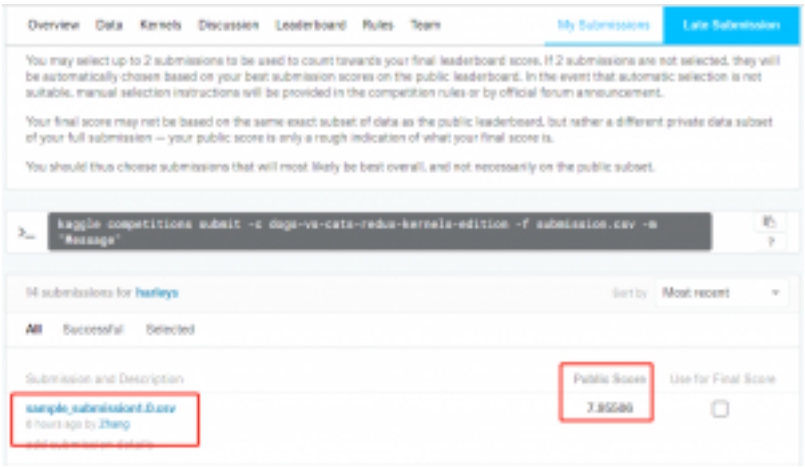
【Kaggle 竞赛】AlexNet 模型定义及实现

2018 年 11 月 30 日 by harley ·

在图像识别/目标检测领域，基本上 CNN 的天下，从基础的 AlexNet，再到后面更深的 GoogleNet、VGGNet 等，再到收敛速度更快、泛化性更强 ResNet 等残差网络，从 2012 年到现在 CNN 网络在图像识别/目标检测领域可谓是一个很好的方法。

根据我的经验，在 Kaggle 竞赛中，基础的 LeNet5、AlexNet 等 CNN 模型是不够用的，我在猫狗识别竞赛中使用 AlexNet 模型，并进行微调，同时稍微调整了不同的超参数，最后得到的 Public Score 只有 7.95506，排名 1247/1314。cifar10 竞赛稍微好些，Public Score0.27250，排名 171/231。

哎，压力山大，这个排名太差了，我都不好意思说出来，可终归是自己算法工程师之路的一个阶段，写下了也是让自己有压力，现阶段还需要好好沉淀自己。

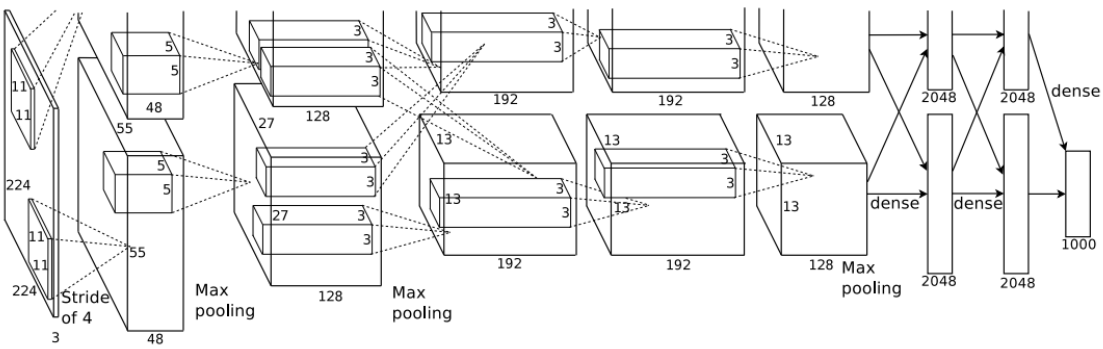


然后，我去网上查了资料，这个文章指出要选择好相应的 state-of-art 模型，state-of-art 模型有 ResNet、Inception v3、SPPNet 等，只要选择好了与任务相对应的 state-of-art 模型，在代码无 bug，没有实验设置错误的情况下，排进 Top 50% 甚至 Top 15% 难度很小。

当然，实际结果我还没有验证，因为我目前还在**没有实验设置错误这个阶段挣扎。**

AlexNet 模型概述

AlexNet 模型结构示意图如下所示：



从图中可以看出，AlexNet 包含输入层、5 个卷积层和 3 个全连接层。其中有 3 个卷积层还进行了最大池化。AlexNet 模型创新之处如下：

算法	作用
ReLU、多个GPU	提高训练速度
重叠池化	提高精度，不容易产生过拟合
局部归一化	提高精度
数据扩充、Dropout	减少过拟合

LRN（局部响应归一化）技术介绍：

Local Response Normalization(LRN)技术主要是深度学习训练时的一种提高准确度的技术方法。其中 caffe、tensorflow 等里面是很常见的方法，其跟激活函数是有区别的，LRN 一般是在激活、池化后进行的一种处理方法。LRN 归一化技术首次在 AlexNet 模型中提出这个概念。

AlexNet 模型实现

环境准备

系统：Windows10/Linux 系统

软件：Python3、TensorFlow 框架（主要是用一些低级 api，没有用高层封装(TensorFlow-Slim 、TFLearn 、Keras 和 Estimator）。

我这里把局部响应归一化放在池化层后面，但是网上有些版本放在卷积层后面。

输入的原始图像大小为 224×224×3 (RGB 图像)，在训练时会经过预处理变为 227×227×3。训练集图片尺寸可能不一定是 227*227，在输入前需要预处理裁剪成 227*227。

程序设计

```
# coding:utf-8
# filename:train.py
# Environment:windows10,python3,numpy,TensorFlow1.9,numpy,time
# Function:负责定义设计 AlexNet 网络

import os
import numpy as np
```

```
import tensorflow as tf
# 准备数据程序模块
import input_data
import time

IMG_W=227
IMG_H=227
IMG_C=3
BATCH_SIZE = 20
n_epoch = 10000
n_classes = 2
# 本地电脑训练对应路径地址
train_dir = "F:/Software/Python_Project/Classification-cat-dog/train/"
logs_train_dir =
"F:/Software/Python_Project/Classification-cat-dog/logs/"

# 云服务器训练对应路径地址
# train_dir = '/data/Dogs-Cats-Redux-Kernels-Edition/train/'
# logs_train_dir = '/data/Dogs-Cats-Redux-Kernels-Edition/logs/

#---------------------------------- 定 义 构 建 网 络
----------------------------
```

```
# 占位符,用于得到传递进来的真实训练样本,输入数据为猫狗识别数据集 batch
x = tf.placeholder(tf.float32,shape=[None,IMG_W,IMG_H,IMG_C],name='x')
y_ = tf.placeholder(tf.int32,shape=[None,],name='y_')

def inference(input_tensor, train, regularizer,n_classes):
    # conv1,输出矩阵大小(55,55,96)
    with tf.variable_scope('layer1-conv1'):
        conv1_weights = tf.get_variable("weight",[11,11,3,96],initializer=tf.truncated_normal_initializer(stddev=0.1))
        conv1_biases = tf.get_variable("bias", [96], initializer=tf.constant_initializer(0.0))  # 定义偏置变量并初始化
        conv1 = tf.nn.conv2d(input_tensor, conv1_weights, strides=[1, 4, 4, 1], padding='VALID')  # 'VALID'不添加
        relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases))  # 使用 ReLu 激活函数，完成去线性化

    # pool1 && norm1,输出矩阵大小(27,27,96)
    with tf.name_scope("layer2-pool1"):
```

```
            pool1 = tf.nn.max_pool(relu1, ksize = [1,3,3,1],strides=[1,2,2,1],padding="VALID")
            norm1 = tf.nn.lrn(pool1, depth_radius=4, bias=1.0, alpha=0.001/9.0,beta=0.75, name='norm1') # lrn 层，局部响应归一化

        # conv2,输出矩阵大小(27,27,256)
        with tf.variable_scope("layer3-conv2"):
            conv2_weights = tf.get_variable("weight",[5,5,96,256],initializer=tf.truncated_normal_initializer(stddev=0.1))
            conv2_biases = tf.get_variable("bias", [256], initializer=tf.constant_initializer(0.0))
            conv2 = tf.nn.conv2d(norm1, conv2_weights, strides=[1, 1, 1, 1], padding='SAME')
            relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_biases)) # 使用 ReLu 激活函数，完成去线性化

        # pool2 && norm2,输出矩阵大小(13,13,256)
        with tf.name_scope("layer4-pool2"):
            pool2 = tf.nn.max_pool(relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')
```

```
        norm2 = tf.nn.lrn(pool2, depth_radius=4, bias=1.0,
alpha=0.001/9.0,beta=0.75, name='norm1') # lrn 层，局部响应归一化

    # conv3,输出矩阵大小(13,13,384)
    with tf.variable_scope("layer5-conv3"):
        conv3_weights =
tf.get_variable("weight",[3,3,256,384],initializer=tf.truncated_normal_i
nitializer(stddev=0.1))
        conv3_biases = tf.get_variable("bias", [384],
initializer=tf.constant_initializer(0.0))
        conv3 = tf.nn.conv2d(norm2, conv3_weights, strides=[1, 1,
1, 1], padding='SAME')
        relu3 = tf.nn.relu(tf.nn.bias_add(conv3, conv3_biases))
# 使用 ReLu 激活函数，完成去线性化

    # conv4,输出矩阵大小(13,13,384)
    with tf.variable_scope("layer6-conv4"):
        conv4_weights =
tf.get_variable("weight",[3,3,384,384],initializer=tf.truncated_normal_i
nitializer(stddev=0.1))
        conv4_biases = tf.get_variable("bias", [384],
initializer=tf.constant_initializer(0.0))
```

```
            conv4 = tf.nn.conv2d(relu3, conv4_weights, strides=[1, 1, 1,
1], padding='SAME')
            relu4 = tf.nn.relu(tf.nn.bias_add(conv4, conv4_biases))
# 使用 ReLu 激活函数，完成去线性化

        # conv5,输出矩阵大小(13,13,256)
        with tf.variable_scope("layer7-conv5"):
            conv5_weights =
tf.get_variable("weight",[3,3,384,256],initializer=tf.truncated_normal_i
nitializer(stddev=0.1))
            conv5_biases = tf.get_variable("bias", [256],
initializer=tf.constant_initializer(0.0))
            conv5 = tf.nn.conv2d(relu4, conv5_weights, strides=[1, 1, 1,
1], padding='SAME')
            relu5 = tf.nn.relu(tf.nn.bias_add(conv5, conv5_biases))
# 使用 ReLu 激活函数，完成去线性化

        # pool5 && norm5,输出矩阵大小(6,6,256),13=(13-3)/2+1
        with tf.name_scope("layer8-pool5"):
            pool5 = tf.nn.max_pool(relu5, ksize=[1, 3, 3, 1], strides=[1,
2, 2, 1], padding='VALID')        # 'VALID'不添加
```

```
    # 将第 5 层池化层的输出转换为第 6 层全连接层的输入格式(向量)
    pool_shape = pool5.get_shape().as_list()
    nodes = pool_shape[1]*pool_shape[2]*pool_shape[3]     # 计算矩阵拉直成向量之后的长度,(pool_shape[1]*pool_shape[2]*pool_shape[3])=(6*6*256)
    reshaped = tf.reshape(pool5,[-1,nodes])                    # 通过 tf.reshaped 函数将第 5 层池化层的输出变成一个 batch 的向量

    # fc1,输出矩阵大小(-1,4096)
    with tf.variable_scope('layer9-fc1'):
        fc1_weights = tf.get_variable("weight", [nodes, 4096],

initializer=tf.truncated_normal_initializer(stddev=0.1))
        if regularizer != None: tf.add_to_collection('losses', regularizer(fc1_weights))
        fc1_biases = tf.get_variable("bias", [4096], initializer=tf.constant_initializer(0.1))

        fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_weights) + fc1_biases)
```

```
            if train: fc1 = tf.nn.dropout(fc1, 0.5)
# 使用丢失输出技巧 dropout

        # fc2,输出矩阵大小(-1,4096)
        with tf.variable_scope('layer10-fc2'):
            fc2_weights = tf.get_variable("weight", [4096, 4096],

initializer=tf.truncated_normal_initializer(stddev=0.1))
            if regularizer != None: tf.add_to_collection('losses',
regularizer(fc2_weights))
            fc2_biases = tf.get_variable("bias", [4096],
initializer=tf.constant_initializer(0.1))

            fc2 = tf.nn.relu(tf.matmul(fc1, fc2_weights) + fc2_biases)
            if train: fc2 = tf.nn.dropout(fc2, 0.5)
# 使用丢失输出技巧 dropout

        # fc3,输出层输出矩阵大小(-1,n_classes)=(-1,10)
        with tf.variable_scope('layer11-fc3'):
            fc3_weights = tf.get_variable("weight", [4096,
n_classesasses],
```

```
initializer=tf.truncated_normal_initializer(stddev=0.1))
        if regularizer != None: tf.add_to_collection('losses', regularizer(fc3_weights))
        fc3_biases = tf.get_variable("bias", [n_classes], initializer=tf.constant_initializer(0.1))
        logit = tf.matmul(fc2, fc3_weights) + fc3_biases

    # shape:(batch_size,num_classes)=(16,10)
    return logit

# ------------------------------网络结束，定义损失、评估模型------------------------------------
regularizer = tf.contrib.layers.l2_regularizer(0.0001)
# logits 是一个 batch_size*10 的二维数组
logits = inference(x,True,regularizer,n_classes)
print(logits)                                                    # 仅供测试程序时用，迭代训练模型时建议注释掉
# (小处理)将 logits 乘以 1 赋值给 logits_eval，定义 name，方便在后续调用模型时通过 tensor 名字调用输出 tensor
b = tf.constant(value=1,dtype=tf.float32)
logits_eval = tf.multiply(logits,b,name='logits_eval')
```

```
#计算交叉熵作为刻画预测值和真实值之间差距的损失函数
cross_entroy =
tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
labels=y_)
#计算在当前 batch 中所有样例的交叉熵平均值
loss =
tf.reduce_mean(cross_entroy,name='loss')+tf.add_n(tf.get_collection('l
osses'))
#使用 tf.train.AdamOptimizer 优化算法来优化损失函数
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)

#计算模型在一个 batch 数据上的正确率
correct_prediction = tf.equal(tf.cast(tf.argmax(logits,1),tf.int32), y_)
acc = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(loss,acc)                                    # 仅供测
试程序时用，迭代训练模型时建议注释掉
```

输出结果

这里还没有开始训练，所以只是加了 2 个 printf 来打印输出 logits、loss、acc 三个张量。输出结果如下图：

```
There are 12500 cats
```

There are 12500 dogs

label: 1

<matplotlib.figure.Figure at 0x1d50d0d3da0>

<matplotlib.figure.Figure at 0x1d50d0e0d68>

Tensor("layer11-fc3/add:0", shape=(?, 2), dtype=float32)

Tensor("add:0", shape=(), dtype=float32) Tensor("Mean:0", shape=(), dtype=float32)

总结

这里主要使用的是 TensorFlow 的低级 api 构建 AlexNet 网络，如果是像 VGGNet、ResNet50 等这些很深的模型，建议要用 TensorFLow 的高级封装(TensorFlow-Slim、TFLearn、Keras 和 Estimator）去编写模型定义程序。