PeekingDuck

Release developer

CVHub AI Singapore

Oct 04, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is PeekingDuck?

PeekingDuck is an open-source, modular framework in Python, built for Computer Vision (CV) inference. The name "PeekingDuck" is a play on: "Peeking" in a nod to CV; and "Duck" in duck typing used in Python.

1.2 Features

1.2.1 Build realtime CV pipelines

PeekingDuck enables you to build powerful CV pipelines with minimal lines of code.

1.2.2 Leverage on SOTA models

PeekingDuck comes with various state of the art (SOTA) *object detection*, *pose estimation*, *object tracking*, *crowd counting*, and *instance segmentation* models. Mix and match different nodes to construct solutions for various *use cases*.

1.2.3 Create custom nodes

You can create *custom nodes* to meet your own project's requirements. PeekingDuck can also be *imported as a library* to fit into your existing workflows.

1.3 How PeekingDuck Works

Nodes are the building blocks of PeekingDuck. Each node is a wrapper for a pipeline function, and contains information on how other PeekingDuck nodes may interact with it.

PeekingDuck has 6 types of nodes:

A **pipeline** governs the behavior of a chain of nodes. The diagram below shows a sample pipeline. Nodes in a pipeline are called in sequential order, and the output of one node will be the input to another. For example, `input.visual` produces *img*, which is taken in by `model.yolo`, and `model.yolo` produces *bboxes*, which is taken in by `draw.bbox`. For ease of visualization, not all the inputs and outputs of these nodes are included in this diagram.

1.4 Acknowledgements

This project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG-RP-2019-050). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

1.5 License

PeekingDuck is under the open source Apache License 2.0 (:

Even so, your organization may require legal proof of its right to use PeekingDuck, due to circumstances such as the following:

- Your organization is using PeekingDuck in a jurisdiction that does not recognize this license
- Your legal department requires a license to be purchased
- Your organization wants to hold a tangible legal document as evidence of the legal right to use and distribute PeekingDuck

Contact us if any of these circumstances apply to you.

1.6 Communities

- AI Singapore community forum
- Discuss on GitHub

CHAPTER

TWO

GETTING STARTED

2.1 Documentation Convention

Parts of this documentation and the tutorials are run from the command-line interface (CLI) environment, e.g., via *Terminal* in Linux/macOS, or via *Anaconda* in Windows. There will be examples of commands you need to type as inputs and text that PeekingDuck will display as outputs. The input commands can be dependent on the current folder where they are typed.

The following text color scheme is used to illustrate these different contexts:

Color	Context	Example
Blue	Current folder	[~user/src]
Green	User input: what you type in	> peekingduck --version
Black	PeekingDuck's output	peekingduck, version v1.2.0

The command prompt is assumed to be the symbol >, your home directory is assumed to be `~user`, and the symbol means to press the <`Enter`> key.

Putting it altogether, a sample terminal session looks like this:

Terminal Session

```
[~user/src] > peekingduck --version
peekingduck, version v1.2.0
```

2.2 Standard Install

2.2.1 Install PeekingDuck

Then run:

PeekingDuck supports Python 3.6 to 3.9.

It is recommended to install PeekingDuck in a Python virtual environment (such as `pkd` in the above commands), as it creates an isolated environment for a Python project to install its own dependencies and avoid package version conflicts with other projects.

Note: For Apple Silicon Mac users, please see *Custom Install - Apple Silicon Mac*.

2.2.2 Verify PeekingDuck Installation

To check that PeekingDuck is installed successfully, run the following command:

Terminal Session

```
[~user] > peekingduck verify-install
```

Changed in version 1.3.0: The verify installation command has been changed from `--verify_install` to `verify-install`.

You should see a video of a person waving his hand (taken from here) with bounding boxes overlaid as shown below:

The video will auto-close when it is run to the end (about 20 seconds, depending on system speed).
To exit earlier, click to select the video window and press q.

2.3 Custom Install

This section covers advanced PeekingDuck installation steps for users with ARM64 devices or Apple Silicon Macs.

2.3.1 Arm64

To install PeekingDuck on an ARM-based device, such as a Raspberry Pi, include the `--no-dependencies` flag, and separately install the other dependencies listed in PeekingDuck's [requirements.txt]:

Terminal Session

```
[~user] > pip install peekingduck --no-dependencies
[~user] > [ install additional dependencies as specified within requirements.txt ]
```

Verify the installation using:

Terminal Session

```
[~user] > peekingduck verify-install
```

See here for changes to the verify installation command in version 1.3.0.

You should see a video of *a person waving his hand with bounding boxes overlaid*.

2.3.2 Apple Silicon Mac

Apple released their advanced ARM-based Apple Silicon M1 chip in late 2020, a significant change from the previous Intel processors. We've successfully installed PeekingDuck on Apple Silicon Macs running macOS Big Sur and macOS Monterey.

1. Prerequisites:
   - Install homebrew
   - Install miniforge using homebrew:

   Terminal Session

   ```
   [~user] > brew install miniforge
   ```

2. Create conda virtual environment and install base packages:

   Terminal Session

   ```
   [~user] > conda create -n pkd python=3.8
   [~user] > conda activate pkd
   [~user] > conda install click colorama opencv openblas pyyaml requests scipy shapely tqdm typeguard
   ```

3. Install Apple's Tensorflow build that is optimized for Apple Silicon Macs:
   - For macOS Monterey:

   Terminal Session

   ```
   [~user] > conda install -c apple tensorflow-deps
   [~user] > pip install tensorflow-macos tensorflow-metal
   ```

   - For macOS Big Sur:

   Terminal Session

   ```
   [~user] > conda install -c apple tensorflow-deps=2.6.0
   ```

```
[~user] > pip install tensorflow-estimator==2.6.0 tensorflow-macos==2.6.0
[~user] > pip install tensorflow-metal==0.2.0
```

4. Install PyTorch (currently CPU-only):

Terminal Session

```
[~user] > pip install torch torchvision
```

5. Install PeekingDuck and verify installation:

Terminal Session

```
[~user] > pip install peekingduck --no-dependencies
[~user] > peekingduck verify-install
```

See here for changes to the verify installation command in version 1.3.0.

You should see a video of a person waving his hand (taken from here) with bounding boxes overlaid as shown below:

The video will auto-close when it is run to the end (about 20 seconds, depending on system speed).
To exit earlier, click to select the video window and press q.

CHAPTER

THREE

TUTORIALS

The tutorials are presented in order of increasing difficulty, from the basic *Hello Computer Vision* to the advanced *Peaking Duck*. It is recommended to go through these tutorials in order, especially if you are new to PeekingDuck.

3.1 "Hello Computer Vision"

Computer Vision (or CV) is a field in AI that develops techniques to help computers to "see" and "understand" the contents of digital images like photographs and videos, and to derive meaningful information. Common CV applications include object detection to detect what objects are present in the image and pose estimation to detect the position of human limbs relative to the body.

PeekingDuck allows you to build a CV pipeline to analyze and process images and/or videos. This pipeline is made up of nodes: each node can perform certain CV-related tasks.

This section presents two basic "hello world" examples to demonstrate how to use PeekingDuck for pose estimation and object detection.

3.1.1 Pose Estimation

To perform pose estimation with PeekingDuck, initialize a new PeekingDuck project using the following commands:

Terminal Session

[~user] > mkdir pose_estimation
[~user] > cd pose_estimation
[~user/pose_estimation] > peekingduck init

peekingduck init will prepare the `pose_estimation` folder for use with PeekingDuck. It creates a default pipeline file called `pipeline_config.yml` and a `src` folder that will be covered in the later tutorials. The `pipeline_config.yml` file looks like this:

```yaml
nodes:
- input.visual:
    source: https://storage.googleapis.com/peekingduck/videos/wave.mp4
- model.posenet
```

(continues on next page)

(continued from previous page)

```
- draw.poses
- output.screen
```

The above forms a **pose estimation pipeline** and it comprises four nodes that do the following:

1. *input.visual*: reads the file `wave.mp4` from PeekingDuck's cloud storage
2. *model.posenet*: runs the PoseNet pose estimation model on it
3. *draw.poses*: draws a human pose skeleton over the person tracking his hand movement
4. *output.screen*: outputs everything onto the screen for display

Now, run the pipeline using

Terminal Session

[~user/pose_estimation] > peekingduck run

You should see the following video of a person waving his hand.
Skeletal poses are drawn on him which track the hand movement.

Fig. 1: PeekingDuck's Pose Estimation Screenshot

You have successfully run a PeekingDuck pose estimation pipeline!

The video will auto-close when it is completed.
To exit earlier, click to select the video window and press q.

3.1.2 Object Detection

To perform object detection, initialize a new PeekingDuck project using the following commands:

Terminal Session

[~user] > mkdir object_detection
[~user] > cd object_detection
[~user/object_detection] > peekingduck init

Then modify `pipeline_config.yml` as follows:

```
1 nodes:
2 - input.visual:
3     source: https://storage.googleapis.com/peekingduck/videos/wave.mp4
4 - model.yolo
5 - draw.bbox
6 - output.screen
```

The key differences between this and the earlier pipeline are:

Line 4: *model.yolo* runs the YOLO object detection model
Line 5: *draw.bbox* draws the bounding box to show the detected person

Run the new **object detection pipeline** with peekingduck run.

You will see the same video with a bounding box surrounding the person.

That's it: you have created a new object detection pipeline by changing only two lines!

Note:

Try replacing `wave.mp4` with your own video file and run both models.
For best effect, your video file should contain people performing some activities.

Fig. 2: PeekingDuck's Object Detection Screenshot

3.1.3 Using a WebCam

If your computer has a webcam attached, you can use it by changing the first `input` node (line 2) as follows:

```
1 nodes:
2 - input.visual:
3     source: 0          # use webcam for live video
4 - model.posenet        # use pose estimation model
5 - draw.poses           # draw skeletal poses
6 - output.screen
```

Now do a peekingduck run and you will see yourself onscreen. Move your hands around and see PeekingDuck tracking your poses.

To exit, click to select the video window and press q.

Note: PeekingDuck assumes the webcam is defaulted to input source 0. If your system is configured differently, you would have to specify the input source by changing the *`input.visual`* configuration. See *changing node configuration*.

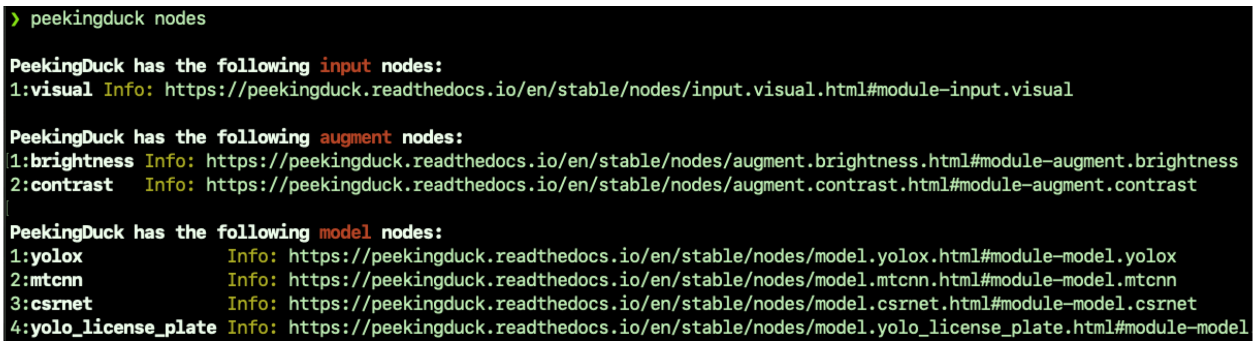
3.1.4 Pipelines, Nodes and Configs

PeekingDuck comes with a rich collection of nodes that you can use to create your own CV pipelines. Each node can be customized by changing its configurations or settings.

To get a quick overview of PeekingDuck's nodes, run the following command:

Terminal Session

```
[~user] > peekingduck nodes
```



You will see a comprehensive list of all PeekingDuck's nodes with links to their `readthedocs` pages for more information.

PeekingDuck supports 6 types of nodes:

A PeekingDuck pipeline is created by stringing together a series of nodes that perform a logical sequence of operations. Each node has its own set of configurable settings that can be modified to change its behavior. An example pipeline is shown below:

3.1.5 Bounding Box vs Image Coordinates

PeekingDuck has two (x, y) coordinate systems, with top-left corner as origin $(0, 0)$:

- **Absolute image coordinates**

  For an image of width W and height H, the absolute image coordinates are integers from $(0, 0)$ to $(W - 1, H - 1)$. E.g., for a 720 x 480 image, the absolute coordinates range from $(0, 0)$ to $(719, 479)$.

- **Relative bounding box coordinates**

  For an image of width W and height H, the relative image coordinates are real numbers from $(0.0, 0.0)$ to $(1.0, 1.0)$. E.g., for a 720 x 480 image, the relative coordinates range from $(0.0, 0.0)$ to $(1.0, 1.0)$.

This means that in order to draw a bounding box onto an image, the bounding box relative coordinates would have to be converted to the image absolute coordinates.

Using the above figure as an illustration, the bounding box coordinates are given as $(0.18, 0.10)$ top-left and $(0.52, 0.88)$ bottom-right. To convert them to image coordinates, multiply the x-coordinates by the image width and the y-

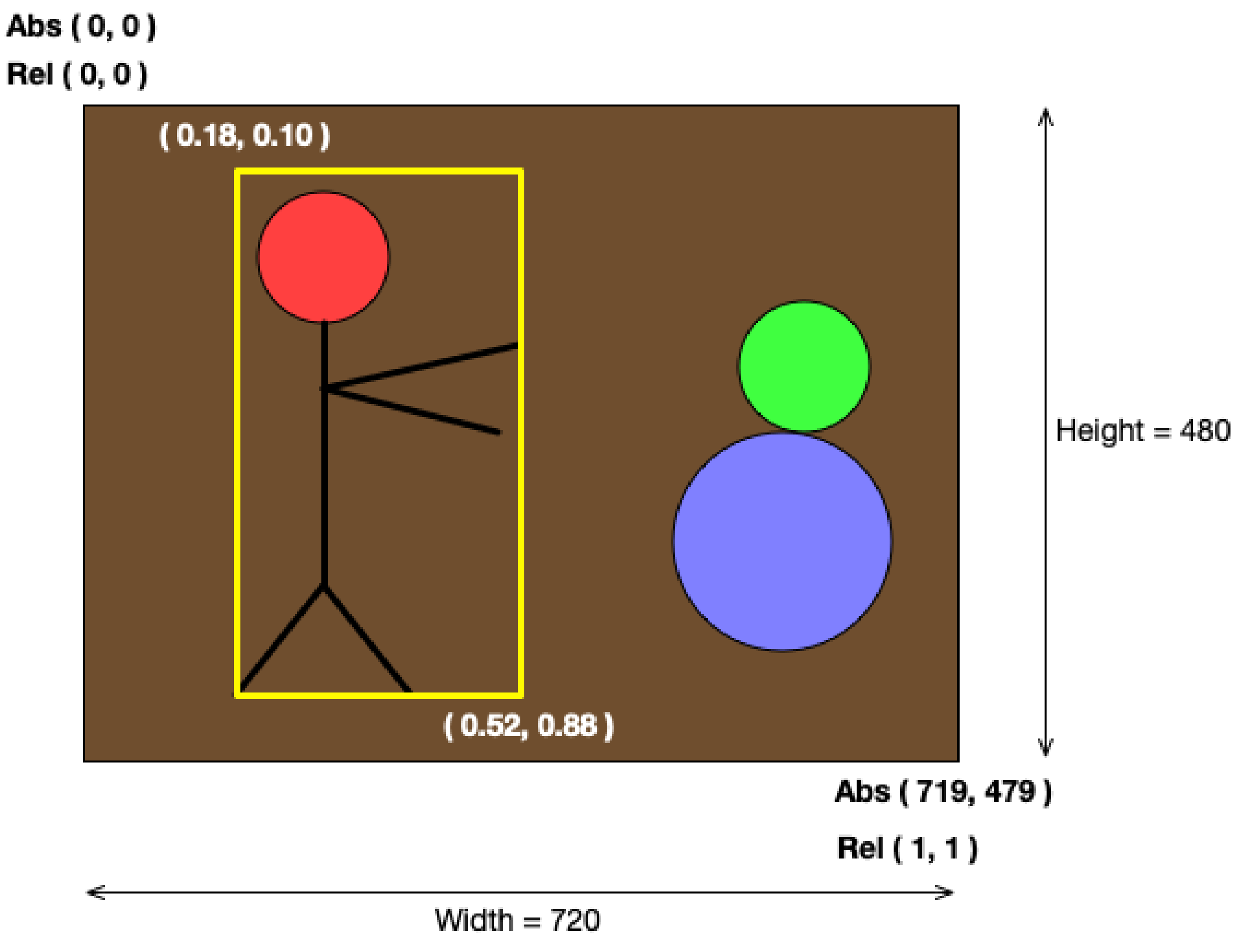


Fig. 3: PeekingDuck's Image vs Bounding Box Coordinates

coordinates by the image height, and round the results into integers.

$$0.18 \rightarrow 0.18 \times 720 = 129.6 = 130 \quad (int)$$
$$0.10 \rightarrow 0.10 \times 480 = 48.0 = 48 \quad (int)$$

$$0.52 \rightarrow 0.52 \times 720 = 374.4 = 374 \quad (int)$$
$$0.88 \rightarrow 0.88 \times 480 = 422.4 = 422 \quad (int)$$

Thus, the image coordinates are $(130, 48)$ top-left and $(374, 422)$ bottom-right.

Note: The `model` nodes return results in relative coordinates.

3.2 Duck Confit

This tutorial presents intermediate recipes for cooking up new PeekingDuck pipelines by modifying the nodes and their configs.

3.2.1 More Object Detection

This section will demonstrate how to change the settings of PeekingDuck's nodes to vary their functionalities.

If you had completed the earlier *object detection tutorial*, you will have the necessary folder and can skip to the next step. Otherwise, create a new PeekingDuck project as shown below:

Terminal Session

[~user] > mkdir object_detection
[~user] > cd object_detection
[~user/object_detection] > peekingduck init

Next, download this demo video cat_and_computer.mp4 and save it into the `object_detection` folder.

The folder should contain the following:

```
object_detection/
├── cat_and_computer.mp4
├── pipeline_config.yml
└── src/
```

To perform object detection on the `cat_and_computer.mp4` file, edit the `pipeline_config.yml` file as follows:

```yaml
nodes:
- input.visual:
    source: cat_and_computer.mp4
- model.yolo:
    detect: ["cup", "cat", "laptop", "keyboard", "mouse"]
- draw.bbox:
    show_labels: True    # configure draw.bbox to display object labels
- output.screen
```

Here is a step-by-step explanation of what has been done:

Line 2 *input.visual*: tells PeekingDuck to load the `cat_and_computer.mp4`.
Line 4 *model.yolo*: by default, the YOLO model detects `person` only.
The `cat_and_computer.mp4` contains other classes of objects like cup, cat, laptop, etc.
So we have to change the model settings to detect the other object classes.
Line 6 *draw.bbox*: reconfigure this node to display the detected object class label.

Run the above with the command peekingduck run.

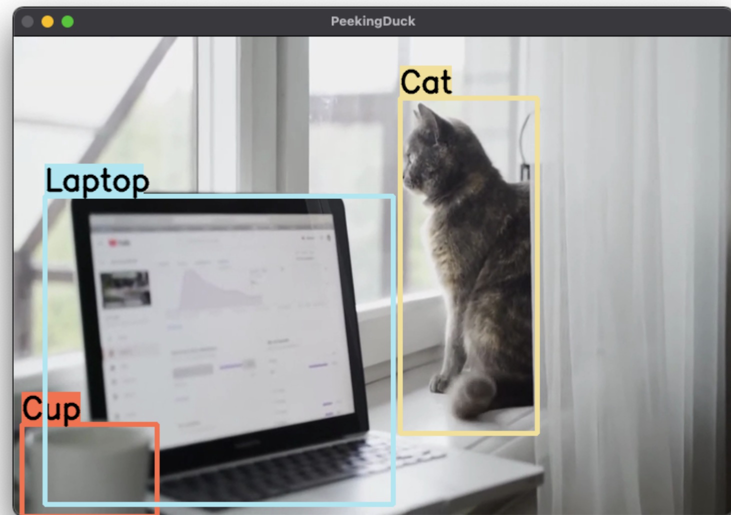


Fig. 4: Cat and Computer Screenshot

You should see a display of the `cat_and_computer.mp4` with the various objects being highlighted by PeekingDuck in bounding boxes. The 30-second video will auto-close at the end, or you can press q to end early.

Note: The YOLO model can detect 80 different *object classes*. By default, it only detects the `"person"` class. Use `detect: ["*"]` in the `pipeline_config.yml` to configure the model to detect all 80 classes.

3.2.2 Record Video File with FPS

This section demonstrates how to record PeekingDuck's output into a video file. In addition, we will modify the pipeline by adding new nodes to calculate the frames per second (FPS) and to show the FPS.

Edit `pipeline_config.yml` as shown below:

```
nodes:
- input.visual:
    source: cat_and_computer.mp4
- model.yolo:
```

(continues on next page)

(continued from previous page)

```
5      detect: ["cup", "cat", "laptop", "keyboard", "mouse"]
6    - draw.bbox:
7      show_labels: True
8    - dabble.fps                      # add new dabble node
9    - draw.legend:                    # show fps
10     show: ["fps"]
11   - output.screen
12   - output.media_writer:            # add new output node
13     output_dir: /folder/to/save/video  # this is a folder name
```

The additions are:

Line 8 *dabble.fps*: adds new *dabble* node to the pipeline. This node calculates the FPS.

Line 9 *draw.legend*: adds new *draw* node to display the FPS.

Line 12 *output.media_writer*: adds new *output* node to save PeekingDuck's output to a local video file. It requires a local folder path. If the folder is not available, PeekingDuck will create the folder automatically. The filename is auto-generated by PeekingDuck based on the input source.

Run the above with the command peekingduck run. You will see the same video being played, but now it has the FPS counter. When the video ends, an `mp4` video file will be created and saved in the specified folder.

Note: You can view all the available nodes and their respective configurable settings in PeekingDuck's *API documentation*.

3.2.3 Configuration - Behind the Scenes

Here is an explanation on what goes on behind the scenes when you configure a node. Every node has a set of default configuration. For instance, *draw.bbox* default configuration is:

```
input: ["bboxes", "img", "bbox_labels"]
output: ["none"]

show_labels: False
```

The keys `input` and `output` are compulsory and common across every node.

`input` specifies the data types the node would consume, to be read from the pipeline.

`output` specifies the data types the node would produce, to be put into the pipeline.

By default, `show_labels` is disabled. When you enable it with `show_labels: True`, what PeekingDuck does is to override the default `show_labels: False` configuration with your specified `True` value. You will see another instance of this at work in the advanced *Peaking Duck* tutorial on *Tracking People Within a Zone*.

3.2.4 Augmenting Images

PeekingDuck has a class of *augment* nodes that can be used to perform preprocessing or postprocessing of images/videos. Augment currently lets you modify the brightness and contrast, and remove distortion from a wide-angle camera image. For more details on image undistortion, refer to the documentation on *augment.undistort* and *dabble.camera_calibration*.

The pipeline_config.yml below shows how to use the *augment.brightness* node within the pipeline:

```
nodes:
- input.visual:
    source: https://storage.googleapis.com/peekingduck/videos/wave.mp4
- model.yolo
- augment.brightness:
    beta: 50         # ranges from -100 (darken) to +100 (brighten)
- draw.bbox
- output.screen
```

The following figure shows the difference between the original vs the brightened image:

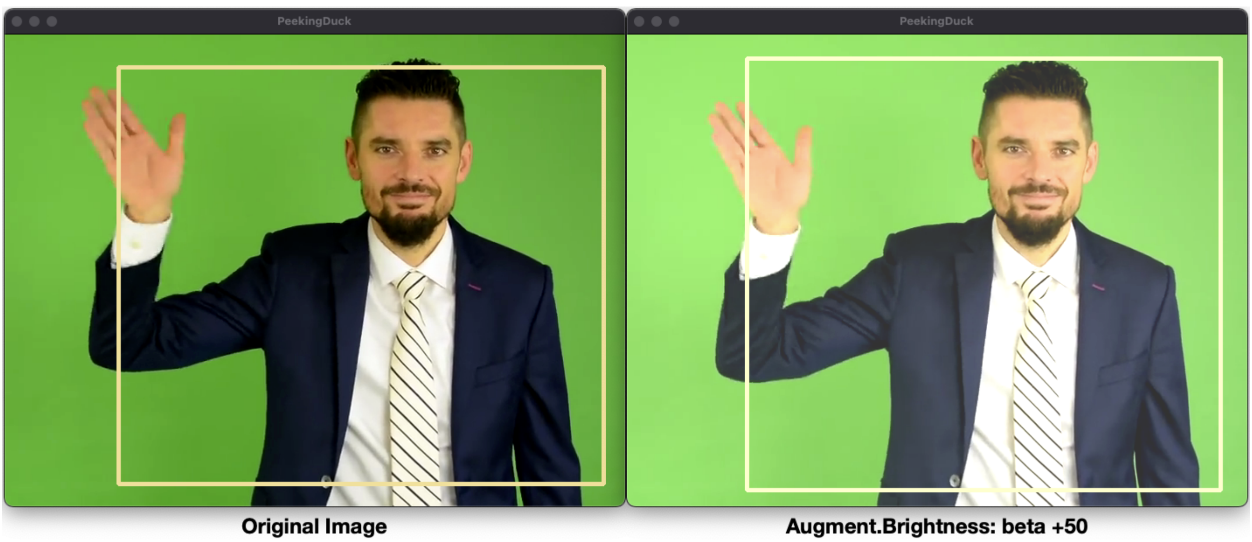


Fig. 5: Augment Brightness: Original vs Brightened Image

Note:

Royalty free video of cat and computer from: https://www.youtube.com/watch?v=-C1TEGZavko
Royalty free video of man waving hand from: https://www.youtube.com/watch?v=IKj_z2hgYUM

3.3 Custom Nodes

This tutorial will show you how to create your own custom nodes to run with PeekingDuck. Perhaps you'd like to take a snapshot of a video frame, and post it to your API endpoint; or perhaps you have a model trained on a custom dataset, and would like to use PeekingDuck's `input`, `draw`, and `output` nodes. PeekingDuck is designed to be very flexible — you can create your own nodes and use them with ours.

Let's start by creating a new PeekingDuck project:

Terminal Session

[~user] > mkdir custom_project
[~user] > cd custom_project
[~user/custom_project] > peekingduck init

This creates the following `custom_project` folder structure:

```
custom_project/
├── pipeline_config.yml
└── src/
    └── custom_nodes/
        └── configs/
```

The sub-folders `src`, `custom_nodes`, and `configs` are empty: they serve as placeholders for contents to be added.

3.3.1 Recipe 1: Object Detection Score

When the YOLO object detection model detects an object in the image, it assigns a bounding box and a score to it. This score is the "confidence score" which reflects how likely the box contains an object and how accurate is the bounding box. It is a decimal number that ranges from 0.0 to 1.0 (or 100%). This number is internal and not readily viewable.

We will create a custom node to retrieve this score and display it on screen. This tutorial will use the cat_and_computer.mp4 video from the earlier *object detection tutorial*. Copy it into the `custom_project` folder.

Use the following command to create a custom node: peekingduck create-node It will prompt you to answer several questions. Press <Enter> to accept the default `custom_nodes` folder name, then key in draw for node type and score for node name. Finally, press <Enter> to answer `Y` when asked to proceed.

The entire interaction is shown here, the answers you type in are shown in green text:

Terminal Session

[~user/custom_project] > peekingduck create-node
Creating new custom node…
Enter node directory relative to ~user/custom_project [src/custom_nodes]:
Select node type (input, augment, model, draw, dabble, output): draw
Enter node name [my_custom_node]: score

Node directory: ~user/custom_project/src/custom_nodes

```
Node type: draw
Node name: score

Creating the following files:
    Config file: ~user/custom_project/src/custom_nodes/configs/draw/score.yml
    Script file: ~user/custom_project/src/custom_nodes/draw/score.py
Proceed? [Y/n]:
Created node!
```

The `custom_project` folder structure should look like this:

```
custom_project/
├── cat_and_computer.mp4
├── pipeline_config.yml
└── src/
    └── custom_nodes/
        ├── configs/
        │   └── draw/
        │       └── score.yml
        └── draw/
            └── score.py
```

`custom_project` now contains **three files** that we need to modify to implement our custom node function.

1. **src/custom_nodes/configs/draw/score.yml**:

   `score.yml` initial content:

```
1  # Mandatory configs
2  # Receive bounding boxes and their respective labels as input. Replace with
3  # other data types as required. List of built-in data types for PeekingDuck can
4  # be found at https://peekingduck.readthedocs.io/en/stable/glossary.html.
5  input: ["bboxes", "bbox_labels"]
6  # Example:
7  # Output `obj_attrs` for visualization with `draw.tag` node and `custom_key` for
8  # use with other custom nodes. Replace as required.
9  output: ["obj_attrs", "custom_key"]
10 
11 # Optional configs depending on node
12 threshold: 0.5 # example
```

   The first file `score.yml` defines the properties of the custom node. Lines 5 and 9 show the mandatory configs `input` and `output`.

   `input` specifies the data types the node would consume, to be read from the pipeline. `output` specifies the data types the node would produce, to be put into the pipeline.

   To display the bounding box confidence score, our node requires three pieces of input data: the bounding box, the score to display, and the image to draw on. These are defined as the data types *bboxes*, *bbox_scores*, and *img* respectively in the *API docs*.

   Our custom node only displays the score on screen and does not produce any outputs for the pipeline, so the output is "*none*".

   There are also no optional configs, so lines 11 - 12 can be removed.

`score.yml` updated content:

```yaml
# Mandatory configs
input: ["img", "bboxes", "bbox_scores"]
output: ["none"]

# No optional configs
```

Note: Comments in yaml files start with # It is possible for a node to have `input: ["none"]`

2. **src/custom_nodes/draw/score.py**:

   The second file `score.py` contains the boilerplate code for creating a custom node. Update the code to implement the desired behavior for the node.

   Show/Hide Code for score.py

```python
"""
Custom node to show object detection scores
"""

from typing import Any, Dict, List, Tuple
import cv2
from peekingduck.pipeline.nodes.abstract_node import AbstractNode

YELLOW = (0, 255, 255)        # in BGR format, per opencv's convention


def map_bbox_to_image_coords(
    bbox: List[float], image_size: Tuple[int, int]
) -> List[int]:
    """This is a helper function to map bounding box coords (relative) to
    image coords (absolute).
    Bounding box coords ranges from 0 to 1
    where (0, 0) = image top-left, (1, 1) = image bottom-right.

    Args:
        bbox (List[float]): List of 4 floats x1, y1, x2, y2
        image_size (Tuple[int, int]): Width, Height of image

    Returns:
        List[int]: x1, y1, x2, y2 in integer image coords
    """
    width, height = image_size[0], image_size[1]
    x1, y1, x2, y2 = bbox
    x1 *= width
    x2 *= width
    y1 *= height
    y2 *= height
    return int(x1), int(y1), int(x2), int(y2)


class Node(AbstractNode):
```

(continues on next page)

(continued from previous page)

```
    """This is a template class of how to write a node for PeekingDuck,
       using AbstractNode as the parent class.
       This node draws scores on objects detected.

    Args:
       config (:obj:`Dict[str, Any]` | :obj:`None`): Node configuration.
    """

    def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
        """Node initializer

        Since we do not require any special setup, it only calls the __init__
        method of its parent class.
        """
        super().__init__(config, node_path=__name__, **kwargs)

    def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
        """This method implements the display score function.
        As PeekingDuck iterates through the CV pipeline, this 'run' method
        is called at each iteration.

        Args:
              inputs (dict): Dictionary with keys "img", "bboxes", "bbox_scores"

        Returns:
              outputs (dict): Empty dictionary
        """

        # extract pipeline inputs and compute image size in (width, height)
        img = inputs["img"]
        bboxes = inputs["bboxes"]
        scores = inputs["bbox_scores"]
        img_size = (img.shape[1], img.shape[0])  # width, height

        for i, bbox in enumerate(bboxes):
            # for each bounding box:
            #   - compute (x1, y1) top-left, (x2, y2) bottom-right coordinates
            #   - convert score into a two decimal place numeric string
            #   - draw score string onto image using opencv's putText()
            #     (see opencv's API docs for more info)
            x1, y1, x2, y2 = map_bbox_to_image_coords(bbox, img_size)
            score = scores[i]
            score_str = f"{score:0.2f}"
            cv2.putText(
                img=img,
                text=score_str,
                org=(x1, y2),
                fontFace=cv2.FONT_HERSHEY_SIMPLEX,
                fontScale=1.0,
                color=YELLOW,
                thickness=3,
            )
```

(continues on next page)

(continued from previous page)

```python

        return {}                 # node has no outputs
```

The updated node code defines a helper function `map_bbox_to_image_coords` to map the bounding box coordinates to the image coordinates, as explained in *this section*.

The `run` method implements the main logic which processes every bounding box to compute its on-screen coordinates and to draw the bounding box confidence score at its left-bottom position.

3. **pipeline_config.yml**:

   `pipeline_config.yml` initial content:

```yaml
nodes:
- input.visual:
    source: https://storage.googleapis.com/peekingduck/videos/wave.mp4
- model.posenet
- draw.poses
- output.screen
```

   This file implements the pipeline. Modify the default pipeline to the one shown below:

   `pipeline_config.yml` updated content:

```yaml
1 nodes:
2 - input.visual:
3     source: cat_and_computer.mp4
4 - model.yolo:
5     detect: ["cup", "cat", "laptop", "keyboard", "mouse"]
6 - draw.bbox:
7     show_labels: True
8 - custom_nodes.draw.score
9 - output.screen
```

   Line 8 adds our custom node into the pipeline where it will be `run` by PeekingDuck during each pipeline iteration.

Execute peekingduck run to see your custom node in action.

Note: Royalty free video of cat and computer from: https://www.youtube.com/watch?v=-C1TEGZavko

3.3.2 Recipe 2: Keypoints, Count Hand Waves

This tutorial will create a custom node to analyze the skeletal keypoints of the person from the wave.mp4 video in the *pose estimation tutorial* and to count the number of times the person waves his hand.

The PoseNet pose estimation model outputs seventeen keypoints for the person corresponding to the different body parts as documented *here*. Each keypoint is a pair of `(x, y)` coordinates, where `x` and `y` are real numbers ranging from 0.0 to 1.0 (using *relative coordinates*).

Starting with a newly initialized PeekingDuck folder, call peekingduck create-node to create a new `dabble.wave` custom node as shown below:

Terminal Session

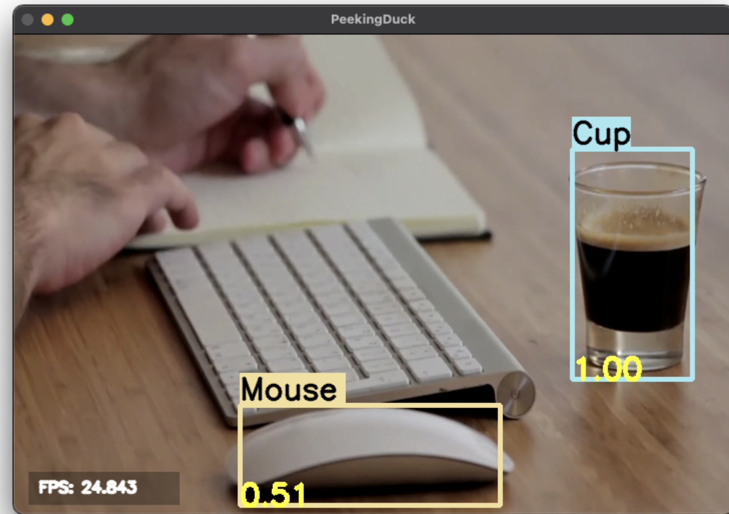


Fig. 6: Custom Node Showing Object Detection Scores

```
[~user] > mkdir wave_project
[~user] > cd wave_project
[~user/wave_project] > peekingduck init
Welcome to PeekingDuck!
2022-02-11 18:17:31 peekingduck.cli INFO: Creating custom nodes folder in ~user/wave_project/src/custom_nodes
[~user/wave_project] > peekingduck create-node
Creating new custom node...
Enter node directory relative to ~user/wave_project [src/custom_nodes]:
Select node type (input, augment, model, draw, dabble, output): dabble
Enter node name [my_custom_node]: wave

Node directory: ~user/wave_project/src/custom_nodes
Node type: dabble
Node name: wave

Creating the following files:
    Config file: ~user/wave_project/src/custom_nodes/configs/dabble/wave.yml
    Script file: ~user/wave_project/src/custom_nodes/dabble/wave.py
Proceed? [Y/n]:
Created node!
```

Also, copy wave.mp4 into the above folder. You should end up with the following folder structure:

```
wave_project/
├── pipeline_config.yml
├── src/
```

```
│   └── custom_nodes/
│       ├── configs/
│       │   └── dabble/
│       │       └── wave.yml
│       └── dabble/
│           └── wave.py
└── wave.mp4
```

To implement this tutorial, the **three files** `wave.yml`, `wave.py` and `pipeline_config.yml` are to be edited as follows:

1. **src/custom_nodes/configs/dabble/wave.yml**:

```yaml
# Dabble node has both input and output
input: ["img", "bboxes", "bbox_scores", "keypoints", "keypoint_scores"]
output: ["none"]

# No optional configs
```

   We will implement this tutorial using a custom *dabble* node, which will take the inputs *img*, *bboxes*, *bbox_scores*, *keypoints*, and *keypoint_scores* from the pipeline. The node has no output.

2. **src/custom_nodes/dabble/wave.py**:

   The `dabble.wave` code structure is similar to the `draw.score` code structure in the other custom node tutorial.

   Show/Hide Code for wave.py

```python
"""
Custom node to show keypoints and count the number of times the person's hand is waved
"""

from typing import Any, Dict, List, Tuple
import cv2
from peekingduck.pipeline.nodes.abstract_node import AbstractNode

# setup global constants
FONT = cv2.FONT_HERSHEY_SIMPLEX
WHITE = (255, 255, 255)       # opencv loads file in BGR format
YELLOW = (0, 255, 255)
THRESHOLD = 0.6               # ignore keypoints below this threshold
KP_RIGHT_SHOULDER = 6         # PoseNet's skeletal keypoints
KP_RIGHT_WRIST = 10


def map_bbox_to_image_coords(
   bbox: List[float], image_size: Tuple[int, int]
) -> List[int]:
   """First helper function to convert relative bounding box coordinates to
   absolute image coordinates.
   Bounding box coords ranges from 0 to 1
   where (0, 0) = image top-left, (1, 1) = image bottom-right.

   Args:
      bbox (List[float]): List of 4 floats x1, y1, x2, y2
      image_size (Tuple[int, int]): Width, Height of image
```

(continues on next page)

(continued from previous page)

```

    Returns:
        List[int]: x1, y1, x2, y2 in integer image coords
    """
    width, height = image_size[0], image_size[1]
    x1, y1, x2, y2 = bbox
    x1 *= width
    x2 *= width
    y1 *= height
    y2 *= height
    return int(x1), int(y1), int(x2), int(y2)


def map_keypoint_to_image_coords(
    keypoint: List[float], image_size: Tuple[int, int]
) -> List[int]:
    """Second helper function to convert relative keypoint coordinates to
    absolute image coordinates.
    Keypoint coords ranges from 0 to 1
    where (0, 0) = image top-left, (1, 1) = image bottom-right.

    Args:
        bbox (List[float]): List of 2 floats x, y (relative)
        image_size (Tuple[int, int]): Width, Height of image

    Returns:
        List[int]: x, y in integer image coords
    """
    width, height = image_size[0], image_size[1]
    x, y = keypoint
    x *= width
    y *= height
    return int(x), int(y)


def draw_text(img, x, y, text_str: str, color_code):
    """Helper function to call opencv's drawing function,
    to improve code readability in node's run() method.
    """
    cv2.putText(
        img=img,
        text=text_str,
        org=(x, y),
        fontFace=cv2.FONT_HERSHEY_SIMPLEX,
        fontScale=0.4,
        color=color_code,
        thickness=2,
    )


class Node(AbstractNode):
    """Custom node to display keypoints and count number of hand waves
```

(continues on next page)

(continued from previous page)

```
    Args:
        config (:obj:`Dict[str, Any]` | :obj:`None`): Node configuration.
    """

    def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
        super().__init__(config, node_path=__name__, **kwargs)
        # setup object working variables
        self.right_wrist = None
        self.direction = None
        self.num_direction_changes = 0
        self.num_waves = 0

    def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
        """This node draws keypoints and count hand waves.

        Args:
              inputs (dict): Dictionary with keys
                 "img", "bboxes", "bbox_scores", "keypoints", "keypoint_scores".

        Returns:
              outputs (dict): Empty dictionary.
        """

        # get required inputs from pipeline
        img = inputs["img"]
        bboxes = inputs["bboxes"]
        bbox_scores = inputs["bbox_scores"]
        keypoints = inputs["keypoints"]
        keypoint_scores = inputs["keypoint_scores"]

        img_size = (img.shape[1], img.shape[0])  # image width, height

        # get bounding box confidence score and draw it at the
        # left-bottom (x1, y2) corner of the bounding box (offset by 30 pixels)
        the_bbox = bboxes[0]             # image only has one person
        the_bbox_score = bbox_scores[0]  # only one set of scores

        x1, y1, x2, y2 = map_bbox_to_image_coords(the_bbox, img_size)
        score_str = f"BBox {the_bbox_score:0.2f}"
        cv2.putText(
            img=img,
            text=score_str,
            org=(x1, y2 - 30),           # offset by 30 pixels
            fontFace=cv2.FONT_HERSHEY_SIMPLEX,
            fontScale=1.0,
            color=WHITE,
            thickness=3,
        )

        # hand wave detection using a simple heuristic of tracking the
        # right wrist movement
```

(continues on next page)

(continued from previous page)

```
    the_keypoints = keypoints[0]              # image only has one person
    the_keypoint_scores = keypoint_scores[0]  # only one set of scores
    right_wrist = None
    right_shoulder = None

    for i, keypoints in enumerate(the_keypoints):
       keypoint_score = the_keypoint_scores[i]

       if keypoint_score >= THRESHOLD:
          x, y = map_keypoint_to_image_coords(keypoints.tolist(), img_size)
          x_y_str = f"({x}, {y})"

          if i == KP_RIGHT_SHOULDER:
             right_shoulder = keypoints
             the_color = YELLOW
          elif i == KP_RIGHT_WRIST:
             right_wrist = keypoints
             the_color = YELLOW
          else:                    # generic keypoint
             the_color = WHITE

          draw_text(img, x, y, x_y_str, the_color)

    if right_wrist is not None and right_shoulder is not None:
       # only count number of hand waves after we have gotten the
       # skeletal poses for the right wrist and right shoulder
       if self.right_wrist is None:
          self.right_wrist = right_wrist            # first wrist data point
       else:
          # wait for wrist to be above shoulder to count hand wave
          if right_wrist[1] > right_shoulder[1]:
             pass
          else:
             if right_wrist[0] < self.right_wrist[0]:
                direction = "left"
             else:
                direction = "right"

             if self.direction is None:
                self.direction = direction          # first direction data point
             else:
                # check if hand changes direction
                if direction != self.direction:
                   self.num_direction_changes += 1
                # every two hand direction changes == one wave
                if self.num_direction_changes >= 2:
                   self.num_waves += 1
                   self.num_direction_changes = 0   # reset direction count

             self.right_wrist = right_wrist         # save last position
             self.direction = direction

```

(continues on next page)

(continued from previous page)

```
            wave_str = f"#waves = {self.num_waves}"
            draw_text(img, 20, 30, wave_str, YELLOW)

        return {}
```

This (long) piece of code implements our custom *dabble* node. It defines three helper functions to convert relative to absolute coordinates and to draw text on-screen. The number of hand waves is displayed at the top-left corner of the screen.

A simple heuristic is used to count the number of times the person waves his hand. It tracks the direction in which the right wrist is moving and notes when the wrist changes direction. Upon encountering two direction changes, e.g., left -> right -> left, one wave is counted.

The heuristic also waits until the right wrist has been lifted above the right shoulder before it starts tracking hand direction and counting waves.

3. **pipeline_config.yml**:

```
nodes:
- input.visual:
    source: wave.mp4
- model.yolo
- model.posenet
- dabble.fps
- custom_nodes.dabble.wave
- draw.poses
- draw.legend:
    show: ["fps"]
- output.screen
```

We modify `pipeline_config.yml` to run both the object detection and pose estimation models to obtain the required inputs for our custom *dabble* node.

Execute peekingduck run to see your custom node in action.

Note: Royalty free video of man waving from: https://www.youtube.com/watch?v=IKj_z2hgYUM

3.3.3 Recipe 3: Debugging

When working with PeekingDuck's pipeline, you may sometimes wonder what is available in the *data pool*, or whether a particular data object has been correctly computed. This tutorial will show you how to use a custom node to help with troubleshooting and debugging PeekingDuck's pipeline.

Continuing from the above tutorial, create a new `dabble.debug` custom node:

Terminal Session

[~user/wave_project] > peekingduck create-node

Creating new custom node…

Enter node directory relative to ~user/wave_project [src/custom_nodes]:

Select node type (input, augment, model, draw, dabble, output): dabble

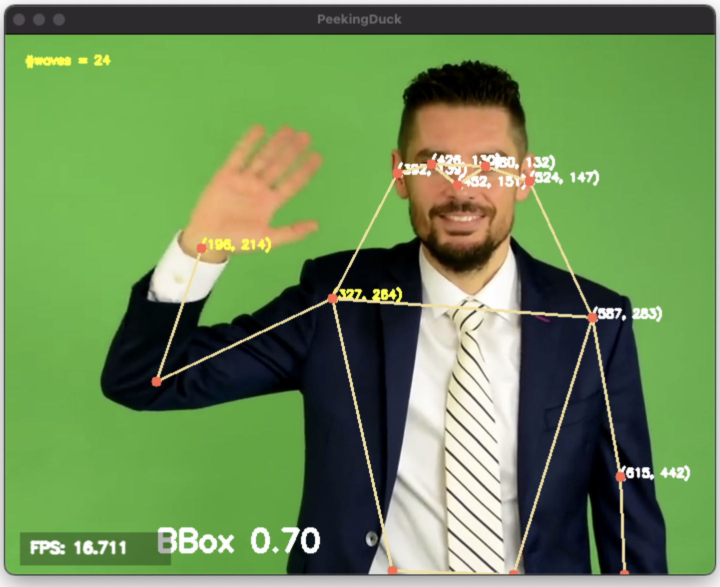


Fig. 7: Custom Node Counting Hand Waves

Enter node name [my_custom_node]: debug

Node directory: ~user/wave_project/src/custom_nodes
Node type: dabble
Node name: debug

Creating the following files:
    Config file: ~user/wave_project/src/custom_nodes/configs/dabble/debug.yml
    Script file: ~user/wave_project/src/custom_nodes/dabble/debug.py
Proceed? [Y/n]:
Created node!

The updated folder structure is:

```
wave_project/
├── pipeline_config.yml
├── src
│   └── custom_nodes
│       ├── configs
│       │   └── dabble
│       │       ├── debug.yml
│       │       └── wave.yml
│       └── dabble
│           ├── debug.py
│           └── wave.py
└── wave.mp4
```

Make the following **three** changes:

1. Define `debug.yml` to receive "*all*" inputs from the pipeline, as follows:

```yaml
# Mandatory configs
input: ["all"]
output: ["none"]

# No optional configs
```

2. Update `debug.py` as shown below:

   Show/Hide Code for debug.py

```python
"""
A custom node for debugging
"""

from typing import Any, Dict

from peekingduck.pipeline.nodes.abstract_node import AbstractNode


class Node(AbstractNode):
    """This is a simple example of creating a custom node to help with debugging.

    Args:
        config (:obj:`Dict[str, Any]` | :obj:`None`): Node configuration.
    """

    def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
        super().__init__(config, node_path=__name__, **kwargs)

    def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
        """A simple debugging custom node

        Args:
              inputs (dict): "all", to view everything in data pool

        Returns:
              outputs (dict): "none"
        """

        self.logger.info("-- debug --")
        # show what is available in PeekingDuck's data pool
        self.logger.info(f"input.keys={list(inputs.keys())}")
        # debug specific data: bboxes
        bboxes = inputs["bboxes"]
        bbox_labels = inputs["bbox_labels"]
        bbox_scores = inputs["bbox_scores"]
        self.logger.info(f"num bboxes={len(bboxes)}")
        for i, bbox in enumerate(bboxes):
              label, score = bbox_labels[i], bbox_scores[i]
              self.logger.info(f"bbox {i}:")
              self.logger.info(f"  label={label}, score={score:0.2f}")
              self.logger.info(f"  coords={bbox}")
```

(continues on next page)

(continued from previous page)

```python

        return {}  # no outputs
```

The custom node code shows how to see what is available in PeekingDuck's pipeline data pool by printing the input dictionary keys. It also demonstrates how to debug a specific data object, such as *bboxes*, by printing relevant information for each item within the data.

3. Update pipeline_config.yml:

```yaml
nodes:
- input.visual:
    source: wave.mp4
- model.yolo
- model.posenet
- dabble.fps
- custom_nodes.dabble.wave
- custom_nodes.dabble.debug
- draw.poses
- draw.legend:
    show: ["fps"]
- output.screen
```

Now, do a peekingduck run and you should see a sample debug output like the one below:

Terminal Session

[~user/wave_project] > peekingduck run

2022-03-02 18:42:51 peekingduck.declarative_loader INFO: Successfully loaded pipeline_config file.

2022-03-02 18:42:51 peekingduck.declarative_loader INFO: Initializing input.visual node...

2022-03-02 18:42:51 peekingduck.declarative_loader INFO: Config for node input.visual is updated to: 'source': wave.mp4

2022-03-02 18:42:51 peekingduck.pipeline.nodes.input.visual INFO: Video/Image size: 710 by 540

2022-03-02 18:42:51 peekingduck.pipeline.nodes.input.visual INFO: Filepath used: wave.mp4

2022-03-02 18:42:51 peekingduck.declarative_loader INFO: Initializing model.yolo node...

[... many lines of output deleted here ...]

2022-03-02 18:42:53 peekingduck.declarative_loader INFO: Initializing custom_nodes.dabble.debug node...

2022-03-02 18:42:53 peekingduck.declarative_loader INFO: Initializing draw.poses node...

2022-03-02 18:42:53 peekingduck.declarative_loader INFO: Initializing draw.legend node...

2022-03-02 18:42:53 peekingduck.declarative_loader INFO: Initializing output.screen node...

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: – debug –

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: input.keys=['img', 'pipeline_end', 'filename', 'saved_video_fps', 'bboxes', 'bbox_labels', 'bbox_scores', 'keypoints', 'keypoint_scores', 'keypoint_conns', 'hand_direction', 'num_waves', 'fps']

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: num bboxes=1

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: bbox 0:

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: label=Person, score=0.91

2022-03-02 18:42:55 custom_nodes.dabble.debug INFO: coords=[0.40047657 0.21553655 0.85199741 1.02150181]

3.3.4 Other Recipes to Create Custom Nodes

This section describes two faster ways to create custom nodes for users who are familiar with PeekingDuck.

CLI Recipe

You can skip the step-by-step prompts from peekingduck create-node by specifying all the options on the command line, for instance:

Terminal Session

[~user/wave_project] > peekingduck create-node --node_subdir src/custom_nodes --node_type dabble --node_name wave

The above is the equivalent of the tutorial *Recipe 1: Object Detection Score custom node creation*. For more information, see peekingduck create-node --help.

Pipeline Recipe

PeekingDuck can also create custom nodes by parsing your pipeline configuration file. Starting with the basic folder structure from peekingduck init:

```
wave_project/
├── pipeline_config.yml
├── src
│   └── custom_nodes
│       └── configs
└── wave.mp4
```

and the following modified `pipeline_config.yml` file:

```yaml
nodes:
- input.visual:
    source: wave.mp4
- model.yolo
- model.posenet
- dabble.fps
- custom_nodes.dabble.wave
- custom_nodes.dabble.debug
- draw.poses
- draw.legend:
    show: ["fps"]
- output.screen
```

You can tell PeekingDuck to parse your pipeline file with peekingduck create-node --config_path pipeline_config.yml:

Terminal Session

```
[~user/wave_project] > peekingduck create-node --config_path pipeline_config.yml
2022-03-14 11:21:21 peekingduck.cli INFO: Creating custom nodes declared in
~user/wave_project/pipeline_config.yml.
2022-03-14 11:21:21 peekingduck.declarative_loader INFO: Successfully loaded pipeline file.
2022-03-14 11:21:21 peekingduck.cli INFO: Creating files for custom_nodes.dabble.wave:
    Config file: ~user/wave_project/src/custom_nodes/configs/dabble/wave.yml
    Script file: ~user/wave_project/src/custom_nodes/dabble/wave.py
2022-03-14 11:21:21 peekingduck.cli INFO: Creating files for custom_nodes.dabble.debug:
    Config file: ~user/wave_project/src/custom_nodes/configs/dabble/debug.yml
    Script file: ~user/wave_project/src/custom_nodes/dabble/debug.py
```

PeekingDuck will read `pipeline_config.yml` and create the two specified custom nodes `custom_nodes.dabble.wave` and `custom_nodes.dabble.debug`. Your folder structure will now look like this:

```
wave_project/
├── pipeline_config.yml
├── src
│   └── custom_nodes
│       ├── configs
│       │   └── dabble
│       │       ├── debug.yml
│       │       └── wave.yml
│       └── dabble
│           ├── debug.py
│           └── wave.py
└── wave.mp4
```

From here, you can proceed to edit the custom node configs and source files.

3.4 Peaking Duck

PeekingDuck includes some "power" nodes that are capable of processing the contents or outputs of the other nodes and to accumulate information over time. An example is the *dabble.statistics* node which can accumulate statistical information, such as calculating the cumulative average and maximum of particular objects (like people or cars). This tutorial presents advanced recipes to showcase the power features of PeekingDuck, such as using *dabble.statistics* for object counting and tracking.

3.4.1 Interfacing with SQL

This tutorial demonstrates how to save data to an SQLite database. We will extend the tutorial for *counting hand waves* with a new custom *output* node that writes information into a local SQLite database.

Note: The above tutorial assumes `sqlite3` has been installed in your system. If your system does not have `sqlite3`, please see the SQLite Home Page for installation instructions.

First, create a new custom `output.sqlite` node in the `custom_project` folder:

Terminal Session

```
[~user/wave_project] > peekingduck create-node
Creating new custom node...
Enter node directory relative to ~user/wave_project [src/custom_nodes]:
Select node type (input, augment, model, draw, dabble, output): output
Enter node name [my_custom_node]: sqlite

Node directory: ~user/wave_project/src/custom_nodes
Node type: output
Node name: sqlite

Creating the following files:
    Config file: ~user/wave_project/src/custom_nodes/configs/output/sqlite.yml
    Script file: ~user/wave_project/src/custom_nodes/output/sqlite.py
Proceed? [Y/n]:
Created node!
```

The updated folder structure would be:

```
wave_project/
├── pipeline_config.yml
├── src/
│   └── custom_nodes/
│       ├── configs/
│       │   ├── dabble/
│       │   │   └── wave.yml
│       │   └── output/
│       │       └── sqlite.yml
│       ├── dabble/
│       │   └── wave.py
│       └── output/
│           └── sqlite.py
└── wave.mp4
```

Edit the following **five files** as described below:

1. **src/custom_nodes/configs/output/sqlite.yml**:

```yaml
# Mandatory configs
input: ["hand_direction", "num_waves"]
output: ["none"]

# No optional configs
```

   The new `output.sqlite` custom node will take in the hand direction and the current number of hand waves to save to the external database.

2. **src/custom_nodes/output/sqlite.py**:

   Show/Hide Code for sqlite.py

```python
"""
Custom node to save data to external database.
```

(continues on next page)

(continued from previous page)

```
"""

from typing import Any, Dict
from datetime import datetime
from peekingduck.pipeline.nodes.abstract_node import AbstractNode
import sqlite3

DB_FILE = "wave.db"           # name of database file


class Node(AbstractNode):
   """Custom node to save hand direction and current wave count to database.

   Args:
      config (:obj:`Dict[str, Any]` | :obj:`None`): Node configuration.
   """

   def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
      super().__init__(config, node_path=__name__, **kwargs)

      self.conn = None
      try:
         # try to establish connection to database,
         # will create DB_FILE if it does not exist
         self.conn = sqlite3.connect(DB_FILE)
         self.logger.info(f"Connected to {DB_FILE}")
         sql = """ CREATE TABLE IF NOT EXISTS wavetable (
                     datetime text,
                     hand_direction text,
                     wave_count integer
                  ); """
         cur = self.conn.cursor()
         cur.execute(sql)
      except sqlite3.Error as e:
         self.logger.info(f"SQL Error: {e}")

   def update_db(self, hand_direction: str, num_waves: int) -> None:
      """Helper function to save current time stamp, hand direction and
      wave count into DB wavetable.
      """
      now = datetime.now()
      dt_str = f"{now:%Y-%m-%d %H:%M:%S}"
      sql = """ INSERT INTO wavetable(datetime,hand_direction,wave_count)
               values (?,?,?) """
      cur = self.conn.cursor()
      cur.execute(sql, (dt_str, hand_direction, num_waves))
      self.conn.commit()

   def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
      """Node to output hand wave data into sqlite database.

      Args:
```

(continues on next page)

(continued from previous page)

```
            inputs (dict): Dictionary with keys "hand_direction", "num_waves"

        Returns:
            outputs (dict): Empty dictionary
        """

        hand_direction = inputs["hand_direction"]
        num_waves = inputs["num_waves"]
        self.update_db(hand_direction, num_waves)

        return {}
```

This tutorial uses the `sqlite3` package to interface with the database.

On first run, the node initializer will create the `wave.db` database file. It will establish a connection to the database and create a table called `wavetable` if it does not exist. This table is used to store the hand direction and wave count data.

A helper function `update_db` is called to update the database. It saves the current date time stamp, hand direction and wave count into the `wavetable`.

The node's `run` method retrieves the required inputs from the pipeline's data pool and calls `self.update_db` to save the data.

3. **src/custom_nodes/configs/dabble/wave.yml**:

```
# Dabble node has both input and output
input: ["img", "bboxes", "bbox_scores", "keypoints", "keypoint_scores"]
output: ["hand_direction", "num_waves"]

# No optional configs
```

To support the `output.sqlite` custom node's input requirements, we need to modify the `dabble.wave` custom node to return the current hand direction `hand_direction` and the current wave count `num_waves`.

4. **src/custom_nodes/dabble/wave.py**:

```
    ... same as previous ...
    return {
        "hand_direction": self.direction if self.direction is not None else "None",
        "num_waves": self.num_waves,
    }
```

This file is the same as the `wave.py` in the *counting hand waves* tutorial, except for the changes in the last few lines as shown above. These changes outputs the `hand_direction` and `num_waves` to the pipeline's data pool for subsequent consumption by the `output.sqlite` custom node.

5. **pipeline_config.yml**:

```
11 ... same as previous ...
12 - custom_nodes.output.sqlite
```

Likewise, the pipeline is the same as in the previous tutorial, except for line 12 that has been added to call the new custom node.

Run this project with peekingduck run and when completed, a new `wave.db` sqlite database file would be created in the current folder. Examine the created database as follows:

Terminal Session

```
[~user/wave_project] > sqlite3
SQLite version 3.37.0 2021-11-27 14:13:22
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open wave.db
sqlite> .schema wavetable
CREATE TABLE wavetable (
                datetime text,
                hand_direction text,
                wave_count integer
        );
sqlite> select * from wavetable where wave_count > 0 limit 5;
2022-02-15 19:26:16|left|1
2022-02-15 19:26:16|right|1
2022-02-15 19:26:16|left|2
2022-02-15 19:26:16|right|2
2022-02-15 19:26:16|right|2
sqlite> select * from wavetable order by datetime desc limit 5;
2022-02-15 19:26:44|right|72
2022-02-15 19:26:44|right|72
2022-02-15 19:26:44|right|72
2022-02-15 19:26:44|right|72
2022-02-15 19:26:43|right|70
```

Press CTRL-D to exit from `sqlite3`.

3.4.2 Counting Cars

This tutorial demonstrates using the `dabble.statistics` node to count the number of cars traveling across a highway over time and the `draw.legend` node to display the relevant statistics.

Create a new PeekingDuck project, download the highway cars video and save it into the project folder.

Terminal Session

```
[~user] > mkdir car_project
[~user] > cd car_project
[~user/car_project] > peekingduck init
```

The car_project folder structure:

```
car_project/
├── highway_cars.mp4
├── pipeline_config.yml
└── src
    └── custom_nodes
        └── configs
```

Edit pipeline_config.yml as follows:

```yaml
nodes:
- input.visual:
    source: highway_cars.mp4
- model.yolo:
    detect: ["car"]
    score_threshold: 0.3
- dabble.bbox_count
- dabble.fps
- dabble.statistics:
    identity: count
- draw.bbox
- draw.legend:
    show: ["fps", "count", "cum_max", "cum_min"]
- output.screen
```

Run it with peekingduck run and you should see a video of cars travelling across a highway with a legend box on the bottom left showing the realtime count of the number of cars on-screen, the cumulative maximum and minimum number of cars detected since the video started. The sample screenshot below shows:

- the count that there are currently 3 cars on-screen
- the cumulative maximum number of cars "seen" previously was 5
- the cumulative minimum number of cars was 1

Fig. 8: Counting Cars on a Highway

Note: Royalty free video of cars on highway from: https://www.youtube.com/watch?v=8yP1gjg4b2w

3.4.3 Object Tracking

Object tracking is the application of CV models to automatically detect objects in a video and to assign a unique identity to each of them. These objects can be either living (e.g. person) or non-living (e.g. car). As they move around in the video, these objects are identified based on their assigned identities and tracked according to their movements.

This tutorial demonstrates using *dabble.statistics* with a custom node to track the number of people walking down a path.

Create a new PeekingDuck project, download the people walking video and save it into the project folder.

Terminal Session

```
[~user] > mkdir people_walking
[~user] > cd people_walking
[~user/people_walking] > peekingduck init
```

Create the following `pipeline_config.yml`:

```yaml
nodes:
- input.visual:
    source: people_walking.mp4
- model.yolo:
    detect: ["person"]
- dabble.tracking
- dabble.statistics:
    maximum: obj_attrs["ids"]
- dabble.fps
- draw.bbox
- draw.tag:
    show: ["ids"]
- draw.legend:
    show: ["fps", "cum_max", "cum_min", "cum_avg"]
- output.screen
```

The above pipeline uses the YOLO model to detect people in the video and uses the *dabble.tracking* node to track the people as they walk. Each person is assigned a tracking ID and *dabble.tracking* returns a list of tracking IDs. *dabble.statistics* is used to process these tracking IDs: since each person is assigned a monotonically increasing integer ID, the maximum ID within the list tells us the number of persons tracked so far. *draw.tag* shows the ID above the tracked person. *draw.legend* is used to display the various statistics: the FPS, and the cumulative maximum, minimum and average relating to the number of persons tracked.

Do a peekingduck run and you will see the following display:

Note: Royalty free video of people walking from: https://www.youtube.com/watch?v=du74nvmRUzo

Fig. 9: People Walking

Tracking People within a Zone

Suppose we are only interested in people walking down the center of the video (imagine a carpet running down the middle). We can create a custom node to tell PeekingDuck to focus on the middle zone, by filtering away the detected bounding boxes outside the zone.

Start by creating a custom node `dabble.filter_bbox`:

Terminal Session

```
[~user/people_walking] > peekingduck create-node
Creating new custom node...
Enter node directory relative to ~user/people_walking [src/custom_nodes]:
Select node type (input, augment, model, draw, dabble, output): dabble
Enter node name [my_custom_node]: filter_bbox

Node directory: ~user/people_walking/src/custom_nodes
Node type: dabble
Node name: filter_bbox

Creating the following files:
    Config file: ~user/people_walking/src/custom_nodes/configs/dabble/filter_bbox.yml
    Script file: ~user/people_walking/src/custom_nodes/dabble/filter_bbox.py
Proceed? [Y/n]:
Created node!
```

The folder structure looks like this:

```
people_walking/
├── people_walking.mp4
├── pipeline_config.yml
└── src
    └── custom_nodes
        ├── configs
        │   └── dabble
        │       └── filter_bbox.yml
        └── dabble
            └── filter_bbox.py
```

Change `pipeline_config.yml` to the following:

```yaml
nodes:
- input.visual:
    source: people_walking.mp4
- model.yolo:
    detect: ["person"]
- dabble.bbox_to_btm_midpoint
- dabble.zone_count:
    resolution: [720, 480]
    zones: [
      [[0.35,0], [0.65,0], [0.65,1], [0.35,1]],
    ]
- custom_nodes.dabble.filter_bbox:
    zones: [
      [[0.35,0], [0.65,0], [0.65,1], [0.35,1]],
    ]
- dabble.tracking
- dabble.statistics:
    maximum: obj_attrs["ids"]
- dabble.fps
- draw.bbox
- draw.zones
- draw.tag:
    show: ["ids"]
- draw.legend:
    show: ["fps", "cum_max", "cum_min", "cum_avg", "zone_count"]
- output.screen
```

We make use of *`dabble.zone_count`* and *`dabble.bbox_to_btm_midpoint`* nodes to create a zone in the middle. The zone is defined by a rectangle with the four corners (0.35, 0.0) - (0.65, 0.0) - (0.65, 1.0) - (0.35, 1.0). (For more info, see *Zone Counting*) This zone is also defined in our custom node `dabble.filter_bbox` for bounding box filtering. What `dabble.filter_bbox` will do is to take the list of bboxes as input and output a list of bboxes within the zone, dropping all bboxes outside it. Then, *`dabble.tracking`* is used to track the people walking and *`dabble.statistics`* is used to determine the number of people walking in the zone, by getting the maximum of the tracked IDs. *`draw.legend`* has a new item *zone_count* which displays the number of people walking in the zone currently.

The `filter_bbox.yml` and `filter_bbox.py` files are shown below:

1. **src/custom_nodes/configs/dabble/filter_bbox.yml**:

```yaml
# Mandatory configs
input: ["bboxes"]
output: ["bboxes"]
```

(continues on next page)

(continued from previous page)

```

zones: [
  [[0,0], [0,1], [1,1], [1,0]],
]
```

Note: The zones default value of `[[0,0], [0,1], [1,1], [1,0]]` will be overridden by those specified in `pipeline_config.yml` above. See *Configuration - Behind The Scenes* for more details.

2. **src/custom_nodes/dabble/filter_bbox.py**:

   Show/Hide Code for filter_bbox.py

```python
"""
Custom node to filter bboxes outside a zone
"""

from typing import Any, Dict
import numpy as np
from peekingduck.pipeline.nodes.abstract_node import AbstractNode


class Node(AbstractNode):
   """Custom node to filter bboxes outside a zone

   Args:
      config (:obj:`Dict[str, Any]` | :obj:`None`): Node configuration.
   """

   def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
      super().__init__(config, node_path=__name__, **kwargs)

   def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
      """Checks bounding box x-coordinates against the zone left and right borders.
      Retain bounding box if within, otherwise discard it.

      Args:
            inputs (dict): Dictionary with keys "bboxes"

      Returns:
            outputs (dict): Dictionary with keys "bboxes".
      """
      bboxes = inputs["bboxes"]
      zones = self.config["zones"]
      zone = zones[0]         # only work with one zone currently
      # convert zone with 4 points to a zone bbox with (x1, y1), (x2, y2)
      x1, y1 = zone[0]
      x2, y2 = zone[2]
      zone_bbox = np.asarray([x1, y1, x2, y2])

      retained_bboxes = []
      for bbox in bboxes:
```

(continues on next page)

(continued from previous page)

```
        # filter by left and right borders (ignore top and bottom)
        if bbox[0] > zone_bbox[0] and bbox[2] < zone_bbox[2]:
            retained_bboxes.append(bbox)

    return {"bboxes": np.asarray(retained_bboxes)}
```

Do a peekingduck run and you will see the following display:

Fig. 10: Count People Walking in a Zone

3.5 Calling PeekingDuck in Python

3.5.1 Using PeekingDuck's Pipeline

As an alternative to running PeekingDuck using the command-line interface (CLI), users can also import PeekingDuck as a Python module and run it in a Python script. This demo corresponds to the *Record Video File with FPS* Section of the *Duck Confit* tutorial.

In addition, we will demonstrate basic debugging techniques which users can employ when troubleshooting PeekingDuck projects.

Setting Up

Create a PeekingDuck project using:

Terminal Session

```
[~user] > mkdir pkd_project
[~user] > cd pkd_project
[~user/pkd_project] > peekingduck init
```

Then, download the cat and computer video to the `pkd_project` folder and create a Python script `demo_debug.py` in the same folder.

You should have the following directory structure at this point:

```
pkd_project/
├── cat_and_computer.mp4
├── demo_debug.py
├── pipeline_config.yml
└── src/
```

Creating a Custom Node for Debugging

Run the following to create a *dabble* node for debugging:

Terminal Session

```
[~user/pkd_project] > peekingduck create-node --node_subdir src/custom_nodes --node_type dabble --node_name debug
```

The command will create the `debug.py` and `debug.yml` files in your project directory as shown:

```
pkd_project/
├── cat_and_computer.mp4
├── demo_debug.py
├── pipeline_config.yml
└── src/
    └── custom_nodes/
        ├── configs/
        │   └── dabble/
        │       └── debug.yml
        └── dabble/
            └── debug.py
```

Change the content of `debug.yml` to:

```yaml
input: ["all"]
output: ["none"]
```

Line 1: The data type `all` allows the node to receive all outputs from the previous nodes as its input. Please see the *Glossary* for a list of available data types.

Change the content of `debug.py` to:

Show/Hide Code

```
1 from typing import Any, Dict
2
3 import numpy as np
4
5 from peekingduck.pipeline.nodes.abstract_node import AbstractNode
6
7
8 class Node(AbstractNode):
9     def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
10         super().__init__(config, node_path=__name__, **kwargs)
11         self.frame = 0
12
13     def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:  # type: ignore
14         if "cat" in inputs["bbox_labels"]:
15             print(
16                 f"{self.frame} {inputs['bbox_scores'][np.where(inputs['bbox_labels'] == 'cat')]}"
17             )
18         self.frame += 1
19         return {}
```

Lines 14 - 17: Print out the frame number and the confidence scores of bounding boxes which are detected as "cat".

Line 18: Increment the frame number each time `run()` is called.

Creating the Python Script

Copy over the following code to `demo_debug.py`:

Show/Hide Code

```
from pathlib import Path

from peekingduck.pipeline.nodes.dabble import fps
from peekingduck.pipeline.nodes.draw import bbox, legend
from peekingduck.pipeline.nodes.input import visual
from peekingduck.pipeline.nodes.model import yolo
from peekingduck.pipeline.nodes.output import media_writer, screen
from peekingduck.runner import Runner
from src.custom_nodes.dabble import debug


def main():
    debug_node = debug.Node(pkd_base_dir=Path.cwd() / "src" / "custom_nodes")

    visual_node = visual.Node(source=str(Path.cwd() / "cat_and_computer.mp4"))
    yolo_node = yolo.Node(detect=["cup", "cat", "laptop", "keyboard", "mouse"])
    bbox_node = bbox.Node(show_labels=True)
```

(continues on next page)

(continued from previous page)

```
18
19      fps_node = fps.Node()
20      legend_node = legend.Node(show=["fps"])
21      screen_node = screen.Node()
22
23      media_writer_node = media_writer.Node(output_dir=str(Path.cwd() / "results"))
24
25      runner = Runner(
26          nodes=[
27              visual_node,
28              yolo_node,
29              debug_node,
30              bbox_node,
31              fps_node,
32              legend_node,
33              screen_node,
34              media_writer_node,
35          ]
36      )
37      runner.run()
38
39
40  if __name__ == "__main__":
41      main()
```

Lines 9, 13: Import and initialize the `debug` custom node. Pass in the `path/to/project_dir/src/custom_nodes` via `pkd_base_dir` for the configuration YAML file of the custom node to be loaded properly.

Lines 15 - 23: Create the PeekingDuck nodes necessary to replicate the demo shown in the *Record Video File with FPS* tutorial. To change the node configuration, you can pass the new values to the `Node()` constructor as keyword arguments.

Lines 25 - 37: Initialize the PeekingDuck `Runner` from runner.py with the list of nodes passed in via the `nodes` argument.

Note: A PeekingDuck node can be created in Python code by passing a dictionary of config keyword - config value pairs to the `Node()` constructor.

Running the Python Script

Run the `demo_debug.py` script using:

Terminal Session

```
[~user/pkd_project] > python demo_debug.py
```

You should see the following output in your terminal:

```
1 2022-02-24 16:33:06 peekingduck.pipeline.nodes.input.visual  INFO:  Config for node input.visual is updated to: 'source': ~user/pkd_project/cat_and_computer.mp4
2 2022-02-24 16:33:06 peekingduck.pipeline.nodes.input.visual  INFO:  Video/Image size: 720 by 480
3 2022-02-24 16:33:06 peekingduck.pipeline.nodes.input.visual  INFO:  Filepath used: ~user/pkd_project/cat_and_computer.mp4
4 2022-02-24 16:33:06 peekingduck.pipeline.nodes.model.yolo  INFO:  Config for node model.yolo is updated to: 'detect': [41, 15, 63, 66, 64]
5 2022-02-24 16:33:06 peekingduck.pipeline.nodes.model.yolov4.yolo_files.detector  INFO:  Yolo model loaded with following configs:
6     Model type: v4tiny,
7     Input resolution: 416,
8     IDs being detected: [41, 15, 63, 66, 64]
9     Max Detections per class: 50,
10     Max Total Detections: 50,
11     IOU threshold: 0.5,
12     Score threshold: 0.2
13 2022-02-24 16:33:07 peekingduck.pipeline.nodes.draw.bbox  INFO:  Config for node draw.bbox is updated to: 'show_labels': True
14 2022-02-24 16:33:07 peekingduck.pipeline.nodes.dabble.fps  INFO:  Moving average of FPS will be logged every: 100 frames
15 2022-02-24 16:33:07 peekingduck.pipeline.nodes.output.media_writer  INFO:  Config for node output.media_writer is updated to: 'output_dir': ~user/pkd_project/results
16 2022-02-24 16:33:07 peekingduck.pipeline.nodes.output.media_writer  INFO:  Output directory used is: ~user/pkd_project/results
17 0 [0.90861976]
18 1 [0.9082737]
19 2 [0.90818006]
20 3 [0.8888804]
21 4 [0.8877487]
22 5 [0.9071386]
23 6 [0.870267]
24
25 [Truncated]
```

Lines 17 - 23: The debugging output showing the frame number and the confidence score of bounding boxes predicted as "cat".

3.5.2 Integrating with Your Workflow

The modular design of PeekingDuck allows users to pick and choose the nodes they want to use. Users are also able to use PeekingDuck nodes with external packages when designing their pipeline.

In this demo, we will show how users can construct a custom PeekingDuck pipeline using:

- Data loaders such as tf.keras.preprocessing.image_dataset_from_directory (available in `tensorflow>=2.3.0`),
- External packages (not implemented as PeekingDuck nodes) such as easyocr, and
- Visualization packages such as matplotlib.

The notebook corresponding to this tutorial, `calling_peekingduck_in_python.ipynb`, can be found in the notebooks folder of the PeekingDuck repository and is also available as a Colab notebook.

Show/Hide Instructions for Linux/Mac (Intel)/Windows

Note: The uninstallation step is necessary to ensure that the proper version of OpenCV is installed.

You may receive an error message about the incompatibility between `awscli` and `colorama==0.4.4`. `awscli` is conservative about pinning versions to maintain backward compatibility. The code presented in this tutorial has been tested to work and we have chosen to prioritize PeekingDuck's dependency requirements.

Show/Hide Instructions for Mac (Apple Silicon)

Note: We install the problematic packages `easyocr` and `oidv6` first and then uninstall the `pip`-related OpenCV packages which were installed as dependencies. Mac (Apple silicon) needs `conda`'s OpenCV.

There will be a warning that `easyocr` needs some version of Pillow which can be ignored.

We are using Open Images Dataset V6 as the dataset for this demo. We recommend using the third-party oidv6 PyPI package to download the images necessary for this demo.

Run the following command after installing the prerequisites:

Terminal Session

```
[~user] > mkdir pkd_project
[~user] > cd pkd_project
[~user/pkd_project] > oidv6 downloader en --dataset data/oidv6 --type_data train --classes car --limit 10 --yes
```

Copy `calling_peekingduck_in_python.ipynb` to the `pkd_project` folder and you should have the following directory structure at this point:

```
pkd_project/
├── calling_peekingduck_in_python.ipynb
└── data/
    └── oidv6/
        ├── boxes/
        ├── metadata/
        └── train/
            └── car/
```

Import the Modules

Show/Hide Code

```python
import os
from pathlib import Path

import cv2
import easyocr
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
9 from peekingduck.pipeline.nodes.draw import bbox
10 from peekingduck.pipeline.nodes.model import yolo_license_plate
11
12 %matplotlib inline
```

Lines 9 - 10: You can also do:

```
from peekingduck.pipeline.nodes.draw import bbox as pkd_bbox
from peekingduck.pipeline.nodes.model import yolo_license_plate as pkd_yolo_license_plate

bbox_node = pkd_bbox.Node()
yolo_license_plate_node = pkd_yolo_license_plate.Node()
```

to avoid potential name conflicts.

Initialize PeekingDuck Nodes

Show/Hide Code

```
1 yolo_lp_node = yolo_license_plate.Node()
2
3 bbox_node = bbox.Node(show_labels=True)
```

Lines 3: To change the node configuration, you can pass the new values to the `Node()` constructor as keyword arguments.

Refer to the *API Documentation* for the configurable settings for each node.

Create a Dataset Loader

Show/Hide Code

```
1 data_dir = Path.cwd().resolve() / "data" / "oidv6" / "train"
2 dataset = tf.keras.preprocessing.image_dataset_from_directory(
3     data_dir, batch_size=1, shuffle=False
4 )
```

Lines 2 - 4: We create the data loader using `tf.keras.preprocessing.image_dataset_from_directory()`; you can also create your own data loader class.

Create a License Plate Parser Class

Show/Hide Code

```
class LPReader:
    def __init__(self, use_gpu):
        self.reader = easyocr.Reader(["en"], gpu=use_gpu)

    def read(self, image):
        """Reads text from the image and joins multiple strings to a
        single string.
```

(continues on next page)

(continued from previous page)

```
        """
        return " ".join(self.reader.readtext(image, detail=0))

reader = LPReader(False)
```

We create the license plate parser class in a Python class using `easyocr` to demonstrate how users can integrate the PeekingDuck pipeline with external packages.

Alternatively, users can create a custom node for parsing license plates and run the pipeline through the CLI instead. Refer to the *custom nodes* tutorial for more information.

The Inference Loop

Show/Hide Code

```
def get_best_license_plate(frame, bboxes, bbox_scores, width, height):
    """Returns the image region enclosed by the bounding box with the highest
    confidence score.
    """
    best_idx = np.argmax(bbox_scores)
    best_bbox = bboxes[best_idx].astype(np.float32).reshape((-1, 2))
    best_bbox[:, 0] *= width
    best_bbox[:, 1] *= height
    best_bbox = np.round(best_bbox).astype(int)

    return frame[slice(*best_bbox[:, 1]), slice(*best_bbox[:, 0])]

num_col = 3
# For visualization, we plot 3 columns, 1) the original image, 2) image with
# bounding box, and 3) the detected license plate region with license plate
# number prediction shown as the plot title
fig, ax = plt.subplots(
    len(dataset), num_col, figsize=(num_col * 3, len(dataset) * 3)
)
for i, (element, path) in enumerate(zip(dataset, dataset.file_paths)):
    image_orig = cv2.imread(path)
    image_orig = cv2.cvtColor(image_orig, cv2.COLOR_BGR2RGB)
    height, width = image_orig.shape[:2]

    image = element[0].numpy().astype("uint8")[0].copy()

    yolo_lp_input = {"img": image}
    yolo_lp_output = yolo_lp_node.run(yolo_lp_input)

    bbox_input = {
        "img": image,
        "bboxes": yolo_lp_output["bboxes"],
        "bbox_labels": yolo_lp_output["bbox_labels"],
    }
    _ = bbox_node.run(bbox_input)

    ax[i][0].imshow(image_orig)
```

(continues on next page)

(continued from previous page)

```
38     ax[i][1].imshow(image)
39     # If there are any license plates detected, try to predict the license
40     # plate number
41     if len(yolo_lp_output["bboxes"]) > 0:
42         lp_image = get_best_license_plate(
43             image_orig, yolo_lp_output["bboxes"],
44             yolo_lp_output["bbox_scores"],
45             width,
46             height,
47         )
48         lp_pred = reader.read(lp_image)
49         ax[i][2].imshow(lp_image)
50         ax[i][2].title.set_text(f"Pred: {lp_pred}")
```

Lines 1 - 11: We define a utility function for retrieving the image region of the license plate with the highest confidence score to improve code clarity. For more information on how to convert between bounding box and image coordinates, please refer to the *Bounding Box vs Image Coordinates* tutorial.

Lines 27 - 35: By carefully constructing the input for each of the nodes, we can perform the inference loop within a custom workflow.

Lines 37 - 38: We plot the data using `matplotlib` for debugging and visualization purposes.

Lines 41 - 48: We integrate the inference loop with external packages such as the license plate parser we have created earlier using `easyocr`.

3.6 Using Your Own Models

PeekingDuck offers pre-trained `model` nodes that can be used to tackle a wide variety of problems, but you may need to train your own model on a custom dataset sometimes. This tutorial will show you how to package your model into a custom `model` node, and use it with PeekingDuck. We will be tackling a manufacturing use case here - classifying images of steel castings into "defective" or "normal" classes.

Casting is a manufacturing process in which a material such as metal in liquid form is poured into a mold and allowed to solidify. The solidified result is also called a casting. Sometimes, defective castings are produced, and quality assurance departments are responsible for preventing defective pieces from being used downstream. As inspections are usually done manually, this adds a significant amount of time and cost, and thus there is an incentive to automate this process.

The images of castings used in this tutorial are the front faces of steel pump impellers. From the comparison below, it can be seen that the defective casting has a rough, uneven edges while the normal casting has smooth edges.

3.6.1 Model Training

PeekingDuck is designed for model *inference* rather than model *training*. This optional section shows how a simple Convolutional Neural Network (CNN) model can be trained separately from the PeekingDuck framework. If you have already trained your own model, the *following section* describes how you can convert it to a custom model node, and use it within PeekingDuck for inference.

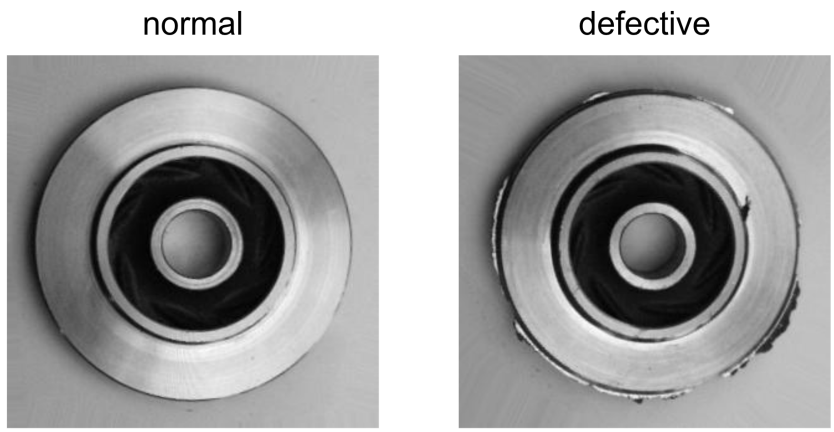


Fig. 11: Normal Casting Compared to Defective Casting

Setting Up

Install the following prerequisite for visualization.

```
> conda install matplotlib
```

Create the following project folder:

Terminal Session

```
[~user] > mkdir castings_project
[~user] > cd castings_project
```

Download the castings dataset and unzip the file to the `castings_project` folder.

Note: The castings dataset used in this example is modified from the original dataset from Kaggle.

You should have the following directory structure at this point:

```
castings_project/
└── castings_data/
        ├── inspection/
        ├── train/
        └── validation/
```

Update Training Script

Create an empty `train_classifier.py` file within the `castings_project` folder, and update it with the following code:

train_classifier.py:

Show/Hide Code for train_classifier.py

```python
"""
Script to train a classification model on images, save the model, and plot the␣
↪training results

Adapted from: https://www.tensorflow.org/tutorials/images/classification
"""

import pathlib
from typing import List, Tuple

import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers.experimental.preprocessing import Rescaling

# setup global constants
DATA_DIR = "./castings_data"
WEIGHTS_DIR = "./weights"
RESULTS = "training_results.png"
EPOCHS = 10
BATCH_SIZE = 32
IMG_HEIGHT = 180
IMG_WIDTH = 180


def prepare_data() -> Tuple[tf.data.Dataset, tf.data.Dataset, List[str]]:
    """
    Generate training and validation datasets from a folder of images.

    Returns:
        train_ds (tf.data.Dataset): Training dataset.
        val_ds (tf.data.Dataset): Validation dataset.
        class_names (List[str]): Names of all classes to be classified.
    """

    train_dir = pathlib.Path(DATA_DIR, "train")
    validation_dir = pathlib.Path(DATA_DIR, "validation")

    train_ds = tf.keras.preprocessing.image_dataset_from_directory(
        train_dir,
        image_size=(IMG_HEIGHT, IMG_WIDTH),
        batch_size=BATCH_SIZE,
    )

```

(continues on next page)

(continued from previous page)

```
    val_ds = tf.keras.preprocessing.image_dataset_from_directory(
        validation_dir,
        image_size=(IMG_HEIGHT, IMG_WIDTH),
        batch_size=BATCH_SIZE,
    )

    class_names = train_ds.class_names

    return train_ds, val_ds, class_names


def train_and_save_model(
    train_ds: tf.data.Dataset, val_ds: tf.data.Dataset, class_names: List[str]
) -> tf.keras.callbacks.History:
    """
    Train and save a classification model on the provided data.

    Args:
        train_ds (tf.data.Dataset): Training dataset.
        val_ds (tf.data.Dataset): Validation dataset.
        class_names (List[str]): Names of all classes to be classified.

    Returns:
        history (tf.keras.callbacks.History): A History object containing␣
↪recorded events from
                model training.
    """

    num_classes = len(class_names)

    model = Sequential(
        [
            Rescaling(1.0 / 255, input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
            layers.Conv2D(16, 3, padding="same", activation="relu"),
            layers.MaxPooling2D(),
            layers.Conv2D(32, 3, padding="same", activation="relu"),
            layers.MaxPooling2D(),
            layers.Conv2D(64, 3, padding="same", activation="relu"),
            layers.MaxPooling2D(),
            layers.Dropout(0.2),
            layers.Flatten(),
            layers.Dense(128, activation="relu"),
            layers.Dense(num_classes),
        ]
    )

    model.compile(
        optimizer="adam",
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=["accuracy"],
    )

```

(continues on next page)

(continued from previous page)

```
    print(model.summary())
    history = model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    model.save(WEIGHTS_DIR)

    return history


def plot_training_results(history: tf.keras.callbacks.History) -> None:
    """
    Plot training and validation accuracy and loss curves, and save the plot.

    Args:
        history (tf.keras.callbacks.History): A History object containing recorded events from
                model training.
    """
    acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
    epochs_range = range(EPOCHS)

    plt.figure(figsize=(16, 8))
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, acc, label="Training Accuracy")
    plt.plot(epochs_range, val_acc, label="Validation Accuracy")
    plt.legend(loc="lower right")
    plt.title("Training and Validation Accuracy")

    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, loss, label="Training Loss")
    plt.plot(epochs_range, val_loss, label="Validation Loss")
    plt.legend(loc="upper right")
    plt.title("Training and Validation Loss")
    plt.savefig(RESULTS)


if __name__ == "__main__":
    train_ds, val_ds, class_names = prepare_data()
    history = train_and_save_model(train_ds, val_ds, class_names)
    plot_training_results(history)
```

Training the Model

Train the model by running the following command.

Terminal Session

```
[~user/castings_project] > python train_classifier.py
```

Note: For macOS Apple Silicon, the above code only works on macOS 12.x Monterey with the latest tensorflow-macos and tensorflow-metal versions. It will crash on macOS 11.x Big Sur due to bugs in the outdated tensorflow versions.

The model will be trained for 10 epochs, and when training is completed, a new `weights` folder and `training_results.png` will be created:

```
castings_project/
├── train_classifier.py
├── training_results.png
├── castings_data/
│       ├── inspection/
│       ├── train/
│       └── validation/
└── weights/
        ├── keras_metadata.pb
        ├── saved_model.pb
        ├── assets/
        └── variables/
```

The plots from `training_results.png` shown below indicate that the model has performed well on the validation dataset, and we are ready to create a custom `model` node from it.

3.6.2 Using Your Trained Model with PeekingDuck

This section will show you how to convert your trained model into a custom PeekingDuck node, and give an example of how you can integrate this node in a PeekingDuck pipeline. It assumes that you are already familiar with the process of creating custom nodes, covered in the earlier *custom node* tutorial.

Converting to a Custom Model Node

First, let's create a new PeekingDuck project within the existing `castings_project` folder.

Terminal Session

```
[~user/castings_project] > peekingduck init
```

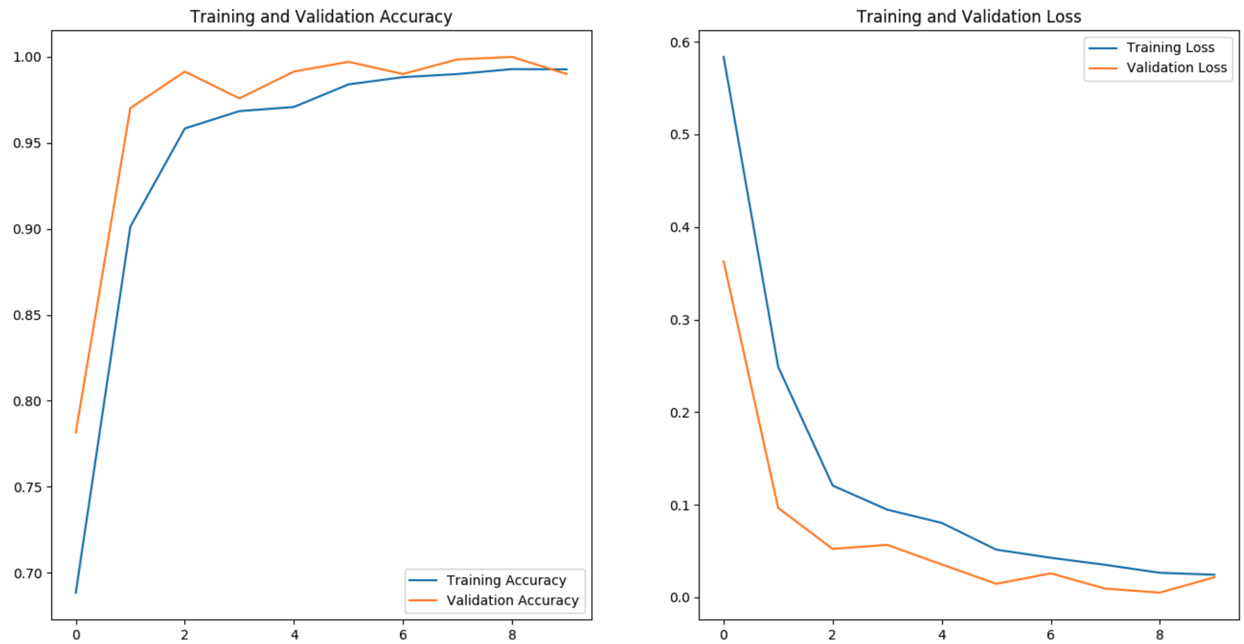


Fig. 12: Model Training Results

Next, we'll use the peekingduck create-node command to create a custom node:

Terminal Session

[~user/castings_project] > peekingduck create-node
Creating new custom node...
Enter node directory relative to ~user/castings_project [src/custom_nodes]:
Select node type (input, augment, model, draw, dabble, output): model
Enter node name [my_custom_node]: casting_classifier

Node directory: ~user/castings_project/src/custom_nodes
Node type: model
Node name: casting_classifier

Creating the following files:
    Config file: ~user/castings_project/src/custom_nodes/configs/model/casting_classifier.yml
    Script file: ~user/castings_project/src/custom_nodes/model/casting_classifier.py
Proceed? [Y/n]:
Created node!

The `castings_project` folder structure should now look like this:

```
castings_project/
```

```
├── pipeline_config.yml
├── train_classifier.py
├── training_results.png
├── castings_data/
│   ├── inspection/
│   ├── train/
│   └── validation/
├── src/
│   └── custom_nodes/
│       ├── configs/
│       │   └── model/
│       │       └── casting_classifier.yml
│       └── model/
│           └── casting_classifier.py
└── weights/
    ├── keras_metadata.pb
    ├── saved_model.pb
    ├── assets/
    └── variables/
```

`castings_project` now contains **two files** that we need to modify to implement our custom node.

1. **src/custom_nodes/configs/model/casting_classifier.yml**:

   `casting_classifier.yml` updated content:

```yaml
input: ["img"]
output: ["pred_label", "pred_score"]

weights_parent_dir: weights
class_label_map: {0: "defective", 1: "normal"}
```

2. **src/custom_nodes/model/casting_classifier.py**:

   `casting_classifier.py` updated content:

   Show/Hide Code for casting_classifier.py

```python
"""
Casting classification model.
"""

from typing import Any, Dict

import cv2
import numpy as np
import tensorflow as tf

from peekingduck.pipeline.nodes.node import AbstractNode

IMG_HEIGHT = 180
IMG_WIDTH = 180


class Node(AbstractNode):
    """Initializes and uses a CNN to predict if an image frame shows a normal
```

(continues on next page)

(continued from previous page)

```
    or defective casting.
    """

    def __init__(self, config: Dict[str, Any] = None, **kwargs: Any) -> None:
        super().__init__(config, node_path=__name__, **kwargs)
        self.model = tf.keras.models.load_model(self.weights_parent_dir)

    def run(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        """Reads the image input and returns the predicted class label and
        confidence score.

        Args:
            inputs (dict): Dictionary with key "img".

        Returns:
            outputs (dict): Dictionary with keys "pred_label" and "pred_score".
        """
        img = cv2.cvtColor(inputs["img"], cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))
        img = np.expand_dims(img, axis=0)
        predictions = self.model.predict(img)
        score = tf.nn.softmax(predictions[0])

        return {
            "pred_label": self.class_label_map[np.argmax(score)],
            "pred_score": 100.0 * np.max(score),
        }
```

The custom node takes in the built-in PeekingDuck *img* data type, makes a prediction based on the image, and produces two custom data types: `pred_label`, the predicted label ("defective" or "normal"); and `pred_score`, which is the confidence score of the prediction.

Using the Classifier in a PeekingDuck Pipeline

We'll now pair this custom node with other PeekingDuck nodes to build a complete solution. Imagine an automated inspection system like the one shown below, where the castings are placed on a conveyor belt and a camera takes a picture of each casting and sends it to the PeekingDuck pipeline for prediction. A report showing the predicted result for each casting is produced, and the quality inspector can use it for further analysis.

Edit the `pipeline_config.yml` file to use the *input.visual* node to read in the images, and the *output.csv_writer* node to produce the report. We will test our solution on the 10 casting images in `castings_data/inspection`, where each image's filename is a unique casting ID such as `28_4137.jpeg`.

pipeline_config.yml:

`pipeline_config.yml` updated content:

```
nodes:
- input.visual:
    source: castings_data/inspection
- custom_nodes.model.casting_classifier
```

(continues on next page)

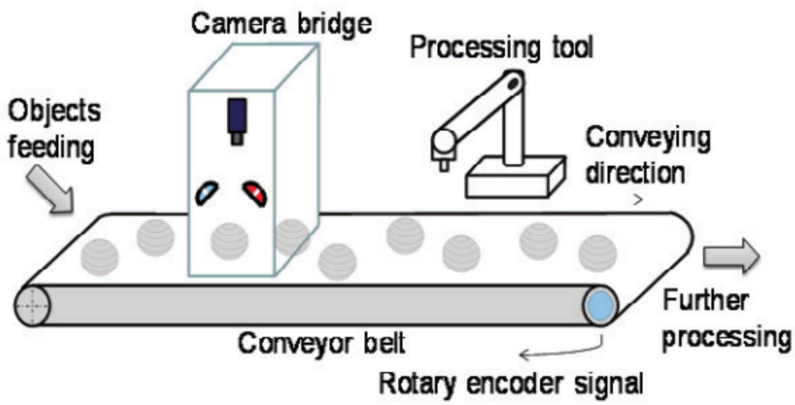


Fig. 13: Vision Based Inspection of Conveyed Objects (Source: ScienceDirect)

(continued from previous page)

```
5  - output.csv_writer:
6      stats_to_track: ["filename", "pred_label", "pred_score"]
7      file_path: casting_predictions.csv
8      logging_interval: 0
```

Line 2 *input.visual*: tells PeekingDuck to load the images from `castings_data/inspection`.
Line 4 Calls the custom model node that you have just created.
Line 5 *output.csv_writer*: produces the report for the quality inspector in a CSV file `castings_predictions_DDMMYY-hh-mm-ss.csv` (time stamp appended to `file_path`). This node receives the *filename* data type from *input.visual*, the custom data types `pred_label` and `pred_score` from the custom model node, and writes them to the columns of the CSV file.

Run the above with the command peekingduck run.

Open the created CSV file and you would see the following results. Half of the castings have been predicted as defective with high confidence scores. As the file name of each image is its unique casting ID, the quality inspector would be able to check the results with the actual castings if needed.

To visualize the predictions alongside the casting images, create an empty Python script named `visualize_results.py`, and update it with the following code:

visualize_results.py:

Show/Hide Code for visualize_results.py

```python
"""
Script to visualize the prediction results alongside the casting images
"""

import csv

import cv2
import matplotlib.pyplot as plt
```

(continues on next page)

Time	filename	pred_label	pred_score
11:50:31	28_3123.jpeg	defective	100
11:50:31	28_3416.jpeg	defective	100
11:50:31	28_4137.jpeg	defective	99.9213099
11:50:31	28_5297.jpeg	defective	100
11:50:31	28_7653.jpeg	defective	100
11:50:31	28_9918.jpeg	normal	99.6197104
11:50:31	28_9920.jpeg	normal	99.99367
11:50:31	28_9925.jpeg	normal	99.9608338
11:50:31	28_9926.jpeg	normal	97.5584447
11:50:31	28_9928.jpeg	normal	97.8244841

Fig. 14: Casting Prediction Results

(continued from previous page)

```
9
10 CSV_FILE = "casting_predictions_280422-11-50-30.csv"  # change file name accordingly
11 INSPECTION_IMGS_DIR = "castings_data/inspection/"
12 RESULTS_FILE = "inspection_results.png"
13
14 fig, axs = plt.subplots(2, 5, figsize=(50, 20))
15
16 with open(CSV_FILE) as csv_file:
17     csv_reader = csv.reader(csv_file, delimiter=",")
18     next(csv_reader, None)
19     for i, row in enumerate(csv_reader):
20         # csv columns follow this order: 'Time', 'filename', 'pred_label', 'pred_score'
21         image_path = INSPECTION_IMGS_DIR + row[1]
22         image_orig = cv2.imread(image_path)
23         image_orig = cv2.cvtColor(image_orig, cv2.COLOR_BGR2RGB)
24
25         row_idx = 0 if i < 5 else 1
26         axs[row_idx][i % 5].imshow(image_orig)
27         axs[row_idx][i % 5].set_title(row[1] + " - " + row[2], fontsize=35)
28         axs[row_idx][i % 5].axis("off")
29
30 fig.savefig(RESULTS_FILE)
```

In Line 10, replace the name of `CSV_FILE` with the name of the CSV file produced on your system, as a timestamp would have been appended to the file name.

Run the following command to visualize the results.

Terminal Session

```
[~user/castings_project] > python visualize_results.py
```

An `inspection_results.png` would be created, as shown below. The top row of castings are clearly defective, as they have rough, uneven edges, while the bottom row of castings look normal. Therefore, the prediction results are accurate for this batch of inspected castings. The quality inspector can provide feedback to the manufacturing team to further investigate the defective castings based on the casting IDs.

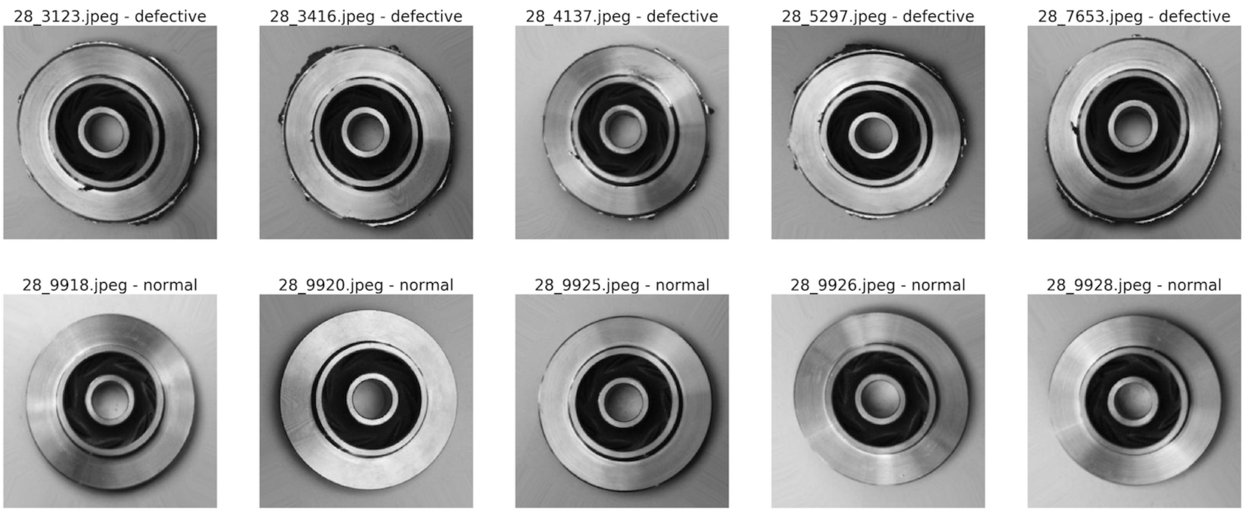


Fig. 15: Casting Prediction Visualization

This concludes the guided example on using your own custom models.

3.6.3 Custom Object Detection Models

The previous example was centered on the task of image classification. *Object detection* is another common task in Computer Vision. PeekingDuck offers several pre-trained *object detection* model nodes which can detect up to 80 different types of objects, such as persons, cars, and dogs, just to name a few. For the complete list of detectable objects, refer to the *Object Detection IDs* page. Quite often, you may need to train a custom object detection model on your own dataset, such as defects on a printed circuit board (PCB) as shown below. This section discusses some important considerations for the object detection task, supplementing the guided example above.

PeekingDuck's object detection model nodes conventionally receive the *img* data type, and produce the *bboxes*, *bbox_labels*, and *bbox_scores* data types. An example of this can be seen in the API documentation for a node such as `model.efficientdet`. We strongly recommend keeping to these data type conventions for your custom object detection node, ensuring that they adhere to the described format, e.g. *img* is in BGR format, and *bboxes* is a NumPy array of a certain shape.

This allows you to leverage on PeekingDuck's ecosystem of existing nodes. For example, by ensuring that your custom model node receives *img* in the correct format, you are able to use PeekingDuck's `input.visual` node, which can read from multiple visual sources such as a folder of images or videos, an online cloud source, or a CCTV/webcam live feed. By ensuring that your custom model node produces *bboxes* and *bbox_labels* in the correct format, you are able to use PeekingDuck's `draw.bbox` node to draw bounding boxes and associated labels around the detected objects.

By doing so, you would have saved a significant amount of development time, and can focus more on developing and finetuning your custom object detection model. This was just a simple example, and you can find out more about PeekingDuck's nodes from our *API Documentation*, and PeekingDuck's built-in data types from our *Glossary*.

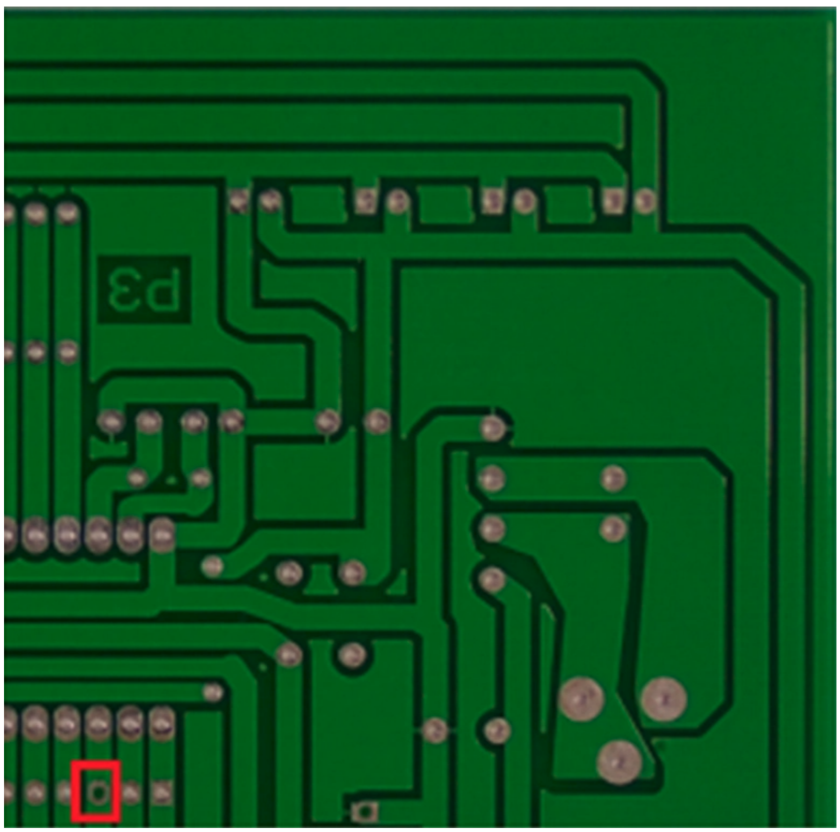


Fig. 16: Object Detection of Defects on PCB (Source: The Institution of Engineering and Technology)

CHAPTER

FOUR

PEEKINGDUCK ECOSYSTEM

This section covers the extensions to the PeekingDuck ecosystem.

4.1 PeekingDuck Viewer

The PeekingDuck Viewer offers you an interactive GUI application to manage and run PeekingDuck pipelines, and to view and analyze the output video.

4.1.1 Running the Viewer

The PeekingDuck Viewer can be activated using the CLI `--viewer` option:

Terminal Session

```
[~user] > peekingduck run --viewer
```

A screenshot of the Viewer and its GUI components is shown below:

Once the Viewer screen appears, PeekingDuck will begin executing the current pipeline. The pipeline output is displayed as a video in the center of the screen, with a progress bar below it.

If pipeline input length is deterministic (e.g. using a video file as the source), the progress bar functions like a normal progress bar moving from start to end. Upon completion, the progress bar will be replaced with a slider that you can use to navigate the output video.

If the length is non-deterministic (e.g. capturing a webcam video), then the progress bar will function in a non-deterministic manner: animating itself to indicate progress but without an end point (as PeekingDuck has no idea how long the webcam video will be). In this case, click the `Play/Stop` button to end the webcam video capture, and the progress bar will become a slider.

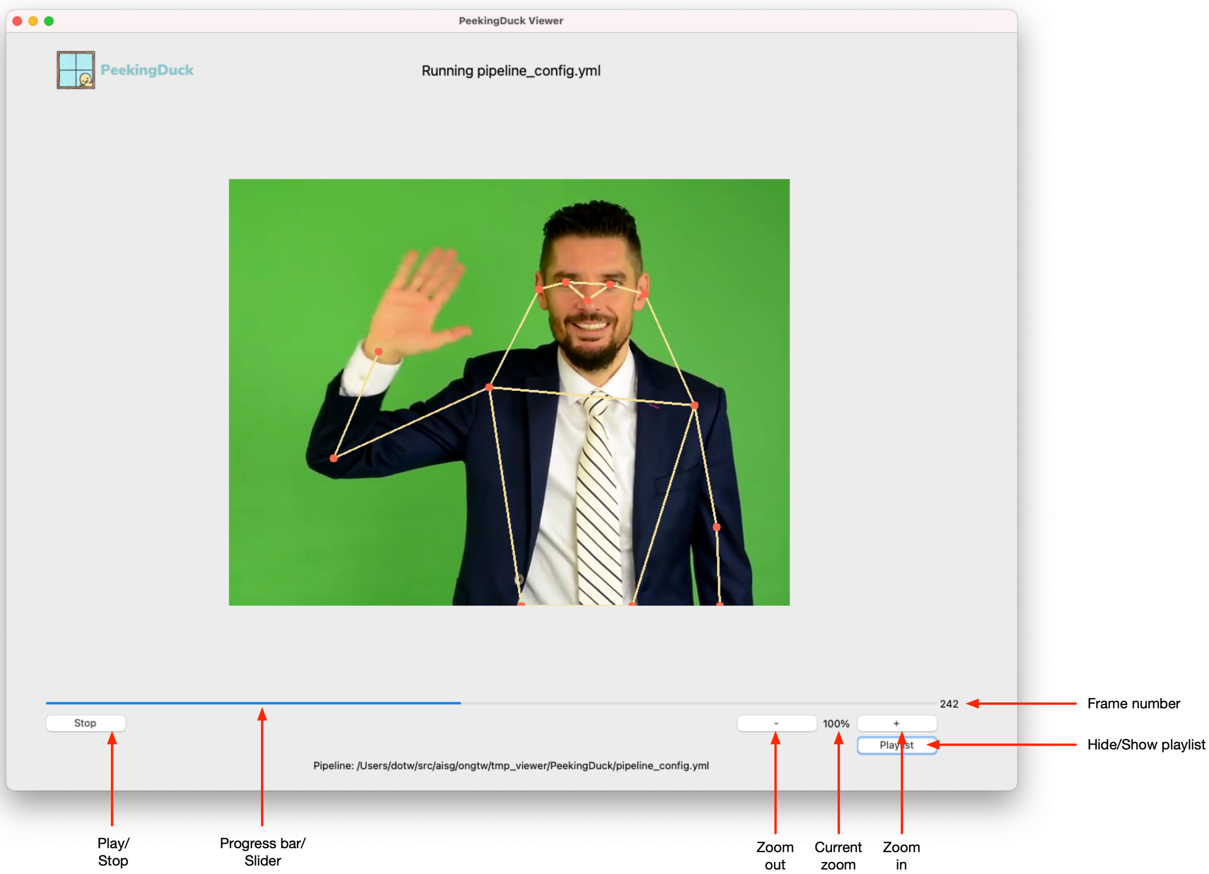


Fig. 1: The PeekingDuck Viewer screen, with explanations for the main controls.

4.1.2 Navigating the Output Video

You can examine the output video of the executed pipeline by using the `Play/Stop` button to replay the entire video.

You may also scrub through the video using the slider to go directly to the frames of interest. The current video frame number is shown to the right of the slider, serving as a position indicator. To "jump" to a particular position on the slider, click the *right* mouse button on that position. To move frame-by-frame forward/backward, click the *left* mouse button anywhere to the right/left of the current slider position.

The + (zoom in) and - (zoom out) buttons allow you to adjust the video size. You may also use keyboard shortcuts to adjust the zoom: `CTRL - -` zoom out, `CTRL - +` zoom in, `CTRL - =` reset zoom

4.1.3 Using the Pipeline Playlist

Clicking the `Playlist` button will show/hide the playlist.

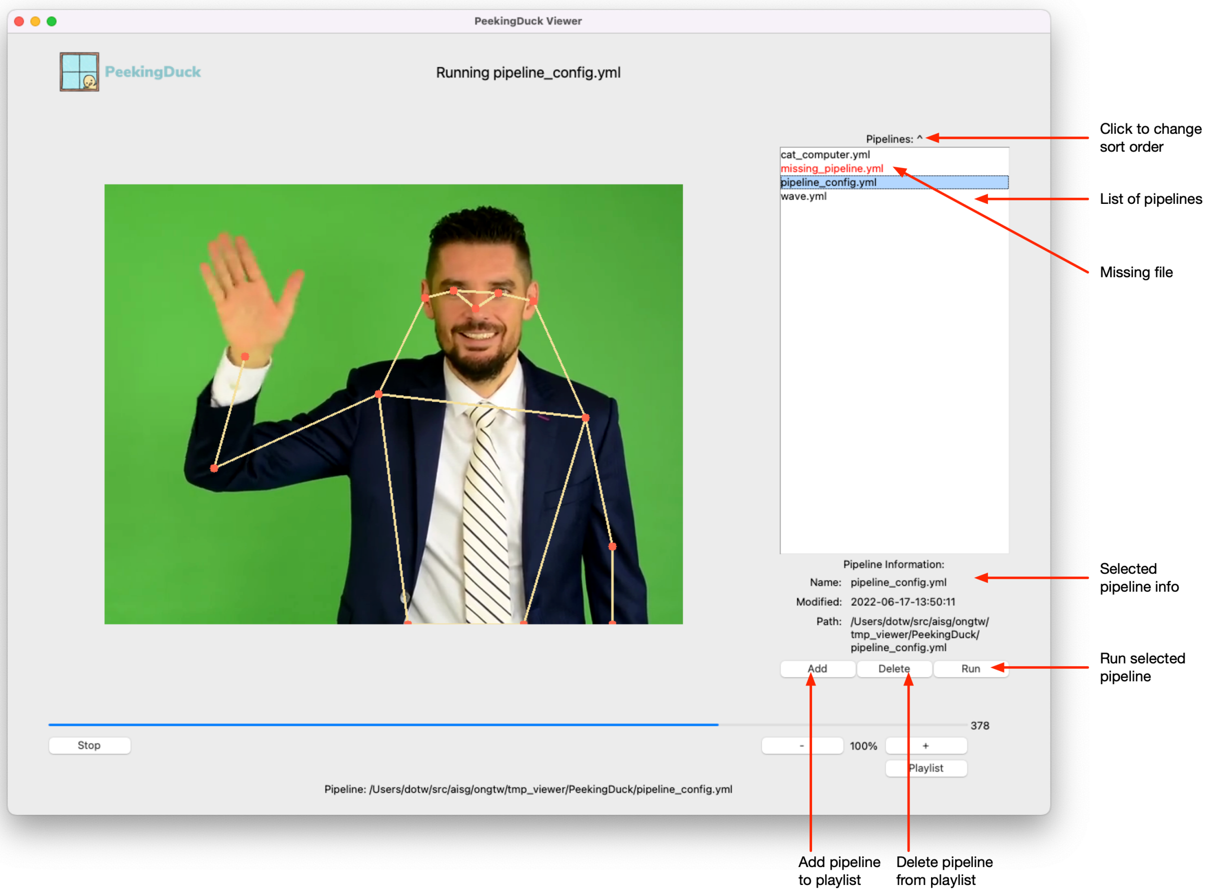


Fig. 2: PeekingDuck Viewer with playlist shown.

The above screenshot shows the playlist on the right. The playlist is a collection of pipeline files that can be run with PeekingDuck. The current pipeline is automatically added to the playlist. This playlist is specific to you and is saved across different PeekingDuck Viewer runs.

Click to select a pipeline in the playlist. The pipeline's information will be displayed in the `Pipeline Information` panel below. It shows the pipeline's name, last modified date/time, and full file path.

To run the currently selected pipeline, click the `Run` button.

The `Add` button lets you manually add a pipeline file to the playlist. It will display a File Explorer dialog. Use it to select a PeekingDuck pipeline YAML file and it will be added to your playlist.

The `Delete` button will remove the currently selected pipeline from the playlist, after you have confirmed the deletion.

If the pipeline in the playlist is red, it means the pipeline YAML file is missing. This could mean the pipeline had been added earlier, but its YAML file had since been deleted or moved to another folder. Delete the missing pipeline entry to remove it from the playlist.

The list of pipelines can be sorted in reverse order by clicking the playlist header.

Note: The playlist is saved in *~/.peekingduck/playlist.yml*, where ~ is the user's home folder.

4.1.4 Exiting the Viewer

To exit the Viewer, close the Viewer window.

CHAPTER

FIVE

MODEL RESOURCES & INFORMATION

5.1 Object Detection Models

5.1.1 List of Object Detection Models

The table below shows the object detection models available for each task category.

Category	Model	Documentation
General	EfficientDet	`model.efficientdet`
	YOLOv4	`model.yolo`
	YOLOX	`model.yolox`
Face	MTCNN	`model.mtcnn`
	YOLOv4 (Face)	`model.yolo_face`
License plate	YOLOv4 (License Plate)	`model.yolo_license_plate`

5.1.2 Benchmarks

Inference Speed

The table below shows the frames per second (FPS) of each model type.

Model	Type	Size	CPU		GPU	
			single	multiple	single	multiple
YOLO	v4tiny	416	22.42	21.71	65.24	57.50
	v4	416	2.62	2.59	30.40	28.71
EfficientDet	0	512	5.24	5.25	29.51	29.39
	1	640	2.53	2.49	23.79	24.44
	2	768	1.54	1.50	19.86	20.51
	3	896	0.78	0.75	14.69	14.84
	4	1024	0.43	0.42	11.74	11.88
MTCNN	–	–	32.42	18.53	56.35	51.45
YOLOX	yolox-tiny	416	19.43	19.29	55.36	55.38
	yolox-s	640	15.10	15.44	53.81	53.74
	yolox-m	640	8.29	8.04	42.79	43.83
	yolox-l	640	4.59	4.75	35.30	36.08

Hardware

The following hardware were used to conduct the FPS benchmarks:

- CPU: 2.8 GHz 4-Core Intel Xeon (2020, Cascade Lake) CPU and 16GB RAM
- GPU: NVIDIA A100, paired with 2.2 GHz 6-Core Intel Xeon CPU and 85GB RAM

Test Conditions

The following test conditions were followed:

- *input.visual*, the model of interest, and *dabble.fps* nodes were used to perform inference on videos
- 2 videos were used to benchmark each model, one with only 1 human (single), and the other with multiple humans (multiple)
- Both videos are about 1 minute each, recorded at ~30 FPS, which translates to about 1,800 frames to process per video
- 1280×720 (HD ready) resolution was used, as a bridge between 640×480 (VGA) of poorer quality webcams, and 1920×1080 (Full HD) of CCTVs

Model Accuracy

The table below shows the performance of our object detection models using the detection evaluation metrics from COCO. Description of these metrics can be found here.

Model	Type	Size	AP	AP IoU=.50	AP IoU=.75	AP small	AP medium	AP large	AR max=1	AR max=10	AR max=100	AR small	AR medium	AR large
YOLO	v4tiny	416	17.4	32.7	16.6	6.4	20.1	25.6	16.7	22.8	21.1	6.1	23.7	32.1
	v4	416	43.7	64.0	48.1	23.1	49.6	60.9	33.3	49.1	50.0	26.2	56.1	70.3
Efficient-Det	0	512	29.7	44.3	32.4	7.4	34.4	49.2	25.3	34.5	34.8	7.8	39.7	58.4
	1	640	35.2	50.8	38.8	14.3	40.1	53.9	28.8	40.5	40.9	15.6	46.3	62.8
	2	768	38.5	54.4	42.1	18.9	42.7	57.1	30.9	43.9	44.4	20.8	48.9	65.5
	3	896	41.1	57.0	45.2	22.2	45.1	58.7	32.6	46.7	47.3	24.8	51.5	66.9
	4	1024	43.4	59.2	47.8	24.2	47.6	60.4	33.8	49.1	49.7	27.3	53.9	68.7
YOLOX	yolox-tiny	416	32.4	50.5	33.9	13.4	35.4	49.5	28.2	43.5	45.7	20.7	51.7	65.9
	yolox-s	416	35.6	53.4	37.8	14.0	39.3	55.7	30.3	46.0	48.1	20.9	54.7	70.8
	yolox-m	416	41.6	59.7	44.4	18.8	46.9	62.8	33.9	51.6	53.7	26.9	60.9	76.8
	yolox-l	416	44.5	62.5	47.6	21.9	50.6	65.5	35.5	54.2	56.3	31.0	64.0	78.1

Dataset

The MS COCO (val 2017) dataset is used. We integrated the COCO API into the PeekingDuck pipeline for loading the annotations and evaluating the outputs from the models. All values are reported in percentages.

All images from the 80 object categories in the MS COCO (val 2017) dataset were processed.

Test Conditions

The following test conditions were followed:

- The tests were performed using pycocotools on the MS COCO dataset
- The evaluation metrics have been compared with the original repository of the respective object detection models for consistency

5.1.3 Object Detection IDs

General Object Detection

The tables below provide the associated indices for each class in object detectors.
To detect all classes, specify `detect: ["*"]` under the object detection node configuration in `pipeline_config.yml`.

Class name	ID		Class name	ID	
	YOLO / YOLOX	EfficientDet		YOLO / YOLOX	EfficientDet
person	0	0	elephant	20	21
bicycle	1	1	bear	21	22
car	2	2	zebra	22	23
motorcycle	3	3	giraffe	23	24
aeroplane	4	4	backpack	24	26
bus	5	5	umbrella	25	27
train	6	6	handbag	26	30
truck	7	7	tie	27	31
boat	8	8	suitcase	28	32
traffic light	9	9	frisbee	29	33
fire hydrant	10	10	skis	30	34
stop sign	11	12	snowboard	31	35
parking meter	12	13	sports ball	32	36
bench	13	14	kite	33	37
bird	14	15	baseball bat	34	38
cat	15	16	baseball glove	35	39
dog	16	17	skateboard	36	40
horse	17	18	surfboard	37	41
sheep	18	19	tennis racket	38	42
cow	19	20	bottle	39	43

Class name	ID		Class name	ID	
	YOLO / YOLOX	EfficientDet		YOLO / YOLOX	EfficientDet
wine glass	40	45	dining table	60	66
cup	41	46	toilet	61	69
fork	42	47	tv	62	71
knife	43	48	laptop	63	72
spoon	44	49	mouse	64	73
bowl	45	50	remote	65	74
banana	46	51	keyboard	66	75
apple	47	52	cell phone	67	76
sandwich	48	53	microwave	68	77
orange	49	54	oven	69	78
broccoli	50	55	toaster	70	79
carrot	51	56	sink	71	80
hot dog	52	57	refrigerator	72	81
pizza	53	58	book	73	83
donut	54	59	clock	74	84
cake	55	60	vase	75	85
chair	56	61	scissors	76	86
couch	57	62	teddy bear	77	87
potted plant	58	63	hair drier	78	88
bed	59	64	toothbrush	79	89

Face Detection

This table provides the associated indices for the `model.yolo_face` node.

Class name	ID
no mask	0
mask	1

5.2 Pose Estimation Models

5.2.1 List of Pose Estimation Models

The table below shows the pose estimation models available for each task category.

Category	Model	Documentation
Whole body	HRNet	`model.hrnet`
	PoseNet	`model.posenet`
	MoveNet	`model.movenet`

5.2.2 Benchmarks

Inference Speed

The table below shows the frames per second (FPS) of each model type.

Model	Type	Size	CPU		GPU	
			single	multiple	single	multiple
PoseNet	50	225	64.46	51.95	136.31	89.37
	75	225	57.62	47.01	132.84	83.73
	100	225	44.70	37.60	132.73	81.24
	resnet	225	18.77	17.21	73.15	51.65
HRNet (YOLO)	(v4tiny)	256 × 192 (416)	5.86	1.09	21.91	13.86
MoveNet	SinglePose Lightning	192	40.78	40.54	99.47	–
	SinglePose Thunder	256	25.13	24.87	92.05	–
	MultiPose Lightning	256 or multiple of 32	25.33	24.90	80.64	79.32

Hardware

The following hardware were used to conduct the FPS benchmarks:

- `CPU`: 2.8 GHz 4-Core Intel Xeon (2020, Cascade Lake) CPU and 16GB RAM
- `GPU`: NVIDIA A100, paired with 2.2 GHz 6-Core Intel Xeon CPU and 85GB RAM

Test Conditions

The following test conditions were followed:

- *`input.visual`*, the model of interest, and *`dabble.fps`* nodes were used to perform inference on videos
- 2 videos were used to benchmark each model, one with only 1 human (`single`), and the other with multiple humans (`multiple`)
- Both videos are about 1 minute each, recorded at ~30 FPS, which translates to about 1,800 frames to process per video
- 1280×720 (HD ready) resolution was used, as a bridge between 640×480 (VGA) of poorer quality webcams, and 1920×1080 (Full HD) of CCTVs

Model Accuracy

The table below shows the performance of our pose estimation models using the keypoint evaluation metrics from COCO. Description of these metrics can be found here.

Model	Type	Size	AP	AP OKS=.50	AP OKS=.75	AP medium	AP large	AR	AR OKS=.50	AR OKS=.75	AR medium	AR large
PoseNet	50	225	5.2	15.5	2.7	0.8	11.8	9.6	22.7	7.1	1.4	20.7
	75	225	7.2	19.7	3.6	1.3	15.9	12.1	26.5	9.3	2.2	25.5
	100	225	7.7	20.8	4.4	1.5	17.1	12.6	27.7	10.1	2.3	26.5
	resnet	225	11.9	27.4	8.3	2.2	25.3	17.3	32.5	15.9	2.9	36.8
HRNet (YOLO)	(v4tiny)	256 × 192 (416)	35.8	61.5	37.5	30.1	44.0	40.2	64.4	42.7	33.0	50.2
MoveNet	single-pose_lightning	256 x 256	7.3	15.7	5.7	1.3	15.4	8.8	17.6	7.7	1.1	19.2
	single-pose_thunder	256 x 256	11.6	21.3	10.7	3.0	23.1	13.1	22.5	12.8	2.8	27.1
	multi-pose_lightning	256 x 256	18.7	36.8	16.3	9.0	31.8	21.0	38.5	19.2	9.3	37.0

Dataset

The MS COCO (val 2017) dataset is used. We integrated the COCO API into the PeekingDuck pipeline for loading the annotations and evaluating the outputs from the models. All values are reported in percentages.

All images from the "person" category in the MS COCO (val 2017) dataset were processed.

Test Conditions

The following test conditions were followed:

- The tests were performed using pycocotools on the MS COCO dataset
- The evaluation metrics have been compared with the original repository of the respective pose estimation models for consistency

5.2.3 Keypoint IDs

Whole Body

Keypoint	ID	Keypoint	ID
nose	0	left wrist	9
left eye	1	right wrist	10
right eye	2	left hip	11
left ear	3	right hip	12
right ear	4	left knee	13
left shoulder	5	right knee	14
right shoulder	6	left ankle	15
left elbow	7	right ankle	16
right elbow	8		

5.3 Object Tracking Models

5.3.1 List of Object Tracking Models

The table below shows the object tracking models available for each task category.

Category	Model	Documentation
General	IoU Tracker	`dabble.tracking`
	OpenCV MOSSE Tracker	`dabble.tracking`
Human	JDE	`model.jde`
	FairMOT	`model.fairmot`

5.3.2 Benchmarks

Inference Speed

The table below shows the frames per second (FPS) of each model type.

Model	Object Detector Type	Input Size	CPU	GPU
IoU Tracker with YOLOX	yolox-m	–	7.87	36.18
OpenCV MOSSE Tracker with YOLOX	yolox-m	–	6.74	21.45
JDE	–	–	1.86	26.32
FairMOT	–	864 × 480	0.30	22.60

Hardware

The following hardware were used to conduct the FPS benchmarks:

- `CPU`: 2.8 GHz 4-Core Intel Xeon (Cascade Lake) CPU and 16GB RAM
- `GPU`: NVIDIA A100, paired with 2.2 GHz 6-Core Intel Xeon CPU and 85GB RAM

Test Conditions

The following test conditions were followed:

- `input.visual`, the model of interest, and `dabble.fps` nodes were used to perform inference on videos
- A video sequence from the MOT Challenge dataset (MOT16-04) was used
- The video sequence has 1050 frames and is encoded at 30 FPS, which translates to about 35 seconds
- 1280×720 (HD ready) resolution was used, as a bridge between 640×480 (VGA) of poorer quality webcams, and 1920×1080 (Full HD) of CCTVs

Model Accuracy

The table below shows the performance of our object tracking models using multiple object tracker (MOT) metrics from MOT Challenge. Description of these metrics can be found here.

Model	Object Detector Type	MOTA	IDF1	ID Sw.	FP	FN
IoU Tracker with YOLOX	yolox-m	34.1	40.9	960	8997	62830
OpenCV MOSSE Tracker with YOLOX	yolox-m	32.8	38	2349	7695	65268
JDE	–	70.1	65.1	1321	6412	25292
FairMOT	–	81.8	80.9	536	3663	15903

Dataset

The MOT16 (train) dataset is used. We integrated the MOT Challenge API into the PeekingDuck pipeline for loading the annotations and evaluating the outputs from the models. *MOTA* and *IDF1* are reported in percentages while *IDS*, *FP*, and *FN* are raw numbers.

Only the "pedestrian" category in MOT16 (train) was processed.

5.4 Crowd Counting Models

5.4.1 List of Crowd Counting Models

The table below shows the crowd counting models available.

Model	Documentation
CSRNet	`model.csrnet`

5.4.2 Benchmarks

Model Accuracy

The table below shows the performance of CSRNet obtained from the original GitHub repo, using Mean Absolute Error (MAE) as the metric. The reported metrics are close to the results from the CSRNet paper.

Model	Type	Dataset	MAE
CSRNet	dense	ShanghaiTech Part A	65.92
	sparse	ShanghaiTech Part B	11.01

Dataset

The ShanghaiTech dataset was used. It contains 1,198 annotated images split into 2 parts: Part A contains 482 images with highly congested scenes, while Part B contains 716 images with relatively sparse crowd scenes.

5.5 Instance Segmentation Models

5.5.1 List of Instance Segmentation Models

The table below shows the instance segmentation models available.

Model	Documentation
Mask R-CNN	*model.mask_rcnn*
YolactEdge	*model.yolact_edge*

5.5.2 Benchmarks

Inference Speed

The table below shows the frames per second (FPS) of each model type.

Model	Type	Size	CPU		GPU	
			single	multiple	single	multiple
Mask R-CNN	r50-fpn	800-1333	0.76	0.72	22.30	18.58
	r101-fpn	800-1333	0.61	0.57	17.14	14.83
YolactEdge	r50-fpn	550	2.99	2.93	40.84	33.94
	r101-fpn	550	2.32	2.27	29.55	25.89
	mobilenetv2	550	4.93	4.64	48.59	36.66

Hardware

The following hardware were used to conduct the FPS benchmarks:

- `CPU`: 2.8 GHz 4-Core Intel Xeon (2020, Cascade Lake) CPU and 16GB RAM
- `GPU`: NVIDIA A100, paired with 2.2 GHz 6-Core Intel Xeon CPU and 85GB RAM

Test Conditions

The following test conditions were followed:

- *input.visual*, the model of interest, and *dabble.fps* nodes were used to perform inference on videos
- 2 videos were used to benchmark each model, one with only 1 human (`single`), and the other with multiple humans (`multiple`)
- Both videos are about 1 minute each, recorded at ~30 FPS, which translates to about 1,800 frames to process per video

- 1280×720 (HD ready) resolution was used, as a bridge between 640×480 (VGA) of poorer quality webcams, and 1920×1080 (Full HD) of CCTVs

Model Accuracy

The table below shows the performance of our Instance Segmentation models using the detection evaluation metrics from COCO. Description of these metrics can be found here.

Evaluation on masks

Model	Type	Size	AP	AP IoU=.50	AP IoU=.75	AP small	AP medium	AP large	AR max=1	AR max=10	AR max=100	AR small	AR medium	AR large
Mask R-CNN	r50-fpn	800-1333	34.5	56.0	36.7	17.8	37.9	47.1	29.7	45.6	47.6	27.4	51.4	63.8
	r101-fpn	800-1333	37.1	59.0	39.6	20.4	41.1	49.8	31.4	49.1	51.4	31.9	55.6	67.3
Yolact-Edge	r50-fpn	550	27.8	45.6	28.9	10.4	30.0	43.9	26.3	37.5	38.2	16.3	41.9	57.2
	r101-fpn	550	29.6	47.8	31.1	11.3	32.3	46.3	27.4	38.9	39.7	17.4	43.6	59.6
	mobilenetv2	550	21.9	37.2	22.6	7.0	22.9	34.7	22.5	31.7	32.3	12.0	34.8	48.3

Evaluation on bounding boxes

Model	Type	Size	AP	AP IoU=.50	AP IoU=.75	AP small	AP medium	AP large	AR max=1	AR max=10	AR max=100	AR small	AR medium	AR large
Mask R-CNN	r50-fpn	800-1333	37.8	59.2	41.1	21.6	41.2	49.3	31.4	49.5	51.9	32.6	55.7	66.6
	r101-fpn	800-1333	41.8	62.2	45.4	24.9	45.8	54.3	34.4	54.6	57.3	38.2	61.4	72.4
Yolact-Edge	r50-fpn	550	30.3	49.8	32.2	14.4	32.1	44.6	27.4	40.1	41.2	21.6	43.7	57.5
	r101-fpn	550	32.6	52.5	34.9	15.2	35.0	47.6	28.6	41.8	42.9	22.6	45.9	59.9
	mobilenetv2	550	23.2	40.8	23.8	9.3	23.4	35.1	22.9	33.5	34.5	15.8	35.2	49.1

“”””

Dataset

The MS COCO (val 2017) dataset is used. We integrated the COCO API into the PeekingDuck pipeline for loading the annotations and evaluating the outputs from the models. All values are reported in percentages.

All images from the 80 object categories in the MS COCO (val 2017) dataset were processed.

Test Conditions

The following test conditions were followed:

- The tests were performed using pycocotools on the MS COCO dataset
- The evaluation metrics have been compared with the original repository of the respective instance segmentation models for consistency

5.5.3 Instance Segmentation IDs

General Instance Segmentation

The tables below provide the associated indices for each class.
To detect all classes, specify `detect: ["*"]` under the instance segmentation node configuration in `pipeline_config.yml`.

Class name	ID		Class name	ID	
	Mask R-CNN	YolactEdge		Mask R-CNN	YolactEdge
person	0	0	elephant	21	20
bicycle	1	1	bear	22	21
car	2	2	zebra	23	22
motorcycle	3	3	giraffe	24	23
aeroplane	4	4	backpack	26	24
bus	5	5	umbrella	27	25
train	6	6	handbag	30	26
truck	7	7	tie	31	27
boat	8	8	suitcase	32	28
traffic light	9	9	frisbee	33	29
fire hydrant	10	10	skis	34	30
stop sign	12	11	snowboard	35	31
parking meter	13	12	sports ball	36	32
bench	14	13	kite	37	33
bird	15	14	baseball bat	38	34
cat	16	15	baseball glove	39	35
dog	17	16	skateboard	40	36
horse	18	17	surfboard	41	37
sheep	19	18	tennis racket	42	38
cow	20	19	bottle	43	39

Class name	ID		Class name	ID	
	Mask R-CNN	YolactEdge		Mask R-CNN	YolactEdge
wine glass	45	40	dining table	66	60
cup	46	41	toilet	69	61
fork	47	42	tv	71	62
knife	48	43	laptop	72	63
spoon	49	44	mouse	73	64
bowl	50	45	remote	74	65
banana	51	46	keyboard	75	66
apple	52	47	cell phone	76	67
sandwich	53	48	microwave	77	68
orange	54	49	oven	78	69
broccoli	55	50	toaster	79	70
carrot	56	51	sink	80	71
hot dog	57	52	refrigerator	81	72
pizza	58	53	book	83	73
donut	59	54	clock	84	74
cake	60	55	vase	85	75
chair	61	56	scissors	86	76
couch	62	57	teddy bear	87	77
potted plant	63	58	hair drier	88	78
bed	64	59	toothbrush	89	79

5.6 Bibliography

This document contains links, references, academic literature, and github repositories for related Computer Vision technologies and projects.

5.6.1 Legend

Symbol	Remarks
	Available in PeekingDuck
	Singapore-based research

5.6.2 Object Detection

Reference	Paper	Code
YOLOX	✓	✓
YOLOv4	✓	✓
EfficientDet	✓	✓
MTCNN	✓	✓
Recent advances in deep learning for object detection (2020)	✓	NA

5.6.3 Pose Estimation

Reference	Paper	Code
HRNet	✓	✓
PoseNet	✓	✓
MoveNet	TF Blog	TF Hub
NTU RGB+D Dataset (2016)	✓	NA

5.6.4 Crowd Counting

Reference	Paper	Code
CSRNet	✓	✓

5.6.5 Object Tracking

Reference	Paper	Code
JDE	✓	✓
FairMOT	✓	✓

5.6.6 Instance Segmentation

Reference	Paper	Code
Mask R-CNN	✓	Torchvision Models
YolactEdge	✓	✓

CHAPTER

SIX

EDGE AI

PeekingDuck supports running optimized TensorRT[1] models on devices with NVIDIA GPUs. Using the TensorRT model on these devices provides a speed boost over the regular TensorFlow/PyTorch version. A potential use case is running PeekingDuck on an NVIDIA Jetson device for Edge AI inference.

Currently, PeekingDuck includes TensorRT versions of the following models:

1. MoveNet model for pose estimation,
2. YOLOX model for object detection.

6.1 Installing TensorRT

The following packages are required to run PeekingDuck's TensorRT models:

1. TensorFlow
2. PyTorch
3. PyCUDA

As the actual installation steps vary greatly depending on the user's device, operating system, software environment, and pre-installed libraries/packages, we are unable to provide step-by-step installation instructions.

The user may refer to NVIDIA's TensorRT Documentation for detailed TensorRT installation information.

6.2 Using TensorRT Models

To use the TensorRT version of a model, change the `model_format` of the model configuration to `tensorrt`.

The following `pipeline_config.yml` shows how to use the MoveNet TensorRT model for pose estimation:

```
nodes:
- input.visual:
    source: https://storage.googleapis.com/peekingduck/videos/wave.mp4
- model.movenet:
    model_format: tensorrt
    model_type: singlepose_lightning
- draw.poses
- dabble.fps
- draw.legend:
```

(continues on next page)

[1] NVIDIA TensorRT Reference

(continued from previous page)

```
    show: ["fps"]
- output.screen
```

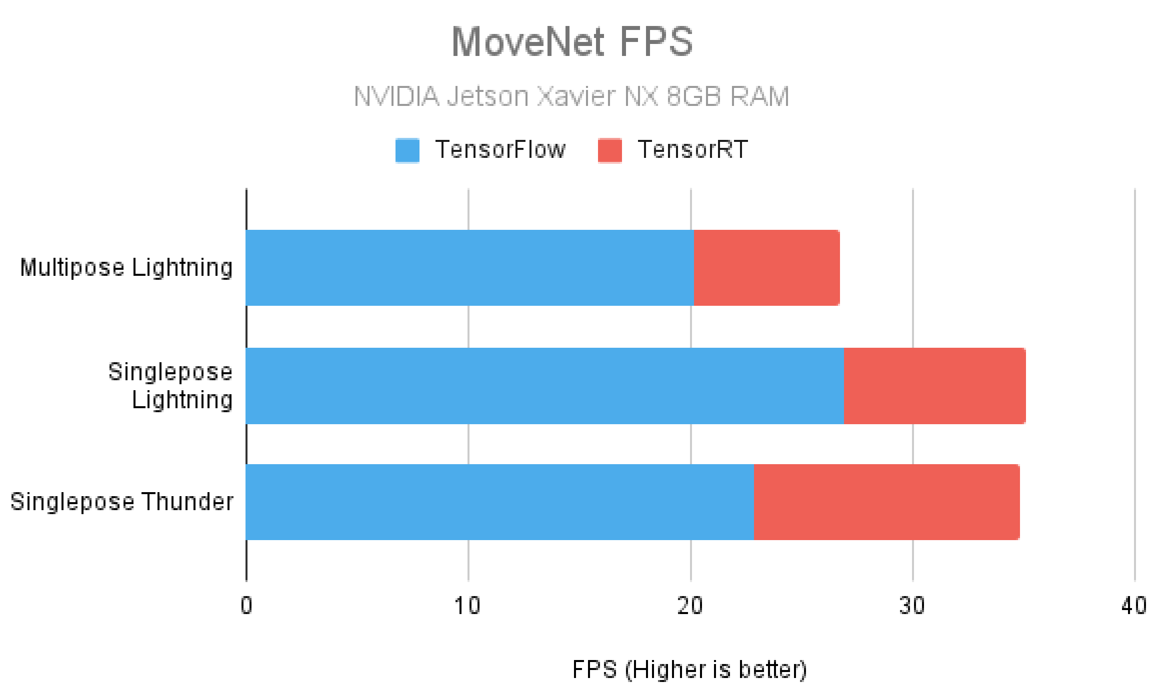
The following `pipeline_config.yml` shows how to use the YOLOX TensorRT model for object detection:

```
nodes:
- input.visual:
    source: https://storage.googleapis.com/peekingduck/videos/cat_and_computer.mp4
- model.yolox:
    detect: ["cup", "cat", "laptop", "keyboard", "mouse"]
    model_format: tensorrt
    model_type: yolox-tiny
- draw.bbox:
    show_labels: True    # configure draw.bbox to display object labels
- dabble.fps
- draw.legend:
    show: ["fps"]
- output.screen
```

6.3 Performance Speedup

The following charts show the speed up obtainable with the TensorRT models. The numbers were obtained from our in-house testing with the actual devices.

6.3.1 NVIDIA Jetson Xavier NX with 8GB RAM



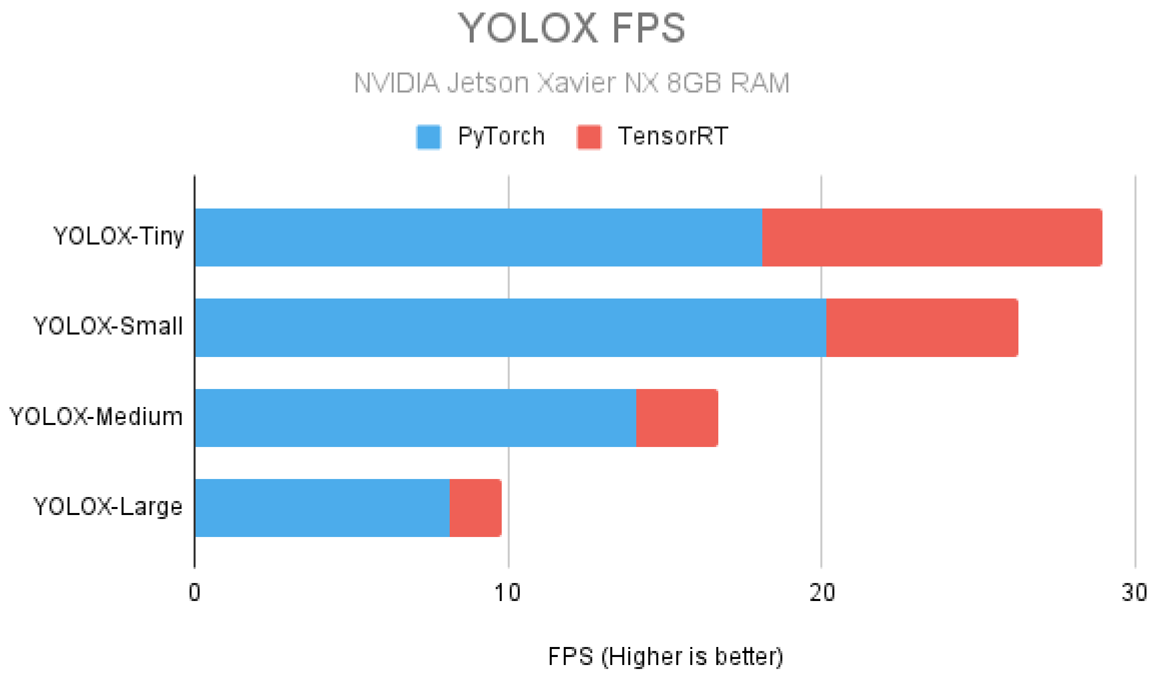


Fig. 1: Jetson Xavier NX specs used for testing:[Page 83, 2]
`CPU`: 6 cores (6MB L2 + 4MB L3) `GPU`: 384-core Volta, 48 Tensor cores `RAM`: 8 GB

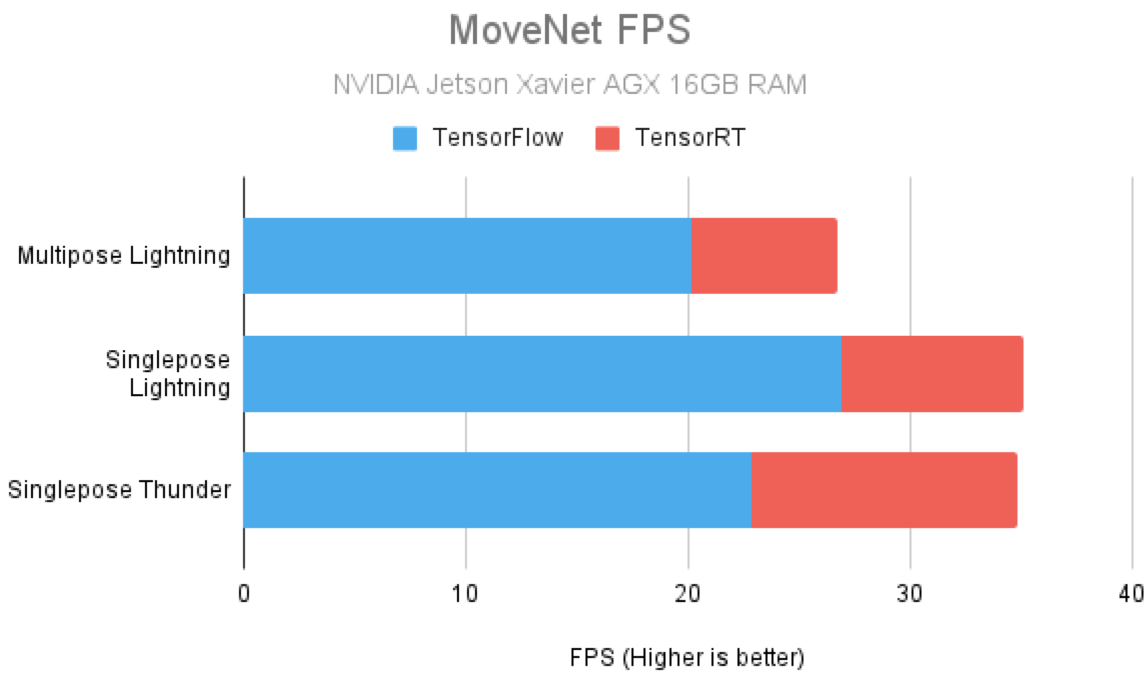
6.3.2 NVIDIA Jetson Xavier AGX with 16GB RAM

Test Conditions

The following test conditions were followed:

- *`input.visual`*, the model of interest, and *`dabble.fps`* nodes were used to perform inference on videos
- 2 videos were used to benchmark each model, one with only 1 human (`single`), and the other with multiple humans (`multiple`)
- Both videos are about 1 minute each, recorded at ~30 FPS, which translates to about 1,800 frames to process per video
- 1280×720 (HD ready) resolution was used, as a bridge between 640×480 (VGA) of poorer quality webcams, and 1920×1080 (Full HD) of CCTVs
- FPS numbers are averaged over 5 separate runs

[2] NVIDIA Jetson Xavier NX Tech Specs
[3] NVIDIA Jetson Xavier AGX Tech Specs



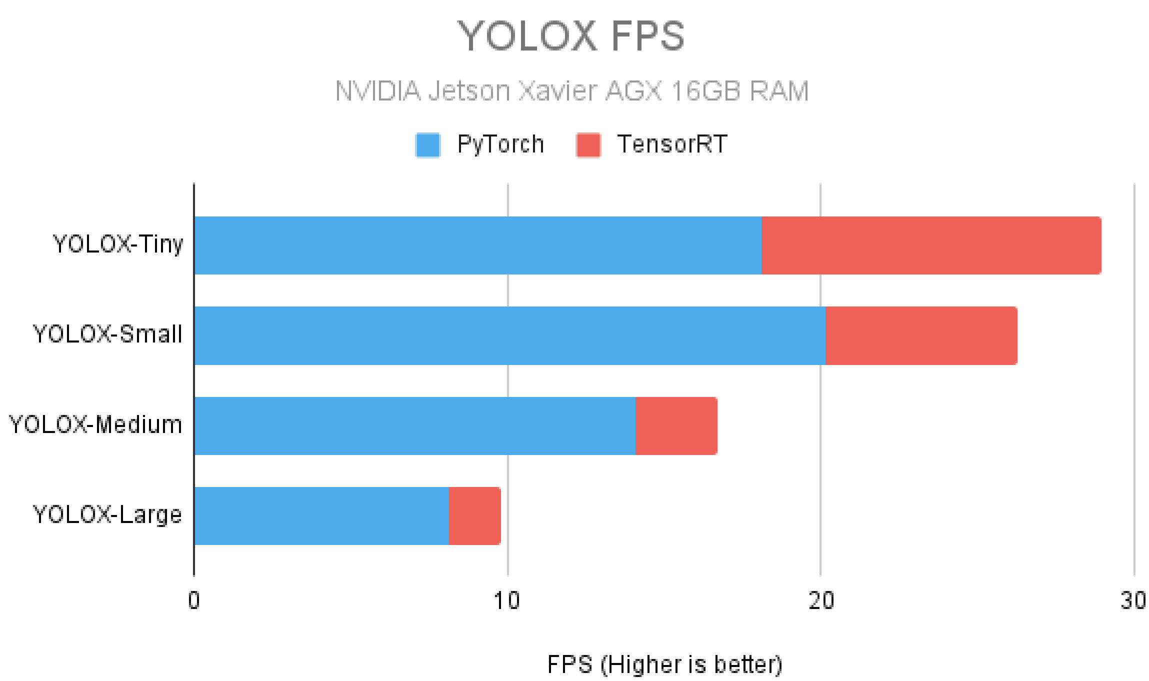


Fig. 2: Jetson Xavier AGX specs used for testing:[3]
CPU: 8 cores (8MB L2 + 4MB L3) GPU: 512-core Volta, 64 Tensor cores RAM: 16 GB

6.4 References

CHAPTER

SEVEN

USE CASES

Computer Vision (CV) problems come in various forms, and the gallery below shows common CV use cases which can be tackled by PeekingDuck right out of the box. Areas include privacy protection, smart monitoring, and COVID-19 prevention and control. Users are encouraged to mix and match different PeekingDuck nodes and create your own *custom nodes* for your specific use case - the only limit is your imagination!

7.1 Privacy Protection

7.1.1 Privacy Protection (Faces)

Overview

As organizations collect more data, there is a need to better protect the identities of individuals in public and private places. Our solution performs face anonymization, and can be used to comply with the General Data Protection Regulation (GDPR) or other data privacy laws.

Our solution automatically detects and mosaics (or blurs) human faces. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file privacy_protection_faces.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/privacy_protection_faces.yml>
```

How It Works

There are two main components to face anonymization:

1. Face detection, and
2. Face de-identification.

1. Face Detection

We use an open source face detection model known as MTCNN to identify human faces. This allows the application to identify the locations of human faces in a video feed. Each of these locations is represented as a pair of *x, y* coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each human face detected. For more information on how to adjust the MTCNN node, check out the *MTCNN configurable parameters*.

2. Face De-Identification

To perform face de-identification, we pixelate or gaussian blur the areas bounded by the bounding boxes.

Nodes Used

These are the nodes used in the earlier demo (also in privacy_protection_faces.yml):

```
nodes:
- input.visual:
    source: 0
- model.mtcnn
- draw.mosaic_bbox
- output.screen
```

1. Face Detection Node

As mentioned, we use the MTCNN model for face detection. It is able to detect human faces with face masks. Please take a look at the *benchmarks* of object detection models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Face De-Identification Nodes

You can mosaic or blur the faces detected using the `draw.mosaic_bbox` or `draw.blur_bbox` in the run config declaration.

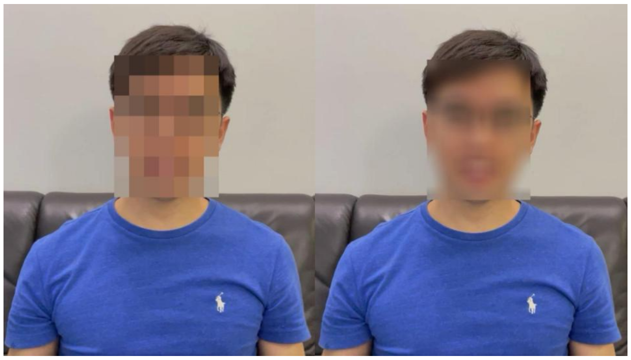


Fig. 1: De-identification with mosaic (left) and blur (right).

3. Adjusting Nodes

With regard to the MTCNN model, some common node behaviors that you might want to adjust are:

- `min_size`: Specifies in pixels the minimum height and width of a face to be detected. (default = 40) You may want to decrease the minimum size to increase the number of detections.
- `network_thresholds`: Specifies the threshold values for the Proposal Network (P-Net), Refine Network (R-Net), and Output Network (O-Net) in the MTCNN model. (default = [0.6, 0.7, 0.7]) Calibration is performed at each stage in which bounding boxes with confidence scores less than the specified threshold are discarded.
- `score_threshold`: Specifies the threshold value in the final output. (default = 0.7) Bounding boxes with confidence scores less than the specified threshold in the final output are discarded. You may want to lower `network_thresholds` and `score_threshold` to increase the number of detections.

In addition, some common node behaviors that you might want to adjust for the `dabble.mosaic_bbox` and `dabble.blur_bbox` nodes are:

- `mosaic_level`: Defines the resolution of a mosaic filter ($width \times height$); the value corresponds to the number of rows and columns used to create a mosaic. (default = 7) For example, the default value creates a 7×7 mosaic filter. Increasing the number increases the intensity of pixelization over an area.
- `blur_level`: Defines the standard deviation of the Gaussian kernel used in the Gaussian filter. (default = 50) The higher the blur level, the greater the blur intensity.

7.1.2 Privacy Protection (License Plates)

Overview

Posting images or videos of our vehicles online might lead to others misusing our license plate numbers to reveal our personal information. Our solution performs license plate anonymization, and can also be used to comply with the General Data Protection Regulation (GDPR) or other data privacy laws.

Our solution automatically detects and blurs vehicles' license plates. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file privacy_protection_license_plates.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/privacy_protection_license_plates.yml>
```

How It Works

There are two main components to license plate anonymization:

1. License plate detection, and
2. License plate de-identification.

1. License Plate Detection

We use open-source object detection models under the YOLOv4 family to identify the locations of the license plates in an image/video feed. Specifically, we offer the YOLOv4-tiny model, which is faster, and the YOLOv4 model, which provides higher accuracy. The locations of detected license plates are returned as an array of coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each license plate detected. For more information on how to adjust the license plate detector node, check out the *license plate detector configurable parameters*.

2. License Plate De-Identification

To perform license plate de-identification, the areas bounded by the bounding boxes are blurred using a Gaussian blur function.

Nodes Used

These are the nodes used in the earlier demo (also in privacy_protection_license_plates.yml):

```
nodes:
- input.visual:
    source: <path/to/video with cars>
- model.yolo_license_plate
- draw.blur_bbox
- output.screen
```

1. License Plate Detection Node

By default, `model.yolo_license_plate` uses the `v4` model type to detect license plates. If faster inference speed is required, the `v4tiny` model type can be used instead.

2. License Plate De-Identification Nodes

You can choose to mosaic or blur the detected license plate using the `draw.mosaic_bbox` or `draw.blur_bbox` node in the run config declaration.

Fig. 2: De-identification with mosaic (left) and blur (right).

3. Adjusting Nodes

With regard to the YOLOv4 model, some common node configurations that you might want to adjust are:

- `score_threshold`: The bounding boxes with confidence score less than the specified score threshold are discarded. (default = 0.1)
- `iou_threshold`: The overlapping bounding boxes above the specified Intersection over Union (IoU) threshold are discarded. (default = 0.3)

In addition, some common node behaviors that you might want to adjust for the `dabble.mosaic_bbox` and `dabble.blur_bbox` nodes are:

- `mosaic_level`: Defines the resolution of a mosaic filter ($width \times height$); the value corresponds to the number of rows and columns used to create a mosaic. (default = 7) For example, the default value creates a 7×7 mosaic filter. Increasing the number increases the intensity of pixelization over an area.
- `blur_level`: Defines the standard deviation of the Gaussian kernel used in the Gaussian filter. (default = 50) The higher the blur level, the greater the blur intensity.

7.1.3 Privacy Protection (People & Computer Screens)

Overview

Videos and pictures often contain people and other sensitive visual information (e.g., the display on computer screens), even though this information might not be needed at all for visual processing. Our solution performs full body anonymization and computer screen blurring to protect the identities of individuals and the sensitive information on computer screens. It can be used to comply with the General Data Protection Regulation (GDPR) or other data privacy laws.

In this example use case, we want to count the number of people in the office, but also want to avoid compromising the privacy of the office inhabitants or information displayed on computer screens.

Our solution automatically detects people, laptop and computer screens, and then blurs them. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file privacy_protection_people_screens.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/privacy_protection_people_screens.yml>
```

How It Works

There are 2 main components to our solution:

1. Person and computer screen segmentation, and
2. Person and computer screen blurring.

1. Person and Computer Screen Segmentation

We use an open source instance segmentation model known as Mask R-CNN to obtain the masks of persons, computer screens and laptops. The masks are akin to the input frames or images, except that it only has a single channel and each pixel on the mask is a binary of either 1 or 0, which indicates whether a specific class of object is present (1) or absent (0) in a particular location of the image. For more information on how to adjust the `mask_rcnn` node, check out its *configurable parameters*.

2. Person and Computer Screen Blurring

To blur the people and computer screens, we pixelate or gaussian blur the image pixels where the pixel values of the relevant masks are equal to 1 (indicating presence of object).

Nodes Used

These are the nodes used in the earlier demo (also in privacy_protection_people_screens.yml):

```
nodes:
- input.visual:
    source: <path/to/video>
- model.mask_rcnn:
    detect: ["tv", "laptop"]
- draw.instance_mask:
    effect: {blur: 50}
- model.mask_rcnn:
    detect: ["person"]
- dabble.bbox_count
- draw.instance_mask:
    effect: {blur: 50}
- draw.bbox:
    show_labels: True
- draw.legend:
    show: ["count"]
- output.screen
```

This config includes the use of two model.mask_rcnn and draw.instance_mask nodes to separate the detected instances of "person" class from the "tv" and "laptop" classes, so that drawing and counting of bboxes are only performed on the "person" class. This repetition is not required if only anonymization is performed.

1. Instance Segmentation Node

In this example use case, we used the Mask R-CNN model for instance segmentation. It can detect persons as well as computer monitors. Please take a look at the *benchmarks* of instance segmentation models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. People and Screens De-Identification Node

The detected people and screens are blurred using the `draw.instance_mask` node.

3. Object Counting Node

dabble.bbox_count counts the total number of detected bounding boxes. This node has no configurable parameters.

4. Display Bounding Box Node

Then we draw bounding boxes around the detected persons using the *draw.bbox* node.

5. Person Count Display Node

The total number of detected persons are shown using the *draw.legend* node.

6. Adjusting Nodes

With regard to the Mask R-CNN model, some common node behaviors that you might want to adjust are:

- `model_type`: Defines the type of backbones to be used.
- `score_threshold`: Bounding boxes with classification score below the threshold will be discarded.
- `mask_threshold`: The confidence threshold for binarizing the masks' pixel values, whether an object is detected at a particular pixel.

In addition, some common node behaviors that you might want to adjust for the *draw.instance_mask* node are:

- `blur`: Blurs the area using this value as the "blur_kernel_size" parameter. Larger values gives more intense blurring.
- `mosaic`: Mosaics the area using this value as the resolution of a mosaic filter ($width \times height$). The value corresponds to the number of rows and columns used to create a mosaic. For example, the setting (`mosaic: 25`) creates a 25×25 mosaic filter. Increasing the number increases the intensity of pixelation over an area.

Privacy Protection (Faces)	*Privacy Protection (License Plates)*
Privacy Protection (People and Screens)	

7.2 Smart Monitoring

7.2.1 Crowd Counting

Overview

In Computer Vision (CV), crowd counting refers to the technique of counting or estimating the number of people in a crowd. This can be used to estimate the number of people attending an event, monitor crowd levels and prevent human stampedes.

Our solution utilizes CSRNet to estimate the size of a crowd. In addition, it generates a heat map that can be used to pinpoint possible bottlenecks at a venue. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file crowd_counting.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/crowd_counting.yml>
```

You may like to try it on this sample video.

How It Works

There are two main components to our solution:

1. Crowd counting, and
2. Heat map generation.

1. Crowd Counting

We use an open source crowd counting model known as CSRNet to predict the number of people in a sparse or dense crowd. The solution uses the sparse crowd model by default and can be configured to use the dense crowd model if required. The dense and sparse crowd models were trained using data from ShanghaiTech Part A and Part B respectively.

As a rule of thumb, you might want to use the dense crowd model if the people in a given image or video frame are packed shoulder to shoulder, e.g., stadiums. For more information on how to adjust the CSRNet node, check out its *configurable parameters*.

2. Heat Map Generation (Optional)

We generate a heat map using the density map estimated by the model. Areas that are more crowded are highlighted in red while areas that are less crowded are highlighted in blue.

Nodes Used

These are the nodes used in the earlier demo (also in crowd_counting.yml):

```
nodes:
- input.visual:
    source: <path/to/video with crowd>
- model.csrnet:
    model_type: dense
- draw.heat_map
- draw.legend:
    show: ["count"]
- output.screen
```

1. Crowd Counting Node

As mentioned, we use CSRNet to estimate the size of a crowd. As the models were trained to recognize congested scenes, the estimates are less accurate if the number of people is low, i.e., below ten. In such scenarios, you should consider using the *object detection models* included in our repo.

2. Heat Map Generation Node (Optional)

The heat map generation node superimposes a heat map over a given image or video frame.

3. Adjusting Nodes

Some common node behaviors that you might want to adjust are:

- `model_type`: This specifies the model to be used, i.e., `sparse` or `dense`. By default, our solution uses the sparse crowd model. As a rule of thumb, you might want to use the dense crowd model if the people in a given image or video frame are packed shoulder to shoulder, e.g., stadiums.
- `width`: This specifies the input width. By default, the width of an image will be resized to 640 for inference. The height of the image will be resized proportionally to preserve its aspect ratio. In general, decreasing the width of an image will improve inference speed. However, this might impact the accuracy of the model.

7.2.2 Object Counting (Present)

Overview

Object counting (present) is a solution within PeekingDuck's suite of *smart monitoring* use cases. It counts the number of objects detected by PeekingDuck's object detection models at the present point in time, and calculates statistics such as the cumulative average, maximum and minimum for further analytics. Up to *80 types* of objects can be counted, including humans, vehicles, animals and even household objects. Thus, this can be applied to a wide variety of scenarios, from traffic control to counting livestock.

See also:

For advanced counting tasks such as counting tracked objects over time or counting within specific zones, refer to PeekingDuck's other *smart monitoring* use cases.

In the GIF above, the count and statistics change as the number of detected persons change. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file object_counting_present.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/object_counting_present.yml>
```

How It Works

There are 3 main components to this solution:

1. Object detection,
2. Count detections, and
3. Calculate statistics.

1. Object Detection

We use an open source object detection model known as YOLOv4 and its smaller and faster variant known as YOLOv4-tiny to identify the bounding boxes of chosen objects we want to detect. This allows the application to identify where objects are located within the video feed. The location is returned as two *x, y* coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each object detected. For more information on how to adjust the `yolo` node, check out its *configurable parameters*.

2. Count Detections

To count the number of objects detected, we simply take the sum of the number of bounding boxes detected for the object category.

3. Calculate Statistics

The cumulative average, minimum and maximum over time is calculated from the count from each frame.

Nodes Used

These are the nodes used in the earlier demo (also in object_counting_present.yml):

```
nodes:
- input.visual:
    source: 0
- model.yolo:
    detect: ["person"]
- dabble.bbox_count
- dabble.statistics:
    identity: count
- draw.bbox
- draw.legend:
    show: ["count", "cum_avg", "cum_max", "cum_min"]
- output.screen
```

1. Object Detection Node

By default, the node uses the YOLOv4-tiny model for object detection, set to detect people. Please take a look at the *benchmarks* of object detection models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Object Counting Node

`dabble.bbox_count` takes the detected bounding boxes and outputs the total count of bounding boxes. This node has no configurable parameters.

3. Statistics Node

The `dabble.statistics` node calculates the *cum_avg*, *cum_max* and *cum_min* from the output of the object counting node.

4. Adjusting Nodes

For the object detection model used in this demo, please see the *documentation* for adjustable behaviors that can influence the result of the object counting node.

For more adjustable node behaviors not listed here, check out the *API Documentation*.

7.2.3 Object Counting (Over Time)

Overview

Object counting over time involves detecting and tracking unique objects, and incrementing the count when new objects appear. When applied to the vehicles in the GIF below, it can count the total number of vehicles passing by over a period of time, aiding transportation planning by identifying periods of peak traffic. This use case is not limited to just vehicles, as up to *80 types* of objects can be monitored (including animals), giving rise to a wide breadth of potential applications.

See also:

While it is also possible to count people over time with this use case, more accurate results can be obtained by using the *People Counting (Over Time)* use case.

See also:

If you wish to only count the number objects at an instance rather than a cumulative total over a period of time, the simpler *Object Counting (Present)* use case without requiring object tracking would be more suitable.

Object counting over time is achieved by detecting the objects using an object detection model, then tracking each unique object. As a new object appears, the number of counted objects is incremented. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file object_counting_over_time.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/object_counting_over_time.yml>
```

How It Works

Object counting over time comprises three main components:

1. Object detection,
2. Tracking the outputs of object detection, and
3. Incrementing the count.

1. Object Detection

The EfficientDet model is used here to predict the bounding boxes of objects of interest. This allows the application to identify where each object is located within the video feed. The location is returned as a pair of *x, y* coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top-left corner of the bounding box, and (x_2, y_2) is the bottom right.

2. Tracking the Outputs of Object Detection

An Intersection over Union (IoU) tracker adapted from this paper is used on the bounding boxes from the object detection model to produce tracked identities (IDs) for each bounding box. The IoU tracker continues a track by associating the detection with the highest IoU to the last detection in the previous frame. For example, Car 8 in frame **n** continues to be tracked as Car 8 in frame **n+1** as both instances of Car 8 are within close proximity (high IoU) of each other. This assumes that the object detector correctly predicts a bounding box per frame for each object to be tracked, and also assumes that the frame rate of the video is high enough to allow unambigious IoU overlaps between consecutive frames.

Another available option is the Minimum Output Sum of Squared Error (MOSSE) tracker which we have adapted from the OpenCV package. It is a correlation filter based tracker which uses Fast Fourier Transform (FFT) to perform operations in the frequency domain, reducing computational complexity. More details can be found from this paper.

3. Incrementing the Count

Monotonically increasing integer IDs beginning from *0* are assigned to new unique objects. For example, the first tracked object is assigned an ID of *0*, the second tracked object is assigned an ID of *1*, and so on. Thus the total number of unique objects that have appeared in the entire duration is simply the cumulative maximum.

Nodes Used

These are the nodes used in the earlier demo (also in object_counting_over_time.yml):

```yaml
nodes:
- input.visual:
    source: <path/to/video with cars>
- model.efficientdet:
    detect: ["car"]
- dabble.tracking:
    tracking_type: "iou"
- dabble.statistics:
    maximum: obj_attrs["ids"]
- draw.bbox
- draw.tag:
    show: ["ids"]
- draw.legend:
    show: ["cum_max"]
- output.screen
```

1. Object Detection Node

In the demo, the `model.efficientdet` node is used for object detection, set to detect cars. As mentioned in the earlier *How It Works* section, for object tracking to work well, the upstream object detector needs to produce predictions which are as accurate as possible. Please take a look at the *benchmarks* of object detection models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Tracking Node

The `dabble.tracking` node used here is not an AI model but uses heuristics, hence it falls under the category of `dabble` nodes instead of `model` nodes. It needs to be paired with an upstream object detector node, but this also gives it a key advantage - it can track any of the *80 types* of detectable objects. In contrast, the *People Counting (Over Time)* use case uses a single model node purpose-built for both human detection and tracking, giving it more accuracy but limiting its usage to only humans.

3. Statistics Node

The `dabble.statistics` node retrieves the maximum detected ID for each frame. If the ID exceeds the previous maximum, the *cum_max* (cumulative maximum) is updated. As monotonically increasing integer IDs beginning from *0* are assigned to new unique objects, the maximum ID is equal to the total number of unique objects over time.

4. Adjusting Nodes

Some common node behaviors that you might need to adjust are:

For `model.efficientdet`:

- `model_type`: `0`, `1`, `2`, `3`, or `4`. The larger the number, the higher the accuracy, at the cost of inference speed.
- `detect`: Object class IDs to be detected. Refer to *Object Detection IDs table* for the class IDs for each model.

For `dabble.tracking`:

- `tracking_type`: Choose either `["iou", "mosse"]`, described earlier in the *How It Works* section.

For more adjustable node behaviors not listed here, check out the *API Documentation*.

Counting Objects Within Zones

It is possible to extend this use case with the *Zone Counting* use case. For example, if the road were a dual carriageway and we are only interested counting the vehicles on one side of the road, we could split the video into 2 different zones and only count the vehicles within the chosen zone. An example of how this can be done is given in the *Tracking People within a Zone* tutorial.

7.2.4 People Counting (Over Time)

Overview

People counting over time involves detecting and tracking different persons, and incrementing the count when a new person appears. This use case can reduce dependency on manual counting, and be applied to areas such as retail analytics, queue management, or occupancy monitoring.

Our solution automatically detects, tracks and counts people over time. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file people_counting_over_time.yml as shown:

Terminal Session

[~user] > peekingduck run --config_path <path/to/people_counting_over_time.yml>

You may like to try it on this sample video.

How It Works

People counting over time comprises three main components:

1. Human detection,
2. Appearance embedding tracking, and
3. Incrementing the count.

1. Human Detection

We use an open source detection model known as JDE to detect persons. JDE has been trained on pedestrian detection and person search datasets. This allows the application to identify the locations of persons in a video feed. Each of these locations is represented as a pair of *x, y* coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each person detected. For more information on how to adjust the JDE node, check out the *JDE configurable parameters*.

2. Appearance Embedding Tracking

To learn appearance embeddings for tracking, a metric learning algorithm with triplet loss is used. Observations are assigned to tracklets using the Hungarian algorithm. The Kalman filter is used to smooth the trajectories and predict the locations of previous tracklets in the current frame. The model outputs an ID for each detection based on the appearance embedding learned.

3. Incrementing the Count

Monotonically increasing integer IDs beginning from *0* are assigned to new unique persons. For example, the first tracked person is assigned an ID of *0*, the second tracked person is assigned an ID of *1*, and so on. Thus the total number of unique persons that have appeared in the entire duration is simply the cumulative maximum.

Nodes Used

These are the nodes used in the earlier demo (also in people_counting_over_time.yml):

```
nodes:
- input.visual:
    source: <path/to/video with people>
- model.jde
- dabble.statistics:
    maximum: obj_attrs["ids"]
- draw.bbox
```

(continues on next page)

(continued from previous page)

```
- draw.tag:
    show: ["ids"]
- draw.legend:
    show: ["cum_max"]
- output.screen
```

1. JDE Node

This node employs a single network to **simultaneously** output detection results and the corresponding appearance embeddings of the detected boxes. Therefore JDE stands for Joint Detection and Embedding. Please take a look at the *benchmarks* of object tracking models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Statistics Node

The `dabble.statistics` node retrieves the maximum detected ID for each frame. If the ID exceeds the previous maximum, the *cum_max* (cumulative maximum) is updated. As monotonically increasing integer IDs beginning from *0* are assigned to new unique persons, the maximum ID is equal to the total number of unique persons over time.

3. Adjusting Nodes

With regard to the `model.jde` node, some common behaviors that you might want to adjust are:

- `iou_threshold`: Specifies the threshold value for Intersection over Union of detections (default = 0.5).
- `score_threshold`: Specifies the threshold values for the detection confidence (default = 0.5). You may want to lower this value to increase the number of detections.
- `nms_threshold`: Specifies the threshold value for non-maximal suppression (default = 0.4). You may want to lower this value to increase the number of detections.
- `min_box_area`: Specifies the minimum value for area of detected bounding box. Calculated by $width \times height$ (default = 200).
- `track_buffer`: Specifies the threshold to remove track if track is lost for more frames than this value (default = 30).

Counting People Within Zones

It is possible to extend this use case with the *Zone Counting* use case. For example, if a CCTV footage shows the entrance of a mall as well as a road, and we are only interested to apply people counting to the mall entrance, we could split the video into 2 different zones and only count the people within the chosen zone. An example of how this can be done is given in the *Tracking People within a Zone* tutorial.

7.2.5 Zone Counting

Overview

Zone counting creates different zones within a single image and counts the number of objects within each zone separately. This is useful in many applications, such as counting vehicles travelling on one side of a road, or counting the shoppers entering a mall.

See also:

To only count objects within a single zone and ignore all other objects, see the *Tracking People within a Zone* tutorial.

Zone counting is done by counting the number of objects detected by the object detection models that fall within the specified zones. For example, we can count the number of people in the blue and red zones, as shown in the GIF above. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file zone_counting.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/zone_counting.yml>
```

How It Works

There are three main components to obtain the zone counts:

1. The detections from the object detection model, which are the bounding boxes,
2. The bottom midpoint of the bounding boxes, derived from the bounding boxes, and
3. The zones, which can be set in the `dabble.zone_count` configurable parameters.

1. Object Detection

We use an open source object detection model known as YOLOv4 and its smaller and faster variant known as YOLOv4-tiny to identify the bounding boxes of objects we want to detect. This allows the application to identify where objects are located within the video feed. The location is returned as a pair of *x, y coordinates* in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each object detected. For more information on how to adjust the `yolo` node, check out its *configurable parameters*.

2. Bounding Box to Bottom Midpoint

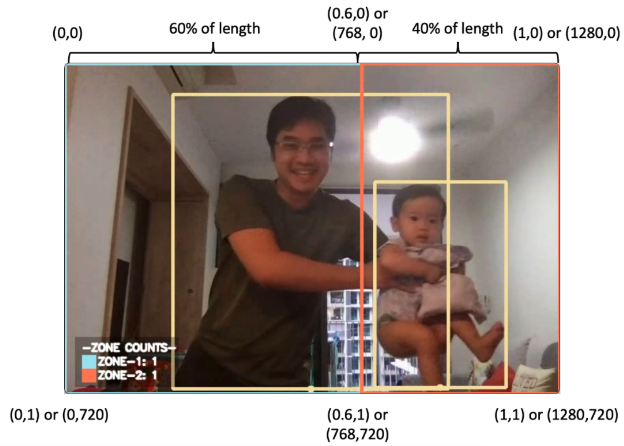
Given the top left (x_1, y_1) and bottom right (x_2, y_2) coordinates of each bounding box, the bottom midpoint (x_{bm}, y_{bm}) can be computed by taking lowest *y* coordinate $y_{bm} = y_2$, and the midpoint of the *x* coordinates $x_{bm} = (x1 + x2)/2$.

We found that using the bottom midpoint is the most efficient way to tell if something is in a specified zone. We attribute this to the use of the top-down or angled camera footages, which are commonly found in the use cases. The bottom midpoints of the bounding boxes usually correspond to the locations of the objects in these footages.

3. Zones

Zones are created by specifying the *x, y* coordinates of all the corner points that form the area of the zone **in a clockwise direction**. The coordinates can be in either fractions of the resolution or pixels. As an example, blue zone in the *zone counting GIF* was created using the following zone:

```
[[0, 0], [0.6, 0], [0.6, 1], [0, 1]]
```



Given a resolution of 1280 by 720, these correspond to the top-left of the image, 60% of the length at the top of the image, 60% of the length at the bottom of the image, and the bottom-left of the image. These points form the rectangular blue zone in a clockwise direction.

Note that because the *x*, *y* coordinates are fractions of the image resolution, the resolution config for `dabble.zone_count` needs to be set correctly.

For finer control over the exact coordinates, the pixel coordinates can be used instead. Using the same example, the blue zone can be created using the following zone configuration:

```
[[0, 0], [768, 0], [768, 720], [0, 720]]
```

When using pixel coordinates, the resolution is not needed. However, users should check to ensure that the pixel coordinates given fall within the image resolution so that the zone will work as intended.

Elaboration for this adjustment can be found the *"4. Adjusting Nodes"* section.

Note: Zones do not have to be rectangular in shape. They can be of any polygonal shape, dictated by the number and position of the *x*, *y* coordinates set in a zone.

4. Zone Counts

Given the bottom midpoints of all detected objects, we check if the points fall within the areas of the specified zones. If it falls inside any zone, an object count is added for that specific zone. This continues until all objects detected are accounted for, which gives the final count of objects in each specified zone.

Nodes Used

These are the nodes used in the earlier demo (also in zone_counting.yml):

```
nodes:
- input.visual:
    source: 0
- model.yolo:
    detect: ["person"]
- dabble.bbox_to_btm_midpoint
- dabble.zone_count:
    resolution: [1280, 720] # Adjust this to your camera's input resolution
    zones: [
```

(continues on next page)

(continued from previous page)

```
    [[0, 0], [0.6, 0], [0.6, 1], [0, 1]],
    [[0.6, 0], [1, 0], [1, 1], [0.6, 1]]
    ]
- draw.bbox
- draw.btm_midpoint
- draw.zones
- draw.legend:
    show: ["zone_count"]
- output.screen
```

1. Object Detection Node

By default, the node uses the YOLOv4-tiny model for object detection, set to detect people. Please take a look at the *benchmarks* of object detection models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Bottom Midpoint Node

The bottom midpoint node is called by including `dabble.bbox_to_btm_midpoint` in the pipeline config declaration. This outputs the bottom midpoints of all detected bounding boxes. The node has no configurable parameters.

3. Zone Counting Node

The zone counting node is called by including `dabble.zone_count` in the run config declaration. This uses the bottom midpoints of all detected bounding boxes and outputs the number of object counts in each specified zone. The node configurable parameters can be found below.

4. Adjusting Nodes

The zone counting detections depend on the configuration set in the object detection models, such as the type of object to detect, etc. For the object detection model used in this demo, please see the `yolo` node *documentation* for adjustable behaviors that can influence the result of the zone counting node.

With regards to the zone counting node, some common node behaviors that you might need to adjust are:

- `resolution`: If you are planning to use fractions to set the coordinates for the area of the zone, the resolution should be set to the image/video/livestream resolution used.
- `zones`: Used to specify the different zones which you would like to set. The coordinates for each zone are given in a list in a clockwise order. See the *Nodes Used* section on how to properly configure multiple zones.

For more adjustable node behaviors not listed here, check out the *API Documentation*.

Zone Counting	*Crowd Counting*
Object Counting (Over Time)	*People Counting (Over Time)*
Object Counting (Present)	

7.3 COVID-19 Prevention and Control

7.3.1 Face Mask Detection

Overview

Wearing of face masks in public places can help prevent the spread of COVID-19 and other infectious diseases. AI Singapore has developed a solution that checks whether or not a person is wearing a face mask. This can be used in places such as malls or shops to ensure that visitors adhere to the guidelines.

We have trained a custom YOLOv4 model to detect whether or not a person is wearing a face mask. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file face_mask_detection.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/face_mask_detection.yml>
```

How It Works

The main component is the detection of face mask using the custom YOLOv4 model.

Face Mask Detection

We use an open source object detection model known as YOLOv4 and its smaller and faster variant known as YOLOv4-tiny to identify the bounding boxes of human faces with and without face masks. This allows the application to identify the locations of faces and their corresponding classes (no_mask = 0 or mask = 1) in a video feed. Each of these locations are represented as a pair of *x, y* coordinates in the form $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) is the top-left corner of the bounding box, and (x_2, y_2) is the bottom right. These are used to form the bounding box of each human face detected.

The `model.yolo_face` node detects human faces with and without face masks using the YOLOv4-tiny model by default. The classes are differentiated by the labels and the colors of the bounding boxes when multiple faces are detected. For more information on how to adjust the `yolo_face` node, check out its *configurable parameters*.

Nodes Used

These are the nodes used in the earlier demo (also in face_mask_detection.yml):

```
nodes:
- input.visual:
    source: 0
- model.yolo_face
- draw.bbox:
    show_labels: true
- output.screen
```

1. Face Mask Detection Node

The `model.yolo_face` node is used for face detection and to classify if the face is masked or unmasked. To simply detect faces without needing to classify if the face is masked, you can also consider the `model.mtcnn` node.

2. Adjusting Nodes

Some common node behaviors that you might want to adjust are:

- `model_type`: This specifies the variant of YOLOv4 to be used. By default, the *v4tiny* model is used, but for better accuracy, you may want to try the *v4* model.
- `detect`: This specifies the class to be detected where no_mask = 0 and mask = 1. By default, the model detects faces with and without face masks (default = [0, 1]).
- `score_threshold`: This specifies the threshold value. Bounding boxes with confidence score lower than the threshold are discarded. You may want to lower the threshold value to increase the number of detections.

7.3.2 Group Size Checking

Overview

As part of COVID-19 measures, the Singapore Government has set restrictions on the group sizes of social gatherings. AI Singapore has developed a vision-based group size checker that checks if the group size limit has been violated. This can be used in many places, such as in malls to ensure that visitors adhere to guidelines, or in workplaces to ensure employees' safety.

To check if individuals belong to a group, we check if the physical distance between them is close. The most accurate way to measure distance is to use a 3D sensor with depth perception, such as a RGB-D camera or a LiDAR. However, most cameras such as CCTVs and IP cameras usually only produce 2D videos. We developed heuristics that are able to give an approximate measure of physical distance from 2D videos, addressing this limitation. This is further elaborated in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file group_size_checking.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/group_size_checking.yml>
```

How It Works

There are three main components to obtain the distance between individuals:

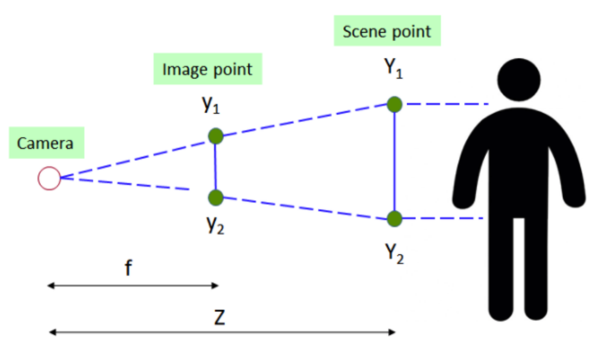
1. Human pose estimation using AI,
2. Depth and distance approximation, and
3. Linking individuals to groups.

1. Human Pose Estimation

We use an open source human pose estimation model known as PoseNet to identify key human skeletal points. This allows the application to identify where individuals are located within the video feed. The coordinates of the various skeletal points will then be used to determine the distance between individuals.

2. Depth and Distance Approximation

To measure the distance between individuals, we have to estimate the 3D world coordinates from the keypoints in 2D coordinates. To achieve this, we compute the depth Z from the x, y coordinates using the relationship below:



Using the similar triangle rule, we are able to compute Z.

$$\frac{y_1 - y_2}{Y_1 - Y_2} = \frac{f}{Z}$$

where:

- Z = depth or distance of scene point from camera
- f = focal length of camera

- y = y position of image point
- Y = y position of scene point

$Y_1 - Y_2$ is a reference or "ground truth length" that is required to obtain the depth. After numerous experiments, it was decided that the optimal reference length would be the average height of a human torso (height from human hip to center of face). Width was not used as this value has high variance due to the different body angles of an individual while facing the camera.

3. Linking Individuals to Groups

Once we have the 3D world coordinates of the individuals in the video, we can compare the distances between each pair of individuals. If they are close to each other, we assign them to the same group. This is a dynamic connectivity problem and we use the quick find algorithm to solve it.

Nodes Used

These are the nodes used in the earlier demo (also in group_size_checking.yml):

```
nodes:
- input.visual:
    source: 0
- model.posenet
- dabble.keypoints_to_3d_loc:
    focal_length: 1.14
    torso_factor: 0.9
- dabble.group_nearby_objs:
    obj_dist_threshold: 1.5
- dabble.check_large_groups:
    group_size_threshold: 2
- draw.poses
- draw.group_bbox_and_tag
- output.screen
```

1. Pose Estimation Model

By default, we are using the PoseNet model with a ResNet backbone for pose estimation. Please take a look at the *benchmarks* of pose estimation models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Adjusting Nodes

Some common node behaviors that you might need to adjust are:

- `focal_length` & `torso_factor`: We calibrated these settings using a Logitech c170 webcam, with 2 individuals of heights about 1.7m. We recommend running a few experiments on your setup and calibrate these accordingly.
- `obj_dist_threshold`: The maximum distance between 2 individuals, in meters, for them to be considered to be part of a group.
- `group_size_threshold`: The acceptable group size limit.

For more adjustable node behaviors not listed here, check out the *API Documentation*.

3. Using Object Detection (Optional)

It is possible to use *object detection models* instead of pose estimation. To do so, replace the model node accordingly, and replace the node `dabble.keypoints_to_3d_loc` with `dabble.bbox_to_3d_loc`. The reference or "ground truth length" in this case would be the average height of a human, multiplied by a small factor.

You might need to use this approach if running on a resource-limited device such as a Raspberry Pi. In this situation, you'll need to use the lightweight models; we find lightweight object detectors are generally better than lightweight pose estimation models in detecting humans.

The trade-off here is that the estimated distance between individuals will be less accurate. This is because for object detectors, the bounding box will be compared with the average height of a human, but the bounding box height decreases if the person is sitting down or bending over.

Using with Social Distancing

To combat COVID-19, individuals are encouraged to maintain physical distance from each other. We've developed a social distancing tool that checks if individuals are too close to each other.

The nodes for social distancing can be stacked with group size checker, to perform both at the same time. Check out the *Social Distancing use case* to find out which nodes are used.

7.3.3 Social Distancing

Overview

To support the fight against COVID-19, AI Singapore developed a solution to encourage individuals to maintain physical distance from each other. This can be used in many places, such as in malls to encourage social distancing in long queues, or in workplaces to ensure employees' well-being. An example of the latter is HP Inc., which collaborated with us to deploy this solution on edge devices in its manufacturing facility in Singapore.

The most accurate way to measure distance is to use a 3D sensor with depth perception, such as a RGB-D camera or a LiDAR. However, most cameras such as CCTVs and IP cameras usually only produce 2D videos. We developed heuristics that are able to give an approximate measure of physical distance from 2D videos, addressing this limitation. This is explained in the *How It Works* section.

Demo

To try our solution on your own computer, *install* and run PeekingDuck with the configuration file social_distancing.yml as shown:

Terminal Session

```
[~user] > peekingduck run --config_path <path/to/social_distancing.yml>
```

How It Works

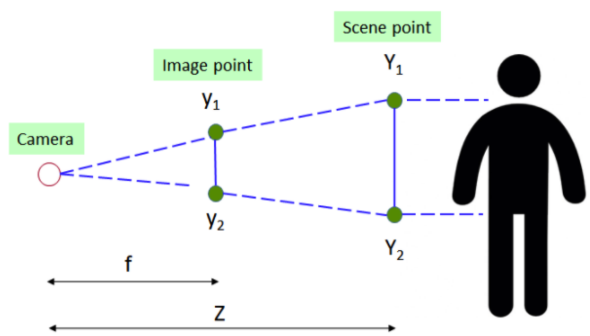
There are two main components to obtain the distance between individuals: #. Human pose estimation using AI, and #. Depth and distance approximation using heuristics.

1. Human Pose Estimation

We use an open source human pose estimation model known as PoseNet to identify key human skeletal points. This allows the application to identify where individuals are located within the video feed. The coordinates of the various skeletal points will then be used to determine the distance between individuals.

2. Depth and Distance Approximation

To measure the distance between individuals, we have to estimate the 3D world coordinates from the keypoints in 2D coordinates. To achieve this, we compute the depth Z from the *x, y* coordinates using the relationship below:



Using the similar triangle rule, we are able to compute Z.

$$\frac{y_1 - y_2}{Y_1 - Y_2} = \frac{f}{Z}$$

where:

- Z = depth or distance of scene point from camera
- f = focal length of camera
- y = *y* position of image point
- Y = *y* position of scene point

$Y_1 - Y_2$ is a reference or "ground truth length" that is required to obtain the depth. After numerous experiments, it was decided that the optimal reference length would be the average height of a human torso (height from human hip to center of face). Width was not used as this value has high variance due to the different body angles of an individual while facing the camera.

Once we have the 3D world coordinates of the individuals in the video, we can compare the distances between each pair of individuals and check if they are too close to each other.

Nodes Used

These are the nodes used in the earlier demo (also in social_distancing.yml):

```
nodes:
- input.visual:
    source: 0
- model.posenet
- dabble.keypoints_to_3d_loc:
    focal_length: 1.14
    torso_factor: 0.9
- dabble.check_nearby_objs:
    near_threshold: 1.5
    tag_msg: "TOO CLOSE!"
- draw.poses
- draw.tag:
    show: ["flags"]
- output.screen
```

1. Pose Estimation Model

By default, we are using the PoseNet model with a ResNet backbone for pose estimation. Please take a look at the *benchmarks* of pose estimation models that are included in PeekingDuck if you would like to use a different model or model type better suited to your use case.

2. Adjusting Nodes

Some common node behaviors that you might need to adjust are:

- `focal_length` & `torso_factor`: We calibrated these settings using a Logitech c170 webcam, with 2 individuals of heights about 1.7m. We recommend running a few experiments on your setup and calibrate these accordingly.
- `tag_msg`: The message to show when individuals are too close.
- `near_threshold`: The minimum acceptable distance between 2 individuals, in meters. For example, if the threshold is set at 1.5m, and 2 individuals are standing 2.0m apart, `tag_msg` doesn't show as they are standing further apart than the threshold. The larger this number, the stricter the social distancing.

For more adjustable node behaviors not listed here, check out the *API Documentation*.

3. Using Object Detection (Optional)

It is possible to use *object detection models* instead of pose estimation. To do so, replace the model node accordingly, and replace the `dabble.keypoints_to_3d_loc` node with `dabble.bbox_to_3d_loc`. The reference or "ground truth length" in this case would be the average height of a human, multiplied by a small factor.

You might need to use this approach if running on a resource-limited device such as a Raspberry Pi. In this situation, you'll need to use the lightweight models, and we find that lightweight object detectors are generally better than lightweight pose estimation models in detecting humans.

The trade-off here is that the estimated distance between individuals will be less accurate. This is because for object detectors, the bounding box will be compared with the average height of a human, but the bounding box height decreases if the person is sitting down or bending over.

Using with Group Size Checker

As part of COVID-19 measures, the Singapore Government has set restrictions on the group sizes of social gatherings. We've developed a group size checker that checks if the group size limit has been violated.

The nodes for group size checker can be stacked with social distancing, to perform both at the same time. Check out the *Group Size Checking use case* to find out which nodes are used.

Social Distancing	*Group Size Checking*
Face Mask Detection	

CHAPTER

EIGHT

FAQ AND TROUBLESHOOTING

8.1 How can I post-process and visualize model outputs?

The common output of all `model` nodes is *bboxes*. *bboxes* can be used for subsequent actions like counting (`dabble.bbox_count`), drawing (`draw.bbox`), tagging (`draw.tag`), etc. You can also create custom nodes which take *bboxes* as an input to visualize your results.

8.2 How can I dynamically use all prior outputs as the input at run time?

Specifying "*all*" as the input allows the node to receive all prior outputs as the input. This is used by nodes such as `draw.legend` and `output.csv_writer`.

8.3 How do I debug custom nodes?

You can add code in custom nodes to print the contents of their inputs. For more info, please see the tutorial on *debugging*.

8.4 Why does `input.visual` progress stop before 100%?

`input.visual` provides progress information if it is able to get a total frame count for the input. This number is obtained using `opencv`'s `CV_CAP_PROP_FRAME_COUNT` API, which attempts to estimate the total frame count using the input media's metadata duration and FPS. However, the total frame count is only an estimate. It is not guaranteed to be accurate because it is affected by potential errors, such as frame corruption, video decoder failure, inaccurate FPS, and rounding errors.

8.5 Why does the output screen flash briefly and disappear on my second run?

If you are running PeekingDuck on the Windows Subsystem for Linux (WSL), this erroneous behavior may be caused by a WSL bug where the key buffer is not flushed. Please refer to this GitHub issue for more details.

CHAPTER

NINE

GLOSSARY

The following are built-in data types recognized by PeekingDuck nodes. Users can define custom data types when working with custom nodes.

(input) `all` (`Any`)
: This data type contains all the outputs from preceding nodes, granting a large degree of flexibility to nodes that receive it. Examples of such nodes include `draw.legend`, `dabble.statistics`, and `output.csv_writer`.

`bboxes` (`numpy.ndarray`)
: A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`)
: A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`)
: A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

`btm_midpoint` (`List[Tuple[int, int]]`)
: A list of tuples each representing the (x, y) coordinates of the bottom middle of a bounding box for use in zone analytics. The order corresponds to *bboxes*.

`count` (`int`)
: An integer representing the number of counted objects.

`cum_avg` (`float`)
: Cumulative average of an attribute over time.

`cum_max` (`float | int`)
: Cumulative maximum of an attribute over time.

`cum_min` (`float | int`)
: Cumulative minimum of an attribute over time.

`density_map` (`numpy.ndarray`)
: A NumPy array of shape (H, W) representing the number of persons per pixel. H and W are the height and width of the input image, respectively. The sum of the array is the estimated total number of people.

`filename` (`str`)
: The filename of video/image being read.

`fps` (`float`)
: A float representing the Frames Per Second (FPS) when processing a live video stream or a recorded video.

img (**numpy.ndarray**)
A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

keypoints (**numpy.ndarray**)
A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by `-1`.

keypoint_conns (**numpy.ndarray**)
A NumPy array of shape $(N, D_n', 2, 2)$ containing the (x, y) coordinates of adjacent keypoint pairs where N is the number of detected poses, and D_n' is the number of valid keypoint pairs for the the n-th pose where both keypoints are detected.

keypoint_scores (**numpy.ndarray**)
A NumPy array of shape (N, K) containing the confidence scores of detected poses where N is the number of detected poses and K is the number of individual keypoints. The confidence score has a range of $[0, 1]$.

large_groups (**List[int]**)
A list of integers representing the group IDs of groups that have exceeded the size threshold.

masks (**numpy.ndarray**)
A NumPy array of shape (N, H, W) containing N detected binarized masks where H and W are the height and width of the masks. The order corresponds to *bbox_labels*.

(input) none
No inputs required.

(output) none
No outputs produced.

obj_3D_locs (**List[numpy.ndarray]**)
A list of N NumPy arrays representing the 3D coordinates (x, y, z) of an object associated with a detected bounding box.

obj_attrs (**Dict[str, Any]**)
A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes.

pipeline_end (**bool**)
A boolean that evaluates to `True` when the pipeline is completed. Suitable for operations that require the entire inference pipeline to be completed before running.

saved_video_fps (**float**)
FPS of the recorded video, upon filming.

zones (**List[List[Tuple[float, ...]]]**)
A nested list of Z zones. Each zone is described by 3 **or more** points which contains the (x, y) coordinates forming the boundary of a zone. The order corresponds to *zone_count*.

zone_count (**List[int]**)
A list of integers representing the count of a pre-selected object class (for example, "person") detected in each specified zone. The order corresponds to *zones*.

Deprecated since version 1.2.0: `obj_tags` (`List[str]`) is deprecated and now subsumed under *obj_attrs*. `dabble.check_nearby_objs` now accesses this attribute by using the `flags` key of *obj_attrs*. `draw.tag` has been refactored for more drawing flexibility by accepting *obj_attrs* as input.

Deprecated since version 1.2.0: `obj_groups` (`List[int]`) is deprecated and now subsumed under *obj_attrs*. Affected nodes (`dabble.group_nearby_objs`, `dabble.check_large_groups`, and `draw.group_bbox_and_tag`) now access this attribute by using the `groups` key of *obj_attrs*.

CHAPTER

TEN

API DOCUMENTATION

input	Reads data from a given input.
augment	Performs image processing.
model	Deep learning model nodes for computer vision.
dabble	Algorithms that perform calculations/heuristics on the outputs of `model`.
draw	Draws results/outputs to an image.
output	Writes/displays the outputs of the pipeline.

10.1 input

Description

Reads data from a given input.

Deprecated since version 1.2.0: `input.live` and `input.recorded` are deprecated. They have been replaced by the *input.visual* node.

Modules

input.visual	Reads inputs from multiple visual sources - image or video file on local storage - folder of images or videos - online cloud source - CCTV or webcam live feed

10.1.1 input.visual

Description

Reads inputs from multiple visual sources - image or video file on local storage - folder of images or videos - online cloud source - CCTV or webcam live feed

class Node(*config=None*, *node_path=''*, *pkd_base_dir=None*, ***kwargs*)

Receives visual sources as inputs.

Inputs

`none`: No inputs required.

Outputs

img (numpy.ndarray): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

filename (str): The filename of video/image being read.

pipeline_end (bool): A boolean that evaluates to True when the pipeline is completed. Suitable for operations that require the entire inference pipeline to be completed before running.

saved_video_fps (float): FPS of the recorded video, upon filming.

Configs

- **filename** (str) – **default = "video.mp4"**. If source is a live stream/webcam, filename defines the name of the MP4 file if the media is exported. If source is a local file or directory of files, then filename is the current file being processed, and the value specified here is overridden.
- **mirror_image** (bool) – **default = False**. Flag to set extracted image frame as mirror image of input stream.
- **resize** (Dict[str, Any]) – **default = { do_resizing: False, width: 1280, height: 720 }** Dimension of extracted image frame.
- **source** (Union[int, str]) – **default = https://storage.googleapis.com/peekingduck/videos/wave.mp4**. Input source can be: - filename : local image or video file - directory name : all media files will be processed - http URL for online cloud source : http[s]://... - rtsp URL for CCTV : rtsp://... - 0 for webcam live feed Refer to OpenCV documentation for more technical information.
- **frames_log_freq** (int) – **default = 100**.[1] Logs frequency of frames passed in CLI
- **saved_video_fps** (int) – **default = 10**.[Page 118, 1] This is used by *output.media_writer* to set the FPS of the output file and its behavior is determined by the type of input source. If source is an image file, this value is ignored as it is not applicable. If source is a video file, this value will be overridden by the actual FPS of the video. If source is a live stream/webcam, this value is used as the FPS of the output file. It is recommended to set this to the actual FPS obtained on the machine running PeekingDuck (using *dabble.fps*).
- **threading** (bool) – **default = False**.[1] Flag to enable threading when reading frames from camera / live stream. The FPS can increase up to 30%. There is no need to enable threading if reading from a video file.
- **buffering** (bool) – **default = False**.[1] Boolean to indicate if threaded class should buffer image frames. If reading from a video file and threading is True, then buffering should also be True to avoid "lost frames": which happens when the video file is read faster than it is processed. One side effect of setting threading=True, buffering=True for a live stream/webcam is the onscreen video could appear to be playing in slow-mo.

Technotes:

The following table summarizes the combinations of threading and buffering:

Threading		False	True	
Buffering		False/True	False	True
Sources	Image file	Ok	Ok	Ok
	Video file	Ok	!	Ok
	Webcam, http/rtsp stream	Ok	+	!!

[1] advanced configuration

Table Legend:

Ok : normal behavior + : potentially faster FPS ! : lost frames if source is faster than PeekingDuck !! : "slow-mo" video, potential out-of-memory error due to buffer overflow if source is faster than PeekingDuck

Note: If threading=False, then the secondary parameter buffering is ignored regardless if it is set to True/False.

Here is a video to illustrate the differences between a normal video vs a "slow-mo" video using a 30 FPS webcam: the video on the right appears to be playing in slow motion compared to the normal video on the left. This happens as both threading and buffering are set to True, and the threaded *input.visual* reads the webcam at almost 60 FPS. Since the hardware is physically limited at 30 FPS, this means every frame gets duplicated, resulting in each frame being processed and shown twice, thus "stretching out" the video.

10.2 augment

Description

Performs image processing. This can be done before or after the model.

Modules

augment.brightness	Adjusts the brightness of an image.
augment.contrast	Adjusts the contrast of an image.
augment.undistort	Removes distortion from a wide-angle camera image.

10.2.1 augment.brightness

Description

Adjusts the brightness of an image.

class Node(*config=None*, ***kwargs*)

Adjusts the brightness of an image, by adding a bias/beta parameter.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

beta (`int`) – **[-100, 100], default = 0**. Increasing the value of beta increases image brightness, and vice versa.

10.2.2 augment.contrast

Description

Adjusts the contrast of an image.

class Node(*config=None*, ***kwargs*)

Adjusts the contrast of an image, by multiplying with a gain/alpha parameter.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

alpha (`float`) – **[0.0, 3.0], default = 1.0**. Increasing the value of alpha increases the contrast.

10.2.3 augment.undistort

Description

Removes distortion from a wide-angle camera image.

class Node(*config=None*, ***kwargs*)

Undistorts an image by removing radial and tangential distortion. This may help to improve the performance of certain models.

Before using this node for the first time, please follow the tutorial in *dabble.camera_calibration* to calculate the camera coefficients of the camera you are using, and ensure that the file_path that the coefficients are stored in is the same as the one specified in the configs.

The images below show an example of an image before and after undistortion. Note that after undistortion, the shape of the image will change and the FOV will be reduced slightly.

Fig. 1: Before undistortion (left) and after undistortion (right)

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

file_path (`str`) – **default = "PeekingDuck/data/camera_calibration_coeffs.yml"**. Path of the YML file containing calculated camera coefficients.

10.3 model

Description

Deep learning model nodes for computer vision.

Modules

model.csrnet	Congested Scene Recognition network: Dilated convolutional neural networks for understanding the highly congested scenes.
model.efficientdet	Scalable and efficient object detection.
model.fairmot	Human detection and tracking model that balances the importance between detection and re-ID tasks.
model.hrnet	High-Resolution Network: Deep high-resolution representation learning for human pose estimation.
model.jde	Joint Detection and Embedding model for human detection and tracking.
model.mask_rcnn	Instance segmentation model for generating high-quality masks.
model.movenet	Fast Pose Estimation model.
model.mtcnn	Multi-task Cascaded Convolutional Networks for face detection.
model.posenet	Fast Pose Estimation model.
model.yolact_edge	Instance segmentation model for real-time inference
model.yolo	One-stage Object Detection model.
model.yolo_face	Fast face detection model that can distinguish between masked and unmasked faces.
model.yolo_license_plate	License Plate Detection model.
model.yolox	High performance anchor-free YOLO object detection model.

10.3.1 model.csrnet

Description

Congested Scene Recognition network: Dilated convolutional neural networks for understanding the highly congested scenes.

class Node(*config=None*, ***kwargs*)

Initializes and uses CSRNet model to predict the density map and crowd count.

The csrnet node is capable of predicting the number of people in dense and sparse crowds. The dense and sparse crowd models were trained using data from ShanghaiTech Part A and ShanghaiTech Part B respectively. As the models were trained to recognize congested scenes, the estimates are less accurate if the number of people are low (e.g. less than 10).

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`density_map` (`numpy.ndarray`): A NumPy array of shape (H, W) representing the number of persons per pixel. H and W are the height and width of the input image, respectively. The sum of the array is the estimated total number of people.

`count` (`int`): An integer representing the number of counted objects.

Configs

- **model_type** (`str`) – **{"dense", "sparse"}, default="sparse"**. Defines the type of CSRNet model to be used. The node uses the sparse crowd model by default and can be changed to using the dense crowd model. As a rule of thumb, the dense crowd model should be used if the people in a given image or video frame are packed shoulder to shoulder, e.g., stadiums.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **width** (`int`) – **default = 640**. By default, the width of an image will be resized to 640 for inference. The height of the image will be resized proportionally to preserve its aspect ratio. In general, decreasing the width of an image will improve inference speed. However, this might impact the accuracy of the model.

References

CSRNet: Dilated Convolutional Neural Networks for Understanding the Highly Congested Scenes: https://arxiv.org/pdf/1802.10062.pdf

Model weights trained by https://github.com/Neerajj9/CSRNet-keras

Inference code adapted from https://github.com/Neerajj9/CSRNet-keras

10.3.2 model.efficientdet

Description

Scalable and efficient object detection.

`class Node`(*config=None*, ***kwargs*)

Initializes an EfficientDet model to detect bounding boxes from an image.

The EfficientDet node is capable of detecting objects from 80 categories. The table of categories can be found *here*.

EfficientDet node has five levels of compound coefficient (0 - 4). A higher compound coefficient will scale up all dimensions of the backbone network width, depth, input resolution, feature network, and box/class prediction at the same time, which results in better performance but slower inference time. The default compound coefficient is 0 and can be changed to other values.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

Configs

- **model_type** (`int`) – **{0, 1, 2, 3, 4}, default = 0**. Defines the compound coefficient for EfficientDet.
- **score_threshold** (`float`) – **[0, 1], default = 0.3**. Bounding boxes with confidence score below the threshold will be discarded.
- **detect** (`List[Union[int, str]]`) – **default = [0]**. List of object class names or IDs to be detected. To detect all classes, refer to the *tech note*.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.

References

EfficientDet: Scalable and Efficient Object Detection: https://arxiv.org/abs/1911.09070

Code adapted from https://github.com/xuannianz/EfficientDet.

10.3.3 model.fairmot

Description

Human detection and tracking model that balances the importance between detection and re-ID tasks.

class Node(*config=None*, ***kwargs*)

Initializes and uses FairMOT tracking model to detect and track people from the supplied image frame.

FairMOT is based on the anchor-free object detector CenterNet with modifications to balance the importance between detection and re-identification tasks in an object tracker.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

obj_attrs (Dict[str, Any]): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `model.fairmot` produces the ids attribute which contains the tracking IDs of the detections.

Configs

- **weights_parent_dir** (Optional[str]) – **default = null**. Change the parent directory where weights will be stored by replacing null with an absolute path to the desired directory.
- **score_threshold** (float) – **default = 0.5**. Object confidence score threshold.
- **K** (int) – **default = 500**. Maximum number of objects output during the object detection stage.
- **min_box_area** (int) – **default = 100**. Minimum value for area of detected bounding box. Calculated by width * height.
- **track_buffer** (int) – **default = 30**. Threshold to remove track if track is lost for more frames than value.
- **input_size** (List[int]) – **default = [864, 480]**. Size (width, height) of the input image to the model. Raw video/image frames will be resized to the input_size before they are fed to the model.

References

FairMOT: On the Fairness of Detection and Re-Identification in Multiple Object Tracking https://arxiv.org/abs/2004.01888

Model weights trained by: https://github.com/ifzhang/FairMOT

10.3.4 model.hrnet

Description

High-Resolution Network: Deep high-resolution representation learning for human pose estimation. Requires an object detector.

class Node(*config=None*, ***kwargs*)

Initializes and uses HRNet model to infer poses from detected bboxes. Note that HRNet must be used in conjunction with an object detector applied prior.

The HRNet applied to human pose estimation uses the representation head, called HRNetV1.

The HRNet node is capable of detecting single human figures simultaneously per inference, with 17 keypoints estimated for each detected human figure. The keypoint indices table can be found *here*.

Inputs

img (numpy.ndarray): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

bboxes (numpy.ndarray): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

keypoints (numpy.ndarray): A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by -1.

keypoint_scores (numpy.ndarray): A NumPy array of shape (N, K) containing the confidence scores of detected poses where N is the number of detected poses and K is the number of individual keypoints. The confidence score has a range of $[0, 1]$.

keypoint_conns (numpy.ndarray): A NumPy array of shape $(N, D'_n, 2, 2)$ containing the (x, y) coordinates of adjacent keypoint pairs where N is the number of detected poses, and D'_n is the number of valid keypoint pairs for the the n-th pose where both keypoints are detected.

Configs

- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **resolution** (`Dict[str, int]`) – **default = { height: 192, width: 256 }**. Resolution of input array to HRNet model.
- **score_threshold** (`float`) – **[0, 1], default = 0.1**. Threshold to determine if detection should be returned

References

Deep High-Resolution Representation Learning for Visual Recognition: https://arxiv.org/abs/1908.07919

10.3.5 model.jde

Description

Joint Detection and Embedding model for human detection and tracking.

class Node(*config*, ***kwargs*)

Initializes and uses JDE tracking model to detect and track people from the supplied image frame.

JDE is a fast and high-performance multiple-object tracker that learns the object detection task and appearance embedding task simultaneously in a shared neural network.

Inputs

img (numpy.ndarray): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

bboxes (numpy.ndarray): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

bbox_labels (numpy.ndarray): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

bbox_scores (numpy.ndarray): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `model.fairmot` produces the `ids` attribute which contains the tracking IDs of the detections.

Configs

- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **iou_threshold** (`float`) – **default = 0.5**. Threshold value for Intersecton-over-Union of detections.
- **nms_threshold** (`float`) – **default = 0.4**. Threshold values for non-max suppression.
- **score_threshold** (`float`) – **default = 0.5**. Object confidence score threshold.
- **min_box_area** (`int`) – **default = 200**. Minimum value for area of detected bounding box. Calculated by $width \times height$.
- **track_buffer** (`int`) – **default = 30**. Threshold to remove track if track is lost for more frames than value.

References

Towards Real-Time Multi-Object Tracking: https://arxiv.org/abs/1909.12605v2

Model weights trained by: https://github.com/Zhongdao/Towards-Realtime-MOT

10.3.6 model.mask_rcnn

Description

Instance segmentation model for generating high-quality masks.

class Node(*config=None*, ***kwargs*)

Initializes and uses Mask R-CNN to infer from an image frame.

The Mask-RCNN node is capable detecting objects and their respective masks from 80 categories. The table of object categories can be found *here*. The `"r50-fpn"` backbone is used by default, and the `"r101-fpn"` for the ResNet 101 backbone variant can also be chosen.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

masks (numpy.ndarray): A NumPy array of shape (N, H, W) containing N detected binarized masks where H and W are the height and width of the masks. The order corresponds to *bbox_labels*.

Configs

- **model_type** (str) – **{"r50-fpn", "r101-fpn"}, default = "r50-fpn"**. Defines the type of backbones to be used.
- **weights_parent_dir** (Optional[str]) – **default = null**. Change the parent directory where weights will be stored by replacing null with an absolute path to the desired directory.
- **min_size** (int) – **default = 800**. Minimum size of the image to be rescaled before feeding it to the backbone.
- **max_size** (int) – **default = 1333**. Maximum size of the image to be rescaled before feeding it to the backbone.
- **detect** (List[Union[int, string]]) – **default = [0]**. List of object class names or IDs to be detected. To detect all classes, refer to the *tech note*.
- **max_num_detections** – (int): **default = 100**. Maximum number of detections per image, for all classes.
- **iou_threshold** (float) – **[0, 1], default = 0.5**. Overlapping bounding boxes with Intersection over Union (IoU) above the threshold will be discarded.
- **score_threshold** (float) – **[0, 1], default = 0.5**. Bounding boxes with classification score below the threshold will be discarded.
- **mask_threshold** (float) – **[0, 1], default = 0.5**. The confidence threshold for binarizing the masks' pixel values; determines whether an object is detected at a particular pixel.

References

Mask R-CNN: A conceptually simple, flexible, and general framework for object instance segmentation.: https://arxiv.org/abs/1703.06870

Inference code adapted from: https://pytorch.org/vision/0.11/_modules/torchvision/models/detection/mask_rcnn.html

The weights for Mask-RCNN Model with ResNet50 FPN backbone were adapted from: https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-bf2d0c1e.pth

10.3.7 model.movenet

Description

Fast Pose Estimation model.

class **Node**(*config=None*, ***kwargs*)

MoveNet node that initializes a MoveNet model to detect human poses from an image.

The MoveNet node is capable of detecting up to 6 human figures for multipose lightning and single person for singlepose lightning/thunder. If there are more than 6 persons in the image, multipose lightning will only detect 6. This also applies to singlepose models, where only 1 person will be detected in a multi persons image, do take note that detection performance will suffer when using singlepose models on multi persons images. 17 keypoints are estimated and the keypoint indices table can be found *here*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`keypoints` (`numpy.ndarray`): A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by `-1`.

`keypoint_scores` (`numpy.ndarray`): A NumPy array of shape (N, K) containing the confidence scores of detected poses where N is the number of detected poses and K is the number of individual keypoints. The confidence score has a range of $[0, 1]$.

`keypoint_conns` (`numpy.ndarray`): A NumPy array of shape $(N, D_n', 2, 2)$ containing the (x, y) coordinates of adjacent keypoint pairs where N is the number of detected poses, and D_n' is the number of valid keypoint pairs for the the n-th pose where both keypoints are detected.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

Configs

- **model_format** (`str`) – **{"tensorflow", "tensorrt"}, default="tensorflow"** Defines the weights format of the model.
- **model_type** (`str`) – **{" singlepose_lightning", "singlepose_thunder", "multipose_lightning" }, default="multipose_lightning"** Defines the detection model for MoveNet either single or multi pose. Lightning is smaller and faster but less accurate than Thunder version.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **bbox_score_threshold** (`float`) – **[0,1], default = 0.2** Detected bounding box confidence score threshold, only boxes above threshold will be kept in the output.
- **keypoint_score_threshold** (`float`) – **[0,1], default = 0.3** Detected keypoints confidence score threshold, only keypoints above threshold will be kept in output.

10.3.8 model.mtcnn

Description

Multi-task Cascaded Convolutional Networks for face detection. Works best with unmasked faces.

class Node(*config=None*, ***kwargs*)

Initializes and uses the MTCNN model to infer bboxes from an image frame.

The MTCNN node is a single-class model capable of detecting human faces. To a certain extent, it is also capable of detecting bounding boxes around faces with face masks (e.g. surgical masks).

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

Configs

- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **min_size** (`int`) – **default = 40**. Minimum height and width of face in pixels to be detected.
- **scale_factor** (`float`) – **[0, 1], default = 0.709**. Scale factor to create the image pyramid. A larger scale factor produces more accurate detections at the expense of inference speed.
- **network_thresholds** (`List[float]`) – **[0, 1], default = [0.6, 0.7, 0.7]**. Threshold values for the Proposal Network (P-Net), Refine Network (R-Net) and Output Network (O-Net) in the MTCNN model.

  Calibration is performed at each stage in which bounding boxes with confidence scores less than the specified threshold are discarded.
- **score_threshold** (`float`) – **[0, 1], default = 0.7**. Bounding boxes with confidence scores less than the specified threshold in the final output are discarded.

References

Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks: https://arxiv.org/ftp/arxiv/papers/1604/1604.02878.pdf

Model weights trained by https://github.com/blaueck/tf-mtcnn

Changed in version 1.2.0: `mtcnn_min_size` is renamed to `min_size`. `mtcnn_factor` is renamed to `scale_factor`. `mtcnn_thresholds` is renamed to `network_thresholds`. `mtcnn_score` is renamed to `score_threshold`.

10.3.9 model.posenet

Description

Fast Pose Estimation model.

`class Node`(*config=None*, ***kwargs*)

Initializes a PoseNet model to detect human poses from an image.

The PoseNet node is capable of detecting multiple human figures simultaneously per inference and for each detected human figure, 17 keypoints are estimated. The keypoint indices table can be found *here*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`keypoints` (`numpy.ndarray`): A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by `-1`.

`keypoint_scores` (`numpy.ndarray`): A NumPy array of shape (N, K) containing the confidence scores of detected poses where N is the number of detected poses and K is the number of individual keypoints. The confidence score has a range of $[0, 1]$.

`keypoint_conns` (`numpy.ndarray`): A NumPy array of shape $(N, D'_n, 2, 2)$ containing the (x, y) coordinates of adjacent keypoint pairs where N is the number of detected poses, and D'_n is the number of valid keypoint pairs for the the n-th pose where both keypoints are detected.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

Configs

- **model_type** (`Union[str, int]`) – **{"resnet", 50, 75, 100}, default="resnet"**. Defines the backbone model for PoseNet.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **resolution** (`Dict`) – **default = { height: 225, width: 225 }**. Resolution of input array to PoseNet model.
- **max_pose_detection** (`int`) – **default = 10**. Maximum number of poses to be detected.
- **score_threshold** (`float`) – **[0, 1], default = 0.4**. Detected keypoints confidence score threshold, only keypoints above threshold will be kept in output.

References

PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model: https://arxiv.org/abs/1803.08225

Code adapted from https://github.com/rwightman/posenet-python

10.3.10 model.yolact_edge

Description

Instance segmentation model for real-time inference

class Node(*config=None*, ***kwargs*)

Initializes and uses YolactEdge to infer from an image frame

The YolactEdge node is capable of detecting objects from 80 categories. The table of object categories can be found *here*.

Inputs

img (numpy.ndarray): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

bboxes (numpy.ndarray): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

bbox_labels (numpy.ndarray): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

bbox_scores (numpy.ndarray): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

masks (numpy.ndarray): A NumPy array of shape (N, H, W) containing N detected binarized masks where H and W are the height and width of the masks. The order corresponds to *bbox_labels*.

Configs

- **model_type** (str) – (str): **{"r101-fpn", "r50-fpn", "mobilenetv2"}, default="r50-fpn"**.
- **weights_parent_dir** (Optional[str]) – **default = null**. Change the parent directory where weights will be stored by replacing null with an absolute path to the desired directory.
- **input_size** (int) – **default = 550**. Input image resolution of the YolactEdge model.
- **detect** (List[Union[int, string]]) – **default=[0]**. List of object class names or IDs to be detected. To detect all classes, refer to the *tech note*.
- **max_num_detections** – (int): **default=100**. Maximum number of detections per image, for all classes.
- **iou_threshold** (float) – **[0, 1], default = 0.5**. Overlapping bounding boxes with Intersection over Union (IoU) above the threshold will be discarded.
- **score_threshold** (float) – **[0, 1], default = 0.2**. Bounding boxes with confidence score (product of objectness score and classification score) below the threshold will be discarded.

References

YolactEdge: Real-time Instance Segmentation on the Edge https://arxiv.org/abs/2012.12259

Inference code and model weights: https://github.com/haotian-liu/yolact_edge

10.3.11 model.yolo

Description

One-stage Object Detection model.

class Node(*config=None*, ***kwargs*)

Initializes and uses YOLO model to infer bboxes from image frame.

The yolo node is capable of detecting objects from 80 categories. It uses YOLOv4-tiny by default and can be changed to using YOLOv4. The table of categories can be found *here*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

Configs

- **model_type** (`str`) – **{"v4", "v4tiny"}, default="v4tiny"**. Defines the type of YOLO model to be used.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **num_classes** (`int`) – **default = 80**. Maximum number of objects to be detected.
- **detect** (`List[Union[int, str]]`) – **default = [0]**. List of object class names or IDs to be detected. To detect all classes, refer to the *tech note*.
- **max_output_size_per_class** (`int`) – **default = 50**. Maximum number of detected instances for each class in an image.
- **max_total_size** (`int`) – **default = 50**. Maximum total number of detected instances in an image.
- **iou_threshold** (`float`) – **[0, 1], default = 0.5**. Overlapping bounding boxes above the specified IoU (Intersection over Union) threshold are discarded.
- **score_threshold** (`float`) – **[0, 1], default = 0.2**. Bounding box with confidence score less than the specified confidence score threshold is discarded.

References

YOLOv4: Optimal Speed and Accuracy of Object Detection: https://arxiv.org/pdf/2004.10934v1.pdf

Model weights trained by https://github.com/hunglc007/tensorflow-yolov4-tflite

Inference code adapted from https://github.com/zzh8829/yolov3-tf2

Changed in version 1.2.0: `yolo_iou_threshold` is renamed to `iou_threshold`. `yolo_score_threshold` is renamed to `score_threshold`.

10.3.12 model.yolo_face

Description

Fast face detection model that can distinguish between masked and unmasked faces.

class Node(*config=None*, ***kwargs*)

Initializes and uses the YOLO face detection model to infer bboxes from image frame.

The YOLO face model is a two class model capable of differentiating human faces with and without face masks.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

Configs

- **model_type** (`str`) – **{"v4", "v4tiny"}, default="v4tiny"**. Defines the type of YOLO model to be used.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **detect** (`List[int]`) – **default = [0, 1]**. List of object class IDs to be detected where *no_mask* is `0` and *mask* is `1`.
- **max_output_size_per_class** (`int`) – **default = 50**. Maximum number of detected instances for each class in an image.
- **max_total_size** (`int`) – **default = 50**. Maximum total number of detected instances in an image.
- **iou_threshold** (`float`) – **[0, 1], default = 0.1**. Overlapping bounding boxes above the specified IoU (Intersection over Union) threshold are discarded.
- **score_threshold** (`float`) – **[0, 1], default = 0.7**. Bounding box with confidence score less than the specified confidence score threshold is discarded.

References

YOLOv4: Optimal Speed and Accuracy of Object Detection: https://arxiv.org/pdf/2004.10934v1.pdf

Model weights trained using pretrained weights from Darknet: https://github.com/AlexeyAB/darknet

Changed in version 1.2.0: `yolo_iou_threshold` is renamed to `iou_threshold`. `yolo_score_threshold` is renamed to `score_threshold`.

10.3.13 model.yolo_license_plate

Description

License Plate Detection model.

class Node(*config=None*, ***kwargs*)

Initializes and uses YOLO model to infer bboxes from image frame.

This customized YOLO node is capable of detecting objects from a single class (License Plate). It uses YOLOv4 by default and can be changed to use YOLOv4-tiny if FPS is critical over accuracy.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

Configs

- **model_type** (`str`) – **{"v4", "v4tiny"}, default="v4"**. Defines the type of YOLO model to be used.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **iou_threshold** (`float`) – **[0, 1], default = 0.3**. Overlapping bounding boxes above the specified IoU (Intersection over Union) threshold are discarded.
- **score_threshold** (`float`) – **[0, 1], default = 0.1**. Bounding box with confidence score less than the specified confidence score threshold is discarded.

References

YOLOv4: Optimal Speed and Accuracy of Object Detection: https://arxiv.org/pdf/2004.10934v1.pdf

Model weights trained using pretrained weights from Darknet: https://github.com/AlexeyAB/darknet

Changed in version 1.2.0: `yolo_iou_threshold` is renamed to `iou_threshold`. `yolo_score_threshold` is renamed to `score_threshold`.

10.3.14 model.yolox

Description

High performance anchor-free YOLO object detection model.

class Node(*config=None*, ***kwargs*)

Initializes and uses YOLOX to infer from an image frame.

The YOLOX node is capable detecting objects from 80 categories. The table of object categories can be found *here*. The `"yolox-tiny"` model is used by default and can be changed to one of `("yolox-tiny", "yolox-s", "yolox-m", "yolox-l")`.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

`bbox_scores` (`numpy.ndarray`): A NumPy array of shape (N) containing confidence scores $[0, 1]$ of detected objects. The order corresponds to *bboxes* and *bbox_labels*.

Configs

- **model_format** (`str`) – **{"pytorch", "tensorrt"}, default="pytorch"** Defines the weights format of the model.
- **model_type** (`str`) – **{"yolox-tiny", "yolox-s", "yolox-m", "yolox-l"}, default="yolox-tiny"**. Defines the type of YOLOX model to be used.
- **weights_parent_dir** (`Optional[str]`) – **default = null**. Change the parent directory where weights will be stored by replacing `null` with an absolute path to the desired directory.
- **input_size** (`int`) – **default=416**. Input image resolution of the YOLOX model.
- **detect** (`List[Union[int, str]]`) – **default=[0]**. List of object class names or IDs to be detected. To detect all classes, refer to the *tech note*.
- **iou_threshold** (`float`) – **[0, 1], default = 0.45**. Overlapping bounding boxes with Intersection over Union (IoU) above the threshold will be discarded.
- **score_threshold** (`float`) – **[0, 1], default = 0.25**. Bounding boxes with confidence score (product of objectness score and classification score) below the threshold will be discarded.

- **agnostic_nms** (`bool`) – **default = True**. Flag to determine if class-agnostic NMS (`torchvision.ops.nms`) or class-aware NMS (`torchvision.ops.batched_nms`) should be used.
- **half** (`bool`) – **default = False**. Flag to determine if half-precision floating-point should be used for inference.
- **fuse** (`bool`) – **default = False**. Flag to determine if the convolution and batch normalization layers should be fused for inference.

References

YOLOX: Exceeding YOLO Series in 2021: https://arxiv.org/abs/2107.08430

Inference code and model weights: https://github.com/Megvii-BaseDetection/YOLOX

10.4 dabble

Description

Algorithms that perform calculations/heuristics on the outputs of `model`.

Modules

dabble.bbox_count	Counts the number of detected boxes.
dabble.bbox_to_3d_loc	Estimates the 3D coordinates of an object given a 2D bounding box.
dabble.bbox_to_btm_midpoint	Converts bounding boxes to a single point of reference.
dabble.camera_calibration	Calculates camera coefficients to be used to remove distortion from a wide-angle camera image.
dabble.check_large_groups	Checks if number of objects in a group exceeds a threshold.
dabble.check_nearby_objs	Checks if detected objects are near each other.
dabble.fps	Calculates the FPS of video.
dabble.group_nearby_objs	Assigns objects in close proximity to groups.
dabble.keypoints_to_3d_loc	Estimates the 3D coordinates of a person given 2D pose coordinates.
dabble.statistics	Calculates the cumulative average, minimum, and maximum of a single variable of interest over time.
dabble.tracking	Performs multiple object tracking for detected bboxes.
dabble.zone_count	Counts the number of detected objects within a boundary.

10.4.1 dabble.bbox_count

Description

Counts the number of detected boxes.

class Node(*config=None*, ***kwargs*)

Counts the total number of detected objects.

Inputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`count` (`int`): An integer representing the number of counted objects.

Configs

None.

10.4.2 dabble.bbox_to_3d_loc

Description

Estimates the 3D coordinates of an object given a 2D bounding box.

class Node(*config=None*, ***kwargs*)

Uses 2D bounding boxes information to estimate 3D location.

Inputs

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`obj_3D_locs` (`List[numpy.ndarray]`): A list of N NumPy arrays representing the 3D coordinates (x, y, z) of an object associated with a detected bounding box.

Configs

- **focal_length** (`float`) – **default = 1.14**. Approximate focal length of webcam used, in metres. Example on measuring focal length can be found here.
- **height_factor** (`float`) – **default = 2.5**. A factor used to estimate real-world distance from pixels, based on average human height in metres. The value varies across different camera set-ups, and calibration may be required. Please refer to the *Social Distancing use case* for more information.

10.4.3 dabble.bbox_to_btm_midpoint

Description

Converts bounding boxes to a single point of reference.

class Node(*config=None*, ***kwargs*)

Converts bounding boxes to a single point which is the bottom midpoint of the bounding box.

This node is primarily used for zone counting. The bottom midpoint is an unambiguous way of telling whether an object is in the zone specified, as the bottom midpoint usually corresponds to the point where the object is located.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`btm_midpoint` (`List[Tuple[int, int]]`): A list of tuples each representing the (x, y) coordinates of the bottom middle of a bounding box for use in zone analytics. The order corresponds to *bboxes*.

Configs

None.

10.4.4 dabble.camera_calibration

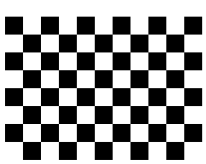
Description

Calculates camera coefficients to be used to remove distortion from a wide-angle camera image.

class Node(*config=None*, ***kwargs*)

Calculates camera coefficients for undistortion.

To calculate your camera, first download the following checkerboard and print it out in a suitable size and attach it to a hard surface, or display it on a sufficiently large device screen, such as a computer or a tablet. For most use cases, an A4-sized checkerboard works well, but depending on the position and distance of the camera, a bigger checkerboard may be required.



Next, create an empty `pipeline_config.yml` in your project folder and modify it as follows:

```yaml
nodes:
- input.visual:
    source: 0 # change this to the camera you are using
    threading: True
```

(continues on next page)

(continued from previous page)

```
    mirror_image: True
- dabble.camera_calibration
- output.screen
```

Run the above pipeline with peekingduck run. If you are unfamiliar with the pipeline file and running peekingduck, you may refer to the *HelloCV tutorial*. You should see a display of your camera with some instructions overlaid. Follow the instructions to position the checkerboard at 5 different positions in the camera. If the process is successful, the camera coefficients will be calculated and written to a file and you can start using the `augment.undistort` node.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

- **num_corners** (`List[int]`) – **default = [10, 7]**. A list containing the number of internal corners along the vertical and horizontal axes. For example, in the given image above, the checkerboard is of size 11x8, so the number of internal corners is 10x7. If you are using the given checkerboard above, you do not need to change this parameter.
- **scale_factor** (`int`) – **default = 2**. Factor to scale the image by when finding chessboard corners. For example, with a scale of 4, an image of size (1080 x 1920) will be scaled down to (270 x 480) when detecting the corners. Increasing this value reduces computation time. If the node is unable to detect corners, reducing this value may help.
- **file_path** (`str`) – **default = "PeekingDuck/data/camera_calibration_coeffs.yml"**. Path of the YML file to store the calculated camera coefficients.

10.4.5 dabble.check_large_groups

Description

Checks if number of objects in a group exceeds a threshold.

class Node(*config=None*, ***kwargs*)

Checks which groups have exceeded the group size threshold. The group associated with each object is accessed by the `groups` key of *obj_attrs*.

Inputs

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `dabble.check_large_groups` requires the `groups` attribute.

Outputs

`large_groups` (`List[int]`): A list of integers representing the group IDs of groups that have exceeded the size threshold.

Configs

group_size_threshold (`int`) – **default = 5**. Threshold of group size.

Changed in version 1.2.0: `draw.check_large_groups` used to take in `obj_tags` (`List[str]`) as an input data type, which has been deprecated and now subsumed under *obj_attrs*. The same attribute is accessed by using the `groups` key of *obj_attrs*.

10.4.6 dabble.check_nearby_objs

Description

Checks if detected objects are near each other.

class Node(*config=None*, ***kwargs*)

Checks if any objects are near each other.

It does so by comparing the 3D locations of all objects to see which ones are near each other. If the distance between two objects is below the minimum threshold, both would be flagged as near with `tag_msg`. These flags can be accessed by the `flags` key of *obj_attrs*.

Inputs

`obj_3D_locs` (`List[numpy.ndarray]`): A list of N NumPy arrays representing the 3D coordinates (x, y, z) of an object associated with a detected bounding box.

Outputs

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `dabble.check_nearby_objs` produces the `flags` attribute which contains either the `tag_msg` for objects that are near each other or an empty string for objects with no other objects nearby.

Configs

- **near_threshold** (`float`) – **default = 2.0**. Threshold of distance, in metres, between two objects. Objects with distance less than `near_threshold` would be considered as 'near'.
- **tag_msg** (`str`) – **default = "TOO CLOSE!"**. Tag to identify objects which are near others.

Changed in version 1.2.0: `draw.check_nearby_objs` used to return `obj_tags` (`List[str]`) as an output data type, which has been deprecated and now subsumed under *obj_attrs*. The same attribute is accessed by using the `flags` key of *obj_attrs*.

10.4.7 dabble.fps

Description

Calculates the FPS of video.

class Node(*config=None*, ***kwargs*)

Calculates the FPS of the image frame.

This node calculates instantaneous FPS and a 10 frame moving average FPS. A preferred output setting can be set via the configuration file.

Inputs

`pipeline_end` (`bool`): A boolean that evaluates to `True` when the pipeline is completed. Suitable for operations that require the entire inference pipeline to be completed before running.

Outputs

`fps` (`float`): A float representing the Frames Per Second (FPS) when processing a live video stream or a recorded video.

Configs

- **fps_log_display** (`bool`) – **default = True**. Enables logging of 10 frame moving average FPS during execution of PeekingDuck.
- **fps_log_freq** (`int`) – **default = 100**. Frequency of logging moving average FPS for every n frames.
- **dampen_fps** (`bool`) – **default = True**. If `True`, returns moving average FPS. If `False`, returns instantaneous FPS .

10.4.8 dabble.group_nearby_objs

Description

Assigns objects in close proximity to groups.

class Node(*config=None*, ***kwargs*)

Groups objects that are near each other.

It does so by comparing the 3D locations of all objects, and assigning objects near each other to the same group. The group associated with each object is accessed by the `groups` key of *obj_attrs*.

Inputs

`obj_3D_locs` (`List[numpy.ndarray]`): A list of N NumPy arrays representing the 3D coordinates (x, y, z) of an object associated with a detected bounding box.

Outputs

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `dabble.group_nearby_objs` produces the `groups` attribute.

Configs

obj_dist_threshold (`float`) – **default = 1.5**. Threshold of distance, in metres, between two objects. Objects with distance less than `obj_dist_threshold` would be assigned to the same group.

Changed in version 1.2.0: `draw.group_nearby_objs` used to return `obj_tags` (`List[str]`) as an output data type, which has been deprecated and now subsumed under *obj_attrs*. The same attribute is accessed by the `groups` key of *obj_attrs*.

10.4.9 dabble.keypoints_to_3d_loc

Description

Estimates the 3D coordinates of a person given 2D pose coordinates.

class Node(*config=None*, ***kwargs*)

Uses pose keypoint information of the torso to estimate 3D location.

Inputs

`keypoints` (`numpy.ndarray`): A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by `-1`.

Outputs

obj_3D_locs (List[numpy.ndarray]): A list of N NumPy arrays representing the 3D coordinates (x, y, z) of an object associated with a detected bounding box.

Configs

- **focal_length** (float) – **default = 1.14**. Approximate focal length of webcam used, in metres. Example on measuring focal length can be found here.
- **torso_factor** (float) – **default = 0.9**. A factor used to estimate real-world distance from pixels, based on average human torso length in metres. The value varies across different camera set-ups, and calibration may be required.

10.4.10 dabble.statistics

Description

Calculates the cumulative average, minimum, and maximum of a single variable of interest over time.

class Node(*config=None*, ***kwargs*)

Calculates the cumulative average, minimum, and maximum of a single variable of interest (defined as `current result` here) over time. The configurations for this node offer several functions to reduce the incoming data type into a single `current result` of type `int` or `float`, which is valid for the current video frame. `current result` is then used to recalculate the values of the cumulative average, minimum, and maximum for PeekingDuck's running duration thus far.

The configuration for this node is described below using a combination of the Extended BNF and Augmented BNF metasyntax. Concrete examples are provided later for illustration.

```
pkd_data_type   = ? PeekingDuck built-in data types ?
                  e.g. count, large_groups, obj_attrs
user_data_type  = ? user data types produced by custom nodes ?
                  e.g. my_var, my_attrs
dict_key        = ? Python dictionary keys, with optional nesting ?
                  e.g. ["ids"], ["details"]["age"]
data_type       = pkd_data_type | user_data_type
target_attr     = data_type | data_type "[" dict_key "]"

unary_function  = "identity" | "length" | "maximum" | "minimum"
unary_expr      = unary_function ":" target_attr

num_operator    = "==" | ">=" | "<=" | ">" | "<"
num_operand     = ? Python integers or floats ?
num_comparison  = num_operator num_operand

str_operator    = "=="
str_operand     = ? Python strings enclosed by single or double quotes ?
str_comparison  = str_operator str_operand

cond_function   = "cond_count"
cond_expr       = cond_function ":" target_attr ( num_comparison | str_comparison )

configuration   = unary_expr | cond_expr
```

Points to note:

- Square brackets ([]) are used to define <dict_key>, and should not be used elsewhere in the configuration.
- Operands are processed differently depending on whether they are enclosed by single/double quotes, or not. If enclosed, the operand is assumed to be of type str and classified as <str_operand>. If not, the operand is classified as <num_operand> and converted into float for further processing.

The table below illustrates how configuration choices reduce the incoming data type into the <current result>.

<pkd_data_type>: value or <user_data_type>: value	<target_attr>	<unary_expr> or <cond_expr>	<current result>
count: 8	count	identity: count	8
obj_attrs: { ids: [1,2,4], details: { gender: ["male","male","female"], age: [52,17,48] }}	obj_attrs["ids"]	length: obj_attrs["ids"]	3
	obj_attrs ["details"] ["age"]	maximum: obj_attrs ["details"] ["age"]	52
	obj_attrs ["details"] ["gender"]	cond_count: obj_attrs ["details"] ["gender"] == "male"	2
	obj_attrs ["details"] ["age"]	cond_count: obj_attrs ["details"] ["age"] < 60	3

Inputs

all (Any): This data type contains all the outputs from preceding nodes, granting a large degree of flexibility to nodes that receive it. Examples of such nodes include *draw.legend*, *dabble.statistics*, and *output.csv_writer*.

Outputs

cum_avg (float): Cumulative average of an attribute over time.

Note that *cum_avg* will not be updated if there are no detections. For example, if *cum_avg* = 10 for video frame 1, and there are no detections in the following 500 frames, *cum_avg* is still 10 for video frame 501.

cum_max (float | int): Cumulative maximum of an attribute over time.

cum_min (float | int): Cumulative minimum of an attribute over time.

Configs

- **identity** (str) – **default=null** Accepts <target_attr> of types int or float, and returns the same value.
- **length** (str) – **default=null** Accepts <target_attr> of types List[Any] or Dict[str, Any], and returns its length.
- **minimum** (str) – **default=null** Accepts <target_attr> of types List[float | int] or Dict[str, float | int], and returns the minimum element within for the current frame.

Not to be confused with the *cum_min* output data type, which represents the cumulative minimum over time.

- **maximum** (`str`) – **default=null** Accepts `<target_attr>` of types `List[float | int]` or `Dict[str, float | int]`, and returns the maximum element within for the current frame. Not to be confused with the *cum_max* output data type, which represents the cumulative maximum over time.
- **cond_count** (`str`) – **default=null** Accepts `<target_attr>` of types `List[float | int | str]`, and checks if each element in the list fulfils the condition described by `<num_comparison>` or `<str_comparison>`. The number of elements that fulfil the condition are counted towards `<current result>`.

10.4.11 dabble.tracking

Description

Performs multiple object tracking for detected bboxes.

class Node(*config=None*, ***kwargs*)

Uses bounding boxes detected by an object detector model to track multiple objects. `dabble.tracking` is a useful alternative to `model.fairmot` and `model.jde` as it can track bounding boxes detected by the upstream object detector and is not limited to only `"person"` detections.

Currently, two types of tracking algorithms can be selected: MOSSE and IOU. Information on the algorithms' performance can be found *here*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `dabble.tracking` produces the `ids` attribute which contains the tracking IDs of the detections.

Configs

- **tracking_type** (`str`) – **{"iou", "mosse"}, default="iou"**. Type of tracking algorithm to be used. For more information about the trackers, please view the *Object Counting (Over Time) use case*.
- **iou_threshold** (`float`) – **[0, 1], default=0.1**. Minimum IoU value to be used with the matching logic.
- **max_lost** (`int`) – **[0, sys.maxsize), default=10**. Maximum number of frames to keep "lost" tracks after which they will be removed. Only used when `tracking_type = iou`.

10.4.12 dabble.zone_count

Description

Counts the number of detected objects within a boundary.

class Node(*config=None*, ***kwargs*)

Uses the bottom midpoints of all detected bounding boxes and outputs the number of object counts in each specified zone.

Given the bottom mid-points of all detected objects, this node checks if the points fall within the area of the specified zones. The zone counting detections depend on the configuration set in the object detection models, such as the type of object to detect.

Inputs

`btm_midpoint` (`List[Tuple[int, int]]`): A list of tuples each representing the (x, y) coordinates of the bottom middle of a bounding box for use in zone analytics. The order corresponds to *bboxes*.

Outputs

`zones` (`List[List[Tuple[float, ...]]]`): A nested list of Z zones. Each zone is described by 3 **or more** points which contains the (x, y) coordinates forming the boundary of a zone. The order corresponds to *zone_count*.

`zone_count` (`List[int]`): A list of integers representing the count of a pre-selected object class (for example, "person") detected in each specified zone. The order corresponds to *zones*.

Configs

- **resolution** (`List[int]`) – **default = [1280, 720]**. Resolution of input array to calculate pixel coordinates of zone points.
- **zones** (`List[List[List[Union[int, float]]]]`) – **default = [[[0, 0], [640, 0], [640, 720], [0, 720]], [[0.5, 0], [1, 0], [1, 1], [0.5, 1]]]** Used for creation of specific zones with either the absolute pixel values or % of resolution as a fraction between [0, 1].

10.5 draw

Description

Draws results/outputs to an image.

Deprecated since version 1.2.0: `draw.image_processor` is deprecated, and replaced by the nodes *`augment.brightness`* and *`augment.contrast`*.

Modules

`draw.bbox`	Draws bounding boxes over detected objects.
`draw.blur_bbox`	Blurs area bounded by bounding boxes over detected object.
`draw.btm_midpoint`	Draws the bottom middle point of a bounding box.
`draw.group_bbox_and_tag`	Draws large bounding boxes with tags, over identified groups of bounding boxes.
`draw.heat_map`	Superimposes a heat map over an image.
`draw.instance_mask`	Draws instance segmentation masks.
`draw.legend`	Displays selected information from preceding nodes in a legend box.
`draw.mosaic_bbox`	Mosaics area bounded by bounding boxes over detected object
`draw.poses`	Draws keypoints on a detected pose.
`draw.tag`	Draws a tag (from *obj_attrs*) above each bounding box.
`draw.zones`	Draws the 2D boundaries of a zone.

10.5.1 draw.bbox

Description

Draws bounding boxes over detected objects.

class Node(*config=None*, ***kwargs*)

Draws bounding boxes on image.

The `draw.bbox` node uses *bboxes* and, optionally, *bbox_labels* from the model predictions to draw the bbox predictions onto the image.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

Outputs

`none`: No outputs produced.

Configs

show_labels (`bool`) – **default = False**. If `True`, shows class label, e.g., "person", above the bounding box.

10.5.2 draw.blur_bbox

Description

Blurs area bounded by bounding boxes over detected object.

class Node(*config=None*, ***kwargs*)

Blurs area bounded by bounding boxes on image.

The `draw.blur_bbox` node blurs the areas of the image bounded by the bounding boxes output from an object detection model.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

blur_kernel_size (`int`) – **default = 50**. This defines the kernel size used in the blur filter. Larger values of `blur_kernel_size` gives more intense blurring.

10.5.3 draw.btm_midpoint

Description

Draws the bottom middle point of a bounding box.

class Node(*config=None*, ***kwargs*)

The `draw.btm_midpoint` node uses *bboxes* from the model predictions to draw the bottom midpoint of each bbox as a dot onto the image. For better understanding of the use case, refer to the *Zone Counting use case*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`btm_midpoint` (`List[Tuple[int, int]]`): A list of tuples each representing the (x, y) coordinates of the bottom middle of a bounding box for use in zone analytics. The order corresponds to *bboxes*.

Outputs

`none`: No outputs produced.

Configs

None.

10.5.4 draw.group_bbox_and_tag

Description

Draws large bounding boxes with tags, over identified groups of bounding boxes.

class Node(*config=None*, ***kwargs*)

Draws large bounding boxes with tags over multiple object bounding boxes which have been identified as belonging to the same group.

The *large_groups* data type from `dabble.check_large_groups`, and the `groups` key of the *obj_attrs* data type from `dabble.group_nearby_objs`, are inputs for this node which identifies the different groups, and the group associated with each bounding box.

For better understanding, refer to the *Group Size Checking use case*.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes. `draw.group_bbox_and_tag` requires the `groups` attribute from `dabble.group_nearby_objs`.

`large_groups` (`List[int]`): A list of integers representing the group IDs of groups that have exceeded the size threshold.

Outputs

`none`: No outputs produced.

Configs

tag (`str`) – **default = "LARGE GROUP!"**. The string message printed when a large group is detected.

Changed in version 1.2.0: `draw.group_bbox_and_tag` used to take in `obj_tags` (`List[str]`) as an input data type, which has been deprecated and now subsumed under *obj_attrs*. The same attribute is accessed by using the `groups` key of *obj_attrs*.

10.5.5 draw.heat_map

Description

Superimposes a heat map over an image.

class Node(*config=None*, ***kwargs*)

Superimposes a heat map over an image.

The `draw.heat_map` node helps to identify areas that are more crowded. Areas that are more crowded are highlighted in red while areas that are less crowded are highlighted in blue.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`density_map` (`numpy.ndarray`): A NumPy array of shape (H, W) representing the number of persons per pixel. H and W are the height and width of the input image, respectively. The sum of the array is the estimated total number of people. This is produced by nodes such as `model.csrnet`.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

None.

10.5.6 draw.instance_mask

Description

Draws instance segmentation masks.

`class Node`(*config=None*, ***kwargs*)

Draws instance segmentation masks on image.

The `draw.mask` node draws instance segmentation masks onto the detected object instances.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`masks` (`numpy.ndarray`): A NumPy array of shape (N, H, W) containing N detected binarized masks where H and W are the height and width of the masks. The order corresponds to *bbox_labels*.

`bbox_labels` (`numpy.ndarray`): A NumPy array of shape (N) containing strings representing the labels of detected objects. The order corresponds to *bboxes* and *bbox_scores*.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

- **instance_color_scheme** (`str`) – **{"random", "hue_family"}, default = "hue_family"** This defines what colors to use for the standard masks. "hue_family": use the same hue for each instance belonging to the same class, but with a slightly different saturation. "random": use a random color for all instances.
- **effect** (`dict`) – **{contrast: null, brightness: null, gamma_correction: null, blur: null, mosaic: null}** This defines the effect (if any) to apply to either the masked (objects) or unmasked (background) areas of the image. If no effect is selected, a "standard" instance segmentation mask will be drawn and colored according to the instance_color_scheme. For example, to apply the contrast effect to the objects in the image, set the following config in pipeline_config.yml:

  effect : {contrast: 1.2}

  Note that at most one effect can be enabled at a time.

Effect	Description	Data Type	Range
contrast	Adjusts contrast using this value as the "alpha" parameter.	float	[0.0, 3.0]
brightness	Adjusts brightness using this value as the "beta" parameter.	int	[-100, 100]
gamma_correction	Adjusts gamma using this value as the "gamma" parameter.	float	[0.0, +inf]
blur	Blurs the area using this value as the "blur_kernel_size" parameter. Larger values gives more intense blurring.	int	[1, +inf]
mosaic	Mosaics the area using this value as the resolution of a mosaic filter (width × height). The number corresponds to the number of rows and columns used to create a mosaic. For example, the setting (`mosaic: 25`) creates a 25 × 25 mosaic filter. Increasing the number increases the intensity of pixelation over an area.	int	[1, +inf]

- **effect_area** (`str`) – **{"objects", "background"}, default = "objects"** This defines where the effect should be applied. "objects": the effect is applied to the masked areas of the image. "background": the effect is applied to the unmasked areas of the image.
- **contours** (`dict`) – **{show: False, thickness: 2}**

Contours	Description	Data Type	Range
show	This determines whether to show the contours of the masks.	bool	N.A.
thickness	This defines the thickness of the contours.	int	[1, +inf]

10.5.7 draw.legend

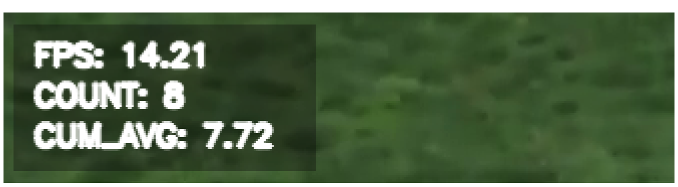
Description

Displays selected information from preceding nodes in a legend box.

class Node(*config=None*, ***kwargs*)

Draws a translucent legend box on a corner of the image, containing selected information produced by preceding nodes in the format `<data type>: <value>`. Supports in-built PeekingDuck data types defined in *Glossary* as well as custom data types produced by custom nodes.

This example screenshot shows *fps* from `dabble.fps`, *count* from `dabble.bbox_count` and *cum_avg* from `dabble.statistics` displayed within the legend box.



With the exception of the *zone_count* data type from `dabble.zone_count`, all other selected in-built PeekingDuck data types or custom data types must be of types `int`, `float`, or `str`. Note that values of float type such as *fps* and *cum_avg* are displayed in 2 decimal places.

Inputs

`all` (`Any`): This data type contains all the outputs from preceding nodes, granting a large degree of flexibility to nodes that receive it. Examples of such nodes include `draw.legend`, `dabble.statistics`, and `output.csv_writer`.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

- **box_opacity** (`float`) – **default = 0.3**. Opacity of legend box background. A value of 0.0 causes the legend box background to be fully transparent, while a value of 1.0 causes it to be fully opaque.
- **font** (`Dict[str, Union[float, int]]`) – **default = {size: 0.7, thickness: 2}.** Size and thickness of font within legend box. Examples of visually acceptable options are: 720p video: {size: 0.7, thickness: 2} 1080p video: {size: 1.0, thickness: 3}
- **position** (`str`) – **{"top", "bottom"}, default = "bottom"**. Position to draw legend box. "top" draws it at the top-left position while "bottom" draws it at bottom-left.
- **show** (`List[str]`) – **default = []**. Include in this list the desired data type(s) to be drawn within the legend box, such as `["fps", "count", "cum_avg"]` in the example screenshot. Custom data types produced by custom nodes are also supported. If no data types are included, an error will be produced.

Changed in version 1.2.0: Merged previous `all_legend_items` and `include` configs into a single `show` config for greater clarity. Added support for drawing custom data types produced by custom nodes, to improve the flexibility of this node.

10.5.8 draw.mosaic_bbox

Description

Mosaics area bounded by bounding boxes over detected object

class Node(*config=None*, ***kwargs*)

Mosaics areas bounded by bounding boxes on image.

The `draw.mosaic_bbox` node helps to anonymize detected objects by pixelating the areas bounded by bounding boxes in an image.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

Outputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

Configs

mosaic_level (int) – **default = 7**. Defines the resolution of a mosaic filter (width × height). The number corresponds to the number of rows and columns used to create a mosaic. For example, the default setting (`mosaic_level = 7`) creates a 7×7 mosaic filter. Increasing the number increases the intensity of pixelization over an area.

10.5.9 draw.poses

Description

Draws keypoints on a detected pose.

class Node(*config=None*, ***kwargs*)

Draws poses onto image.

The `draw.poses` node uses the *keypoints*, *keypoint_scores*, and *keypoint_conns* predictions from pose models to draw the human poses onto the image. For better understanding, check out the pose models such as `HRNet` and `PoseNet`.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`keypoints` (`numpy.ndarray`): A NumPy array of shape $(N, K, 2)$ containing the (x, y) coordinates of detected poses where N is the number of detected poses, and K is the number of individual keypoints. Keypoints with low confidence scores (below threshold) will be replaced by `-1`.

`keypoint_scores` (`numpy.ndarray`): A NumPy array of shape (N, K) containing the confidence scores of detected poses where N is the number of detected poses and K is the number of individual keypoints. The confidence score has a range of $[0, 1]$.

`keypoint_conns` (`numpy.ndarray`): A NumPy array of shape $(N, D'_n, 2, 2)$ containing the (x, y) coordinates of adjacent keypoint pairs where N is the number of detected poses, and D'_n is the number of valid keypoint pairs for the the n-th pose where both keypoints are detected.

Outputs

`none`: No outputs produced.

Configs

None.

10.5.10 draw.tag

Description

Draws a tag (from *obj_attrs*) above each bounding box.

class Node(*config=None*, ***kwargs*)

Draws a tag above each bounding box in the image, using information from selected attributes in *obj_attrs*. In the general example below, *obj_attrs* has 2 attributes (*<attr a>* and *<attr b>*). There are *n* detected bounding boxes, and each attribute has *n* corresponding tags stored in a list. The `show` config described subsequently is used to choose the attribute or attributes to be drawn.

```
{"obj_attrs": {<attr a>: [<tag 1>, ..., <tag n>], <attr b>: [<tag 1>, ..., <tag n>]}
↪}
```

The following type conventions need to be observed:

- Each attribute must be of type `List`, e.g., `<attr a>: [<tag 1>, ..., <tag n>]`
- Each tag must be of type `str`, `int`, `float`, or `bool` to be convertable into `str` type for drawing

In the example below, *obj_attrs* has 3 attributes (*"ids"*, *"gender"* and *"age"*), where the last 2 attributes are nested within *"details"*. There are 2 detected bounding boxes, and thus each attribute consists of a list with 2 tags.

```
# Example
{"obj_attrs": {"ids":[1,2], "details": {"gender": ["female","male"], "age": [52,17]}
↪}
```

The table below illustrates how `show` can be configured to achieve different outcomes for this example. Key takeaways are:

- To draw nested attributes, include all the keys leading to them (within the *obj_attrs* dictionary), separating each key with a `->`.
- To draw multiple comma-separated attributes above each bounding box, add them to the list of `show` config.

No.	show config	Tag above 1st bounding box	Tag above 2nd bounding box
1.	["ids"]	"1"	"2"
2.	["details -> gender"]	"female"	"male"
3.	["details -> age", "details -> gender"]	"52, female"	"17, male"

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`bboxes` (`numpy.ndarray`): A NumPy array of shape $(N, 4)$ containing normalized bounding box coordinates of N detected objects. Each bounding box is represented as (x_1, y_1, x_2, y_2) where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner. The order corresponds to *bbox_labels* and *bbox_scores*.

`obj_attrs` (`Dict[str, Any]`): A dictionary of attributes associated with each bounding box, in the same order as *bboxes*. Different nodes that produce this *obj_attrs* output type may contribute different attributes.

Outputs

`none`: No outputs produced.

Configs

- **show** (`List[str]`) – **default = []**. List of desired attributes to be drawn. For more details on how to use this config, see the section above.
- **tag_color** (`List[int]`) – **default = [77, 103, 255]**. Define the color of the drawn tag, in BGR format. Defined values have to be integers, and $0 \leq value \leq 255$.

Changed in version 1.2.0: *draw.tag* used to take in `obj_tags` (`List[str]`) as an input data type, which has been deprecated and now subsumed under *obj_attrs*, giving this node more flexibility. Also, the `tag_color` config is added to provide the option of changing the tag's color.

10.5.11 draw.zones

Description

Draws the 2D boundaries of a zone.

class Node(*config=None*, ***kwargs*)

Draws the boundaries of each specified zone onto the image.

The `draw.zones` node uses the *zones* output from the `dabble.zone_count` node to draw a bounding box that represents the zone boundaries onto the image.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`zones` (`List[List[Tuple[float, ...]]]`): A nested list of Z zones. Each zone is described by 3 **or more** points which contains the (x, y) coordinates forming the boundary of a zone. The order corresponds to *zone_count*.

Outputs

`none`: No outputs produced.

Configs

None.

10.6 output

Description

Writes/displays the outputs of the pipeline.

Modules

output.csv_writer	Records the nodes' outputs to a CSV file.
output.media_writer	Writes the output image/video to file.
output.screen	Shows the outputs on your display.

10.6.1 output.csv_writer

Description

Records the nodes' outputs to a CSV file.

`class Node`(*config=None*, ***kwargs*)

Tracks user-specified parameters and outputs the results in a CSV file.

Inputs

`all` (`List`) – A placeholder that represents a flexible input. Actual inputs to be written into the CSV file can be configured in `stats_to_track`.

Outputs

`none`: No outputs produced.

Configs

- **stats_to_track** (`List[str]`) – **default = ["keypoints", "bboxes", "bbox_labels"]**. Parameters to log into the CSV file. The chosen parameters must be present in the data pool.
- **file_path** (`str`) – **default = "PeekingDuck/data/stats.csv"**. Path of the CSV file to be saved. The resulting file name would have an appended timestamp.
- **logging_interval** (`int`) – **default = 1**. Interval between each log, in terms of seconds.

10.6.2 output.media_writer

Description

Writes the output image/video to file.

`class Node`(*config=None*, ***kwargs*)

Outputs the processed image or video to a file. A timestamp is appended to the end of the file name.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`filename` (`str`): The filename of video/image being read.

`saved_video_fps` (`float`): FPS of the recorded video, upon filming.

`pipeline_end` (`bool`): A boolean that evaluates to `True` when the pipeline is completed. Suitable for operations that require the entire inference pipeline to be completed before running.

Outputs

`none`: No outputs produced.

Configs

output_dir (`str`) – **default = "PeekingDuck/data/output"**. Output directory for files to be written locally.

10.6.3 output.screen

Description

Shows the outputs on your display.

class Node(*config=None*, ***kwargs*)

Streams the output on your display.

Inputs

`img` (`numpy.ndarray`): A NumPy array of shape $(height, width, channels)$ containing the image data in BGR format.

`filename` (`str`): The filename of video/image being read.

Outputs

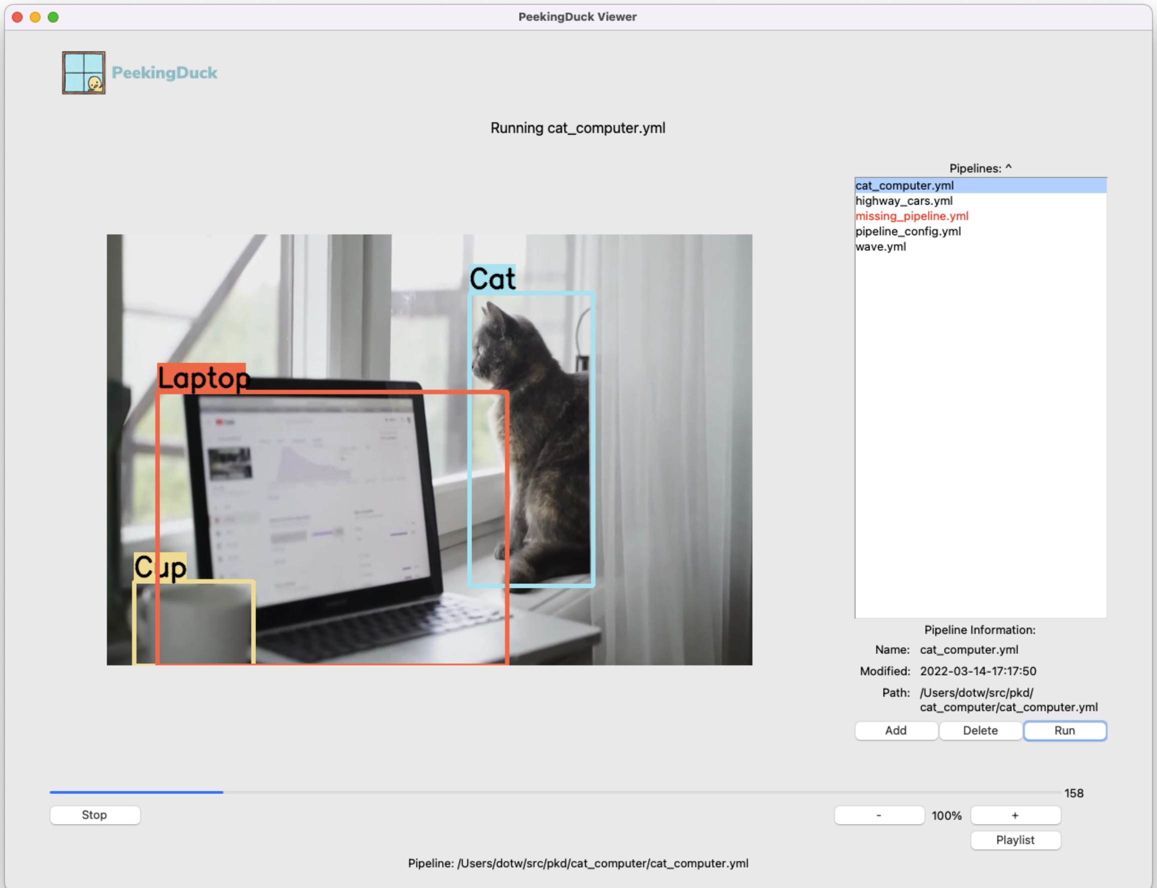
`pipeline_end` (`bool`): A boolean that evaluates to `True` when the pipeline is completed. Suitable for operations that require the entire inference pipeline to be completed before running.

Configs

- **window_name** (`str`) – **default = "PeekingDuck"** Name of the displayed window.
- **window_size** (`Dict[str, Union[bool, int]]`) – **default = { do_resizing: False, width: 1280, height: 720 }** Resizes the displayed window to the chosen width and weight, if `do_resizing` is set to `true`. The size of the displayed window can also be adjusted by clicking and dragging.
- **window_loc** (`Dict[str, int]`) – **default = { x: 0, y: 0 }** X and Y coordinates of the top left corner of the displayed window, with reference from the top left corner of the screen, in pixels.

Note: See Also:

PeekingDuck Viewer: a GUI for running PeekingDuck pipelines.



The PeekingDuck Viewer offers a GUI to view and analyze pipeline output. It has controls to re-play output video, scrub to a frame of interest, zoom video, and a playlist for managing multiple pipelines.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

F