AI User Safety Application

Logo Detection and Screen Capture Script

Designing Specification
Version 0.1

By: Varun Bhaseen
(varun.bhaseen@sjsu.edu)

Project Advisor:
Professor Vijay Eranti

Contents

Version History

Version	Changes
0.1	• An added design approach for Logo Validation using YOLO V3 • Added Screenshot capture using Selenium web driver in Python • Updated Test templates section with Mocking function template • Removed Unit test cases for Selenium web driver

1. Introduction

The purpose of our project is to identify phishing websites by validating the URL, Logo on each webpage. Here I will present the functionality related to Logo Validation using the deep learning model YOLO and the script that is being used for screenshot capturing of any given webpage. The deep learning model used for logo validation is YOLO V3 and YOLO V4. The models are trained on the ImageNet dataset. The approach selected for retraining the YOLO model to detect the logo is called **Transfer Learning.** YOLO is an object detection deep learning model. Its primary purpose in our project is to detect a logo on a webpage and share the confidence score on how accurate the logo is to an actual logo of a target brand or company.

2. Requirements

For designing the logo validation model following are the key requirements that we need:

a. A pre-trained YOLO model on ImageNet. Here pre-trained means we need a model with weights in form of either .h5 file (hierarchical data format), .ckpt file (checkpoint file), or .pkl file (pickle file).
b. HPC (High-Performance Computing) environment is required to unfreeze and retrain the YOLO model on our dataset
c. Jupyter Notebook and Anaconda environment with all the standard deep learning libraries preinstalled
d. Visual studio code, chrome driver, and Google Chrome browser to run and simulate scripts for the capturing of a webpage as a screenshot from a URL.
e. Using Logodet dataset to generalize the YOLO model across multiple target brands and the dataset is in XML
f. Chrome web driver for Selenium web driver library to manage and control the chrome web browser to take screenshots for any web page using a URL
g. A well-annotated Dataset for training the model. This was manually carried out and compiled by my teammate Gulnara Timokhina using the hyper-label application. I built a sample dataset using VOTT and way back machine (Archive.org) webpages for amazon only.

3. References

References to other documents (architecture, figures, URLs, …) IEEE reference
Below are the references for pre-trained YOLO V3 model, YOLO model architecture, and Dataset sources:

a. YOLO V3 model Keras: https://github.com/qqwweee/keras-yolo3
b. YOLO V4 model PyTorch: https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/
c. Dataset: Logodet
d. Converting the dataset XML to YOLO txt format: https://roboflow.com/convert/pascal-voc-xml-to-yolov4-pytorch-txt
e. Selenium web driver architecture for JavaScript in Python: https://pythonbasics.org/selenium-execute-javascript/

4. Functional Overview

Currently, I am working on two functional areas, first is the YOLO model for logo detection (Keras and PyTorch framework) and a Screenshot of a webpage from URL using Selenium. Below is the detailed functionality and architecture for respective functional areas:

4.1. Logo Detection using YOLO

4.1.1. Functionality Architecture

YOLO stands for "You Only Look Once". YOLO is a deep learning model for object detection in an image or a video. We are using YOLO in our project to detect the Logo of a target brand or company on a given web page as an object.

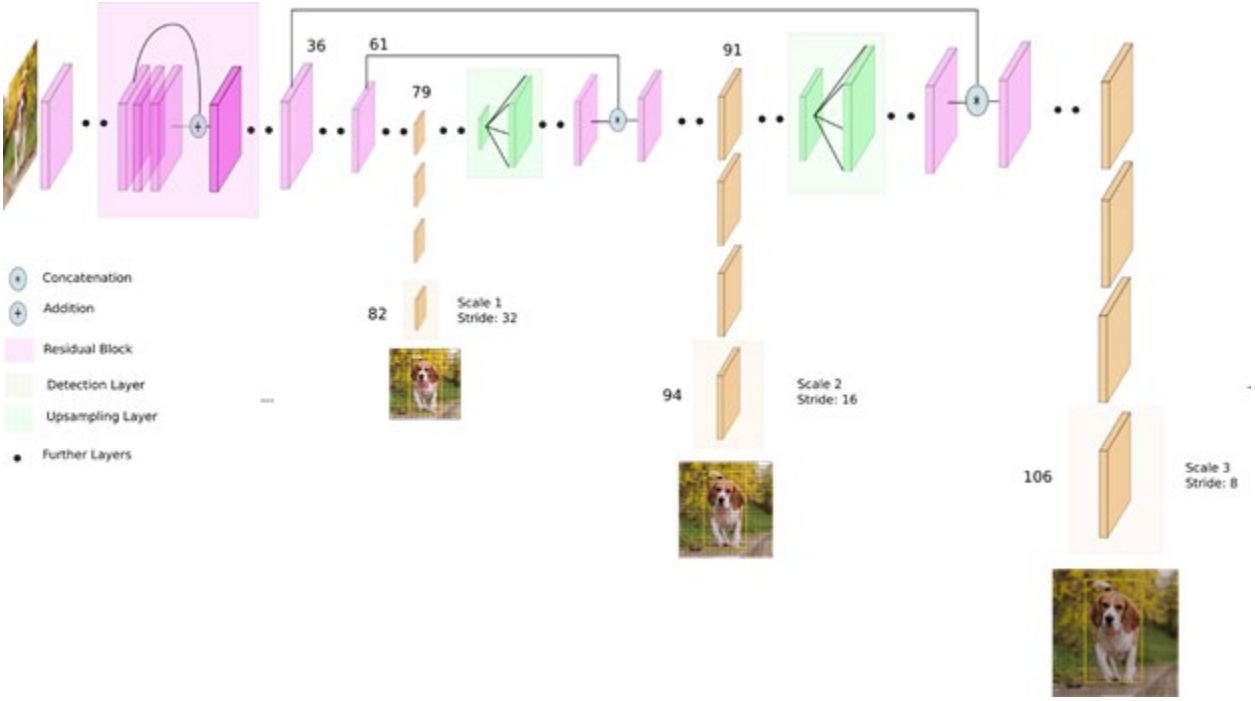

Figure 1 YOLO V3 Model Architecture (Source: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b)

4.1.2. Configuration/ External Interfaces

a. YOLO needs an input for any dataset in the format: image_name; xmin; ymin; xmax; ymax; label
b. Annotating each image manually in the above format using VOTT or Hyper-Label
c. Label each image as a .txt format (Hence x number of images will have x number of .txt label files)
d. Tensorboard or Tensorboardx plugin for visualization
e. Setting up CUDA API in an HPC environment (currently colab pro and RTX 2080) for training

4.1.3. Debug

4.1.3.1. Logging

a. Model performance logging will be evaluated using Tensorboard or Tensorboardx at each epoch at each timestamp.

4.1.3.2. Model Analysis and Tuning

a. *Evaluator (proposed)*: We are planning to use evaluator from TFX (TensorFlow Extended) to

analyze and validate the models and check the performance.

b. *Dataset Splitting*: We will split the dataset into training, validation, and testing as suggested in the workbook
c. *Metrics*: Since the model is primarily doing classification, we are planning to use F1, recall, precision, accuracy as metrics to evaluate model performance
d. *What*-If *tool (proposed)*: we are planning to use something like a what-if tool to further analyze the performance of the overall ensemble model that we will generate using this tool provided by Google.
e. *ModelValidator TFX (Proposed)*: We may be using the TFX component ModelValidator to validate the model.

4.1.3.3. Model Deployment

a. *TF Lite (Tensorflow Lite)*: The final model will be exported to a TF lite framework to work in a web browser Chrome

4.2. Screen capture using selenium

4.2.1. Functionality

Currently, as a prototype before we move to the production version, I have developed a script related to screen capture in chrome from a URL. Selenium web driver takes JavaScript within a python code for managing a web browser and web content. We can use multiple combinations of JavaScript functions within python to control the web-based content

4.2.2. Configuration/ External Interfaces

a. Selenium web driver needs to access the binaries for a web browser to control and manage the browser.
b. I have provided a Chrome Web Driver within the project path for Selenium to use and control the chrome browser independently

5. Implementation

5.1. YOLO Implementation for Logo Validation

5.1.1. Development and testing environment

For the development and testing environment following are the key tasks that are undertaken for YOLO implementation:

a. Annotate the dataset in the standard structure required for YOLO model training and split the dataset accordingly.
b. Implement the YOLO model and carry out Transfer learning for both Keras and Pytorch frameworks
c. Use TFX wherever possible to ensure further deeper management for different layers for the deep learning model
d. Using Tensorboard or Tensorboardx for evaluation and validating the performance of the model against the various metrics for training and testing data

e. Tune the model for generalized performance across different target brands and take appropriate measures to reduce overfitting and underfitting (like regularization, early stopping, etc.)
f. Create an ensemble model of YOLO and LSTM for logo validation and URL validation respectively and evaluate the performance
g. Save the model as a .h5 file (hierarchical data format file) with all weights after the training and validations
h. The implementation approach can be seen below as identified by Tensorflow that we are adopting for the project:

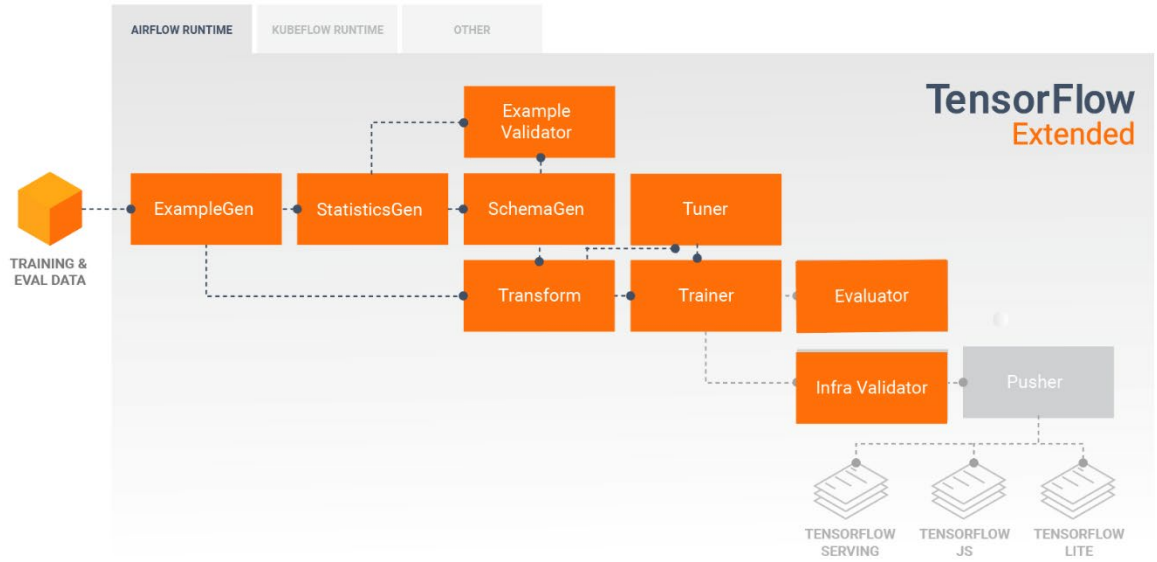

Figure 2: Tensorflow Extended Pipeline architecture for production deployments

5.1.2. Production environment

For the production environment following are the key tasks that need to be considered for model implementation:

a. Convert the TF model into a TF lite model so that it can be used for web browser as suggested in Figure 2

5.2. Screen Capture Implementation

5.2.1. Development and testing environment

Screen capture implementation in the development and testing environment is currently carried out on python using the selenium web driver framework.

5.2.2. Production environment

Translate the approach taken in python to JavaScript for screenshot capture and analysis using deep learning models. Also, ensure that all best practices for logging and debugging are considered.

6. Testing

6.1. Unit tests

6.1.1. Logo Validation Model Unit Testing approach

ID	MLDL001
Functionality	Logo Validation
Model	YOLO
Technique	Transfer Learning
Data Format Input	Images
Dataset	Self-Annotated Logo Dataset
Sources	• LogoDet • Archive.org • Google Image Search Crawling
Volume	100 samples per target brand (Total 1000 Images)
Data Debugging	• Train: 80% • Validation: 20% of Train • Test: 20%
Model Metrics	• Confidence Score • Accuracy
Model Evaluation	• TFX Evaluator • TFX ModelValidator

Model Debugging	• Data Normalization
	• Hidden layer weight check-in Tensorboard graphs
	• Adjust Hyperparameter Values:
	  ○ Learning Rate
	  ○ Regularization
	  ○ Batch size
	  ○ Depth and width of layers

6.2. Testing Script templates

Tensorflow Unit test template	`import TensorFlow as tf`
	
	`class UnetTest(tf.test.TestCase):`
	
	`  def setUp(self):`
	`  super(UnetTest, self).setUp()`
	`  . . .`
	
	`  def tearDown(self):`
	`  pass`
	
	`  def test_normalize(self):`
	`  . . .`
	
	`if __name__ == '__main__':`

	`tf.test.main()`

Mocking functions template	`def load_data(self):`
	`  """ Loads and Preprocess data """`
	`  self.dataset, self.info = DataLoader().load_data(self.config.data)`
	`  self._preprocess_data()`
	
	`def _preprocess_data(self):`
	`  """ Splits into training and test and set training parameters"""`
	`  train = self.dataset['train'].map(self._load_image_train, num_parallel_calls=tf.data.experimental.AUTOTUNE)`
	`  test = self.dataset['test'].map(self._load_image_test)`
	
	`  self.train_dataset = train.cache().shuffle(self.buffer_size).batch(self.batch_size).repeat()`
	`  self.train_dataset = self.train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)`
	`  self.test_dataset = test.batch(self.batch_size)`
	
	`def _load_image_train(self, datapoint):`
	`  """ Loads and preprocess  a single training image """`
	`  input_image = tf.image.resize(datapoint['image'], (self.image_size, self.image_size))`
	`  input_mask = tf.image.resize(datapoint['segmentation_mask'], (self.image_size, self.image_size))`
	
	`  if tf.random.uniform(()) > 0.5:`
	`  input_image = tf.image.flip_left_right(input_image)`
	`  input_mask = tf.image.flip_left_right(input_mask)`
	
	`  input_image, input_mask = self._normalize(input_image, input_mask)`
	
	`  return input_image, input_mask`
	
	`def _load_image_test(self, datapoint):`
	`  """ Loads and preprocess a single test image"""`
	
	`  input_image = tf.image.resize(datapoint['image'], (self.image_size, self.image_size))`
	`  input_mask = tf.image.resize(datapoint['segmentation_mask'], (self.image_size, self.image_size))`
	
	`  input_image, input_mask = self._normalize(input_image, input_mask)`
	
	`  return input_image, input_mask`

ModelValidator	`from tfx import components`
	`import tensorflow_model_analysis as tfma`
	
	`...`
	
	`# For TFMA evaluation`
	
	`eval_config = tfma.EvalConfig(`

```
    model_specs=[
        # This assumes a serving model with signature
'serving_default'. If
        # using estimator based EvalSavedModel, add
signature_name='eval' and
        # remove the label_key. Note, if using a TFLite model, then
you must set
        # model_type='tf_lite'.
        tfma.ModelSpec(label_key='<label_key>')
    ],
    metrics_specs=[
        tfma.MetricsSpec(
            # The metrics added here are in addition to those saved
with the
            # model (assuming either a keras model or EvalSavedModel
is used).
            # Any metrics added into the saved model (for example
using
            # model.compile(..., metrics=[...]), etc) will be
computed
            # automatically.
            metrics=[
                tfma.MetricConfig(class_name='ExampleCount'),
                tfma.MetricConfig(
                    class_name='BinaryAccuracy',
                    threshold=tfma.MetricThreshold(
                        value_threshold=tfma.GenericValueThreshold(
                            lower_bound={'value': 0.5}),
                        change_threshold=tfma.GenericChangeThreshold
(
                            direction=tfma.MetricDirection.HIGHER_IS
_BETTER,
                            absolute={'value': -1e-10})))
            ]
        )
    ],
    slicing_specs=[
        # An empty slice spec means the overall slice, i.e. the
whole dataset.
        tfma.SlicingSpec(),
        # Data can be sliced along a feature column. In this case,
data is
        # sliced along feature column trip_start_hour.
```

	tfma.SlicingSpec(feature_keys=['trip_start_hour'])])

Evaluator	import tensorflow_model_analysis as tfma
	
	output_path = evaluator.outputs['evaluation'].get()[0].uri
	
	# Load the evaluation results.
	eval_result = tfma.load_eval_result(output_path)
	
	# Visualize the metrics and plots using
	tfma.view.render_slicing_metrics,
	# tfma.view.render_plot, etc.
	tfma.view.render_slicing_metrics(tfma_result)
	...
	
	# Load the validation results
	validation_result = tfma.load_validation_result(output_path)
	if not validation_result.validation_ok:
	  ...

7. Appendix Code Snippets

7.1. Code snippet: Screenshot using Selenium

NOTE: This is not complete code. Entire code is more than 150 lines

```
import os
from subprocess import Popen, PIPE
from selenium import webdriver

abspath = lambda *p: os.path.abspath(os.path.join(*p))
ROOT = abspath(os.path.dirname(__file__))
DRIVER = 'C:/Users/bhase/Downloads/chromedriver'

def execute_command(command):
    result = Popen(command, shell=True, stdout=PIPE).stdout.read()
    if len(result) > 0 and not result.isspace():
        raise Exception(result)


def do_screen_capturing(url, screen_path, width, height):
    print ("Capturing screen..")
    driver = webdriver.Chrome(DRIVER)
```

```
    # it save service log file in same directory
    # if you want to have log file stored else where
    # initialize the webdriver.PhantomJS() as
    # driver = webdriver.PhantomJS(service_log_path='/var/log/phantomjs/ghostdr
iver.log')
    # to maximize the browser window

    if width and height:
        driver.set_window_size(width, height)
    driver.get(url)
    driver.maximize_window()
    driver.set_script_timeout(30)
    driver.execute_script('document.body.style.zoom = "50%"')
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight)")
    driver.save_screenshot(screen_path)


def do_crop(params):
    print ("Croping captured image..")
    command = [
```

7.2. Code snippet: YOLO model for Logo Validation (Keras)

Note: This is not the full notebook just a snippet. Feel free to reach out for complete code.

```
ef _conv_block(inp, convs, skip=True):
  x = inp
  count = 0
  for conv in convs:
    if count == (len(convs) - 2) and skip:
      skip_connection = x
    count += 1
    if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x) # peculia
r padding as darknet prefer left and top
    x = Conv2D(conv['filter'],
           conv['kernel'],
           strides=conv['stride'],
           padding='valid' if conv['stride'] > 1 else 'same', # peculiar
 padding as darknet prefer left and top
           name='conv_' + str(conv['layer_idx']),
           use_bias=False if conv['bnorm'] else True)(x)
    if conv['bnorm']: x = BatchNormalization(epsilon=0.001, name='bnorm_
' + str(conv['layer_idx']))(x)
```

```
    if conv['leaky']: x = LeakyReLU(alpha=0.1, name='leaky_' + str(conv[
'layer_idx']))(x)
  return add([skip_connection, x]) if skip else x

def make_yolov3_model():
  input_image = Input(shape=(None, None, 3))
  # Layer  0 => 4
  x = _conv_block(input_image, [{'filter': 32, 'kernel': 3, 'stride': 1,
 'bnorm': True, 'leaky': True, 'layer_idx': 0},
                 {'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True
, 'leaky': True, 'layer_idx': 1},
                 {'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True
, 'leaky': True, 'layer_idx': 2},
                 {'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True
, 'leaky': True, 'layer_idx': 3}])
  # Layer  5 => 8
  x = _conv_block(x, [{'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm':
 True, 'leaky': True, 'layer_idx': 5},
           {'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'le
aky': True, 'layer_idx': 6},
           {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'le
aky': True, 'layer_idx': 7}])
  # Layer  9 => 11
  x = _conv_block(x, [{'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm':
 True, 'leaky': True, 'layer_idx': 9},
           {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'le
aky': True, 'layer_idx': 10}])
  # Layer 12 => 15
```