Bottleneck

Understand how bottleneck blocks decrease memory usage for large ResNet models.

Chapter Goals:

- Learn about the bottleneck block and why it's used
- Implement a function for a bottleneck block

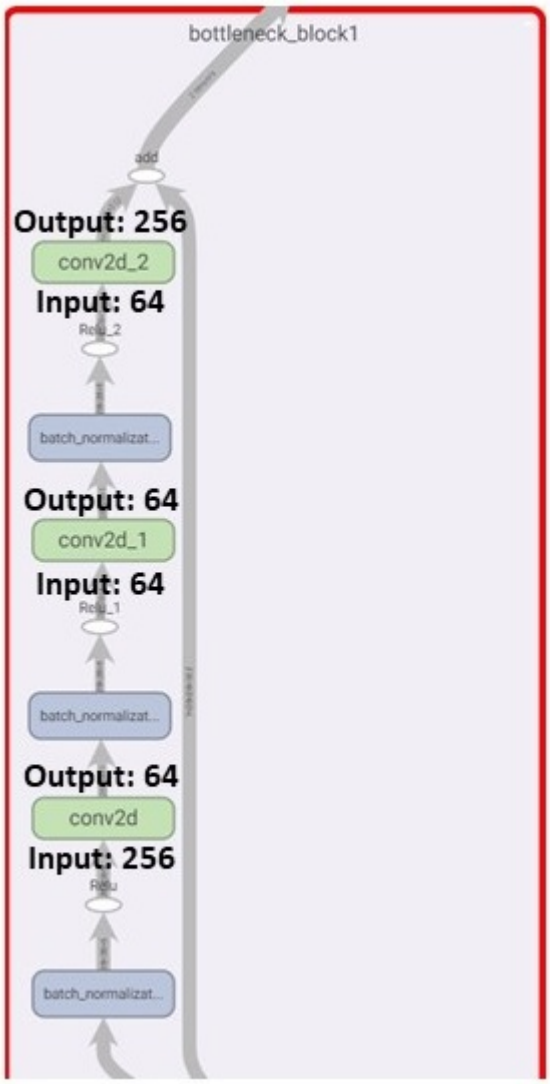
A. Size considerations

The ResNet block we described in the previous chapter is the main building block for models with fewer than 50 layers. Once we hit 50 or more layers, we want to take advantage of the large model depth, and utilize more filters in each convolution layer. However, using more filters results in added weight parameters, which can lead to incredibly long training time.

To counter this, ResNet incorporates the same squeeze and expand concept used by the SqueezeNet fire module. The ResNet blocks for 50+ layer models will now use 3 convolution layers rather than 2, where the first convolution layer squeezes the number of channels in the data and the third convolution layer expands the number of channels. We refer to these blocks as *bottleneck* blocks.

B. Bottleneck block

The third convolution layer of a bottleneck block uses four times as many filters as a regular ResNet block. This means that the input to bottleneck blocks will have four times as many channels (remember that the output of one block is the input to the next). Hence, the first convolution layer acts as a squeeze layer, to reduce the number of channels back to the regular amount.



Example: A bottleneck block (with a shortcut), for input data with 256 channels. The number of channels for the input and output of each convolution layer are labeled in the diagram.

Another similarity to the SqueezeNet fire module is the mixed usage of 1x1 and 3x3 kernels. The bottleneck block uses 1x1 kernels for the first and third convolution layers, while the middle convolution layer still uses 3x3 kernels. This helps reduce the number of weight parameters while still maintaining good performance.

C. Parameter comparison

In the SqueezeNet Lab, we introduced the equation to calculate the number of weight parameters in a convolution layer:

$$P = H_K \times W_K \times F \times C + F$$

where $H_K \times W_K$ is the kernel dimensions, F is the number of filters, and C is the number of input channels.

For a regular ResNet block with 64 filters that takes in an input with 64 channels, the number of weight parameters, (P), used in the block is

$$P_1 = P_2 = 3 \times 3 \times 64 \times 64 + 64 = 36{,}928$$

$$\mathbf{P = P_1 + P_2 = 73{,}856}$$

where P_1 and P_2 represent the number of weight parameters used in the first and second convolution layers, respectively.

For a bottleneck block with 64 filters, the input will have 256 channels. Therefore, the number of weight parameters, P_B, used in the block is

$$P_1 = 1 \times 1 \times 64 \times 256 + 64 = 16{,}448$$

$$P_2 = 3 \times 3 \times 64 \times 64 + 64 = 36{,}928$$

$$P_3 = 1 \times 1 \times 256 \times 64 + 256 = 16{,}640$$

$$\mathbf{P_B = P_1 + P_2 + P_3 = 70{,}016}$$

where P_1, P_2, and P_3 represent the number of weight parameters used in the first, second, and third convolution layers, respectively.

So despite the input and output having four times as many filters, the bottleneck block actually uses fewer weight parameters. This allows us to create very deep ResNet models (e.g. 200 layers) and still be able to train them in a reasonable amount of time.

Time to Code!

In this chapter, you'll complete the `bottleneck_block` function (line **63**), which creates a ResNet bottleneck block.

The function creates the bottleneck block inside a *variable scope* (`with` block), which helps organize the model (similar to how a directory organizes files). Inside the variable scope, code has already been filled in to create the first two convolution layers and shortcut.

Your task is to create the third convolution layer, then return the output of the building block.

Set `pre_activated3` equal to `self.pre_activation` with `conv2` as the first argument and `is_training` as the second argument.

We use `pre_activated3` as the input to the third convolution layer. The convolution layer will use four times as many filters as the previous two convolution layers, and the same kernel size and stride size as the first convolution layer.

Set `conv3` equal to `self.custom_conv2d` applied with the specified input arguments.

The output of the building block is the sum of the shortcut and the output of the third convolution layer.

Return the sum of `conv3` and `shortcut`, still inside the `with` block.

```
import tensorflow as tf

# block_layer_sizes loaded in backend

class ResNetModel(object):
    # Model Initialization
    def __init__(self, min_aspect_dim, resize_dim, num_layers, output_size,
        data_format='channels_last'):
        self.min_aspect_dim = min_aspect_dim
        self.resize_dim = resize_dim
        self.filters_initial = 64
        self.block_strides = [1, 2, 2, 2]
        self.data_format = data_format
        self.output_size = output_size
        self.block_layer_sizes = block_layer_sizes[num_layers]
        self.bottleneck = num_layers >= 50

    # Applies consistent padding to the inputs
    def custom_padding(self, inputs, kernel_size):
        pad_total = kernel_size - 1
        pad_before = pad_total // 2
        pad_after = pad_total - pad_before
        if self.data_format == 'channels_first':
            padded_inputs = tf.pad(
                inputs,
                [[0, 0], [0, 0], [pad_before, pad_after], [pad_before, pad_after]])
        else:
            padded_inputs = tf.pad(
                inputs,
                [[0, 0], [pad_before, pad_after], [pad_before, pad_after], [0, 0]])
        return padded_inputs

    # Customized convolution layer w/ consistent padding
    def custom_conv2d(self, inputs, filters, kernel_size, strides, name=None):
        if strides > 1:
            padding = 'valid'
            inputs = self.custom_padding(inputs, kernel_size)
        else:
            padding = 'same'
```

```
        return tf.layers.conv2d(
            inputs=inputs, filters=filters, kernel_size=kernel_size,
            strides=strides, padding=padding, data_format=self.data_format,
            name=name)

    # Apply pre-activation to input data
    def pre_activation(self, inputs, is_training):
        axis = 1 if self.data_format == 'channels_first' else 3
        bn_inputs = tf.layers.batch_normalization(inputs, axis=axis, training=is_training)
        pre_activated_inputs = tf.nn.relu(bn_inputs)
        return pre_activated_inputs

    # Returns pre-activated inputs and the shortcut
    def pre_activation_with_shortcut(self, inputs, is_training, shortcut_params):
        pre_activated_inputs = self.pre_activation(inputs, is_training)
        shortcut = inputs
        shortcut_filters = shortcut_params[0]
        if shortcut_filters is not None:
            strides = shortcut_params[1]
            shortcut = self.custom_conv2d(pre_activated_inputs, shortcut_filters, 1, strides)
        return pre_activated_inputs, shortcut

    # ResNet bottleneck block
    def bottleneck_block(self, inputs, filters, strides, is_training, index, shortcut_filters
        with tf.variable_scope('bottleneck_block{}'.format(index)):
            shortcut_params = (shortcut_filters, strides)
            pre_activated1, shortcut = self.pre_activation_with_shortcut(inputs, is_training,
            conv1 = self.custom_conv2d(pre_activated1, filters, 1, 1)
            pre_activated2 = self.pre_activation(conv1, is_training)
            conv2 = self.custom_conv2d(pre_activated2, filters, 3, strides)
            # CODE HERE
```