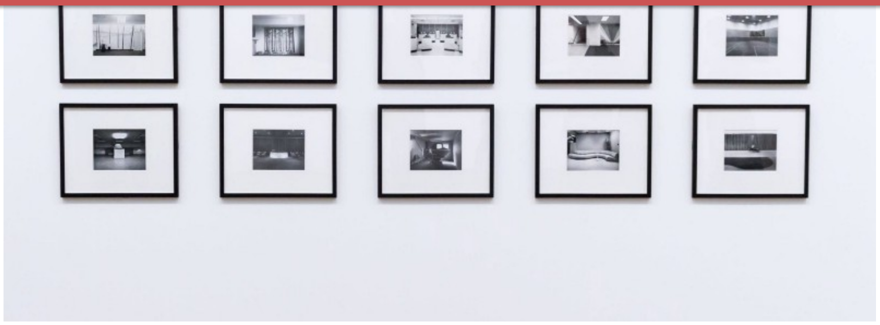


Build your own Image classifier with Tensorflow and Keras

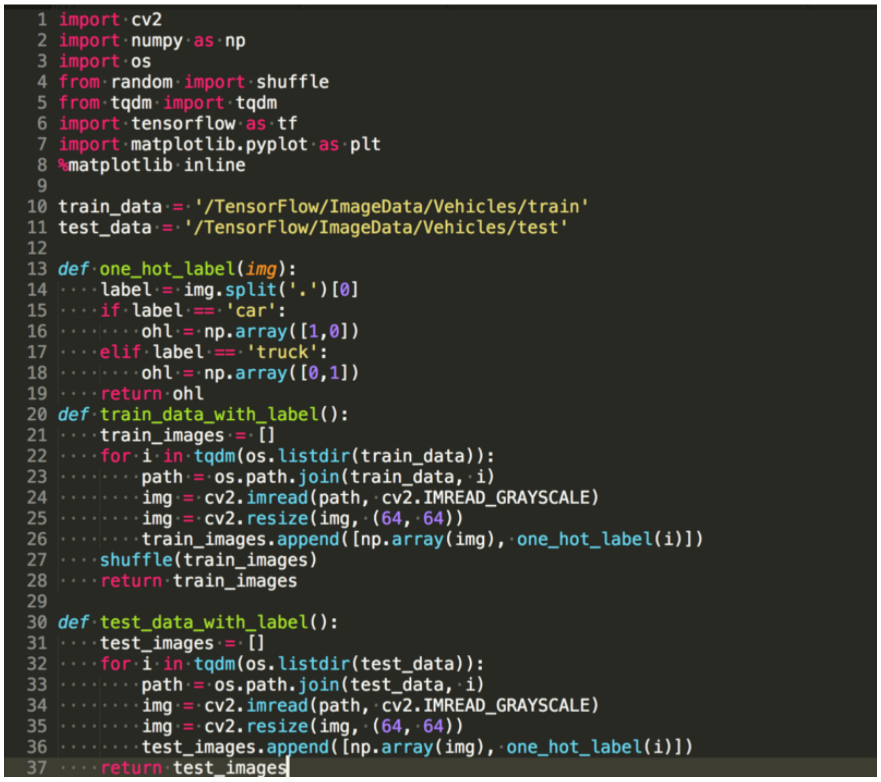
Arun Prakash Follow
May 15, 2018 · 6 min read

We have already seen why convolutional neural network is suitable for image Processing. Now we can build our own image classifier using Convolutional neural network. We are implementing this using Python and Tensorflow.

Once we complete the installation of Python and Tensorflow we can get started with the training data setup. We can download the images of our choice from google. In this implementation we are focusing on classifying images of cars and trucks. So download 125 images of cars and 125 images of trucks. we can use a simple chrome add-on (Fatkun Batch downloader) to download all the images in a page. Make sure all the car images are renamed as car.<<seq num>>.jpeg and truck images are renamed as truck.<<seq num>>.jpeg, because we are going to label the training images based on its name.

If the image setup is ready then we can split the dataset into train and test datasets. keep 100 images in each class as training set and 25 images in each class as testing set. Keep the training and testing images in a separate folder.

Below is the code for preparing the image data and converting the image into n-dimentional pixel arrays. we will be using opencv for this task.



```python
import cv2
import numpy as np
import os
from random import shuffle
from tqdm import tqdm
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline

train_data = '/TensorFlow/ImageData/Vehicles/train'
test_data = '/TensorFlow/ImageData/Vehicles/test'

def one_hot_label(img):
    label = img.split('.')[0]
    if label == 'car':
        ohl = np.array([1,0])
    elif label == 'truck':
        ohl = np.array([0,1])
    return ohl
def train_data_with_label():
    train_images = []
    for i in tqdm(os.listdir(train_data)):
        path = os.path.join(train_data, i)
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, (64, 64))
        train_images.append([np.array(img), one_hot_label(i)])
    shuffle(train_images)
    return train_images

def test_data_with_label():
    test_images = []
    for i in tqdm(os.listdir(test_data)):
        path = os.path.join(test_data, i)
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, (64, 64))
        test_images.append([np.array(img), one_hot_label(i)])
    return test_images
```

In the above code `one_hot_label` function will add the labels to all the images based on the image name. say the image name is car.12.jpeg then we are splitting the name using "." and based on the first element we can label the image data. Here we are using the one hot encoding. In one hot encoding say if we have 5 classes then the only the valid class will have the value as 1 and rest will be zero. In our case we have two classes ["car", "truck"] so all the car images will have the label [1,0] and all the truck images will have [0,1].

Function `test_data_with_label` will be converting our image data into numpy array of size 64*64. Keep in mind that the original images we downloaded from the web will be having different resolutions and here we are reshaping every image into 64*64, it's completely an arbitrary value you can even reshape your image into 128*128 or even 16*16, make sure you keep atleast some significant imformation of the image even after reshaping. And also we are converting all the images into grayscale images because the color of the truck/car doesn't matter here. In this function we are converting the image into pixel array and adding the one-hot encoded label to it.

Other function `test_data_with_label` will also do the same for test dataset, the only difference is we are not shuffling the testing data. The reason for shuffling the training set is we want to make sure our model gets images of both classes in it's every batch. If we don't shuffle then we may end up passing only the car images in an entire batch and only truck images in another batch. We are trying to get closer to the result iteratively and our model will be calculating the error based on the prediction and the label for each batch, so if we dont shuffle our data then the learning of our model won't be ideal because we are letting a single class dictate the direction of the path for too long might send the path too far and in a worst case scenario to a local optimum from which the optimization problem would not be able to recover even after seeing the other classes.

Now we have the training and testing data ready, all we need to do is build our model. For model creation we are going to use Keras.

```
from keras.models import Sequential
from keras.layers import *
from keras.optimizers import *
```

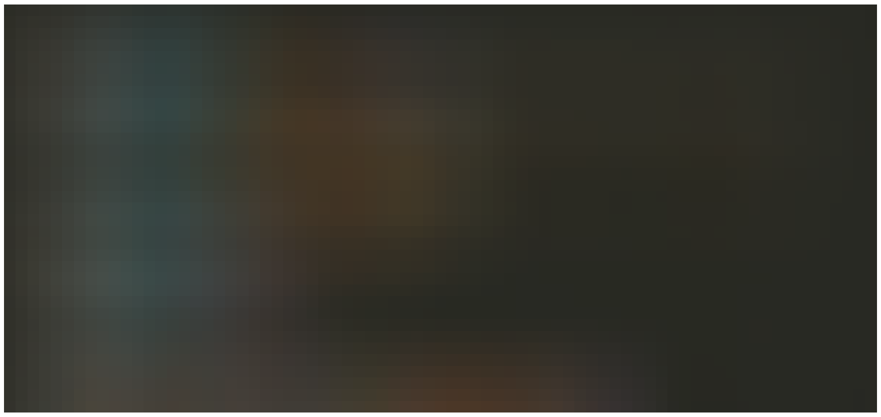
We need to import Sequential model, layers and optimizers from keras. Then load the data to a variable.

```
training_images = train_data_with_label()
testing_images = test_data_with_label()


tr_img_data = np.array([i[0] for i in
training_images]).reshape(-1,64,64,1)
tr_lbl_data = np.array([i[1] for i in training_images])


tst_img_data = np.array([i[0] for i in
testing_images]).reshape(-1,64,64,1)
tst_lbl_data = np.array([i[1] for i in testing_images])
```

The image will be of size 64*64 and this needs to be flattened to be passed through the convolution layer, so we are reshaping it again to -1*64*64*1. 1 represents the color code as grayscale.



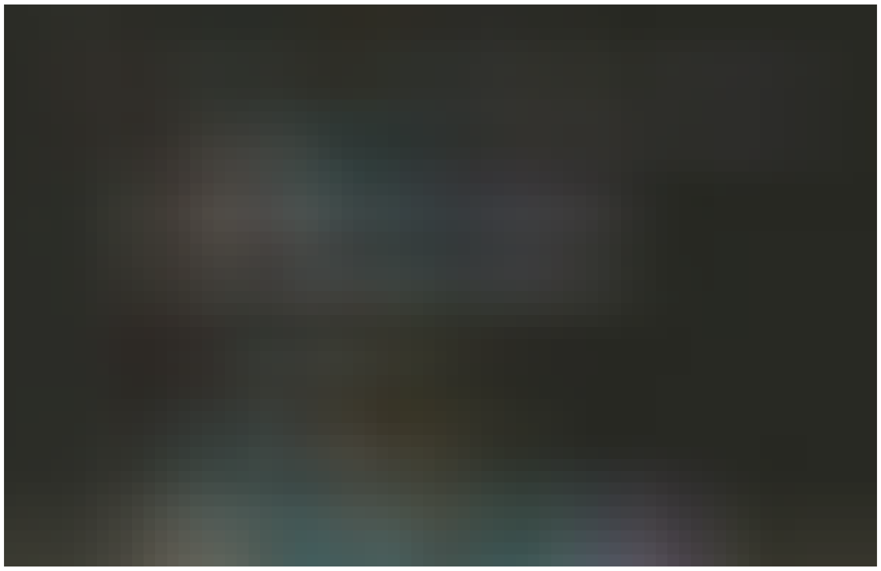
Here we are using 3 convolution layers with single stride, zero padding and relu activation(you can even try changing the activation functions and see how the model behaves). If you notice for each layer the filter count is increasing, because the initial layer represents the high level features of the image and the deeper layers will represent more detailed features and so they usually have more number of filters. And the filter count is a arbitrary number and we can play with it and see the model behaviour.

After three convolution layers we have one dropout layer and this is to avoid overfitting problem. And once the image pass through the convolution layers it has to be flattened again to be fed into fully connected layers(it's called a dense layer in keras, here all the neurons in first layer is connected to all the neurons in the second layer.

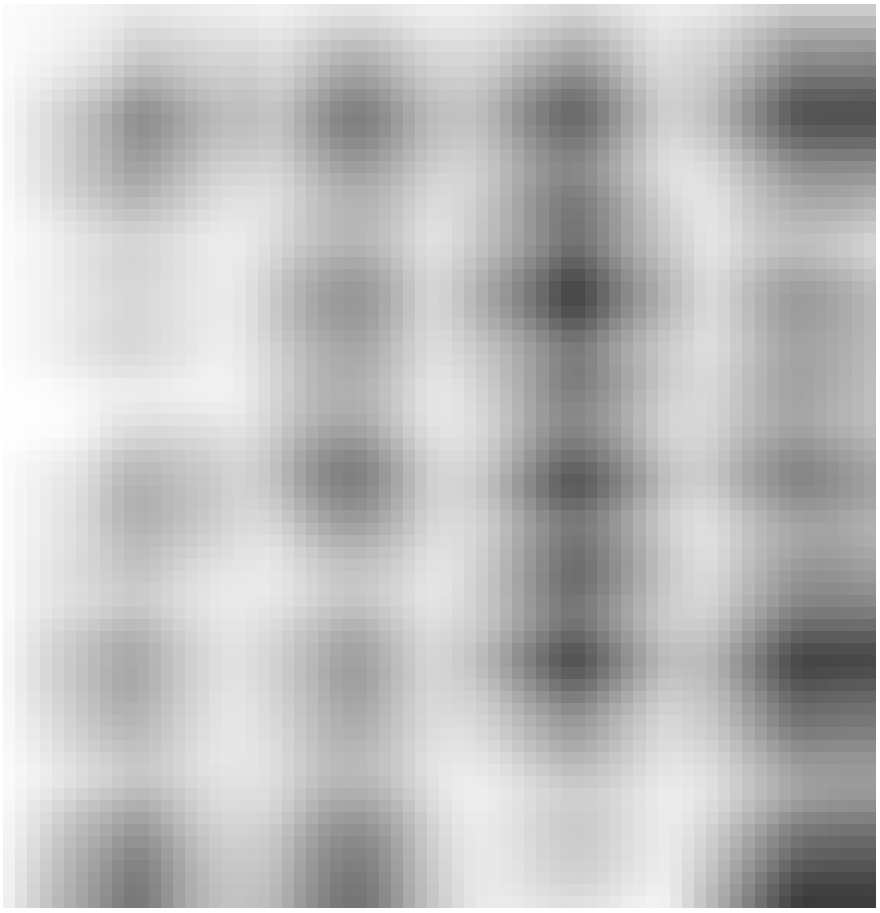
We have 2 dense layers and the first one is having 512 neurons and relu activation, this is also arbitary and we can have the neuron count as per our choice. the second dense layer will have only 2 neurons as we have only two classes to classify, usually the number of neurons in the output layer will be equal to number of classes in our problem. This layer will use softmax activation. softmax activation will calculate the probabilities of each target class over all possible target classes and the sum of all the probablities will always be 1. The input will be classified into any of the target class based on the higher probaility value in softmax. In our case if the output of softmax is [0.7,0.3] then the input image is a Car.

We are using Adam optimizer with "categorical_crossentropy" as loss function and learning rate of 0.001. We train our model for 50 epochs (for every epoch the model will adjust its parameter value to minimize the loss) and the accuracy we got here is around 99%.

Now its time to test our model against the test dataset we have. Below is the code for plotting the image.



Here we go with our result!



For just 200 training images, the model gave some pretty good results!

. . .

Francium Tech is a technology company laser focussed on delivering top quality software of scale at extreme speeds. Numbers and Size of the data don't scare us. If you have any requirements or want a free health check of your systems or architecture, feel free to shoot an email to contact@francium.tech, we will get in touch with you!