Machine Learning Engineer Nanodegree

Capstone Project

Jonathan Kolbrak
January 24, 2021

I. Wildlife Image Classifier Project Overview

Wildlife cameras, Figure 1, are common for determining where animals live, travel or for counting their numbers in an area. These cameras typically take pictures when triggered by a motion sensor. The images are then either stored on a memory card or sent to the customer via a cellular network. These cameras take pictures of anything that triggers the motion sensor. This can result in large amounts of wasted storage space or wasted data usage sending picture back to the user. A wildlife image classifier is proposed for the purpose of cataloging the images for faster viewing later, based on animal type. Alternatively, this could be used to identify images of interest to send to the user.

Figure 1 - Wildlife Camera

Microsoft used this kind of technology to detect snow leopards from Asia in a video. https://snowleopards.microsoft.com/. Other examples are in [1] and [2].

Computer Vision (CV) has many applications. This project is an example of a classification problem. Other examples of computer vision applications are, from [3]:

- Image classification – Identify if something is anywhere in an image
- Image classification with localization – Identify where in an image something is located
- Object detection – This is similar to localization but often contains images where there are more than one object type.
- Object segmentation – Tries to identify the pixels that belong to that object
- And many others.

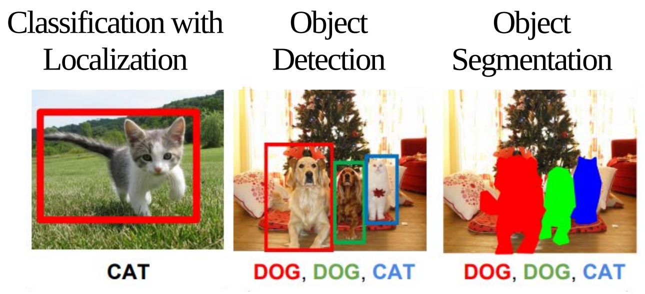


Figure 2 - Computer Vision Application Examples

Figure 2 is examples CV applications from http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf

Project Problem

This project trains an image classier to identify wildlife common in the state of Oregon. The model was trained using transfer learning from an existing model, such as MobileNetV2, against the Oregon Wildlife [4] dataset. The original dataset of MobilNet does not contain categories specifically trained to identify these specific types of animals. The base model was trained with a new classifier layer on the output corresponding to the 20 new categories in the dataset. Then some layers of the base model were retrained with the new dataset, called fine tuning, to improve the classification accuracy.

Tasks

The following are some of the tasks required to complete the development of the wildlife classifier:
1. Obtain and preprocess the dataset from Oregon Wildlife dataset.
2. Perform data cleaning and augmentation, such as scaling, rotation, mirror, etc.
3. Decide upon a base model, MobileNetV2, to use as a basis for transfer learning.
4. Set up and train a new model using transfer learning, Figure 3, using TensorFlow.
5. Compare the classification percentage, of the test dataset, for the transfer learning model with that of the base model.

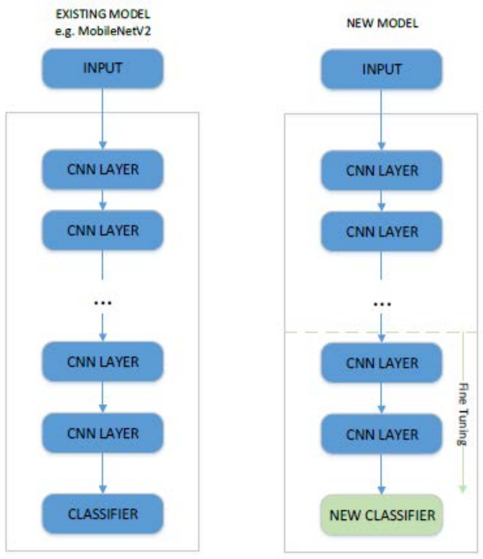


Figure 3 - Transfer Learning

Metrics

The performance of the new model will be based on the image classification accuracy. The baseline performance is the MobilNetV2 model with a new classifier layer. This is then compared to the new model, with fine tuning, to judge improvement in the model performance.

II. Analysis

Data Exploration

The dataset used in this project is from [4] and is a collection of images that represent some of the common animals found in the state of Oregon on kaggle.com. The data is organized into 20 folders of images. Each of the 20 categories of images contains roughly the same amount of images, see Figure 4. The images are of random sizes. Re-sizing of the images will need to be preformed in order to input the images into the MobilNetV2 model.

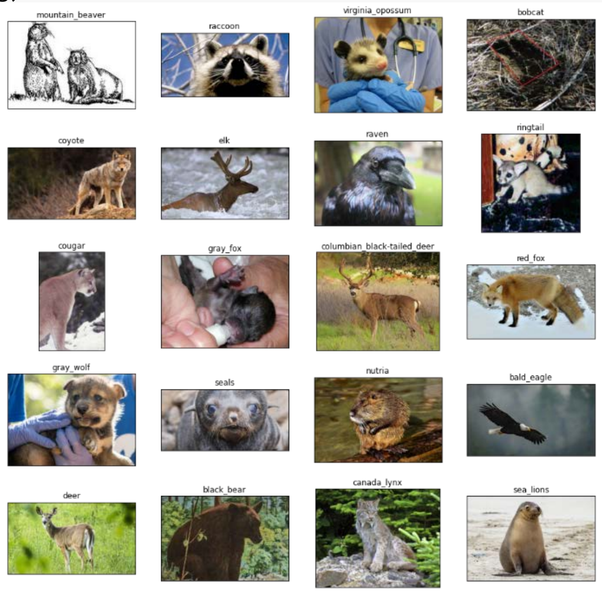
```
PATH = '../input/oregon-wildlife/oregon_wildlife/oregon_wildlife/'
animal_list = os.listdir(PATH)

for animal in animal_list:
    number = len(os.listdir(PATH+animal))
    print('There are ', number, animal, 'images')
print('There are ', len(animal_list), 'total categories')

There are  577 mountain_beaver images
There are  728 raccoon images
There are  728 virginia_opossum images
There are  696 bobcat images
There are  736 coyote images
There are  660 elk images
There are  656 raven images
There are  588 ringtail images
There are  680 cougar images
There are  668 gray_fox images
There are  735 columbian_black-tailed_deer images
There are  759 red_fox images
There are  730 gray_wolf images
There are  698 seals images
There are  701 nutria images
There are  748 bald_eagle images
There are  764 deer images
There are  718 black_bear images
There are  717 canada_lynx images
There are  726 sea_lions images
There are  20 total categories
```

Figure 4 - Number of Images per Category

It is noted that there are duplicate images in the dataset. In order to avoid overfitting because of the duplicates, random transformations of random flips and random rotations were applied to the images. The alternate would be to create a new smaller dataset and remove the duplicate images. This approach was not done due to time constraints.

Finally, a sample image of each category was loaded in order to check the validity of each category. A manual check of files in kaggle was also performed. Note: images here are not sized as in the notebook.

```python
...
fig=plt.figure(figsize=(16, 16))
columns = 4
rows = 5
i=0
for animal in animal_list:
    i+=1
    file = os.listdir(PATH+animal)[0]
    img = Image.open(PATH+animal+'/'+file)
    fig.add_subplot(rows, columns, i, xticks=[], yticks=[])
    plt.title(animal)
    plt.imshow(img)
```



Algorithms and Techniques

This problem is intended to be solved through the use of transfer learning. Transfer learning leverages the advanced training of complex models that have been trained on large datasets. The concept of transfer learning follows the below sequence of steps:

1 Obtain the layers of the previously trained model, with the exception of possibly the top layers.
2 Freeze the layers of the existing model imported in step 1.
3 Add some new layers onto the frozen ones that turn the output into a classifier for the new dataset.
4 Train the new model (frozen + new classifier) on the new dataset.
5 (Optional) Unfreeze some, or all, of the original model layers and retrain the entire model on the new dataset. This step is referred to as fine tuning.

The following code illustrates the steps taken to import the moblinet_v2 model, the preprocessing steps and the downloading of the model layers and weights.

```
from tensorflow.keras import layers
#The next step process the input data to be between [1,1] as required by the model
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input

# Create the base model from the pre-trained model MobileNet V2
# Note the 'include_top=FALSE' as creating a new output layer
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE, include_top=False
, weights='imagenet')

Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/mobilen
et_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5
9412608/9406464 [==============================] - 0s 0us/step

# Here the new output layers are added to the base model
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
prediction_layer = tf.keras.layers.Dense(num_classes, activation='softmax', name='predi
ction')
prediction_batch = prediction_layer(feature_batch_average)
```

In order for the model to be trained on the new dataset, some preprocessing must be done. This consists of: an augmentation step that randomly flips and rotates images. This is how duplicate images in the dataset were addressed. Then we preprocess the image to the correct shape (in this case 224, 224, 3).

```
data_augmentation = tf.keras.Sequential([
tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
tf.keras.layers.experimental.preprocessing.RandomRotation(0.2),
])

inputs = tf.keras.Input(shape=(img_height, img_width, 3))
x = data_augmentation(inputs)
x = preprocess_input(inputs)
x = base_model(x)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.models.Model(inputs, outputs)
```

Benchmark

The benchmark was the performance of the MobileNetV2 with transfer learning of a new output classifier but without fine tuning, which we'll call the base model. After training, the base model obtained a validation dataset accuracy of 85.47% and a test dataset accuracy of 86.21%. These two metrics provide the benchmark to compare the fine tuned model against.

III. Methodology

Data Preprocessing and Implementation

As an image classification model, there are certain steps that need to be addressed.

The first step is to define in the code certain parameters that are necessary for correct training, such as the image size necessary for the model, as well as the size of the batches of images to feed into the model:

```
#Set the batch size, and image height and width
batch_size = 32
img_height = 224
img_width = 224
IMG_SIZE = (img_height, img_width)
```

Next, since the images are stored in folders with the names of the desired categories the keras function image_dataset_from_directory() was used to create the training dataset and validation datasets.

```
#Generate the training dataset
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  PATH,
  validation_split=0.2,
  subset="training",
  label_mode='int',
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
Found 14013 files belonging to 20 classes.
Using 11211 files for training.

#Generate the validation dataset
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  PATH,
  validation_split=0.2,
  subset="validation",
  label_mode='int',
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
Found 14013 files belonging to 20 classes.
Using 2802 files for validation.
```

The final step was to create a test dataset from the validation dataset. In this case, we are finding the cardinality, tf.data.experimental.cardinality, of the validation dataset (val_ds). Then create a test dataset from that by using the take tf.data.Dataset.take() method and 20% (val_batches // 5) where the '//' is a floor division. Then take the unused elements of val_ds to remain in the val_ds.

In order to train the model as quickly as possible, the buffering and pre-fetching of the data is setup as follows, where AUTOTUNE allows the buffer size to be dynamically optimized:

```python
AUTOTUNE = tf.data.experimental.AUTOTUNE  #This sets up the runtime to dynamically tune
 the buffersize

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Now that all the housekeeping items have been setup, the data processing steps begin. Here sequential keras layers are created which will implement RandomFlips but only the horizontal direction. This makes sense to only use the horizontal direction as all the images will be of animals which, unless the camera turns over, should rarely, if ever, be upside down. Then random rotation of the image of factor=0.2 results in an output rotating by a random amount in the range [-20% * 2pi, 20% * 2pi].

```python
data_augmentation = tf.keras.Sequential([
tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
tf.keras.layers.experimental.preprocessing.RandomRotation(0.2),
])

inputs = tf.keras.Input(shape=(img_height, img_width, 3))
x = data_augmentation(inputs)
x = preprocess_input(inputs)
x = base_model(x)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.models.Model(inputs, outputs)
```

Next, the model is compiled which configures the model in preparation for training:

```python
base_learning_rate = 0.0001
model.compile(optimizer=tf.keras.optimizers.Adam(lr=base_learning_rate),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

The following step is to fit the model to the data. The base model case 10 epoch were used to fit the new output classifier to the data. A convenient way to see the progress of the model is to plot the Accuracy and Loss curves of the training and validation datasets, Figure 5.

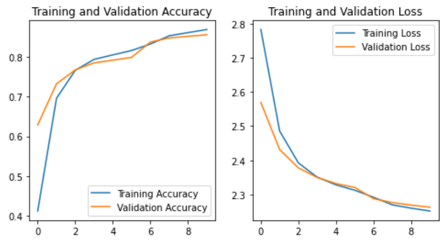


Figure 5 - Base Model Accuracy and Loss

After this round of training reasonable results were obtained of 86% on the test dataset. The goal is to implement a further round of training called fine tuning to improve upon these results.

Refinement

The first thing to do in the fine tuning is to set the base model as trainable and then set the layers to be frozen as non-trainable. In this case the last two ‘blocks’ of the MobileNetV2 model were set as trainable.

```python
base_model.trainable = True
num_layers = len(base_model.layers)
print('Number of layers in the base model: ', num_layers)
fine_tune_at = num_layers-16 #this will unfreeze the block15 and block16
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False
```

After the desired layers are set to trainable the model is recompiled. One important point is that the training rate of the fine tune model was set to 10% of the training rate of the original model. This was done in order to prevent overfitting of the model. Finally, the model with the trainable blocks is retrained with the same code as previously used.

This resulted in an improvement in the classification accuracy from 86% on the test dataset to 92.6% after fine tuning.

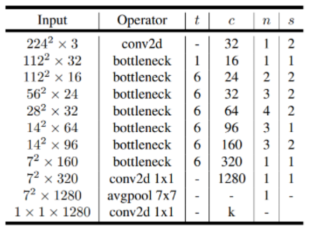
IV. Results

Model Evaluation and Validation

The MobileNetV2 was chosen as the base model for its robustness and efficiency. The goal of an application such as this would be to deploy it an edge or mobile device with limited compute power for inference. As such, this model would be preferred over MobileNetV1, YOLOv2 for its fewer parameters and Madds (multiply-adds), while retaining comparable accuracy [6][7].

The following things were tried on the model in order to test their impact on the performance:

1. Enabling and disabling the data augmentation layer. This was the layer that introduced random flips and rotations to the data. The result was that the training accuracy was essentially the same.
2. The base model consists of blocks, see Figure 6, the fine tuning step was performed with two variations. First was fine tuning of only block 16, and the second was fine tuning block 15 and block 16. The accuracy improved by fine tuning both blocks.
3. Finally, the resizing of the input images was varied. Initially, the images were scaled to a size of 160x160 to try and speed up the training time. Later, once the data flow was established the sizing of the images was increased to 224x224. This resulted in about a 1% improvement in accuracy. It is expected that by shrinking the image so small too much of the desired object was lost.



Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 6 - MobileNetV2 Conv Block

Some analysis was done to explore the functionality of the model and the layers. Feature maps of a layer were generated. It was interesting to do this multiple times and note the distinction as the early layers found more specific shapes, i.e. lines. In deeper layers, the feature maps became more generalized and while important to the model less informative to the human as readable information about the performance of the model. A feature map was generated that contained an output at the end of Block 1.

```python
#Here a single image is loaded in order to generate the feature maps
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from numpy import expand_dims
animal = 'elk'
file = os.listdir(PATH+animal)[3]
#Load a sample image and size resize to the shape expected by the model (img_height, im
g_width)
img = load_img(PATH+animal+'/'+file, target_size=(img_height, img_width))
#Convert to an array
img = img_to_array(img)
#Expand dimenstion so that it represents a single 'sample
img = expand_dims(img, axis=0)
#Prepare the imgage for Mobilnet
img = preprocess_input(img)
#This will process the single image loaded above throught the base model
feature_maps = feature_model.predict(img)
```

The above code when plotted with matplotlib (not shown) results in feature maps show below in, Figure 7, for an image of an Elk .

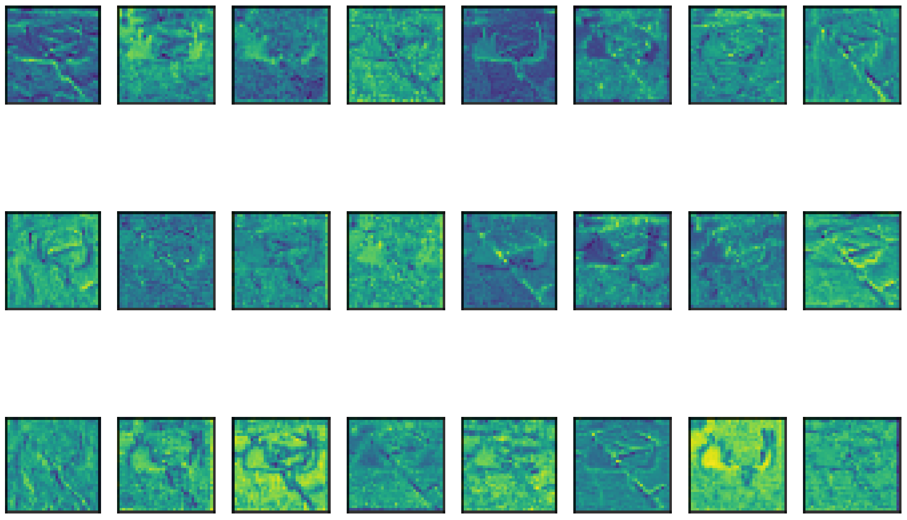


Figure 7 - Elk Feature Maps

Justification

The base model achieved reasonable results of 86.2% on the test dataset. In an application where all inference would be done on battery power, the improved performance is desired in order to keep the platform fielded as long as possible without maintenance.

This is a summary of the training of the base model and trained model:

Model	Training Accuracy	Validation Accuracy	Test Accuracy
Base Model 160x160	86.4 %	85.7 %	87.8 %
Fine Tuned 160x160	93.7 %	90.6 %	90.4 %
Base Model 224x224	86.8 %	85.5 %	86.2 %
Fine Tuned 224x224	96.5 %	92.9 %	92.6 %

V. Conclusion

The final visualization of a confusion matrix was created that clearly illustrates how well the model does for different image categories. Here we see the expected high accuracy along the diagonal. It is also not surprising that the model had more difficulty distinguishing between a “deer” and a “columbian_black-tailed_deer,” as the differences between the two are very small. Even the “nutria” vs “mountain_beaver” error is reasonable as a nutria looks like a beaver without a flat tail. If there was a particular case that had a higher error, further investigation into the source data may be necessary to determine if something was incorrect.

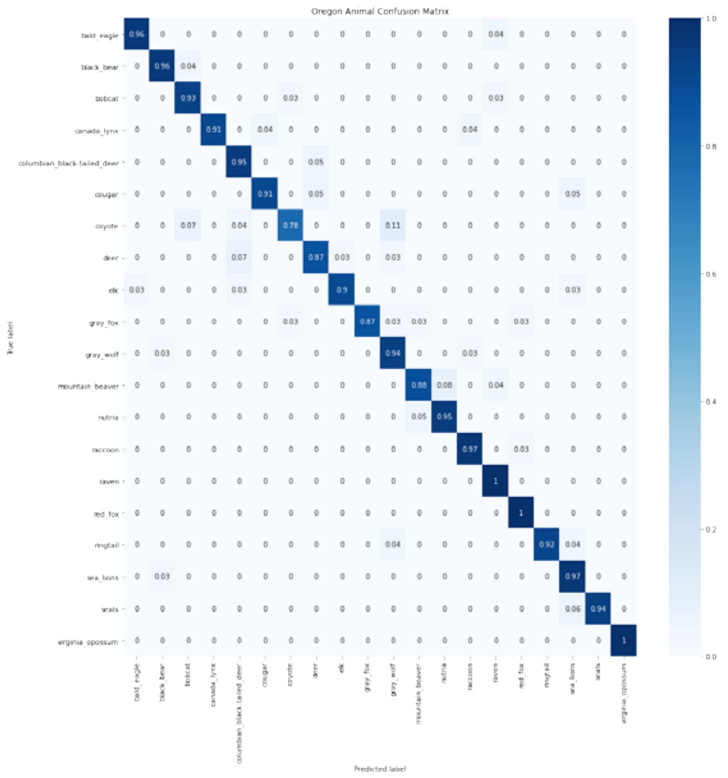
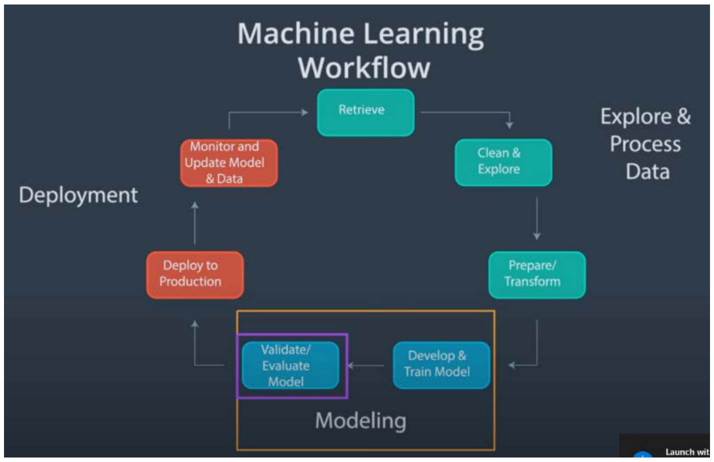


Figure 8 - Oregon Image Classifier Confustion Matrix

The machine learning work flow presented in the course is shown below. In this project all points of the work flow were implemented except the deployment steps.



One of the more interesting aspects of the project was the generation of the feature maps. It was very interesting to see features that were identified in the early layers, where you can still identify the starting image but then progress to more abstract features.

The most challenging aspect of the project was definitely getting the process in TensorFlow/Keras and figuring out how to implement the steps. This included batching the images to load into the model, and creating the datasets from folder structure using TensorFlow tools.

The code and complete run of the model is located at this kaggle address:
https://www.kaggle.com/kiwikolby/fork-of-oregon-wildlife-mle-capstone

VI. Improvement

The first place to try and improve the overall model performance would be to make the entire base model trainable and not just the two final blocks. In this case the model presented in this project would serve as a baseline for comparison.

The second improvement would be considering the use of other mobile or edge based inference platforms that don't have accelerators, such as Raspberry Pi's. In this case, it would be interesting to compare results in terms of model size, Madds, and cpu resources required between MobileNetV2, MobileNetV3, SqueezeNet, SqueezeNext, and ShuffleNet.

The third improvement would be, as noted earlier, to remove duplicate images from the dataset. Also, some wildlife cameras take thermal images. Since the cameras are triggered at night, and the desire is to not scare the animals away from the camera., theses cameras take infrared picture. In this case, augmenting color images with infrared images would be interesting. An example of this work is in [8], "DenseFuse: A Fusion Approach to Infrared and Visible Images" by Hui Li and Xiao-Jun Wu.

VII. References

[1] https://www.pnas.org/content/115/25/E5716
[2] https://www.kaggle.com/c/iwildcam2018/overview
[3] https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/
[4] https://www.kaggle.com/virtualdvid/oregon-wildlife
[5] https://keras.io/guides/transfer_learning/
[6] https://arxiv.org/pdf/1704.04861v1.pdf
[7] https://arxiv.org/pdf/1801.04381.pdf
[8] https://arxiv.org/pdf/1804.08361.pdf