Transfer Learning for Custom Datasets in the Small-Data Regime

TEAM - 14

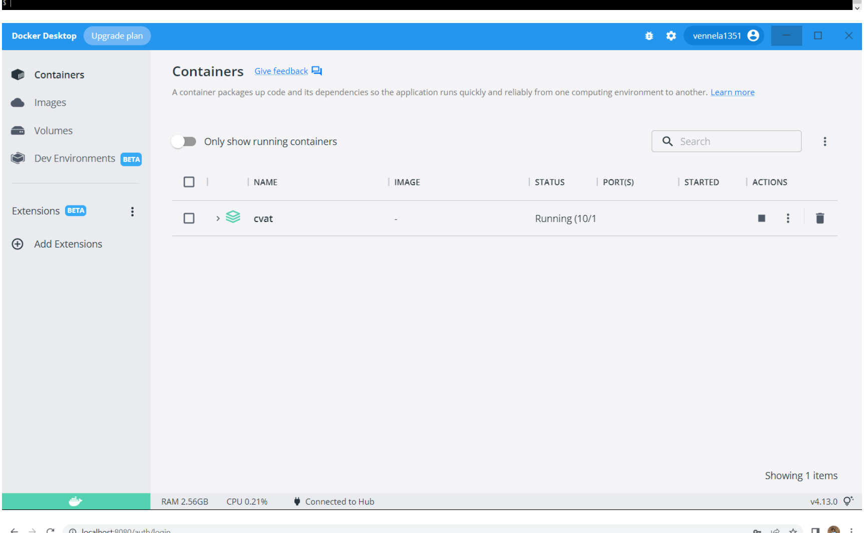
Milestone 1: Environment Preparation

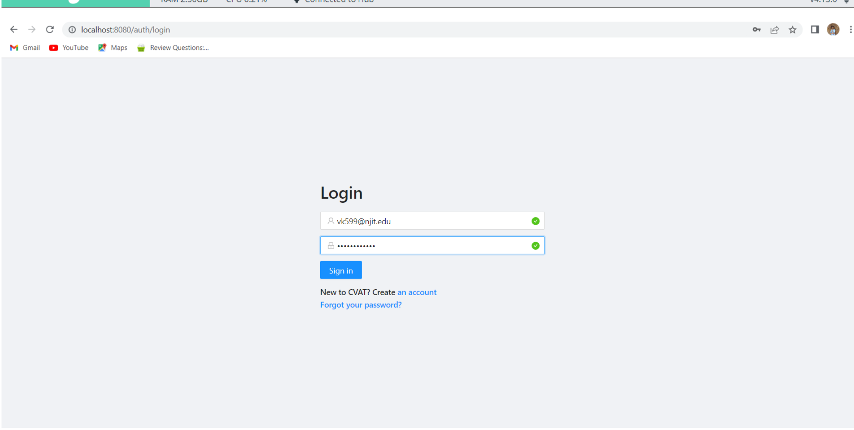
STEPS INCLUDED:

1.install the 'wsl' using wsl --install commond in command prompt/ power shell.

2.Then install the docker window and signup the docker window before that restart the computer.

3.Now install the the gitbash for windows 4.And type the command cvat inorder to open the cvat. the commands are as follows git clone https://github.com/opencv/cvat cd cvat docker-compose up -d winpty docker exec -it cvat_server bash -ic 'python3 ~/manage.py createsuperuser' enter the username and password Make sure the docker window is ruuning parallel. Now we can find that cvat is ruuning and is shiwn in docker window. Installation is done .





Milestone 2: Data Acquisition

1. Create a project on Google Developers Console https://console.developers.google.com
2. Enable Custom Search API and generate API key
3. Create a search engine https://cse.google.com/cse/all
4. Installation

Here the google API is used for Data Acquisition. We generate a key for the data acquistion and start the Acquistion . There are 10 Categories. For each category , we aquire 100 images

Milestone 3: Annotation

Upload the images in CVAT instance and annotate all object categories

Here we take the images and covert them into a dataset having Training and testing data. We Peform Object Segmentation using Deep Extreme Cut (DEXTR). First we create a project in cvat and add the labels to it

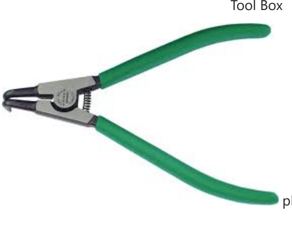
1. The label are named as Wrenches, Screwdrives,Toll box,shleves, wallpanels,pliers,storage cabinets,workbenches,totes,hammer.
2. then we create a task.
3. Now upload a zip file having a 1000 images
4. Now fit the image in the rectangle and assign the image under correct label
5. Repeat the process for all the 1000 images.
6. make sure each label has a 100 images under it.
7. Now create an coount in roboflow
8. Now convert the fitted images into a MScoco file.
9. Download the zip file.
10. The Zip file is the Dataset
11. It contains test,train,valid and readme file and annotation for the images
12. link for roboflow https://universe.roboflow.com/dataminingmilestone3/milestone3-ta1bc

Images of all categories

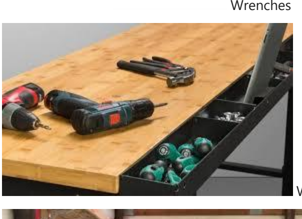
Screwdrivers


Hammer


Tool Box


pliers

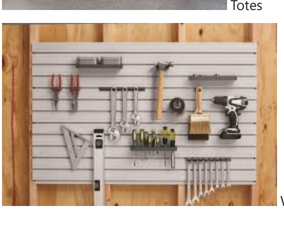
Wrenches


Work Benches


Shelve


Storage Cabinet


Totes


Wall Panels

Milestone 4: Semantic Segmentation

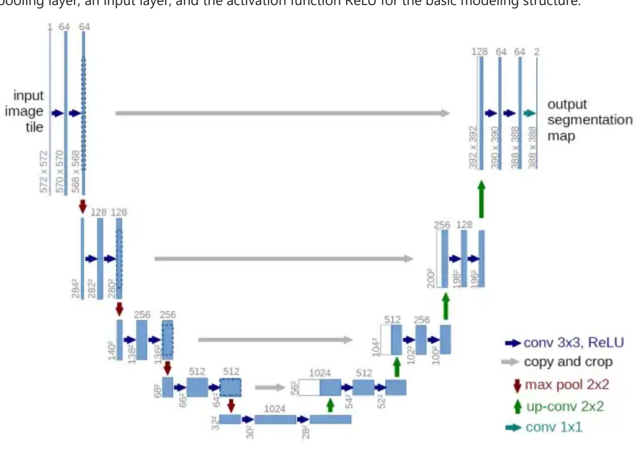
Segmentation Here inorder to perform segmentation , we use UNet on garage dataset.

Image segmentation is the task of partitioning an image into multiple segments.

This makes it a whole lot easier to analyze the given image. Semantic segmentation refers to the process of linking each pixel in an image to a class label.

Constructing an architecture is a naive approach towards constructing a neural network architecture for this task is to simply stack a number of convolutional layers (with same padding to preserve dimensions) and output a final segmentation map. This directly learns a mapping from the input image to its corresponding segmentation through the successive transformation of feature mappings. However, it's quite computationally expensive to preserve the full resolution throughout the network.

For building the U-Net architecture, we will utilize the TensorFlow deep learning framework, as discussed already. Hence, we will import the TensorFlow library for this purpose as well as the Keras framework, which is now an integral part of TensorFlow model structures. From our previous understanding of the U-Net architecture, we know that some of the essential imports include the convolutional layer, the max-pooling layer, an input layer, and the activation function ReLU for the basic modeling structure.



In [ ]:
```
from pycocotools import coco, cocoeval, _mask
from pycocotools import mask as maskUtils
import array
import numpy as np
import skimage.io as io
import matplotlib.pyplot as plt
import pylab
import os
pylab.rcParams['figure.figsize'] = (8.0, 10.0)
%matplotlib inline
```

In [ ]:
```
class DataGen(tf.keras.utils.Sequence):

  def __init__(self , path_input , path_mask , batch_size = 8 , image_size = 128):

    self.ids = os.listdir(path_input)
    self.path_input = path_input
    self.path_mask = path_mask
    self.batch_size = batch_size
    self.image_size = image_size
    self.on_epoch_end()

  def __load__(self , id_name):

    image_path = os.path.join(self.path_input , id_name)
    mask_path = os.path.join(self.path_mask , id_name)

    image = cv2.imread(image_path , 1) # 1 specifies RGB format
    image = cv2.resize(image , (self.image_size , self.image_size)) # resizing before inserting to the network

    mask = cv2.imread(mask_path , -1)
    mask = cv2.resize(mask , (self.image_size , self.image_size))
    mask = mask.reshape((self.image_size , self.image_size , 1))

    #normalize image
    image = image / 255.0
    mask = mask / 255.0

    return image , mask

  def __getitem__(self , index):

    if (index + 1)*self.batch_size > len(self.ids):
      self.batch_size = len(self.ids) - index * self.batch_size

    file_batch = self.ids[index * self.batch_size : (index + 1) * self.batch_size]

    images = []
    masks = []

    for id_name in file_batch :

      _img , _mask = self.__load__(id_name)
      images.append(_img)
      masks.append(_mask)


    images = np.array(images)
    masks = np.array(masks)


    return images , masks


  def on_epoch_end(self):
    pass


  def __len__(self):

    return int(np.ceil(len(self.ids) / float(self.batch_size)))
```

In [ ]:
```
def down_block(
    input_tensor,
    no_filters,
    kernel_size=(3, 3),
    strides=(1, 1),
    padding="same",
    kernel_initializer="he_normal",
    max_pool_window=(2, 2),
    max_pool_stride=(2, 2)
):
    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(input_tensor)

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)
    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(conv)

    conv = BatchNormalization(scale=True)(conv)

    # conv for skip connection
    conv = Activation("relu")(conv)

    pool = MaxPooling2D(pool_size=max_pool_window, strides=max_pool_stride)(conv)

    return conv, pool
```

In [ ]:
```
def bottle_neck(
    input_tensor,
    no_filters,
    kernel_size=(3, 3),
    strides=(1, 1),
    padding="same",
    kernel_initializer="he_normal"
):
    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(input_tensor)

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)

    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(conv)

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)

    return conv
```

In [ ]:
```
def up_block(
    input_tensor,
    no_filters,
    skip_connection,
    kernel_size=(3, 3),
    strides=(1, 1),
    upsampling_factor = (2,2),
    max_pool_window = (2,2),
    padding="same",
    kernel_initializer="he_normal"):


    conv = Conv2D(
        filters = no_filters,
        kernel_size= max_pool_window,
        strides = strides,
        activation = None,
        padding = padding,
        kernel_initializer=kernel_initializer
    )(UpSampling2D(size = upsampling_factor)(input_tensor))

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)


    conv = concatenate( [skip_connection , conv]  , axis = -1)


    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(conv)

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)

    conv = Conv2D(
        filters=no_filters,
        kernel_size=kernel_size,
        strides=strides,
        activation=None,
        padding=padding,
        kernel_initializer=kernel_initializer
    )(conv)

    conv = BatchNormalization(scale=True)(conv)

    conv = Activation("relu")(conv)

    return conv
```

In [ ]:
```
def output_block(input_tensor,
    padding="same",
    kernel_initializer="he_normal"
):

    conv = Conv2D(
        filters=2,
        kernel_size=(3,3),
        strides=(1,1),
        activation="relu",
        padding=padding,
        kernel_initializer=kernel_initializer
    )(input_tensor)


    conv = Conv2D(
        filters=1,
        kernel_size=(1,1),
        strides=(1,1),
        activation="sigmoid",
        padding=padding,
        kernel_initializer=kernel_initializer
    )(conv)


    return conv
```

In [ ]:
```
def UNet(input_shape = (128,128,3)):

    filter_size = [64,128,256,512,1024]

    inputs = Input(shape = input_shape)

    d1 , p1 = down_block(input_tensor= inputs,
                         no_filters=filter_size[0],
                         kernel_size = (3,3),
                         strides=(1,1),
                         padding="same",
                         kernel_initializer="he_normal",
                         max_pool_window=(2,2),
                         max_pool_stride=(2,2))


    d2 , p2 = down_block(input_tensor= p1,
                         no_filters=filter_size[1],
                         kernel_size = (3,3),
                         strides=(1,1),
                         padding="same",
                         kernel_initializer="he_normal",
                         max_pool_window=(2,2),
                         max_pool_stride=(2,2))



    d3 , p3 = down_block(input_tensor= p2,
                         no_filters=filter_size[2],
                         kernel_size = (3,3),
                         strides=(1,1),
                         padding="same",
                         kernel_initializer="he_normal",
                         max_pool_window=(2,2),
                         max_pool_stride=(2,2))



    d4 , p4 = down_block(input_tensor= p3,
                         no_filters=filter_size[3],
                         kernel_size = (3,3),
                         strides=(1,1),
                         padding="same",
                         kernel_initializer="he_normal",
                         max_pool_window=(2,2),
                         max_pool_stride=(2,2))


    b = bottle_neck(input_tensor= p4,
                         no_filters=filter_size[4],
                         kernel_size = (3,3),
                         strides=(1,1),
                         padding="same",
                         kernel_initializer="he_normal")



    u4 = up_block(input_tensor = b,
                  no_filters = filter_size[3],
                  skip_connection = d4,
                  kernel_size=(3, 3),
                  strides=(1, 1),
                  upsampling_factor = (2,2),
                  max_pool_window = (2,2),
                  padding="same",
                  kernel_initializer="he_normal")

    u3 = up_block(input_tensor = u4,
                  no_filters = filter_size[2],
                  skip_connection = d3,
                  kernel_size=(3, 3),
                  strides=(1, 1),
                  upsampling_factor = (2,2),
                  max_pool_window = (2,2),
                  padding="same",
                  kernel_initializer="he_normal")


    u2 = up_block(input_tensor = u3,
                  no_filters = filter_size[1],
                  skip_connection = d2,
                  kernel_size=(3, 3),
                  strides=(1, 1),
                  upsampling_factor = (2,2),
                  max_pool_window = (2,2),
                  padding="same",
                  kernel_initializer="he_normal")


    u1 = up_block(input_tensor = u2,
                  no_filters = filter_size[0],
                  skip_connection = d1,
                  kernel_size=(3, 3),
                  strides=(1, 1),
                  upsampling_factor = (2,2),
                  max_pool_window = (2,2),
                  padding="same",
                  kernel_initializer="he_normal")



    output = output_block(input_tensor=u1 ,
                         padding = "same",
                         kernel_initializer= "he_normal")

    model = Model(inputs = inputs , outputs = output)


    return model
```

In [ ]:
```
model = UNet(input_shape = (128,128,3))
model.compile(optimizer = Adam(lr = 1e-4), loss = 'binary_crossentropy', metrics = ['accuracy'])
```

In [ ]:
```
image_size = 128
epochs = 1
batch_size = 8
```

In [ ]:
```
train_gen = DataGen(path_input = "/content/train2014" , path_mask = "/content/mask_train_2014/" , batch_size =
val_gen = DataGen(path_input =  "/content/val2014", path_mask =  "/content/mask_val_2014", batch_size = batch_s


train_steps =  len(os.listdir( "/content/train2014"))/batch_size;


model.fit_generator(train_gen , validation_data = val_gen , steps_per_epoch = train_steps , epochs=epochs);
```

In [ ]:
```
x, y = val_gen.__getitem__(4)
result = model.predict(x)

result = result > 0.5



fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)

ax = fig.add_subplot(1, 2, 1)
ax.imshow(np.reshape(y[0]*255, (image_size, image_size)), cmap="gray")

ax = fig.add_subplot(1, 2, 2)
ax.imshow(np.reshape(result[0]*255, (image_size, image_size)), cmap="gray")
```

In [ ]: