MIDTERM PROJECT

NEIL IVAN S. ORENCIA

Implementing Object Detection on a Dataset

Selection of Dataset and Algorithm

Selected Dataset

The dataset utilized for this midterm activity is the Oxford-IIIT Pet Dataset, developed by the Visual Geometry Group at Oxford. This dataset consists of 37 categories of pet images, with approximately 200 images per class. The images exhibit significant variations in scale, pose, and lighting conditions. The following annotations are available for every image in the dataset: (a) species and breed name; (b) a tight bounding box (ROI) around the head of the animal; and (c) a pixel level foreground-background segmentation (Trimap).

The dataset includes 12 cat breeds and 25 dog breeds, listed as follows:

Cat Breeds:

- Abyssinian
- Bengal
- Birman
- Bombay
- British Shorthair
- Egyptian Mau
- Maine Coon
- Persian
- Ragdoll
- Russian Blue
- Siamese
- Sphynx

Dog Breeds:

- American Bulldog
- American Pit Bull Terrier
- Basset Hound
- Beagle
- Boxer
- Chihuahua
- English Cocker Spaniel
- English Setter
- German Shorthaired
- Great Pyrenees
- Havanese
- Japanese Chin
- Keeshond
- Leonberger
- Miniature Pinscher
- Newfoundland
- Pomeranian
- Pug
- Saint Bernard
- Samoyed
- Scottish Terrier
- Shiba Inu
- Staffordshire Bull Terrier
- Wheaten Terrier
- Yorkshire Terrier

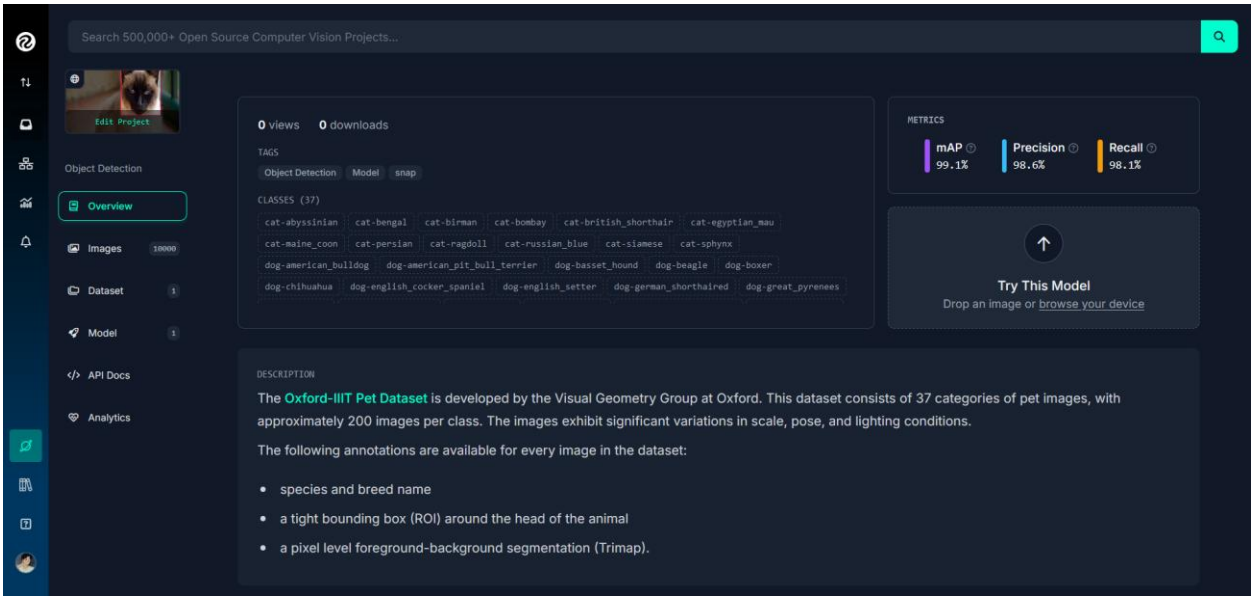

Figure 1: The Oxford-IIIT Pet Dataset in Roboflow

The dataset used for the project contains 10,000 images spread over 37 classes with about 200 images per class. The dataset was divided into three subsets: 7,000 images (70%) were assigned for training, while 2,000 images (20%) were assigned for testing, and 1,000 images (10%) were assigned for validation purposes. The preprocessing steps are auto-orientation to remove the need for standard orientation of images and resizing images to a fixed resolution 640x640 pixels. No data augmentation techniques have been applied. In the Google Colab, the dataset was obtained from Roboflow in the formats provided for various object detection models. Although it contains annotation when the dataset was downloaded from its official website, the class list needed to be adjusted after the upload to Roboflow to ensure that everything matched up correctly.

Though the dataset was mainly for breed classification, it was used for object detection in this midterm project because the object detection inherently has an advantage in handling scenarios with complex backgrounds, multiple animals, and the need for precision localization. Unlike traditional classification, which gives a single prediction over an entire image, object detection identifies and classifies individual animals within an image and further provides their exact locations with bounding boxes. This capability is very useful if the image contains several animals of different breeds, so each animal can be detected and classified separately.

Object detection is also designed to be invariant to pose, scale, and lighting conditions because it focuses on specific regions of interest rather than processing the entire image indiscriminately. These features make object detection a more versatile and effective approach for breed classification, especially in real-world scenarios where environmental complexity and variability are significant. Moreover, the choice to implement

object detection reflects the student's interest in exploring a more nuanced and challenging approach, aligning with its practical and technical benefits for this classification task.

Selected Algorithm

There are two algorithms selected for application in cat and dog breed object detection — YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector).

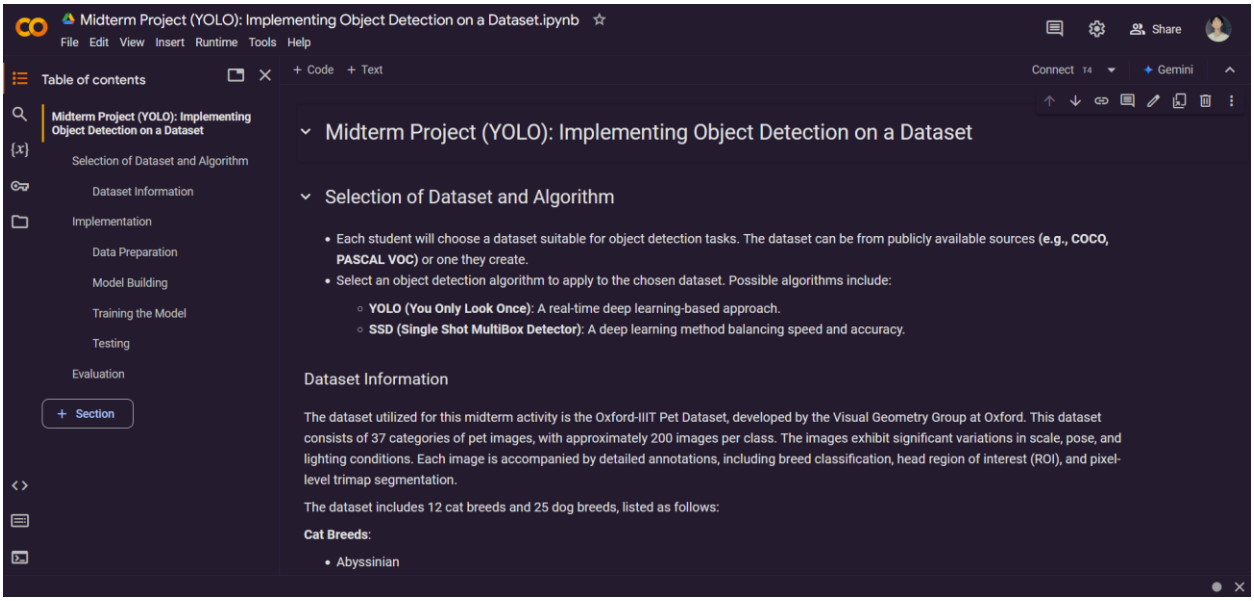

Figure 2: Screenshot of YOLO Implementation in Google Colab

YOLO is an object detection algorithm that uses a single neural network to predict the bounding boxes and class probabilities for multiple objects in an image simultaneously. This was introduced in 2015, changing the way the field approaches it because it combined two things-object localization and classification-all within one framework. It works by dividing the image into a grid and assigning the detection tasks to each grid cell. Every grid cell predicts the bounding boxes and the confidence scores, which describe the probability of objects and the accuracy of the position of the box. YOLO is also very fast and efficient. This makes it ideal for real-time object detection. This is because it allows for the entire image to be processed in one forward pass through the network, hence cutting down computation time in comparison to region-based methods.

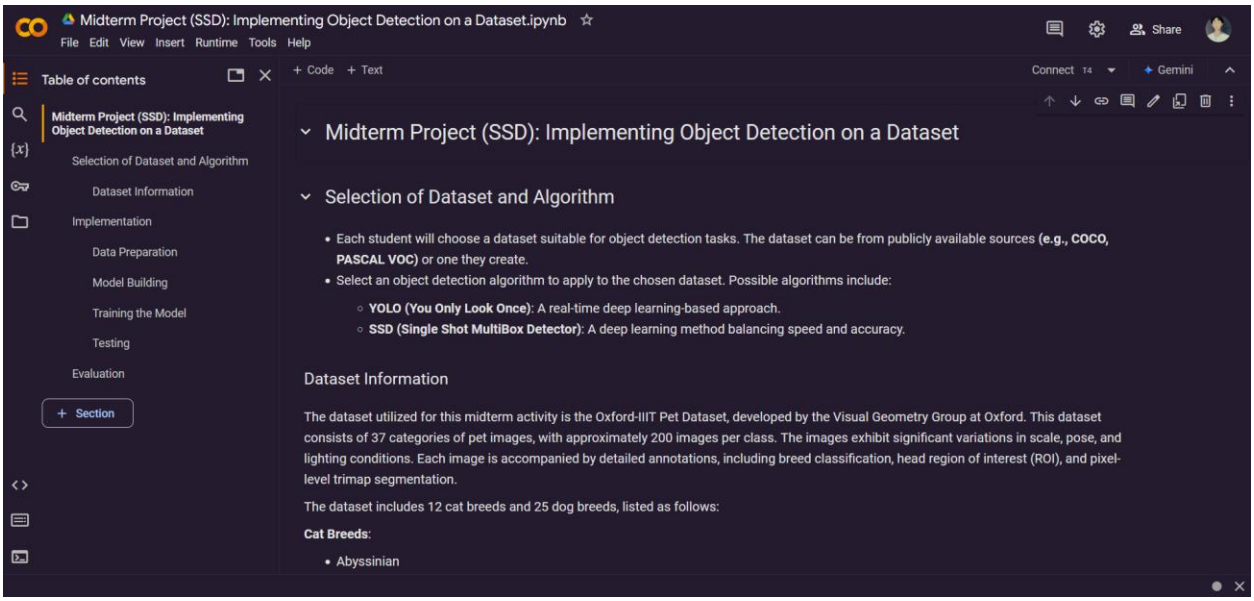

Figure 3: Screenshot of SSD Implementation in Google Colab

SSD is an object detection algorithm which uses feedforward convolutional neural networks to detect the presence of an object of almost any size in a single shot. This is enabled due to its generation of direct bounding boxes and the associated class predictions from feature maps in multiple levels. Hence, the designed algorithm of balancing speed with accuracy due to multiscale feature representation of anchor boxes for high localization is obtained. The architecture consists of a base network for feature extraction, followed by additional convolutional layers which produce predictions at multiple scales. Therefore, SSD performs very well in detecting small objects and has been widely applied in applications requiring fast object detection with accuracy.

Data Preparation

The dataset was processed using the Roboflow workspace after the setup of the dataset and its annotations were properly configured. During preprocessing, the images were automatically oriented to account for varying orientations in the raw data, and they were resized to 640x640 to maintain consistency with the input dimensions required by the object detection model, and no augmentations was applied as the focus was on only the base performance of the models without any extra variability addition.

In Google Colab, two versions of the dataset were imported to suit the respective algorithms in the object detection models. The first one was in TFRecord format, which was preferred for the Single Shot MultiBox Detector (SSD) since TFRecord is optimized for TensorFlow models. It allows for efficient storage and sequential reading of large datasets, which is necessary to handle SSD's multi-scale feature maps and high-volume data processing requirements in the training process.

```
rf = Roboflow(api_key="TUXckiOYdWfPKbzjriHg")
project = rf.workspace("feature-extraction-p1jos").project("cat-dog-breeds")
version = project.version(1)
dataset = version.download("tfrecord")
```

On the other hand, the YOLO model algorithm had its dataset imported in the YOLOv9 PyTorch format as it is fully compatible with the advanced data preprocessing pipeline in YOLOv9 to ensure smooth integration and less overhead in conversion. This version was chosen instead of its predecessors as well as other subsequent versions due to its cutting-edge performance and advanced features. Unlike previous versions, which used frameworks such as Darknet or earlier PyTorch implementations, YOLOv9 includes the most recent developments in object detection, such as improved model architectures, better multi-scale feature representation, and optimized inference speed. All these developments, coupled with the choice of the YOLOv9 PyTorch format was crucial to the efficiency and accuracy of the training for this project.

```
rf = Roboflow(api_key="TUXckiOYdWfPKbzjriHg")
project = rf.workspace("feature-extraction-p1jos").project("cat-dog-breeds")
version = project.version(1)
dataset = version.download("yolov9")
```

Single Shot Multi-Box Detector

Single Shot Multi-Box Detector was one of the object detection model algorithm chosen to be utilized for the Oxford-IIIT Pet Dataset due to its efficiency in handling multi-scale object detection tasks. The dataset includes a variety of cat and dog breeds which varies in size and appearance. SSD's architecture, which employs multi-scale feature maps and anchor boxes, is particularly well-suited for detecting objects at different scales within a single forward pass. Additionally, SSD offers a balance between speed and accuracy which makes it a practical choice for detecting objects in datasets like Oxford-IIIT, where precision is required to differentiate between fine-grained categories such as different breeds.

Model Building

```
!pip install protobuf==3.20.*
!pip uninstall Cython -y
!git clone --depth 1 https://github.com/tensorflow/models
```

The code sets up the preparation environment for building the SSD model with TensorFlow's Object Detection API. It installs a particular version of the Protobuf library, which is used in processing serialized data in TensorFlow models. It then removes the Cython library for compatibility reasons. Finally, the TensorFlow Models repository, which hosts pre-trained models and utilities for object detection, is cloned from GitHub.

```
%%bash
cd models/research/
protoc object_detection/protos/*.proto --python_out=.
```

The code compiles the protocol buffer `(.proto)` files necessary for TensorFlow's Object Detection API into Python files. Moving to the `models/research/` directory ensures that all commands are run in the right context. The protoc command will process all of the `.proto` files within the `object_detection/protos/` directory, translate them into Python-compatible files `(.py)`, and save them in the same directory.

```
import re

input_file_path =
'/content/models/research/object_detection/packages/tf2/setup.py'
```

```
output_file_path = '/content/models/research/setup.py'

with open(input_file_path) as f:
    setup_content = f.read()

with open(output_file_path, 'w') as f:
    updated_content = re.sub('tf-models-official>=2.5.1', 'tf-models-
official==2.8.0', setup_content)
    f.write(updated_content)
```

The code changes the version requirement for the `tf-models-official` package in the `setup.py`. It starts by reading the original content of the `setup.py` file found at `/content/models/research/object_detection/packages/tf2/`. The `re.sub` function is then used to change the version constraint for `tf-models-official` from >=2.5.1 to ==2.8.0, especially if specific dependencies are being utilized or the TensorFlow version being implemented. The updated content is then written to a new file located at `/content/models/research/setup.py`, preserving the original file but providing a modified version for the setup process.

```
%%bash
pip install pyyaml==5.3
pip install /content/models/research/
pip install tensorflow==2.8.0
pip install tensorflow_io==0.23.1
```

The code installs the dependencies that are required finishing the setup TensorFlow's Object Detection API. It starts by installing version 5.3 of the pyyaml library, necessary for parsing YAML configuration files typically used in machine learning models. Then, it installs TensorFlow Models Research from the local directory to allow access to a variety of pre-trained models as well as utilities for object detection. Finally, the required version of TensorFlow (2.8.0) and TensorFlow I/O (0.23.1) is installed to ensure compatibility with the SSD Object Detection API and other related functionalities.

Training the Model

```
!cp /content/ssd_mobilenet_v2_320x320_coco17_tpu-8/pipeline.config
/content/models/research/object_detection/configs/cat_dog_pipeline.config
```

The code copies the config for `ssd_mobilenet_v2_320x320_coco17_tpu-8` model to a new location and renames it for customization. Specifically, it duplicates the `pipeline.config` file, which contains all the settings required for training and evaluation, such as model architecture, dataset paths, and hyperparameters. The copied file is placed in the `object_detection/configs/` under the name `cat_dog_pipeline.config`.

```
import tensorflow as tf
import re

config_path =
"/content/models/research/object_detection/configs/cat_dog_pipeline.config"

with open(config_path, "r") as file:
    config = file.read()

config = re.sub(r"label_map_path:.*",
                "label_map_path: '/content/Cat-&-Dog-Breeds-1/train/cat-
dog_label_map.pbtxt'",
                config)

config = re.sub(r"train_input_reader.*\{[^}]*input_path:.*\}",
                """train_input_reader {
    label_map_path: '/content/Cat-&-Dog-Breeds-1/train/cat-
dog_label_map.pbtxt'
    tf_record_input_reader {
        input_path: '/content/Cat-&-Dog-Breeds-1/train/cat-dog.tfrecord'
    }
}""",
                config, flags=re.DOTALL)

config = re.sub(r"eval_input_reader.*\{[^}]*input_path:.*\}",
                """eval_input_reader {
    label_map_path: '/content/Cat-&-Dog-Breeds-1/valid/cat-
dog_label_map.pbtxt'
    shuffle: false
    num_epochs: 1
    tf_record_input_reader {
        input_path: '/content/Cat-&-Dog-Breeds-1/valid/cat-dog.tfrecord'
    }
}""",
                config, flags=re.DOTALL)

config = re.sub(r"fine_tune_checkpoint:.*",
```

```
                "fine_tune_checkpoint:
'/content/ssd_mobilenet_v2_320x320_coco17_tpu-8/checkpoint/ckpt-0'",
                config)

config = re.sub(r"num_classes:.*", "num_classes: 37", config)
config = re.sub(r"batch_size:.*", "batch_size: 16", config)
config = re.sub(r"fine_tune_checkpoint_type:.*",
                "fine_tune_checkpoint_type: 'detection'",
                config)

config = re.sub(r"num_steps:.*", "num_steps: 50000", config)
config = re.sub(r"learning_rate_base:.*", "learning_rate_base: 0.0002",
config)
config = re.sub(r"warmup_learning_rate:.*", "warmup_learning_rate: 0.00001",
config)
config = re.sub(r"total_steps:.*", "total_steps: 50000", config)
config = re.sub(r"learning_rate_decay_steps:.*", "learning_rate_decay_steps:
5000", config)

with open(config_path, "w") as file:
    file.write(config)

print("Configuration file updated successfully!")
```

The code edits the `pipeline.config` to train the SSD algorithm for training the selected dataset. The main updated parameters were the label map path and the paths to TFRecord files for the training and evaluation datasets to ensure the right input was used in training the model. It sets the fine-tuning checkpoint, the number of classes being 37, adjusts the batch size to 16, and some learning parameters as such learning rate, warmup rate, decay steps, and total training steps to 50,000.

```
%cd /content/models/research/

!python object_detection/model_main_tf2.py \
    --
pipeline_config_path="/content/models/research/object_detection/configs/cat_do
g_pipeline.config" \
    --model_dir="/content/models/research/object_detection/training/" \
    --num_train_steps=50000 \
    --use_tpu=False \
    --alsologtostderr
```

The code initiates the training. It changes into the directory containing the necessary scripts and then runs the `model_main_tf2.py` script with certain arguments. The `pipeline_config_path` is pointing to the updated configuration file that defines the model's training parameters, while `model_dir` is pointing to the directory where training outputs, such as checkpoints and logs, will be saved. The script sets the number of training steps to 50,000 and disables TPU usage, thus ensuring that the model trains on the available CPU or GPU. The `--alsologtostderr` flag enables logging messages to be displayed in the console for real-time feedback during training.

```
!python object_detection/exporter_main_v2.py \
    --input_type=image_tensor \
    --
pipeline_config_path="/content/models/research/object_detection/configs/cat_do
g_pipeline.config" \
    --
trained_checkpoint_dir="/content/models/research/object_detection/training/" \
    --
output_directory="/content/models/research/object_detection/exported_model/"
```

The code exports the trained SSD model for inference following the training phase. The `exporter_main_v2.py` script is run, providing several key arguments, including `input_type` set to `image_tensor` and, as indicated above, specifying that the model will take images as input during the inference part; `pipeline_config_path` to a configuration file that was used when training in order to ensure consistency in the exported model; the `trained_checkpoint_dir` wherein the final checkpoint from the training is placed; and `output_directory` wherein the exported model, including the SavedModel format, will be stored for testing and using the model.

Testing the Model

For the testing phase of the model, a structured workflow has been implemented for object detection in multiple utility functions, each aimed to address specific tasks within a pipeline. There are multiple functions in the code that work together as part of a pipeline for processing and detecting objects in images.

The `load_model()` loads the pre-trained object detection model from a specified path or URL. It ensures that the model is ready for inference and can be used to detect objects in input images. The `load_label_map()` loads a mapping of class IDs to human-readable labels. This is important for interpreting the model's predictions, as the model outputs numeric IDs that correspond to different object classes.

The `get_image_paths()` retrieves the file paths for all images in the provided directory. It is responsible for gathering the input images that will be processed by the object detection pipeline. The `crop_to_square(image)` takes an image and crops it to a square, ensuring that the aspect ratio is consistent and suitable for the model to process. This helps with resizing the image without distortion.

The `process_single_image(image_path, detect_fn, label_map)` processes an individual image by first loading it, running object detection on it, and then annotating the image with detected bounding boxes, class labels, and confidence scores. This function ties together the detection and visualization aspects of the pipeline.

The `display_grid(images)` is responsible for displaying the processed images in a grid format, allowing for easy visualization of the results. It helps in quickly reviewing the output of the object detection process. The `process_images()` serves as the main function that iterates over the list of image paths, applies the necessary preprocessing, runs inference, and processes the output. It coordinates the overall process, calling functions like `process_single_image()` for each image.

The `print_metrics(metrics)` prints the evaluation metrics for assessing the performance of the object detection model. It helps in analyzing the accuracy and effectiveness of the model.

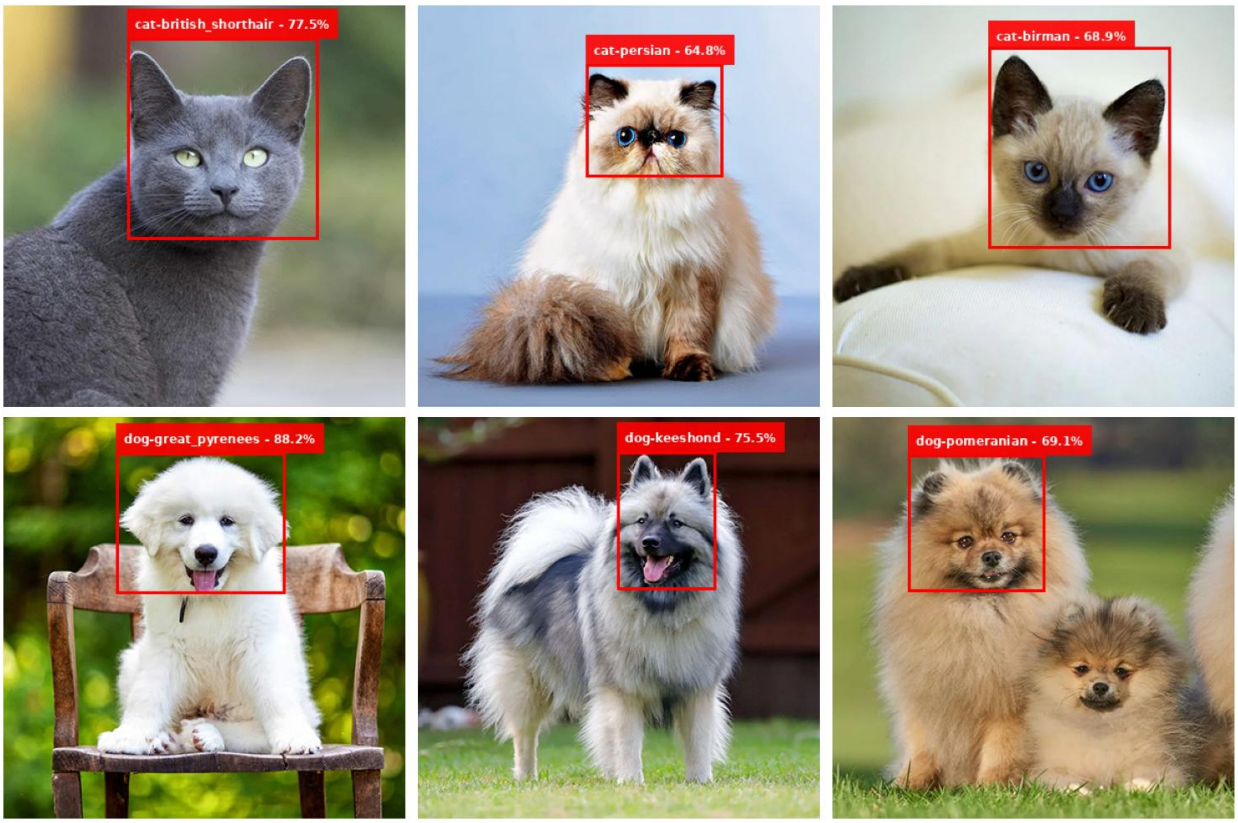

Figure 1: SSD Algorithm Testing Results

Six free stock images from the Internet, which included three images of cats and three of dogs, were used to test the model algorithm in terms of detecting and classifying the breeds of the subjects in the images.

The first image has been correctly identified as a British Shorthair cat with the average confidence of 77.55% and in a detection time of 4.64 seconds. The second image was accurately classified as a Persian cat with an average confidence of 64.83% in a detection time of 0.07 seconds. In the third image, a Siamese cat was not correctly identified, but was inaccurately labeled as a Birman cat with an average confidence of 68.95% and a detection time of 0.07 seconds.

For the dog images, the fourth image was identified correctly as a Great Pyrenees dog with an average confidence of 88.19% and a detection time of 0.03 seconds. The fifth image was correctly classified as a Keeshond dog with an average confidence of 75.53% and a detection time of 0.04 seconds. The sixth one was a two-Pomeranian image, which was correctly identified; however, the model detects only one dog with strong confidence, reaching an average confidence of 69.08% and a detection time of 0.03 seconds.

Overall, the model showed an acceptable ability and satisfactory capability to detect and classify breeds with high accuracy on most images. However, some limitations, such as misclassification and failure to detect multiple instances in a single image, are areas where further improvement is needed.

You Only Look Once

The YOLO (You Only Look Once) model algorithm is selected for use with the Oxford-IIIT Pet Dataset because of its real-time object detection, which provides a high level of detection accuracy for complicated datasets. Its architecture works well in scenarios demanding precision localization and classification due to the division of the image into a grid, whereby YOLO predicts a bounding box and class probabilities for every cell in the grid. Its grid-based detection allows it to handle object overlaps and scales.

Model Building

```
from ultralytics import YOLO
yolo_model = YOLO('yolov9m.pt')
```

The code initializes the YOLO model for object detection with the Ultralytics YOLO framework. This is achieved by importing the YOLO class from the ultralytics library, which enables access to the functionalities of the YOLO algorithm. The line `YOLO('yolov9m.pt')` loads the pre-trained YOLOv9m (medium) model weights, which are optimized for a balance between detection accuracy and computational efficiency.

Training the Model

```
yolo_model.train(
    data="/content/Cat-&-Dog-Breeds-1/data.yaml",
    epochs=25,
    batch=16,
    imgsz=640,
    lr0=0.01,
)
```

The code includes the training process of the YOLOv9 model. It employs several parameters specific to the dataset and computational limitations. The data parameter is a reference to the YAML file, which holds information regarding the dataset, including the paths to the training and validation data as well as the total number of classes for the Oxford-IIIT Pet Dataset, which is 37. The epochs are set to 25, a compromise between effective training and the constraints of the Google Colab environment, given the sheer amount of image data and the lack of resources to handle longer training sessions, such as 30 or more epochs, without running over runtime limits. The batch size is 16, chosen to balance memory usage and training speed. The input image size is 640x640 pixels, the

standard resolution for YOLO models, allowing enough detail for object detection without being too computationally expensive. Finally, the initial learning rate is 0.01, such that the optimizer starts training at an optimal point from which it can learn appropriately and avoid instability or slow convergence.

Testing the Model

For the testing phase of the YOLO model, a structured workflow has been implemented for object detection in multiple utility functions, each aimed to address specific tasks within a pipeline. There are multiple functions in the code that work together as part of a pipeline for processing and detecting objects in images.

The `upload_images()` prompts the user to upload six images through Google Colab's file uploader. It ensures exactly six images are selected, limiting excess uploads. The returned dictionary contains the image names and data for further processing.

The `crop_to_square()` crops an image to a square aspect ratio by centering the crop region. This standardization is used for visual consistency and compatibility with the YOLO model's input requirements.

The `calculate_metrics()` computes performance metrics like the number of detections, high-confidence detections (confidence > 0.5), average confidence, and class distribution.

The `process_single_image()` handles the detection pipeline for one image. It preprocesses the image, runs it through the YOLO model to detect objects, calculates metrics, and visualizes results by overlaying bounding boxes and labels. The processed image is cropped for consistency.

The `display_grid()` displays multiple images in a 3x2 grid using Matplotlib. This visualization helps users review detection results for all uploaded images simultaneously.

The `print_metrics()` outputs detection metrics in a clear, formatted text layout. It summarizes the model's performance per image, including confidence levels and class distributions.

The `process_images()` serves as the main pipeline. It integrates all the above functions—uploads images, processes each one with the YOLO model, displays the processed images, and prints metrics.

To recap, the pipeline begins with image uploads through `upload_images()`, followed by individual processing through `process_single_image()`, which combines preprocessing, detection, metric calculation, and visualization. After all images are

processed, they are displayed in a grid through `display_grid()`, and the metrics for each image are printed through `print_metrics()`.

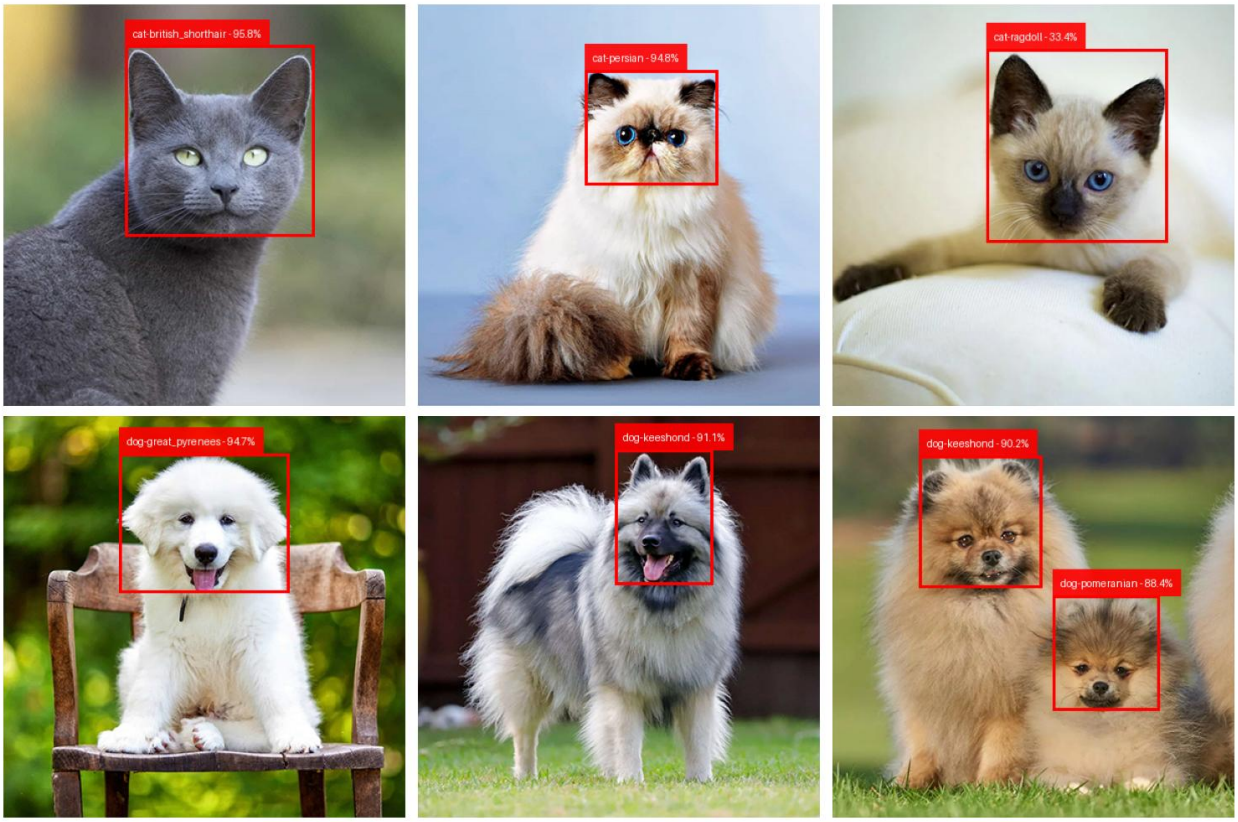

Figure 2: YOLO Algorithm Testing Results

The model was tested using the similar set of six free stock images sourced from the Internet, comprising three images of cats and three of dogs, to assess its effectiveness in detecting and classifying the breeds present in the images.

The first image was correctly identified as a British Shorthair cat, achieving an average confidence of 95.80% with a detection time of 0.06 seconds. The second image was also correctly classified as a Persian cat with an average confidence of 94.75% and a detection time of 0.05 seconds. The third image, which contained a Siamese cat, was incorrectly identified as a Ragdoll cat, with a lower average confidence of 33.42% and a detection time of 0.05 seconds.

For the dog images, the fourth image was accurately identified as a Great Pyrenees dog, achieving an average confidence of 94.69% and a detection time of 0.05 seconds. The fifth image was correctly classified as a Keeshond dog with an average confidence of 91.14% and a detection time of 0.04 seconds. In the sixth image, which featured two Pomeranian dogs, the model correctly identified one as a Pomeranian but misclassified the other as a Keeshond, resulting in an average confidence of 89.32% and a detection time of 0.04 seconds.

In general, the model performed very well with a good amount of detection and classification accuracy as well as high confidence for most cases. However, misclassification of certain breeds and inconsistencies when detecting multiple subjects in one image indicate areas where further refinement is required.

Evaluation

Overall Model Performance		
	SSD	YOLO
Mean Precision	57.31%	98.30%
Mean Recall	83.07%	96.74%
Mean Average Precision	48.16%	88.88%
Inference Speed		
	SSD	YOLO
Preprocess Time	5.13 ms	0.29 ms
Inference Time	42.55 ms	23.62 ms
Postprocess Time	1.24 ms	1.15 ms

Table 1: Overall Model Performance and Inference Speed for SSD and YOLO

The comparison between YOLO and SSD models gives good insight into their capabilities and the suitability of the models for object detection tasks. Each model has its strengths and weaknesses, which are reflected in their performance metrics and inference speeds, which are critical factors for applications that require accuracy and efficiency.

The results for the SSD model reveal several significant insights into its performance and efficiency in object detection. The mean precision indicates that of all the objects that the model detected, 57.31% were correctly identified as true positives. This is a fairly moderate precision, indicating that though the model is able to detect objects, it might still be producing a lot of false positives. A recall of 83.07% means that the SSD model correctly detected almost 83.07% of all actual positive instances in the dataset. With such a high recall, it means that the model is good at capturing most of the objects present but may not be very selective in its detection, which might decrease the precision. The mAP score of 48.16% reflects the overall accuracy across different classes and IoU thresholds. This score indicates that the model recalls reasonably but that precision and, more importantly, the overall average performance remain to be improved.

The preprocess time is 5.13 ms, meaning that the time to prepare input data for model processing is quite efficient. The inference time reflects the amount of time it takes a model to process an image and make predictions. In this case, it's relatively fast, at approximately 42.55 ms. The postprocessing time at 1.24 ms was minimal, meaning that once predictions were made, they were formatted quickly for output.

Given its strengths at speed and recall, the performance of the SSD model algorithm can conclude to have as many detections as possible but also at some cost of being imprecise. Although it holds solid performance in terms of speed and recall, enhancing this precision would be essential especially for applications requiring high accuracies in object detection tasks.

The results for the YOLO model show a very effective performance in terms of object detection with high accuracy and fast processing speed. Mean precision at 98.30% is extremely high, meaning that nearly all the detected objects are true positives. The model makes very few false positives and thus appears to be very reliable in predictions. A recall of 96.74% shows that the model correctly identifies a very high percentage of the actual objects in the images. Such a high recall value signifies that the model has a good ability to capture most instances of objects and has less missed detections, which is referred to as false negatives. The mAP score of 88.88% further supports the strong overall performance of the model on various classes and detection thresholds.

The preprocessing time of 0.29 ms is very low, implying the model can quickly prepare the input data for the time of inference, which is a requirement for real-time applications. An inference time of 23.62 ms means that the model processes images rapidly. The postprocessing time of 1.15 ms is minimal too, and this implies it is efficient in outputting results after predictions are made.

The results suggest that YOLO is the first choice among the two object detection models used because its speed comes hand in hand with accuracy, balancing the precision-recall trade-off together with low latency in inference.

The outcomes presented indicate YOLO to be one of the most efficient and accurate models with an extremely good speed along with good detection performance. Such an accurate model, along with precision, recall, and mAP values, makes it suitable for more complex and critical applications that require both high precision and speed. Conversely, though SSD is slightly low in precision and is much slower than the previous inference, it could find suitable application in less demanding or highly computational resource-limited situations.

Category	Model	Precision	Recall	AP@0.50	AP
Cat-Abyssinian	YOLO	99.21%	97.14%	99.34%	89.05%

	SSD	40.74%	78.57%	32.01%	32.01%
Cat-Bengal	YOLO	96.28%	92.54%	98.47%	86.33%
	SSD	43.75%	72.41%	31.68%	31.68%
Cat-Birman	YOLO	93.29%	94.00%	98.21%	88.17%
	SSD	42.86%	90.00%	38.57%	38.57%
Cat-Bombay	YOLO	99.91%	100%	99.50%	88.11%
	SSD	58.06%	85.71%	49.77%	49.77%
Cat-British Shorthair	YOLO	99.47%	95.92%	99.38%	90.83%
	SSD	48.65%	85.71%	41.70%	41.70%
Cat-Egyptian Mau	YOLO	99.18%	98.11%	99.45%	91.06%
	SSD	43.48%	90.91%	39.53%	39.53%
Cat-Main Coon	YOLO	99.77%	98.08%	99.35%	87.99%
	SSD	51.92%	100%	51.92%	51.92%
Cat-Persian	YOLO	98.03%	99.28%	99.46%	89.50%
	SSD	78.12%	80.65%	63.00%	63.00%
Cat-Ragdoll	YOLO	94.75%	93.41%	96.88%	87.81%
	SSD	64.86%	77.42%	50.22%	50.22%
Cat-Russian Blue	YOLO	97.06%	97.87%	99.34%	91.14%
	SSD	42.55%	95.24%	40.53%	40.53%
Cat-Siamese	YOLO	98.19%	96.77%	99.45%	90.45%
	SSD	55.10%	93.10%	51.30%	51.30%
Cat-Sphynx	YOLO	99.71%	100%	99.50%	92.15%
	SSD	84.38%	100%	84.38%	84.38%
Dog-American Bulldog	YOLO	96.42%	100%	98.71%	91.15%
	SSD	52.94%	64.29%	34.03%	34.03%
Dog-American Pitbull Terrier	YOLO	99.85%	98.11%	99.46%	91.99%
	SSD	50.00%	76.92%	38.46%	38.46%
Dog-Basset Hound	YOLO	97.54%	100%	99.50%	92.41%
	SSD	44.23%	85.19%	37.68%	37.68%

Dog-Beagle	YOLO	99.35%	84.21%	96.53%	85.77%
	SSD	57.38%	77.78%	44.63%	44.63%
Dog-Boxer	YOLO	99.65%	84.62%	93.88%	84.65%
	SSD	51.79%	70.73%	36.63%	36.63%
Dog-Chihuahua	YOLO	99.40%	98.28%	99.35%	90.47%
	SSD	62.86%	88.00%	55.31%	55.31%
Dog-English Cocker Spaniel	YOLO	100%	97.34%	99.35%	88.71%
	SSD	52.50%	70.00%	36.75%	36.75%
Dog-English Setter	YOLO	100%	97.34%	99.35%	88.71%
	SSD	47.62%	74.07%	35.27%	35.27%
Dog-German Shorthaired	YOLO	98.32%	98.94%	99.45%	89.08%
	SSD	33.33%	61.54%	20.51%	20.51%
Dog-Great Pyreness	YOLO	99.29%	97.92%	98.29%	85.83%
	SSD	69.57%	59.26%	41.22%	41.22%
Dog-Havanese	YOLO	98.08%	98.23%	99.48%	85.89%
	SSD	55.56%	100%	55.56%	55.56%
Dog-Javanese Chin	YOLO	99.24%	100%	99.50%	89.18%
	SSD	88.89%	100%	88.89%	88.89%
Dog-Keeshond	YOLO	97.88%	98.18%	99.46%	87.99%
	SSD	62.16%	79.31%	49.30%	49.30%
Dog-Leonberger	YOLO	100%	94.23%	98.73%	89.65%
	SSD	43.48%	68.97%	29.99%	29.99%
Dog-Miniature Pinscher	YOLO	98.36%	97.03%	98.66%	86.95%
	SSD	37.78%	73.91%	27.92%	27.92%
Dog-Newfoundland	YOLO	96.09%	98.25%	99.36%	87.02%
	SSD	57.50%	82.14%	47.23%	47.23%
Dog-Pomeranian	YOLO	96.09%	98.25%	99.36%	87.02%
	SSD	66.67%	100%	66.67%	66.67%
Dog-Pug	YOLO	98.42%	98.15%	99.48%	90.99%

	SSD	80.65%	96.15%	77.54%	77.54%
Dog-Saint Bernard	YOLO	98.31%	96.85%	99.29%	90.61%
	SSD	67.65%	79.31%	53.65%	53.65%
Dog-Samoyed	YOLO	99.88%	98.57%	99.49%	88.51%
	SSD	58.33%	95.45%	55.68%	55.68%
Dog-Scottish Terrier	YOLO	100%	88.56%	97.94%	86.46%
	SSD	79.41%	84.38%	67.00%	67.00%
Dog-Shiba Inu	YOLO	99.63%	95.00%	99.42%	90.22%
	SSD	84.62%	84.62%	71.60%	71.60%
Dog-Staffordshire Bull Terrier	YOLO	99.22%	96.08%	98.98%	91.38%
	SSD	54.76%	85.19%	46.65%	46.65%
Dog-Wheaten Terrier	YOLO	99.24%	100%	99.50%	85.70%
	SSD	44.74%	80.95%	36.22%	36.22%
Dog-Yorkshire Terrier	YOLO	91.89%	100%	98.96%	88.11%
	SSD	61.54%	85.71%	52.75%	52.75%

Table 2: Class-wise Performance for SSD and YOLO

The table above shows the comparison between YOLO and SSD in all the classes used in the model. It shows a clear difference in their performance metrics, where YOLO is significantly better than SSD in all the measured parameters. YOLO has achieved great precision and recall values for both cat and dog classes, often above 95%. For instance, for the Sphynx cat, YOLO gives a precision of 99.71% and recall of 100%, giving an AP@0.50 of 99.50% and an AP of 92.15%. Similarly, for the American Pit Bull Terrier, YOLO gives a precision of 99.85%, recall of 98.11%, AP@0.50 of 99.46%, and an AP of 91.99%. These metrics show that YOLO is able to provide highly accurate detections with minimal errors consistently.

On the other hand, the SSD has a much wider range of performance, with lower precision and recall across the multiple classes. For example, for the Sphynx cat, the SSD achieved 84.38% precision and 100.00% recall, thus obtaining an AP@0.50 of 84.38%. However, in the case of the German Shorthaired Pointer, the SSD's performance goes down drastically to a precision of just 33.33%, recall of 61.54%, and an AP@0.50 of 20.51%. This inconsistency indicates that SSD cannot deal effectively with specific classes, so it is less reliable for applications that require high accuracy.

However, the SSD could significantly benefit from more training steps. Increasing `num_train_steps` would mean that the model could converge much better, thus narrowing the gap between its performance and that of YOLO. For cat classes, such as with an AP of 32.01% for the Abyssinian cat compared to YOLO's 89.05%, and for dog classes, such as with an AP of 34.03% for the American Bulldog compared to YOLO's 91.15%, definitely, there is an indication of possibly incomplete training or insufficient optimization.

This variance highlights the advantage of YOLO in applications where precision and recall are very important, especially in real-world applications. YOLO presents consistent AP@0.50 values over 98% for most classes and was rather robust overall, making it a better choice in those applications. SSD is less reliable but can do well when the application calls for computational efficiency or less demanding accuracy. However, in this case, SSD needs more training steps and optimization to better compete with YOLO.

In conclusion, although YOLO is producing outstanding results with precision and recall close to perfection for most classes, the performance of SSD still has a lot of room for improvement. More `num_train_steps` and fine-tuning might turn SSD into a more competitive model, closing the gap between its current performance and the great performance shown by YOLO.