**Practical Assignment**

**Objective: - Object Detection with Oxford-IIIT Pet Dataset**

Stanford with IIIT have created a 37 category pet dataset with roughly 200 images for each class. The images have a large variations in scale, pose and lighting. All images have an associated ground truth annotation of breed, head ROI, and pixel level trimap segmentation.

**Dataset Link: -**

**Dataset :-**[**https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz**](https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz)

**Ground Truth :-**[**https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz**](https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz)

**Task: -** Create a Web Application using Flask. Use the end user should be able to upload an image and get results with the prediction score. Use any CNN architecture launched after 2017.

**Deployment: -** Any Free Platform(Try to look out for free options.)

**Assignment Submission: -** Only submit the hosted app link. OR GitHub Link

**TensorFlow Tutorial - Transfer Learning for Multi-Class Image Classification**

**Step-by-step guide to using the pre-trained ResNet50 V2 model for image classification of Oxford-IIIT Pets dataset**

**(1) Initial Setup**

* Highly recommended to run this notebook in Google Colab
* All the essential Python libraries we need for this project are already pre-installed in the default Google Colab environment

**import** numpy **as** np

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**import** tensorflow **as** tf

**import** tensorflow\_datasets **as** tfds

**from** tensorflow **import** keras

**from** tensorflow.keras **import** layers

**from** keras **import** callbacks

*# Check Tensorflow version*

tf**.** \_\_version\_\_

## (2) Import Data

* For this project, we will be working with the Oxford-IIIT dataset (<https://www.robots.ox.ac.uk/~vgg/data/pets/>)
* The Oxford-IIIT pet dataset is a 37 category pet image dataset with roughly 200 images for each class. The images have large variations in scale, pose and lighting. All images have an associated ground truth annotation of breed.
* While there are various sources where the image dataset can be downloaded from, we will use the tfds module to load the images.
* tfds stands for Tensorflow Datasets, and it is a Tensorflow component that defines a collection of datasets ready-to-use with TensorFlow

#### Load and split the raw image data

* We load the pets dataset (named as oxford\_iiit\_pet in tfds) as a tf.data.Dataset object, and the raw dataset has two sets: train and test
* For our project, we want to split the dataset into three sets: train set, validation set, and test set.
* The train/val split is set as 90:10 ratio, and we also ensure the images are shuffled
* We can use percentages (%) to slice the datasets, so that there is no need for us to input the exact index for splitting
* Note that when we set the parameter with\_info=True, we will also obtain the dataset documentation along with the images.
* This dataset documentation (which we save as ds\_info) contains a host of information, but perhaps the most important one is the label name (i.e. pet breed) for each image class
* The output will be captured as a tuple: (image datasets, dataset information) when we set as\_supervised=T

(train\_raw, val\_raw, test\_raw), ds\_info **=** tfds**.**load(name**=**'oxford\_iiit\_pet',

split**=**['train[:90%]',

'train[90%:]',

'test'],

shuffle\_files**=True**,

as\_supervised**=True**, *# Returns (image, label)*

with\_info**=True** *# To retrieve dataset info and label*  )

ds\_info

### Display several examples of the image dataset (from the train set) using the in-built tfds.show\_examples function

tfds**.**show\_examples(train\_raw, ds\_info, image\_key**=**'image')

## (3) Understanding the dataset

#### Retrieve relevant counts to better understand the dataset we will be working on

*# Get number of classes*

num\_classes **=** ds\_info**.**features['label']**.**num\_classes

print('Number of classes:', num\_classes)

num\_train\_examples **=** tf**.**data**.**experimental**.**cardinality(train\_raw)**.**numpy()

num\_val\_examples **=** tf**.**data**.**experimental**.**cardinality(val\_raw)**.**numpy()

num\_test\_examples **=** tf**.**data**.**experimental**.**cardinality(test\_raw)**.**numpy()

print('Number of training samples:', num\_train\_examples)

print('Number of validation samples:', num\_val\_examples)

print('Number of test samples:', num\_test\_examples)

**def** get\_value\_counts(ds):

label\_list **=** []

**for** images, labels **in** ds:

label\_list**.**append(labels**.**numpy())

label\_counts **=** pd**.**Series(label\_list)**.**value\_counts(sort**=True**)

print(label\_counts)

get\_value\_counts(train\_raw)

get\_value\_counts(val\_raw)

*# Function to obtain the name for the label integer*

get\_label\_name **=** ds\_info**.**features['label']**.**int2str

*# Build the custom function to display image and label name*

**def** view\_single\_image(ds):

image, label **=** next(iter(ds))

print('Image shape: ', image**.**shape)

plt**.**imshow(image)

\_ **=** plt**.**title(get\_label\_name(label))

view\_single\_image(train\_raw)

**(4) Data Preparation**

**(i) Image Resizing**

* Because the raw images come in different sizes, we want to resize them to the same size before parsing them into the neural network later
* More specifically, we want both the length and width to be 224 pixels (which is what ResNet50 expects)

IMG\_SIZE **=** 224

train\_ds **=** train\_raw**.**map(**lambda** x, y: (tf**.**image**.**resize(x, (IMG\_SIZE, IMG\_SIZE)), y))

val\_ds **=** val\_raw**.**map(**lambda** x, y: (tf**.**image**.**resize(x, (IMG\_SIZE, IMG\_SIZE)), y))

test\_ds **=** test\_raw**.**map(**lambda** x, y: (tf**.**image**.**resize(x, (IMG\_SIZE, IMG\_SIZE)), y)

**(ii) Label One-hot Encoding**

* There are 37 classes (i.e. pet breeds) in the dataset that we are using for multi-class image classification
* As such, we proceed to one-hot encode the labels so that we get a output vector of length 37

**def** one\_hot\_encode(image, label):

label **=** tf**.**one\_hot(label, num\_classes)

**return** image, label

train\_ds **=** train\_ds**.**map(one\_hot\_encode)

val\_ds **=** val\_ds**.**map(one\_hot\_encode)

test\_ds **=** test\_ds**.**map(one\_hot\_encode)

#### Let's have a look at what our dataset object looks like currently

train\_ds

### (iii) Image (Data) Augmentation

* Image augmentation artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc.
* The purpose is to reduce model overfitting by exposing our model to variations and small transformations in the original data. It is useful especially when we do not have a large dataset.
* Note that the image augmentation needs to be realistic. For example, flipping car images upside down may not be the best choice here since we expect most cars to be photographed with the wheels on the ground (barring severe accidents)
* To perform data augmentation, we use the Keras preprocessing layers API. Each type of image augmentation that we want to introduce is defined as a layer within a Keras Sequential class.
* We will only perform a random horizontal flip for the images for the sake of simplicity, though do note that there is a wide range of different augmentations available to us: <https://keras.io/api/layers/preprocessing_layers/image_augmentation/>

data\_augmentation **=** keras**.**Sequential(

[layers**.**RandomFlip('horizontal'),

*# layers.RandomRotation(factor=(-0.025, 0.025)),*

*# layers.RandomTranslation(height\_factor=0.1, width\_factor=0.1),*

*# layers.RandomContrast(factor=0.1),*

#### View effects of augmentation

**for** image, label **in** train\_ds**.**take(1):

plt**.**figure(figsize**=**(10, 10))

**for** i **in** range(4):

ax **=** plt**.**subplot(2, 2, i**+**1)

aug\_img **=** data\_augmentation(tf**.**expand\_dims(image, axis**=**0))

plt**.**imshow(aug\_img[0]**.**numpy()**.**astype("uint8"))

plt**.**title(get\_label\_name(int(label[0])))

plt**.**axis("off")

**(iv) Batching and Prefetching**

* We can batch the data and use prefetching to optimize loading speed and model efficiency
* A batch size of 32 is a good value to start with
* The number of elements to prefetch can be automatically determined by making use of tf.data.AUTOTUNE, which prompt the runtime to tune the value dynamically for us.

BATCH\_SIZE **=** 32

*# Batch the data and use prefetching to optimize loading speed*

*# AVOID use of caching (Google Colab RAM limits)*

train\_ds **=** train\_ds**.**batch(batch\_size**=**BATCH\_SIZE,

drop\_remainder**=True**)**.**prefetch(tf**.**data**.**AUTOTUNE)

val\_ds **=** val\_ds**.**batch(batch\_size**=**BATCH\_SIZE,

drop\_remainder**=True**)**.**prefetch(tf**.**data**.**AUTOTUNE)

test\_ds **=** test\_ds**.**batch(batch\_size**=**BATCH\_SIZE,

drop\_remainder**=True**)**.**prefetch(tf**.**data**.**AUTOTUNE)

## (5) Model Setup

* With the data prepared, it is time to define the model we want to use for the multi-class classification task
* For this task, we will make use of transfer learning so that we do not need to train a deep learning model from scratch (which is tedious and requires plenty of data)
* Keras comes with a host of pre-trained models that we can leverage for transfer learning: <https://keras.io/api/applications/>
* We will be using ResNet50V2 given that it offers a good balance of accuracy, size, and speed
* More information about ResNet can be found here: <https://keras.io/api/applications/resnet/>

### (i) Setup base model

* Instantiate a ResNet50V2 object from keras.applications
* Set include\_top=False because we want to remove the top layers of the pre-trained model (that was trained to classify ImageNet) and introduce our own final layers for our specific car image classification task
* Keep the weights as imagenet, because we want to keep the ResNet weights that were trained on the ImageNet dataset
* This process is known as transfer learning, because we use the pre-trained ImageNet weights

base\_model **=** keras**.**applications**.**ResNet50V2(

include\_top**=False**, *# Exclude ImageNet classifier at the top.*

weights**=**"imagenet",

input\_shape**=**(IMG\_SIZE, IMG\_SIZE, 3)

)

**(ii) Freeze pre-trained weights of the base model**

* When a trainable weight becomes non-trainable, its value is no longer updated during training.

*# Freeze the base\_model*

base\_model**.**trainable **=** **False**

**(iii) Modify inputs**

* We pre-process the inputs (i.e., images) so that they are compatible with what the pre-trained ResNet50v2 architecture expects

*# Create new model on top*

inputs **=** keras**.**Input(shape**=**(IMG\_SIZE, IMG\_SIZE, 3))

x **=** data\_augmentation(inputs)

x **=** keras**.**applications**.**resnet\_v2**.**preprocess\_input(x)

x **=** base\_model(x, training**=False**)

* Although the base model becomes trainable, it is still running in inference mode since we passed training=False when calling it when we built the model.
* This means that the batch normalization layers inside will not update their batch statistics (i.e., mean and variance)
* If they did, they would wreak havoc on the representations learned by the model so far because the training done would have been undone.

**(iv) Rebuild top layers**

* Given that we have removed the top ImageNet classifier layer of ResNet50V2, we can now build a custom top layer that is specific to our image classification task i.e., classify an image based on the 37 different pet breeds types.

*# Rebuild top layers*

x **=** layers**.**GlobalAveragePooling2D()(x)

x **=** layers**.**BatchNormalization()(x)

x **=** layers**.**Dropout(0.2)(x) *# Regularize with dropout*

outputs **=** layers**.**Dense(num\_classes, activation**=**"softmax", name**=**"pred")(x)

model **=** keras**.**Model(inputs, outputs)

#### Display model summary

model**.**summary()

#### Compile model

* Utilize the commonly used Adam optimizer (leave the learning rate as default)
* Since it is a multi-class classification, we will use categorical cross entropy and categorical accuracy as our loss and performance metric respectively

model**.**compile(optimizer**=**keras**.**optimizers**.**Adam(),

loss**=**keras**.**losses**.**CategoricalCrossentropy(),

metrics**=**[keras**.**metrics**.**CategoricalAccuracy()]

)

#### Include early stopping

* Prevent overfitting

earlystopping **=** callbacks**.**EarlyStopping(monitor**=**'val\_loss',

mode**=**'min',

patience**=**5,

restore\_best\_weights**=True**)

## (6) Model Training

#### Fit model

* Ensure that the Google Colab runtime using GPU hardware accelerator
* Fit the model over the training dataset for 15 epochs
* Store the training output as a variable called history

EPOCHS **=** 25

history **=** model**.**fit(train\_ds, epochs**=**EPOCHS, validation\_data**=**val\_ds, verbose**=**1,

callbacks **=**[earlystopping])

#### Plot the accuracy of training and validation sets over epochs

**def** plot\_hist(hist):

plt**.**plot(hist**.**history['categorical\_accuracy'])

plt**.**plot(hist**.**history['val\_categorical\_accuracy'])

plt**.**title('Categorical accuracy')

plt**.**ylabel('accuracy')

plt**.**xlabel('epoch')

plt**.**legend(['train', 'validation'], loc**=**'upper left')

plt**.**show()

plot\_hist(history)

**(7) Model Evaluation**

preds **=** model**.**predict(test\_ds)

#### Evaluate model on test dataset

result **=** model**.**evaluate(test\_ds)

#### Display results

dict(zip(model**.**metrics\_names, result))

## (8) Finetune Model (OPTIONAL)

* Once our model has converged on the new data, we can try to unfreeze all or part of the base model and retrain the whole model end-to-end with a very low learning rate (e.g., 0.0001)
* This is an optional last step that can potentially give us incremental improvements. However, it can also potentially lead to quick overfitting.

#### Unfreeze base model

* We only want to unfreeze top 15 layers that are NOT batch normalization layers

**for** layer **in** model**.**layers[**-**15:]:

**if** **not** isinstance(layer, layers**.**BatchNormalization):

layer**.**trainable **=** **True**

#### Recompile model

* This step needs to be done to take into account the change above, where we set the trainable attribute of the base model layers to True

model**.**compile(optimizer**=**keras**.**optimizers**.**Adam(learning\_rate**=**1e-5), *# Set a very low learning rate*

loss**=**keras**.**losses**.**CategoricalCrossentropy(),

metrics**=**[keras**.**metrics**.**CategoricalAccuracy()]

)

#### Retrain model

* Using fewer epochs given the high risk of overfitting in fine-tuning

EPOCHS **=** 5

history\_2 **=** model**.**fit(train\_ds,

epochs**=**EPOCHS,

validation\_data**=**val\_ds,

verbose**=**1,

callbacks **=**[earlystopping])

#### Display fine-tuned model results

result\_2 **=** model**.**evaluate(test\_ds)

dict(zip(model**.**metrics\_names, result\_2))