Comparison of Image Classification and CAPTCHA Classification Models for Development Purposes

Contents

Context

What is Image Classification?

Image classification is a supervised learning problem in which a model is trained to recognize a collection of target classes (objects to identify in pictures) using labelled sample photos. Picture classification, localization, image segmentation, and object identification are some of the key challenges in computer vision. Image classification is one of the most essential issues among them. Image classification applications are employed in a variety of applications, including medical imaging, satellite image object identification, traffic management systems, and, in the case of our study, cracking Image CAPTCHAs.

A basic summary of how it works is that a computer analyzes an image in the form of its pixels. It accomplishes this by treating the picture as an array of matrices, the size of which is determined by

the image resolution. Image classification is accomplished in digital image processing by automatically grouping these sets of pixels into defined categories, referred to as "classes".

Traditional machine learning and AI-based Deep Learning are the two major methodologies of image classification. Deep learning will be the major focus of this paper since it offers the most versatility in terms of models that may be investigated. Convolutional Neural Networks (CNNs) will be investigated in this case of image classification because they are purpose-built for the task.

A CNN is a machine learning framework that was created utilizing machine learning ideas. Without the need for human interaction, CNNs can learn and train from data on their own. When employing CNNs, just a little amount of pre-processing is required because they create and adapt their own picture filters, which must be properly designed for most algorithms and models.

CNN Terminology

A CNN's top-level overview is as follows: A neuron is the fundamental unit of a CNN; they are statistical functions that calculate the weighted average of inputs and apply a mathematical function to the output. Layers are a collection of neurons, each of which has a specific purpose. Depending on the depth sought and the CNN's function, a CNN system can include anywhere from a few to hundreds of layers.

The CAPTCHA Problem

Hardhat Enterprises is the title of the company I work for in this capstone course. The creation and manufacture of offensive cyber tools and capability is the company's main focus. One of these goals is to discover a means to improve on present CAPTCHA cracking attempts (Completely Automated Public Turing test to tell Computers and Humans Apart). There are two types of CAPTCHA available at present time: text and image. The focus of this study will be on Image CAPTCHAs, with the goal of improving on current approaches by comparing existing CNN models and seeing if stacking any of these models (ensemble) has any influence on the output. The current solution is that of a combination of Faster R-CNN and NASNet for feature extraction, it is generally fast, accurate and a sound solution for the images that contain multiple objects. The aim of this project is to break down the difference between Neural Network architectures, and the contexts they are used in, as well as demonstrating new combinations of models to further investigate.

Current Solution: Faster-RCNN + NASNet

In the current solution supplied in the GitHub for the Image breaking captcha team currently implements an object detection solution called Faster R-CNN which is a Google implementation. To understand why this is an optimal solution for fast and timely CAPTCHA solving, it is beneficial to understand how R-CNNs work first.

"Region-based Convolutional Neural Network" is abbreviated as R-CNN. There are two steps to the core concept. It starts by identifying a manageable number of bounding-box object region candidates via selective search. After that, each region's CNN features are extracted separately for classification. This is perfect for a CAPTCHA cracking model since most photos contain an item(s) that must be detected in order to identify whether or not the image contains that object. On its own

however, the R-CNN model is expensive and slow. This is why “faster” versions of this model were developed, to provide a viable real-time solution to an interesting problem.

CNN Demonstration

I'll present a summary of a fundamental simple ANN (Artificial Neural Network) image classification model, followed by a simple CNN image classification model in Tensorflow, to create a foundation for model research. This will demonstrate why CNNs are superior to ANN models and provide a basic understanding of how they function. All of these models will be performed in Google CoLab for standardization reasons, as it is a free notebook application that can leverage GPUs for processing. The CIFAR-10 dataset, which contains 60000 32x32 color pictures with 10 classes and 6000 images each class, will be used throughout this report. We're utilizing this dataset since it includes a pre-labeled dataset with enough photos to offer a suitable baseline for additional testing.

First, we load the packages needed for this exercise. We load the CIFAR-10 dataset from Tensorflow and check the shape of some of the datasets.

```
# Import Packages
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import numpy as np
```

```
[ ] (X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()
    X_train.shape
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 7s 0us/step
170508288/170498071 [==============================] - 7s 0us/step
(50000, 32, 32, 3)
```

```
[ ] X_test.shape
```

```
(10000, 32, 32, 3)
```

```
[ ] y_train.shape
```

```
(50000, 1)
```

Through more testing we find that y_train is a 2D array, however for our classification having a 1D array is good enough so we will reshape it into a 1D array.

```
[ ] y_train[:5]
```

```
array([[6],
       [9],
       [9],
       [4],
       [1]], dtype=uint8)
```

```
[ ] y_train = y_train.reshape(-1,)
    y_train[:5]
```

```
array([6, 9, 9, 4, 1], dtype=uint8)
```

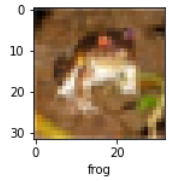
```
[ ] y_test = y_test.reshape(-1,)
```

Next, we classify what the classes are for the dataset and see an example of one of the images using matplotlib:

```
[ ]  classes = ["airplane","automobile","bird","cat","deer","dog","frog","horse","ship","truck"]
```

```
[ ]  def plot_sample(X, y, index):
       plt.figure(figsize = (15,2))
       plt.imshow(X[index])
       plt.xlabel(classes[y[index]])
```

```
[ ]  plot_sample(X_train, y_train, 0)
```



Now we quickly normalize the images to a number from 0 to 1. Each image has 3 channels (r,g,b) and each value in the channel can range from 0 to 255. So, to normalize in our desired range, we just divide it by 255.

```
[ ]  X_train = X_train / 255.0
     X_test = X_test / 255.0
```

Now we can start training models.

ANN Image Classification

Tensorflow has built in functions to build easy to use NN models. The function that we will be using is the Sequential model which allows for step-by-step construction of a model layer-by-layer. For simplistic purposes, this model will consist of 4 separate layers: A flatten layer to flatten each image from rgb to a single array, 2 ‘ReLu’ layers to perform calculations, and a final ‘softmax’ layer to give each output a value. The optimizer used will be very basic for these two models and the metrics will be based on accuracy. 5 epochs will be used as any more will be a waste of time for this particular model. The number of epochs will determine how many times the model runs through the training set.

```
[ ] ann = models.Sequential([
                            layers.Flatten(input_shape=(32,32,3)),
                            layers.Dense(3000, activation='relu'),
                            layers.Dense(1000, activation='relu'),
                            layers.Dense(10, activation='softmax')
    ])

    ann.compile(optimizer='SGD',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])

    ann.fit(X_train, y_train, epochs=5)
```

```
Epoch 1/5
1563/1563 [==============================] - 12s 6ms/step - loss: 1.8112 - accuracy: 0.3546
Epoch 2/5
1563/1563 [==============================] - 10s 6ms/step - loss: 1.6251 - accuracy: 0.4279
Epoch 3/5
1563/1563 [==============================] - 10s 7ms/step - loss: 1.5433 - accuracy: 0.4559
Epoch 4/5
1563/1563 [==============================] - 11s 7ms/step - loss: 1.4811 - accuracy: 0.4779
Epoch 5/5
1563/1563 [==============================] - 10s 7ms/step - loss: 1.4318 - accuracy: 0.4950
<keras.callbacks.History at 0x7f2ca2174550>
```

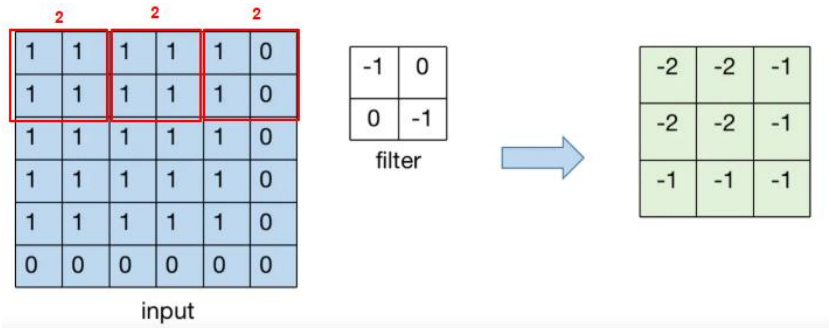
As you can see, after 5 epochs there is an accuracy of about 50%. Which is not ideal. Let's try a CNN.

CNN Image Classification

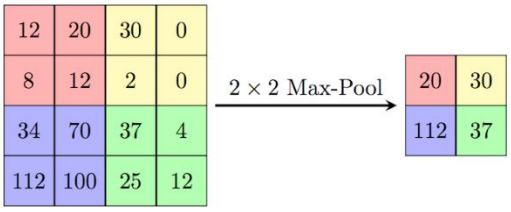
The model that we will be using for this example involves 7 layers. Again, this is a basic example for demonstration purposes only.

This model introduces 2 new concepts of a Conv2D layer and a MaxPooling2D layer:

- Conv2D: A kernel (filter) that passes over an image, from left to right, top to bottom, applying a convolution product. The convolution product is an elementwise (or pointwise) multiplication. The sum of this result is the resulting pixel on the output image. Below is a basic example:



- MaxPooling2D: A down sampled (pool) feature map is created by calculating the highest value for patches of a feature map and then using that value to generate a down sampled (pool) feature map. After a convolutional layer, it's commonly utilized. It's widely used to quickly decrease the size of an image.



Together with the layer types introduced in the previous model, we can create a basic CNN model that increases the accuracy over the much simpler ANN.

```
[ ]  cnn = models.Sequential([
                            layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=(32,32,3)),
                            layers.MaxPooling2D((2,2)),

                            layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu'),
                            layers.MaxPooling2D((2,2)),

                            layers.Flatten(),
                            layers.Dense(64, activation='relu'),
                            layers.Dense(10, activation='softmax')
     ])
```

```
[ ]  cnn.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])
```

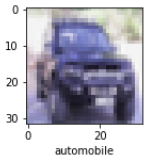
```
[ ]  cnn.fit(X_train, y_train, epochs=10)
```

```
Epoch 1/10
1563/1563 [==============================] - 17s 6ms/step - loss: 1.5319 - accuracy: 0.4478
Epoch 2/10
1563/1563 [==============================] - 9s 6ms/step - loss: 1.1586 - accuracy: 0.5953
Epoch 3/10
1563/1563 [==============================] - 10s 6ms/step - loss: 1.0208 - accuracy: 0.6444
Epoch 4/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.9376 - accuracy: 0.6750
Epoch 5/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.8757 - accuracy: 0.6969
Epoch 6/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.8202 - accuracy: 0.7157
Epoch 7/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.7748 - accuracy: 0.7300
Epoch 8/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.7333 - accuracy: 0.7432
Epoch 9/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.6911 - accuracy: 0.7588
Epoch 10/10
1563/1563 [==============================] - 10s 6ms/step - loss: 0.6566 - accuracy: 0.7695
<keras.callbacks.History at 0x7f2c25bc30d0>
```

You can see that even at the end of 5 epochs, accuracy was around 70% which is a significant improvement over ANN. An advantage of using CNNs is that computation is much less as maxpooling reduces the image dimensions while still preserving the features.

Let’s test the accuracy with one of the test images:

```
[ ] cnn.evaluate(X_test, y_test)

    313/313 [==============================] - 1s 4ms/step - loss: 0.9222 - accuracy: 0.6995
    [0.9222467541694641, 0.6995000243186951]

[ ] y_pred = cnn.predict(X_test)

[ ] y_classes = [np.argmax(element) for element in y_pred]
    y_classes[:5]

    [3, 8, 8, 0, 6]

[ ] y_test[:5]

    array([3, 8, 8, 0, 6], dtype=uint8)

[ ] plot_sample(X_test, y_test, 9)
```



Obviously with only 70% this is not a desired outcome, but it is a good start. The aim will be to find a model, or combination of models that can be used with the best accuracy, especially for when we move to higher resolution CAPTCHA images. Training on low image datasets is also much less computationally expensive.

Image Classification Model Research

For this report, I will be going through 3 different implementations of different Image Classification CNNs and compare the accuracy against the simple form of the CNN demonstrated in the previous example. This is to provide some more background before getting into the technical innovational side of things. Later in this report, I will also cover 3 different object detection models to see if they provide any improvement over the current implementation or the image classification models.

LeNet

The most widely used CNN architecture is LeNet, which was also the first CNN model established in 1998. LeNet was created to classify handwritten digits from 0 to 9 in the MNIST Dataset. It has seven layers, each with its own set of configurable settings. It takes a 32 × 32 pixel image, which is significantly larger than the images used to train the network. The activation function that is employed is ReLu. The structure is as follows:

1. Convolutional layer with 6 5x5 filters with a stride of 1
2. A 2x2 Average Pooling layer with a stride of 2
3. Convolutional layer with 16 5x5 filters with a stride of 1
4. A 2x2 Average Pooling layer with a stride of 2
5. Fully connected layer consisting of 120 nodes
6. Fully connected layer consisting of 84 nodes
7. Softmax layer with 10 outputs

As seen below:

```
[28] LeNet.summary()

Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 28, 28, 6)         456

 average_pooling2d_8 (Averag  (None, 14, 14, 6)        0
 ePooling2D)

 conv2d_9 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_9 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten_4 (Flatten)         (None, 400)               0

 dense_12 (Dense)            (None, 120)               48120

 dense_13 (Dense)            (None, 84)                10164

 dense_14 (Dense)            (None, 10)                850

=================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
```

Note that if we ran this model as is on the current X_train and X_test, after 10 epochs, the accuracy will be around:

```
Epoch 9/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.9272 - accuracy: 0.6719
Epoch 10/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.8958 - accuracy: 0.6820
<keras.callbacks.History at 0x7f1864462d10>
```

This is because LeNet is built for classifying handwritten black and white text. So, in order to do this, we must change the initial image to a greyscale form:

```
[32] X_train_gray_norm = (X_train_gray - 128) / 128
     X_test_gray_norm = (X_test_gray - 128) / 128

[33] X_train_gray_norm.shape

     (50000, 32, 32, 1)
```

This results in an improved accuracy after 10 epochs:

```
Epoch 9/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.5402 - accuracy: 0.8114
Epoch 10/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.4726 - accuracy: 0.8337
<keras.callbacks.History at 0x7f18642cf590>
```

This is a decent result for a CNN that was built for classifying images of text.

AlexNet

This network has a similar design to the original LeNet, but it has a lot more filters, allowing it to categorize a lot more objects. Furthermore, rather of regularization, "dropout" is used to cope with overfitting. Because it was designed to categorize 1000 different classes rather than 10, this network may be a bit excessive for the present dataset.

Dropout is a technique where you remove nodes in a neural network to simulate training large numbers of architectures simultaneously. For this model we are required to upsize the image to 64x64 as seen below:

```
[8]  def process_image(image,label):
         image=tf.image.per_image_standardization(image)
         image=tf.image.resize(image,(64,64))

         return image,label
```

The model for this network has many more layers in comparison to LeNet utilizing batch normalization and dropout layers:

```
[9]  AlexNet = keras.models.Sequential([
        keras.layers.Conv2D(filters=128, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(64,64,3)),
        keras.layers.BatchNormalization(),
        keras.layers.MaxPool2D(pool_size=(2,2)),
        keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"),
        keras.layers.BatchNormalization(),
        keras.layers.MaxPool2D(pool_size=(3,3)),
        keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
        keras.layers.BatchNormalization(),
        keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1), activation='relu', padding="same"),
        keras.layers.BatchNormalization(),
        keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1), activation='relu', padding="same"),
        keras.layers.BatchNormalization(),
        keras.layers.MaxPool2D(pool_size=(2,2)),
        keras.layers.Flatten(),
        keras.layers.Dense(1024,activation='relu'),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(1024,activation='relu'),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(10,activation='softmax')

    ])
```

With almost 3 million total parameters, in comparison to LeNet's 62,000:

```
=================================================================
Total params: 2,915,338
Trainable params: 2,913,034
Non-trainable params: 2,304
_________________________________________________________________
```

Because of the complex nature of this network, it took 50 epochs to see some good results:

```
Epoch 50/50
1562/1562 [==============================] - 11s 7ms/step - loss: 0.0592 - accuracy: 0.9807
- val_loss: 2.4238 - val_accuracy: 0.6069
```

ResNet50

ResNet is a popular deep learning model that was initially released in 2015. ResNet is one of the most popular and effective NN models available today. A residual block is what ResNets are built out of. This is based on the notion of "skip-connections" and makes extensive use of batch-normalization (which standardizes the inputs to a layer for each mini-batch) in order to train hundreds of layers without compromising performance.

The concept was that the deeper layers shouldn't make any more faults in training than their shallower counterparts. Skip-connections were established to put this concept into effect.

ResNet50 is a residual CNN that is 50 layers deep. The reason I chose this model in particular is because it is a very good medium to explore ResNets. Tensorflow also has this model with the weights pre-trained on a Dataset named 'ImageNet' for easy implementation.

It would take too long to describe all of the layers, but here are the important parts. The classification section:

```
def classifier(inputs):
    x = tf.keras.layers.GlobalAveragePooling2D()(inputs)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(1024, activation="relu")(x)
    x = tf.keras.layers.Dense(512, activation="relu")(x)
    x = tf.keras.layers.Dense(10, activation="softmax", name="classification")(x)
    return x
```

The overall model, as well as an up_sampling2d layer to increase the size of the images to 224x224 (also note the massive amount of parameters):

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 up_sampling2d (UpSampling2D  (None, 224, 224, 3)      0
 )

 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 1024)              2098176

 dense_1 (Dense)             (None, 512)               524800

 classification (Dense)      (None, 10)                5130

=================================================================
Total params: 26,215,818
Trainable params: 26,162,698
Non-trainable params: 53,120
_________________________________________________________________
```

As this is a massive neural network, I don't have the resources to do multiple epochs. However, after just one pass through the network, an accuracy of 92% was achieved. Showing the capability for accuracy. The issue appears if a CAPTCHA would be completed in a timely manner or not.

```
[25] EPOCHS = 1
     history = model.fit(X_train, y_train, epochs=EPOCHS, validation_data = (X_test, y_test), batch_size=64)

     782/782 [==============================] - 1050s 1s/step - loss: 0.3444 - accuracy: 0.8878 - val_loss: 0.2110 - val_accuracy: 0.9293
```

Object Detection Model Research

What is Object Detection?

Object detection is a critical computer vision problem that detects occurrences of certain visual objects in digital pictures. The purpose of object detection is to create computational models that answer the most basic question that computer vision applications have: "What things are there and where are they?" The fast advancements of deep learning algorithms have substantially boosted the pace of object detection in recent years. With deep learning networks and the computing power of GPU's, the performance of object detectors and trackers has greatly improved, achieving significant breakthroughs in object detection.

One-stage vs two-stage deep learning object detectors

In general, deep learning-based object detectors extract features from the input image or video frame. An object detector solves two subsequent tasks: Find and arbitrary number of objects (possibly zero); and classify every single object and estimate its size with a bounding box. To simplify the process, you can separate those tasks into two stages. Other methods combine both tasks into one step (single-stage detectors) to achieve higher performance at the cost of accuracy.

Two-stage detectors: In two-stage detectors, deep features are utilized to suggest approximate object areas before they are used for classification and bounding box regression for the object candidate. The maximum detection accuracy is achieved by two-stage algorithms, although they are often slower. The performance is not as strong as one-stage detectors due to the multiple inference stages per image. The currently implemented solution of Faster R-CNN + NASNet is an example of a two-stage method.

One-stage detectors: One-stage detectors predict bounding boxes over the images without the region proposal step. Because this procedure takes less time, it may be employed in real-time applications. The disadvantage is that they have difficulty detecting irregularly shaped objects or groups of small objects, which should not be a problem with CAPTCHA images. YOLO, SSD, and RetinaNet are some of the most popular one-stage detectors.

Faster R-CNN and pre-trained classification models

Even though R-CNN was already covered briefly on page 2, it is worth going over again to understand how it works. R-CNN consists of 3 simple steps:

1. Scan the input image for possible objects using an algorithm that generates around 2000 region proposals
2. Run a CNN on top of each of these region proposals
3. Take the output of each CNN and classify the region and tighten the bounding box surrounding the object if the object exists.

The Faster R-CNN used for this project is developed by Google and trained on the Open Images v4 dataset. For the test done with each model combination, the same sample image will be used for standardization:

The reason for choosing this image is that Image CAPTCHAs are never very complicated. This image contains enough cars, or traffic lights to be a good test of the output of these models. Keep in mind that while I am using Google CoLab, there is a bit of variance in output times as the GPUs used in the runtime, often vary in the free version.

Faster R-CNN + Inception ResNet v2

Inception CNN ResNet v2 was trained on over a million images from the ImageNet database. The 164-layer network can identify photos into 1000 different item types (but has been scaled down to 600 for this model). As a result, over a wide variety of images, this network has learned rich feature representations. The network accepts images with a resolution of 299x299 pixels. It is a robust and powerful network when combined with Faster R-CNN. Let's begin by importing all of the necessary packages:

```
[ ]  # For running inference on the TF-Hub module.
     import tensorflow as tf

     import tensorflow_hub as hub

     # For downloading the image.
     import matplotlib.pyplot as plt
     import tempfile
     from six.moves.urllib.request import urlopen
     from six import BytesIO

     # For drawing onto the image.
     import numpy as np
     from PIL import Image
     from PIL import ImageColor
     from PIL import ImageDraw
     from PIL import ImageFont
     from PIL import ImageOps

     # For measuring the inference time.
     import time
```

I found some pre-built functions for displaying the image, resizing the image, and drawing the bounding boxes on the image. I won't post the whole screenshot just because each model is a bit different. However, the final notebook will be uploaded for review.

```
[ ]  def display_image(image):
       fig = plt.figure(figsize=(20, 15))
       plt.grid(False)
       plt.imshow(image)


     def download_and_resize_image(url, new_width=256, new_height=256,
                                   display=False):
       _, filename = tempfile.mkstemp(suffix=".jpg")
       response = urlopen(url)
       image_data = response.read()
       image_data = BytesIO(image_data)
       pil_image = Image.open(image_data)
       pil_image = ImageOps.fit(pil_image, (new_width, new_height), Image.ANTIA
       pil_image_rgb = pil_image.convert("RGB")
       pil_image_rgb.save(filename, format="JPEG", quality=90)
       print("Image downloaded to %s." % filename)
       if display:
         display_image(pil_image)
       return filename


     def draw_bounding_box_on_image(image,
                                    ymin,
                                    xmin,
                                    ymax,
                                    xmax,
                                    color,
                                    font,
                                    thickness=4,
                                    display_str_list=()):
       """Adds a bounding box to an image."""
       draw = ImageDraw.Draw(image)
       im_width, im_height = image.size
       (left, right, top, bottom) = (xmin * im_width, xmax * im_width,
                                     ymin * im_height, ymax * im_height)
       draw.line([(left, top), (left, bottom), (right, bottom), (right, top),
                  (left, top)],
                 width=thickness,
                 fill=color)
```

Next load the image (Some images can't be downloaded due to not having the right permissions):

```
image_url = "https://www.courant.com/resizer/7-BKIrVYKscG8JM4KHVl3GBktd0=/1200x630/filters:format(jpg):quality(70)/cloudfront-us-east-1.images.arcpublishing.com/tronc/AOIOFMQOPVEEBE6MA5UOPGUYTU.jpg"
downloaded_image_path = download_and_resize_image(image_url, 1280, 856, True)
```

Now we have to load the model from Tensorflow hub. The model is pre-trained, so it doesn't require any time-consuming training.

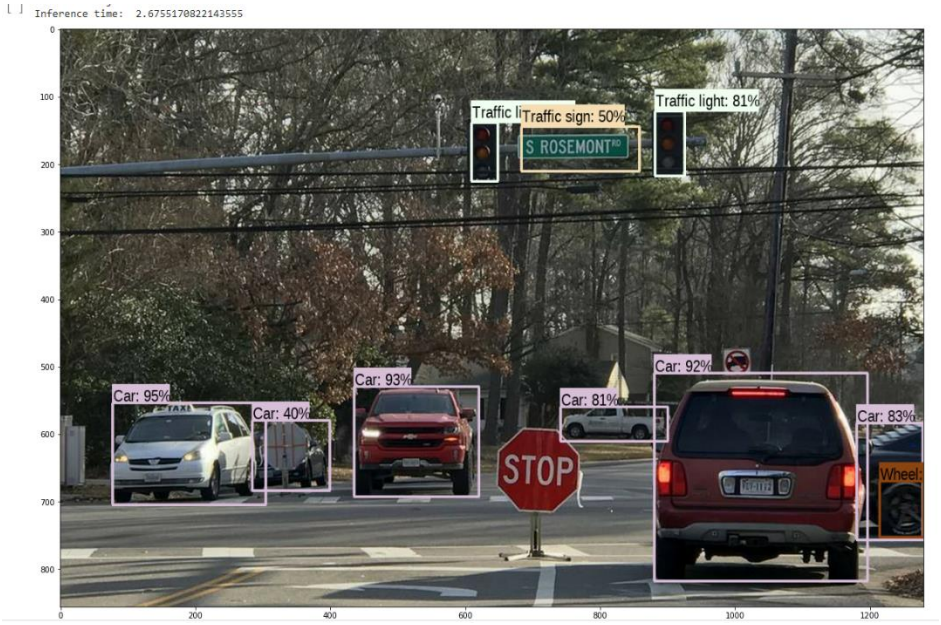
```
[ ]  module_handle = "https://tfhub.dev/google/faster_rcnn/openimages_v4/inception_resnet_v2/1"

     detector = hub.load(module_handle).signatures['default']
```

Finally, some functions to load the image into the model, as well as run the object detector against the model:

```
[ ]  def load_img(path):
       img = tf.io.read_file(path)
       img = tf.image.decode_jpeg(img, channels=3)
       return img
```

```
[ ]  def run_detector(detector, path):
       img = load_img(path)

       converted_img  = tf.image.convert_image_dtype(img, tf.float32)[tf.newaxis, ...]
       start_time = time.time()
       result = detector(converted_img)
       end_time = time.time()

       result = {key:value.numpy() for key,value in result.items()}

       print("Found %d objects." % len(result["detection_scores"]))
       print("Inference time: ", end_time-start_time)

       image_with_boxes = draw_boxes(
           img.numpy(), result["detection_boxes"],
           result["detection_class_entities"], result["detection_scores"])

       display_image(image_with_boxes)
```

This is the result after 2.67 seconds of inference. This is an extremely good time and compared to the 57 seconds I got an hour ago shows the variance in consistency around the free GPU supplied. However, if you have a decent GPU, it should run fairly quickly, definitely within the 30 second range.



It shows the traffic lights, every car, and even the traffic sign (even with low confidence). How an approach like this could be utilized is by using the bounding boxes, and trying to detect if any

squares on the grid of the CAPTCHA contain any bounding boxes. It wouldn't be totally accurate for some versions but incredibly accurate for others.

Due to difficulties with the testing notebook recently I was unable to get a good output from colab so this is what the report is limited to. However, if you wish to explore more options have a look at this link for more models:

https://tfhub.dev/s?q=faster_rcnn