Shortcut

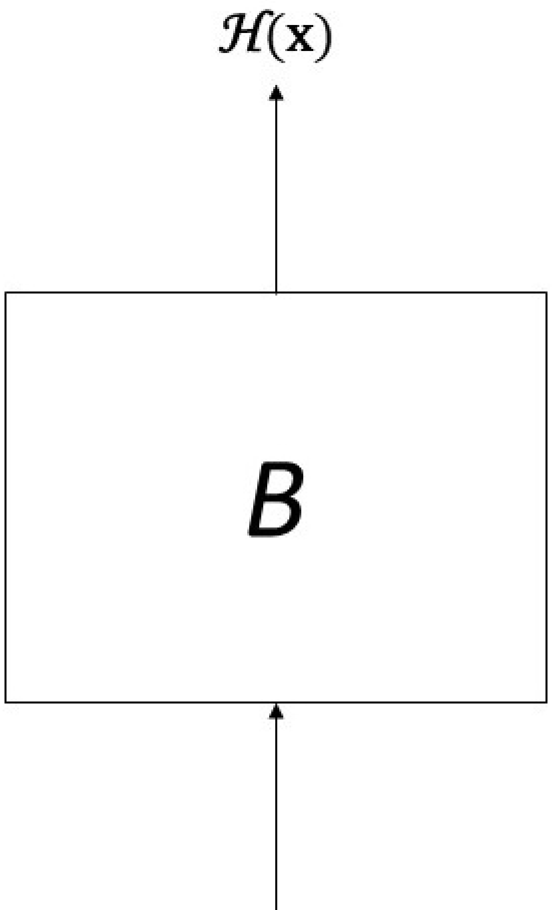
Understand how shortcuts can improve performance in large models.

Chapter Goals:

- Learn about mapping functions and identity mapping
- Understand the purpose of a shortcut for residual learning
- Implement a wrapper for the pre-activation function that also returns the shortcut

A. Mapping functions

To understand the intuition behind ResNet, we need to discuss its building blocks. Each ResNet building block takes in an input, $\mathbf{x}$, and produces some output $\mathcal{H}(\mathbf{x})$, where $\mathcal{H}$ represents the block's *mapping function*. The mapping function, $\mathcal{H}$, is a mathematical representation of the block itself; it takes in an input and, using the weights within the block, produces an output.



The image above shows the input and output for block B with mapping function $\mathcal{H}$.

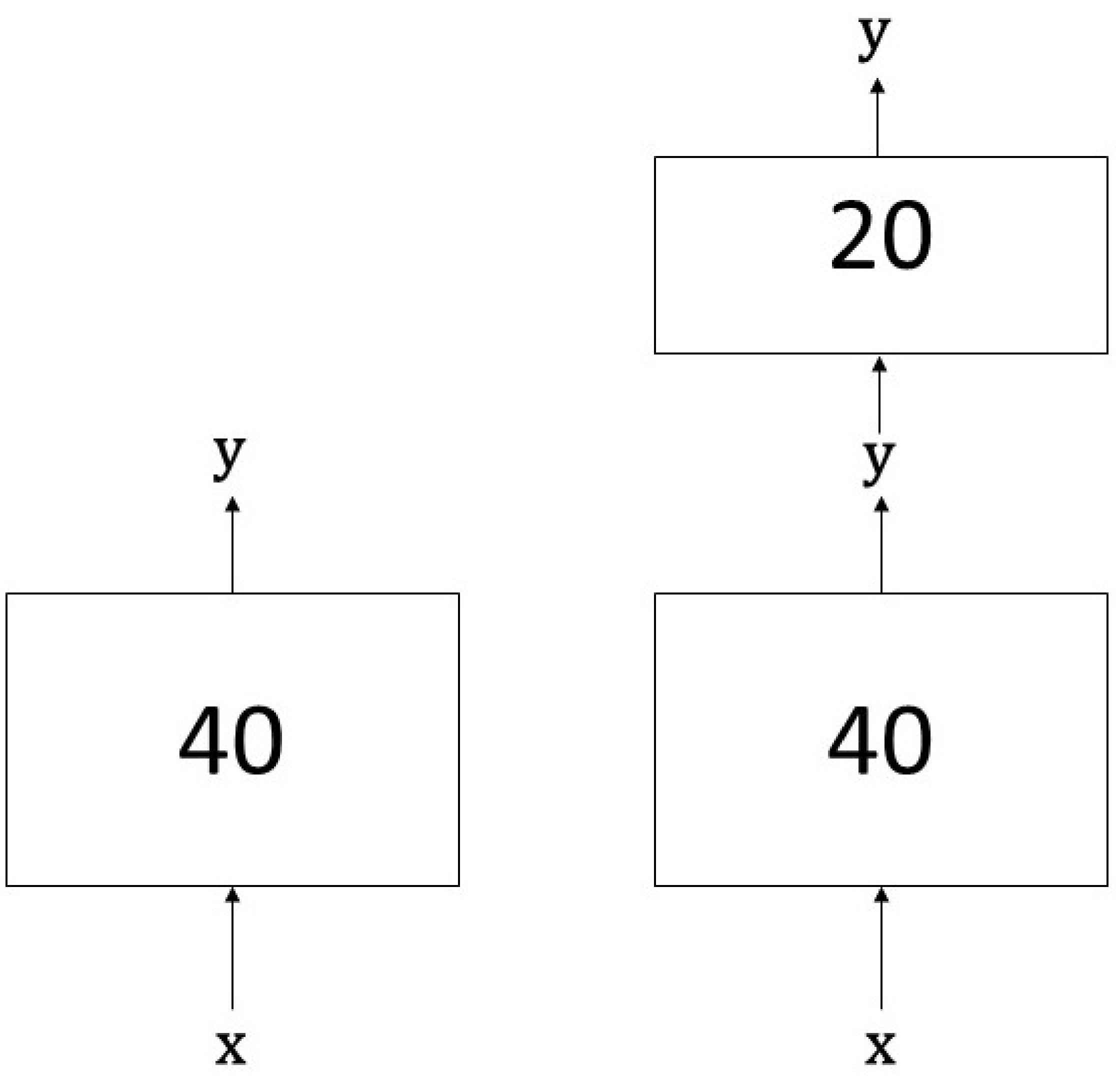
For now, it suffices to know that a block is just a stack of convolution layers (we’ll discuss the inner workings of a block in the next two chapters).

B. Identity mapping

We've previously discussed the issue of *degradation*, where a model's performance decreases after a certain number of layers are added. Despite its effects, degradation is actually a pretty counter-intuitive problem.

Let's say our model performs well at 40 convolution layers, so we want to add 20 more convolution layers (60 layers total). There is a simple way for the larger model to achieve the same performance as the smaller model: use the same weights as the smaller model for the first 40 layers, then make the last 20 convolution layers an *identity mapping*.

An identity mapping just means the output for a layer (or set of layers, e.g. a ResNet block) is the same as the input. So if the larger model used the same weights as the smaller model, followed by an identity mapping, its outputs would be identical to those of the smaller model.



The image above shows a 40 layer model (left) takes in input $\mathbf{x}$ and produces $\mathbf{y}$. 60 layer model (right) also produces $\mathbf{y}$ by making its last 20 layers an identity mapping.

However, for some reason the larger model is worse, despite having a simple solution to at least match the smaller model's performance. It turns out that training a model's layers to learn identity mappings is actually pretty difficult.

C. Residual learning

Let's say for some block B, its optimal mapping function is $\mathcal{H}_B$. If the optimal function is close to an identity mapping (i.e. $\mathcal{H}_B(\mathbf{x}) = \mathbf{x}$), learning the function will be a hard task. Instead, we can have the block learn a different mapping function, $\mathcal{F}_B$, such that

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

We also add the input, $\mathbf{x}$, to the block's output. This is referred to as a *shortcut* *connection*, or just *shortcut*. So the modified block's output will now be

$\mathcal{F}_B(\mathbf{x}) + \mathbf{x}$, which is equivalent to the optimal function, $\mathcal{H}_B(\mathbf{x})$. The process of learning $\mathcal{F}_B(\mathbf{x})$ is known as *residual learning*. When the optimal mapping function, $\mathcal{H}_B(\mathbf{x})$, is close to the identity mapping, it is much easier for a block to learn $\mathcal{F}_B(\mathbf{x})$ than $\mathcal{H}_B(\mathbf{x})$. In the next chapter, we'll explain why.

D. Projection shortcut

When a block uses a stride size greater than 1 for its convolution layers, its output will have reduced height and width compared to its input. Since the shortcut is the same as the input data, it cannot be added to the block output (addition requires equal dimension sizes). In this case, we use a *projection shortcut*.

A projection shortcut is the result of applying a convolution layer, with 1x1 kernels, to the pre-activated input data. This convolution layer ensures that the shortcut has the same dimensions as the block's output, by using the same stride size and number of filters.

We apply the convolution layer to the pre-activated input data, rather than directly to the input data, for consistency with the rest of the block (where each convolution layer is preceded by pre-activation).

Time to Code!

In this chapter you'll complete the `pre_activation_with_shortcut` function (line **53**), which applies pre-activation to input data and also returns a shortcut.

The function already has the pre-activated input and shortcut configuration set up. Your task is to write code inside the `if` block, which sets the shortcut to a convolution layer on top of the pre-activated inputs.

The stride length for the convolution layer is given in `shortcut_params[1]`.

Set `strides` equal to the second element of `shortcut_params`.

The convolution layer will use 1x1 kernels, so we can just set `kernel_size` to 1 (single integer means same width and height for the kernel).

Set `shortcut` equal to `self.custom_conv2d` with `pre_activated_inputs` as the first argument, `shortcut_filters` as the number of filters, a kernel size of `1`, and `strides` as the stride size.

```
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:
            # CODE HERE

            pass
        return pre_activated_inputs, shortcut
```