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：Convolutional Neural Networks and ResNets		学号：201900130151
日期：2021.11.2	班级： 人工智能	姓名： 莫甫龙
Email：m1533979510@163.com		

实验目的：

了解卷积神经网络的结构网络

了解 Resnet 网络

实验软件和硬件环境：

Vs code

Win11

实验原理和方法：

```
a = np.pad(a, ((0,0), (1,1), (0,0), (3,3), (0,0)), 'constant', constant_values = (..,..))
```

使用 np.pad 函数来使用 0 扩充边界。

卷积神经网络的前向传播：

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

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

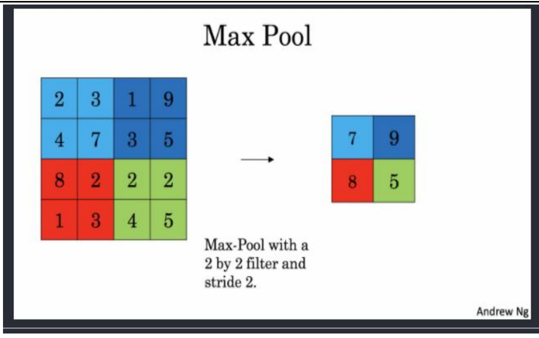
$$n_C = \text{number of filters used in the convolution}$$

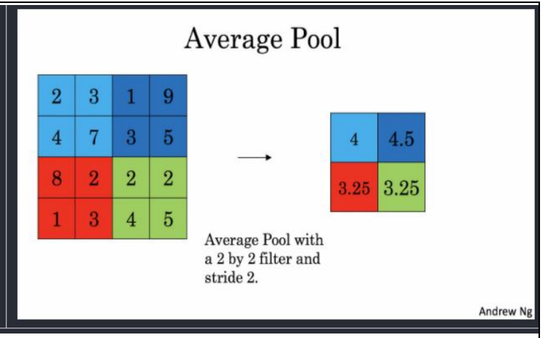
池化层的前向传播：

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

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

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





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。

均值池化层： 在输入矩阵中滑动一个大小为 fxf 的窗口，计算窗口中所有值的平均值，然后作为输出。

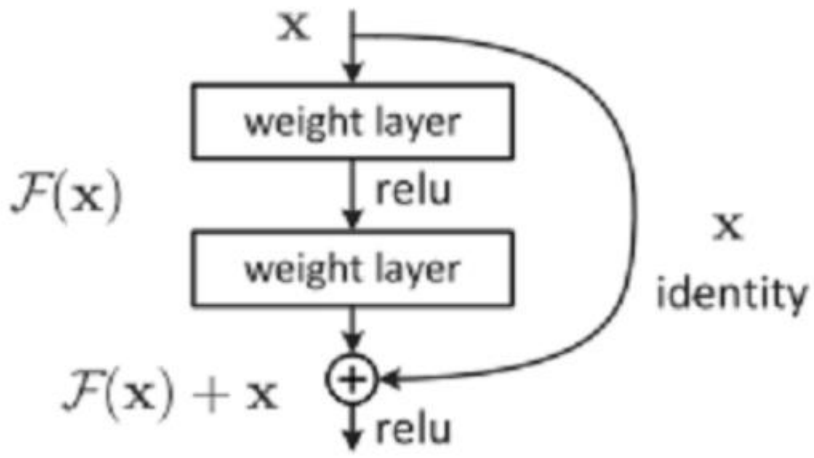
卷积层的反向传播：

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

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

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

Resnet



实验步骤：（不要求罗列完整源代码）

Convolutional Neural Networks:Step by Step:

Zero_pad:

```
### START CODE HERE ### (≈ 1 line)
X_pad = np.pad(X,((0,0),(pad,pad),(pad,pad),(0,0)),'constant')
### END CODE HERE ###
```

conv_single_step:

```
# Element-wise product between a_slice and W. Add bi
s =W*a_slice_prev
# Sum over all entries of the volume s
Z = np.sum(s)+b
### END CODE HERE ###
```

conv_forward:

```
    # Retrieve dimensions from A_prev's shape (≈1 line)
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve dimensions from W's shape (≈1 line)
    (f, f, n_C_prev, n_C) = W.shape

    # Retrieve information from "hparameters" (≈2 lines)
    stride = hparameters['stride']
    pad = hparameters['pad']

    # Compute the dimensions of the CONV output volume using the formula given above. Hint: use int() to floor. (≈2 line
    n_H = int((n_H_prev-f+2*pad)/stride+1)
    n_W = int((n_W_prev-f+2*pad)/stride+1)

    # Initialize the output volume Z with zeros. (≈1 line)
    Z = np.zeros((m,n_H,n_W,n_C))

    # Create A_prev_pad by padding A_prev
    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

                # Find the corners of the current "slice" (≈4 lines)
                vert_start = h * stride
                vert_end = vert_start + f
                horiz_start = w * stride
                horiz_end = horiz_start + f

                # Use the corners to define the (3D) slice of a_prev_pad (See Hint above the cell). (≈1 line)
                a_slice_prev = a_prev_pad[vert_start:vert_end,horiz_start:horiz_end,:]

                # Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈1 line)
                Z[i, h, w, c] = conv_single_step(a_slice_prev, W[:,:,:,c], b[:,:,:,c])

### END CODE HERE ###

# Making sure your output shape is correct
assert(Z.shape == (m, n_H, n_W, n_C))

# Save information in "cache" for the backprop
cache = (A_prev, W, b, hparameters)

return Z, cache
```

pool_forward:

```python
# Retrieve dimensions from the input shape
(m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

# Retrieve hyperparameters from "hparameters"
f = hparameters["f"]
stride = hparameters["stride"]

# Define the dimensions of the output
n_H = int(1 + (n_H_prev - f) / stride)
n_W = int(1 + (n_W_prev - f) / stride)
n_C = n_C_prev

# Initialize output matrix A
A = np.zeros((m, n_H, n_W, n_C))
```

```python
### START CODE HERE ###
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

                # Find the corners of the current "slice" (≈4 lines)
                vert_start = h * stride
                vert_end = vert_start+f
                horiz_start = w * stride
                horiz_end = horiz_start + f

                # Use the corners to define the current slice on the ith training example of A_prev, channel c. (≈1 line)
                a_prev_slice = A_prev[i,vert_start:vert_end,horiz_start:horiz_end,c]

                # Compute the pooling operation on the slice. Use an if statment to differentiate the modes. Use np.max/np.mean.
                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)

### END CODE HERE ###

# Store the input and hparameters in "cache" for pool_backward()
cache = (A_prev, hparameters)

# Making sure your output shape is correct
assert(A.shape == (m, n_H, n_W, n_C))
```

conv_backward:

```python
    # Retrieve information from "cache"
    (A_prev, W, b, hparameters) = cache

    # Retrieve dimensions from A_prev's shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve dimensions from W's shape
    (f, f, n_C_prev, n_C) = W.shape

    # Retrieve information from "hparameters"
    stride = hparameters['stride']
    pad = hparameters['pad']

    # Retrieve dimensions from dZ's shape
    (m, n_H, n_W, n_C) = dZ.shape

    # Initialize dA_prev, dW, db with the correct shapes
    dA_prev = np.zeros_like(A_prev)
    dW = np.zeros_like(W)
    db = np.zeros_like(b)

    # Pad A_prev and dA_prev
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)
```

```python
    for i in range(m):                          # loop over the training examples

        # select ith training example from A_prev_pad and dA_prev_pad
        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

                    # Find the corners of the current "slice"
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f

                    # Use the corners to define the slice from a_prev_pad
                    a_slice = a_prev_pad[vert_start:vert_end,horiz_start:horiz_end,:]

                    # Update gradients for the window and the filter's parameters using the code formulas given
                    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]

        # Set the ith training example's dA_prev to the unpaded da_prev_pad (Hint: use X[pad:-pad, pad:-pad, :])
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad,pad:-pad,:]
    ### END CODE HERE ###

    # Making sure your output shape is correct
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

create_mask_from_window:

```python
    """

    ### START CODE HERE ### (≈1 line)
    mask = (np.max(x)==x)
    ### END CODE HERE ###

    return mask
```

distribute_value:

```python
    ### START CODE HERE ###
    # Retrieve dimensions from shape (≈1 line)
    (n_H, n_W) = shape

    # Compute the value to distribute on the matrix (≈1 line)
    average = dz/(n_H*n_W)

    # Create a matrix where every entry is the "average" value (≈1 line)
    a = np.ones(shape) * average
    ### END CODE HERE ###

    return a
```

pool_backward:

```python
    # Retrieve information from cache (≈1 line)
    (A_prev, hparameters) = cache

    # Retrieve hyperparameters from "hparameters" (≈2 lines)
    stride = hparameters['stride']
    f = hparameters['f']

    # Retrieve dimensions from A_prev's shape and dA's shape (≈2 lines)
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    # Initialize dA_prev with zeros (≈1 line)
    dA_prev = np.zeros_like(A_prev)
```

```python
for i in range(m):                          # loop over the training examples

    # select training example from A_prev (≈1 line)
    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)

                # Find the corners of the current "slice" (≈4 lines)
                vert_start = h * stride
                vert_end = vert_start + f
                horiz_start = w * stride
                horiz_end = horiz_start + f

                # Compute the backward propagation in both modes.
                if mode == "max":

                    # Use the corners and "c" to define the current slice from a_prev (≈1 line)
                    a_prev_slice = a_prev[vert_start:vert_end, horiz_start:horiz_end, c]
                    # Create the mask from a_prev_slice (≈1 line)
                    mask = create_mask_from_window(a_prev_slice)
                    # Set dA_prev to be dA_prev + (the mask multiplied by the correct entry of dA) (≈1 line)
                    dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += np.multiply(mask, dA[i,h,w,c])

                elif mode == "average":

                    # Get the value a from dA (≈1 line)
                    da = dA[i,h,w,c]
                    # Define the shape of the filter as fxf (≈1 line)
                    shape = (f,f)
                    # Distribute it to get the correct slice of dA_prev. i.e. Add the distributed value of da. (≈1 line)
                    dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += distribute_value(da, shape)
```

Convolutional Neural Networks: Application:

create_placeholders:

```python
    ### START CODE HERE ### (≈2 lines)
    X = tf.placeholder(tf.float32, shape=(None, n_H0, n_W0, n_C0))
    Y = tf.placeholder(tf.float32,shape=(None,n_y))
    ### END CODE HERE ###

    return X, Y
```

initialize_parameters:

```python
    tf.set_random_seed(1)                              # so that your "random" numbers match ours

    ### START CODE HERE ### (approx. 2 lines of code)
    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))
    ### END CODE HERE ###

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

forward_propagation:

```python
# Retrieve the parameters from the dictionary "parameters"
W1 = parameters['W1']
W2 = parameters['W2']

### START CODE HERE ###
# CONV2D: stride of 1, padding 'SAME'
Z1 = tf.nn.conv2d(X, W1, strides=(1, 1, 1, 1), padding='SAME')
# RELU
A1 = tf.nn.relu(Z1)
# MAXPOOL: window 8x8, sride 8, padding 'SAME'
P1 = tf.nn.max_pool(A1, ksize=(1, 8, 8, 1), strides=(1, 8, 8, 1), padding='SAME')
# CONV2D: filters W2, stride 1, padding 'SAME'
Z2 = tf.nn.conv2d(P1,W2,strides=(1,1,1,1),padding='SAME')
# RELU
A2 = tf.nn.relu(Z2)
# MAXPOOL: window 4x4, stride 4, padding 'SAME'
P2 = tf.nn.max_pool(A2, ksize=(1, 4, 4, 1), strides=(1, 4, 4, 1), padding='SAME')
# FLATTEN
P2 = tf.contrib.layers.flatten(P2)
# FULLY-CONNECTED without non-linear activation function (not not call softmax).
# 6 neurons in output layer. Hint: one of the arguments should be "activation_fn=None"
Z3 = tf.contrib.layers.fully_connected(P2, num_outputs = 6, activation_fn=None)
### END CODE HERE ###
```

compute_cost:

```python
### START CODE HERE ### (1 line of code)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = Z3, labels = Y))
### END CODE HERE ###
```

model:

```python
X, Y = create_placeholders(n_H0,n_W0,n_C0,n_y)
### END CODE HERE ###

# Initialize parameters
### START CODE HERE ### (1 line)
parameters = initialize_parameters()
### END CODE HERE ###

# Forward propagation: Build the forward propagation in the tensorflow graph
### START CODE HERE ### (1 line)
Z3 = forward_propagation(X,parameters)
### END CODE HERE ###

# Cost function: Add cost function to tensorflow graph
### START CODE HERE ### (1 line)
cost = compute_cost(Z3,Y)
### END CODE HERE ###

# Backpropagation: Define the tensorflow optimizer. Use an AdamOptimizer that minimizes
### START CODE HERE ### (1 line)
optimizer =tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(cost)
### END CODE HERE ###

# Initialize all the variables globally
init = tf.global_variables_initializer()
```

```python
# Start the session to compute the tensorflow graph
with tf.Session() as sess:

    # Run the initialization
    sess.run(init)

    # Do the training loop
    for epoch in range(num_epochs):

        minibatch_cost = 0.
        num_minibatches = int(m / minibatch_size) # number of minibatches of size minibatch_size in the train set
        seed = seed + 1
        minibatches = random_mini_batches(X_train, Y_train, minibatch_size, seed)

        for minibatch in minibatches:

            # Select a minibatch
            (minibatch_X, minibatch_Y) = minibatch
            # IMPORTANT: The line that runs the graph on a minibatch.
            # Run the session to execute the optimizer and the cost, the feedict should contain a minibatch for (
            ### START CODE HERE ### (1 line)
            _ , temp_cost =sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})
            ### END CODE HERE ###

            minibatch_cost += temp_cost / num_minibatches


        # Print the cost every epoch
        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)
```

Resnet:

identity_block:

```python
# Second component of main path (≈3 lines)
X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b', kernel_initializer = glorot_
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)

# Third component of main path (≈2 lines)
X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c', kernel_initializer = glorot
X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)


# Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
X = Add()([X, X_shortcut])
X = Activation('relu')(X)

### END CODE HERE ###
```

convolutional_block:

```python
# Second component of main path (≈3 lines)
X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b', kernel_initializer = glorot_
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)

# Third component of main path (≈2 lines)
X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c', kernel_initializer = glorot
X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)



##### SHORTCUT PATH #### (≈2 lines)
X_shortcut = Conv2D(filters = F3, kernel_size = (1, 1), strides = (s,s), padding = 'valid', name = conv_name_base + '1', kernel_initializer
X_shortcut = BatchNormalization(axis = 3, name = bn_name_base + '1')(X_shortcut)

# Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
X = Add()([X, X_shortcut])
X = Activation('relu')(X)

### END CODE HERE ###
```

ResNet50：

```
### START CODE HERE ###

# Stage 3 (≈4 lines)
X = convolutional_block(X, f = 3, filters = [128, 128, 512], stage = 3, block='a', s = 2)
X = identity_block(X, 3, [128, 128, 512], stage=3, block='b')
X = identity_block(X, 3, [128, 128, 512], stage=3, block='c')
X = identity_block(X, 3, [128, 128, 512], stage=3, block='d')


# Stage 4 (≈6 lines)
X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage = 4, block='a', s = 2)
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')


# Stage 5 (≈3 lines)
X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage = 5, block='a', s = 2)
X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')


# AVGPOOL (≈1 line). Use "X = AveragePooling2D(...)(X)"
X = AveragePooling2D(pool_size=(2, 2))(X)

### END CODE HERE ###
```

结论分析与体会：

在 TensorFlow 里面有一些可以直接拿来用的函数可以直接实现要实现的功能，这个在实际应用中会方便很多。

Resnet 的出现，让深度网络模型成为了可能，让深度不再局限于 20 层。

就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1—3 道问答题：

1. 在装 tensorflow 环境的时候，因为没有考虑到 python 和 tensorflow 之间的版本依赖问题，导致配的环境有问题，有些 tensorflow1 的函数在 tensorflow2 中不能运行（因为已经被删了）。