Logits

Use global average pooling to obtain the model's logits.

Chapter Goals:

- Learn about global average pooling and its benefits
- Obtain the per class model logits

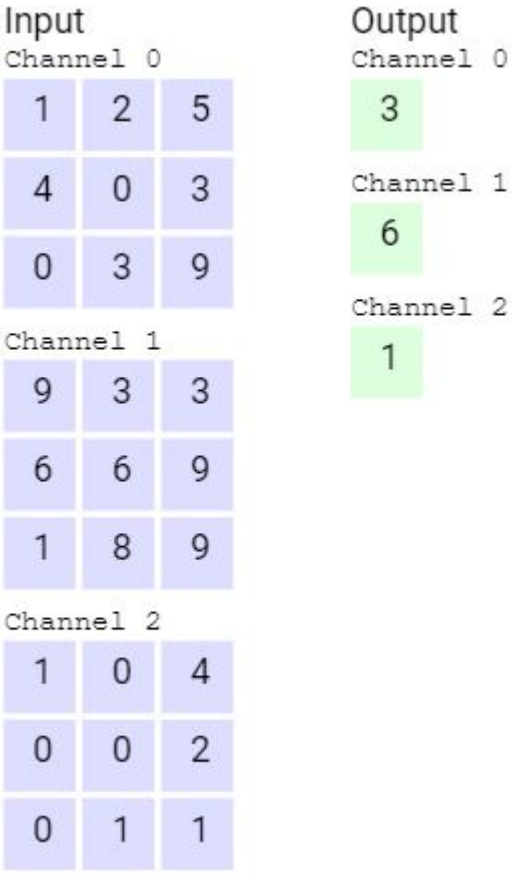
A. Channel-based logits

In the **CNN** section, we applied flattening and a couple of fully-connected layers (with dropout) to obtain our model's logits. However, rather than flattening our data, we can instead use a convolution layer to convert the number of channels in our data to the number of image classes (categories). Then, we apply a special type of average pooling layer to obtain logits for each channel (i.e. image class).

B. Global average pooling

Average pooling follows the exact same concept as max pooling, with the only difference being that the filter performs an *average* rather than a max operation. When the average pooling filter is the same height and width as the input data (i.e. the output is a single average per channel), this is known as *global average pooling*.

We use global average pooling to obtain our per class logits. Each channel corresponds to a unique class, and the channel's averaged value represents the logit for that class.



Global average pooling for an input with 3 channels. Note that our model's average pooling layer operates on 10 channels.

C. Advantages

There are a couple of advantages to using global average pooling rather than fully-connected layers. The first is that global average pooling is more native to the CNN structure, by obtaining logits via channels rather than converting the data to flattened vectors. This allows the CNN to obtain more accurate logits for each image class.

Second, since global average pooling is just a pooling layer, it uses no parameters. This means there is no risk of overfitting at the global average pooling layer. In contrast, fully-connected layers use many weight parameters. Although dropout can help diminish the problem, the risk of overfitting still remains.

Time to Code!

In this chapter you'll be completing the `get_logits` function, which is used at the end of `model_layers` to convert the final convolution layer to logits.

The input to the function, `conv_layer` is a convolution layer applied to the dropout from the previous chapter.

We then use an average pooling layer for global average pooling of `conv_layer`. In TensorFlow, an average pooling layer is applied through the `tf.layers.average_pooling2d` function. The function takes in the exact same arguments as `tf.layers.max_pooling2d`.

For global average pooling, we want the filter size to be the same height and width as the input data. Since the data is in NHWC format, we set the pooling filter size to `[conv_layer.shape[1], conv_layer.shape[2]]`, which represent the H (height) and W (width) dimensions of the data. We use a stride size of `1` because the filter has the same height/width as the data.

Set `avg_pool1` equal to `tf.layers.average_pooling2d` applied with `conv_layer` as the first argument, the filter size as the second argument, and the stride size as the third argument.

The output of global average pooling is still in NHWC format, with height and width of size 1. Since we want the `logits` to be vectors of size `self.output_size`, we need to flatten the data. To do this, we'll use TensorFlow's built-in `tf.layers.flatten` function.

The required argument of `tf.layers.flatten` is an input tensor where the first dimension represents the batch size. The output of `tf.layers.flatten` is the flattened data across the batch. In terms of NHWC, the output has shape `(N, H*W*C)`. The function also takes in a keyword argument for `name`.

Set `logits` equal to `tf.layers.flatten` applied with `avg_pool1` as the first argument and `'logits'` as the `name` argument.
Then return `logits`.

```
import tensorflow as tf

class SqueezeNetModel(object):
    # Model Initialization
    def __init__(self, original_dim, resize_dim, output_size):
        self.original_dim = original_dim
        self.resize_dim = resize_dim
        self.output_size = output_size

    # Convert final convolution layer to logits
    def get_logits(self, conv_layer):
        # CODE HERE
        pass
```

```
    # Convolution layer wrapper
    def custom_conv2d(self, inputs, filters, kernel_size, name):

        return tf.layers.conv2d(
            inputs=inputs,
            filters=filters,
            kernel_size=kernel_size,
            activation=tf.nn.relu,
            padding='same',
            name=name)

    # Max pooling layer wrapper
    def custom_max_pooling2d(self, inputs, name):
        return tf.layers.max_pooling2d(
            inputs=inputs,
            pool_size=[2, 2],
            strides=2,
            name=name)

    # SqueezeNet fire module
    def fire_module(self, inputs, squeeze_depth, expand_depth, name):
        with tf.variable_scope(name):
            squeezed_inputs = self.custom_conv2d(
                inputs,
                squeeze_depth,
                [1, 1],
                'squeeze')
            expand1x1 = self.custom_conv2d(
                squeezed_inputs,
                expand_depth,
                [1, 1],
                'expand1x1')
            expand3x3 = self.custom_conv2d(
                squeezed_inputs,
                expand_depth,
                [3, 3],
                'expand3x3')
            return tf.concat([expand1x1, expand3x3], axis=-1)

    # Utility function for multiple fire modules
    def multi_fire_module(self, layer, params_list):
        for params in params_list:
            layer = self.fire_module(
                layer,
                params[0],
                params[1],
                params[2])
        return layer

    # Model Layers
    # inputs: [batch_size, resize_dim, resize_dim, 3]
    def model_layers(self, inputs, is_training):
        conv1 = self.custom_conv2d(
            inputs,
            64,
            [3, 3],
            'conv1')
        pool1 = self.custom_max_pooling2d(
            conv1,
            'pool1')
        fire_params1 = [
            (32, 64, 'fire1'),
```

```
        (32, 64, 'fire2')
    ]
    multi_fire1 = self.multi_fire_module(
        pool1,
        fire_params1)
    pool2 = self.custom_max_pooling2d(
        multi_fire1,
        'pool2')
    fire_params2 = [
        (32, 128, 'fire3'),
        (32, 128, 'fire4')
    ]
    multi_fire2 = self.multi_fire_module(
        pool2,
        fire_params2)
    dropout1 = tf.layers.dropout(multi_fire2, rate=0.5,
        training=is_training)
    final_conv_layer = self.custom_conv2d(
        dropout1,
        self.output_size,
        [1, 1],
        'final_conv')
    return self.get_logits(final_conv_layer)
```