计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：卷积神经网络构建及应用		学号：201800130086
日期：2020.11.16	班级： 18 智能班	姓名： 徐鹏博
Email：hsupengbo@163.com		

实验目的：

理解卷积神经网络，构建神经网络模型并应用

- 完成 Convolutional model -Step by Step
- 完成 Convolutional model - Application

实验软件和硬件环境：

Anaconda3-64bit
Tensorflow1.15
Python 3.6

实验原理和方法：

我们要实现一个拥有卷积层（CONV）和池化层（POOL）的网络，它包含了前向和反向传播。

卷积模块，包含函数：使用 0 扩充边界，卷积窗口，前向卷积， 反向卷积（可选）

池化模块，包含函数：前向池化 ，创建掩码， 值分配 ，反向池化（可选）

卷积神经网络 - 前向传播

在前向传播的过程中，我们将使用多种过滤器对输入的数据进行卷积操作，每个过滤器会产生一个 2D 的矩阵，我们可以把它们堆叠起来，于是这些 2D 的卷积矩阵就变成了高维的矩阵。

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_C = 过滤器数量$$

池化层的前向传播

要在同一个函数中实现最大值池化层和均值池化层

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$

$$n_C = n_{C_{prev}}$$

最大值池化层：在输入矩阵中滑动一个大小为 fxf 的窗口，选取窗口里的值中的最大值，然后作为输出的一部分。

均值池化层：在输入矩阵中滑动一个大小为 fxf 的窗口，计算窗口里的值中的平均值，然后这个均值作为输出的一部分。

卷积层的反向传播

$$dA+= \sum_{h=0}^{n_H} \sum_{w=0}^{nW} W_c \times dZ_{hw}$$

$$dW_c+= \sum_{h=0}^{n_H} \sum_{w=0}^{nW} a_{slice} \times dZ_{hw}$$

$$db = \sum_{h} \sum_{w} dZ_{hw}$$

实验步骤：（不要求罗列完整源代码）

Convolutional Neural Networks: Step by Step

Convolutional Neural Networks

Zero-Padding

```
def zero_pad(X, pad):
    X_pad = np.pad(X,((0,0),(pad,pad),(pad,pad),(0,0)),'constant')
    return X_pad
```

Single step of convolution

```
def conv_single_step(a_slice_prev, W, b):
    s = a_slice_prev * W + b
    Z = np.sum(s)
    return Z
```

Convolutional Neural Networks - Forward pass

```
def conv_forward(A_prev, W, b, hparameters):
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape
    stride = hparameters['stride']
    pad = hparameters['pad']
    n_H = int((n_H_prev - f+ 2*pad)/stride + 1)
    n_W = int((n_W_prev - f+ 2*pad)/stride + 1)
    Z = np.zeros((m,n_H,n_W,n_C))
    A_prev_pad = zero_pad(A_prev, pad)
    for i in range(m):                          # loop over the batch of training examples
        a_prev_pad = A_prev_pad[i,:,:,:]  # Select ith training example's padded activation
        for h in range(n_H):                    # loop over vertical axis of the output volume
            for w in range(n_W):                # loop over horizontal axis of the output volume
                for c in range(n_C):            # loop over channels (= #filters) of the output volume
```

```
                vert_start = h * stride
                vert_end = vert_start + stride
                horiz_start = w * stride
                horiz_end = horiz_start + stride
                a_slice_prev = a_prev_pad[vert_start:vert_end,horiz_start:horiz_end,:]
                Z[i, h, w, c] = conv_single_step(a_slice_prev, W[:,:,:,c], b[:,:,:,c])
    assert(Z.shape == (m, n_H, n_W, n_C))
    cache = (A_prev, W, b, hparameters)
    return Z, cache
```

Pooling layer

Forward Pooling

```
def pool_forward(A_prev, hparameters, mode = "max"):
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    f = hparameters["f"]
    stride = hparameters["stride"]
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev
    A = np.zeros((m, n_H, n_W, n_C))

    for i in range(m):                    # loop over the training examples
        for h in range(n_H):                  # loop on the vertical axis of the output volume
            for w in range(n_W):                # loop on the horizontal axis of the output volume
                for c in range (n_C):           # loop over the channels of the output volume

                    vert_start = h * stride
                    vert_end = vert_start+f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    a_prev_slice = A_prev[i,vert_start:vert_end,horiz_start:horiz_end,c]
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice)
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice)

    cache = (A_prev, hparameters)
    assert(A.shape == (m, n_H, n_W, n_C))
    return A, cache
```

Backpropagation in convolutional neural networks

Convolutional layer backward pass

```
def conv_backward(dZ, cache):
    (A_prev, W, b, hparameters) = cache
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape
```

```
    stride = hparameters['stride']
    pad = hparameters['pad']
    (m, n_H, n_W, n_C) = dZ.shape

    dA_prev = np.zeros((m,n_H_prev,n_W_prev,n_C_prev))
    dW = np.zeros((f,f,n_C_prev,n_C))
    db = np.zeros((f,f,n_C_prev,n_C))
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)

    for i in range(m):
        a_prev_pad = A_prev_pad[i,:,:,:]
        da_prev_pad = dA_prev_pad[i,:,:,:]
        for h in range(n_H):                   # loop over vertical axis of the output volume
            for w in range(n_W):               # loop over horizontal axis of the output volume
                for c in range(n_C):           # loop over the channels of the output volume

                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    a_slice = a_prev_pad[vert_start:vert_end,horiz_start:horiz_end,:]
                    da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
                    dW[:,:,:,c] += a_slice * dZ[i,h,w,c]
                    db[:,:,:,c] += dZ[i,h,w,c]
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad,pad:-pad,:]
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))
    return dA_prev, dW, db
```

Pooling layer - backward pass

```
def create_mask_from_window(x):
    mask = np.max(x)
    return mask

def distribute_value(dz, shape):
    (n_H, n_W) = shape
    average = dz / (n_H * n_W)
    a = np.ones((n_H,n_W))*average
    return a
def pool_backward(dA, cache, mode = "max"):
    (A_prev, hparameters) = cache
    stride = hparameters['stride']
    f = hparameters['f']

    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape
```

```
    dA_prev = np.zeros(A_prev.shape)

    for i in range(m):                       # loop over the training examples
        a_prev = A_prev[i,:,:,:]
        for h in range(n_H):                 # loop on the vertical axis
            for w in range(n_W):             # loop on the horizontal axis
                for c in range(n_C):         # loop over the channels (depth)
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f

                    if mode == "max":
                        a_prev_slice = a_prev[vert_start:vert_end,horiz_start:horiz_end,c]
                        mask = create_mask_from_window(a_prev_slice)
                        dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += np.multiply(mask, dA[i, h,
w, c])

                    elif mode == "average":
                        da = dA[i,h,w,c]
                        shape = (f,f)
                        dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += distribute_value(da, shape)
    assert(dA_prev.shape == A_prev.shape)
    return dA_prev
```

Convolution model - Application

Create placeholders

```
def create_placeholders(n_H0, n_W0, n_C0, n_y):
    X = tf.placeholder(tf.float32, shape=(None, n_H0, n_W0, n_C0))
    Y = tf.placeholder(tf.float32,shape=(None,n_y))
    return X, Y
```

Initialize parameters

```
def initialize_parameters():
    tf.set_random_seed(1)
    W1 = tf.get_variable("W1", [4, 4, 3, 8], initializer=tf.contrib.layers.xavier_initializer(seed=0))
    W2 = tf.get_variable("W2", [2, 2, 8, 16], initializer=tf.contrib.layers.xavier_initializer(seed=0))

    parameters = {"W1": W1,
                  "W2": W2}
    return parameters
```

Forward propagation

```
def forward_propagation(X, parameters):
    W1 = parameters['W1']
    W2 = parameters['W2']
```

```
    Z1 = tf.nn.conv2d(X, W1, strides=(1, 1, 1, 1), padding='SAME')
    A1 = tf.nn.relu(Z1)
    P1 = tf.nn.max_pool(A1, ksize=(1, 8, 8, 1), strides=(1, 8, 8, 1), padding='SAME')
    Z2 = tf.nn.conv2d(P1, W2, strides=(1, 1, 1, 1), padding='SAME')
    A2 = tf.nn.relu(Z2)
    P2 = tf.nn.max_pool(A2, ksize=(1, 4, 4, 1), strides=(1, 4, 4, 1), padding='SAME')
    P2 = tf.contrib.layers.flatten(P2)
    Z3 = tf.contrib.layers.fully_connected(P2, num_outputs = 6, activation_fn=None)
    return Z3
```

Compute cost

```
def compute_cost(Z3, Y):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits = Z3, labels = Y))
    return cost
```

Model

```
def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.009,
          num_epochs = 100, minibatch_size = 64, print_cost = True):
    ops.reset_default_graph()                     # to be able to rerun the model without overwriting tf
variables
    tf.set_random_seed(1)                         # to keep results consistent (tensorflow seed)
    seed = 3                                      # to keep results consistent (numpy seed)
    (m, n_H0, n_W0, n_C0) = X_train.shape
    n_y = Y_train.shape[1]
    costs = []
    X, Y = create_placeholders(n_H0,n_W0,n_C0,n_y)
    parameters = initialize_parameters()
    Z3 = forward_propagation(X,parameters)
    cost = compute_cost(Z3,Y)

    optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(cost)

    init = tf.global_variables_initializer()
    with tf.Session() as sess:

        sess.run(init)

        for epoch in range(num_epochs):

            minibatch_cost = 0.
            num_minibatches = int(m / minibatch_size)
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train, minibatch_size, seed)

            for minibatch in minibatches:
                (minibatch_X, minibatch_Y) = minibatch
                _ , temp_cost = sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})
```

```
            minibatch_cost += temp_cost / num_minibatches

        if print_cost == True and epoch % 5 == 0:
            print ("Cost after epoch %i: %f" % (epoch, minibatch_cost))
        if print_cost == True and epoch % 1 == 0:
            costs.append(minibatch_cost)

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()
     predict_op = tf.argmax(Z3, 1)
    correct_prediction = tf.equal(predict_op, tf.argmax(Y, 1))

    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print(accuracy)
    train_accuracy = accuracy.eval({X: X_train, Y: Y_train})
    test_accuracy = accuracy.eval({X: X_test, Y: Y_test})
    print("Train Accuracy:", train_accuracy)
    print("Test Accuracy:", test_accuracy)

    return train_accuracy, test_accuracy, parameters
```

结论分析与体会：

卷积层的反向传播相比前向传播复杂一些。先实现 dA,dW,db 的计算，反向传播函数 conv_backward 需要把所有的训练样本的过滤器、权值、高度、宽度都要加进来，然后使用千米三个公式计算对应的梯度

前向传播首先是经过卷积层，然后滑动地取卷积层最大值构成了池化层，需要记录最大值的位置，才能反向传播到卷积层。

在 TensorFlow 里面有一些可以直接拿来用的函数，比如
tf.nn.conv2d, tf.nn.relu(), tf.contrib.layers.flatten(), tf.contrib.layers.fully_connected().
前向传播模型的大概步骤就是
CONV2D→RELU→MAXPOOL→CONV2D→RELU→MAXPOOL→FULLCONNECTED