ResNet Block

Learn about the main building block of ResNet and how it works.

Chapter Goals:

- Understand the purpose of residual learning
- Learn the structure of a ResNet building block
- Create a function for a ResNet building block

A. Learning the identity

In the previous chapter, we touched upon *residual learning*, i.e. the process of a block learning the *residual function*:

$$\mathcal{F}_B(\mathbf{x}) = \mathcal{H}_B(\mathbf{x}) - \mathbf{x}$$

where $\mathcal{H}_B(\mathbf{x})$ is the optimal mapping function.

You might ask why we use residual learning, since the residual function is pretty similar to the optimal function. The reason is because it is much easier for a block to learn a *zero mapping* than an identity mapping.

A zero mapping simply returns an output of all 0's. This is very easy to learn; the block can just be trained so that all its weights are 0. However, if the block learns a zero mapping, i.e. $\mathcal{F}_B(\mathbf{x}) = \mathbf{0}$, then the modified output (shortcut added) becomes

$$\mathcal{F}_B(\mathbf{x}) + \mathbf{x} = \mathbf{0} + \mathbf{x} = \mathbf{x}$$

That's exactly the identity mapping! So in learning the easy zero mapping, we've created a block that corresponds to an identity mapping, which is a much harder task.

B. Improving performance

We can now add many layers to a model and still avoid degradation, by having the additional layers represent identity mappings (which is possible

due to residual learning). What's more amazing, though, is that residual learning actually *increases* model performance when adding many layers.

Let's say we have a ResNet model with 20 blocks, which already has really good performance. We'll add 5 more blocks to the model and train it with residual learning. If the first 20 blocks of our larger model learn the same weights as the smaller model, the 5 additional blocks don't need to modify the output of the first 20 blocks by much, since the smaller model was already really good.

In other words, we want each of our 5 additional blocks to learn a mapping function relatively close to the identity mapping. That's exactly what residual learning is designed to do, i.e. allow additional blocks to make small tweaks to improve model performance.

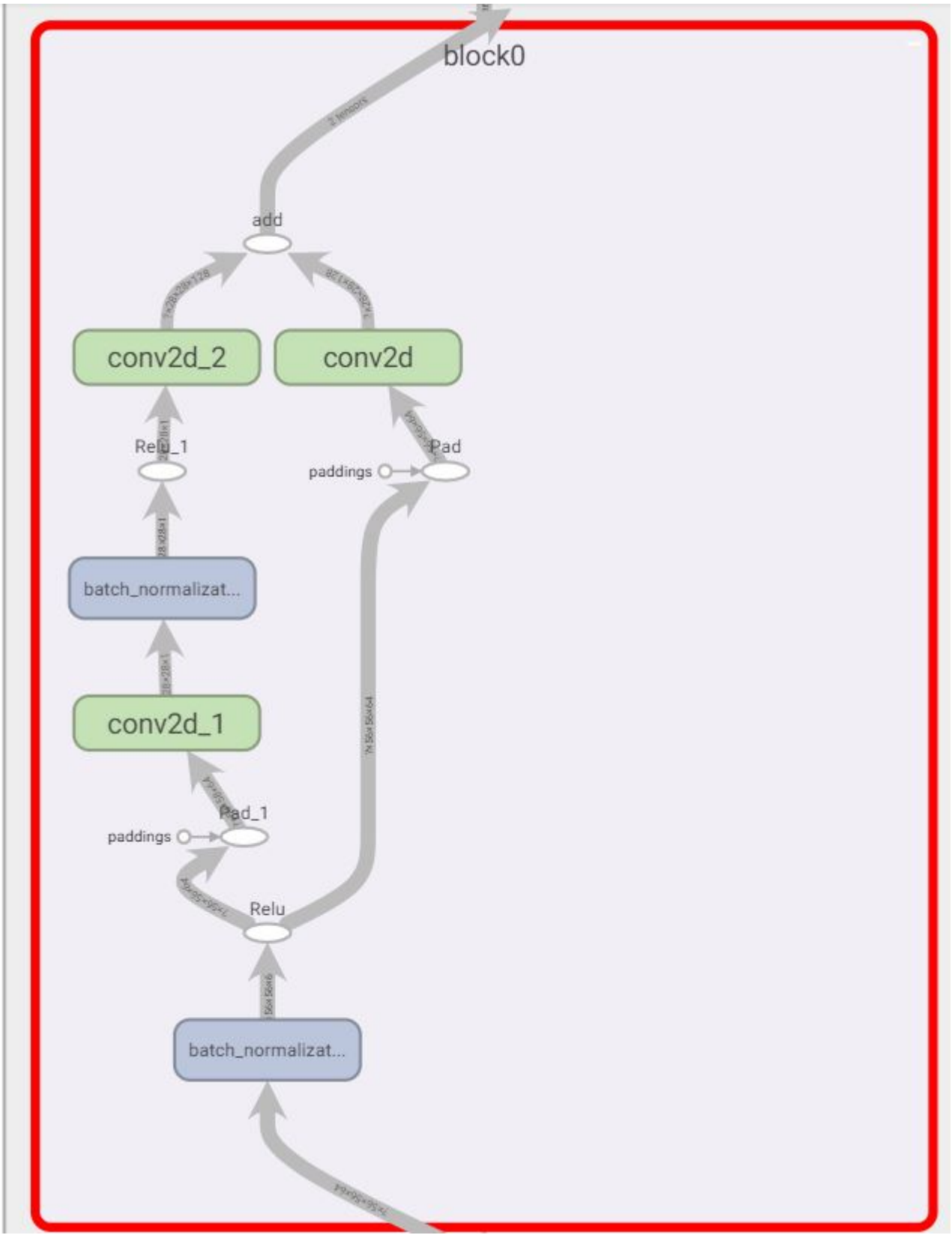
C. Structure of a block

We've talked a lot about blocks in the context of ResNet and residual learning. Now it's time to get into the actual structure of a block.

As mentioned in the previous chapter, a block is made up of multiple convolution layers. Specifically, it uses two convolution layers, both with 3x3 kernels and pre-activated inputs.

The second convolution layer always uses stride size of 1, so the input and output have the same height and width dimensions. The first convolution layer will also mainly use a stride size of 1. The only times it uses a stride size of 2 is for dimension reduction purposes.

Since a ResNet block uses residual learning, the shortcut is added to the block's residual output, $\mathcal{F}_B(\mathbf{x})$, to produce the overall output.



Structure of a ResNet block, with a projection shortcut. The left arrow path represents the block's structure, while the right arrow path represents the projection shortcut.

Time to Code!

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

The function creates the building 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 convolution layer and shortcut.

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

Set `pre_activated2` equal to `self.pre_activation` with `conv1` as the first argument and `is_training` as the second argument.

We use `pre_activated2` as the input to the second convolution layer. The second convolution layer will also use `filters` for the number of filters and a `kernel_size` of 3. This time, the stride size will be 1.

Set `conv2` equal to `self.custom_conv2d` with the specified input arguments.

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

Return the sum of `conv2` 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 building block
    def regular_block(self, inputs, filters, strides, is_training, index, shortcut_filters=No
        with tf.variable_scope('regular_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, 3, strides)
            # CODE HERE
```