18]       2,359,808
             ReLU-21          [-1, 512, 18, 18]               0
           Conv2d-22          [-1, 512, 18, 18]       2,359,808
             ReLU-23          [-1, 512, 18, 18]               0
        MaxPool2d-24            [-1, 512, 9, 9]               0
           Conv2d-25            [-1, 512, 9, 9]       2,359,808
             ReLU-26            [-1, 512, 9, 9]               0
           Conv2d-27            [-1, 512, 9, 9]       2,359,808
             ReLU-28            [-1, 512, 9, 9]               0
           Conv2d-29            [-1, 512, 9, 9]       2,359,808
             ReLU-30            [-1, 512, 9, 9]               0
        MaxPool2d-31            [-1, 512, 4, 4]               0
================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.26
Forward/backward pass size (MB): 96.55
Params size (MB): 56.13
Estimated Total Size (MB): 152.94
----------------------------------------------------------------
```

- There are two ways we could proceed:
  - Option 1) Extract features using the convolutional base, and then save them on disk.
  - Option 2) Extend the model by adding `Linear` layers on top, and train it.

- **Option 2)**
  - Extending the `conv_base` model and running it end to end on the inputs

In [39]:
```
import torch.nn as nn

ft_model = nn.Sequential(
    conv_base,               # pretrained convolutional layers
    nn.Flatten(),
    nn.Linear(512 * 4 * 4, 256),  # depends on input size
    nn.ReLU(),
    nn.Linear(256, 1)        # binary classification logit
)
```

In [40]:
```
from torchsummary import summary

model = ft_model.cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(3, 150, 150), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 150, 150]           1,792
              ReLU-2         [-1, 64, 150, 150]               0
            Conv2d-3         [-1, 64, 150, 150]          36,928
              ReLU-4         [-1, 64, 150, 150]               0
         MaxPool2d-5           [-1, 64, 75, 75]               0
            Conv2d-6          [-1, 128, 75, 75]          73,856
              ReLU-7          [-1, 128, 75, 75]               0
            Conv2d-8          [-1, 128, 75, 75]         147,584
              ReLU-9          [-1, 128, 75, 75]               0
        MaxPool2d-10          [-1, 128, 37, 37]               0
           Conv2d-11          [-1, 256, 37, 37]         295,168
             ReLU-12          [-1, 256, 37, 37]               0
           Conv2d-13          [-1, 256, 37, 37]         590,080
             ReLU-14          [-1, 256, 37, 37]               0
           Conv2d-15          [-1, 256, 37, 37]         590,080
             ReLU-16          [-1, 256, 37, 37]               0
        MaxPool2d-17          [-1, 256, 18, 18]               0
           Conv2d-18          [-1, 512, 18, 18]       1,180,160
             ReLU-19          [-1, 512, 18, 18]               0
           Conv2d-20          [-1, 512, 18, 18]       2,359,808
             ReLU-21          [-1, 512, 18, 18]               0
           Conv2d-22          [-1, 512, 18, 18]       2,359,808
             ReLU-23          [-1, 512, 18, 18]               0
        MaxPool2d-24            [-1, 512, 9, 9]               0
           Conv2d-25            [-1, 512, 9, 9]       2,359,808
             ReLU-26            [-1, 512, 9, 9]               0
           Conv2d-27            [-1, 512, 9, 9]       2,359,808
             ReLU-28            [-1, 512, 9, 9]               0
           Conv2d-29            [-1, 512, 9, 9]       2,359,808
             ReLU-30            [-1, 512, 9, 9]               0
        MaxPool2d-31            [-1, 512, 4, 4]               0
          Flatten-32                 [-1, 8192]               0
           Linear-33                  [-1, 256]       2,097,408
             ReLU-34                  [-1, 256]               0
           Linear-35                    [-1, 1]             257
================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.26
Forward/backward pass size (MB): 96.61
Params size (MB): 64.13
Estimated Total Size (MB): 161.01
----------------------------------------------------------------
```

- Before training the model, it is often desirable to freeze the convolutional base: that is, to prevent the pretrained convolutional layers from updating their weights during training.

    - Note that freezing is not strictly required. Whether to freeze or fine-tune the convolutional base depends on your specific task, dataset size, and computational budget.
    - In this course example, we will freeze the convolutional base to use it as a fixed feature extractor for simplicity and efficiency.
- *Freezing* means that the gradients are not computed for those parameters, so the pretrained feature extractor remains unchanged while only the new classifier layers are trained.

- In PyTorch, you freeze a model by setting each parameter's attribute `requires_grad` to `False.

In [41]:
```
# Suppose conv_base is the pretrained VGG16 feature extractor
for param in conv_base.parameters():
    param.requires_grad = False

# Attach a new classifier head
import torch.nn as nn

ft_model = nn.Sequential(
    conv_base,
    nn.Flatten(),
    nn.Linear(512 * 4 * 4, 256),  # depends on the conv output size
    nn.ReLU(),
    nn.Linear(256, 1)             # binary classification logit
)

print("Number of trainable parameters:",
      sum(p.numel() for p in ft_model.parameters() if p.requires_grad))
```

```
Number of trainable parameters: 2097665
```

In [42]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Load pretrained VGG16 and take the convolutional feature extractor
vgg16 = models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1)
conv_base = vgg16.features

# 2) Freeze the convolutional base (course example; not strictly required in genera
for p in conv_base.parameters():
    p.requires_grad = False

# 3) (One-time) verify conv output shape for input size (3, 150, 150)
with torch.no_grad():
    conv_base.eval()
    dummy = torch.zeros(1, 3, 150, 150)
    feat = conv_base(dummy)         # Expect: (1, 512, 4, 4)
    print("Conv feature shape:", tuple(feat.shape))
```

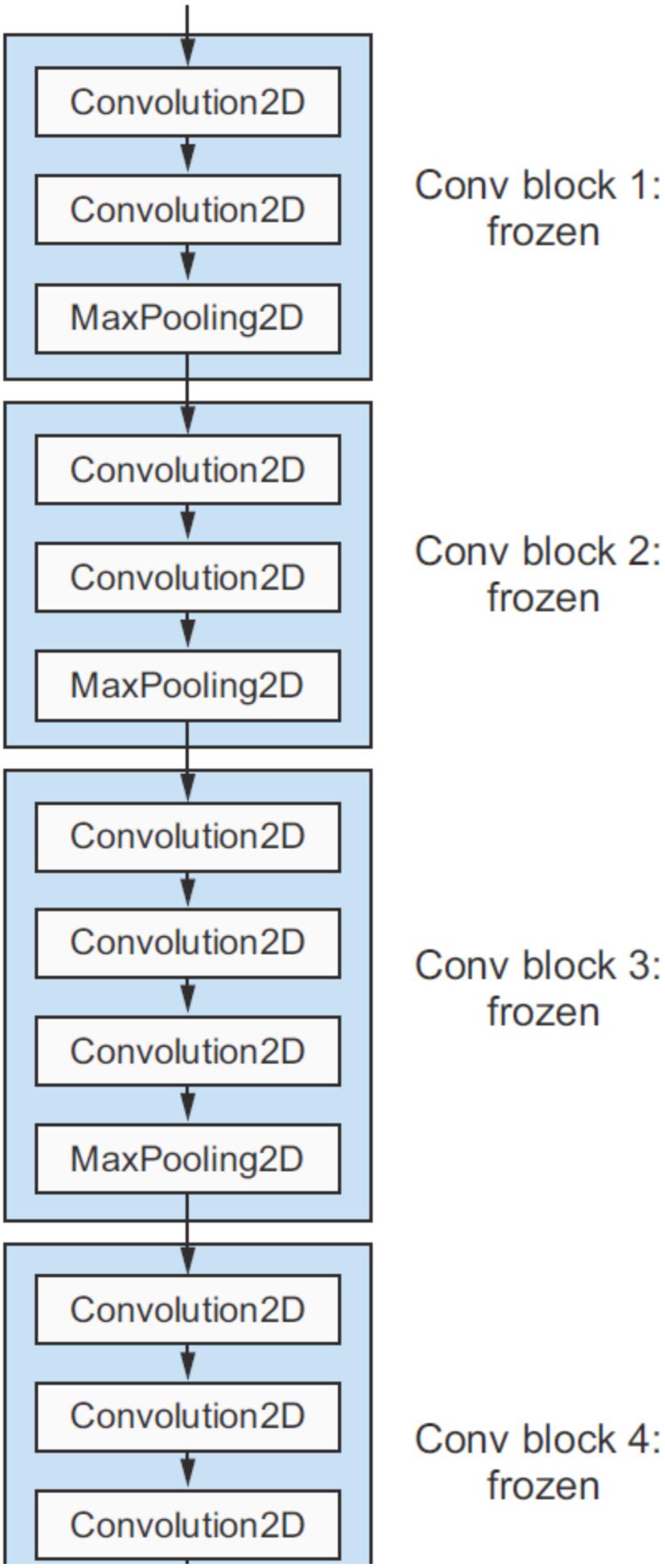
```
# We know 150x150 through VGG16's pooling → (C=512, H=4, W=4), so flatten dim = 512
FLAT_DIM = 512 * 4 * 4

# 4) Define the classifier head on top of the frozen conv base
class VGG16BinaryClassifier(nn.Module):
    def __init__(self, conv_base, flat_dim=FLAT_DIM):
        super().__init__()
        self.conv_base = conv_base
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(flat_dim, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 1)     # binary logit
        )
    def forward(self, x):
        x = self.conv_base(x)
        x = self.classifier(x)
        return x

model = VGG16BinaryClassifier(conv_base).to(device)

# 5) Loss & optimizer
criterion = nn.BCEWithLogitsLoss()
# Simpler & clearer for students: pass *all* params; frozen ones (requires_grad=Fal
optimizer = optim.RMSprop(model.parameters(), lr=1e-4)

# 6) Training loop (same settings as before: epochs=30, batch=20, BCEWithLogits, RM
EPOCHS = 30
history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
best_val_acc, best_state = 0.0, None

for epoch in range(1, EPOCHS + 1):
    # ---- Train ----
    model.train()
    running_loss, running_correct, total = 0.0, 0, 0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device).float().view(-1, 1)  # (N,1) with {0,1}

        optimizer.zero_grad()
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            preds = (torch.sigmoid(logits) >= 0.5).float()
            running_correct += (preds == labels).sum().item()
            running_loss += loss.item() * images.size(0)
            total += images.size(0)

    train_loss = running_loss / total
    train_acc  = running_correct / total
```

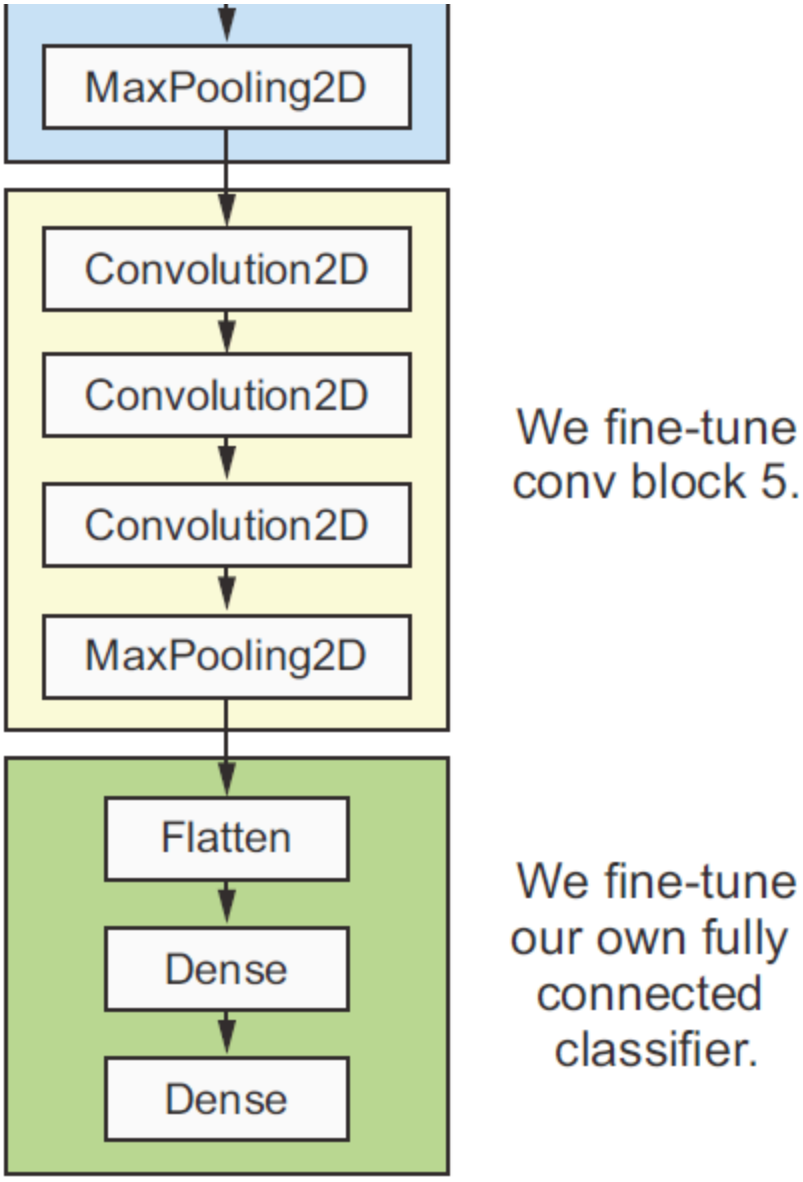
```
    # ---- Validation ----
    model.eval()
    val_loss_sum, val_correct, val_total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device).float().view(-1, 1)

            logits = model(images)
            loss = criterion(logits, labels)
            preds = (torch.sigmoid(logits) >= 0.5).float()

            val_loss_sum += loss.item() * images.size(0)
            val_correct  += (preds == labels).sum().item()
            val_total    += labels.size(0)

    val_loss = val_loss_sum / val_total
    val_acc  = val_correct / val_total

    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)

    print(f"Epoch {epoch:02d}/{EPOCHS} | "
          f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} | "
          f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().it

# 7) Load best weights (recommended)
if best_state is not None:
    model.load_state_dict(best_state)
    print(f"Loaded best model (val_acc={best_val_acc:.4f}).")
```

```
Conv feature shape: (1, 512, 4, 4)
Epoch 01/30 | Train Loss: 0.2901 Acc: 0.8695 | Val Loss: 0.1719 Acc: 0.9280
Epoch 02/30 | Train Loss: 0.2048 Acc: 0.9140 | Val Loss: 0.1461 Acc: 0.9340
Epoch 03/30 | Train Loss: 0.1913 Acc: 0.9185 | Val Loss: 0.1454 Acc: 0.9360
Epoch 04/30 | Train Loss: 0.1763 Acc: 0.9240 | Val Loss: 0.1522 Acc: 0.9320
Epoch 05/30 | Train Loss: 0.1513 Acc: 0.9420 | Val Loss: 0.1492 Acc: 0.9340
Epoch 06/30 | Train Loss: 0.1519 Acc: 0.9400 | Val Loss: 0.1461 Acc: 0.9360
Epoch 07/30 | Train Loss: 0.1371 Acc: 0.9485 | Val Loss: 0.1587 Acc: 0.9310
Epoch 08/30 | Train Loss: 0.1367 Acc: 0.9475 | Val Loss: 0.1506 Acc: 0.9360
Epoch 09/30 | Train Loss: 0.1375 Acc: 0.9445 | Val Loss: 0.1558 Acc: 0.9360
Epoch 10/30 | Train Loss: 0.1249 Acc: 0.9500 | Val Loss: 0.1573 Acc: 0.9400
Epoch 11/30 | Train Loss: 0.1141 Acc: 0.9505 | Val Loss: 0.1521 Acc: 0.9420
Epoch 12/30 | Train Loss: 0.1219 Acc: 0.9500 | Val Loss: 0.1686 Acc: 0.9390
Epoch 13/30 | Train Loss: 0.1062 Acc: 0.9555 | Val Loss: 0.1726 Acc: 0.9370
Epoch 14/30 | Train Loss: 0.1069 Acc: 0.9510 | Val Loss: 0.1612 Acc: 0.9360
Epoch 15/30 | Train Loss: 0.1130 Acc: 0.9520 | Val Loss: 0.1641 Acc: 0.9380
Epoch 16/30 | Train Loss: 0.1110 Acc: 0.9585 | Val Loss: 0.1777 Acc: 0.9340
Epoch 17/30 | Train Loss: 0.0989 Acc: 0.9615 | Val Loss: 0.1651 Acc: 0.9450
Epoch 18/30 | Train Loss: 0.0968 Acc: 0.9575 | Val Loss: 0.1554 Acc: 0.9390
Epoch 19/30 | Train Loss: 0.0939 Acc: 0.9610 | Val Loss: 0.1525 Acc: 0.9400
Epoch 20/30 | Train Loss: 0.1005 Acc: 0.9615 | Val Loss: 0.1676 Acc: 0.9420
Epoch 21/30 | Train Loss: 0.0807 Acc: 0.9690 | Val Loss: 0.1583 Acc: 0.9420
Epoch 22/30 | Train Loss: 0.0920 Acc: 0.9625 | Val Loss: 0.1631 Acc: 0.9400
Epoch 23/30 | Train Loss: 0.1036 Acc: 0.9575 | Val Loss: 0.1567 Acc: 0.9450
Epoch 24/30 | Train Loss: 0.0764 Acc: 0.9745 | Val Loss: 0.1607 Acc: 0.9450
Epoch 25/30 | Train Loss: 0.0922 Acc: 0.9630 | Val Loss: 0.1770 Acc: 0.9380
Epoch 26/30 | Train Loss: 0.0782 Acc: 0.9660 | Val Loss: 0.1734 Acc: 0.9420
Epoch 27/30 | Train Loss: 0.0797 Acc: 0.9685 | Val Loss: 0.1556 Acc: 0.9420
Epoch 28/30 | Train Loss: 0.0762 Acc: 0.9680 | Val Loss: 0.1786 Acc: 0.9370
Epoch 29/30 | Train Loss: 0.0765 Acc: 0.9730 | Val Loss: 0.1795 Acc: 0.9450
Epoch 30/30 | Train Loss: 0.0839 Acc: 0.9665 | Val Loss: 0.1808 Acc: 0.9370
Loaded best model (val_acc=0.9450).
```

Fine-tuning

- Another widely used technique for model reuse is *fine-tuning*.
    - Unfreezing a few of the top layers of a frozen model base, and jointly training both the newly added part of the model and these top layers.


Convolution2D
Convolution2D
MaxPooling2D
Conv block 1: frozen
Convolution2D
Convolution2D
MaxPooling2D
Conv block 2: frozen
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 3: frozen
Convolution2D
Convolution2D
Convolution2D
Conv block 4: frozen



- The steps for fine-tuning a network
  - Add the custom network on top of an already-trained base network.
  - Freeze the base network.
  - Train the part you added.
  - Unfreeze some layers in the base network.
  - Jointly train both these layers and the part you added.

In [43]:
```
from torchsummary import summary

model = conv_base.cpu()

# summary(model, input_size=(channels, height, width))
summary(model, input_size=(3, 150, 150), device='cpu')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 150, 150]           1,792
              ReLU-2         [-1, 64, 150, 150]               0
            Conv2d-3         [-1, 64, 150, 150]          36,928
              ReLU-4         [-1, 64, 150, 150]               0
         MaxPool2d-5           [-1, 64, 75, 75]               0
            Conv2d-6          [-1, 128, 75, 75]          73,856
              ReLU-7          [-1, 128, 75, 75]               0
            Conv2d-8          [-1, 128, 75, 75]         147,584
              ReLU-9          [-1, 128, 75, 75]               0
        MaxPool2d-10          [-1, 128, 37, 37]               0
           Conv2d-11          [-1, 256, 37, 37]         295,168
             ReLU-12          [-1, 256, 37, 37]               0
           Conv2d-13          [-1, 256, 37, 37]         590,080
             ReLU-14          [-1, 256, 37, 37]               0
           Conv2d-15          [-1, 256, 37, 37]         590,080
             ReLU-16          [-1, 256, 37, 37]               0
        MaxPool2d-17          [-1, 256, 18, 18]               0
           Conv2d-18          [-1, 512, 18, 18]       1,180,160
             ReLU-19          [-1, 512, 18, 18]               0
           Conv2d-20          [-1, 512, 18, 18]       2,359,808
             ReLU-21          [-1, 512, 18, 18]               0
           Conv2d-22          [-1, 512, 18, 18]       2,359,808
             ReLU-23          [-1, 512, 18, 18]               0
        MaxPool2d-24            [-1, 512, 9, 9]               0
           Conv2d-25            [-1, 512, 9, 9]       2,359,808
             ReLU-26            [-1, 512, 9, 9]               0
           Conv2d-27            [-1, 512, 9, 9]       2,359,808
             ReLU-28            [-1, 512, 9, 9]               0
           Conv2d-29            [-1, 512, 9, 9]       2,359,808
             ReLU-30            [-1, 512, 9, 9]               0
        MaxPool2d-31            [-1, 512, 4, 4]               0
================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
----------------------------------------------------------------
Input size (MB): 0.26
Forward/backward pass size (MB): 96.55
Params size (MB): 56.13
Estimated Total Size (MB): 152.94
----------------------------------------------------------------
```

- We will fine-tune the last three convolutional layers.

In [45]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Load a new pretrained VGG16 convolutional base
```

```
vgg16 = models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1)
conv_base = vgg16.features

# 2) Freeze all convolutional layers initially
for p in conv_base.parameters():
    p.requires_grad = False

# 3) The convolutional output for input (3, 150, 150) is (512, 4, 4)
#    Hence, the flattened feature size is 512 * 4 * 4 = 8192
FLAT_DIM = 512 * 4 * 4

# 4) Define the classifier on top of the convolutional base
class VGG16BinaryClassifier(nn.Module):
    def __init__(self, conv_base, flat_dim=FLAT_DIM):
        super().__init__()
        self.conv_base = conv_base
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(flat_dim, 256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 1)   # output logit for binary classification
        )
    def forward(self, x):
        x = self.conv_base(x)
        x = self.classifier(x)
        return x

# Create a new model instance
model = VGG16BinaryClassifier(conv_base).to(device)

# 5) Unfreeze the top 3 Conv2d layers for fine-tuning
conv_layers = [m for m in model.conv_base.modules() if isinstance(m, nn.Conv2d)]
for m in conv_layers[-3:]:  # last 3 Conv2d layers
    for p in m.parameters():
        p.requires_grad = True

# 6) Set up parameter groups with different learning rates
#    - Fine-tuned convolutional layers: small lr = 1e-5
#    - Classifier layers: regular lr = 1e-4
ft_params = []
for m in conv_layers[-3:]:
    ft_params += list(m.parameters())

classifier_params = list(model.classifier.parameters())

optimizer = optim.RMSprop([
    {"params": ft_params,         "lr": 1e-5},
    {"params": classifier_params, "lr": 1e-4},
])
criterion = nn.BCEWithLogitsLoss()

# 7) Training loop (same settings as before)
EPOCHS_FT = 10  # fine-tune for 10 epochs
history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
best_val_acc, best_state = 0.0, None
```

```python
for epoch in range(1, EPOCHS_FT + 1):
    # ---- Training phase ----
    model.train()
    running_loss, running_correct, total = 0.0, 0, 0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device).float().view(-1, 1)  # reshape to (N,1)

        optimizer.zero_grad()
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            preds = (torch.sigmoid(logits) >= 0.5).float()
            running_correct += (preds == labels).sum().item()
            running_loss += loss.item() * images.size(0)
            total += images.size(0)

    train_loss = running_loss / total
    train_acc  = running_correct / total

    # ---- Validation phase ----
    model.eval()
    val_loss_sum, val_correct, val_total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device).float().view(-1, 1)

            logits = model(images)
            loss = criterion(logits, labels)

            preds = (torch.sigmoid(logits) >= 0.5).float()
            val_loss_sum += loss.item() * images.size(0)
            val_correct  += (preds == labels).sum().item()
            val_total    += labels.size(0)

    val_loss = val_loss_sum / val_total
    val_acc  = val_correct / val_total

    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)

    print(f"[Fine-Tune] Epoch {epoch:02d}/{EPOCHS_FT} | "
          f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} | "
          f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

    # Save best model based on validation accuracy
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().it
```

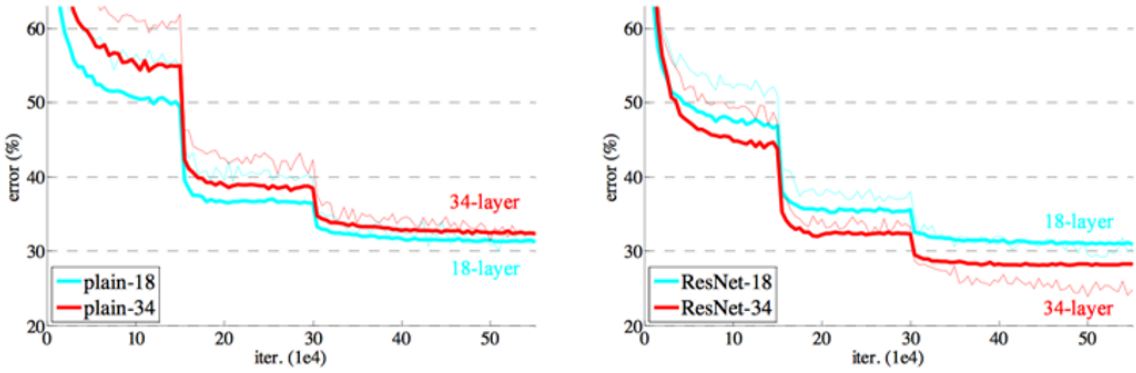
```
# 8) Load the best fine-tuned weights
if best_state is not None:
    model.load_state_dict(best_state)
    print(f"Loaded best fine-tuned model (val_acc={best_val_acc:.4f}).")
```

```
[Fine-Tune] Epoch 01/10 | Train Loss: 0.2706 Acc: 0.8800 | Val Loss: 0.1409 Acc: 0.9
410
[Fine-Tune] Epoch 02/10 | Train Loss: 0.1464 Acc: 0.9425 | Val Loss: 0.1688 Acc: 0.9
270
[Fine-Tune] Epoch 03/10 | Train Loss: 0.1338 Acc: 0.9425 | Val Loss: 0.1307 Acc: 0.9
450
[Fine-Tune] Epoch 04/10 | Train Loss: 0.1171 Acc: 0.9545 | Val Loss: 0.1259 Acc: 0.9
480
[Fine-Tune] Epoch 05/10 | Train Loss: 0.1173 Acc: 0.9555 | Val Loss: 0.1272 Acc: 0.9
510
[Fine-Tune] Epoch 06/10 | Train Loss: 0.1021 Acc: 0.9620 | Val Loss: 0.1372 Acc: 0.9
410
[Fine-Tune] Epoch 07/10 | Train Loss: 0.0983 Acc: 0.9615 | Val Loss: 0.1438 Acc: 0.9
440
[Fine-Tune] Epoch 08/10 | Train Loss: 0.0816 Acc: 0.9670 | Val Loss: 0.1119 Acc: 0.9
580
[Fine-Tune] Epoch 09/10 | Train Loss: 0.0749 Acc: 0.9720 | Val Loss: 0.1341 Acc: 0.9
480
[Fine-Tune] Epoch 10/10 | Train Loss: 0.0675 Acc: 0.9755 | Val Loss: 0.1204 Acc: 0.9
500
Loaded best fine-tuned model (val_acc=0.9580).
```

- **Exercise**
  - Evaluate the final model on the test data.

Residual Networks

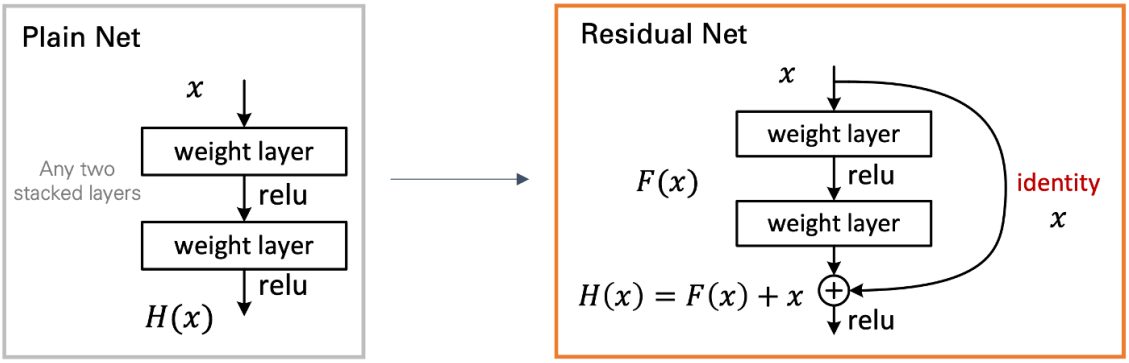
- Residual Networks (ResNets) were introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2015.
- They represented a major breakthrough in deep learning, allowing the successful training of very deep neural networks without suffering from the degradation problem.



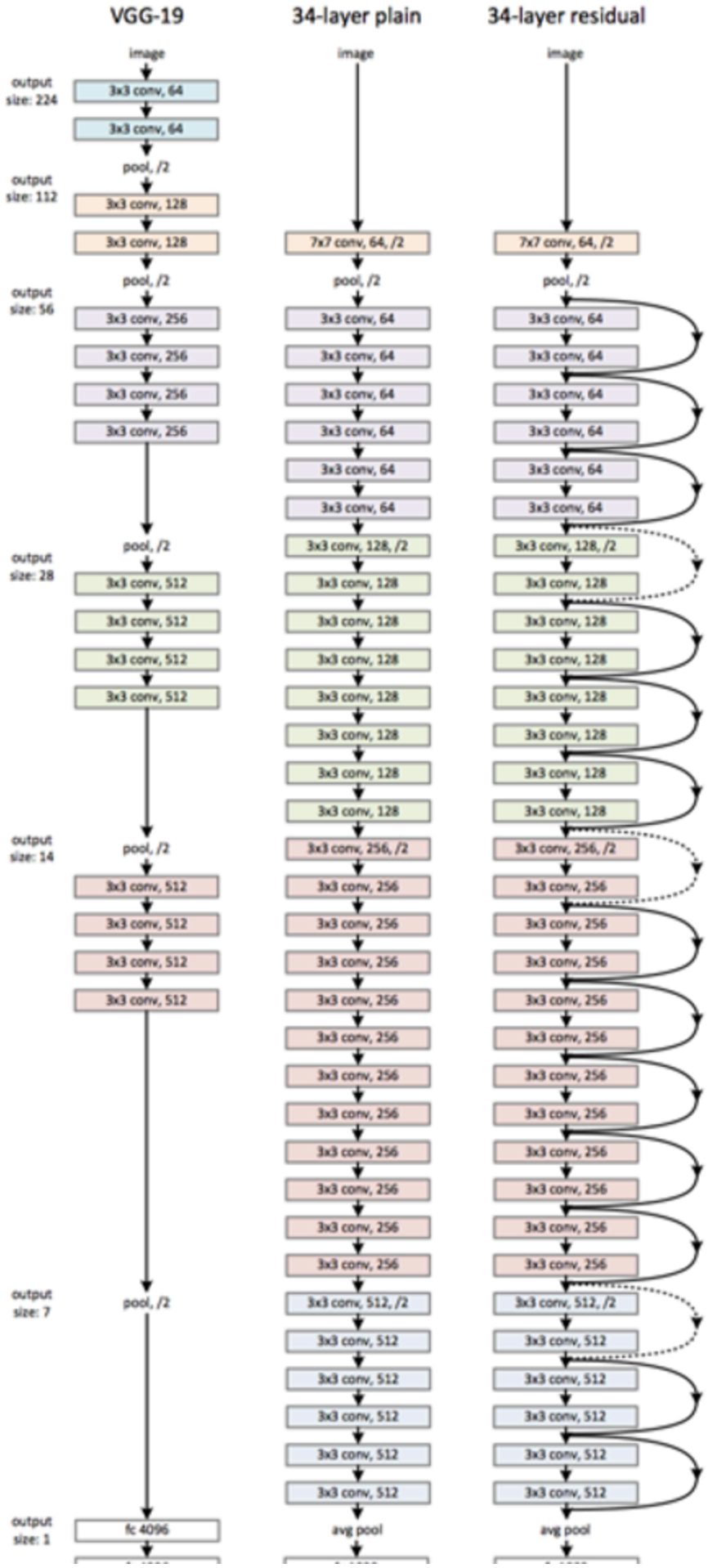
- QUESTION:Should a 34-layer NN be better?

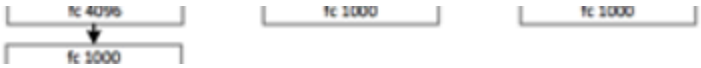
- Motivation
  - Before ResNet, simply stacking more layers in a CNN did not guarantee better performance.
  - Deeper models often performed worse because of:
    - Vanishing or exploding gradients
    - Optimization difficulties
    - Degradation problem: adding layers increased training error, not just test error
  - The ResNet architecture was designed to solve this by introducing shortcut (skip) connections that enable information to flow directly across layers.

- The core idea of ResNet is the residual block, where the input x is directly added to the output of a few convolutional layers: $y = F(x) + x$
  Here,
  - $F(x)$ represents the output of a small sequence of layers (typically two 3*3 convolutions)
  - The addition of x creates a shortcut path that bypasses these layers.
- This identity mapping allows the network to easily learn functions of the form $F(x) = 0$, meaning the block can simply copy its input forward if that's optimal.
- As a result, the model can learn the identity transformation whenever deeper layers are unnecessary, making it much easier to optimize very deep networks.



- ResNet architecture


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool



In [49]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Load a pretrained ResNet-50
resnet = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)

# 2) Replace the classifier head: 2048 -> 1 (binary logit)
#    (ResNet-18 would use 512 instead of 2048.)
in_features = resnet.fc.in_features
resnet.fc = nn.Linear(in_features, 1)

# 3) Freeze the convolutional base first (feature extractor stage)
for name, p in resnet.named_parameters():
    if not name.startswith("fc."):
        p.requires_grad = False  # freeze everything except the new fc

# 4) Unfreeze ONLY the last residual stage (layer4) for fine-tuning
for p in resnet.layer4.parameters():
    p.requires_grad = True

# 5) Move to device
model = resnet.to(device)

# 6) Build parameter groups: layer4 gets small lr, fc gets regular lr
ft_params = list(model.layer4.parameters())
head_params = list(model.fc.parameters())

optimizer = optim.RMSprop([
    {"params": ft_params,  "lr": 1e-5},
    {"params": head_params,"lr": 1e-4},
])
criterion = nn.BCEWithLogitsLoss()

# 7) Fine-tuning loop (same structure/metrics as VGG example)
EPOCHS_FT = 10
history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
best_val_acc, best_state = 0.0, None

for epoch in range(1, EPOCHS_FT + 1):
    # ---- Training ----
    model.train()
    running_loss, running_correct, total = 0.0, 0, 0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device).float().view(-1, 1)  # (N,1) with {0,1}

        optimizer.zero_grad()
```

```
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            preds = (torch.sigmoid(logits) >= 0.5).float()
            running_correct += (preds == labels).sum().item()
            running_loss    += loss.item() * images.size(0)
            total           += images.size(0)

    train_loss = running_loss / total
    train_acc  = running_correct / total

    # ---- Validation ----
    model.eval()
    val_loss_sum, val_correct, val_total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device).float().view(-1, 1)

            logits = model(images)
            loss = criterion(logits, labels)

            preds = (torch.sigmoid(logits) >= 0.5).float()
            val_loss_sum += loss.item() * images.size(0)
            val_correct  += (preds == labels).sum().item()
            val_total    += images.size(0)

    val_loss = val_loss_sum / val_total
    val_acc  = val_correct / val_total

    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)

    print(f"[ResNet-FT] Epoch {epoch:02d}/{EPOCHS_FT} | "
          f"Train Loss: {train_loss:.4f} Acc: {train_acc:.4f} | "
          f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

    # Save best by validation accuracy
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().it

# 8) Load best fine-tuned weights
if best_state is not None:
    model.load_state_dict(best_state)
    print(f"Loaded best ResNet-50 fine-tuned model (val_acc={best_val_acc:.4f}).")
```

```
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.c
ache/torch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 109MB/s]
```

```
[ResNet-FT] Epoch 01/10 | Train Loss: 0.2557 Acc: 0.8945 | Val Loss: 0.1158 Acc: 0.9
610
[ResNet-FT] Epoch 02/10 | Train Loss: 0.1500 Acc: 0.9445 | Val Loss: 0.0980 Acc: 0.9
660
[ResNet-FT] Epoch 03/10 | Train Loss: 0.1253 Acc: 0.9490 | Val Loss: 0.0814 Acc: 0.9
680
[ResNet-FT] Epoch 04/10 | Train Loss: 0.1016 Acc: 0.9665 | Val Loss: 0.0804 Acc: 0.9
700
[ResNet-FT] Epoch 05/10 | Train Loss: 0.1028 Acc: 0.9560 | Val Loss: 0.0728 Acc: 0.9
690
[ResNet-FT] Epoch 06/10 | Train Loss: 0.1038 Acc: 0.9550 | Val Loss: 0.0719 Acc: 0.9
730
[ResNet-FT] Epoch 07/10 | Train Loss: 0.0800 Acc: 0.9705 | Val Loss: 0.0752 Acc: 0.9
670
[ResNet-FT] Epoch 08/10 | Train Loss: 0.0901 Acc: 0.9650 | Val Loss: 0.0753 Acc: 0.9
700
[ResNet-FT] Epoch 09/10 | Train Loss: 0.0675 Acc: 0.9720 | Val Loss: 0.0690 Acc: 0.9
720
[ResNet-FT] Epoch 10/10 | Train Loss: 0.0801 Acc: 0.9700 | Val Loss: 0.0736 Acc: 0.9
710
Loaded best ResNet-50 fine-tuned model (val_acc=0.9730).
```

In [ ]: