华中科技大学

2023

计算机视觉实验报告

上机实验三：基于剪枝算法的深度神经网络压缩

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 11 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 网络设计

a) 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')
```

b) 实验数据准备部分

```
# 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.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)
])
```

c） 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
])
```

d） 神经网络结构及各层名称输出部分

```
#打印神经网络结构
for layer in model.layers:
        for weight in layer.weights:
            print(weight.name, weight.shape)
```

```
# 打印神经网络各层名称
names = [layer.name for layer in model.layers]
print(names, len(names))
```

```
['conv2d', 'conv2d_1', 'batch_normalization', 'max_pooling2d', 'dropout', 'conv2d_2', 'conv2d_3', 'batch_normalization_1', 'max_pooling2d_
1', 'dropout_1', 'conv2d_4', 'batch_normalization_2', 'conv2d_5', 'batch_normalization_3', 'conv2d_6', 'batch_normalization_4', 'max_pooling
2d_2', 'dropout_2', 'conv2d_7', 'batch_normalization_5', 'conv2d_8', 'batch_normalization_6', 'conv2d_9', 'batch_normalization_7', 'max_pool
ing2d_3', 'dropout_3', 'conv2d_10', 'batch_normalization_8', 'conv2d_11', 'batch_normalization_9', 'conv2d_12', 'batch_normalization_10', 'm
ax_pooling2d_4', 'dropout_4', 'flatten', 'dropout_5', 'dense', 'dropout_6', 'dense_1', 'dropout_7', 'dense_2', 'dense_3'] 42
```

e） 最后一层卷积层寻找部分

```
# 打印寻找最后一个卷积层
conv_layer = model.get_layer(name='conv2d_12')
conv_layer
print(conv_layer.name)
for weight in conv_layer.weights:
    print(weight.name, weight.shape)
# conv2d_12/kernel:0 (3, 3, 512, 512) 卷积核大小3*3 输入通道数512 输出通道数512
```

f） 实验模型、数据导入及预处理

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

# 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_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

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)
#  让数据集对象 Dataset 在训练时预取出若干个元素，使得在 GPU 训练的同时 CPU 可以准备数据，从而提升训练流程的效率

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

g） 平均输出特征图输出部分

```
last_conv_layer = model.layers[-12]
last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
conv_outputs = last_conv_layer_model.predict(test_dataset)
# conv_outputs
average_output = tf.reduce_mean(conv_outputs, axis=0)
average_output
```

```
# 打印最后一层卷积层（剪枝前） 在整个测试数据集上的平均输出特征图
plt.imshow(average_output[:,:,0], cmap='gray') #打印第0张平均输出特征图
plt.show()
# plt.imshow(average_output[:,:,1], cmap='gray')
# plt.show()
row_num = int(np.ceil(np.sqrt(average_output.shape[2])))#将通道数开方取整，尽可能地使行列数相同
row_num
for index in range(1,average_output.shape[2]+1):#通过遍历的方式，将每个特征图拿出
    plt.subplot(row_num,row_num,index)
    plt.imshow(average_output[:,:,index-1],cmap='gray')
    plt.axis('off')
plt.show()
```

h） 平均激活值排序部分

```
# 对平均激活值进行排序
average_activations = tf.reduce_mean(conv_outputs, axis=(0,1,2)) # 计算每个神经元在整个测试数据集上的平均激活值
sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
for i in range(len(sorted_indices)): #打印排序后的神经元及其对应的平均激活值
    print(f"Neuron {sorted_indices[i]}: {average_activations[sorted_indices[i]]}")
    # 打印观察每个神经元在整个测试数据集上的平均激活值
```

i） 剪枝部分（含剪枝后模型评估部分）

```
# 单次剪枝代码
sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序 argsort函数返回的是数组值从小到大的索引值
# sorted_indices
# len(sorted_indices)
K = 1 # 要剪枝的神经元数
last_conv_layer = model.layers[-12]
weights, biases = last_conv_layer.get_weights()
for i in range(0, K):
    neuron_index = sorted_indices[i]
    weights[:,:,:,neuron_index] = 0 # 将该神经元的权重设为0
    biases[neuron_index] = 0 # 将该神经元的偏置设为0
last_conv_layer.set_weights([weights, biases]) # 更新最后一层卷积层的权重和偏置
```

```
accuracies = [] # 创建数组记录预测准确率变化
# for i in range(0,5):
for i in range(0,average_output.shape[2]):
    K = i # 要剪枝的神经元数
    last_conv_layer = model.layers[-12]
    last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
    conv_outputs = last_conv_layer_model.predict(test_dataset)
    weights, biases = last_conv_layer.get_weights()
    average_activations = tf.reduce_mean(conv_outputs, axis=(0,1,2)) # 计算每个神经元在整个测试数据集上的平均激活值
    sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
    for j in range(0, K):
#         print('j:', j)
        neuron_index = sorted_indices[j]
        weights[:,:,:,neuron_index] = 0 # 将该神经元的权重设为0
        biases[neuron_index] = 0 # 将该神经元的偏置设为0
    last_conv_layer.set_weights([weights, biases]) # 更新最后一层卷积层的权重和偏置
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # 进行测试
    _, accuracy = model.evaluate(test_dataset, verbose=0)
    accuracies.append(accuracy)
    print(f'k={K}, accuracy={accuracy}')
#     sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
#     for i in range(0,10): #打印排序后的神经元及其对应的平均激活值
#         print(f"Neuron {sorted_indices[i]}: {average_activations[sorted_indices[i]]}")
    print('K:', K)
```

j） accuracy 折线图输出部分

```
plt.plot(range(0, average_output.shape[2]), accuracies)
plt.xlabel('Number of pruned neurons')
plt.ylabel('Test accuracy')
plt.title('Test accuracy change process')
plt.show()
```

2. 实验过程设计

2.1 读取数据集和训练好的模型

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

# 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_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

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)
#  让数据集对象 Dataset 在训练时预取出若干个元素，使得在 GPU 训练的同时 CPU 可以准备数据，从而提升训练流程的效率
```

图 2.1 读取数据集和训练好的模型

2.2 寻找最后一层卷积层

思路：打印神经网络结构->找到最后一层卷积层->调用 model.get_layer()提取最后一个卷积层

```
# 打印神经网络各层名称
names = [layer.name for layer in model.layers]
print(names, len(names))
```

```
['conv2d', 'conv2d_1', 'batch_normalization', 'max_pooling2d', 'dropout', 'conv2d_2', 'conv2d_3', 'batch_normalization_1', 'max_pooling2d_
1', 'dropout_1', 'conv2d_4', 'batch_normalization_2', 'conv2d_5', 'batch_normalization_3', 'conv2d_6', 'batch_normalization_4', 'max_pooling
2d_2', 'dropout_2', 'conv2d_7', 'batch_normalization_5', 'conv2d_8', 'batch_normalization_6', 'conv2d_9', 'batch_normalization_7', 'max_pool
ing2d_3', 'dropout_3', 'conv2d_10', 'batch_normalization_8', 'conv2d_11', 'batch_normalization_9', 'conv2d_12', 'batch_normalization_10', 'm
ax_pooling2d_4', 'dropout_4', 'flatten', 'dropout_5', 'dense', 'dropout_6', 'dense_1', 'dropout_7', 'dense_2', 'dense_3'] 42
```

```
# #打印神经网络结构
# for layer in model.layers:
#         for weight in layer.weights:
#             print(weight.name, weight.shape)
```

```
# 打印寻找最后一个卷积层
conv_layer = model.get_layer(name='conv2d_12')
conv_layer
print(conv_layer.name)
for weight in conv_layer.weights:
    print(weight.name, weight.shape)
# conv2d_12/kernel:0 (3, 3, 512, 512) 卷积核大小3*3 输入通道数512 输出通道数512

conv_layer = model.get_layer(index=-12)
print(conv_layer.name)
for weight in conv_layer.weights:
    print(weight.name, weight.shape)

conv_layer = model.get_layer(index=-20)
print(conv_layer.name)
for weight in conv_layer.weights:
    print(weight.name, weight.shape)
```

```
conv2d_12
conv2d_12/kernel:0 (3, 3, 512, 512)
```

图 2.2 寻找最后一层卷积层代码

2.3 打印平均输出特征图（参考课件代码）

思路：提取最后一个卷积层->根据最后一层卷积层构建 tf.keras.Model 实例

->根据模型实例执行 predict，取得输出结果 ->调用 tf.reduce_mean()计算平均值->通过遍历的方式，打印每一张平均输出特征图

```
last_conv_layer = model.layers[-12]
last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
conv_outputs = last_conv_layer_model.predict(test_dataset)
# conv_outputs
average_output = tf.reduce_mean(conv_outputs, axis=0)
average_output
```

```
<tf.Tensor: shape=(2, 2, 512), dtype=float32, numpy=
array([[[0.35425472, 0.58658934, 0.16616865, ..., 0.3775052 ,
         0.253711  , 0.2337245 ],
        [0.18189467, 0.7158923 , 0.04594743, ..., 0.49970454,
         0.44316137, 0.65379816]],

       [[0.5165768 , 0.41051158, 0.18278298, ..., 0.20512286,
         0.19628273, 0.5447516 ],
        [0.13349736, 0.39791712, 0.3625206 , ..., 0.1542081 ,
         0.47224835, 0.23300618]]], dtype=float32)>
```

```
# 打印最后一层卷积层（剪枝前） 在整个测试数据集上的平均输出特征图
plt.imshow(average_output[:,:,0], cmap='gray') #打印第0张平均输出特征图
plt.show()
# plt.imshow(average_output[:,:,1], cmap='gray')
# plt.show()
row_num = int(np.ceil(np.sqrt(average_output.shape[2])))#将通道数开方取整，尽可能地使行列数相同
row_num
for index in range(1,average_output.shape[2]+1):#通过遍历的方式，将每个特征图拿出
    plt.subplot(row_num,row_num,index)
    plt.imshow(average_output[:,:,index-1],cmap='gray')
    plt.axis('off')
plt.show()
```

图 2.3 平均输出特征图打印代码

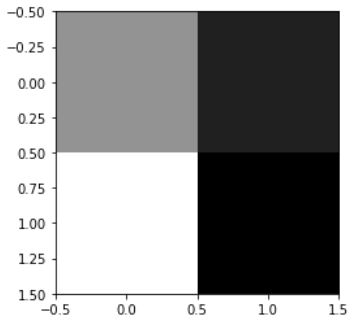

图 2.4 第 0 张平均输出特征图

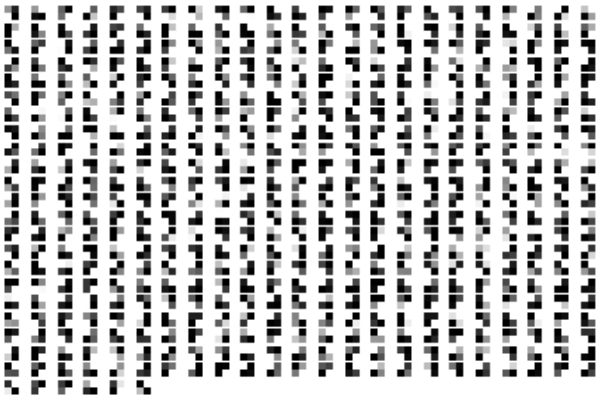

图 2.5 平均输出特征图

2.4 设计单次剪枝并验证单次剪枝结果

思路：调用 argsort 函数，取得数组值从小到大的索引值-> 调用 get_weights()获取 weights 与 biases->将当前未剪枝的 index 最小的神经元的权重和偏置设为 0 -> 调用 set_weights([weights, biases])更新最后一层卷积层的权重和偏置

```
# 单次剪枝代码
sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激
# sorted_indices
# len(sorted_indices)
K = 1 # 要剪枝的神经元数
last_conv_layer = model.layers[-12]
weights, biases = last_conv_layer.get_weights()
for i in range(0, K):
    neuron_index = sorted_indices[i]
    weights[:,:,:,neuron_index] = 0 # 将该神经元的权重设为0
    biases[neuron_index] = 0 # 将该神经元的偏置设为0
last_conv_layer.set_weights([weights, biases]) # 更新最后一层卷积层的权重和偏置
```

图 2.6 单次剪枝代码

验证单次剪枝结果：

思路：打印对应神经元权重和平均输出特征图，确认剪枝成功

```
# #检测权重是否被修改
# last_conv_layer = model.layers[-12]
# last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
# conv_outputs = last_conv_layer_model.predict(test_dataset)
# # conv_outputs
# average_activations = tf.reduce_mean(conv_outputs, axis=(0,1,2)) # 计算每个神经元在整个测试数据集上的平均激活值
# sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
# for i in range(len(sorted_indices)): #打印排序后的神经元及其对应的平均激活值
#     print(f"Neuron {sorted_indices[i]}: {average_activations[sorted_indices[i]]}")
```

```
Neuron 236: 0.0
Neuron 383: 0.05992547422647476
Neuron 125: 0.07034046947956085
Neuron 491: 0.08466646820306778
Neuron 218: 0.08590062707662582
```

```
: # #检测权重是否已经被修改
  # last_conv_layer = model.layers[-12]
  # last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
  # conv_outputs = last_conv_layer_model.predict(test_dataset)
  # average_output = tf.reduce_mean(conv_outputs, axis=0)
  # average_output
  # plt.imshow(average_output[:,:,236], cmap='gray')
  # plt.show()
```

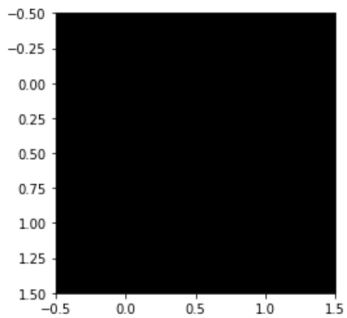

图 2.7 单次剪枝效果验证

2.5 循环剪枝设计

思路：将 K 设为 0

a) ->读取最后一个卷积层

b) ->计算每个神经元在整个测试数据集上的平均激活值

c) ->对平均激活值进行排序

d) ->将 index<=K 的神经元的权重和偏置设为 0

e) ->调用 set_weights([weights, biases])更新最后一层卷积层的权重和偏置

f) ->重新编译模型，进行测试，将测试结果保存在 accuracies 数组中

g) ->K++

->重复 a-g 直到 K= 特征图个数-1

```
accuracies = [] # 创建数组记录预测准确率变化
# for i in range(0,5):
for i in range(0,average_output.shape[2]):
    K = i # 要剪枝的神经元数
    last_conv_layer = model.layers[-12]
    last_conv_layer_model = tf.keras.models.Model(model.inputs, last_conv_layer.output)
    conv_outputs = last_conv_layer_model.predict(test_dataset)
    weights, biases = last_conv_layer.get_weights()
    average_activations = tf.reduce_mean(conv_outputs, axis=(0,1,2)) # 计算每个神经元在整个测试数据集上的平均激活值
    sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
    for j in range(0, K):
#         print('j:', j)
        neuron_index = sorted_indices[j]
        weights[:,:,:,neuron_index] = 0 # 将该神经元的权重设为0
        biases[neuron_index] = 0 # 将该神经元的偏置设为0
    last_conv_layer.set_weights([weights, biases]) # 更新最后一层卷积层的权重和偏置
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # 进行测试
    _, accuracy = model.evaluate(test_dataset, verbose=0)
    accuracies.append(accuracy)
    print(f'k={K}, accuracy={accuracy}')
#     sorted_indices = tf.argsort(average_activations, direction='ASCENDING') # 对平均激活值进行排序
#     for i in range(0,10): #打印排序后的神经元及其对应的平均激活值
#         print(f"Neuron {sorted_indices[i]}: {average_activations[sorted_indices[i]]}")
    print('K:', K)
K: 241
k=242, accuracy=0.7997999787330627
K: 242
k=243, accuracy=0.8004999756813049
K: 243
k=244, accuracy=0.800599992275238
K: 244
k=245, accuracy=0.7949000000953674
K: 245
k=246, accuracy=0.7929999828338623
```

图 2.8 模型最后一层卷积层剪枝

2.6 accuracy 折线图输出

思路：根据 2.6 中的 accuracies 数组，结合 matplotlib.pyplot 绘制折线图

```
plt.plot(range(0, average_output.shape[2]), accuracies)
plt.xlabel('Number of pruned neurons')
plt.ylabel('Test accuracy')
plt.title('Test accuracy change process')
plt.show()
```

图 2.9 accuracy 折线图输出代码

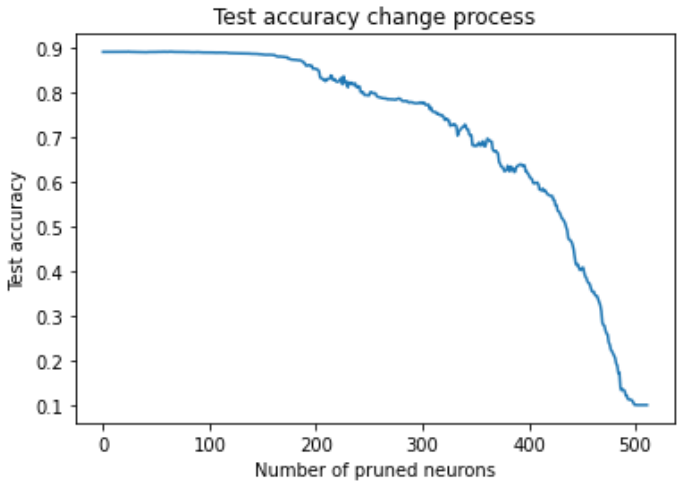

图 2.10 accuracy 折线图

3. 实验分析

3.1 实验结果分析

任务要求：对实验二构建的 CIFAR-10 数据集分类神经网络进行权重剪枝实现模型压缩。

实验结果及分析：

1. 画出最后一层卷积层（剪枝前） 在整个测试数据集上的平均输出特征图

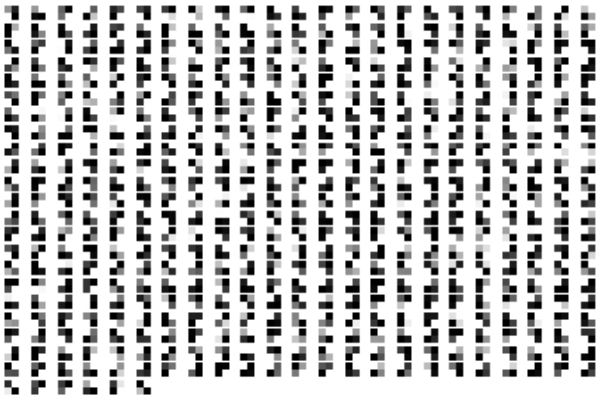

图 3.1 平均输出特征图

平均输出特征图分析：根据实验二的模型，由于输入的 32*32*3 的图像经过了三次最大池化后再进行最后一层的卷积，故每一张特征图大小为 2*2，特征图的个数为 512.

2. 画出横坐标为 K，纵坐标为网络分类 accuracy 的折线图

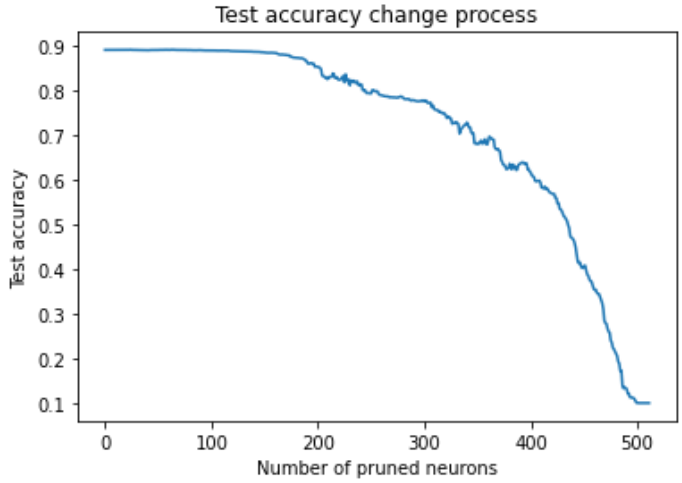

图 3.2 accuracy 折线图

从折线图反应的趋势来看，当剪枝的神经元数量在 100 以内时，模型在测试集上的表现并没有发生很大的改变，当剪枝的神经元数量达到 200 时，模型在测试集上的表现开始出现较大的下降。

综上，从模型压缩的角度来看，对最后一层卷积层激活水平最低的五分之一的神经元进行剪枝，并不会对模型的表现产生很大的影响。因此，在实际应用中可以考虑对该部分进行剪枝，对训练的模型进行压缩。

4. 总结

4.1 实验总结

本次实验基于实验二构建的 CIFAR-10 数据集分类神经网络进行了权重剪枝，完成了所要求的模型压缩任务。在实验的过程中我对模型压缩这一方面有了更深的认识，在今后的实际研究中，我也会更多的应用这一方面的技术，让训练出的模型具有更快的推理速度、占用更小的存储空间、在具备更低能耗的同时提高模型通用性。

4.2 提交文件

包含实验报告、最终源代码（final_model.ipynb）、实验结果文件（平均输出特征图、accuracy 折线图）、保存的模型（my_model_final_best.h5，考虑到大小原因未打包上传）

```
      LastWriteTime         Length Name
      -------------         ------ ----
2023/4/11     21:17          13806 accuracy折线图.png
2023/4/11     22:15         158694 final_model.ipynb
 2023/4/5     23:04      258889928 my_model_final_best.h5
2023/4/11     22:52        2545664 刘方兴-U202015550-实验报告 3.doc
 2023/4/5     23:02        1417194 刘方兴-U202015550-实验报告 3.pdf
2023/4/11     21:11           4151 第0张平均输出特征图.png
```