Exercise 1

在图像分类中，CNN比完全连接的DNN有什么优势？

答案：

这些是 CNN 在图像分类方面优于完全连接的 DNN 的主要优势：

1. 由于连续层仅部分连接，并且由于它大量重用其权重，因此 CNN 的参数比完全连接的 DNN 少得多，这使得训练速度更快，降低了过度拟合的风险，并且需要更少的训练数据。
2. 当 CNN 学习到可以检测特定特征的内核时，它可以检测图像中任何位置的该特征。相反，当 DNN 在一个位置学习特征时，它只能在该特定位置检测到它。 由于图像通常具有非常重复的特征，因此 CNN 能够比 DNN 更好地概括图像处理任务，例如分类，使用更少的训练示例。
3. 最后，DNN 没有关于像素如何组织的先验知识；它不知道相似的像素很近。CNN 的架构嵌入了这种先验知识。较低层通常识别图像小区域中的特征，而较高层将较低层特征组合成更大的特征。这适用于大多数自然图像，使 CNN 与 DNN 相比具有决定性的领先优势。

Exercise 2

考虑一个由三个卷积层组成的CNN，每个层有3×3内核，步幅为2，“same”填充。最低层输出100个特征图，中间层输出200个，最高层输出400个。输入的图像为200×300像素的RGB图像：

CNN中的参数总数是多少？

答案：

由于它的第一个卷积层有 3 × 3 个内核，输入有三个通道（红色、绿色和蓝色），每个特征图有 3 × 3 × 3 个权重，加上一个偏置项。 每个特征图有 28 个参数。 由于第一个卷积层有 100 个特征映射，因此它总共有 2,800 个参数。 第二个卷积层有 3 × 3 个核，它的输入是前一层的 100 个特征图的集合，所以每个特征图有 3 × 3 × 100 = 900 个权重，加上一个偏置项。 因为它有 200 个特征图，所以这一层有 901 × 200 = 180,200 个参数。 最后，第三个也是最后一个卷积层也有 3 × 3 个核，它的输入是前几层的 200 个特征图的集合，所以每个特征图有 3 × 3 × 200 = 1,800 个权重，加上一个偏置项。 由于它有 400 个特征图，因此该层共有 1,801 × 400 = 720,400 个参数。 总而言之，CNN 有 2,800 + 180,200 + 720,400 = 903,400 个参数。

如果我们使用32位浮点数，那么在对单个实例进行预测时，这个网络至少需要多少RAM呢？

答案：

现在让我们计算一下这个神经网络在对单个实例进行预测时（至少）需要多少 RAM。 首先让我们计算每一层的特征图大小。 由于我们使用的步幅为 2 且填充“same”，因此特征图的水平和垂直维度在每一层都除以 2（必要时四舍五入）。 因此，由于输入通道为 200 × 300 像素，因此第一层的特征图为 100 × 150，第二层的特征图为 50 × 75，第三层的特征图为 25 × 38。由于 32 位为 4 个字节，并且 第一个卷积层有 100 个特征图，第一层占用 4 × 100 × 150 × 100 = 600 万字节（6 MB）。 第二层占用 4 × 50 × 75 × 200 = 300 万字节（3 MB）。 最后，第三层占用 4 × 25 × 38 × 400 = 1,520,000 字节（约 1.5 MB）。 然而，一旦计算完一层，就可以释放前一层占用的内存，所以如果一切都优化好，只需要 6 + 3 = 900 万字节（9 MB）的 RAM（当第二层有 刚刚计算出来，但第一层占用的内存还没有释放）。 但是等等，你还需要加上CNN参数占用的内存！ 我们之前计算出它有 903,400 个参数，每个参数占用 4 个字节，因此这增加了 3,613,600 个字节（约 3.6 MB）。 因此，所需的总 RAM（至少）为 12,613,600 字节（约 12.6 MB）。

那么对50张图片的小批量进行训练呢？

答案：

最后，让我们计算在 50 张图像的小批量上训练 CNN 时所需的最小 RAM 量。 在训练期间，TensorFlow 使用反向传播，这需要保留在正向传播期间计算的所有值，直到反向传播开始。 因此，我们必须计算单个实例所有层所需的总 RAM，并将其乘以 50 。此时，让我们开始以兆字节而不是字节为单位进行计数。 我们之前计算过，三个层每个实例分别需要 6、3 和 1.5 MB。 每个实例总共需要 10.5 MB，因此对于 50 个实例，所需的总 RAM 为 525 MB。 再加上输入图像所需的 RAM，即 50 × 4 × 200 × 300 × 3 = 3600 万字节（36 MB），加上模型参数所需的 RAM，约为 3.6 MB（之前计算）， 加上一些用于梯度的 RAM（我们将忽略它，因为它可以随着反向传播在反向传播过程中向下传播而逐渐释放）。 我们总共大约有 525 + 36 + 3.6 = 564.6 MB，这确实是一个乐观的最低限度。

Exercise 3

如果你的GPU在训练CNN时耗尽了内存，五件可以解决问题的事？

答案：

如果您的 GPU 在训练 CNN 时内存不足，您可以尝试以下五种方法来解决问题（除了购买具有更多内存的 GPU 之外）：

1. 减少小批量大小。
2. 在一层或多层中使用更大的步幅来降低维度。
3. 移除一层或多层。
4. 使用 16 位浮点数而不是 32 位浮点数。
5. 将 CNN 分布在多个设备上。

Exercise 4

为什么要添加最大池化层，而不是具有相同步幅的卷积层？

答案：

最大池化层根本没有参数，而卷积层有相当多的参数（见前面的问题）。

Exercise 5

您什么时候想添加局部响应规范化层？

答案：

局部响应规范化层使最强烈激活的神经元在同一位置但在相邻的特征图中抑制神经元，这鼓励不同的特征图专门化并将它们推开，迫使它们探索更广泛的特征。 它通常用于较低层，以拥有更大的低层特征池，供上层构建。

Exercise 6

你能说出AlexNet与LeNet-5相比的主要创新吗？那么GoogLeNet、ResNet、SENet、Xception和EfficientNet的主要创新如何呢？

答案：

与 LeNet-5 相比，AlexNet 的主要创新在于它更大更深，并且将卷积层直接堆叠在彼此之上，而不是在每个卷积层之上堆叠池化层。

GoogLeNet 的主要创新是引入了初始模块，这使得网络比以前的 CNN 架构更深，参数更少。

ResNet 的主要创新是引入了跳跃连接，这使得远远超过 100 层成为可能。 可以说，它的简单性和一致性也颇具创新性。

SENet 的主要创新是在初始网络中的每个初始模块或 ResNet 中的每个残差单元之后使用 SE 块（双层密集网络）来重新校准特征图的相对重要性。

Xception 的主要创新是使用深度可分离的卷积层，它分别查看空间模式和深度模式。

最后，EfficientNet 的主要内涵是复合缩放方法，以有效地将模型缩放到更大的计算预算。

Exercise 7

什么是全卷积网络？如何将密集层转换为卷积层？

答案：

全卷积网络是完全由卷积层和池化层组成的神经网络。 FCN 可以有效地处理任何宽度和高度（至少大于最小尺寸）的图像。 它们对于对象检测和语义分割最有用，因为它们只需要查看图像一次（而不是必须在图像的不同部分多次运行 CNN）。

如果你有一个顶部有一些密集层的 CNN，你可以将这些密集层转换为卷积层来创建一个 FCN：只需用一个卷积层替换最低的密集层，其内核大小等于该层的输入大小，密集层中的每个神经元带有一个过滤器，并使用“valid”填充。 通常步幅应为 1，但如果需要，可以将其设置为更高的值。 激活函数应该与密集层的相同。 其他密集层应该以相同的方式转换，但使用 1×1 过滤器。 实际上，可以通过适当地重塑密集层的权重矩阵，以这种方式转换训练过的 CNN。

Exercise 8

语义分割的主要技术难点是什么？

答案：

语义分割的主要技术难点在于，当信号流过每一层时，CNN 中会丢失大量空间信息，尤其是在池化层和步幅大于 1 的层中。这些空间信息需要恢复以某种方式准确预测每个像素的类别。

Exercise 9

从零开始建立你自己的CNN，并试图在MNIST上达到最高精度。

答案：

In [1]:
```
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
from pathlib import Path
from time import strftime
import tensorboard
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example
```

In [5]:
```
mnist = tf.keras.datasets.mnist.load_data()

(X_train_full, y_train_full), (X_test, y_test) = mnist

X_train_full = X_train_full / 255.
X_test = X_test / 255.

X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]

# 添加新的一列
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

In [13]:
```
tf.keras.backend.clear_session()

tf.random.set_seed(42)
np.random.seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, kernel_size=3, padding="same",
                           activation="relu",
                           kernel_initializer="he_normal"),
    tf.keras.layers.Conv2D(64, kernel_size=3, padding="same",
                           activation="relu",
                           kernel_initializer="he_normal"),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Dense(128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="nadam",
              metrics=["accuracy"])

model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid))

model.evaluate(X_test, y_test)
```

```
Epoch 1/10
1719/1719 [==============================] - 95s 55ms/step - loss: 0.1909 - accuracy: 0.
9420 - val_loss: 0.0471 - val_accuracy: 0.9882
Epoch 2/10
1719/1719 [==============================] - 96s 56ms/step - loss: 0.0839 - accuracy: 0.
9743 - val_loss: 0.0475 - val_accuracy: 0.9850
Epoch 3/10
1719/1719 [==============================] - 97s 57ms/step - loss: 0.0610 - accuracy: 0.
9812 - val_loss: 0.0428 - val_accuracy: 0.9894
Epoch 4/10
1719/1719 [==============================] - 89s 52ms/step - loss: 0.0493 - accuracy: 0.
9844 - val_loss: 0.0434 - val_accuracy: 0.9908
```

```
Epoch 5/10
1719/1719 [==============================] - 88s 51ms/step - loss: 0.0424 - accuracy: 0.
9869 - val_loss: 0.0376 - val_accuracy: 0.9908
Epoch 6/10
1719/1719 [==============================] - 93s 54ms/step - loss: 0.0353 - accuracy: 0.
9890 - val_loss: 0.0404 - val_accuracy: 0.9924
Epoch 7/10
1719/1719 [==============================] - 89s 52ms/step - loss: 0.0318 - accuracy: 0.
9897 - val_loss: 0.0391 - val_accuracy: 0.9914
Epoch 8/10
1719/1719 [==============================] - 87s 51ms/step - loss: 0.0283 - accuracy: 0.
9906 - val_loss: 0.0370 - val_accuracy: 0.9920
Epoch 9/10
1719/1719 [==============================] - 87s 51ms/step - loss: 0.0242 - accuracy: 0.
9922 - val_loss: 0.0444 - val_accuracy: 0.9920
Epoch 10/10
1719/1719 [==============================] - 92s 53ms/step - loss: 0.0223 - accuracy: 0.
9928 - val_loss: 0.0408 - val_accuracy: 0.9918
313/313 [==============================] - 2s 6ms/step - loss: 0.0329 - accuracy: 0.9913
Out[13]: [0.03290453553199768, 0.9912999868392944]
```

Exercise 10

使用迁移学习进行大型图像分类，通过以下步骤：

1. 创建一个每个类至少包含100张图像的训练集。例如，你可以根据地理位置（海滩、山、城市等）对你自己的图片进行分类，或者，您也可以使用现有的数据集（例如，来自TensorFlow Datasets）。
2. 将其分为训练集、验证集和测试集。
3. 构建输入pipeline，应用适当的预处理操作，并可选择添加数据增强。
4. 在这个数据集上微调一个预先训练过的模型。

答案：

参考之前Flowers的例子。

Exercise 11

浏览TensorFlow的Style Transfer tutorial（风格迁移）。这是一种使用深度学习产生艺术的有趣方式。