Pre-activation

Learn why batch normalization and pre-activation are important in ResNet.

Chapter Goals:

- Learn about internal covariate shift and batch normalization
- Understand the difference between pre-activation and post-activation
- Implement the pre-activation function

A. Internal covariate shift

When training models with many weight layers, e.g. ResNet, a problem known as *internal covariate shift* can occur. To understand this problem, we first define a related problem known as *covariate shift*.

A covariate shift occurs when the input data's *distribution* changes and the model cannot handle the change properly. An example would be if a model was trained to classify between different dog breeds, with a training set of only images of brown dogs. Then if we test the model on images of yellow dogs, the performance may not be as good as we expected.

In this case, the model's original input distribution was limited to just brown dogs, and changing the input distribution to a different color of dogs introduced covariate shift.

An internal covariate shift is essentially just a covariate shift that happens between layers of a model. Since the input of one layer is the output of the previous layer, the input distribution for a layer is the same as the output distribution of the previous layer.

Because the output distribution of a layer depends on its weights, and the weights of a model are constantly being updated, each layer's output distribution will constantly change (though by incremental amounts). In a model without many layers, the incremental changes in layer distributions don't really have much impact. However, in models with many layers these incremental changes will eventually add up, and lead to internal covariate shift at deeper layers.

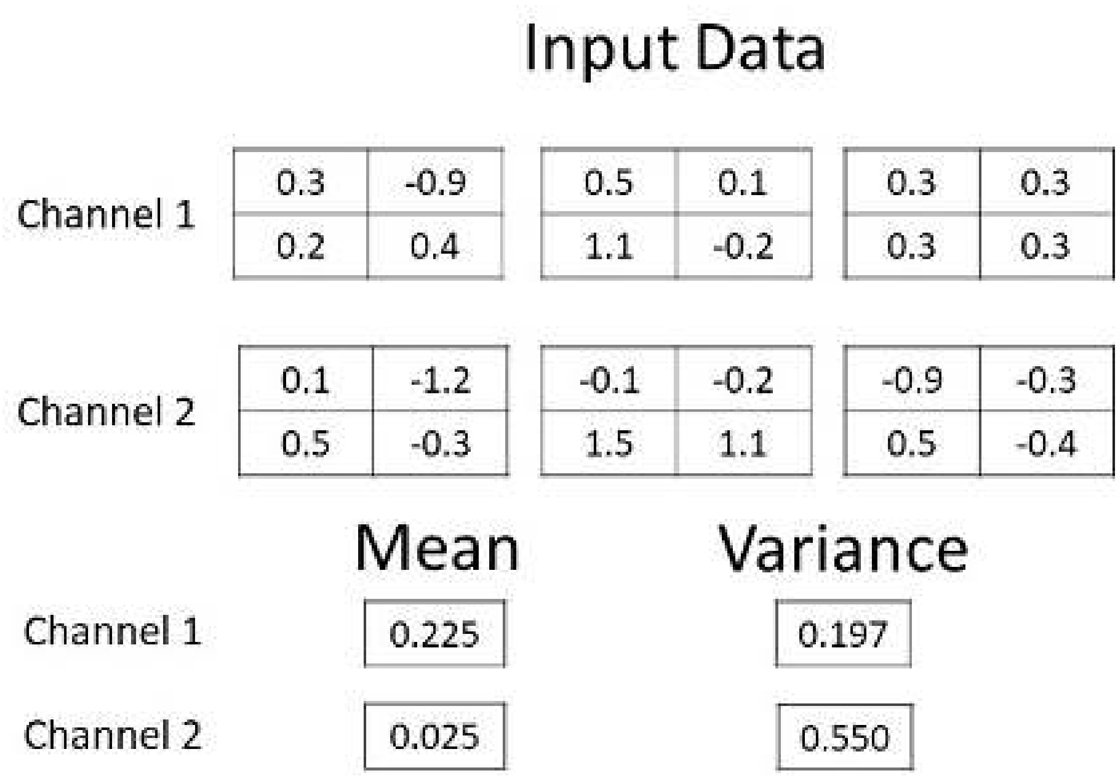
B. Batch normalization

The solution to internal covariate shift is batch normalization. Since internal covariate shift is caused by distribution changes between layers, we can remedy this by enforcing a fixed distribution to the inputs of each layer.

Batch normalization accomplishes this by subtracting the mean from the inputs and dividing by the standard deviation (i.e. square-root variance). This results in a standardized distribution (i.e. mean of 0 and variance of 1).

One thing to note is that batch normalization is applied across a specific dimension of the data. For CNNs, we apply it across the channels dimension, meaning we standardize the values for each channel of the input data. So the mean and variance are actually vectors of `num_channels` values.

Below we show standardization of an input with 2 channels and a batch size of 3.

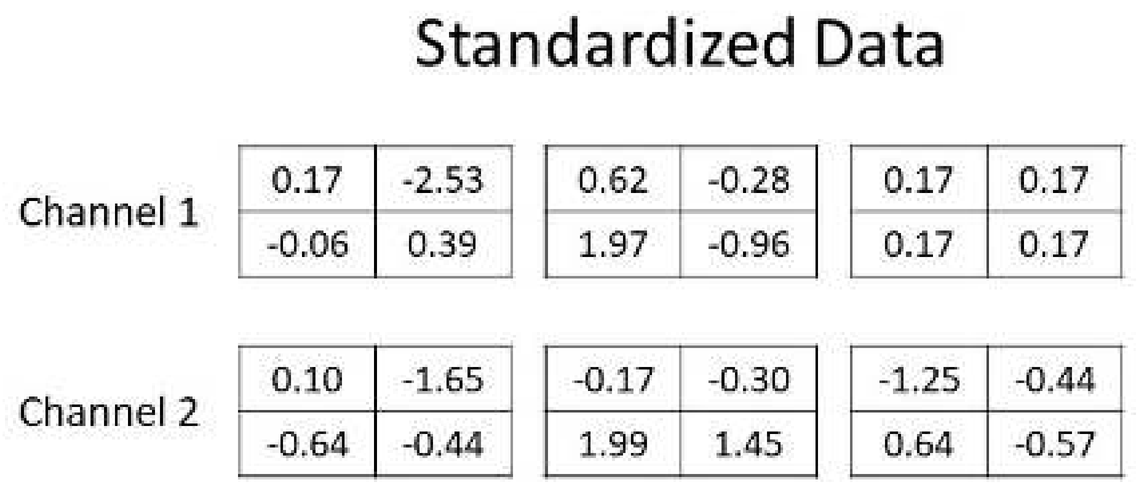


Input Data

Channel 1	0.3	-0.9	0.5	0.1	0.3	0.3
	0.2	0.4	1.1	-0.2	0.3	0.3
Channel 2	0.1	-1.2	-0.1	-0.2	-0.9	-0.3
	0.5	-0.3	1.5	1.1	0.5	-0.4

	Mean	Variance
Channel 1	0.225	0.197
Channel 2	0.025	0.550

Input data (mean and variance per channel are shown)



Standardized Data

Channel 1	0.17	-2.53	0.62	-0.28	0.17	0.17
	-0.06	0.39	1.97	-0.96	0.17	0.17
Channel 2	0.10	-1.65	-0.17	-0.30	-1.25	-0.44
	-0.64	-0.44	1.99	1.45	0.64	-0.57

Standardized data (for the data shown above)

For some layers we might not want a standardized distribution of the inputs. Maybe we want a distribution with a different mean or variance. Luckily, batch normalization has two trainable variables, γ and β, that allow us to change the variance and mean, respectively, of the distribution.

$$\text{BN}(\mathbf{x}) = \gamma * \hat{\mathbf{x}} + \beta$$

The above formula represents batch normalization (BN) applied to input data $\mathbf{x}$. Note that $\hat{\mathbf{x}}$ represents the standardized input data.

Since γ and β are trainable variables, the model will automatically fine-tune their values for each batch normalization layer. In fact, if γ is set to the original standard deviation and β is set to the original mean of the inputs, batch normalization will produce the original input data values.

C. Moving average

During training, batch normalization calculates the mean and variance of its inputs at every training step. To obtain the best possible results during evaluation, we maintain an exponential moving average of the mean and variance vectors for each batch normalization layer. So if our model has m batch normalization layers, we maintain the mean and variance per batch normalization layer, for a total of m means and m variances.

During evaluation, we use the average mean and variance for each batch normalization layer rather than the input data's mean and variance. The average mean and variance provide a better estimate for the overall dataset, which helps our model's accuracy.

At the bottom of the `run_model_setup` function, you can see that we added in some code involving an `update_ops` variable. The code tells our model to update the moving average and variance for each batch normalization layer after every training step.

D. Pre-activation

When we use batch normalization, we apply it right before an activation function, e.g. ReLU. Normally, the activation function in CNNs comes after each convolution layer. This is known as *post-activation*. Version 1 of ResNet used the traditional method of post-activation, meaning its convolution layers would be followed by a batch normalization, which would then be followed by the ReLU activation.

However, Version 2 of ResNet switched up the ordering so that the batch normalization and activation would now come *before* the convolution layer. This is known as *pre-activation*. Pre-activation has been shown to work better with ResNet, which is why we are using it in our code.

You may have noticed that the `custom_conv2d` function didn't use the `activation` keyword when setting `tf.layers.conv2d`. This was purposely done so we could manually apply ReLU activation prior to the convolution layer, rather than letting TensorFlow apply it right after.

Time to Code!

In this chapter you'll be completing the `pre_activation` function, which applies the pre-activation described above.

First we'll set the channels axis (dimension) depending on the data format.

Set `axis` equal to `1` if `self.data_format` is `'channels_first'`, otherwise set it equal to `3`.

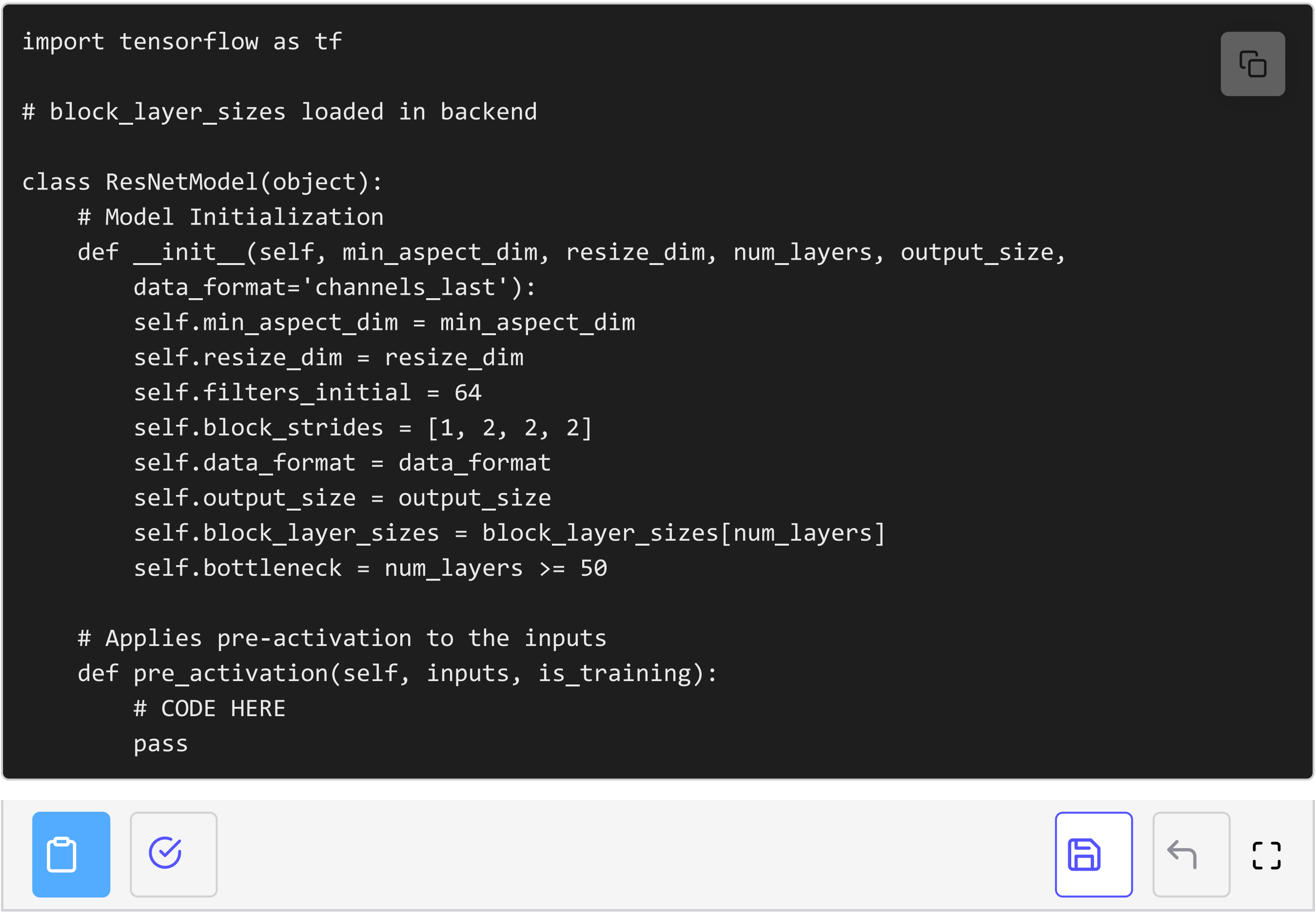
The first step to pre-activation is applying batch normalization to the input data. In TensorFlow, the function we use for batch normalization is `tf.layers.batch_normalization`. The only required argument for the function is the input data. However, there are a ton of keyword arguments, most of which will rarely be changed from their default value.

The keyword arguments that we change are `axis` and `training`. The `axis` argument represents the axis (dimension) we apply batch normalization to. In our code, we apply it to the channels dimension. The `training` argument represents whether the model is training or not, since batch normalization is applied differently for training and evaluation.

Set `bn_inputs` equal to `tf.layers.batch_normalization` with `inputs` as the first argument, along with keyword arguments `axis=axis` and `training=is_training`.

We finish the pre-activation by applying the ReLU function to our batch normalized input data.

Set `pre_activated_inputs` equal to `tf.nn.relu` applied to `bn_inputs`. Then return `pre_activated_inputs`.



```
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 pre-activation to the inputs
    def pre_activation(self, inputs, is_training):
        # CODE HERE
        pass
```