

《人工智能与深度学习》课程

实验报告

学号：04022212

姓名：钟 源

2022 年 04 月 03 日

实验二：实验环境配置和卷积神经网络

一、实验目的

通过实验了解整个实验环境如何配置，同时利用 Keras 和 Tensorflow 框架学习卷积神经网络（CNN）对 CIFAR-10 数据集进行识别分类。

二、实验内容

完成 CNN 对 CIFAR-10 数据集识别分类。注意给出实验代码和过程介绍。

1. 实验代码补全

1) 标签预处理：

```
def one_hot(label, num_classes):  
    label_one_hot = np.eye(num_classes)[label]  
    return label_one_hot
```

2) 构建网络：

```
from keras import Sequential  
  
from keras.layers import Convolution2D, MaxPooling2D, Dense, Flatten, Dropout  
  
cnn = Sequential()  
  
#unit1
```

```

# 二维卷积层 和 二维最大池化层 交替堆叠

cnn.add(Convolution2D(32, kernel_size=[3, 3], input_shape=(32, 32, 3),
activation='relu', padding='same'))

cnn.add(Convolution2D(32, kernel_size=[3, 3], activation='relu', padding='same'))

cnn.add(MaxPooling2D(pool_size=[2, 2], padding='same'))

cnn.add(Dropout(0.5))

#unit2

# 编写网络的第二部分，可自行尝试增加更多的卷积层，改变通道数、激活函数等

# your code here#

cnn.add(Convolution2D(64, kernel_size=[3, 3], activation='relu', padding='same'))

cnn.add(Convolution2D(64, kernel_size=[3, 3], activation='relu', padding='same'))

cnn.add(MaxPooling2D(pool_size=[2, 2], padding='same'))

cnn.add(Dropout(0.5))

# 展平层

cnn.add(Flatten())

# 全连接层

cnn.add(Dense(512, activation='relu'))

cnn.add(Dropout(0.5))

cnn.add(Dense(128, activation='relu'))

cnn.add(Dropout(0.5))

cnn.add(Dense(10, activation='softmax'))

```

3) 编译模型：

```

cnn.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-
3),loss='categorical_crossentropy', metrics=['acc'])

```

2. 实验过程与结果

根据实验手册，实验可分为以下的几个部分：

1) 加载 CIFAR-10 数据集：

使用 `cifar10.load_data` 函数，加载训练数据及标签 `train_data`、`train_label` 和测试数据及标签 `test_data`、`test_label`。其中是训练数据和测试数据分别是 50000 张和 10000 张 32×32 的彩色图像数据，训练标签和测试标签都是 0~9 的数字，分别代表 10 个类别。

2) 数据和标签预处理：

将数据输入神经网络之前，将数据格式化为经过预处理的浮点数张量。在本实验中，图像数据被读取为 Numpy 的 N 维数组对象 `ndarray`，它是一系列同类型数据的集合，以 0 下标为开始进行集合中元素的索引。需要将 RGB 像素值

(0~255 范围内) 转换为浮点数张量, 并缩放到[0,1] 区间 (神经网络适合处理较小的输入值)。

对于多分类任务, 通常会对标签进行 one-hot 编码, 将其转换为 0 和 1 组成的向量。在本实验中, 总共有 10 类不同的物体, 标签值在 0~9 范围内, 因此 one-hot 编码后的向量长度为 10。举个例子, 原本的标签值[2]会被转化为向量[0, 0, 1, 0, 0, 0, 0, 0, 0, 0], 只有索引为 2 的元素是 1, 其余元素都是 0。

查看预处理后的标签如下:

```
>>> train_label[0:5]
array([6, 9, 9, 4, 1])
>>> x_label[0:5]
array([[0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

3) 构建网络:

本实验中, 将复用相同的总体结构, 即卷积神经网络由 Conv2D 层 (relu 激活) 和 MaxPooling2D 层交替堆叠构成。初始输入的尺寸为(32, 32, 3), 经过两个卷积单元后变为(8, 8, 64), 并经过 Flatten 层打平为向量后送入全连接层。全连接层使用 softmax 激活函数 (大小为 10 的 Dense 层), 将对某个类别的概率进行编码。此外, 还设置了一些 dropout 层, 在训练过程中随机将该层的一些输出特征舍弃 (设置为 0), 来避免过拟合。

查看构建神经网络的详细信息如下:

```
>>> cnn.summary()
Model: "sequential_1"

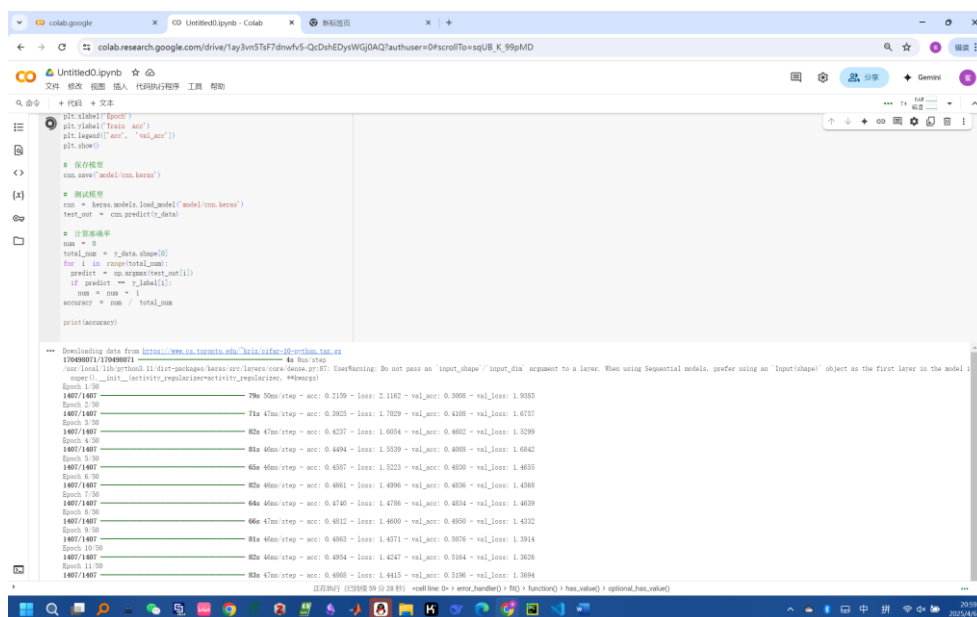
Layer (type)                 Output Shape              Param #
-----
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
conv2d_2 (Conv2D)            (None, 32, 32, 32)        9248
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0
dropout_1 (Dropout)          (None, 16, 16, 32)        0
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
conv2d_4 (Conv2D)            (None, 16, 16, 64)        36928
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)         0
dropout_2 (Dropout)          (None, 8, 8, 64)         0
flatten_1 (Flatten)          (None, 4096)              0
dense_1 (Dense)              (None, 512)              2097664
dropout_3 (Dropout)          (None, 512)              0
dense_2 (Dense)              (None, 128)              65664
dropout_4 (Dropout)          (None, 128)              0
dense_3 (Dense)              (None, 10)               1290
Total params: 2,230,186
Trainable params: 2,230,186
```

4) 编译和训练模型:

在编译模型时, 使用 Adam 优化器, 学习率为 0.001。因为网络最后一层是 softmax 激活函数, 所以使用交叉熵(categorical_crossentropy)作为损失函数。

在训练网络时, 使用 32 个样本组成的小批量, 将模型训练 50 个轮次 (即对 train_data 和 y_train 两个张量中的所有样本进行 50 次迭代)。同时, 分离出 1/10 的训练数据作为验证集, 不对其进行训练, 并且将在每个时期结束时评估此数据的损失和任何模型度量。

由于在本地运行程序较慢, 我们将代码上传至 Google Colab 平台进行运行。



```
plt.xlabel('Epoch')
plt.ylabel('Train acc')
plt.legend(['acc', 'val_acc'])
plt.show()

# 保存模型
cm.save('model/cm.keras')

# 测试模型
cm = keras.models.load_model('model/cm.keras')
test_acc = cm.predict(y_data)

# 计算准确率
num = 0
total_num = y_data.shape[0]
for i in range(total_num):
    predict = np.argmax(test_acc[i])
    num = num + 1 if predict == y_label[i] else num
accuracy = num / total_num
print(accuracy)

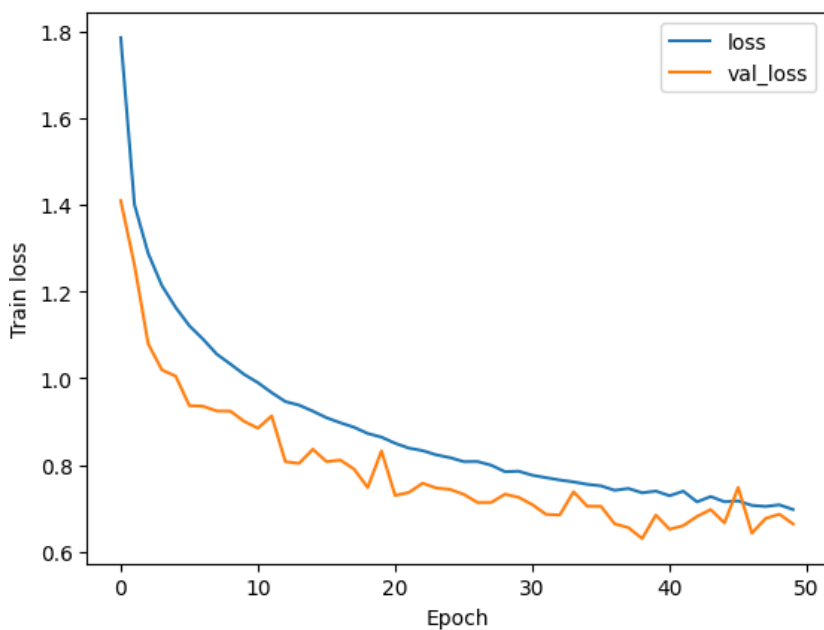
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
15060000/15060000 ————— 4s 400ms/step
Epoch 1/50: 79s 30ms/step - acc: 0.2139 - loss: 2.1162 - val_acc: 0.3908 - val_loss: 1.9085
Epoch 2/50: 71s 47ms/step - acc: 0.3923 - loss: 1.7629 - val_acc: 0.4105 - val_loss: 1.6737
Epoch 3/50: 82s 47ms/step - acc: 0.4237 - loss: 1.6054 - val_acc: 0.4602 - val_loss: 1.5289
Epoch 4/50: 82s 46ms/step - acc: 0.4494 - loss: 1.5339 - val_acc: 0.4888 - val_loss: 1.6842
Epoch 5/50: 65s 46ms/step - acc: 0.4587 - loss: 1.5223 - val_acc: 0.4830 - val_loss: 1.4653
Epoch 6/50: 82s 46ms/step - acc: 0.4881 - loss: 1.4096 - val_acc: 0.4838 - val_loss: 1.4568
Epoch 7/50: 64s 46ms/step - acc: 0.4748 - loss: 1.4798 - val_acc: 0.4834 - val_loss: 1.4639
Epoch 8/50: 66s 47ms/step - acc: 0.4812 - loss: 1.4680 - val_acc: 0.4950 - val_loss: 1.4332
Epoch 9/50: 82s 46ms/step - acc: 0.4883 - loss: 1.4371 - val_acc: 0.5070 - val_loss: 1.3914
Epoch 10/50: 82s 46ms/step - acc: 0.4954 - loss: 1.4247 - val_acc: 0.5164 - val_loss: 1.3626
Epoch 11/50: 82s 47ms/step - acc: 0.4968 - loss: 1.4415 - val_acc: 0.5196 - val_loss: 1.3894
```

最终迭代结果如下图, 可以看到最后得到的训练集损失函数值为 0.7363, 验证集损失函数值为 0.6838; 神经网络的训练精度高达 75.26%, 验证精度高达 73.94%。

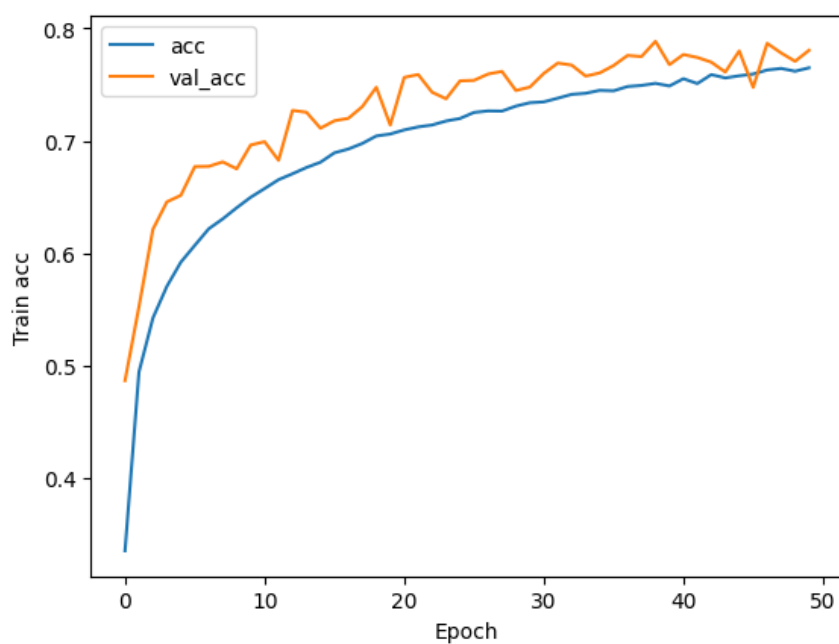
```
1407/1407 ————— 7s 5ms/step - acc: 0.7264 - loss: 0.7981 - val_acc: 0.7518 - val_loss: 0.7110
Epoch 32/50
1407/1407 ————— 10s 5ms/step - acc: 0.7249 - loss: 0.7995 - val_acc: 0.7642 - val_loss: 0.6947
Epoch 33/50
1407/1407 ————— 7s 5ms/step - acc: 0.7320 - loss: 0.8000 - val_acc: 0.7520 - val_loss: 0.7219
Epoch 34/50
1407/1407 ————— 11s 5ms/step - acc: 0.7279 - loss: 0.7902 - val_acc: 0.7546 - val_loss: 0.7173
Epoch 35/50
1407/1407 ————— 10s 5ms/step - acc: 0.7382 - loss: 0.7679 - val_acc: 0.7662 - val_loss: 0.6754
Epoch 36/50
1407/1407 ————— 10s 5ms/step - acc: 0.7306 - loss: 0.7882 - val_acc: 0.7524 - val_loss: 0.7245
Epoch 37/50
1407/1407 ————— 7s 5ms/step - acc: 0.7360 - loss: 0.7782 - val_acc: 0.7558 - val_loss: 0.7231
Epoch 38/50
1407/1407 ————— 7s 5ms/step - acc: 0.7313 - loss: 0.7871 - val_acc: 0.7698 - val_loss: 0.6907
Epoch 39/50
1407/1407 ————— 7s 5ms/step - acc: 0.7339 - loss: 0.7817 - val_acc: 0.7572 - val_loss: 0.7159
Epoch 40/50
1407/1407 ————— 10s 5ms/step - acc: 0.7379 - loss: 0.7686 - val_acc: 0.7634 - val_loss: 0.6936
Epoch 41/50
1407/1407 ————— 10s 5ms/step - acc: 0.7401 - loss: 0.7734 - val_acc: 0.7778 - val_loss: 0.6724
Epoch 42/50
1407/1407 ————— 10s 5ms/step - acc: 0.7381 - loss: 0.7675 - val_acc: 0.7728 - val_loss: 0.6695
Epoch 43/50
1407/1407 ————— 11s 5ms/step - acc: 0.7463 - loss: 0.7519 - val_acc: 0.7606 - val_loss: 0.7067
Epoch 44/50
1407/1407 ————— 7s 5ms/step - acc: 0.7477 - loss: 0.7465 - val_acc: 0.7706 - val_loss: 0.6747
Epoch 45/50
1407/1407 ————— 10s 5ms/step - acc: 0.7467 - loss: 0.7493 - val_acc: 0.7586 - val_loss: 0.7183
Epoch 46/50
1407/1407 ————— 10s 5ms/step - acc: 0.7539 - loss: 0.7412 - val_acc: 0.7706 - val_loss: 0.6827
Epoch 47/50
1407/1407 ————— 10s 5ms/step - acc: 0.7469 - loss: 0.7544 - val_acc: 0.7812 - val_loss: 0.6571
Epoch 48/50
1407/1407 ————— 7s 5ms/step - acc: 0.7463 - loss: 0.7547 - val_acc: 0.7690 - val_loss: 0.6831
Epoch 49/50
1407/1407 ————— 7s 5ms/step - acc: 0.7528 - loss: 0.7330 - val_acc: 0.7728 - val_loss: 0.6892
Epoch 50/50
1407/1407 ————— 6s 5ms/step - acc: 0.7526 - loss: 0.7363 - val_acc: 0.7694 - val_loss: 0.6838
```

5) 绘制损失和精度图:

训练集损失函数以及验证集损失函数随训练次数的变化如下图:



训练集准确率以及验证集准确率随训练次数的变化如下图:



6) 测试模型准确率:

最终将 10000 个测试数据输入模型中, 预测结果并计算准确率, 得到模型的预测准确率为 75.08%。

313/313 ————— 1s 3ms/step
0.7508

三、提高训练

基于实验手册的内容，尝试对现有实验做修改和调整，例如损失函数、激活函数等。比较不同神经网络带来的性能影响，分析猜测其背后的原因。

注 1：以下模型除非特别提及，否则均为在原有模型参数的基础上改动。

注 2：以下语言中的“我们”只是习惯用语，并不代表有多个人的意思。

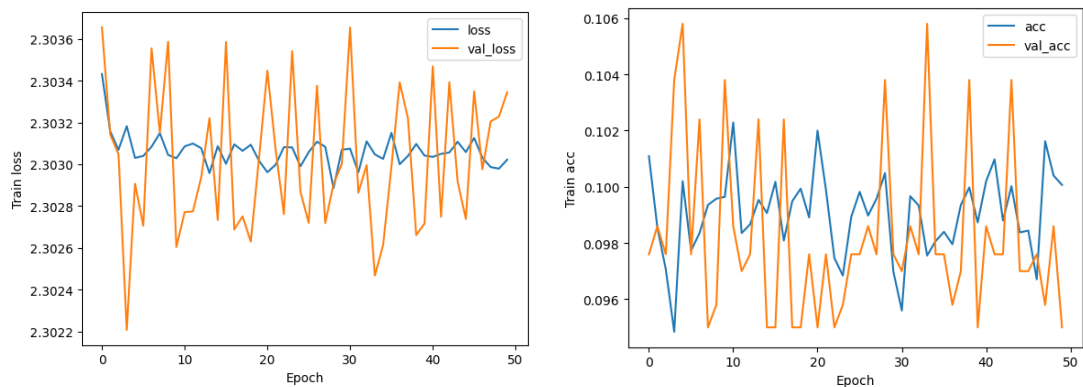
1. 改变学习率

1) 使用较高的学习率：

我们首先尝试改变模型的学习率，当把学习率调整为 $3e-3$ 时，最后几轮的迭代结果如下：

Epoch 40/50	
1407/1407	6s 5ms/step - acc: 0.0979 - loss: 2.3030 - val_acc: 0.0950 - val_loss: 2.3027
Epoch 41/50	
1407/1407	7s 5ms/step - acc: 0.1018 - loss: 2.3030 - val_acc: 0.0986 - val_loss: 2.3035
Epoch 42/50	
1407/1407	10s 5ms/step - acc: 0.1023 - loss: 2.3030 - val_acc: 0.0976 - val_loss: 2.3027
Epoch 43/50	
1407/1407	6s 5ms/step - acc: 0.0994 - loss: 2.3031 - val_acc: 0.0976 - val_loss: 2.3034
Epoch 44/50	
1407/1407	7s 5ms/step - acc: 0.1010 - loss: 2.3030 - val_acc: 0.1038 - val_loss: 2.3029
Epoch 45/50	
1407/1407	7s 5ms/step - acc: 0.0984 - loss: 2.3030 - val_acc: 0.0970 - val_loss: 2.3027
Epoch 46/50	
1407/1407	6s 5ms/step - acc: 0.0988 - loss: 2.3030 - val_acc: 0.0970 - val_loss: 2.3033
Epoch 47/50	
1407/1407	11s 5ms/step - acc: 0.0962 - loss: 2.3029 - val_acc: 0.0976 - val_loss: 2.3030
Epoch 48/50	
1407/1407	7s 5ms/step - acc: 0.1033 - loss: 2.3029 - val_acc: 0.0958 - val_loss: 2.3032
Epoch 49/50	
1407/1407	6s 5ms/step - acc: 0.0975 - loss: 2.3031 - val_acc: 0.0986 - val_loss: 2.3032
Epoch 50/50	
1407/1407	11s 5ms/step - acc: 0.0988 - loss: 2.3030 - val_acc: 0.0950 - val_loss: 2.3033

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示：



最终测试模型的准确率为 10%。

从结果上看，模型训练非常不稳定，最终准确率接近随机猜测。这是符合预期的，我们猜测是因为学习率太高，导致梯度迭代法一直无法收敛，损失函数在最小值附近震荡甚至发散，甚至可能引起梯度爆。

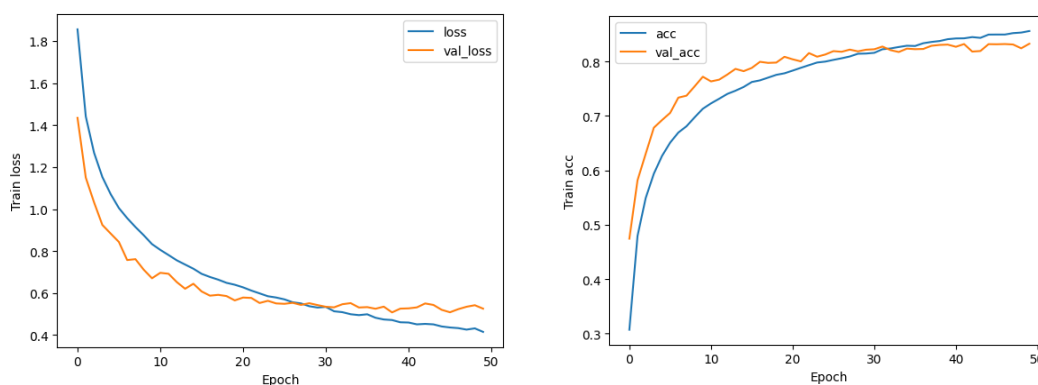
313/313 ————— 1s 3ms/step
0.1

2) 使用较低的学习率:

于是我们便将学习率把学习率调整为 $3e-4$ ，再次训练模型，我们发现在训练的前 10 轮里，准确率就迅速提升到了 77.2%，已经超过了原本模型训练 50 轮的精确度。

```
Epoch 1/50      1407/1407      15s 7ms/step - acc: 0.2210 - loss: 2.0604 - val_acc: 0.4746 - val_loss: 1.4348
Epoch 2/50      1407/1407      14s 5ms/step - acc: 0.4565 - loss: 1.5045 - val_acc: 0.5818 - val_loss: 1.1498
Epoch 3/50      1407/1407      7s 5ms/step - acc: 0.5368 - loss: 1.3003 - val_acc: 0.6304 - val_loss: 1.0320
Epoch 4/50      1407/1407      10s 5ms/step - acc: 0.5802 - loss: 1.1792 - val_acc: 0.6784 - val_loss: 0.9247
Epoch 5/50      1407/1407      10s 5ms/step - acc: 0.6192 - loss: 1.0855 - val_acc: 0.6924 - val_loss: 0.8838
Epoch 6/50      1407/1407      7s 5ms/step - acc: 0.6474 - loss: 1.0135 - val_acc: 0.7056 - val_loss: 0.8435
Epoch 7/50      1407/1407      10s 5ms/step - acc: 0.6630 - loss: 0.9653 - val_acc: 0.7336 - val_loss: 0.7574
Epoch 8/50      1407/1407      10s 5ms/step - acc: 0.6809 - loss: 0.9118 - val_acc: 0.7372 - val_loss: 0.7616
Epoch 9/50      1407/1407      6s 5ms/step - acc: 0.6927 - loss: 0.8855 - val_acc: 0.7544 - val_loss: 0.7113
Epoch 10/50     1407/1407      7s 5ms/step - acc: 0.7124 - loss: 0.8332 - val_acc: 0.7720 - val_loss: 0.6706
```

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示:



最终测试模型的准确率为 81.86%，较原来的模型的准确率有所提升。这说明较小的学习率在此实验中有较好的表现，不过过低的学习率会延长迭代到最优值的时间，甚至有可能陷入局部最优值中。

```
313/313      1s 3ms/step
0.8168
```

2. 改变激活函数

1) 使用 sigmoid 函数:

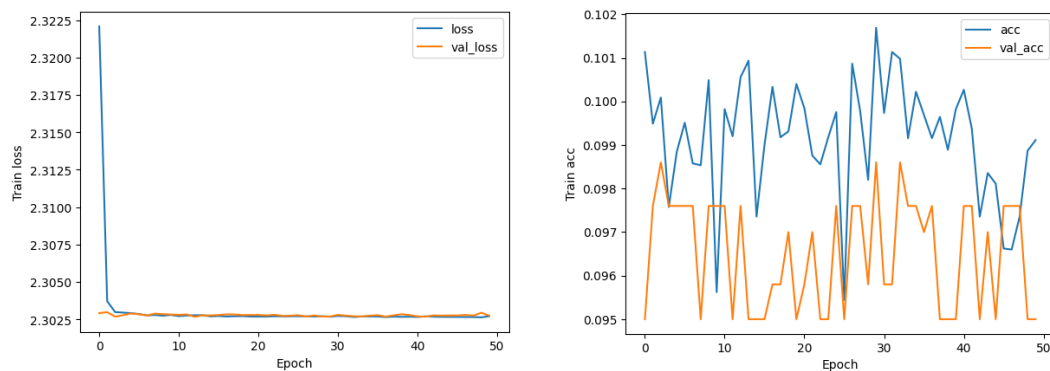
我们尝试将激活函数全部从 ReLU 函数更改为 sigmoid 函数 (Softmax 函数不更改)，发现模型的准确率非常低 (10%，接近随即猜测)。为了排除学习率过大带来的风险，将学习率设为 $1e-4$ 后再次运行，最后几轮的迭代结果如下:


```

Epoch 40/50
1407/1407 ----- 7s 5ms/step - acc: 0.1028 - loss: 2.3026 - val_acc: 0.0950 - val_loss: 2.3028
Epoch 41/50
1407/1407 ----- 11s 5ms/step - acc: 0.1008 - loss: 2.3026 - val_acc: 0.0976 - val_loss: 2.3027
Epoch 42/50
1407/1407 ----- 7s 5ms/step - acc: 0.1015 - loss: 2.3027 - val_acc: 0.0976 - val_loss: 2.3027
Epoch 43/50
1407/1407 ----- 7s 5ms/step - acc: 0.0976 - loss: 2.3027 - val_acc: 0.0950 - val_loss: 2.3028
Epoch 44/50
1407/1407 ----- 7s 5ms/step - acc: 0.0989 - loss: 2.3026 - val_acc: 0.0970 - val_loss: 2.3027
Epoch 45/50
1407/1407 ----- 10s 5ms/step - acc: 0.1002 - loss: 2.3026 - val_acc: 0.0950 - val_loss: 2.3028
Epoch 46/50
1407/1407 ----- 7s 5ms/step - acc: 0.0977 - loss: 2.3026 - val_acc: 0.0976 - val_loss: 2.3028
Epoch 47/50
1407/1407 ----- 10s 5ms/step - acc: 0.0958 - loss: 2.3027 - val_acc: 0.0976 - val_loss: 2.3028
Epoch 48/50
1407/1407 ----- 7s 5ms/step - acc: 0.0985 - loss: 2.3026 - val_acc: 0.0976 - val_loss: 2.3027
Epoch 49/50
1407/1407 ----- 7s 5ms/step - acc: 0.0961 - loss: 2.3027 - val_acc: 0.0950 - val_loss: 2.3029
Epoch 50/50
1407/1407 ----- 11s 5ms/step - acc: 0.0980 - loss: 2.3027 - val_acc: 0.0950 - val_loss: 2.3027

```

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示：



最终测试模型的准确率为 10%。

这可能是因为 sigmoid 是两端饱和函数，相较于 ReLU 函数具有更严重的梯度消失问题，致使模型无法收敛——我们可以看到损失函数刚开始下降得非常快，但随后几乎不再改变。

```

313/313 ----- 2s 5ms/step
0.1

```

2) 使用 swish 函数：

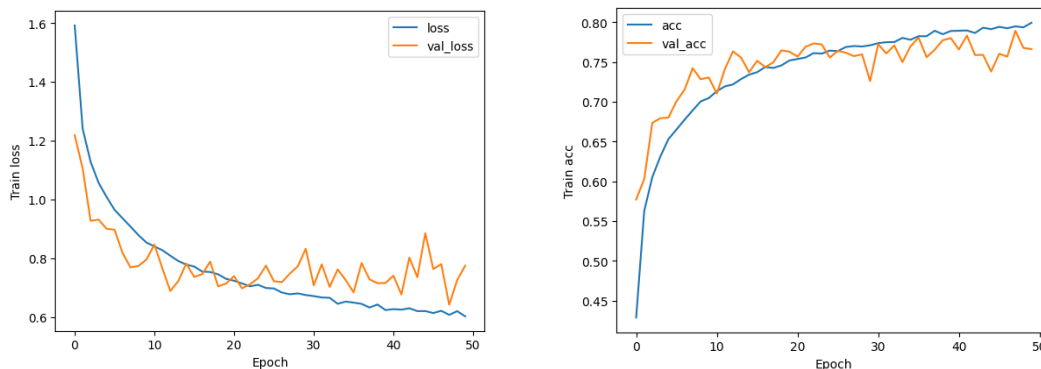
我们又尝试了将激活函数全部从 ReLU 函数更改为 sigmoid 函数 (Softmax 函数不更改)。运行结果如下：

```

Epoch 40/50
1407/1407 ----- 7s 5ms/step - acc: 0.7870 - loss: 0.6188 - val_acc: 0.7800 - val_loss: 0.7156
Epoch 41/50
1407/1407 ----- 7s 5ms/step - acc: 0.7864 - loss: 0.6287 - val_acc: 0.7658 - val_loss: 0.7404
Epoch 42/50
1407/1407 ----- 7s 5ms/step - acc: 0.7908 - loss: 0.6181 - val_acc: 0.7832 - val_loss: 0.6763
Epoch 43/50
1407/1407 ----- 11s 5ms/step - acc: 0.7893 - loss: 0.6313 - val_acc: 0.7588 - val_loss: 0.8020
Epoch 44/50
1407/1407 ----- 7s 5ms/step - acc: 0.7963 - loss: 0.6062 - val_acc: 0.7590 - val_loss: 0.7351
Epoch 45/50
1407/1407 ----- 7s 5ms/step - acc: 0.7925 - loss: 0.6098 - val_acc: 0.7380 - val_loss: 0.8850
Epoch 46/50
1407/1407 ----- 7s 5ms/step - acc: 0.7946 - loss: 0.6079 - val_acc: 0.7604 - val_loss: 0.7631
Epoch 47/50
1407/1407 ----- 7s 5ms/step - acc: 0.7931 - loss: 0.6198 - val_acc: 0.7568 - val_loss: 0.7797
Epoch 48/50
1407/1407 ----- 10s 5ms/step - acc: 0.7942 - loss: 0.6069 - val_acc: 0.7892 - val_loss: 0.6417
Epoch 49/50
1407/1407 ----- 7s 5ms/step - acc: 0.7947 - loss: 0.6142 - val_acc: 0.7678 - val_loss: 0.7257
Epoch 50/50
1407/1407 ----- 7s 5ms/step - acc: 0.8009 - loss: 0.5979 - val_acc: 0.7662 - val_loss: 0.7742

```

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示：



最终测试模型的准确率为 75.45%。这个结果与激活函数是 ReLU 函数时非常接近，这可能是因为 swish 函数近似于 ReLU 函数，可以视作线性函数与 ReLU 函数之间的非线性插值函数。

313/313 ————— 1s 3ms/step
0.7545

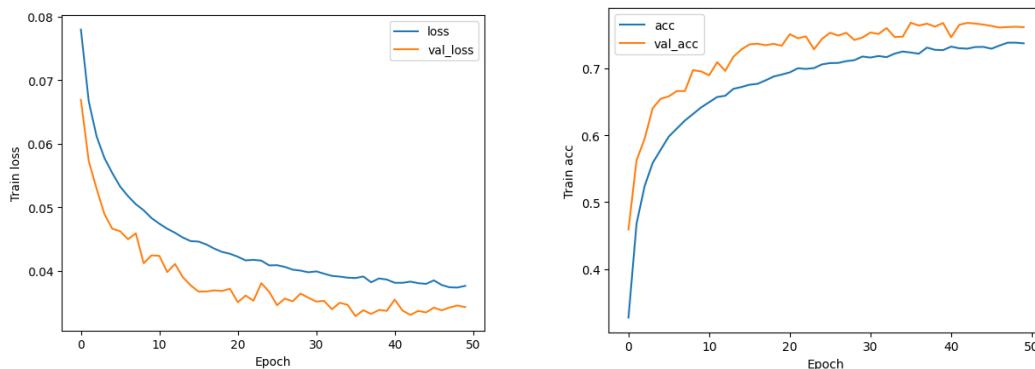
3. 改变损失函数

1) 使用平方损失函数：

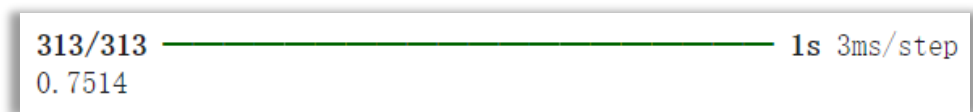
我们接着尝试了将交叉熵损失函数更改为平方损失函数，运行结果如下：

```
Epoch 40/50
1407/1407 ————— 10s 5ms/step - acc: 0.7273 - loss: 0.0388 - val_acc: 0.7680 - val_loss: 0.0338
Epoch 41/50
1407/1407 ————— 7s 5ms/step - acc: 0.7359 - loss: 0.0379 - val_acc: 0.7466 - val_loss: 0.0355
Epoch 42/50
1407/1407 ————— 10s 5ms/step - acc: 0.7310 - loss: 0.0382 - val_acc: 0.7652 - val_loss: 0.0338
Epoch 43/50
1407/1407 ————— 7s 5ms/step - acc: 0.7269 - loss: 0.0385 - val_acc: 0.7682 - val_loss: 0.0331
Epoch 44/50
1407/1407 ————— 7s 5ms/step - acc: 0.7352 - loss: 0.0377 - val_acc: 0.7672 - val_loss: 0.0338
Epoch 45/50
1407/1407 ————— 10s 5ms/step - acc: 0.7312 - loss: 0.0382 - val_acc: 0.7656 - val_loss: 0.0335
Epoch 46/50
1407/1407 ————— 10s 5ms/step - acc: 0.7288 - loss: 0.0385 - val_acc: 0.7636 - val_loss: 0.0343
Epoch 47/50
1407/1407 ————— 10s 5ms/step - acc: 0.7386 - loss: 0.0374 - val_acc: 0.7612 - val_loss: 0.0338
Epoch 48/50
1407/1407 ————— 10s 5ms/step - acc: 0.7401 - loss: 0.0374 - val_acc: 0.7620 - val_loss: 0.0343
Epoch 49/50
1407/1407 ————— 10s 5ms/step - acc: 0.7389 - loss: 0.0372 - val_acc: 0.7624 - val_loss: 0.0346
Epoch 50/50
1407/1407 ————— 7s 5ms/step - acc: 0.7403 - loss: 0.0372 - val_acc: 0.7618 - val_loss: 0.0343
```

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示：



最终测试模型的准确率为 75.14%。

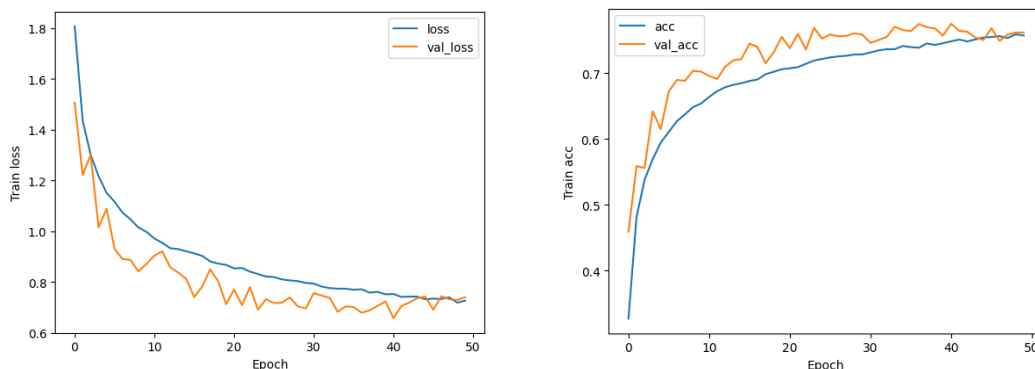


2) 使用 KL 散度损失函数:

我们还尝试了将交叉熵损失函数更改为平方损失函数，运行结果如下:

```
Epoch 40/50 1407/1407 ————— 10s 5ms/step - acc: 0.7443 - loss: 0.7582 - val_acc: 0.7570 - val_loss: 0.7230
Epoch 41/50 1407/1407 ————— 7s 5ms/step - acc: 0.7492 - loss: 0.7498 - val_acc: 0.7750 - val_loss: 0.6572
Epoch 42/50 1407/1407 ————— 10s 5ms/step - acc: 0.7483 - loss: 0.7502 - val_acc: 0.7644 - val_loss: 0.7049
Epoch 43/50 1407/1407 ————— 7s 5ms/step - acc: 0.7496 - loss: 0.7287 - val_acc: 0.7628 - val_loss: 0.7185
Epoch 44/50 1407/1407 ————— 7s 5ms/step - acc: 0.7520 - loss: 0.7442 - val_acc: 0.7540 - val_loss: 0.7363
Epoch 45/50 1407/1407 ————— 10s 5ms/step - acc: 0.7522 - loss: 0.7331 - val_acc: 0.7502 - val_loss: 0.7421
Epoch 46/50 1407/1407 ————— 11s 5ms/step - acc: 0.7579 - loss: 0.7217 - val_acc: 0.7682 - val_loss: 0.6905
Epoch 47/50 1407/1407 ————— 10s 5ms/step - acc: 0.7543 - loss: 0.7274 - val_acc: 0.7490 - val_loss: 0.7435
Epoch 48/50 1407/1407 ————— 10s 5ms/step - acc: 0.7553 - loss: 0.7337 - val_acc: 0.7584 - val_loss: 0.7317
Epoch 49/50 1407/1407 ————— 7s 5ms/step - acc: 0.7605 - loss: 0.7126 - val_acc: 0.7616 - val_loss: 0.7291
Epoch 50/50 1407/1407 ————— 10s 5ms/step - acc: 0.7545 - loss: 0.7327 - val_acc: 0.7618 - val_loss: 0.7391
```

得到的训练集损失函数以及验证集损失函数随训练次数的变化以及训练集准确率以及验证集准确率随训练次数的变化如下图所示:



最终测试模型的准确率为 74.93%。

313/313 — 2s 4ms/step
0.7493

可见在本实验中，交叉熵损失函数与平方损失函数、KL 散度损失函数有相近的效果。

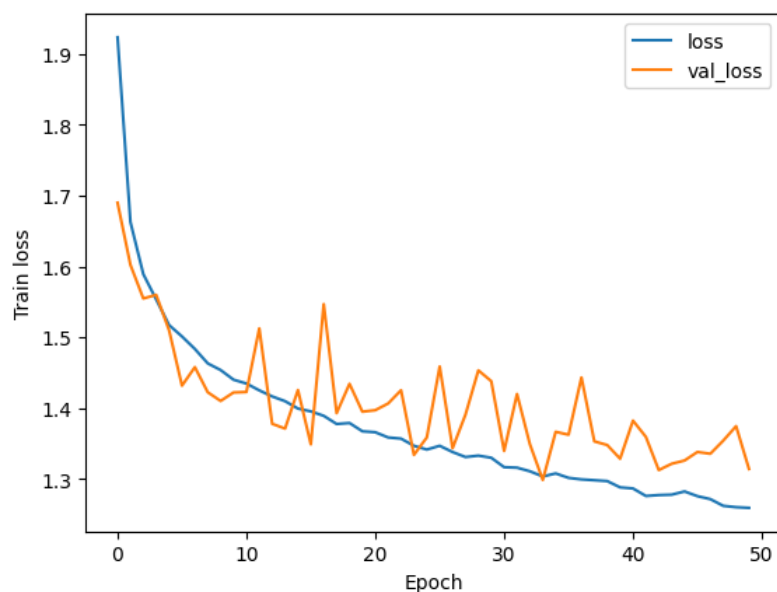
4. 将卷积层全部改为全连接层

在其他参数都保持不变的情况下，我们将原有网络的卷积层连接改为全连接层连接，（即使用全连接神经网络）观察最后预测性能的变化。

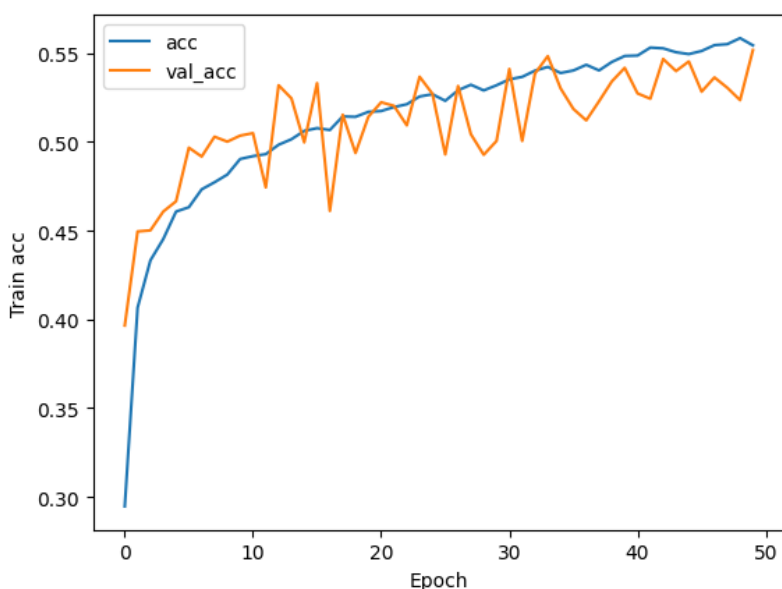
更改的代码如下：

```
##### 构建网络 #####
from keras import Sequential
from keras.layers import Convolution2D, MaxPooling2D, Dense, Flatten, Dropout
cnn = Sequential()
#unit1
cnn.add(Dense(512, activation='relu',input_shape=(32,32,3)))
cnn.add(Dropout(0.5))
cnn.add(Dense(512, activation='relu'))
cnn.add(MaxPooling2D(pool_size=[2, 2], padding='same'))
cnn.add(Dropout(0.5))
#unit2
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(512, activation='relu'))
cnn.add(MaxPooling2D(pool_size=[2, 2], padding='same'))
cnn.add(Dropout(0.5))
# 展平层
cnn.add(Flatten())
# 全连接层
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(10, activation='softmax'))
```

得到的训练集损失函数以及验证集损失函数随训练次数的变化如下图：



得到的训练集准确率以及验证集准确率随训练次数的变化如下图：



最终将 10000 个测试数据输入训练好的模型中，预测结果并计算准确率，得到模型的预测准确率为 53.08%。

313/313 ————— 1s 2ms/step
53.08

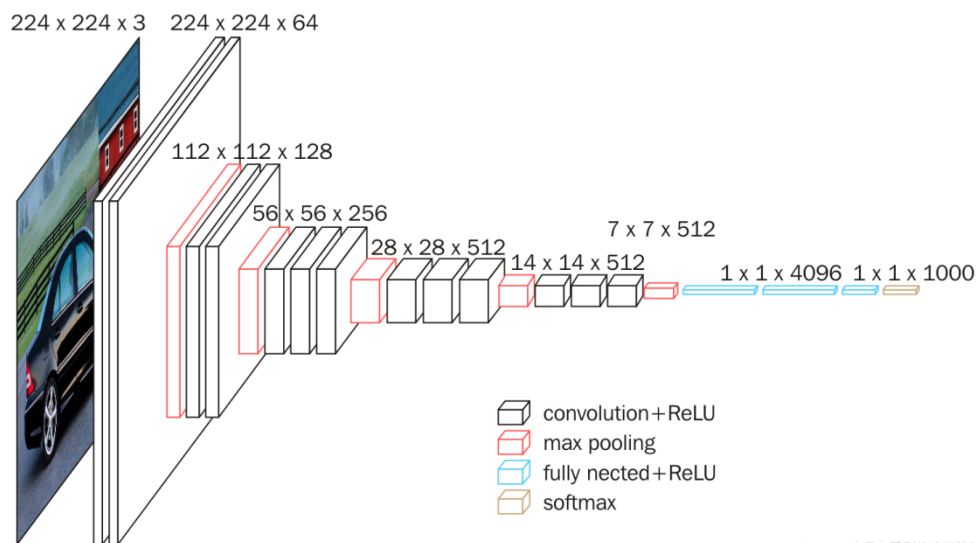
可以看到识别准确率大幅下降，这也许是因为全连接网络很难提取照片中的局部不变性特征，比起卷积神经网络的识别准确率会降低；而且我们看到全连接网络的训练过程明显更加耗时，这应该是因为全连接网络训练的参数过多，会导致训练成本升高，同时还会增加过拟合风险。

5. 使用 VGG 网络

VGG-16 网络包括 13 个卷积层和 3 个全连接层，每个卷积层的结果输入到激活函数之前还需要进行 batch normalization 操作，以免激活函数的输入值过大使得函数值太接近 1 或-1。VGG-16 网络结构较 LeNet-5 等网络线的十分复杂，但同时也有不错的效果。VGG-16 有强大的拟合能力在当时取得了非常的效果，但同时 VGG 网络也有部分不足，包括：

- 1) 巨大参数量导致训练时间过长，调参难度较大；
- 2) 模型所需内存容量大，VGG 的权值文件很大，用到实际应用会比较困难。

VGG 网络的结构如下图所示：



CSDN @一个只会看剧的读研菜鸟

在其他参数都保持不变的情况下，构建 VGG 网络的代码如下：

```
##### 构建网络 #####  
from keras import Sequential  
from keras.layers import Convolution2D, MaxPooling2D, BatchNormalization, Dense,  
Flatten, Dropout  
cnn = Sequential()  
#unit1  
# 二维卷积层 和 二维最大池化层 交替堆叠  
cnn.add(Convolution2D(filters=64, kernel_size=(3, 3), input_shape=(32, 32, 3),  
activation='relu',padding='same'))  
cnn.add(BatchNormalization())  
cnn.add(Convolution2D(filters=64, kernel_size=(3, 3),  
activation='relu',padding='same'))  
cnn.add(BatchNormalization())  
cnn.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='same'))  
cnn.add(Dropout(0.2))
```

```
cnn.add(Convolution2D(filters=128, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=128, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='same'))
cnn.add(Dropout(0.2))

cnn.add(Convolution2D(filters=256, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=256, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=256, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='same'))
cnn.add(Dropout(0.2))

cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='same'))
cnn.add(Dropout(0.2))

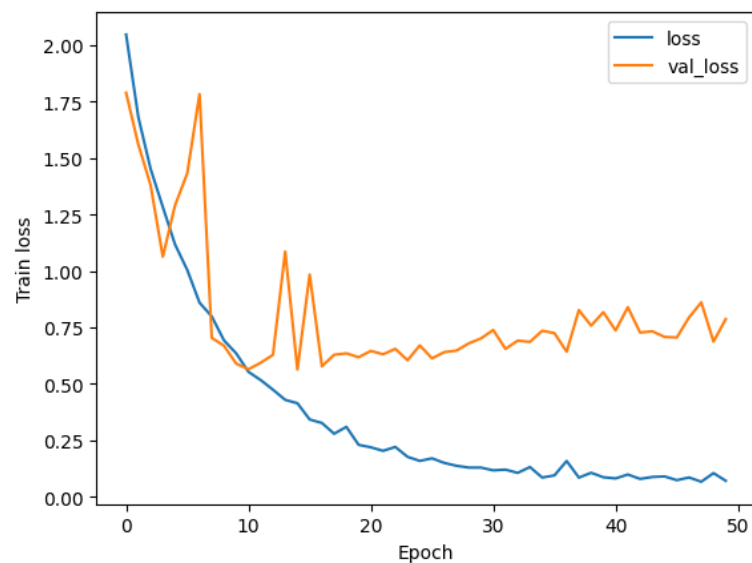
cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
cnn.add(Convolution2D(filters=512, kernel_size=(3, 3), activation='relu',
padding='same'))
cnn.add(BatchNormalization())
```

```

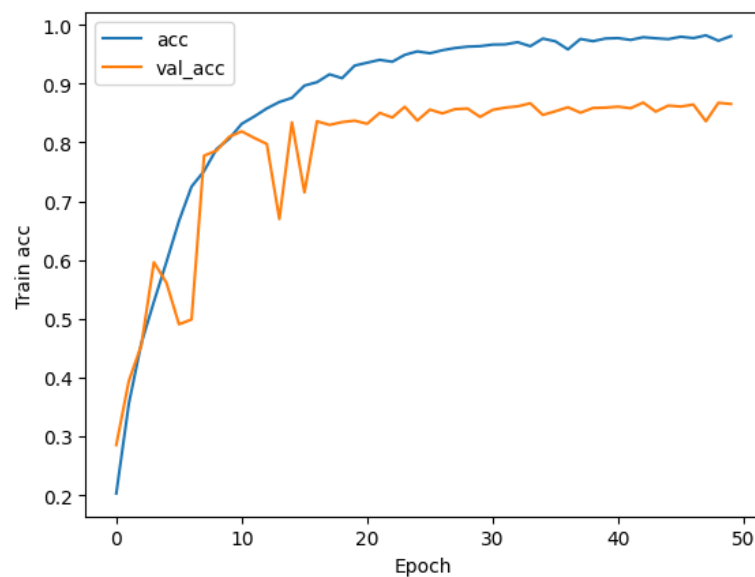
cnn.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='same'))
cnn.add(Dropout(0.2))
# 展平层
cnn.add(Flatten())
# 全连接层
cnn.add(Dense(512, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(128, activation='relu'))
cnn.add(Dropout(0.5))
cnn.add(Dense(10, activation='softmax'))

```

得到的训练集损失函数以及验证集损失函数随训练次数的变化如下图：



得到的训练集准确率以及验证集准确率随训练次数的变化如下图：



最终将 10000 个测试数据输入训练好的模型中，预测结果并计算准确率，得到模型的预测准确率为 85.49%。这个结果相较于原有网络有较大提升，验证了

我们的猜想，即：VGG-16 网络虽然复杂，但训练精度也相对更好一些。

313/313 ————— 6s 14ms/step
0.8549

四、实验心得

实验过程中遇到的什么问题。尝试使用什么方法去解决。通过实验获得了什么感悟与理解。

1. 本次实验最复杂的地方是在环境配置上。尤其是 GPU 版本的 tensorflow 安装以及 CUDA 的安装。由于我本人之前使用过 python，故而其 python 版本较高，与 tensorflow 不兼容，故而在这里耗费了无数时间，最后是寻求同学帮助才解决。但是最后运行的时候速度也较慢，也更容易出现梯度消失等问题，但幸运的是，在同学推荐下，我找到了 Google Colab 平台，在线运行时速度非常快，而且运行结果也更稳定。

2. 起初使用 VGG-16 网络时，发现每次迭代后损失函数值并未下降，且预测准确度始终维持在 10% 上下，经检查发现每个卷积层后未加 `batchnormalized` 层一开始将所有的卷积层改为全连接层时，发现程序运行时报错，经检查发现是因为最大汇聚层之前、相邻全连接层之后添加了 `dropout` 层，将其删去程序就可以正常运行。

通过本次实验我更加深刻理解了卷积神经网络的运作过程，亲自体验了搭建神经网络的过程，解决了之前困扰了我很长时间的卷积神经网络的输入输出维度问题。