let’s take a quick look at some of the images to see what we’re working with, and then we’ll move on to library installation.



Library Installation

To complete this project, we need to install three important libraries: OpenCV, Tensorflow and Numpy

OpenCV is a library that will help us take pictures with our computer’s camera. It also provides many resources for machine learning itself, but we’re going to use TensorFlow to create our model.

Tensorflow is a library that we will be using to create deep learning models such as neural networks. It provides several data preprocessing classes which will help us extract data from images and other input sources.

Numpy is another library that makes it easy to work with arrays. It provides several unique functions that will help in data preprocessing.

Fortunately, these libraries can be quickly installed by using Pip, Python’s default package-management system. All we have to do is enter the following lines of code into terminal.

pip install numpy

pip install tensorflow

Now we can begin data preprocessing!

Data Preprocessing

Image Augmentation

One of the most important parts of creating a computer vision model is image augmentation, which is the process by which we artificially create images by transforming the given inputs to generate more data and reduce chances of overfitting. Feel free to check out the link above to learn more about this topic.

To get started with image augmentation, we can instantiate Tensorflow’s ImageDataGenerator class, which will help us run a series of transformations on each image.

First, we’ll create an instance of the class for our training data to apply the necessary transformations. This can be done as shown below. First, we import the necessary class, and then we simply create an instance through variable assignment.

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_datagen = ImageDataGenerator(rescale=1./255, shear\_range=.2, vertical\_flip=True, horizontal\_flip=True, zoom\_range=.2)

train = train\_datagen.flow\_from\_directory(directory='data/train', target\_size=(128, 128), class\_mode='binary')

The .flow\_from\_directory method extracts all the images from the given folder and places them into a DirectoryIterator, which consists of Numpy arrays in batches.

Now, all we have to do is create a different instance of the class and use the same .flow\_from\_directory as we did before. However, we don’t perform image augmentation since this is the test data, and we want to simulate a real-life situation as accurately as possible.

test\_datagen = ImageDataGenerator(rescale=1./255)

test = test\_datagen.flow\_from\_directory(directory='data/test', target\_size=(128, 128), class\_mode='binary')

Now we have two DirectoryIterators that contain the train and test images separately. Each of these can be directly inputted into the CNN, as they contain the both the inputs (images) and the outputs (labels).

This is because Tensorflow is able to find the image labels based on the directory structure we set up previously. Since the not\_wildfire directory comes before the wildfire directory, non-wildfires correspond with outputs of 0, and wildfires correspond to outputs of 1.

Creating the Network Architecture

An Overview

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:

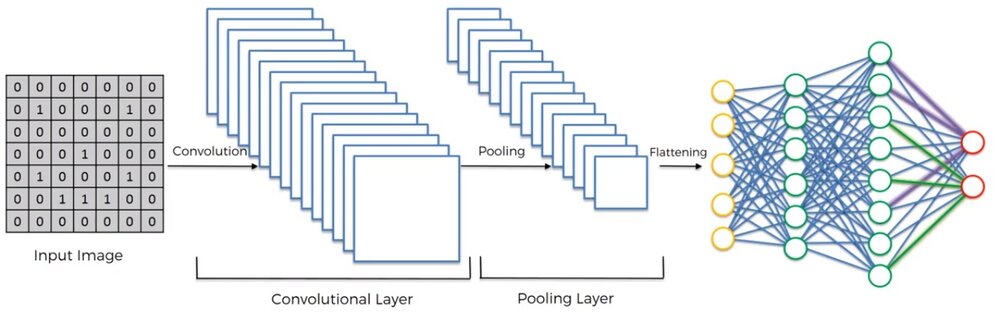
A convolutional layer will be used to create feature maps (aspects of an image that are of interest to the CNN)

This feature map is then inputted into a pooling layer, which condenses the feature map into a smaller array

Several convolutional and pooling layers can be used in a network architecture, depending on the image size and/or complexity of the detection problem

After the final pooling layer, the input is then flattened, or reshaped into a vector

The vector is then inputted into a fully-connected artificial neural network, which delivers the final predicted output



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”]