Ecto-Trigger

Ultra-lightweight CNNs for on-device camera trap object detection

This vignette contains user guidance, explanations, and installation setup tips.

It also includes class documentation and code reference (docstrings) for the Ecto-Trigger codebase.

Authored by: **Ross J. Gardiner**.

June 8, 2025

Chapter 1

Ecto-Trigger

Ecto-Trigger is a toolkit designed to help ecologists develop lightweight AI models which can automate species detection in camera trap images. It is especially useful for scenarios where traditional motion sensors aren't reliable, for example, detecting insects as they lack the body-heat required to trigger conventional PIR sensors.

This project explains Ecto-Trigger, its use cases, how to use it and how to install it. Frequently asked questions are given below to provide initial insights and further chapters explain specific aspects:

- Usage Guide gives detailed information on each of the files within Ecto-Trigger and how you can use them from within a Python programme and from the command line.
- Deployment Guide gives information about how to use Ecto-Trigger in the field, on devices which form camera traps.
- Package Install Guide gives information about how to install package dependencies for Python, to use the Ecto-Trigger toolkit.

All documentation is served via our `web-page`, we have also made this accessible as a `pdf-format vignette`. This all supports our paper: `Towards scalable insect monitoring: Ultra-lightweight CNNs as on-device triggers for insect camera traps`, which contains more advanced technical details and background for our approach.

1.1 FAQs

1.1.1 What is Ecto-Trigger and why would I use it?

Ecto-Trigger is a free and open-source tool-kit that helps you train and use small, efficient AI models to detect the presence (or absence) of a specific object (such as an insect) in camera trap images, and can be used on the camera trap device itself.

It is built for use by ecologists, especially those in a fieldwork scenario where traditional camera trap triggers (such as passive infrared, PIR, sensors) don't work well. For example:

- PIR sensors can miss cold-blooded, small or fast-moving animals, such as insects
- They many often trigger unnecessarily from heat or movement not attributed to the target animal, such as rustling leaves

You would use Ecto-Trigger in a scenario where these limitations are significant, for example, you may have limited storage, bandwidth or human resources, so you cannot afford to continuously record video or time-lapse footage. You may also wish to use low-powered microcontrollers to create your camera trap as these require less energy and so can be easier to scale up or use in environments where solar power is not easy to access.

Ecto-Trigger uses a computer vision based approach, relying on the contents of the image itself - not motion or heat - to decide whether to keep or discard each captured image. These models act a trigger system themselves, which uses only optical information from a camera.

1.1.2 How does it work?

The core idea is that a simple, non-computationally expensive model can be trained to answer one specific question about an image which was just captured: "Does this image, which I'm seeing right now, contain the object I care about?"

The model uses a compact convolutional neural network architecture called MobileNetv2 to answer this. It has been trained on example images (some containing the target object, some not) and provides one number at its output, the likelihood that the image contains the object. Unlike more heavyweight object detectors, it does not compute the bounding box, of the object in the image location or class label, these binary classifiers only tell you if the object is present, helping your programmes decide if they should hold on to the image or throw it away. The model weights we provide with this code-base are trained to detect insects, to help with the development of insect camera traps, but you can train your own variations easily using our tools. These models are engineered in such a way that once trained, they can run on small, affordable devices such as ESP32s3 microcontrollers, to filter a stream of images in real-time while consuming very little energy. These are also compatible with more capable computing platforms such as the Raspberry Pi.

An example pipeline for how Ecto-Trigger models could be built into code running on a camera trap itself is provided below (this implementation is not provided by the Ecto-Trigger tool-kit), further information is given in our paper:

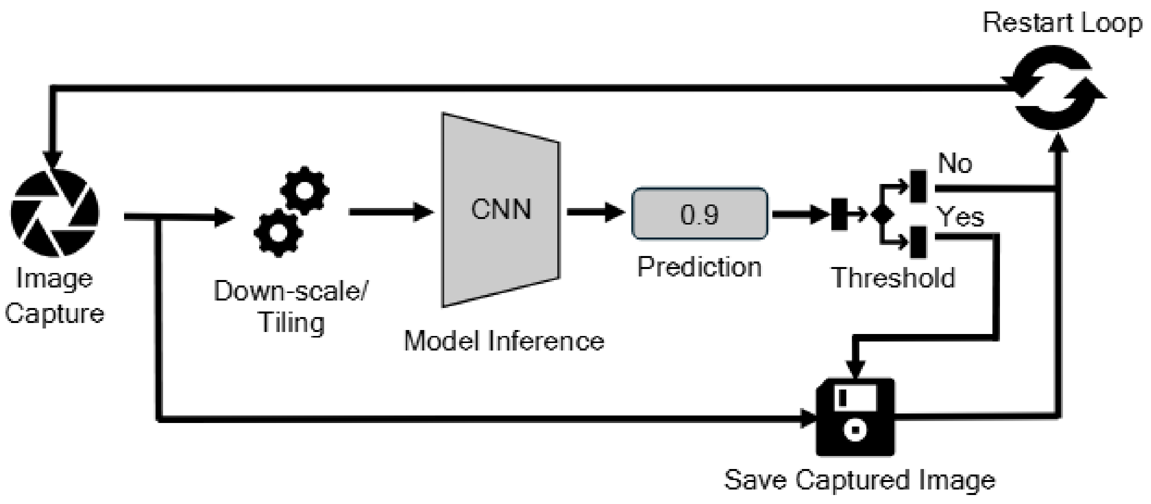

Figure 1.1 pipeline

1.1.3 Do I need to know AI to use Ecto-Trigger?

No, this code-base is designed to be beginner-friendly. To run the code and produce models you do not need to understand advanced concepts in deep learning, but some understanding of Python programming and data-wrangling may be useful to make it easier to get started especially if you wish to train your own models.

1.1.4 Code overview: What's included in the toolkit?

File	Purpose
`model_loader.py`	Build and load deep learning models (for training use)
`model_trainer.py`	Train a model using your labeled image dataset
`model_evaluator.py`	Evaluate model performance on test data

`model_quantiser.py`	Convert trained models to `.tflite` for edge devices
`saliency_map_evaluator.py`	Visualize what the model focuses on
`tflite_model_runner.py`	Run `.tflite` models on Raspberry Pi (no TensorFlow needed)

To find out more about how to use every file, check the usage guidance, which includes a description of all the ins and outs.

1.1.5 How can I get started?

The first thing you need to do is install all the necessary packages for Ecto-Trigger to run using Python. We recommend using a virtual environment to keep things neater, and have added all base requirements to a text file for convenience. To do this, you can use the instructions below:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Once completed, the packages within `requirements.txt` will be installed. If you want to check this, you can use:

```
pip list
```

Which will print out all the packages installed in your virtual environment to the terminal window.

We have provided further installation details, which are accessible via our install guidance page.

1.2 Contributing

This work is distributed under the GPL-3.0, see LICENSE, which means it is open-source, free to use, and free to modify and distribute. We encourage others to contribute to help grow the code-base and add any features they may need or want. To do this, you can make a `pull request`.

1.3 Citation

If you find Ecto-Trigger useful in your work, firstly, we would love to hear about it! Secondly, we would love for you to cite our paper, using the details below for now.

```
@misc{gardiner2025scalableinsectmonitoringultralightweight,
      title={Towards Scalable Insect Monitoring: Ultra-Lightweight CNNs as On-Device Triggers for Insect
      Camera Traps},
      author={Ross Gardiner and Sareh Rowands and Benno I. Simmons},
      year={2025},
      eprint={2411.14467},
      archivePrefix={arXiv},
      primaryClass={q-bio.QM},
      url={https://arxiv.org/abs/2411.14467},
}
```

Chapter 2

Deployment Guide for Ecto-Trigger

This guide explains how to run a trained Ecto-Trigger model on real devices in the field. We provide instructions for two supported platforms:

Raspberry Pi

To execute the models on Raspberry Pi systems, you can choose to use the Tensorflow or TFLite runtime (reccomended). To use the TFLite runtime, check out the guidance `here`. Basic steps:

(On RPi)

```
python3 -m pip install tflite-runtime
```

Using Python, you can execute a quantised inference:

```
import numpy as np
from tflite_model_runner import TFLiteModelRunner

q_model = TFLiteModelRunner.load_tflite_model("model_weights/8/quant/8_int8.tflite")

input_image_array = np.random.uniform(0, 255,
size=(q_model.get_input_details()[0]["shape"][1:])).astype(np.uint8)
input_image_array = np.expand_dims(input_image_array, axis=0)

q_model.set_tensor(q_model.get_input_details()[0]["index"], input_image_array)
q_model.invoke()

output = q_model.get_tensor(q_model.get_output_details()[0]["index"])
print(output[0]) # remember that the output will be in confidence range 0-255
```

Using this example, you can load images into the `input_image_array`, by replacing the part using numpy, e.g. you could use `PiCamera` to take images, and then process them with the model. `output[0]` will always contain the prediction from a given image, which is just given as an integer number.

ESP32-S3

Deploying models onto microcontroller platforms is a little more complicated, as these don't usually support python, so we have to compile code from scratch to execute on each device. To make things as easy as possible, we have provided an example project which uses our models on ESP32s3 chipset with the Platformio extension for VSCode.

We have made a separate repository for this, which includes full guidance and further details.

`InsectCT_lilygo`

If you have not yet trained or quantised a model first, use one of ours developed for insect detection, or follow the our usage guidance to train your own.

Chapter 3

Package Install Guide

It is reccomended to install these dependencies in a virtual environment, this is facilitated by `python`:

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Packages can thus be installed using `pip install <package>`.

Install the following requirements for instantiating and running the model.

```
keras==2.4.1
tensorflow==2.4.1
opencv-python==4.6.0
```

Install these requirements for model training

```
imgaug==0.4.0
numpy==1.23.5
```

Alternatively, to install all the required packages, use our requirements file:

```
pip install -r requirements.txt
```

To use the file `tflite_model_runner.py`, you must install the tflite runtime, this can be useful on devices where you just want to execute the model and don't nessicarily want to install the whole tensorflow package.

```
pip install tflite_runtime
```

Chapter 4

Usage Guide: How to Use Ecto-Trigger

4.1 Overview

Ecto-Trigger comprises a basic `collection of files`. Where each file defines a class and command-line interface for independent tasks. In other words, it is made up of a small number of Python scripts, that each do one job. Ecto-Trigger contains files for training a model, evaluating its performance, preparing to run it on a field device or computing an example saliency map for a given model and image.

Each section in the guide below explains how to use each of the scripts in the Ecto-Trigger toolkit. All the tools are modular, so you can use them together, or independently depending on your needs. Each script is also well documented with comments to make it easy to extend.

4.2 `model_loader.py`

This script provides a class, `ModelLoader`, which has static methods `create_model()`, `load_keras_↩ model()` and `load_tflite_model()` to create, load or prepare models for use in training, evaluation or inference. While you won't typically use this file directly from the command line, its essential for internal use throughout the Ecto-Trigger workflow.

4.2.1 What it does

- Creates new models based on configurable parameters (e.g. image shape, or width multiplier)
- Loads trained Keras models from `.hdf5` files
- Loads quantised TensorFlow Lite models (`.tflite`)

4.2.2 Python usage example

Below is shown how to use the `ModelLoader` class via a Python programme.

```python
from model_loader import ModelLoader

# this line will load a Keras model from a .hdf5 checkpoint file
k_model = ModelLoader.load_keras_model("path/to/weights.hdf5")

# this line will load a compressed tflite model from a .tflite file
q_model = ModelLoader.load_tflite_model("path/to/weights.tflite")

# to create your own model, use the function below
k_model = ModelLoader.create_model((1080, 1080, 3), 0.5, dropout_rate=0.2, freeze_base=False)
```

4.2.3 Inputs and Outputs

```
> Input:
  - A model file (.hdf5 or .tflite), or model creation parameters

> Output:
  - A TensorFlow Keras model object (for training/evaluation), or
  - A TensorFlow Lite interpreter object (for on-device inference)
```

4.3 `model_trainer.py`

This script contains a class, `ModelTrainer`, which allows you to train a binary classification model using your own dataset of labelled images. You can train directly in Python or from the command line.

It supports custom parameterisation, so you can change the input size or model width (see [our paper]() to understand how these affected our trained models). It saves the `hdf5` model along with logs throughout the training process, which can be monitored using TensorBoard.

4.3.1 What it does

- Loads images and YOLO-format labels from your training and validation directories
- Builds and compiles a MobileNetv2-based model for binary classifiation of your data
- Trains the model for a given number of epochs
- Logs performance metrics (such as accuracy) to a directory for visualisation with TensorBoard
- Saves the trained model as a `.hdf5` file

You can load the `ModelTrainer` class from a Python programme as follows:

```
from model_trainer import ModelTrainer

mt = ModelTrainer({"train_data_dir": "/path/to/train/data", "val_data_dir": "/path/to/val/data",
      "batch_size" : 16, "input_shape" : (120,160,3), "alpha": 0.35, "log_dir" : "logs", "model_type":
      "Mobnetv2", "epochs" : 2, "use_pretrained_weights" : False})

mt.train()
```

Alternatively, you can also call `model_trainer.py` directly from the command line:

```
python model_trainer.py \
    --train_data_dir "/path/to/train/data" \
    --val_data_dir "/path/to/validation/data" \
    --batch_size 16 \
    --input_shape "(120, 160, 3)" \
    --alpha 0.35 \
    --log_dir "logs" \
    --epochs 20 \
```

Where:

- `train_data_dir` is the path to your directory of training images and YOLO-format labels
- `val_data_dir` is the path to your directory of validation images and YOLO-format labels
- `batch_size` is the number of images per training step (e.g. 16)
- `input_shape` is the dimensions for the input size, specified as a string tuple, e.g. (120, 160, 3), height, width, number of channels.
- `alpha` controls the model size through changing the model width - smaller values of alpha produces lighter models, but this has an accuracy penalty, see our paper for further details.
- `log_dir` is the directory where training logs will be saved for analysis with TensorBoard.
- `epochs` is the number of full passes though the training data to run before exiting training.

4.3.2 Inputs and Outputs

```
> Input:
  - Folder of labelled training images (YOLO format)
  - Folder of labelled validation images (YOLO format)

> Output:
  - Trained `.hdf5` model file
  - Training logs viewable in TensorBoard
```

4.3.3 YOLO Dataset Format

To organise your dataset in YOLO-format for training, follow this structure:

```
your_data_train/
|-- img001.jpg
|--  img001.txt
|--  img002.jpg
|--  img002.txt
|--  ...

your_data_val/
|--  img001.jpg
|--  img001.txt
|--  img002.jpg
|--  img002.txt
|--  ...
```

Each `.txt` file should contain the YOLO-style annotation for its corresponding image (for more information about the YOLO format, see `here`):

For images with the object of interest: each line in the .txt file should contain a bounding box in this format:

```
0 x_center y_center width height
```

For images without the object: the .txt file will be empty (zero length).

4.3.4 Monitoring Training with TensorBoard

You can check training progress using `Tensorboard` by passing it the log directory:

```
tensorboard --logdir=logs
```

This allows you to monitor accuracy values throughout training, so you can see how its going and when you might want to stop model training.

4.4 `model_evaluator.py`

This script contain a class, `ModelEvaluator`, which helps you evaluate how well your trained model performas on a labelled test dataset. It calculates and prints out the model's accuracy and loss, giving you a quick sense of how confidently and correctly your model is making predictions.

4.4.1 What it does

- Loads a trained Keras model (given in `.hdf5`) format.
- Loads a test dataset from a directory in YOLO-format.
- Evaluates the models performance on the dataset.
- Prints out the accuracy and loss values to the terminal window.

4.4.2 Python usage example

You can call the `ModelEvaluator` directly inside your Python programmes:

```
from model_evaluator import ModelEvaluator

me = ModelEvaluator(16, (120, 160, 3), "model_weights/8/checkpoints/weights.10.hdf5",
      "/path/to/validation/data")

me.evaluate()
```

You can also call `model_evaluator.py` from the command line:

```
python model_evaluator.py \
    --batch_size 16 \
    --weights_path "model_weights/8/checkpoints/weights.10.hdf5" \
    --test_data_dir "/path/to/validation/data"
```

Where:

- `batch_size` is the number of images to process per step (e.g. 16)
- `weights_path` is the path to the trained Keras model to evaluate (`.hdf5` format)
- `test_data_dir` is the folder of labelled images in YOLO format for evaluation

4.4.3 Inputs and Outputs

```
> Input: Trained `.hdf5` model, test image folder
> Output: Accuracy score, loss printed to terminal window
```

4.5 `model_quantiser.py`

This script defines a class, `ModelQuantiser` which can be instantiated in Python as shown to quantise a given model in keras `.hdf5` format and save it as a `.tflite` file. Quantisation helps you convert an Ecto-Trigger Keras model into a smaller, more computationally efficient representation for use on low-powered devices such as Raspberry Pi or ESP32-S3. This allows real-time, on-device object detection in the field.

4.5.1 What it does

- Loads a trained Keras model (supplied in `.hdf5` format)
- Calibrates the quantisation process using a representative dataset (a folder containing a small number (around 100) of example images)
- Converts the `.hdf5` model into TensorFlow Lite (`.tflite`) format.
- Saves the now quantised model, ready for deployment.

4.5.2 Python usage example

You can call the `ModelQuantiser` directly inside your Python programmes:

```
from model_quantiser import ModelQuantiser

mq = ModelQuantiser(
        weights_file="/path/to/weights.hdf5",
        representative_dataset="/path/to/representative_dataset,
        representative_example_nr=100
    )

# Quantise the model and save it
mq.quantise_model(output_path="/path/to/weights.tflite")
```

You can also call `model_quantiser.py` from the command line:

```
python model_quantiser.py \
  --weights_file /path/to/weights.hdf5 \
  --representative_dataset /path/to/representative_dataset \
  --representative_example_nr 100 \
  --output /path/to/weights.tflite
```

Where:

- `weights_file` is the path to the trained Keras model (`.hdf5`) file, to be quantised.
- `representative_dataset` is the path to the folder of sample images to use for calibrating the quantisation process
- `representative_example_nr` is the number of images to use from that folder, e.g. 100.
- `output` is the filepath for saving the resultant `.tflite` model.

4.5.3 Inputs and Outputs

```
> Input:
  - A trained `.hdf5` model
  - A validation dataset (images and matching YOLO-style `.txt` files)

> Output:
  - Accuracy and loss values printed to the terminal
```

4.6 saliency_map_evaluator.py

This script provides a visual way to understand what the model is "looking at" when it makes a prediction. It generates saliency maps, these are heatmaps that highlight the parts of a given input image which most influenced a model's decision. This is a helpful tool for interpretation, debugging and visualisations.

4.6.1 What it does

- Loads a given Keras model (given to the script in `.hdf5` format)
- Processes a given input image (given to the script in `.jpg` or `.png` format)
- Runs the model and computes a saliency map showing where the model focused.
- Creates a composite image which includes: the original image, the saliency heatmap, the confidence score at the output of the model.

4.6.2 Python usage example

If you want to use the script in your own programmes, you can call it programmatically as follows:

```
from saliency_map_evaluator import SaliencyMapGenerator

smg = SaliencyMapGenerator(weights_file="model_weights/8/checkpoints/weights.10.hdf5")

smg.generate_saliency_map(input_image_path="input.png", output_path="saliency_plot.png")
```

You can also call `saliency_map_evaluator.py` from the command line:

```
python saliency_map_evaluator.py \
  --weights_file model_weights/8/checkpoints/weights.10.hdf5
  --input_image input.png \
  --output saliency_plot.png
```

Where:

- `weights_file` is the path to your trained model file in `.hdf5` format
- `input_image` is the path to the image you wish to analyse
- `output` defines the filename and path you want to save the final composite plot to.

4.6.3 Inputs and Outputs

```
> Input:
  - A trained model file (.hdf5)
  - An image file (e.g., .jpg or .png)

> Output:
  - A .png image showing:
      - The input image
      - A heatmap of attention (saliency map)
      - The prediction confidence score
```

4.7 generator.py

This file defines the class `CustomDataGenerator`, which is a data loading utility built for training Ecto-Trigger models. It works with image datasets which follow the YOLO annoation format (as described above), and prepared them for binary classification tasks (e.g. "insect" or "no insect").

4.7.1 What it does

`CustomDataGenerator` creates batches of images and labels which can be used by other scripts for both training and evaluation. It:

- Loads images from a folder, which is specified as an argument.
- Matches each image with its corresponding `.txt` annotation file in YOLO-style.
- Converts all annotations for a given image into binary labels (1 = object present, 0 = no object).
- Resizes and formats the images for input into the model.
- Optionally shuffles and augments the images (see our paper for details on augmentations).
- Supplies batches of images and labels to a model, via a Keras-compatible interface.

This is useful when training models using `model_trainer.py`.

4.7.2 Python usage example

If you want to integrate `CustomDataGenerator` into your own programmes, you can use it inside Python as follows:

```python
from generator import CustomDataGenerator
data_gen = CustomDataGenerator(
    data_directory="/path/to/dataset",    #Folder with .jpg and .txt files in YOLO-format, this is your
      dataset
    batch_size=16,                        #Number of images per batch
    input_shape=(224, 224, 3),            #Input shape to resize all images to
    stop_training_flag={"stop": False},   #Flag for manually stopping training (optional)
    shuffle=True                          #Shuffle the dataset after each epoch
)
#To get an example batch image images and labels
X, y = data_gen[0] #Returns X, a batch of images, and y, a batch of 0s and 1s (i.e. the labels)
```

4.7.3 Inputs and Outputs

```
> Input:
  - Folder of .jpg images
  - Corresponding .txt files (YOLO format):
      - If object is present: "0 x_center y_center width height"
      - If object is absent: (empty .txt file)

> Output:
  - X: NumPy array of shape (batch_size, height, width, channels)
  - y: NumPy array of binary labels (1 if object present, 0 if not)
```

4.8 Suggested Workflow

To use each of these files together to create, train, evaluate and deploy a model, you would follow the order of the workflow below:

1. Prepare your dataset in class-based folders
2. Train a model (`model_trainer.py`)
3. Evaluate it (`model_evaluator.py`)
4. Quantise for deployment (`model_quantiser.py`)
5. Deploy to field hardware (Deployment Guide)
6. Optionally visualize attention maps (`saliency_map_evaluator.py`)

4.9 Common Bugs and How to Address Them

1. **Dataset Format Issues**: Ensure your dataset is in YOLO annotation format. Incorrect formatting can cause errors during data loading, refer to the notes on YOLO annoation format above to help with this.
2. **Class Imbalance**: Imbalanced datasets can lead to poor model performance. Balance your dataset with equal parts of positive (images containing your object of interest) and negative (background image) samples.
3. **Memory Errors During Training**: If you encounter memory issues, reduce the batch size or input image resolution.
4. **Quantisation Errors**: It is normal for quantisation to cause a slight degredation in model accuracy, but if this is large, consider ensuring your representative dataset is diverse.
5. **Deployment Issues on Microcontrollers**: Verify that the model fits within the memory constraints of the target device. Enable PSRAM on ESP32-S3 if needed.

Chapter 5

Namespace Documentation

5.1 callbacks Namespace Reference

Classes

- class SaveWeightsCallback

5.2 generator Namespace Reference

Classes

- class CustomDataGenerator

Functions

- get_augmenter (input_size=(224, 224))

5.2.1 Function Documentation

5.2.1.1 get_augmenter()

```
get_augmenter (
              input_size = (224, 224))
```

```
Create an image augmenter using the imgaug library.

Args:
    input_size (tuple): Desired output size of the images (height, width).

Returns:
    iaa.Sequential: Augmentation pipeline.
```

5.3 model_evaluator Namespace Reference

Classes

- class ModelEvaluator

Functions

- main ()

5.3.1 Function Documentation

5.3.1.1 main()

```
main ()
```

```
Main function for parsing arguments and running the evaluation.
```

5.4 model_loader Namespace Reference

Classes

- class ModelLoader

5.5 model_quantiser Namespace Reference

Classes

- class ModelQuantiser

Functions

- main ()

5.5.1 Function Documentation

5.5.1.1 main()

```
main ()
```

```
Main function to handle command-line arguments and quantise the model.
```

5.6 model_trainer Namespace Reference

Classes

- class ModelTrainer

Functions

- parse_args ()
- main ()

5.6.1 Function Documentation

5.6.1.1 main()

```
main ()
```

```
Main function for training a model.
```

5.6.1.2 parse_args()

```
parse_args ()
```

```
Parse command-line arguments.

Returns:
    argparse.Namespace: Parsed arguments.
```

5.7 saliency_map_evaluator Namespace Reference

Classes

- class SaliencyMapGenerator

Functions

- main ()

5.7.1 Function Documentation

5.7.1.1 main()

```
main ()
```

```
Main function to parse arguments and generate the saliency map.
```

5.8 tflite_model_runner Namespace Reference

Classes

- class TFLiteModelRunner

Chapter 6

Class Documentation

6.1 CustomDataGenerator Class Reference

Inherits Sequence.

Public Member Functions

- __init__ (self, data_directory, batch_size, input_shape, stop_training_flag={"stop":False}, shuffle=True)
- load_image_and_label_paths (self)
- __len__ (self)
- read_img (self, path)
- __getitem__ (self, index)
- read_binary_label (self, label_path)
- on_epoch_end (self)

Public Attributes

- data_directory = data_directory
- batch_size = batch_size
- input_shape = input_shape[:2]
- int nr_channels = input_shape[2]
- shuffle = shuffle
- stop_training_flag = stop_training_flag
- image_paths
- label_paths = self.load_image_and_label_paths()
- indexes = np.arange(len(self.image_paths))
- augmenter = get_augmenter(input_shape[:2])

6.1.1 Detailed Description

```
Custom data generator for Keras models.

This generator deals with YOLO-format object detection data annotations and loads them into a Keras Sequence s
This is done by: 1) loading images and their corresponding labels, 2) resizing them to fit model input shape,
Preprocessing is facilitated by the mobilenet_v2 preprocess_input function, accessed through Keras and augment
If the input shape contains only 1 colour channel, then preprocessing loads the images as greyscale using oper
Each Generator yields batches for training or evaluation, batch size is selected at instantation.

CustomDataGenerator is iterable, the method __get_item__() can be broken out for debugging purposes by setting

Attributes:
    data_directory (str or list): Directory or list of directories containing images and labels.
    batch_size (int): Number of samples per batch.
    input_shape (tuple): Shape of the input images (height, width, channels).
    stop_training_flag (dict): Dictionary containing a 'stop' flag for early stopping.
    shuffle (bool): Whether to shuffle the dataset at the end of each epoch.
```

6.1.2 Constructor & Destructor Documentation

6.1.2.1 __init__()

```
__init__ (
              self,
              data_directory,
              batch_size,
              input_shape,
              stop_training_flag = {"stop":False},
              shuffle = True)
```

```
Initialize the data generator.

Args:
    data_directory (str or list): Directory or list of directories with image and label files.
    batch_size (int): Number of samples per batch.
    input_shape (tuple): Shape of the input images (height, width, channels).
    stop_training_flag (dict): Dictionary with a 'stop' key to signal early stopping, a useful way to stop the
    shuffle (bool): Whether to shuffle data after each epoch. Defaults to True.
```

6.1.3 Member Function Documentation

6.1.3.1 __getitem__()

```
__getitem__ (
              self,
              index)
```

```
Generate one batch of data. This method will raise a StopIteration Exception if self.stop_training_flag is set

Args:
    index (int): Index of the batch.

Returns:
    tuple: Batch of images (X) and labels (y).
```

6.1.3.2 __len__()

```
__len__ (
              self)
```

```
Compute the number of batches per epoch.

Returns:
    int: Number of batches.
```

6.1.3.3 load_image_and_label_paths()

```
load_image_and_label_paths (
              self)
```

```
Load paths to image and label files.

Returns:
    tuple: Lists of image paths and corresponding label paths.
```

6.1.3.4 on_epoch_end()

```
on_epoch_end (
              self)
```

```
Shuffle the dataset at the end of each epoch, if enabled.
```

6.1.3.5 read_binary_label()

```
read_binary_label (
              self,
              label_path)
```

```
Read a binary label from a label file.

Args:
    label_path (str): Path to the label file.

Returns:
    float: Binary label (1.0 for presence, 0.0 for absence).
```

6.1.3.6 read_img()

```
read_img (
              self,
              path)
```

```
Read and preprocess an image.

Args:
    path (str): Path to the image file.

Returns:
    np.ndarray: Preprocessed image.
```

6.1.4 Member Data Documentation

6.1.4.1 augmenter

```
augmenter = get_augmenter(input_shape[:2])
```

6.1.4.2 batch_size

```
batch_size = batch_size
```

6.1.4.3 data_directory

```
data_directory = data_directory
```

6.1.4.4 image_paths

```
image_paths
```

6.1.4.5 indexes

```
indexes = np.arange(len(self.image_paths))
```

6.1.4.6 input_shape

```
input_shape = input_shape[:2]
```

6.1.4.7 label_paths

```
label_paths = self.load_image_and_label_paths()
```

6.1.4.8 nr_channels

```
int nr_channels = input_shape[2]
```

6.1.4.9 shuffle

```
shuffle = shuffle
```

6.1.4.10 stop_training_flag

```
stop_training_flag = stop_training_flag
```

The documentation for this class was generated from the following file:

- generator.py

6.2 ModelEvaluator Class Reference

Public Member Functions

- __init__ (self, batch_size, weights_path, test_data_dir)
- load_model (self)
- create_data_generator (self, input_shape)
- evaluate (self)

Public Attributes

- batch_size = batch_size
- weights_path = weights_path
- test_data_dir = test_data_dir
- dict stop_training_flag = {'stop': False}

6.2.1 Detailed Description

```
A class to handle model evaluation.

Attributes:
    batch_size (int): Batch size for evaluation.
    weights_path (str): Path to the trained model weights.
    test_data_dir (str): Directory containing validation data.
```

6.2.2 Constructor & Destructor Documentation

6.2.2.1 __init__()

```
__init__ (
              self,
              batch_size,
              weights_path,
              test_data_dir)
```

```
Initialize the ModelEvaluator with given parameters.

Args:
    batch_size (int): Batch size for evaluation.
    weights_path (str): Path to the trained model weights.
    val_data_dir (str): Directory containing validation data.
```

6.2.3 Member Function Documentation

6.2.3.1 create_data_generator()

```
create_data_generator (
              self,
              input_shape)
```

```
Create the data generator for test data.

Args:
    input_shape (tuple): Input shape of the model.

Returns:
    CustomDataGenerator: Instance of the custom data generator.
```

6.2.3.2 evaluate()

```
evaluate (
              self)
```

```
Evaluate the model on the test data.

Returns:
    tuple: Test data loss and accuracy.
```

6.2.3.3 load_model()

```
load_model (
              self)
```

```
Load the Keras model from the specified weights path.

Returns:
    tf.keras.Model: Compiled Keras model.
```

6.2.4 Member Data Documentation

6.2.4.1 batch_size

```
batch_size = batch_size
```

6.2.4.2 stop_training_flag

```
dict stop_training_flag = {'stop':  False}
```

6.2.4.3 test_data_dir

```
test_data_dir = test_data_dir
```

6.2.4.4 weights_path

```
weights_path = weights_path
```

The documentation for this class was generated from the following file:

- model_evaluator.py

6.3 ModelLoader Class Reference

Static Public Member Functions

- create_model (input_shape, alpha, dropout_rate=0.5, freeze_base=False)
- load_keras_model (str weights_path)

6.3.1 Detailed Description

```
A utility class to load Keras and TFLite models.
```

6.3.2 Member Function Documentation

6.3.2.1 create_model()

```
create_model (
              input_shape,
              alpha,
              dropout_rate = 0.5,
              freeze_base = False)  [static]
```

```
Create a binary classification model using MobileNetV2 as the base. This method uses the MobileNetv2 implement

Args:
    input_shape (tuple): Input shape for the model (height, width, channels).
    alpha (float): Width multiplier for MobileNetV2.
    dropout_rate (float): Dropout rate for the dropout layer. Default is 0.5.
    freeze_base (bool): Whether or not to make the base model trainable.
Returns:
    tf.keras.Model: Compiled Keras model.
```

6.3.2.2 load_keras_model()

```
load_keras_model (
             str weights_path)  [static]
```

```
Load a trained Keras model from an HDF5 file and print its summary.

Args:
    weights_path (str): Path to the .hdf5 weights file.

Returns:
    tf.keras.Model: Loaded Keras model.
```

The documentation for this class was generated from the following file:

- model_loader.py

6.4 ModelQuantiser Class Reference

Public Member Functions

- __init__ (self, weights_file, representative_dataset, representative_example_nr)
- quantise_model (self, output_path)

Public Attributes

- weights_file = weights_file
- representative_dataset = representative_dataset
- representative_example_nr = representative_example_nr
- model = self._load_model()
- input_shape = self.model.input_shape[1:]
- data_generator

Protected Member Functions

- _load_model (self)
- _test_quantised_model (self, tflite_quant_model)

6.4.1 Detailed Description

```
A class to handle the quantization of a Keras model to TFLite with INT8 precision.

Attributes:
    model (tf.keras.Model): The model to be quantised.
    representative_dataset (str): Path to the representative dataset for quantization.
```

6.4.2 Constructor & Destructor Documentation

6.4.2.1 __init__()

```
__init__ (
              self,
              weights_file,
              representative_dataset,
              representative_example_nr)
```

```
Initialize the ModelQuantiser.

Args:
    weights_file (str): Path to the Keras model weights file.
    representative_dataset (str): Directory containing the representative dataset.
```

6.4.3 Member Function Documentation

6.4.3.1 _load_model()

```
_load_model (
              self)  [protected]
```

```
Load the Keras model from the weights file.

Returns:
    tf.keras.Model: Loaded model.
```

6.4.3.2 _test_quantised_model()

```
_test_quantised_model (
              self,
              tflite_quant_model)  [protected]
```

```
Test the quantised TFLite model by loading it into a TFLite interpreter.

Args:
    tflite_quant_model (bytes): The quantised TFLite model.
```

6.4.3.3 quantise_model()

```
quantise_model (
              self,
              output_path)
```

```
Quantise the model to TFLite format with INT8 precision.

Args:
    output_path (str): Path to save the quantised TFLite model.
```

6.4.4 Member Data Documentation

6.4.4.1 data_generator

```
data_generator
```

Initial value:

```
=  CustomDataGenerator(
            data_directory=[self.representative_dataset],
            batch_size=1,
            input_shape=self.input_shape,
            stop_training_flag=False,
            shuffle=True
        )
```

6.4.4.2 input_shape

```
input_shape = self.model.input_shape[1:]
```

6.4.4.3 model

```
model = self._load_model()
```

6.4.4.4 representative_dataset

```
representative_dataset = representative_dataset
```

6.4.4.5 representative_example_nr

```
representative_example_nr = representative_example_nr
```

6.4.4.6 weights_file

```
weights_file = weights_file
```

The documentation for this class was generated from the following file:

- model_quantiser.py

6.5 ModelTrainer Class Reference

Public Member Functions

- __init__ (self, config)
- train (self)

Public Attributes

- config = config
- model = self._load_model()
- train_generator = self._create_data_generator(config['train_data_dir'])
- val_generator = self._create_data_generator(config['val_data_dir'], shuffle=False)

Protected Member Functions

- _load_model (self)
- _create_data_generator (self, data_directory, shuffle=True)

6.5.1 Detailed Description

```
A class to handle the training of a model using a custom data generator.

Attributes:
    model (tf.keras.Model): The model to be trained.
    train_generator (CustomDataGenerator): Data generator for training data.
    val_generator (CustomDataGenerator): Data generator for validation data.
```

6.5.2 Constructor & Destructor Documentation

6.5.2.1 __init__()

```
__init__ (
              self,
              config)
```

```
Initialize the ModelTrainer.

Args:
    config (dict): Configuration dictionary with training parameters.
```

6.5.3 Member Function Documentation

6.5.3.1 _create_data_generator()

```
_create_data_generator (
              self,
              data_directory,
              shuffle = True)  [protected]
```

```
Create an instance of CustomDataGenerator.

Args:
    data_directory (str or list): Directory containing the dataset.
    shuffle (bool): Whether to shuffle data. Defaults to True.

Returns:
    CustomDataGenerator: Initialized data generator.
```

6.5.3.2 _load_model()

```
_load_model (
              self) [protected]
```

```
Load the model using ModelLoader.

Returns:
    tf.keras.Model: Compiled model.
```

6.5.3.3 train()

```
train (
              self)
```

```
Train the model using the data generators.
```

6.5.4 Member Data Documentation

6.5.4.1 config

```
config = config
```

6.5.4.2 model

```
model = self._load_model()
```

6.5.4.3 train_generator

```
train_generator = self._create_data_generator(config['train_data_dir'])
```

6.5.4.4 val_generator

```
val_generator = self._create_data_generator(config['val_data_dir'], shuffle=False)
```

The documentation for this class was generated from the following file:

- model_trainer.py

6.6 SaliencyMapGenerator Class Reference

Public Member Functions

- __init__ (self, weights_file)
- generate_saliency_map (self, input_image_path, output_path)

Public Attributes

- weights_file = weights_file
- model = self._load_model()
- input_shape = self.model.input_shape[1:]

Protected Member Functions

- _load_model (self)

Static Protected Member Functions

- _preprocess_image (image_path, input_shape)

6.6.1 Detailed Description

```
A class to generate a saliency map for an input image using a trained model.
```

6.6.2 Constructor & Destructor Documentation

6.6.2.1 __init__()

```
__init__ (
              self,
              weights_file)
```

```
Initialize the SaliencyMapGenerator.

Args:
    weights_file (str): Path to the trained model weights file.
```

6.6.3 Member Function Documentation

6.6.3.1 _load_model()

```
_load_model (
              self)  [protected]
```

```
Load the model using the provided weights file.

Returns:
    tf.keras.Model: Loaded Keras model.
```

6.6.3.2 _preprocess_image()

```
_preprocess_image (
              image_path,
              input_shape)  [static], [protected]
```

```
Preprocess the input image.

Args:
    image_path (str): Path to the input image.
    input_shape (tuple): Shape to resize the image to.

Returns:
    tuple: Preprocessed image array and the original image.
```

6.6.3.3 generate_saliency_map()

```
generate_saliency_map (
              self,
              input_image_path,
              output_path)
```

```
Generate and save a saliency map for the input image.

Args:
    input_image_path (str): Path to the input image.
    output_path (str): Path to save the saliency map.
```

6.6.4 Member Data Documentation

6.6.4.1 input_shape

```
input_shape = self.model.input_shape[1:]
```

6.6.4.2 model

```
model = self._load_model()
```

6.6.4.3 weights_file

```
weights_file = weights_file
```

The documentation for this class was generated from the following file:

- saliency_map_evaluator.py

6.7 SaveWeightsCallback Class Reference

Inherits Callback.

Public Member Functions

- __init__ (self, save_dir, save_format="tf")
- on_epoch_end (self, epoch, logs=None)

Public Attributes

- save_dir = save_dir
- save_format = save_format

6.7.1 Constructor & Destructor Documentation

6.7.1.1 __init__()

```
__init__ (
              self,
              save_dir,
              save_format = "tf")
```

6.7.2 Member Function Documentation

6.7.2.1 on_epoch_end()

```
on_epoch_end (
              self,
              epoch,
              logs = None)
```

6.7.3 Member Data Documentation

6.7.3.1 save_dir

```
save_dir = save_dir
```

6.7.3.2 save_format

```
save_format = save_format
```

The documentation for this class was generated from the following file:

- callbacks.py

6.8 TFLiteModelRunner Class Reference

Static Public Member Functions

- load_tflite_model (str tflite_path)
- get_tflite_input_output_details (interpreter)

6.8.1 Member Function Documentation

6.8.1.1 get_tflite_input_output_details()

```
get_tflite_input_output_details (
              interpreter)  [static]
```

```
Get input and output details of a loaded TFLite model.

Args:
    interpreter (tf.lite.Interpreter): Loaded TFLite model interpreter.

Returns:
    tuple: (input_details, output_details)
```

6.8.1.2 load_tflite_model()

```
load_tflite_model (
             str tflite_path)  [static]
```

```
Load a TFLite model for inference and print input/output details.

Args:
    tflite_path (str): Path to the .tflite file.

Returns:
    tf.lite.Interpreter: TFLite interpreter with the model loaded.
```

The documentation for this class was generated from the following file:

- tflite_model_runner.py

Index