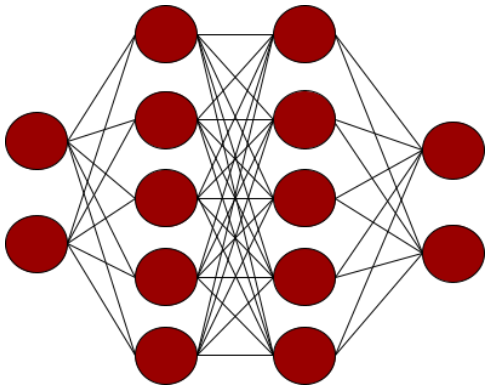
Dropout Regularization

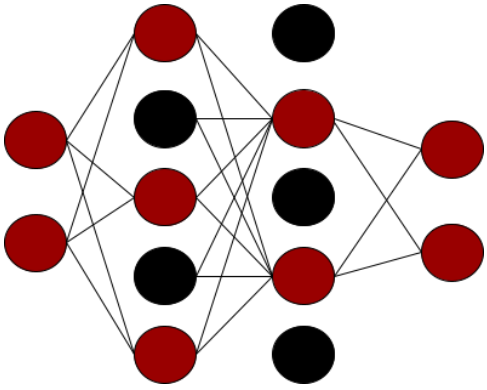
You've been exploring *overfitting*, where a network may become too specialized in a particular type of input data and fare poorly on others. One technique to help overcome this is use of *dropout regularization*.

When a neural network is being trained, each individual neuron will have an effect on neurons in subsequent layers. Over time, particularly in larger networks, some neurons can become overspecialized—and that feeds downstream, potentially causing the network as a whole to become overspecialized and leading to overfitting. Additionally, neighboring neurons can end up with similar weights and biases, and if not monitored this can lead the overall model to become overspecialized to the features activated by those neurons.

For example, consider this neural network, where there are layers of 2, 5, 5, and 2 neurons. The neurons in the middle layers might end up with very similar weights and biases.



While training, if you remove a random number of neurons and connections, and ignore them, their contribution to the neurons in the next layer are temporarily blocked



This reduces the chances of the neurons becoming overspecialized. The network will still learn the same number of parameters, but it should be better at generalization—that is, it should be more resilient to different inputs.

The concept of dropouts was proposed by Nitish Srivastava et al. in their 2014 paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting".

To implement dropouts in TensorFlow, you can just use a simple Keras layer like this:

```
tf.keras.layers.Dropout(0.2),
```

This will drop out at random the specified percentage of neurons (here, 20%) in the specified layer. Note that it may take some experimentation to find the correct percentage for your network.

For a simple example that demonstrates this, consider the Fashion MNIST classifier you explored earlier.

If you change the network definition to have a lot more layers, like this:

```
model = tf.keras.models.Sequential([
                    tf.keras.layers.Flatten(input_shape=(28,28)),
                    tf.keras.layers.Dense(256, activation=tf.nn.relu),
                    tf.keras.layers.Dense(128, activation=tf.nn.relu),
                    tf.keras.layers.Dense(64, activation=tf.nn.relu),
                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

Training this for 20 epochs gave around 94% accuracy on the training set, and about 88.5% on the validation set. This is a sign of potential overfitting.

Introducing dropouts after each dense layer looks like this:

```
model = tf.keras.models.Sequential([
                  tf.keras.layers.Flatten(input_shape=(28,28)),
                  tf.keras.layers.Dense(256, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(128, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(64, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

When this network was trained for the same period on the same data, the accuracy on the training set dropped to about 89.5%. The accuracy on the validation set stayed about the same, at 88.3%. These values are much closer to each other; the introduction of dropouts thus not only demonstrated that overfitting was occurring, but also that adding them can help remove it by ensuring that the network isn't overspecializing to the training data.

Keep in mind as you design your neural networks that great results on your training set are not always a good thing. This could be a sign of overfitting. Introducing dropouts can help you remove that problem, so that you can optimize your network in other areas without that false sense of security!