# Dropout

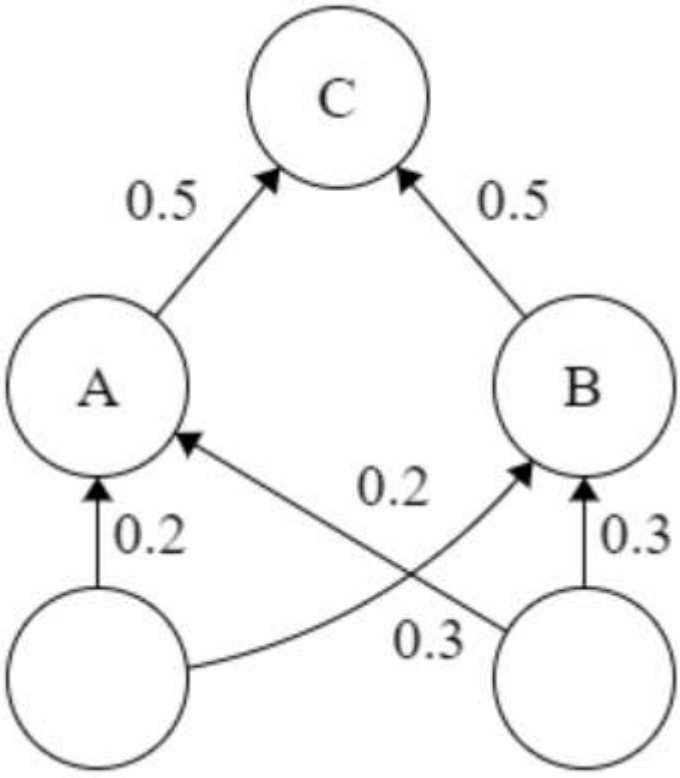
Learn about dropout and how it can reduce overfitting in large neural networks.

Chapter Goals:

- Understand why we use dropout in neural networks
- Apply dropout to a fully-connected layer

## A. Co-adaptation

Co-adaptation refers to when multiple neurons in a layer extract the same, or very similar, hidden features from the input data. This can happen when the connection weights for two different neurons are nearly identical.



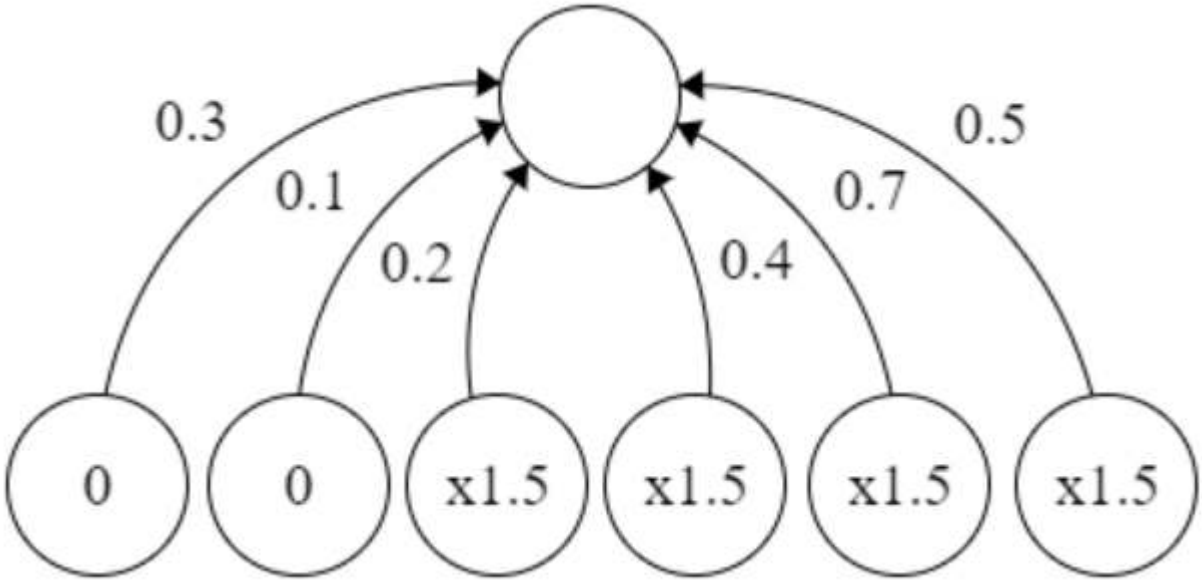

An example of co-adaptation between neurons A and B. Due to identical weights, A and B will pass the same value into C.

When a fully-connected layer has a large number of neurons, co-adaptation is more likely to occur. This can be a problem for two reasons. First, it is a waste of computation when we have redundant neurons computing the same

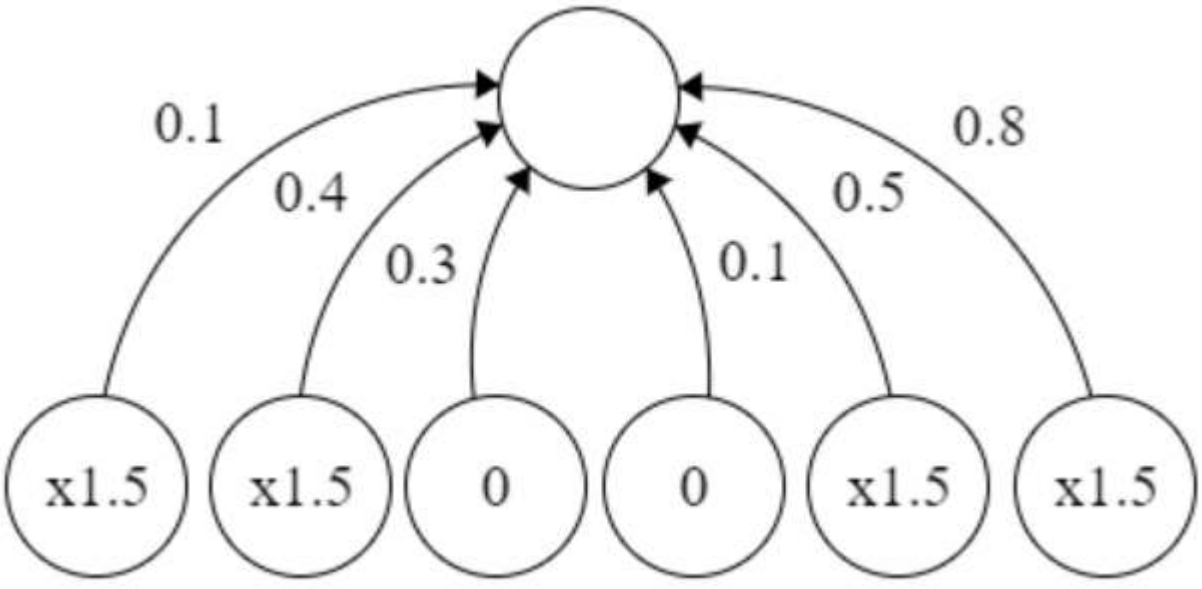
output. Second, if many neurons are extracting the same features, it adds more significance to those features for our model. This leads to overfitting if the duplicate extracted features are specific to only the training set.

### B. Dropout

The way we minimize co-adaptation for fully-connected layers with many neurons is by applying *dropout* during training. In dropout, we randomly shut down some fraction of a layer's neurons at each training step by zeroing out the neuron values. The fraction of neurons to be zero'd out is known as the dropout rate, $r_d$. The remaining neurons have their values multiplied by $\frac{1}{1-r_d}$ so that the overall sum of the neuron values remains the same.




Training Step 1




Training Step 2

Training Step 2

The two images represent dropout applied to a layer of 6 units, shown at multiple training steps. The dropout rate is 1/3, and the remaining 4 neurons at each training step have their value scaled by x1.5.

By randomly dropping a fraction of the neurons, we are essentially choosing a random sample of the neurons to use at each training step. Therefore, each individual neuron works with many different subsets of the other neurons rather than all of them at once. This helps each neuron avoid over-relying on other neurons to correct its mistakes (the underlying cause of co-adaptation), while still allowing the neurons to learn different things from one another.

## Time to Code!

In this chapter, we'll create the `apply_dropout` function, which applies dropout to the `dense` fully-connected layer, with a dropout rate of 0.4. In TensorFlow, dropout is implemented via the `tf.layers.dropout` function. The function takes in a required argument of an input tensor, which in this case will be `dense`.

Some important keyword arguments are:

- `rate`: The dropout rate, default is `0.5`.
- `training`: Whether or not the model is in training mode, default is `False`. We only apply dropout during training.

We use the parameter `is_training` to see whether or not we need to apply dropout.

**Set `dropout` equal to `tf.layers.dropout` applied with required argument `dense`. We'll set the `rate` argument to `0.4` and `training` argument to `is_training`.**
**Then return `dropout`.**

```
import tensorflow as tf

class MNISTModel(object):
    # Model Initialization
    def __init__(self, input_dim, output_size):
        self.input_dim = input_dim
        self.output_size = output_size
```

```
    # Apply dropout to final layer
    def apply_dropout(self, dense, is_training):

        # CODE HERE
        pass

    # CNN Layers
    def model_layers(self, inputs, is_training):
        reshaped_inputs = tf.reshape(
            inputs, [-1, self.input_dim, self.input_dim, 1])
        # Convolutional Layer #1
        conv1 = tf.layers.conv2d(
            inputs=reshaped_inputs,
            filters=32,
            kernel_size=[5, 5],
            padding='same',
            activation=tf.nn.relu,
            name='conv1')
        # Pooling Layer #1
        pool1 = tf.layers.max_pooling2d(
            inputs=conv1,
            pool_size=[2, 2],
            strides=2,
            name='pool1')
        # Convolutional Layer #2
        conv2 = tf.layers.conv2d(
            inputs=pool1,
            filters=64,
            kernel_size=[5, 5],
            padding='same',
            activation=tf.nn.relu,
            name='conv2')
        # Pooling Layer #2
        pool2 = tf.layers.max_pooling2d(
            inputs=conv2,
            pool_size=[2, 2],
            strides=2,
            name='pool2')
        # Dense Layer
        hwc = pool2.shape.as_list()[1:]
        flattened_size = hwc[0] * hwc[1] * hwc[2]
        pool2_flat = tf.reshape(pool2, [-1, flattened_size])
        dense = tf.layers.dense(pool2_flat, 1024,
            activation=tf.nn.relu, name='dense')
        # Apply Dropout
        dropout = self.apply_dropout(dense, is_training)
```