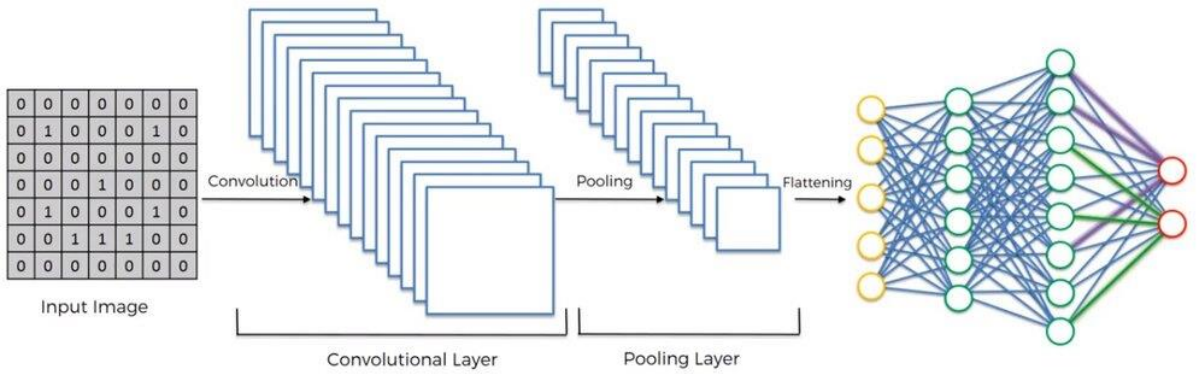
Convolutional neural networks, the type of model we will be using for this project, can be of different sizes and have different architecture setups. For this project, I tested several different models and found an architecture that trains quickly and works well for wildfire detection.

Let's quickly go over the general structure of a CNN so we know what to build:

1. A convolutional layer will be used to create feature maps (aspects of an image that are of interest to the CNN)
2. This feature map is then inputted into a pooling layer, which condenses the feature map into a smaller array
3. Several convolutional and pooling layers can be used in a network architecture, depending on the image size and/or complexity of the detection problem
4. After the final pooling layer, the input is then flattened, or reshaped into a vector
5. The vector is then inputted into a fully-connected artificial neural network, which delivers the final predicted output

This might be a little difficult to understand at first, so here's a concise diagram I found from [Panadda Kongsilp's great article on CNN's](#).



Let's move on to implementing the first step of this architecture—the convolutional, pooling, and flattening layers.

Part One

In this section, we'll work on transforming each image into a vector that can be inputted into a fully-connected network. To do this, we need to first import a variety of classes that we will need. Then, we can begin the convolution + pooling phases.

```
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Conv2D,
MaxPool2D, Flatten
```

The first statement imports the Sequential class, which is the class used to build sequential neural networks. The next statement imports *Dense*, *Conv2D*, *MaxPool2D,* and *Flatten*, which are the classes that will help us build each layer of the neural network.

- To build each convolutional layer, we'll use *Conv2D*
- To build each pooling layer, we'll use *MaxPool2D*
- To flatten all the data, we'll use *Flatten*
- Finally, the layers in the ANN can be made by using *Dense*

To begin building the network, we must instantiate the *Sequential* class so we can use the *.add* method to build the network architecture. This can be done as shown below.

```
cnn = Sequential()
```

Now we can add the convolution and pooling layers to the network as shown below; the amount of these layers can be changed, but keep in mind that changing the network will cause differences in performance. Thus, it is important to test different network architectures and determine which works best for the problem at hand.

After testing the several different size networks, 3 convolution layers and 2 pooling layers seemed to work the most efficiently on my computer. Let's work to implement this in code.

```
cnn.add(Conv2D(filters=32, kernel_size=3,
activation='relu', input_shape=[128, 128, 3]))
cnn.add(MaxPool2D(pool_size=2))

cnn.add(Conv2D(filters=32, kernel_size=3,
activation='relu'))
cnn.add(Conv2D(filters=32, kernel_size=3,
activation='relu'))
cnn.add(MaxPool2D(pool_size=2))
```

The *input_shape* argument is only needed in the first layer because that's the layer where we input the image itself; in the rest of the layers, the input is the output of the previous layer.

Now that we've finished the convolution/pooling process, we can just flatten the output from the final pooling layer so that it can be inputted into an ANN.

```
cnn.add(Flatten())
```

Great! Now we can move on to the next portion of our network architecture: the fully-connected layer.

Part Two

This can be build by creating an ANN with *Dense*, much like how we would if this were a regular classification problem. Once again, the number of *Dense* layers and the nodes in each layer can vary, but the numbers I have selected below created a network with good accuracy and training time.

Let's implement this part of the network into code so that we can test our image classifier.

```
cnn.add(Dense(units=128, activation='relu'))
cnn.add(Dense(units=128, activation='relu'))
cnn.add(Dense(units=128, activation='relu'))
cnn.add(Dense(units=128, activation='relu'))
cnn.add(Dense(units=128, activation='relu'))
cnn.add(Dense(units=1, activation='sigmoid'))
```

As you can see, this code creates an ANN with 4 hidden layers, one input layer, and one output layer. The final layer will output either 0 or 1 depending on the image's classification.

But since we've completed this layer, we've completed the architecture of our neural network. Now all we have to do is compile the CNN and then we can begin training it in the next section.

We can now compile the network by using the *.compile* method of the *Sequential* class as shown below.

```
cnn.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy’]
```