華中科技大學

2023

计算机视觉实验报告

上机实验二：基于卷积神经网络的 CIFAR-10 数据集分类

专　　业：　计算机科学与技术

班　　级：　计科 2008 班

学　　号：　U202015550

姓　　名：　刘方兴

完成日期：　2023 年 4 月 3 日

目 录

1. 深度学习框架

1.1 编程环境

开发环境:

jupyter notebook (本地版)

https://featurize.cn/ (云 GPU)

辅助工具：wandb—对实验过程进行实时监控、实现数据可视化

1.2 框架选择

基本框架：tensorflow

1.3 代码结构设计

1）工具包导入及 GPU 环境设置部分

```
import tensorflow as tf
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import itertools

# # 获取可用的 GPU 设备列表
# gpus = tf.config.experimental.list_physical_devices('GPU')


# # 如果没有可用的 GPU 设备，则输出错误信息
# if not gpus:
#     print("No available GPU devices!")
# else:
#     # 输出可用的 GPU 设备信息
#     for gpu in gpus:
#         print(gpu)
# # 指定使用的 GPU
# gpus = tf.config.experimental.list_physical_devices('GPU')
# tf.config.experimental.set_visible_devices(gpus[0], 'GPU')
```

2）实验数据准备部分

<1>数据增强

```
# 定义训练集和测试集的数据增强
train_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
    tf.keras.layers.experimental.preprocessing.RandomRotation(0.1),
    tf.keras.layers.experimental.preprocessing.RandomTranslation(0.1, 0.1),
    tf.keras.layers.experimental.preprocessing.RandomZoom(0.1),
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])

test_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])
```

<2>超参数定义

```
# 定义超参数
lr = 0.001
batch_size = 256
num_epochs = 100
```

<3>学习率动态调整

```
def lr_schedule(epoch, lr):
    if epoch % 20 == 0 and epoch > 0:
        lr = lr / 2
    return lr
#定义动态学习率调整机制
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lr_schedule)
```

<4>实验数据导入及预处理

```
# 加载 CIFAR-10 数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# 将标签转换为 one-hot 编码
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# 对训练集和测试集的特征进行数据增强和标准化处理
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=10000)
train_dataset = train_dataset.map(lambda x, y: (train_transform(x), y))
train_dataset = train_dataset.batch(batch_size=batch_size)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.map(lambda x, y: (test_transform(x), y))
test_dataset = test_dataset.batch(batch_size=batch_size)
test_dataset = test_dataset.prefetch(tf.data.experimental.AUTOTUNE)

# 定义优化器和损失函数
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
loss_fn = tf.keras.losses.CategoricalCrossentropy()

# 定义回调函数，保存最佳模型
checkpoint_path = 'my_model_final_best.h5'
checkpoint = ModelCheckpoint(checkpoint_path, monitor='val_accuracy', save_best_only=True, mode='max', verbose=1)

# 编译模型
model.compile(optimizer=optimizer, loss=loss_fn, metrics=['accuracy'])

# 训练模型
history = model.fit(train_dataset, epochs=num_epochs, validation_data=test_dataset,callbacks=[checkpoint,lr_scheduler])
```

3）实验模型设计部分

```
# 定义卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), # 输入通道数为3，输出通道数为64，卷积核大小为3x3，填充为1
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为64，输出通道数为64，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.05),

    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'),# 输入通道数为64，输出通道数为128，卷积核大小为3x3，填充为1
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为128，输出通道数为128，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    # 设置 kernel_regularizer 参数来实现 L2 正则化，其中 0.001 是 L2 正则化系数。
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),# 输入通道数为128，输出通道数为256，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为256，输出通道数为256，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.15),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Flatten(),# 将张量展开为一维张量
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, activation='relu'),# 全连接层1，输入维度为256*4*4，输出维度为1024
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(512, activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 全连接层2，输入维度为1024，输出维度为512
    tf.keras.layers.Dense(10, activation='softmax')# 输出层，输入维度为512，输出维度为10
])
```

4）输出结果可视化部分

<1>可视化显示训练集和测试集的 loss、accuracy

```
# 可视化显示训练集和测试集的loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()

# 可视化显示训练集和测试集的accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='lower right')
plt.show()
```

〈2〉可视化每个类别的准确率

```
# 预测测试集的标签
y_pred = model.predict(test_dataset)

# 计算每个类别的准确率
correct_counts = [0] * 10
total_counts = [0] * 10
for i in range(len(y_pred)):
    pred_label = tf.argmax(y_pred[i]).numpy()
    true_label = tf.argmax(y_test[i]).numpy()
    total_counts[true_label] += 1
    if pred_label == true_label:
        correct_counts[true_label] += 1

# 打印每个类别的准确率
for i in range(10):
    acc = correct_counts[i] / total_counts[i]
    print('Class {}: {} accuracy: {:.2%}'.format(i, class_names[i] ,acc))

# 可视化每个类别的准确率
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
accuracies = [correct_counts[i] / total_counts[i] for i in range(10)]
x_pos = np.arange(len(class_names))
plt.bar(x_pos, accuracies)
plt.xticks(x_pos, class_names, rotation='vertical')
plt.ylabel('Accuracy')
plt.title('Accuracy by Class')
plt.show()
```

〈3〉绘制预测结果混淆矩阵

```
# 预测测试集的结果
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# 计算混淆矩阵
confusion_mtx = confusion_matrix(y_true, y_pred_classes)

# 绘制混淆矩阵
def plot_confusion_matrix(cm, classes, normalize=False, cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

plot_confusion_matrix(confusion_mtx, class_names)
```

5）模型保存部分

```
# 定义回调函数，保存最佳模型
checkpoint_path = 'my_model_final_best.h5'
checkpoint = ModelCheckpoint(checkpoint_path, monitor='val_accuracy', save_best_only=True, mode='max', verbose=1)
```

```
model.save('my_model_final.h5') # 保存为.h5格式
```

```
# 读取模型
model = load_model('my_model_final_best.h5')

# 进行测试
test_loss, test_acc = model.evaluate(test_dataset)
print('Test accuracy:', test_acc)
```

2. 实验过程设计

2.1 基本实验模型

```
# 定义卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), # 输入通道数为3，输出通道数为64，卷积核大小
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu'), # 输入通道数为64，输出通道数为64，卷积核大小为3x3，填充为1
    tf.keras.layers.MaxPooling2D(2, 2), # 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(), # 将张量展开为一维张量
    tf.keras.layers.Dense(1024, activation='relu'), # 全连接层1，输入维度为256*4*4，输出维度为1024
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax') # 输出层，输入维度为512，输出维度为10
])
```

基于 tf.keras.Sequential 搭建一个最基本的卷积神经网络模型，模型结构如下（示意图使用 tf.keras.utils.plot_model 打印）：

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 64)        1792

conv2d_1 (Conv2D)            (None, 32, 32, 64)        36928

max_pooling2d (MaxPooling2D  (None, 16, 16, 64)        0
)

conv2d_2 (Conv2D)            (None, 16, 16, 128)       73856

conv2d_3 (Conv2D)            (None, 16, 16, 128)       147584

max_pooling2d_1 (MaxPooling  (None, 8, 8, 128)         0
2D)

conv2d_4 (Conv2D)            (None, 8, 8, 256)         295168

conv2d_5 (Conv2D)            (None, 8, 8, 256)         590080

max_pooling2d_2 (MaxPooling  (None, 4, 4, 256)         0
2D)

flatten (Flatten)            (None, 4096)              0

dense (Dense)                (None, 1024)              4195328

dense_1 (Dense)              (None, 512)               524800

dense_2 (Dense)              (None, 10)                5130

=================================================================
Total params: 5,870,666
Trainable params: 5,870,666
Non-trainable params: 0
_________________________________________________________________
```

图 2.1 基本模型结构

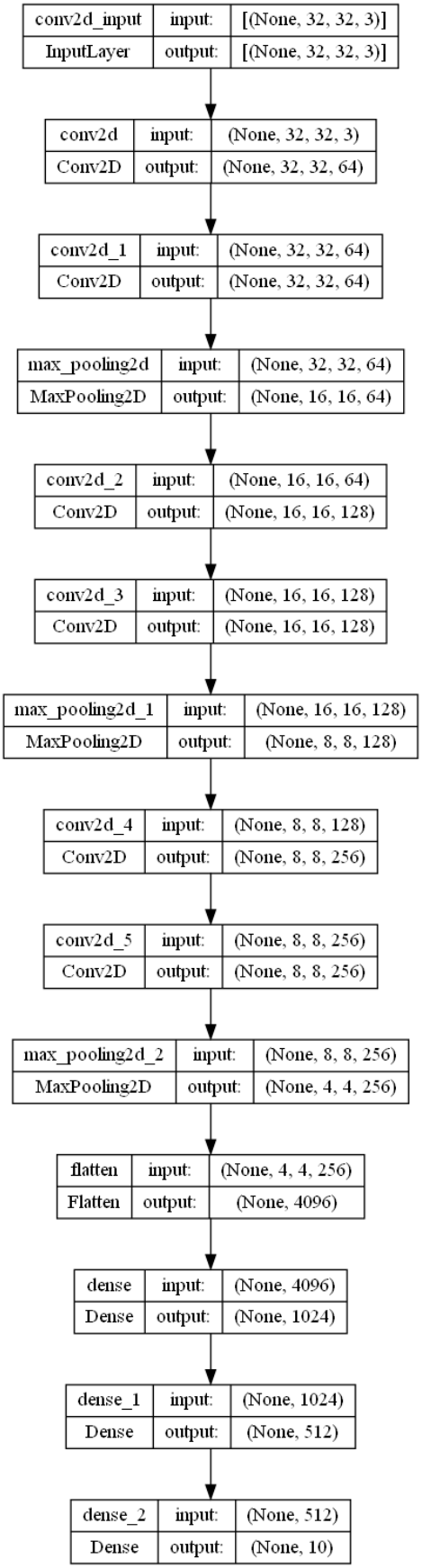
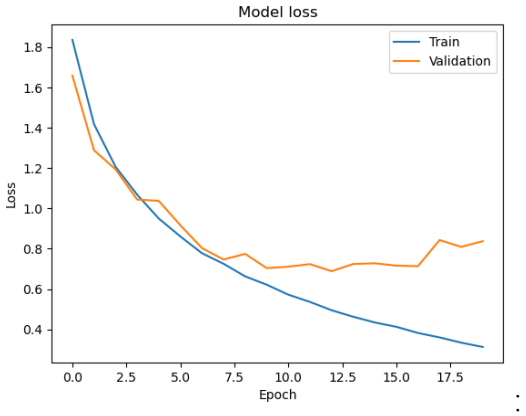
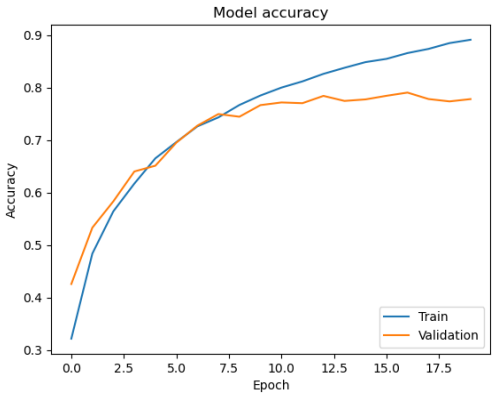

conv2d_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d | input: | (None, 32, 32, 3)
Conv2D | output: | (None, 32, 32, 64)
conv2d_1 | input: | (None, 32, 32, 64)
Conv2D | output: | (None, 32, 32, 64)
max_pooling2d | input: | (None, 32, 32, 64)
MaxPooling2D | output: | (None, 16, 16, 64)
conv2d_2 | input: | (None, 16, 16, 64)
Conv2D | output: | (None, 16, 16, 128)
conv2d_3 | input: | (None, 16, 16, 128)
Conv2D | output: | (None, 16, 16, 128)
max_pooling2d_1 | input: | (None, 16, 16, 128)
MaxPooling2D | output: | (None, 8, 8, 128)
conv2d_4 | input: | (None, 8, 8, 128)
Conv2D | output: | (None, 8, 8, 256)
conv2d_5 | input: | (None, 8, 8, 256)
Conv2D | output: | (None, 8, 8, 256)
max_pooling2d_2 | input: | (None, 8, 8, 256)
MaxPooling2D | output: | (None, 4, 4, 256)
flatten | input: | (None, 4, 4, 256)
Flatten | output: | (None, 4096)
dense | input: | (None, 4096)
Dense | output: | (None, 1024)
dense_1 | input: | (None, 1024)
Dense | output: | (None, 512)
dense_2 | input: | (None, 512)
Dense | output: | (None, 10)

图 2.2 基本模型示意图

实验测试结果






```
196/196 [==============================] - 171s 874ms/step - loss: 1.2060 - accuracy: 0.5644 - val_loss: 1.1931 - val_accuracy: 0.5834
Epoch 4/20
196/196 [==============================] - 172s 879ms/step - loss: 1.0676 - accuracy: 0.6172 - val_loss: 1.0435 - val_accuracy: 0.6403
Epoch 5/20
196/196 [==============================] - 170s 865ms/step - loss: 0.9495 - accuracy: 0.6654 - val_loss: 1.0370 - val_accuracy: 0.6513
Epoch 6/20
196/196 [==============================] - 170s 869ms/step - loss: 0.8606 - accuracy: 0.6963 - val_loss: 0.9159 - val_accuracy: 0.6953
Epoch 7/20
196/196 [==============================] - 177s 905ms/step - loss: 0.7775 - accuracy: 0.7262 - val_loss: 0.8030 - val_accuracy: 0.7276
Epoch 8/20
196/196 [==============================] - 176s 898ms/step - loss: 0.7252 - accuracy: 0.7434 - val_loss: 0.7462 - val_accuracy: 0.7497
Epoch 9/20
196/196 [==============================] - 181s 924ms/step - loss: 0.6627 - accuracy: 0.7670 - val_loss: 0.7742 - val_accuracy: 0.7446
Epoch 10/20
196/196 [==============================] - 183s 933ms/step - loss: 0.6214 - accuracy: 0.7850 - val_loss: 0.7034 - val_accuracy: 0.7667
Epoch 11/20
196/196 [==============================] - 194s 987ms/step - loss: 0.5719 - accuracy: 0.8001 - val_loss: 0.7109 - val_accuracy: 0.7718
Epoch 12/20
196/196 [==============================] - 238s 1s/step - loss: 0.5361 - accuracy: 0.8119 - val_loss: 0.7233 - val_accuracy: 0.7704
Epoch 13/20
196/196 [==============================] - 183s 935ms/step - loss: 0.4949 - accuracy: 0.8263 - val_loss: 0.6884 - val_accuracy: 0.7843
Epoch 14/20
196/196 [==============================] - 175s 893ms/step - loss: 0.4627 - accuracy: 0.8378 - val_loss: 0.7240 - val_accuracy: 0.7747
Epoch 15/20
196/196 [==============================] - 175s 893ms/step - loss: 0.4343 - accuracy: 0.8486 - val_loss: 0.7277 - val_accuracy: 0.7776
Epoch 16/20
196/196 [==============================] - 175s 891ms/step - loss: 0.4127 - accuracy: 0.8548 - val_loss: 0.7158 - val_accuracy: 0.7845
Epoch 17/20
196/196 [==============================] - 175s 891ms/step - loss: 0.3822 - accuracy: 0.8659 - val_loss: 0.7132 - val_accuracy: 0.7906
Epoch 18/20
196/196 [==============================] - 176s 898ms/step - loss: 0.3601 - accuracy: 0.8738 - val_loss: 0.8429 - val_accuracy: 0.7783
Epoch 19/20
196/196 [==============================] - 176s 899ms/step - loss: 0.3341 - accuracy: 0.8849 - val_loss: 0.8088 - val_accuracy: 0.7738
Epoch 20/20
196/196 [==============================] - 175s 890ms/step - loss: 0.3129 - accuracy: 0.8913 - val_loss: 0.8371 - val_accuracy: 0.7782
```

图 2.3 基本模型实验结果图

在运行 20 个 epoch 后，可以看到模型在训练集上的准确率达到了 0.8713，但自从第 10 个 epoch 开始，模型在测试集上的准确率并没有明显的提升，始终保持在 0.76 左右。

这反映出两个问题：

第一：模型的整体识别效果并不是很好，准确率还不够高。

第二：模型仅仅在训练集上表现好，这说明模型存在过拟合的现象。

2.2 数据增强及正则化参数添加

1.在保持网络架构基本不变的前提下，增加数据增强部分：

```
# 定义训练集和测试集的数据增强
train_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
    tf.keras.layers.experimental.preprocessing.RandomRotation(0.1),
    tf.keras.layers.experimental.preprocessing.RandomZoom(0.1),
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])

test_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])
```

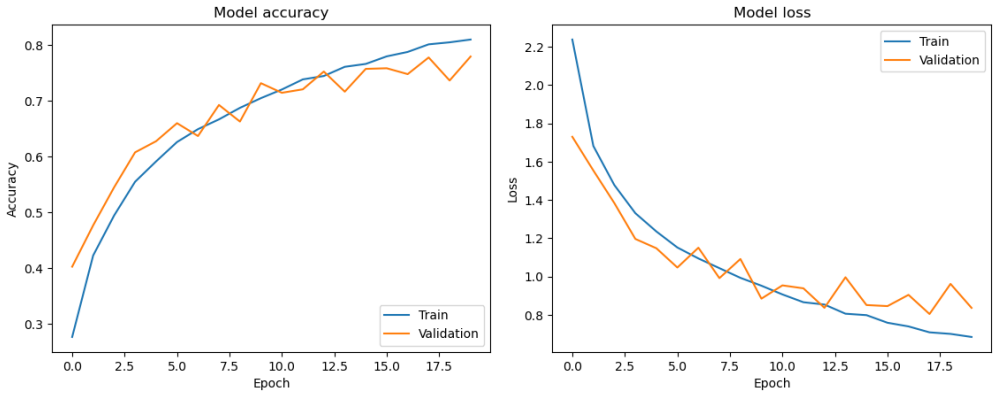
2.在保持网络架构基本不变的前提下，增加 dropout 和 L2 正则化系数：

```
# 定义卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), # 输入通道数为3，输出通道数为64，卷
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为64，
    tf.keras.layers.MaxPooling2D(2, 2), # 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    # 设置 kernel_regularizer 参数来实现 L2 正则化，其中 0.001 是 L2 正则化系数，
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(), # 将张量展开为一维张量
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1024, activation='relu'), # 全连接层1，输入维度为256*4*4，输出维度为1024
    tf.keras.layers.Dense(512, activation='relu',kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.Dense(10, activation='softmax') # 输出层，输入维度为512，输出维度为10
])
```

实验结果：

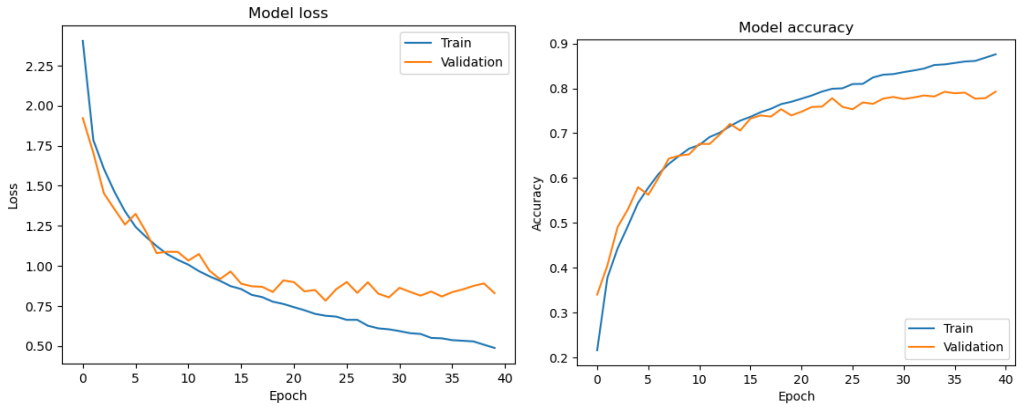
20 个 epoch：

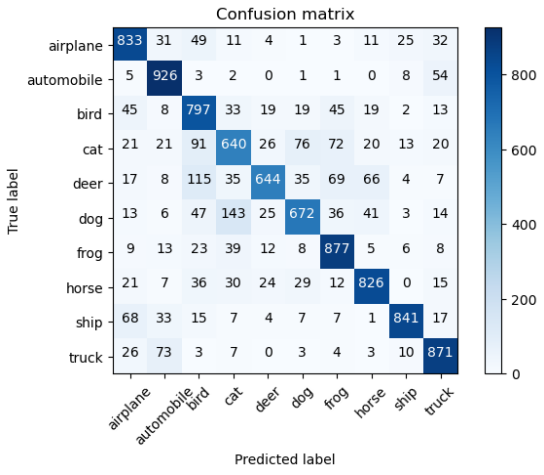
```
Epoch 28/40
98/98 [==============================] - 194s 2s/step - loss: 0.6265 - accuracy: 0.8244 - val_loss: 0.8977 - val_accuracy: 0.7655
Epoch 29/40
98/98 [==============================] - 197s 2s/step - loss: 0.6099 - accuracy: 0.8305 - val_loss: 0.8259 - val_accuracy: 0.7772
Epoch 30/40
98/98 [==============================] - 191s 2s/step - loss: 0.6041 - accuracy: 0.8319 - val_loss: 0.8036 - val_accuracy: 0.7809
Epoch 31/40
98/98 [==============================] - 193s 2s/step - loss: 0.5930 - accuracy: 0.8363 - val_loss: 0.8630 - val_accuracy: 0.7763
Epoch 32/40
98/98 [==============================] - 193s 2s/step - loss: 0.5800 - accuracy: 0.8400 - val_loss: 0.8376 - val_accuracy: 0.7798
Epoch 33/40
98/98 [==============================] - 196s 2s/step - loss: 0.5751 - accuracy: 0.8442 - val_loss: 0.8143 - val_accuracy: 0.7841
Epoch 34/40
98/98 [==============================] - 174s 2s/step - loss: 0.5505 - accuracy: 0.8521 - val_loss: 0.8399 - val_accuracy: 0.7819
Epoch 35/40
98/98 [==============================] - 164s 2s/step - loss: 0.5478 - accuracy: 0.8534 - val_loss: 0.8088 - val_accuracy: 0.7924
Epoch 36/40
98/98 [==============================] - 163s 2s/step - loss: 0.5364 - accuracy: 0.8567 - val_loss: 0.8361 - val_accuracy: 0.7892
Epoch 37/40
98/98 [==============================] - 167s 2s/step - loss: 0.5327 - accuracy: 0.8600 - val_loss: 0.8532 - val_accuracy: 0.7905
Epoch 38/40
98/98 [==============================] - 169s 2s/step - loss: 0.5286 - accuracy: 0.8612 - val_loss: 0.8749 - val_accuracy: 0.7770
Epoch 39/40
98/98 [==============================] - 172s 2s/step - loss: 0.5079 - accuracy: 0.8685 - val_loss: 0.8904 - val_accuracy: 0.7783
Epoch 40/40
98/98 [==============================] - 175s 2s/step - loss: 0.4878 - accuracy: 0.8759 - val_loss: 0.8299 - val_accuracy: 0.7927
```

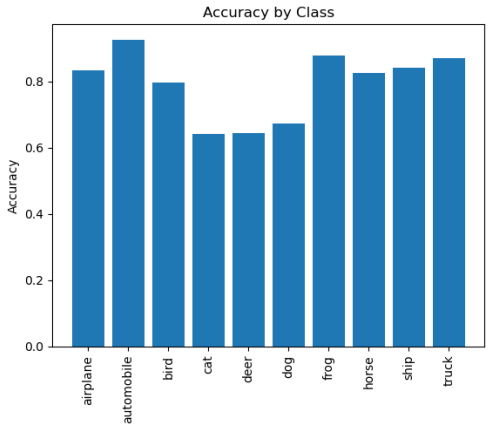


40 个 epoch：

```
Epoch 28/40
98/98 [==============================] - 194s 2s/step - loss: 0.6265 - accuracy: 0.8244 - val_loss: 0.8977 - val_accuracy: 0.7655
Epoch 29/40
98/98 [==============================] - 197s 2s/step - loss: 0.6099 - accuracy: 0.8305 - val_loss: 0.8259 - val_accuracy: 0.7772
Epoch 30/40
98/98 [==============================] - 191s 2s/step - loss: 0.6041 - accuracy: 0.8319 - val_loss: 0.8036 - val_accuracy: 0.7809
Epoch 31/40
98/98 [==============================] - 193s 2s/step - loss: 0.5930 - accuracy: 0.8363 - val_loss: 0.8630 - val_accuracy: 0.7763
Epoch 32/40
98/98 [==============================] - 193s 2s/step - loss: 0.5800 - accuracy: 0.8400 - val_loss: 0.8376 - val_accuracy: 0.7798
Epoch 33/40
98/98 [==============================] - 196s 2s/step - loss: 0.5751 - accuracy: 0.8442 - val_loss: 0.8143 - val_accuracy: 0.7841
Epoch 34/40
98/98 [==============================] - 174s 2s/step - loss: 0.5505 - accuracy: 0.8521 - val_loss: 0.8399 - val_accuracy: 0.7819
Epoch 35/40
98/98 [==============================] - 164s 2s/step - loss: 0.5478 - accuracy: 0.8534 - val_loss: 0.8088 - val_accuracy: 0.7924
Epoch 36/40
98/98 [==============================] - 163s 2s/step - loss: 0.5364 - accuracy: 0.8567 - val_loss: 0.8361 - val_accuracy: 0.7892
Epoch 37/40
98/98 [==============================] - 167s 2s/step - loss: 0.5327 - accuracy: 0.8600 - val_loss: 0.8532 - val_accuracy: 0.7905
Epoch 38/40
98/98 [==============================] - 169s 2s/step - loss: 0.5286 - accuracy: 0.8612 - val_loss: 0.8749 - val_accuracy: 0.7770
Epoch 39/40
98/98 [==============================] - 172s 2s/step - loss: 0.5079 - accuracy: 0.8685 - val_loss: 0.8904 - val_accuracy: 0.7783
Epoch 40/40
98/98 [==============================] - 175s 2s/step - loss: 0.4878 - accuracy: 0.8759 - val_loss: 0.8299 - val_accuracy: 0.7927
```







在运行 40 个 epoch 后，模型基本收敛，最终测试集上的 accuracy 接近 0.87，训练集上的 accuracy 接近 0.8

从防止过拟合的方面看: dropout 和 L2 正则化系数的添加一定程度上减少了模型的过拟合程度，增加了模型的泛化能力。但测试集上的 accuracy 与训练集上的 accuracy 还是存在不小的差距。

从模型预测准确率上来看：数据增强并没有在防止过拟合的基础上为模型的预测准确率带来非常大的提升。

综上，我考虑从模型本身入手，在尝试增加神经网络的深度的同时也将模型的正则化参数调的大一些。

2.3 增加网络深度及正则化参数大小

构建新的卷积神经网络结构：

（1）

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 64)	1792
conv2d_1 (Conv2D)	(None, 32, 32, 64)	36928
batch_normalization (BatchNormalization)	(None, 32, 32, 64)	256
max_pooling2d (MaxPooling2D)	(None, 16, 16, 64)	0
dropout (Dropout)	(None, 16, 16, 64)	0
conv2d_2 (Conv2D)	(None, 16, 16, 128)	73856
conv2d_3 (Conv2D)	(None, 16, 16, 128)	147584

（2）

batch_normalization_1 (BatchNormalization)	(None, 16, 16, 128)	512
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 128)	0
dropout_1 (Dropout)	(None, 8, 8, 128)	0
conv2d_4 (Conv2D)	(None, 8, 8, 256)	295168
batch_normalization_2 (BatchNormalization)	(None, 8, 8, 256)	1024
conv2d_5 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_3 (BatchNormalization)	(None, 8, 8, 256)	1024

（3）

conv2d_6 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 256)	1024
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 256)	0
dropout_2 (Dropout)	(None, 4, 4, 256)	0
conv2d_7 (Conv2D)	(None, 4, 4, 512)	1180160
batch_normalization_5 (BatchNormalization)	(None, 4, 4, 512)	2048
conv2d_8 (Conv2D)	(None, 4, 4, 512)	2359808
batch_normalization_6 (BatchNormalization)	(None, 4, 4, 512)	2048

（4）

conv2d_9 (Conv2D)	(None, 4, 4, 512)	2359808
batch_normalization_7 (BatchNormalization)	(None, 4, 4, 512)	2048
max_pooling2d_3 (MaxPooling2D)	(None, 2, 2, 512)	0
dropout_3 (Dropout)	(None, 2, 2, 512)	0
conv2d_10 (Conv2D)	(None, 2, 2, 512)	2359808
batch_normalization_8 (BatchNormalization)	(None, 2, 2, 512)	2048
conv2d_11 (Conv2D)	(None, 2, 2, 512)	2359808
batch_normalization_9 (BatchNormalization)	(None, 2, 2, 512)	2048

conv2d_12 (Conv2D)	(None, 2, 2, 512)	2359808
batch_normalization_10 (BatchNormalization)	(None, 2, 2, 512)	2048
max_pooling2d_4 (MaxPooling2D)	(None, 1, 1, 512)	0
dropout_4 (Dropout)	(None, 1, 1, 512)	0
flatten (Flatten)	(None, 512)	0
dropout_5 (Dropout)	(None, 512)	0
dense (Dense)	(None, 4096)	2101248
dropout_6 (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 1024)	4195328

（5）

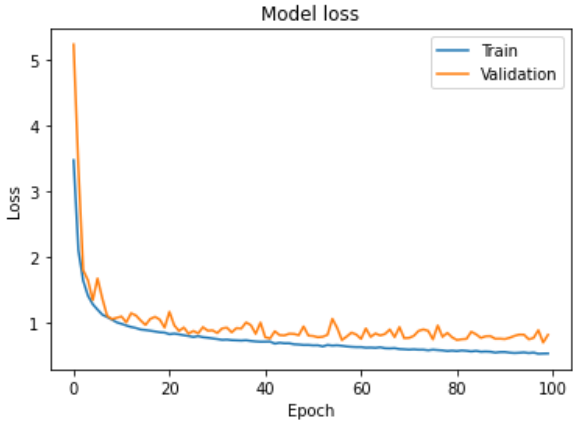
dense_2 (Dense)	(None, 512)	524800
dense_3 (Dense)	(None, 10)	5130

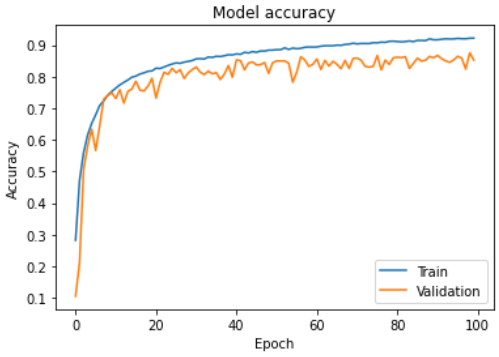
```
=================================================================
Total params: 21,557,322
Trainable params: 21,549,258
Non-trainable params: 8,064
```

（6）

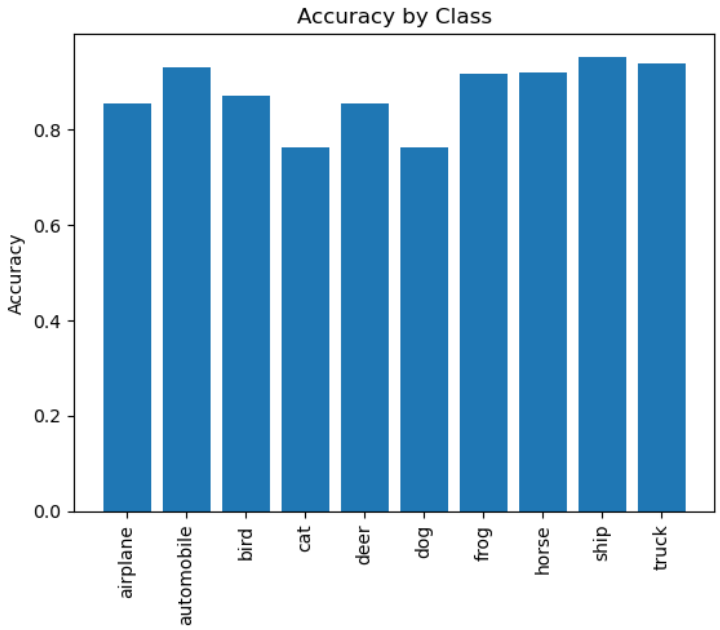
```
# 定义卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), # 输入通道数为3，输出通道数为64，卷积核大小为3x
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为64，输出通道数为
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2), # 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.05),

    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'), # 输入通道数为64，输出通道数为128，卷积核大小为3x3，填充为1
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为128，输出通道
    tf.keras.layers.BatchNormalization(),
    # 设置 kernel_regularizer 参数来实现 L2 正则化，其中 0.001 是 L2 正则化系数。
    tf.keras.layers.MaxPooling2D(2, 2), # 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'), # 输入通道数为128，输出通道数为256，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为256，输出通道
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2), # 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.15),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Flatten(), # 将张量展开为一维张量
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, activation='relu'), # 全连接层1，输入维度为256*4*4，输出维度为1024
    tf.keras.layers.Dense(512, activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 全连接层2，输入维度为1024，输出维度为512
    tf.keras.layers.Dense(10, activation='softmax') # 输出层，输入维度为512，输出维度为10
])
```

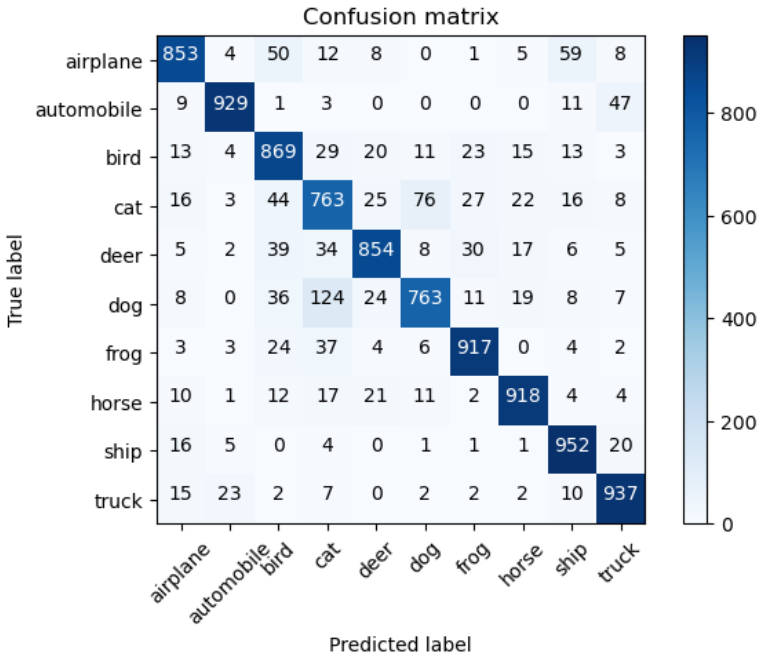
实验结果:






```
Epoch 00096: val_accuracy did not improve from 0.86730
196/196 [==============================] - 41s 208ms/step - loss: 0.5297 - accuracy: 0.9212 - val_loss: 0.7389 - val_accuracy: 0.8646
Epoch 97/100
196/196 [==============================] - ETA: 0s - loss: 0.5369 - accuracy: 0.9203
Epoch 00097: val_accuracy did not improve from 0.86730
196/196 [==============================] - 41s 210ms/step - loss: 0.5369 - accuracy: 0.9203 - val_loss: 0.7607 - val_accuracy: 0.8589
Epoch 98/100
196/196 [==============================] - ETA: 0s - loss: 0.5191 - accuracy: 0.9201
Epoch 00098: val_accuracy did not improve from 0.86730
196/196 [==============================] - 40s 203ms/step - loss: 0.5191 - accuracy: 0.9201 - val_loss: 0.8791 - val_accuracy: 0.8238
Epoch 99/100
196/196 [==============================] - ETA: 0s - loss: 0.5221 - accuracy: 0.9217
Epoch 00099: val_accuracy improved from 0.86730 to 0.87560, saving model to my_model_final_best.h5
196/196 [==============================] - 40s 204ms/step - loss: 0.5221 - accuracy: 0.9217 - val_loss: 0.6899 - val_accuracy: 0.8756
Epoch 100/100
196/196 [==============================] - ETA: 0s - loss: 0.5236 - accuracy: 0.9222
Epoch 00100: val_accuracy did not improve from 0.87560
196/196 [==============================] - 40s 204ms/step - loss: 0.5236 - accuracy: 0.9222 - val_loss: 0.8072 - val_accuracy: 0.8526
```





在运行 100 个 epoch 后，模型基本收敛，最终测试集上的 accuracy 为 0.9222，训练集上的 accuracy 为 0.87560

具体到每个类别，结果如下：

```
Class 0: airplane accuracy: 85.30%
Class 1: automobile accuracy: 92.90%
Class 2: bird accuracy: 86.90%
Class 3: cat accuracy: 76.30%
Class 4: deer accuracy: 85.40%
Class 5: dog accuracy: 76.30%
Class 6: frog accuracy: 91.70%
Class 7: horse accuracy: 91.80%
Class 8: ship accuracy: 95.20%
Class 9: truck accuracy: 93.70%
```

从防止过拟合的方面看：dropout 和 L2 正则化系数的应用在很大程度上减少了模型的过拟合程度，增加了模型的泛化能力。这使得模型在训练集上表现更好的同时在测试集上的表现也有了很大的提升。

从模型预测准确率上来看：新增加的卷积层显著提升了模型预测的准确性。

虽然模型较上一版有了很大的提升，但是具体到每个类别时，我发现 cat 和 dog 的预测准确率偏低，与此同时，accuracy 的变化曲线还反映出了一个问题：模型收敛速度过慢。

为了进一步提高这两种相似种类的识别正确率、加快模型收敛速度，我在对现有框架进行分析后，尝试在如下两个方面对模型进行调整：

1. 进一步对图像进行数据增强
2. 在训练过程中，随着训练轮数的增加，模型参数逐渐接近最优解，此时学习率过大会导致模型在最优点两侧来回波动，难以稳定地收敛到最优解。因此我尝试对学习率进行动态调整。

2.4 基于实验经验构建的最终网络模型（网络模型优化+学习率动态调整）

在基本框架不变的情况下，通过提高网络中正则化参数的大小和设置频率来尝试降低模型的过拟合程度。

同时，模型采用动态学习率调整机制试图帮助优化器更好地控制学习率，从而提高训练效果和收敛速度。

最终网络模型

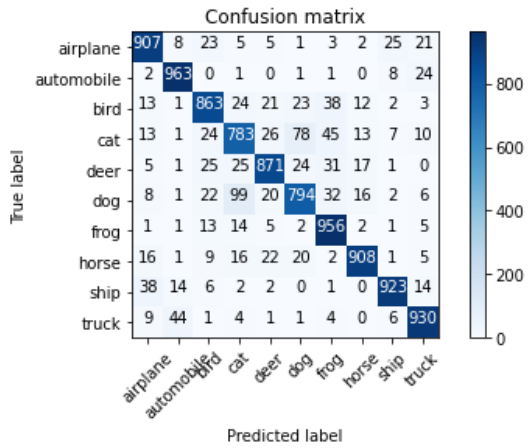
```
# 定义卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), # 输入通道数为3，输出通道数为64，卷积核大小为3x3，填充为1
    tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为64，输出通道数为64，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.05),

    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'), # 输入通道数为64，输出通道数为128，卷积核大小为3x3，填充为1
    tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为128，输出通道数为128，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    # 设置 kernel_regularizer 参数来实现 L2 正则化，其中 0.001 是 L2 正则化系数。
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'), # 输入通道数为128，输出通道数为256，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(256, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)), # 输入通道数为256，输出通道数为256，卷积核大小为3x3，填充为1
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),# 最大池化层，大小为2x2，步幅为2
    tf.keras.layers.Dropout(0.15),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(512, 3, padding='same', activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Flatten(), # 将张量展开为一维张量
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(4096, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, activation='relu'), # 全连接层1，输入维度为256*4*4，输出维度为1024
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(512, activation='relu',kernel_regularizer=regularizers.l2(0.001)), # 全连接层2，输入维度为1024，输出维度为512
    tf.keras.layers.Dense(10, activation='softmax') # 输出层，输入维度为512，输出维度为10
])
```

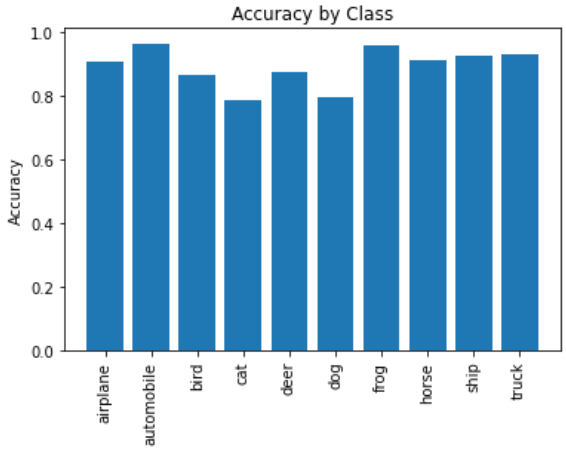
最终数据增强方案

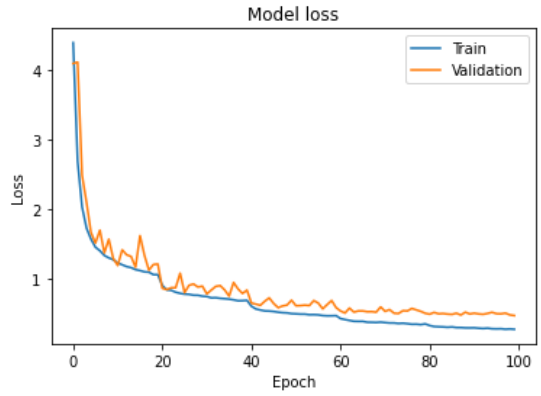
```
# 定义训练集和测试集的数据增强
train_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
    tf.keras.layers.experimental.preprocessing.RandomRotation(0.1),
    tf.keras.layers.experimental.preprocessing.RandomTranslation(0.1, 0.1),
    tf.keras.layers.experimental.preprocessing.RandomZoom(0.1),
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])

test_transform = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Rescaling(1./255)
])
```

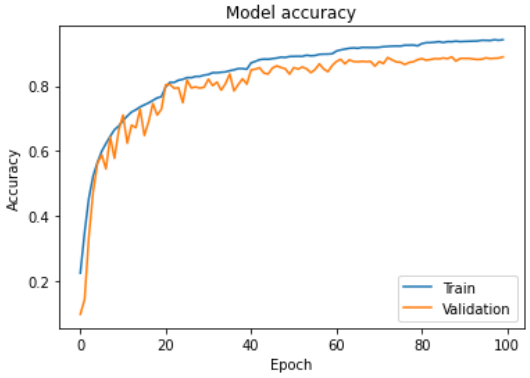
实验结果:

```
196/196 [==============================] - ETA: 0s - loss: 0.2748 - accuracy: 0.9403
Epoch 00097: val_accuracy did not improve from 0.88980
196/196 [==============================] - 48s 246ms/step - loss: 0.2748 - accuracy: 0.9403 - val_loss: 0.4899 - val_accuracy:
0.8846 - lr: 6.2500e-05
Epoch 98/100
196/196 [==============================] - ETA: 0s - loss: 0.2679 - accuracy: 0.9425
Epoch 00098: val_accuracy did not improve from 0.88980
196/196 [==============================] - 48s 246ms/step - loss: 0.2679 - accuracy: 0.9425 - val_loss: 0.4995 - val_accuracy:
0.8855 - lr: 6.2500e-05
Epoch 99/100
196/196 [==============================] - ETA: 0s - loss: 0.2719 - accuracy: 0.9411
Epoch 00099: val_accuracy did not improve from 0.88980
196/196 [==============================] - 50s 254ms/step - loss: 0.2719 - accuracy: 0.9411 - val_loss: 0.4745 - val_accuracy:
0.8862 - lr: 6.2500e-05
Epoch 100/100
196/196 [==============================] - ETA: 0s - loss: 0.2669 - accuracy: 0.9425
Epoch 00100: val_accuracy did not improve from 0.88980
196/196 [==============================] - 50s 254ms/step - loss: 0.2669 - accuracy: 0.9425 - val_loss: 0.4646 - val_accuracy:
0.8897 - lr: 6.2500e-05
```









在运行 100 个 epoch 后，模型收敛，最终测试集上的 accuracy 为 0.9425，训练集上的 accuracy 为 0.8898

具体到每个类别，结果如下:

```
Class 0: airplane accuracy: 90.70%
Class 1: automobile accuracy: 96.30%
Class 2: bird accuracy: 86.30%
```

```
Class 3: cat accuracy: 78.30%
Class 4: deer accuracy: 87.10%
Class 5: dog accuracy: 79.40%
Class 6: frog accuracy: 95.60%
Class 7: horse accuracy: 90.80%
Class 8: ship accuracy: 92.30%
Class 9: truck accuracy: 93.00%
```

从防止过拟合的方面看：修改和新添加的正则化参数减少了模型的过拟合程度，增加了模型的泛化能力。这使得模型在训练集上表现更好的同时在测试集上的表现也有了很大的提升。

从模型预测准确率上来看：新增加的数据增强方法以及学习率动态调整机制在一定程度上进一步提升了模型预测的准确性。

从模型收敛速度看：相较于之前的版本，学习率动态调整机制显著加快了模型的收敛速度，从绘制的图表中也可以清晰的看到，模型最终收敛的很好。

综上，最终的模型在预测准确率、单项预测准确率、收敛速度、泛化能力上都有了明显的提升。

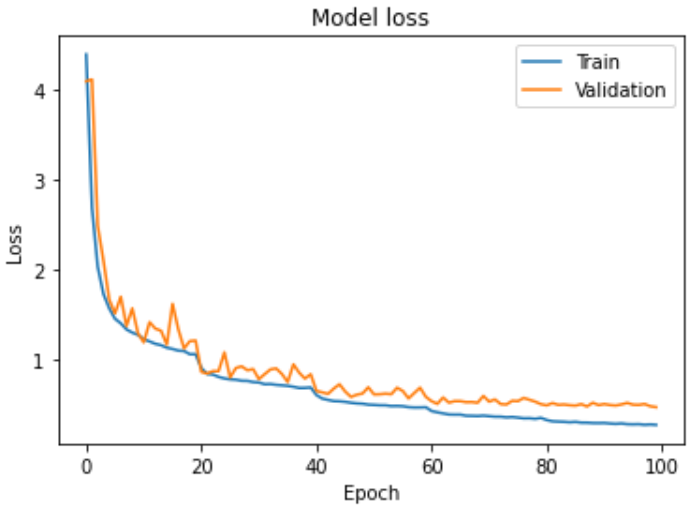
3. 实验分析

3.1 实验结果分析

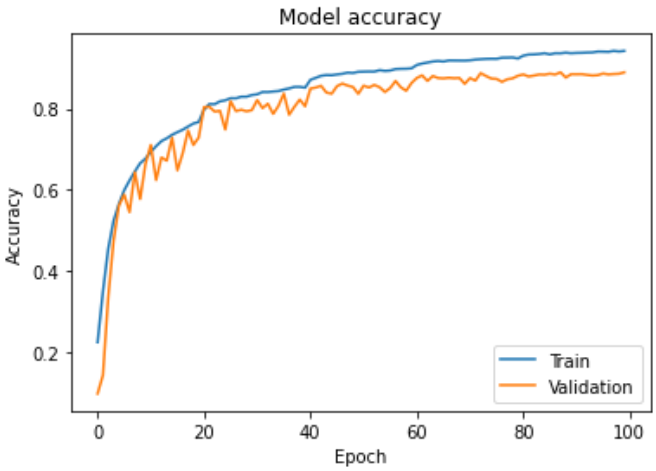
任务要求：设计一个卷积神经网络， 在 CIFAR-10 数据集上实现分类任务。

实验结果：在本实验的架构基础上，经过 100 个 epoch 的训练，最终模型取得的训练结果如下：

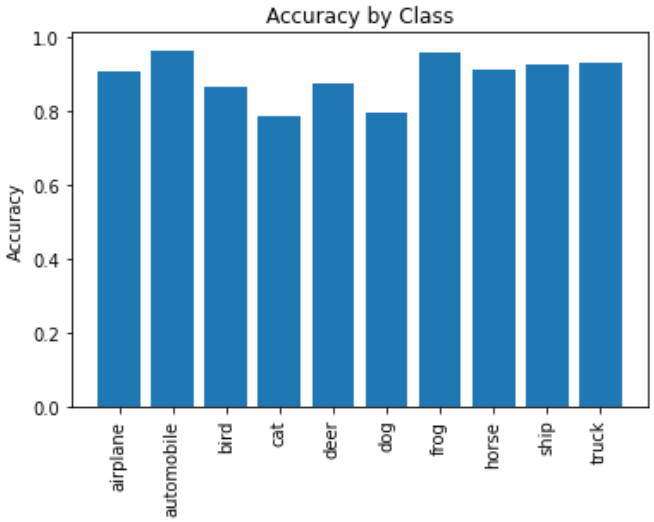
Loss: 测试集：0.45 左右



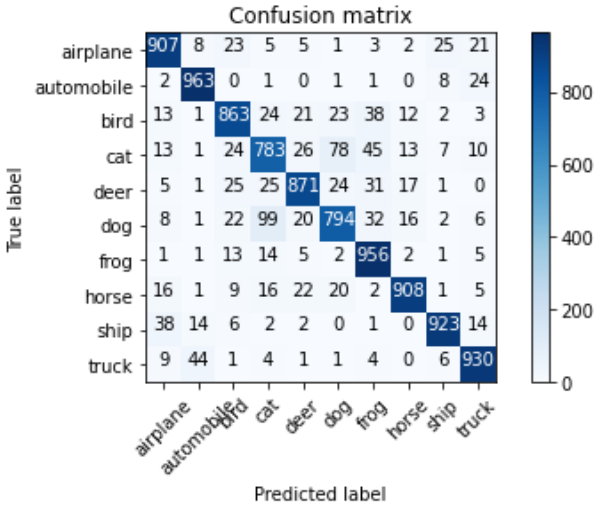
Accuracy: 测试集：0.9 左右



Class of Accuracy:



Confusion Matrix:



根据如上评价指标，可以发现本次设计的网络模型较好的实现了要求的十分类任务。

3.2 不同网络结构对比（含激活函数对比）

1) 网络层数增加

根据上述 2.2 与 2.3 的结果，一定范围内增加网络的层数可以显著提升模型的预测能力。

2) 数据增强

根据上述 2.1 与 2.2 的结果，数据增强操作可以在一定程度上提升模型的预测能力。

3) 正则化参数

根据上述 2.1 和 2.2 的结果，从防止过拟合的方面看。dropout 和 L2 正则化系数的应用在很大程度上减少了模型的过拟合程度，增加了模型的泛化能力。

4) 学习率调整

学习率动态调整机制显著加快了模型的收敛速度，使得模型可以稳定地收敛到最优解

5) 设置不同的激活函数

在实验中尝试将 relu 函数更换为 sigmoid 函数，使用 wandb 在训练过程中进行实时观测。在运行的过程中，可以观察到 sigmoid 激活函数在该问题中表现不佳，relu 函数在设置的模型中的表现明显优于 sigmoid 函数。

4. 总结

4.1 实验总结

本次实验设计了一个相对较深的网络模型，较好的实现了要求的基于卷积神经网络的 CIFAR-10 数据集分类任务。在实验的过程中我也对网络的整体架构以及网络的优化方案有了更加清晰的认识。

4.2 提交文件

包含实验报告、最终源代码（cv2_new_model_lr.ipynb）、部分过程源代码、实验结果文件、保存的模型（.PNG、.ipynb、.h5）