华东师范大学数据科学与工程学院实验报告

课程名称：当代人工智能	**年级：**2021 级	**上机实践成绩：**
指导教师：李翔	**姓名：**李睿恩	**学号：**10215501434
上机实践名称：图像分类及经典 CNN 实现		**上机实践日期：**2023.11.23
上机实践编号：3	**组号：**无	**上机实践时间：**8:30

一、 实验目的

- 通过使用 CNN 搭建神经网络，完成图像分类任务。

二、 实验任务

- 导入 MNIST 数据集，对该数据集进行图像分类。
- 使用至少四种 CNN 模型实现图像分类。LeNet，AlexNet，ResNet 为必须使用的架构。

三、 实验环境

本次实验基于 Colaboratory 环境，调用其上的 T4 GPU 完成。Colaboratory 是由 Google 开发的一种基于云端的交互式环境，提供了免费的计算资源（包括 CPU、GPU 和 TPU），可让用户在浏览器中编写和执行代码，而无需进行任何配置和安装。

本次实验基于 Python 的编程语言，采用了 TensorFlow 模块，搭建了所有神经网络。

四、 实验过程

4.1 实验准备

无论我们在搭建何种架构，有一些前置准备操作总是必须事先完成的。

在本次实验中，我们将采用 TensorFlow 模块以实现神经网络的搭建。为了配合 TensorFlow 的使用，我们还会调用 numpy 模块进行数据处理。图 1 展示了导入的必要模块。

```
1  import numpy as np
2  import tensorflow as tf
3  from tensorflow.keras import layers, models
4  from tensorflow.keras.datasets import mnist
5  from tensorflow.keras.utils import to_categorical
```

图 1 导入必要模块

本次实验使用了 MNIST 数据集以完成图像分类任务。我们调用了 TensorFlow 中已经存在的 MNIST 数据集以完成实验。TensorFlow 中的 MNIST 数据集已经分好了训练集与测试集。图 2 为导入 MNIST 数据集以及观测其中数据的形状，可以发现我们在训练集中有 60000 张规模为 28×28 的图片，测试集中则有 10000 张同等规模的图片。

```
# These variables are all in type of numpy.
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print(train_images.shape)
print(train_labels.shape)
print(test_images.shape)
print(test_labels.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

图 2 导入 MNIST 数据集

MNIST 数据集的特点在于每张图片的大小均为 28×28，这对神经网络来说规模过小，神经网络可能无法搭建过深。为了能够更好地搭建适配的神经网络，并且为了可以让过滤器捕获到图像边缘的特征，我们用 0 来对图像边缘进行填充，使其扩展到了 32×32。这个流程被称为填充（Padding）。

另外，我们还将训练集与测试集的标签转化为了 One-hot 编码。这主要是因为，我们搭建的神经网络的最后一层通常都是基于 Softmax 或 Sigmoid 激活函数的生成 10 个神经元的全连接层，每一个神经元都是一个 0 到 1 之间的实数。这意味着，对于每张图片而言，最后我们得到的实际上是它属于每一类的概率。一个符合直觉的认知是，一张图片被预测属于正确类别的概率越高，说明其准确率越高，因此我们将所有正确的标签转化为了 One-hot 编码，将图片通过神经网络得到的结果与正确的 One-hot 编码作对比，从而得到最后的准确率。具体操作见图 3.

接下来我们将进行各神经网络的搭建。

```
train_images_32 = np.zeros((60000, 32, 32), dtype=train_images.dtype)
test_images_32 = np.zeros((10000, 32, 32), dtype=test_images.dtype)

start_row = (32 - 28) // 2
start_col = (32 - 28) // 2
for i in range(60000):
  train_images_32[i][start_row:start_row+28, start_col:start_col+28] = train_images[i]
for i in range(10000):
  test_images_32[i][start_row:start_row+28, start_col:start_col+28] = test_images[i]

train_images_32 = train_images_32.reshape((60000, 32, 32, 1)).astype('float32') / 255
test_images_32 = test_images_32.reshape((10000, 32, 32, 1)).astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

print(train_images_32.shape)
print(test_images_32.shape)
print(train_labels.shape)
print(test_labels.shape)
```

图 3 图像与标签的数据预处理

4. 2 LeNet

LeNet 是一个在 1998 年由 Yann LeCun 创建的卷积神经网络架构，被广泛用于手写数字识别中。由于它本身就被应用于 MNIST 数据集，因此我们可以非常准确地还原 LeNet 架构。它的架构如表 1 所示。

注意，由于实际使用的激活函数并不是决定神经网络架构最重要的因素，而且激活函数也并没有任何强制的限制，因此我们在本次实验中描述架构的图表中均没有标出激活函数，

但实际上在每个卷积层与全连接层后，均应该有激活函数作为非线性层。

表 1 LeNet 架构

层	类型	图	尺寸	内核尺寸	步幅
OUT	全连接	--	10	--	--
F6	全连接	--	84	--	--
C5	卷积	120	1×1	5×5	1
S4	平均池化	16	5×5	2×2	2
C3	卷积	16	10×10	5×5	1
S2	平均池化	6	14×14	2×2	2
C1	卷积	6	28×28	5×5	1
In	输入	1	32×32	--	--

代码实现如图 4。

```
model = models.Sequential()
model.add(layers.Conv2D(6, (5, 5), activation='relu', input_shape=(32, 32, 1)))
model.add(layers.AveragePooling2D((2,2)))
model.add(layers.Conv2D(16, (5, 5), activation='relu'))
model.add(layers.AveragePooling2D((2, 2)))
model.add(layers.Conv2D(120, (5, 5), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(84, activation='relu'))
model.add(layers.Dense(10, activation='sigmoid'))
```

图 4 LeNet 实现代码

在本次实验中，我们为每一个神经网络编译的模型都是选用 categorical_crossentropy 作为损失函数，采用 Adam 的优化器，以默认值（0.001）作为学习率的初始值。我们会将模型应用于我们处理过的训练集中，并要求从我们处理好的数据中取出 80%作为真正的训练集，剩余的 20%作为验证集，要求对该训练集的学习要进行 5 轮（epochs=5），要求该训练集在每次权重更新之前都只是看到 64 个样本（batch_size=64）。在模型训练完毕后，我们将其用于测试集，观察其最终的结果。

在 LeNet 的实现中，我们经历上述步骤，训练耗时 30.53 秒，在测试集上得到了 98.9%的准确率。这说明我们的 LeNet 搭建效果较好。

4. 3 AlexNet

AlexNet 的 CNN 架构以大比分赢得了 2012 年的 ILSVRC 竞赛。它和 LeNet 架构很相似，只是比 LeNet 更大更深。它直接将卷积层堆叠到其它层之上，而不是在每个卷积层之上堆叠池化层。值得注意的是，由于 AlexNet 并不是为 MNIST 数据集量身打造的，因此我们无法完全还原其原版架构，但我们仍然可以基于其神经网络层的排列顺序，将它改造为适合 MNIST 数据集的神经网络架构。

我们所实际使用的架构如表 2 所示。为了能够适配 MNIST 数据集，我们没有遵循传统架构，而是改变了其除去层的类型与顺序以外的几乎所有内容。

表 2 AlexNet 架构

层	类型	图	尺寸	内核尺寸	步幅	填充
OUT	全连接	--	10	--		--
F9	全连接	--	84	--		--
F8	全连接	--	84	--		--
C7	卷积	64	8×8	3×3	1	1
C6	卷积	128	8×8	3×3	1	1
C5	卷积	128	8×8	3×3	1	1
S4	最大池化	64	8×8	2×2	2	0
C3	卷积	64	16×16	3×3	1	1
S2	最大池化	32	16×16	2×2	2	0
C1	卷积	32	32×32	3×3	1	1
In	输入	1	32×32	--	--	--

一个过深的神经网络很可能会导致对训练集的过拟合。因此，除去层数的增加，AlexNet 的另一个非常重要的特性是其使用了淘汰（Dropout）策略与数据增强（Data Augmentation）策略。淘汰策略指的是神经网络会在某一层随机删除一些输出的神经元，从而减少特征的捕获，该方法可以有效地防止过拟合。数据增强策略指的是将图片进行各种偏移、翻转、改变光照条件等操作，随机移动训练数据。

在我们的神经网络搭建中，我们使用了淘汰策略，但我们没有使用数据增强策略。保留淘汰策略，不但是为了尽可能还原 AlexNet，更是为了防止对训练集的过拟合。但我们没有使用数据增强策略，则是考虑到了 MNIST 数据集本身的特性。MNIST 数据集是手写数字数据集，它的分类非常依赖于图像中数字的形状与位置，一些数据增强操作很可能会对数据集本身的性质造成毁灭性的打击，比如如果将数字 6 旋转 180 度，得到的数字就会变成 9；对数字 8 进行大幅度的偏移，可能会导致该数字变为数字 3……这都必然会导致模型的训练受到出乎意料的影响。为此，我们并没有使用数据增强策略。

AlexNet 的另一个特性在于它使用了局部相应归一化（Local Response Normalization, LRN）。它主要被用于激活函数后的归一化过程，它对局部神经元的响应进行归一化，使得响应较大的神经元抑制响应较小的神经元，从而进一步增强了模型的泛化性能。

代码实现如图 5.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 1)))
model.add(layers.Lambda(lambda x: tf.nn.local_response_normalization(x, depth_radius=2, bias=1.0, alpha=0.00002, beta=0.75)))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.Lambda(lambda x: tf.nn.local_response_normalization(x, depth_radius=2, bias=1.0, alpha=0.00002, beta=0.75)))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))

model.add(layers.Flatten())
model.add(layers.Dense(84, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(84, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))
```

图 5 AlexNet 实现代码

模型经过了 52.06 秒的训练后，我们将模型用于测试集，最终得到了 98.8%的准确率。这说明我们的 AlexNet 搭建较好。

4. 4 ResNet

ResNet 是一个由深度学习领域的专家何恺明等人开发并获得 2015 年 ILSVRC 竞赛冠军的残差网络。原版的神经网络使用了一个由 152 层组成的非常深的卷积神经网络。能够训练如此深层的网络的关键是使用了跳过连接（快捷连接）：输入到一个层中的信号也被添加到位于堆栈上方的层的输出端。

直观来看，一个更深的神经网络似乎预测效果会比一个浅层的神经网络要好。但是，在实际实验中，一个更深的神经网络的预测效果远不如一个浅层的神经网络。这一方面是因为梯度消失与梯度爆炸问题，另一方面则是因为过深的神经网络会造成神经网络的负优化问题，不但没有提升训练精度，还扭曲了网络空间，升高了训练集上的损失函数值。为此，我们希望深层神经网络是恒等映射，并希望深层神经网络可以相比于恒等映射更进一步产生优化。因此，我们诞生了残差单元的思想。

对于较为浅层的 ResNet 而言，残差单元会主要由输入端、两个卷积层与一个输出端构成。在残差单元中，输入 x 会被添加到网络的输出端，求和后的结果才是被激活函数执行的对象。这样的架构可以确保，如果神经网络即将走向负优化，那么先前的最优解仍然可以被保留在输出的结果中，缓解了梯度问题。何恺明原作论文中的残差单元的结构见图 6.

但在实际执行时，残差单元的第一个卷积层后紧跟着的分批归一化（Batch Normalization）与 ReLU 激活层，第二个卷积层后紧跟着的分批归一化。在输入数据经过多层神经网络后，数据会发生内部协变量偏移（Internal Covariate Shift），导致数据分布发生变化，而使用了分批归一化后，我们可以使得网络更容易学到适应数据的有效数据，并且间接减少过拟合。

基于此，我们可以学习传统的 ResNet 架构。传统的 ResNet 架构由输入层、卷积层、最大池化层、平均池化层、全连接层构成，其中有大量的卷积层实质上构成了上述所提到的残差单元。由于 MNIST 数据集规模较小，并不会适应非常深层的神经网络，因此我们在这里搭建 ResNet-18 以对其进行还原。各层数下的 ResNet 架构如图 7.

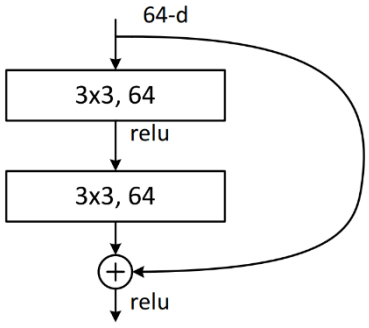

图 6 残差单元的结构

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 7 各层数下的 ResNet 架构

以下是我们对 ResNet-18 的搭建。其中，图 8 所示的 residual_block 是一个函数，它封装

了残差单元，实现了残差单元的功能。图 9 所示的 Resnet_18 函数则根据图 7 中 18-layer 所示的方式搭建了一个 ResNet-18。值得注意的是，图 7 中的这些卷积层之间存在一些通道数量的改变，因此我们需要在卷积层通道数变化的时刻注意通道数量的变换。

另外，我们删去了传统流程中的最后两层平均池化层，这一方面是因为 MNIST 数据集本身图像过小，一步步地卷积、激活、池化已经让我们得到的特征规模极小，已经不能再进一步地降维了，另一方面是这并不是 ResNet 最核心的部分。

```
def residual_block(x, filters, stride=1):
  shortcut = x

  x = layers.Conv2D(filters, 3, strides=stride, padding='same')(x)
  x = layers.BatchNormalization()(x)
  x = layers.ReLU()(x)

  x = layers.Conv2D(filters, 3, strides=1, padding='same')(x)
  x = layers.BatchNormalization()(x)

  if stride != 1 or shortcut.shape[-1] != filters:
    shortcut = layers.Conv2D(filters, 1, strides=stride, padding='same')(shortcut)
    shortcut = layers.BatchNormalization()(shortcut)

  x = layers.add([x, shortcut])
  x = layers.ReLU()(x)

  return x
```

图 8 残差单元

```
def Resnet_18(input_shape=(32, 32, 1), num_classes=10):
  input_tensor = tf.keras.Input(shape=input_shape)

  x = layers.Conv2D(64, 7, strides=2, padding='same')(input_tensor)
  x = layers.ReLU()(x)
  x = layers.MaxPooling2D(3, strides=2, padding='same')(x)

  residual_blocks = [2, 2, 2, 2]
  filters_list = [64, 128, 256, 512]

  for residual, num_blocks, filters in zip(range(len(residual_blocks)), residual_blocks, filters_list):
    for block in range(num_blocks):
      stride = 2 if residual > 0 and block == 0 else 1
      x = residual_block(x, filters, stride=stride)

  # x = layers.AveragePooling2D((7, 7), padding='valid', strides=1)(x)
  # x = layers.AveragePooling2D((2, 2), padding='valid', strides=1)(x)
  x = layers.Flatten()(x)
  x = layers.Dense(1000, activation='relu')(x)
  output = layers.Dense(num_classes, activation='softmax')(x)

  model = models.Model(inputs=input_tensor, outputs=output)
  return model
```

图 9 ResNet-18 实现代码

模型经过 148.23 秒的训练后，我们将模型用于测试集，最终得到了 98.1%的准确率。这说明我们的 ResNet 搭建较好。

4. 5 *VGGNet

VGGNet 曾被称为是通过简单堆叠卷积构建网络的巅峰之作。它是由牛津大学的 Visual Geometry Group 提出的一种架构，该架构获得了 2014 年 ILSVRC 分类任务的亚军以及定位任务的冠军。它的实质是使用了比 AlexNet 更小的卷积核和更深的网络。

它的架构主要是简单的由卷积层、池化层和全连接层的堆积，具体架构见图 10.

我们原计划实现一个目前较为常见的 VGGNet-16。但是，令我们很遗憾的是，由于其中涉及了过多的池化层，图片的维数逐渐越来越小，直到最后图片的维数已经无法实现全部的

最大池化层。如果强行用 0 在图片周围进行填充，那么模型在验证集上的准确率就会只有10%左右，这意味着神经网络没有学习到任何特征。因此，我们将最后一组卷积层与最大池化层舍弃了。并且，考虑到图片的特征过少，我们也删去了一层全连接层。

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 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 10 各层数下的 VGGNet

代码实现如图 11.

```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 1)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 1)))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

# model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
# model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
# model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
# model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

model.add(layers.Flatten())
# model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

图 11 VGGNet 实现代码

模型训练了 143.97 秒后，我们将模型用于测试集，最终得到了 99%的准确率。这说明我们的 VGGNet 搭建较好。

4. 6 *GoogLeNet

GoogLeNet 架构是由来自 Google 研究部的 Christian Szegedy 等人开发的，赢得了 2014 年的 ILSVRC 挑战。这一伟大的性能很大程度上源于它的网络比以前的 CNN 更深。这是通过一个被称为初始化模块的子网使之成为可能。

初始化模块指的是，一个输入的内容会分别给到四个神经网络层中。这四层分别服从：

A. 使用了 1×1 内核，步幅为 1，填充为 1 的卷积层。

B. 使用了 1×1 内核，步幅为 1，填充为 1 的卷积层，之后再使用了一次 3×3 内核，步幅为 1，填充为 1 的卷积层。

C. 使用了 1×1 内核，步幅为 1，填充为 1 的卷积层，之后再使用了一次 5×5 内核，步幅为 1，填充为 1 卷积层。

D. 使用了 3×3 内核，步幅为 1，填充为 1 的最大池化层，之后再使用了一次 1×1 内核，步幅为 1，填充为 1 的卷积层。

四层得到的结果会进入最后的深度级联层（Depth Concatenation Layer），即从所有的四个顶部的卷积层堆叠特征图。整个的一个完整模块被称为初始化模块。这样做的好处是可以捕捉到不同尺寸的图像模式。

它的原始架构还采用了局部响应归一化与 Dropout 操作提升泛化能力，如图 12 所示。

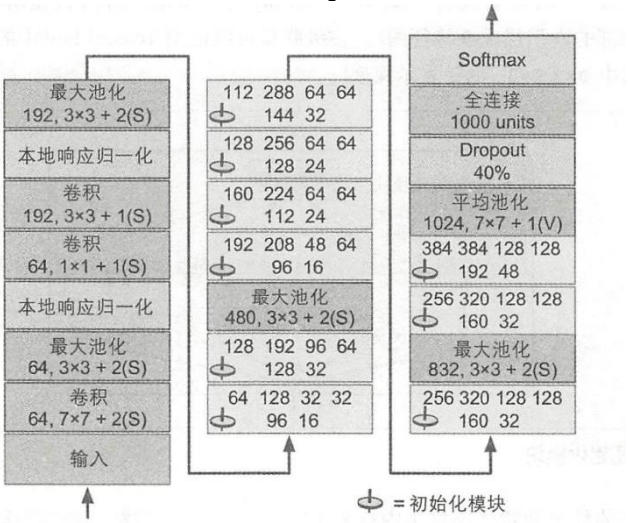

图 12 GoogLeNet 原始架构

然而，其中有一些卷积层与池化层存在的目的在于减少计算量、增加计算速度，使用大量的初始化模块也是为了学习到更丰富的特征。然而，手写数字集本身规模已经很小，过多的层数不但可能无法实现，过多的层数也可能导致神经网络的恶化，训练效果极差。为此，尽管我们会保留大部分的初始化模块、局部响应归一化与 Dropout 操作作为核心，但我们还是删去了部分层。实现代码如图 13.

模型经过了 507.73 秒的训练后，我们将模型用于测试集，最终得到了 99%的准确率。这说明我们的 GoogLeNet 搭建较好。

```python
# 初始化模块与深度级联
def inception_module(x, filters):
  conv1_1_1_1 = layers.Conv2D(filters[0], (1, 1), padding='same', activation='relu')(x)
  conv1_1_1_2 = layers.Conv2D(filters[1], (1, 1), padding='same', activation='relu')(x)
  conv3_3_1 = layers.Conv2D(filters[2], (3, 3), padding='same', activation='relu')(conv1_1_1_2)
  conv1_1_1_3 = layers.Conv2D(filters[3], (1, 1), padding='same', activation='relu')(x)
  conv5_5_1 = layers.Conv2D(filters[4], (5, 5), padding='same', activation='relu')(conv1_1_1_3)
  maxpool = layers.MaxPooling2D((3, 3), strides=(1, 1), padding='same')(x)
  conv1_1_1_4 = layers.Conv2D(filters[5], (1, 1), padding='same', activation='relu')(maxpool)

  # 深度级联
  inception = tf.concat([conv1_1_1_1, conv3_3_1, conv5_5_1, conv1_1_1_4], axis=-1)

  return inception
# 搭建神经网络
def GoogLeNet_model(input_shape=(32, 32, 1), num_classes=10):
  input_tensor = layers.Input(shape=input_shape)
  # x = layers.Conv2D(64, (7, 7), padding='same', activation='relu', strides=(2, 2))(input_tensor)
  # x = layers.MaxPooling2D((3, 3), padding='same', strides=(2, 2))(x)
  # x = layers.Lambda(lambda x: tf.nn.local_response_normalization(x))(x)
  x = layers.Conv2D(64, (1, 1), padding='same', activation='relu', strides=(1, 1))(input_tensor)
  x = layers.Conv2D(192, (3, 3), padding='same', activation='relu', strides=(1, 1))(x)
  x = layers.Lambda(lambda x: tf.nn.local_response_normalization(x))(x)
  x = layers.MaxPooling2D((3, 3), padding='same', strides=(2, 2))(x)

  x = inception_module(x, [64, 96, 128, 16, 32, 32])
  x = inception_module(x, [128, 128, 192, 32, 96, 64])
  x = layers.MaxPooling2D((3, 3), padding='same', strides=(2, 2))(x)
  x = inception_module(x, [192, 96, 208, 16, 48, 64])
  # x = inception_module(x, [160, 112, 224, 24, 64, 64])
  # x = inception_module(x, [128, 128, 256, 24, 64, 64])
  # x = inception_module(x, [112, 144, 288, 32, 64, 64])
  x = inception_module(x, [256, 160, 320, 32, 128, 128])

  x = layers.MaxPooling2D((3, 3), padding='same', strides=(2, 2))(x)
  x = inception_module(x, [256, 160, 320, 32, 128, 128])
  x = inception_module(x, [384, 192, 384, 48, 128, 128])
  x = layers.AveragePooling2D((2,2), padding='valid', strides=(1, 1))(x)
  x = layers.Dropout(0.4)(x)
  x = layers.Flatten()(x)
  x = layers.Dense(512, activation='relu')(x)
  output = layers.Dense(num_classes, activation='softmax')(x)
```

图 13 GoogLeNet 的实现代码

4. 7 *InceptionV2

实际上，GoogLeNet 还有一个名称 InceptionV1，这主要是因为它创新性地提出了深度级联层。在之后的一年，Google 研究部的成员就开发出了 InceptionV2 神经网络架构。它与 GoogLeNet 的区别在于，它使用了两个 3×3 的卷积层代替了一个 5×5 的卷积层，减少了计算量，并且在当时创新性地提出了分批归一化的方式，让模型更有效地学习数据，并且间接减少了数据的过拟合。

主要修改的地方如图 14 所示。

```python
# 初始化模块与深度级联
def inception_module(x, filters):
  conv1_1_1_1 = layers.Conv2D(filters[0], (1, 1), padding='same', activation='relu')(x)
  conv1_1_1_2 = layers.Conv2D(filters[1], (1, 1), padding='same', activation='relu')(x)
  conv3_3_1 = layers.Conv2D(filters[2], (3, 3), padding='same', activation='relu')(conv1_1_1_2)
  conv1_1_1_3 = layers.Conv2D(filters[3], (1, 1), padding='same', activation='relu')(x)
  conv5_5_1 = layers.Conv2D(filters[4], (3, 3), padding='same', activation='relu')(conv1_1_1_3)
  conv5_5_2 = layers.Conv2D(filters[4], (3, 3), padding='same', activation='relu')(conv5_5_1)
  maxpool = layers.MaxPooling2D((3, 3), strides=(1, 1), padding='same')(x)
  conv1_1_1_4 = layers.Conv2D(filters[5], (1, 1), padding='same', activation='relu')(maxpool)

  conv1_1_1_1 = layers.BatchNormalization()(conv1_1_1_1)
  conv3_3_1 = layers.BatchNormalization()(conv3_3_1)
  conv5_5_2 = layers.BatchNormalization()(conv5_5_2)
  conv1_1_1_4 = layers.BatchNormalization()(conv1_1_1_4)

  # 深度级联
  inception = tf.concat([conv1_1_1_1, conv3_3_1, conv5_5_2, conv1_1_1_4], axis=-1)

  return inception
```

图 14 InceptionV2 的实现代码

模型经过了 572.33 秒的训练后，我们将模型用于测试集，最终得到了 98.6%的准确率。这说明我们 InceptionV2 搭建较好。

五、 总结与分析

我们使用的 6 种卷积神经网络架构在测试集上均有着大于 98%的准确率，这一方面是由于 MNIST 数据集的特征可能过于简单，易于学习，另一方面也意味着我们的神经网络搭建较为成功，结果对比见表 3。但是，在搭建的过程中，我们还是遇到了不少的问题。

表 3 不同卷积神经网络架构在 MNIST 数据集的表现

架构	规模	测试集准确率	训练时间（秒）
LeNet-5	相对较浅	98.9%	30.53
AlexNet	相对较浅	98.8%	52.06
ResNet-18	相对较深	98.1%	148.23
VGGNet-16	相对较浅	99.0%	143.97
GoogLeNet	相对较深	99.0%	507.73
InceptionV2	相对较深	98.6%	572.33

有的神经网络在搭建的过程中会提示有关矩阵规模的问题，这有可能是因为我们使用了过多的池化层导致的图像规模的降维，而当我们降维到一定程度后，图片就会小到无法再进一步降维（如 ResNet，VGGNet，GoogLeNet、InceptionV2），此时我们需要删去一些池化层。有的神经网络会在我们搭建原版架构时在验证集上展现出仅仅 10%左右的准确率（如 GoogLeNet），而实际上，随机地猜测所有图片的分类，其期望准确率也是 10%，这说明神经网络完全没有学习到图像的特征，这则可能是因为神经网络层数过深导致了梯度消失问题，从而导致模型几乎没有被优化，或者是学习率并没有被较好地选择，此时我们需要让神经网络层变浅，或者尝试修改学习率。

尽管这些数据在测试集上均有大于 98%的准确率，但还是可以发现其准确率存在一些微弱的区别。直觉上，我们会认为一个非常深层的神经网络可以学习到更多的特征，从而带来更高的准确率，但在此次试验中，恰恰是一些非常深层的，使用了大量技术的模型在测试集上没有最优秀的表现（如 ResNet、InceptionV2）。这可能是因为过于深层的神经网络会遇到梯度消失、梯度爆炸或负优化问题，最终的训练效果反而不如浅层神经网络；也可能是因为神经网络没有使用正则化等技术提升模型的泛化能力，使得模型还是过于注重训练集本身的特征，导致了一定程度的过拟合；也可能是因为图像规模本身较小，而模型本身不适配于尺寸过小的图片。这启发我们在使用神经网络时，可以不要执着于过于深层的神经网络，而是注重于寻找最合适于特定数据集的网络架构。

并且，过深的神经网络还会带来过大的计算负担，使得训练时间变长，对硬件需求变高。在上述代码中，我们观察到了一些很深层的神经网络的训练时间非常漫长，最长可以超过 550 秒，而如果我们将上述代码全部使用 CPU 运行，那么这可能是一个更加漫长的训练过程。考虑到这个因素，我们应该结合精度与时间综合考虑选择神经网络架构。

综上所示，影响模型结果的主要因素有很多。一方面，虽然直觉上理解，深层神经网络有着更好的表现，但本次实验体现出了一个过深的神经网络可能会导致梯度消失、梯度爆炸、负优化与计算需求过高等问题，因此一个神经网络的深度会非常重要。一方面，神经网络防止过拟合的操作有很多，如淘汰策略，分批归一化等，他们并不总是灵丹妙药，他们反而可能因为对图像特征的学习发生了变化导致训练的结果反而走向欠拟合。一方面，如 LeNet 这样的神经网络从一开始就是专门为 MNIST 数据集这样的小型数据集制作的，而其他神经网络则是基于其他数据集诞生的，这意味着这些神经网络实际上可能对数据集本身存在要求，因此数据数量与数据质量都可能影响模型表现。因此，神经网络架构需要针对特定数据集来选择，并且不可以盲目追求深层神经网络，此外还需要适量地提升泛化能力。

六、 参考文献

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

[3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

[4] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[5] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818-2826).