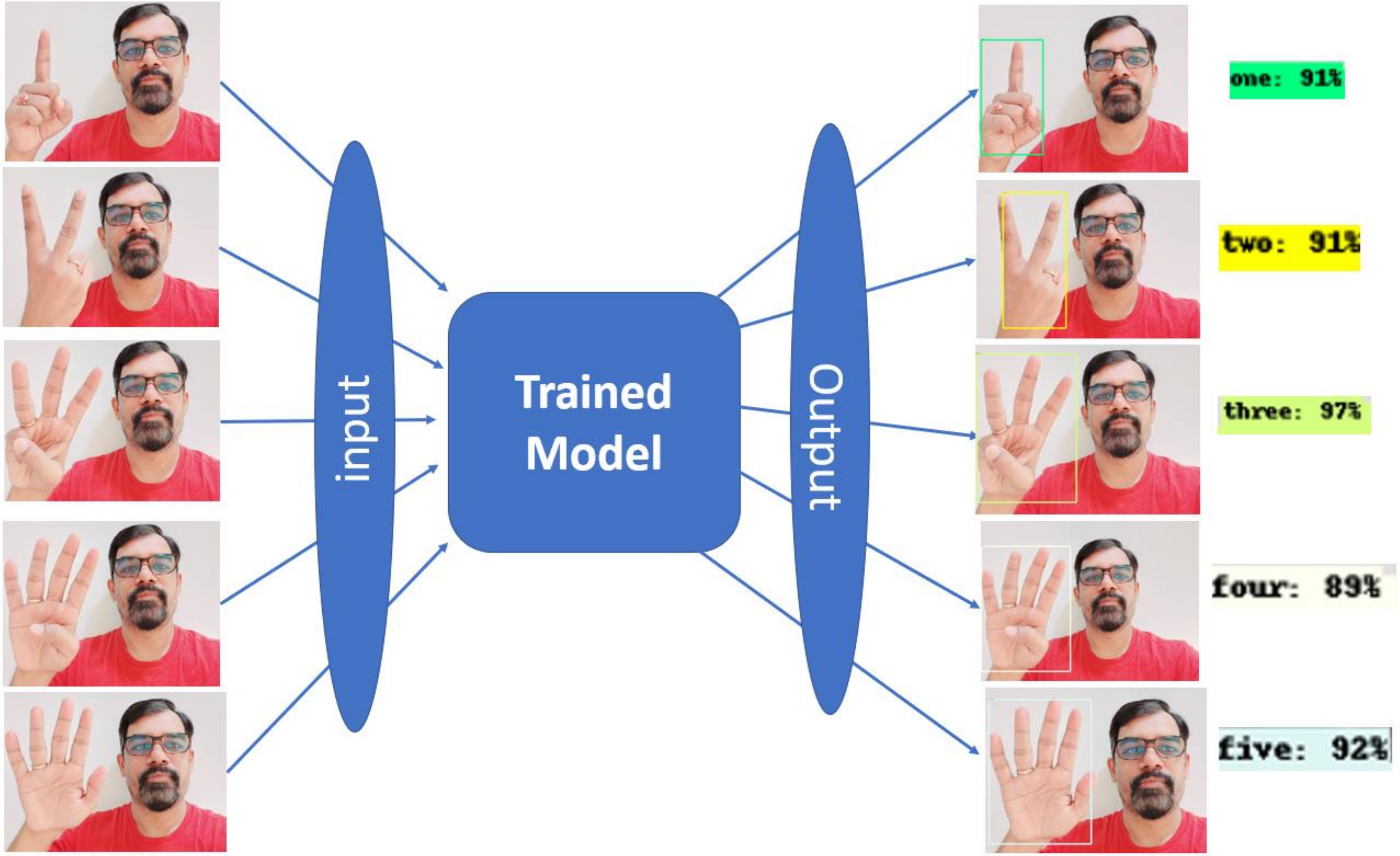
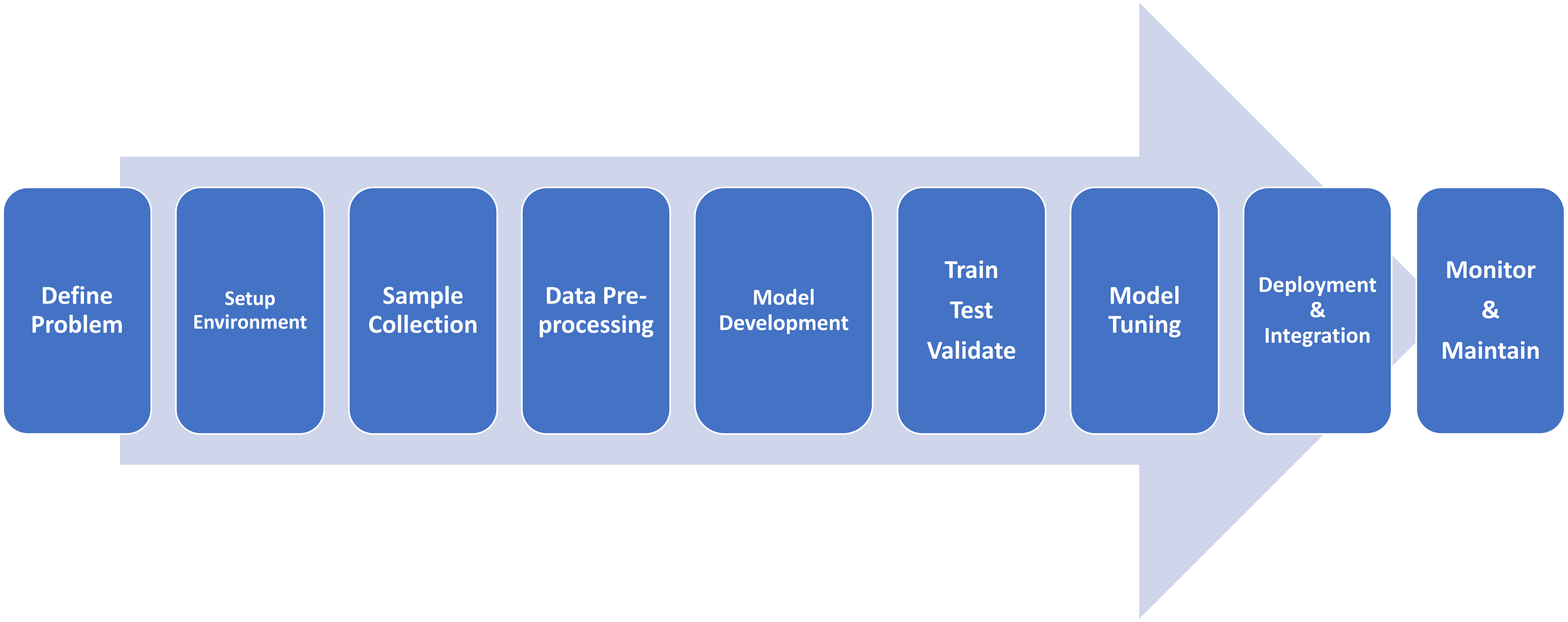
Custom Object Detection – a handbook

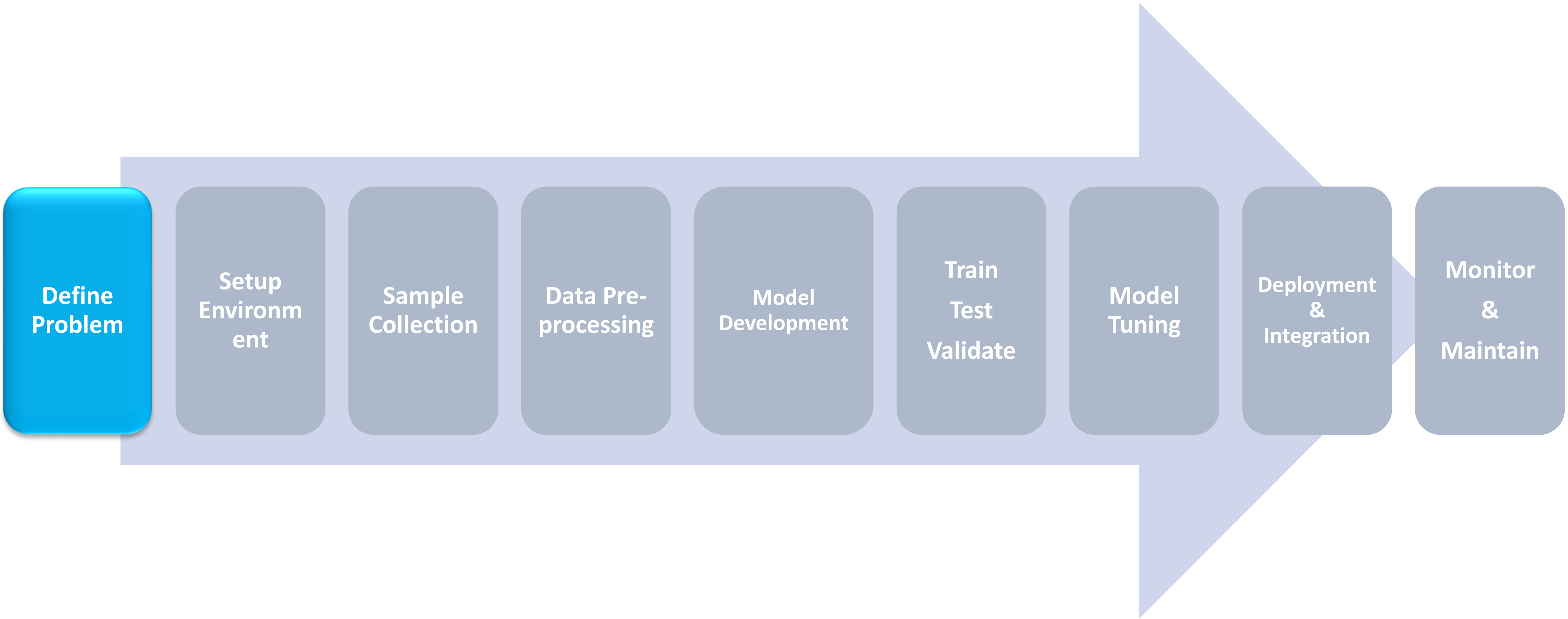


TensorFlow Object Detection API

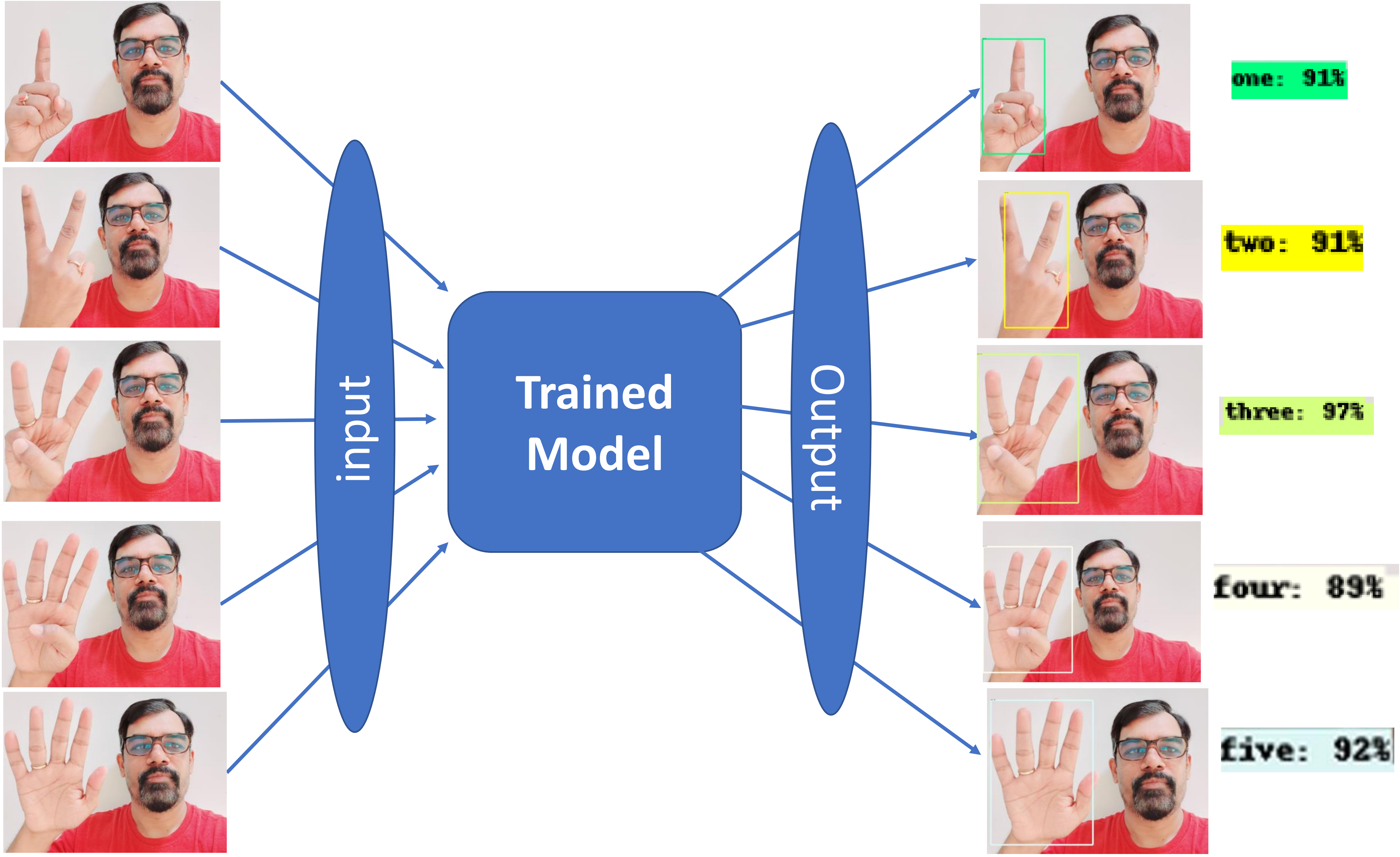
Author : Pramod Singh, PGD(Data Science)

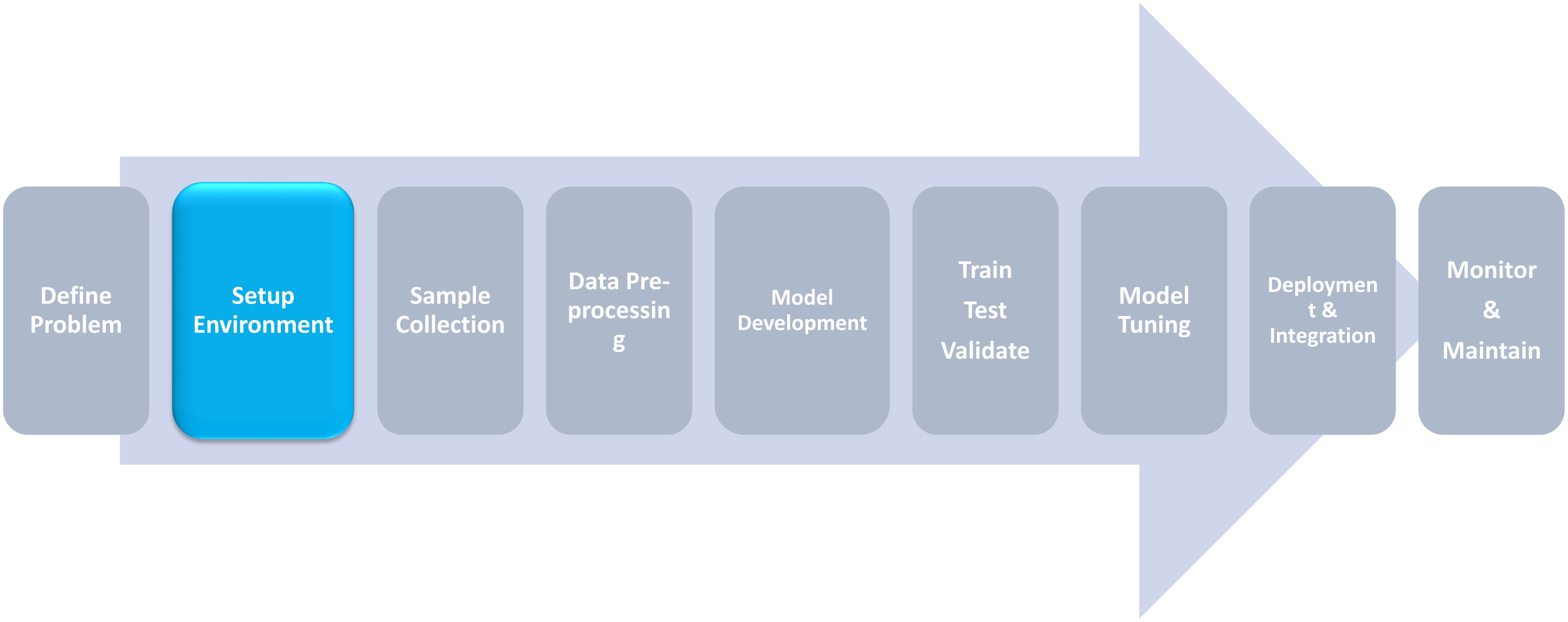
Machine Learning Development Life Cycle




Define Problem
Setup Environm ent
Sample Collection
Data Pre-processing
Model Development
Train
Test
Validate
Model Tuning
Deployment & Integration
Monitor & Maintain

Define Problem





Define Problem
Setup Environment
Sample Collection
Data Pre-processing
Model Development
Train
Test
Validate
Model Tuning
Deployment & Integration
Monitor & Maintain

Setup Environment

Google Collaboratory

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud and also provides free access to computing resources including GPUs.

TensorFlow Installation

an end-to-end open source platform for machine learning.

Install CUDA & CuDNN Toolkit

provides a development environment for creating high performance GPU-accelerated applications.

Install CuDNN Toolkit

is a GPU-accelerated library of primitives for deep neural networks.

COCO API installation

Tensorflow Object Detection API has dependency on pycocotools
Microsoft COCO is a large image dataset designed for object detection, segmentation, and caption generation.
pycocotools is a Python API that assists in loading, parsing and visualizing the annotations in COCO

TensorFlow Object Detection API Installation

A framework that makes it easy to construct, train and deploy object detection models

A collection of object detection models pre-trained on the COCO dataset, the Kitti dataset, the Open Images dataset, the AVA v2.1 dataset, and the iNaturalist Species Detection Dataset

Protobuf Installation/Compilation

Tensorflow Object Detection API uses Protobufs to configure model and training parameters.

Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

Setup Environment

TensorFlow Installation

Installing TensorFlow PIP package

```
!pip install tensorflow
```

Verifying TensorFlow Installation

```
import tensorflow as tf
print(tf.__version__)
```

```
!python -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

Setup Environment

TensorFlow Object Detection API Installation

Downloading the TensorFlow Model Garden

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
cd /content/gdrive/MyDrive/CustomObjectDetection
```

```
!git clone https://github.com/tensorflow/models.git
```

Installing dependencies for Object Detection API

Protobuf Installation/Compilation

```
cd /content/gdrive/MyDrive/CustomObjectDetection/models/research
```

```
!protoc object_detection/protos/*.proto --python_out=.
```

COCO API installation

```
!git clone https://github.com/cocodataset/cocoapi.git
```

```
cd /content/gdrive/MyDrive/CustomObjectDetection/models/research/cocoapi/PythonAPI
```

```
!make
```

```
cp -r pycocotools /content/gdrive/MyDrive/CustomObjectDetection/models/research
```

Installing the Object Detection API

```
cp object_detection/packages/tf2/setup.py .
```

```
!python -m pip install --use-feature=2020-resolver .
```

Setup Environment

Preparing the Workspace

Tensorflow Package

- CustomObjectDetection
  - models
    - community
    - official
    - orbit
    - research

Workspace for custom model

- CustomObjectDetection
  - models
  - workspace
    - finger_count_detection
      - annotations
      - exported-models
      - images
      - models
      - pre-trained-models

Here :

- `annotations`: This folder will be used to store all `*.csv` files and the respective TensorFlow `*.record` files, which contain the list of annotations for our dataset images.
- `exported-models`: This folder will be used to store exported versions of our trained model(s).
- `images`: This folder contains a copy of all the images in our dataset, as well as the respective `*.xml` files produced for each one, once `labelImg` is used to annotate objects.
  - `images/train`: This folder contains a copy of all images, and the respective `*.xml` files, which will be used to train our model.
  - `images/test`: This folder contains a copy of all images, and the respective `*.xml` files, which will be used to test our model.
- `models`: This folder will contain a sub-folder for each of training job. Each subfolder will contain the training pipeline configuration file `*.config`, as well as all files generated during the training and evaluation of our model.
- `pre-trained-models`: This folder will contain the downloaded pre-trained models, which shall be used as a starting checkpoint for our training jobs.

```
!cd /content/gdrive/MyDrive/CustomObjectDetection
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/scripts
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/exported-models
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images/test
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images/train
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/models
!mkdir /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/pre-trained-models
```

Setup Environment

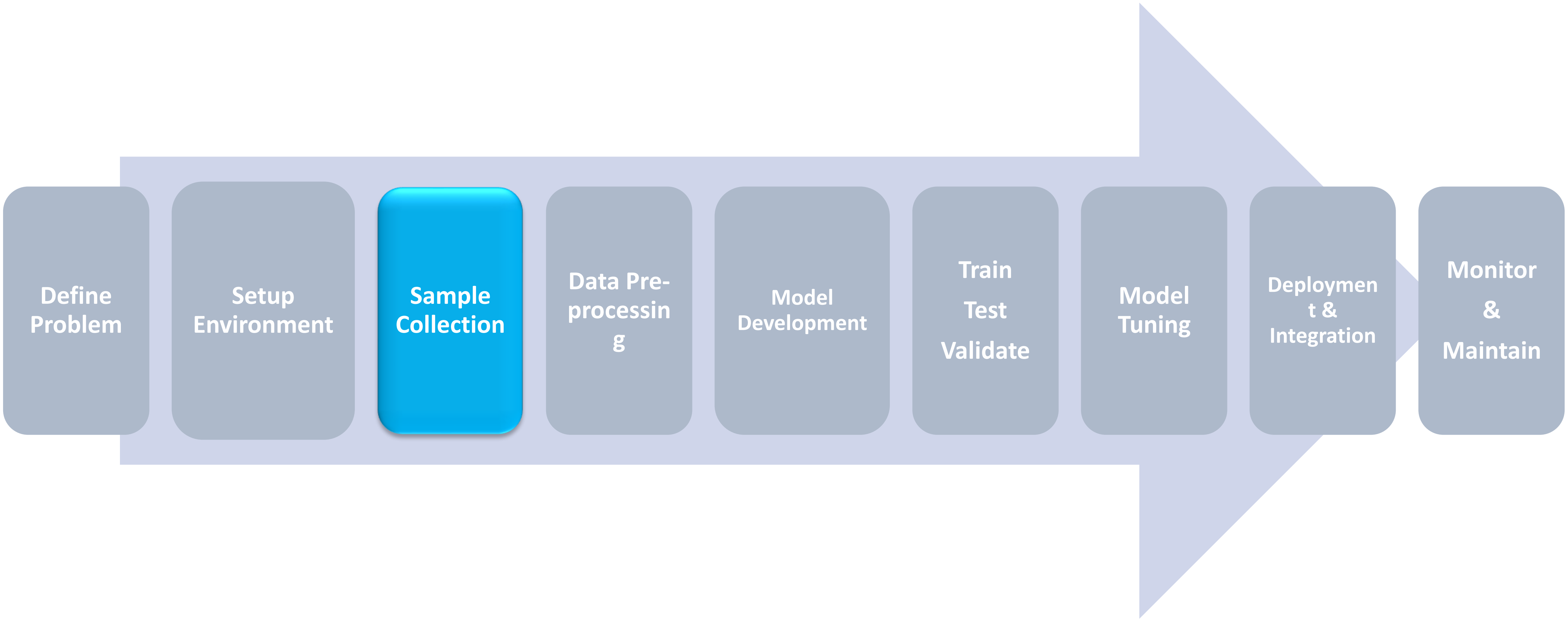
Setting cuda-toolkid

```
!sudo apt install cuda-toolkit-10-2

cd /content/gdrive/MyDrive/CudaPackage

!wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-ubuntu1604.pin

!sudo mv cuda-ubuntu1604.pin /etc/apt/preferences.d/cuda-repository-pin-600

!sudo apt-key adv --fetch-
keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub

!sudo add-apt-
repository "deb http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/ /"
!sudo apt-get -y install cuda

!sudo apt --fix-broken install

!sudo apt install cuda-cudart-10-2

!cp "/content/gdrive/MyDrive/CudaPackage/cudnn-11.4-linux-x64-v8.2.2.26.tgz" "/content/cudnn-11.1.tgz"

!tar -C cudnn/ -zxvf cudnn-11.4-linux-x64-v8.2.2.26.tgz

!sudo cp /content/gdrive/MyDrive/CudaPackage/cudnn/cuda/include/cudnn*.h /usr/local/cuda-11.1/include

!sudo cp /content/gdrive/MyDrive/CudaPackage/cudnn/cuda/lib64/libcudnn* /usr/local/cuda-11.1/lib64
```

Test the setup

There are about 24 tests and all should pass. That means now out setup is complete and ready to work.

```
# Run the test for setup from within TensorFlow/models/research/
!python object_detection/builders/model_builder_tf2_test.py
```

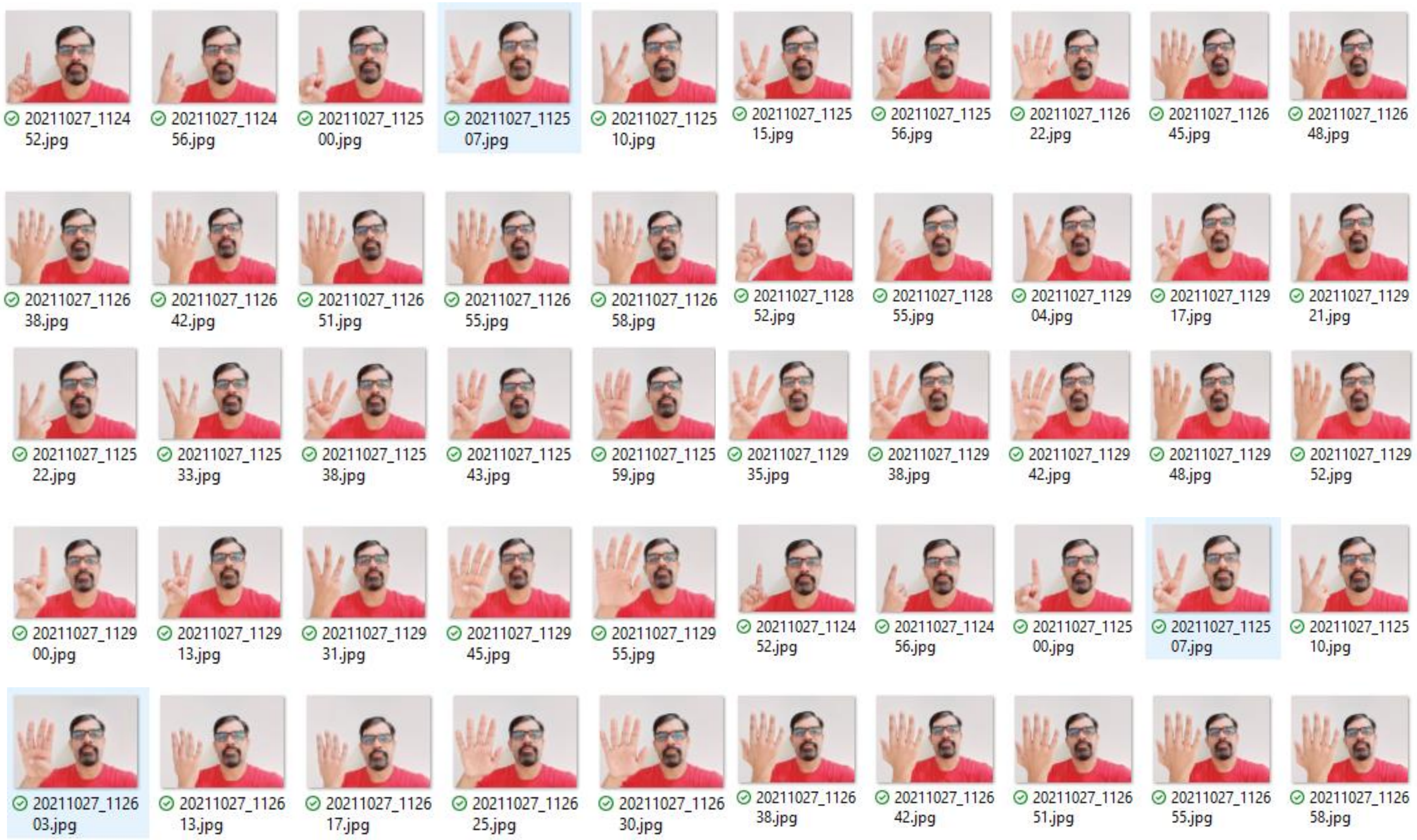

Define Problem
Setup Environment
Sample Collection
Data Pre-processing
Model Development
Train Test Validate
Model Tuning
Deployment & Integration
Monitor & Maintain

Sample Collection

Click / Collects pictures

Thumb rule for good accuracy:

- more picture better accuracy
- variety is very important.



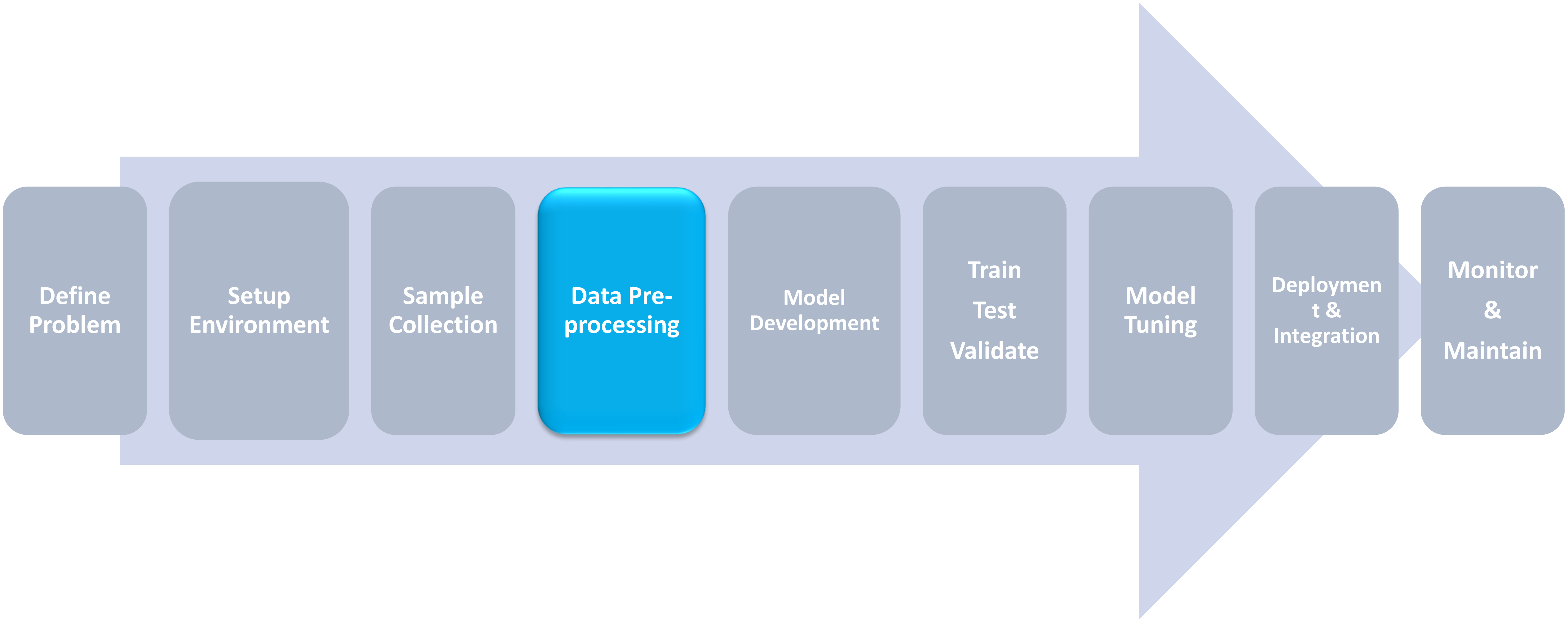

Define Problem
Setup Environment
Sample Collection
Data Pre-processing
Model Development
Train Test Validate
Model Tuning
Deployment & Integration
Monitor & Maintain

Image Annotation

Image annotation plays a significant role in Computer Vision

https://viso.ai/computer-vision/image-annotation/

What is Image Annotation?

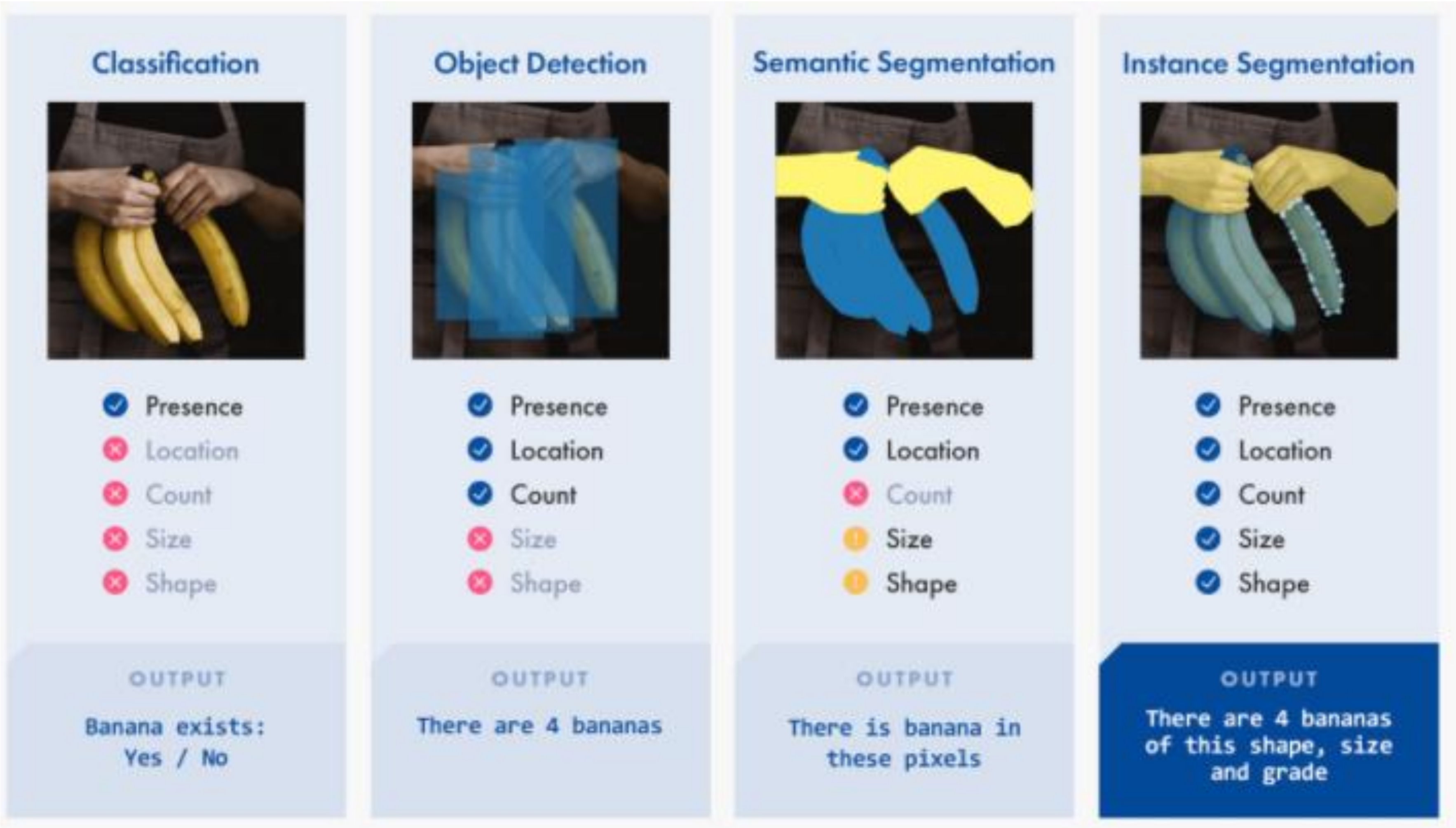
process of labelling images of a dataset.

Why is Image Annotation needed?

it lets the training model know what the important parts of the image are (classes) so that it can later use those notes to identify those classes in new, never-before-seen images

The annotation task usually involves manual work, sometimes with computer-assisted help. A Machine Learning engineer predetermines the labels, known as “classes”, and provides the image-specific information to the computer vision model. After the model is trained and deployed, it will predict and recognize those predetermined features in new images that have not been annotated yet.

Types of Image Annotation

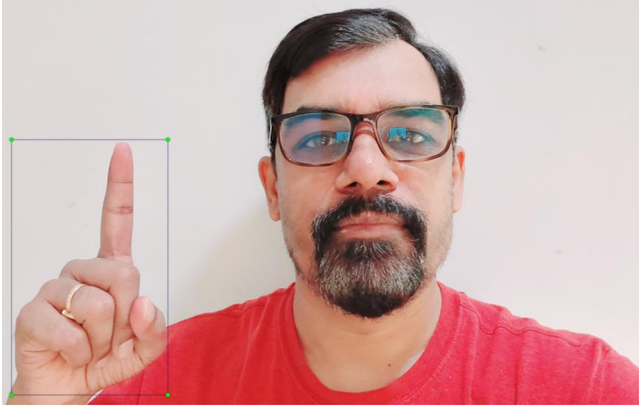


Picture credit : google search

Image Annotation Techniques

Bounding box

These are used to draw a box around the target object, especially when objects are relatively symmetrical, such as vehicles, pedestrians, and road signs. It also is used when the shape of the object is of less interest or when occlusion is less of an issue. Bounding boxes can be two-dimensional (2-D) or three-dimensional (3-D). A 3-D bounding box is also called a cuboid.



Landmarking

This is used to plot characteristics in the data, such as with facial recognition to detect facial features, expressions, and emotions. It also used to annotate body position and alignment, using **pose-point** annotations. In annotating images for sports analytics, for example, you can determine where a baseball pitcher's hand, wrist, and elbow are in relation to one another while the pitcher throws the baseball.

Polygon

This is used to mark each of the highest points (vertices) of the target object and annotate its edges: These are used when objects are more irregular in shape, such as houses, areas of land, or vegetation.

Lines and Splines

Lines and splines annotate the image with straight or curved lines. This is significant for boundary recognition to annotate sidewalks, road marks, and other boundary indicators.

Tracking

This is used to label and plot an object's movement across multiple frames of video. Some image annotation tools have features that include interpolation, which allows an annotator to label one frame, then skip to a later frame, moving the annotation to the new position, where it was later in time. Interpolation fills in the movement and tracks, or interpolates, the object's movement in the interim frames that were not annotated.

Credit : google search

Image Annotation Formats

COCO
Common Objects in Context

COCO has five annotation types:

- object detection,
- keypoint detection,
- stuff segmentation,
- panoptic segmentation, and
- image captioning.

The annotations are stored using JSON.

Pascal VOC
Pattern Analysis, Statistical Modeling and Computational Learning Visual Object Classes

Pascal VOC stores annotation in XML file.

YOLO
You Only Look Once

In YOLO labeling format, a .txt file with the same name is created for each image file in the same directory. Each .txt file contains the annotations for the corresponding image file, that is object class, object coordinates, height and width.

Image Annotation Tools

MakeSense.AI

LabelMe

VGG image annotator

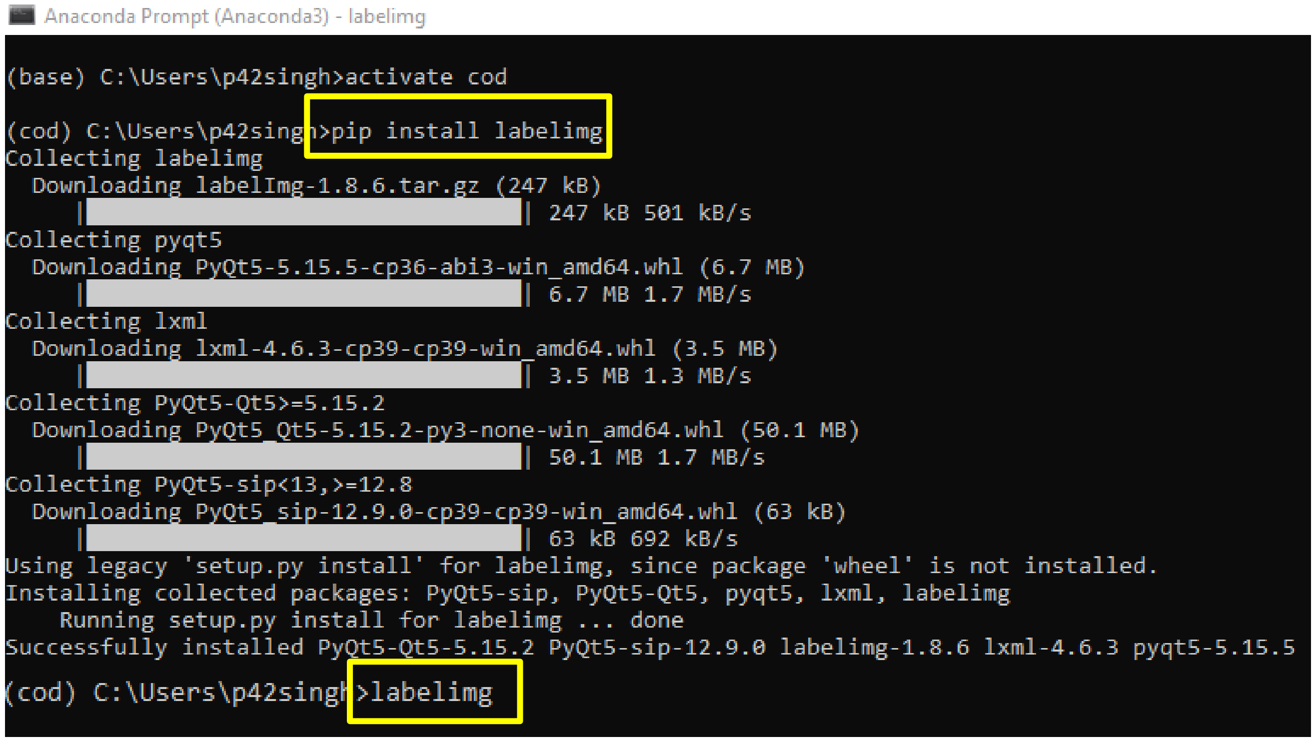
LabelImg

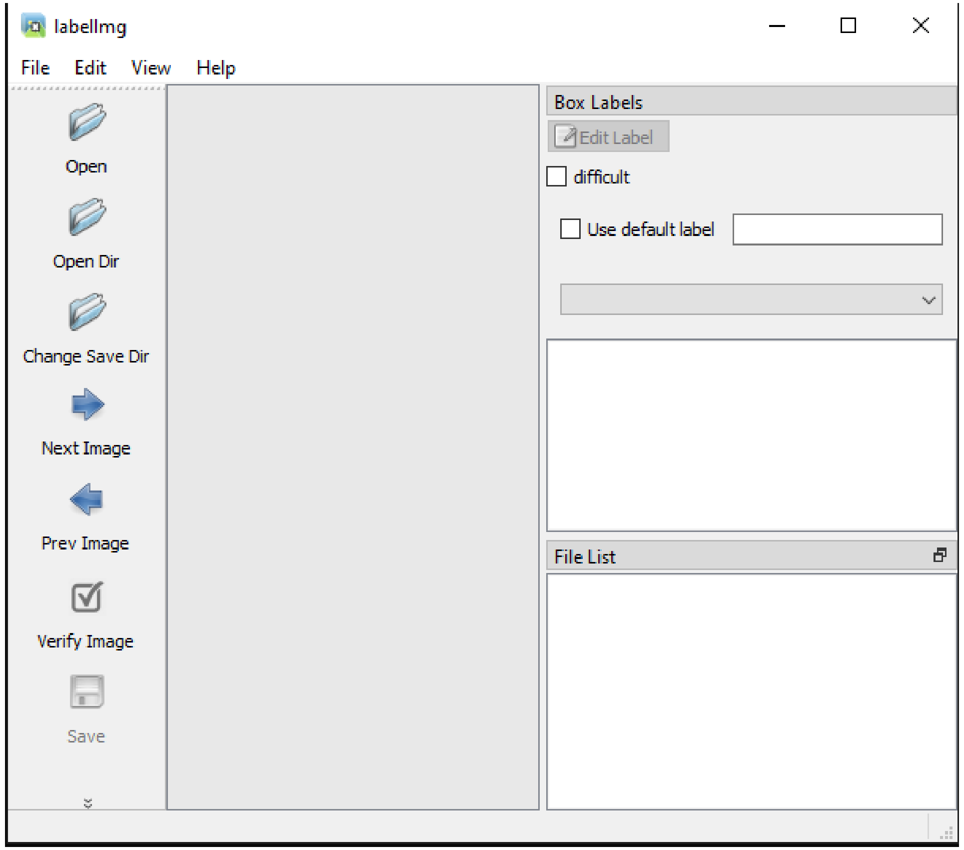
RectLabel

Scalable

Image Annotation Tool - LabelImg

- A graphical image annotation tool.
- Written in Python and uses Qt for its graphical interface.
- Annotations are saved as XML files in PASCAL VOC format, the format used by ImageNet.
- It also supports YOLO and CreateML formats.





A sample XML annotation file based on Pascal VOC format.

```xml
<annotation>
    <folder>train</folder>
    <filename>20211027_112452.jpg</filename>
    <path>C:\DS\ImageAnnotation\train\20211027_112452.jpg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>3264</width>
        <height>2448</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>one</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>52</xmin>
            <ymin>1085</ymin>
            <xmax>852</xmax>
            <ymax>2393</ymax>
        </bndbox>
    </object>
</annotation>
```

Name of the folder

Image file name

Input folder path

height, width in terms of pixels, the depth indicating the number of channels for RGB image depth is 3, for B/W it is 1.

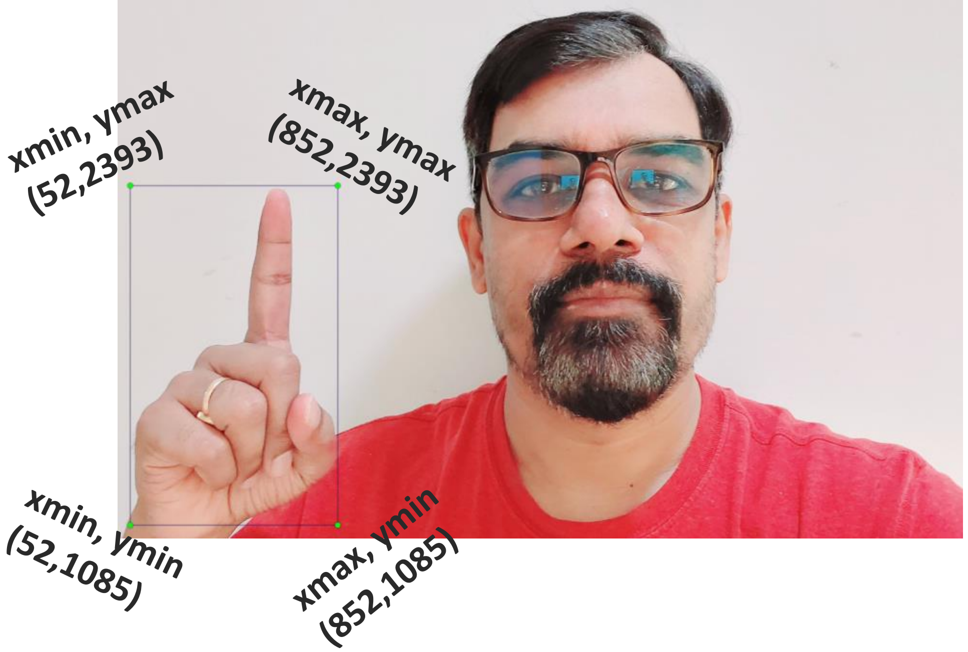
Name/Class of the object

orientation left ,right ,etc

if objects extend beyond bounding box truncated is 1 else 0.

if it is not evaluated difficult is 1 else 0.

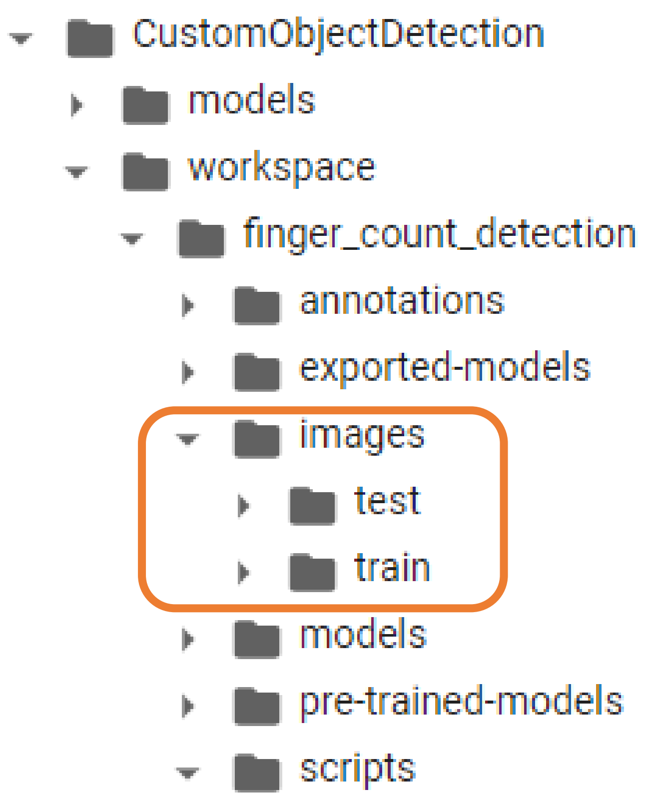
4 coordinates of the bounding box.



Data pre-processing

Partition the Dataset

Typically, the ratio is 9:1, i.e. 90% of the images are used for training and the rest 10% is maintained for testing, but you can chose whatever ratio suits your needs.



Data pre-processing Convert *.xml to TensorFlow Records

Now that we have generated our annotations and split our dataset into the desired training and testing subsets, it is time to convert our annotations into the so called TFRecord format.

Convert *.xml to *.record

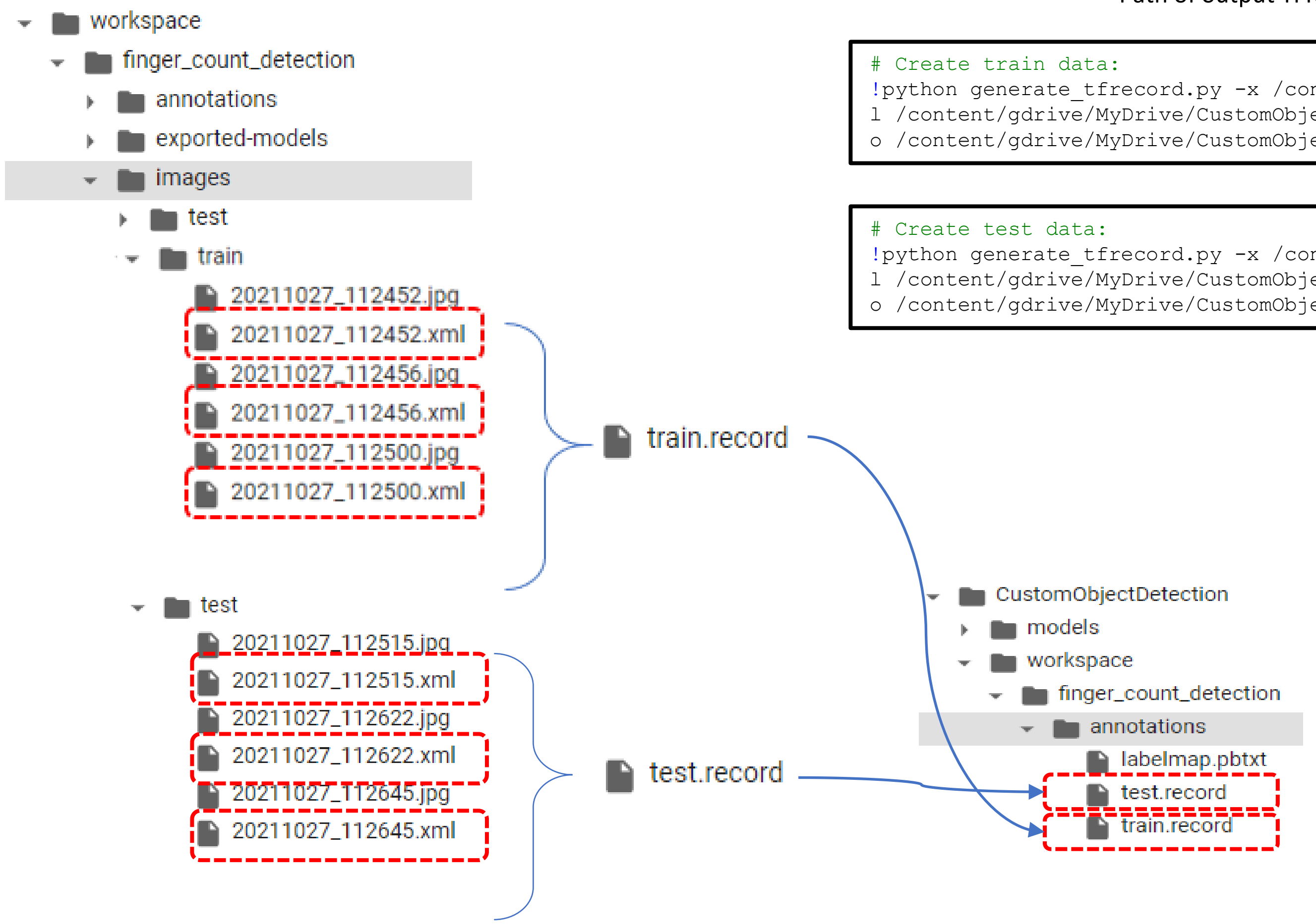
Usage: generate_tfrecord.py [-h] [-x XML_DIR] [-l LABELS_PATH] [-o OUTPUT_PATH]

```
optional arguments:
  -h, --help            show this help message and exit
  -x XML_DIR, --xml_dir XML_DIR
                        Path to the folder where the input .xml files are stored.
  -l LABELS_PATH, --labels_path LABELS_PATH
                        Path to the labels (.pbtxt) file.
  -o OUTPUT_PATH, --output_path OUTPUT_PATH
                        Path of output TFRecord (.record) file.
```

What is label file or .pbtxt

```
# Create train data:
!python generate_tfrecord.py -x /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images/train -
l /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/labelmap.pbtxt -
o /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/train.record
```

```
# Create test data:
!python generate_tfrecord.py -x /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images/test -
l /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/labelmap.pbtxt -
o /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/test.record
```



TensorFlow Records(TFRecords)

Challenge 1

When we are working with a lot of data, it is important to work with a format that is **light** and **fast**, one option is to work with the document's **binary**.

Challenge 2

Loading very large dataset in memory is not feasible and hence we end up writing our code to load them in batches.

TensorFlow with TFRecords solves these challenges

The **TFRecord format** is TensorFlow's own binary storage format.

We don't need to worry about loading data in batches , TensorFlow with TFRecords abstracts this for us how to load the data into memory without we having to program itself.

Binary data takes up less space on disk, takes less time to copy and can be read much more efficiently from disk. Protocol buffers are a cross-platform, cross-language library for efficient serialization of structured data. Protocol messages are defined by .**proto** files, these are often the easiest way to understand a message type.

Structuring TFRecords

- A TFRecord file stores our data as a sequence of binary strings.
- This means we need to specify the structure of our data before we write it to the file.
- TensorFlow provides two components for this purpose: [tf.train.Example]() and [tf.train.SequenceExample](). We have to store each sample of your data in one of these structures, then serialize it and use a [tf.python_io.TFRecordWriter]() to write it to disk.
- The **tf.train.Example** message (or protobuf) is a flexible message type that represents a {"string": value} mapping.

tf.train.Example isn't a normal Python class, but a protocol buffer

```
# Create a Features message using tf.train.Example.

example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
return example_proto.SerializeToString()
```

```
tf_example = tf.train.Example(features=tf.train.Features(feature={
    'image/height': dataset_util.int64_feature(height),
    'image/width': dataset_util.int64_feature(width),
    'image/filename': dataset_util.bytes_feature(filename),
    'image/source_id': dataset_util.bytes_feature(filename),
    'image/encoded': dataset_util.bytes_feature(encoded_jpg),
    'image/format': dataset_util.bytes_feature(image_format),
    'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
    'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
    'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
    'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
    'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
    'image/object/class/label': dataset_util.int64_list_feature(classes),
}))
```

```
tf_example = create_tf_example(group, path)
writer.write(tf_example.SerializeToString())
```

Create Label Map file(labelmap.pbtxt)

BACK

- We will also need to input our classes in TF.
- Datasets use string labels to represent classes(one,two,three,four,five) while the TensorFlow object detection framework works with class ids(numbers).
- Hence we need to create a protobuf message which maps string classes to class id , so they can be converted back and forth as needed. Something like : 1=one, 2=two,3=three,4=four,5=five
- Its called **label map.**
- Label map maps indices to category names (1 for cat & 2 for dog), so that when our convolution network predicts `2`, we know that this corresponds to `dog` and when predicts `1`, we know that this corresponds to `cat`

Label map protobuf message format

```
item {
   id: 1
   name: 'cat'
}
item {
   id: 2
   name: 'dog'
}
```

Label ids should start from 1, because 0 is reserved for internal purpose.

For each class we have an item, for each item we have an id and a name, the Id refers to the id that we use in our TFRecords

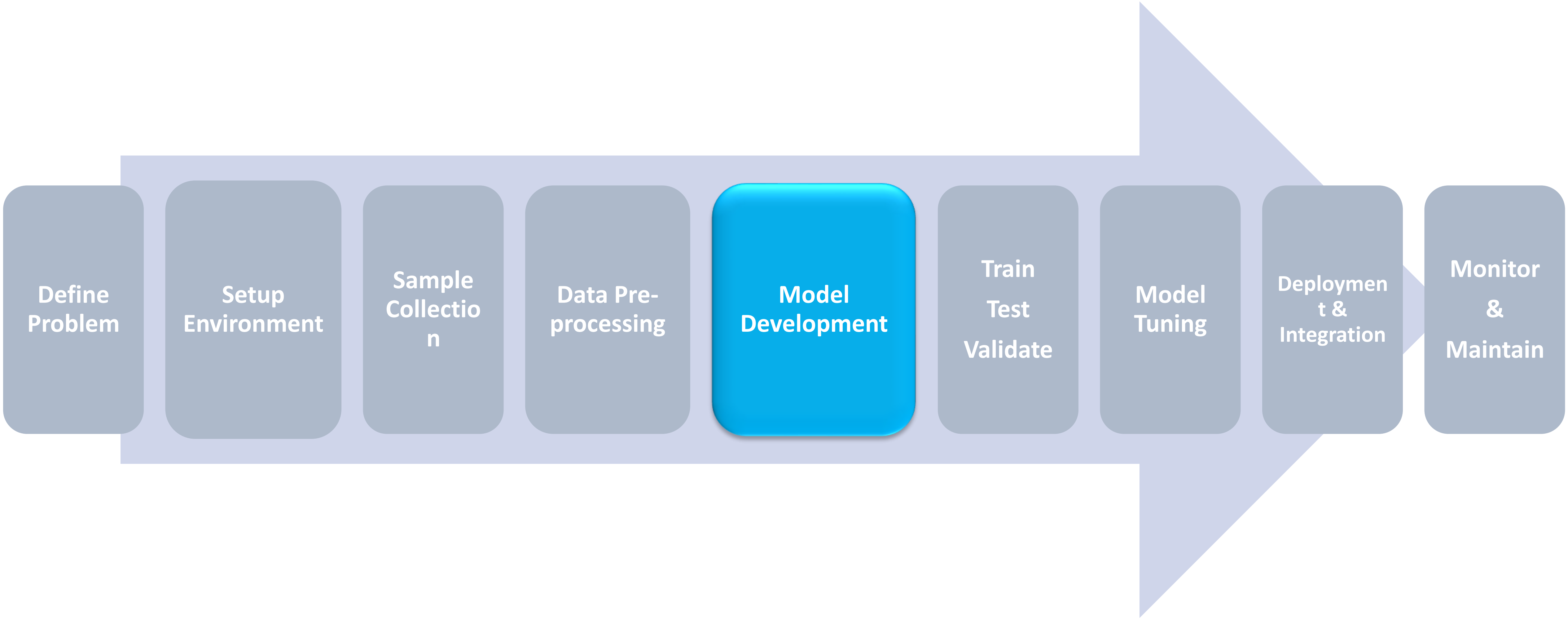
.pbtxt is the file extension.

```
%%writefile /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/labelmap.pbtxt
item {
  id: 1
  name: 'one'
}
item {
  id: 2
  name: 'two'
}
item {
  id: 3
  name: 'three'
}
item {
  id: 4
  name: 'four'
}
item {
  id: 5
  name: 'five'
}
```

- CustomObjectDetection
  - models
  - workspace
    - finger_count_detection
      - annotations
        - labelmap.pbtxt

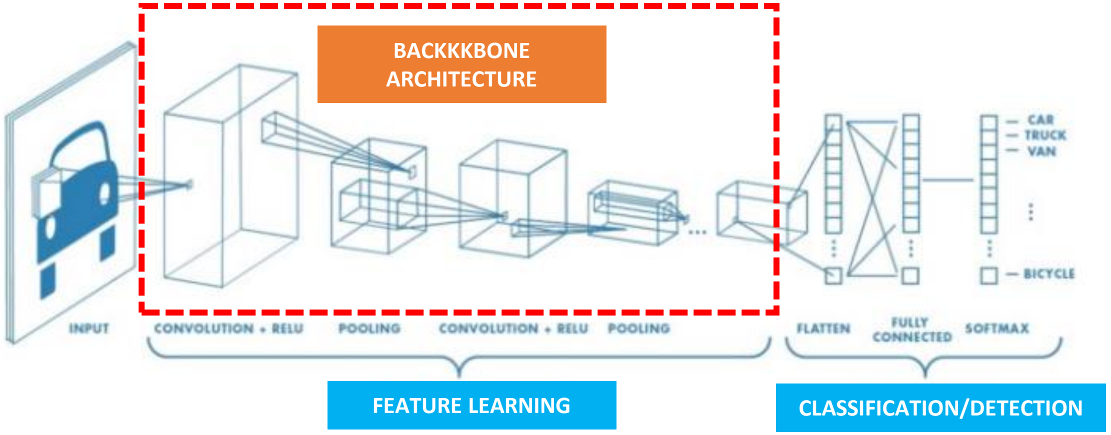
```
item {
  id: 1
  name: 'one'
}
item {
  id: 2
  name: 'two'
}
item {
  id: 3
  name: 'three'
}
item {
  id: 4
  name: 'four'
}
item {
  id: 5
  name: 'five'
}
```

This **label map** is used both by the **training** and **detection** processes and also while creating **TFRecords**.


Define Problem
Setup Environment
Sample Collection
Data Pre-processing
Model Development
Train Test Validate
Model Tuning
Deployment & Integration
Monitor & Maintain

Convolution Neural Network

A lot of progress has been made in recent years on **object detection** with **Convolutional Neural Networks** (CNNs or ConvNets). Hence we will be using CNN.



In CNN ,instead of looking at every single pixel to classify, it extracts the important features from each image with the help of Convolution & Pooling layer.

Convolution layer involves having a **filter** and passing that filter over the image in order to change the underlying image and ultimately narrow down the content of the image to focus on specific, distinct, details.

Convolution+Pooling layer is the backbone of overall CNN architecture.

It is not trivial to decide how many Convolution+Pooling layer is good for a specific problem to obtain the speed and accuracy.

It requires tunning multiple **hyperparameters** and retraining the model and validating the results on each run.

Good thing is that we don't need to do all this from scratch and spend time in reinventing the wheel..

Some great companies & peoples (Researches) have already done great jobs in training various CNN models on available image datasets(COCO from MS & OpenImages from Google) and made these pre-trained models available for us to use. ☺

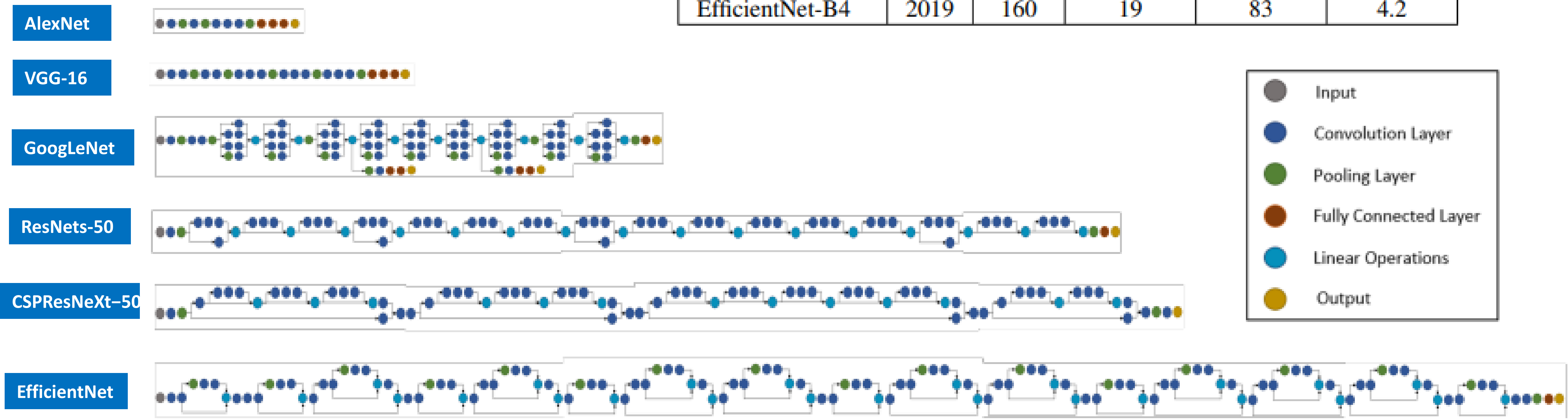
CNN – Backbone Architecture

Backbone architectures are one of the most important component of the object detector.

These networks extract feature from the input image used by the model.

Comparison of Backbone architectures

Model	Year	Layers	Parameters (Million)	Top-1 acc%	FLOPs (Billion)
AlexNet	2012	7	62.4	63.3	1.5
VGG-16	2014	16	138.4	73	15.5
GoogLeNet	2014	22	6.7	-	1.6
ResNet-50	2015	50	25.6	76	3.8
ResNeXt-50	2016	50	25	77.8	4.2
CSPResNeXt-50	2019	59	20.5	78.2	7.9
EfficientNet-B4	2019	160	19	83	4.2



CNN Object Detector Types

An object detector solves two subsequent tasks:

Task #1: Find an arbitrary number of objects (possibly even zero), and

Task #2: Classify every single object and estimate its size with a bounding box.

Object detector which separate those tasks into two stages are called **Two-stage Detector** and the one combines them into one task are called **Single-stage detector.**

Single-stage Detector

2016	YOLO	You Only Look Once
2016	SSD	Single Shot Detector
2017	RetinaNet	
2019	CenterNet	
2020	EfficientDet	
2018	YOLOv3	
2020	YOLOv4	
2021	YOLOR	

Two-stage Detector

2014	R-CNN	Region-based Convolutional Neural Network
2014	SPP	Spatial Pyramid Pooling
2015	Fast R-CNN	Fast Region-based Convolutional Neural Network
2015	Faster R-CNN	Faster Region-based Convolutional Neural Network
2017	Mask R-CNN	Mask Region-based Convolutional Neural Network
2017	FPN	Feature Pyramid Network
2016	R-FCN	Region-based Fully Convolutional Network

Performance comparison of various object detectors on MS COCO and PASCAL VOC 2012 datasets at similar input image size

Model	Year	Backbone	Size	$AP_{[0.5:0.95]}$	$AP_{0.5}$	FPS
R-CNN*	2014	AlexNet	224	-	58.50%	~0.02
SPP-Net*	2015	ZF-5	Variable	-	59.20%	~0.23
Fast R-CNN*	2015	VGG-16	Variable	-	65.70%	~0.43
Faster R-CNN*	2016	VGG-16	600	-	67.00%	5
R-FCN	2016	ResNet-101	600	31.50%	53.20%	~3
FPN	2017	ResNet-101	800	36.20%	59.10%	5
Mask R-CNN	2018	ResNeXt-101-FPN	800	39.80%	62.30%	5
DetectoRS	2020	ResNeXt-101	1333	53.30%	71.60%	~4
YOLO*	2015	(Modified) GoogLeNet	448	-	57.90%	45
SSD	2016	VGG-16	300	23.20%	41.20%	46
YOLOv2	2016	DarkNet-19	352	21.60%	44.00%	81
RetinaNet	2018	ResNet-101-FPN	400	31.90%	49.50%	12
YOLOv3	2018	DarkNet-53	320	28.20%	51.50%	45
CenterNet	2019	Hourglass-104	512	42.10%	61.10%	7.8
EfficientDet-D2	2020	Efficient-B2	768	43.00%	62.30%	41.7
YOLOv4	2020	CSPDarkNet-53	512	43.00%	64.90%	31
Swin-L	2021	HTC++	-	57.70%	-	-

[a]Models marked with * are compared on PASCAL VOC 2012, while others on MS COCO.Rows colored gray are real-time detectors (>30 FPS).

Taken from paper https://arxiv.org/pdf/2104.11892.pdf

Model Development

For the purposes of this tutorial we will not be creating a training job from scratch, but rather we will **reuse** one of the pre-trained models provided by TensorFlow on their TensorFlow 2 Detection Model Zoo.

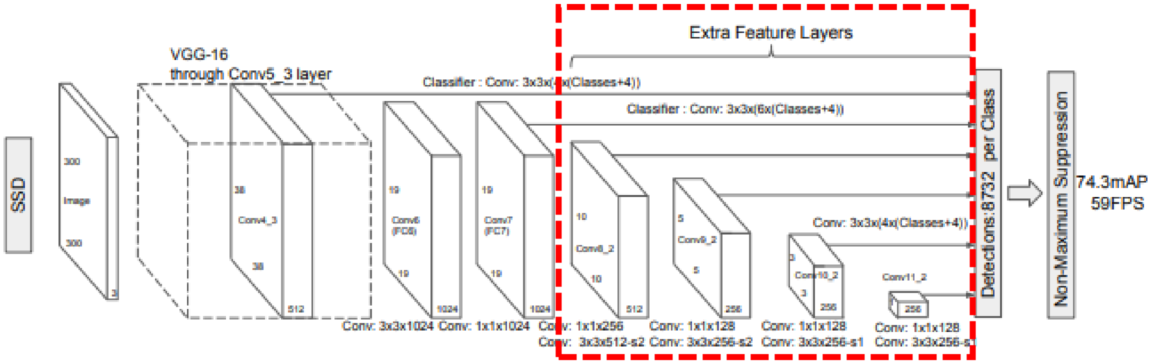
This pre-trained model will be useful for initializing our models when training on our datasets.

The model we shall be using in our examples is the SSD ResNet50 V1 FPN 640x640 model, since it provides a relatively good trade-off between performance and speed. However, there exist a number of other models we can use, all of which are listed in: TensorFlow 2 Detection Model Zoo.

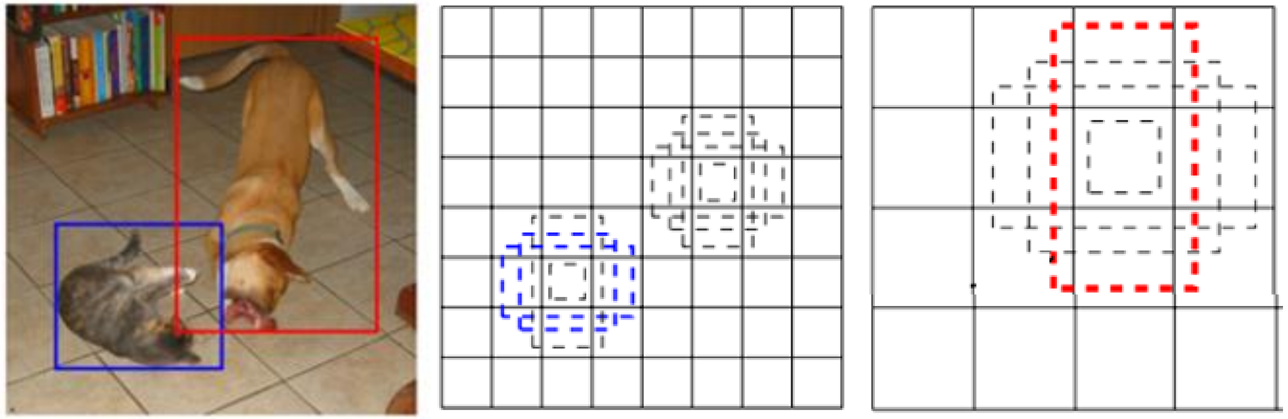
The SSD detector is easy to train and integrate into software systems that require an object detection component. In comparison to other single-stage methods, SSD has much better accuracy, even with smaller input image sizes.

How Single Short Detector Works

The SSD approach is based on a feed-forward convolutional network that produces a fixed-size collection of default bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections.



SSD model adds several feature layers to the end of a base network, which predict the offsets to default boxes of different scales and aspect ratios and their associated confidences.



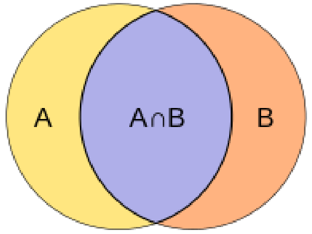
(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

SSD Training Flow

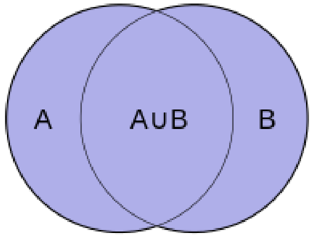
- SSD only needs an input image and ground truth boxes for each object during training. (Image (a) above)
- In a convolutional fashion, we evaluate a small set (e.g. 4) of default boxes of different aspect ratios at each location in several feature maps with different scales (e.g. 8 × 8 and 4 × 4 in image (b) and (c) above)
- During training we need to determine which default boxes correspond to a ground truth detection and train the network accordingly.
- For each ground truth box we are selecting from default boxes that vary over location, aspect ratio, and scale.
- We begin by matching each ground truth box to the default box with the best Jaccard overlap.
- For example, we have matched two default boxes with the cat and one with the dog, which are treated as positives and the rest as negatives.

Picture Credit : google search

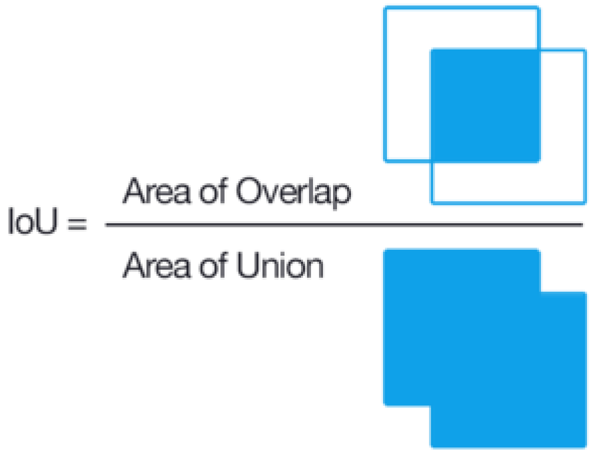
Jaccard overlap

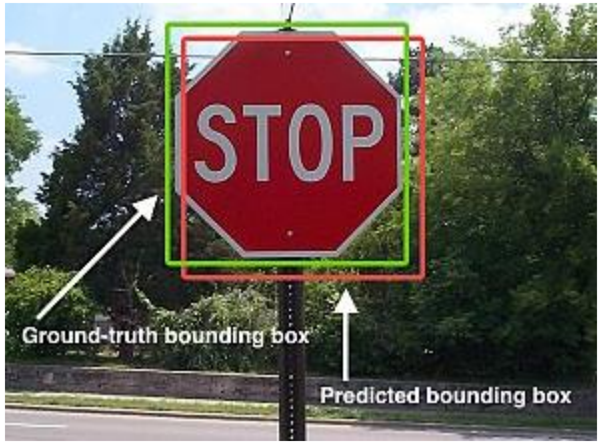


Intersection of Set A & B



Union of Set A & B





Jaccard similarity coefficient is the ratio of **Intersection over Union(IoU)**

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

Note that by design, $0 \leq J(A,B) \leq 1.$

If *A* and *B* are both empty, define *J*(*A*,*B*) = 1.

Higher the IoU, higher the overlap of default boxes with ground truth box

Model Development – pipeline.config

Next

Configuring the Object Detection Training Pipeline

The TensorFlow Object Detection API uses protobuf files to configure the training and evaluation process.

At a high level, the config file is split into 5 parts

1. **The model configuration**. This defines what type of model will be trained (ie. meta-architecture, feature extractor).
2. **The train_config**, which decides what parameters should be used to train model parameters (ie. SGD parameters, input pre-processing and feature extractor initialization values).
3. **The eval_config**, which determines what set of metrics will be reported for evaluation.
4. **The train_input_config**, which defines what dataset the model should be trained on.
5. **The eval_input_config**, which defines what dataset the model will be evaluated on. Typically this should be different than the training input dataset.

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

pipeline.config : num_class, image_resizer

model config

```
ssd {
  num_classes: 5

  image_resizer {
   fixed_shape_resizer {
    height: 640
    width: 640
   }
  }
```

Number of classes to predict

Desired height of image in pixels

Desired width of image in pixels

pipeline.config : feature_extractor

model config

Extracts features for different models.

```
feature_extractor {
  type: "ssd_resnet101_v1_fpn_keras"
  depth_multiplier: 1.0
  min_depth: 16
  conv_hyperparams {
   regularizer {
    l2_regularizer {
     weight: 0.00039999998989515007
    }
   }
   initializer {
    truncated_normal_initializer {
     mean: 0.0
     stddev: 0.029999999329447746
    }
   }

   activation: RELU_6
   batch_norm {
    decay: 0.996999979019165
    scale: true
    epsilon: 0.0010000000474974513
   }
  }
  override_base_feature_extractor_hyperparams: true
```

Depth multiplier for convolutional layers.

Minimum depth for convolutional layers

containing hyperparameters for convolution ops

Regularizer for the weights of the convolution ops.

Initializer for the weights of the convolution op.

Type of activation to apply after convolution

Batch normalization is intended to solve the following problem: Changes in model parameters during learning change the distributions of the outputs of each hidden layer. This means that later layers need to adapt to these (often noisy) changes during training.

Whether to override hyperparameters of the base feature extractor with the one from`conv_hyperparams_fn`

pipeline.config : box_coder

model config

Box coders convert between coordinate frames, namely image-centric (with (0,0) on the top left of image) and anchor-centric (with (0,0) being defined by a specific anchor).

```
box_coder {
  faster_rcnn_box_coder {
    y_scale: 10.0
    x_scale: 10.0
    height_scale: 5.0
    width_scale: 5.0
  }
}
```

Box_coder types:

- faster_rcnn
- keypoint
- mean_stddev
- square

x, y, w, h denote the box's center coordinates, width and height respectively.

pipeline.config : matcher

model config

This class takes a similarity matrix and matches columns to rows based on the maximum value per column. One can specify matched_thresholds and to prevent columns from matching to rows (generally resulting in a negative training example) and unmatched_theshold to ignore the match (generally resulting in neither a positive or negative training example).

```
matcher {
    argmax_matcher {
      matched_threshold: 0.5
      unmatched_threshold: 0.5
      ignore_thresholds: false
      negatives_lower_than_unmatched: true
      force_match_for_each_row: true
      use_matmul_gather: true
    }
  }
```

pipeline.config : anchor_generator

model config

```
anchor_generator {
    multiscale_anchor_generator {
      min_level: 3
      max_level: 7
      anchor_scale: 4.0
      aspect_ratios: 1.0
      aspect_ratios: 2.0
      aspect_ratios: 0.5
      scales_per_octave: 2
    }
  }
```

generates a large number of anchor boxes in a range of shapes and sizes, in many locations of the image. The detection algorithm then incrementally offsets the anchor box closest to the **ground truth(GT)** until it (closely) matches. We can specify the variety of and position of these anchor boxes in the anchor_generator config section.

pipeline.config : box_predictor

model config Box predictor for object detectors

```
box_predictor {
    weight_shared_convolutional_box_predictor {
     conv_hyperparams {
      regularizer {
       l2_regularizer {
        weight: 0.00039999998989515007
       }
      }
      initializer {
       random_normal_initializer {
        mean: 0.0
        stddev: 0.009999999776482582
       }
      }
      activation: RELU_6
      batch_norm {
       decay: 0.996999979019165
       scale: true
       epsilon: 0.0010000000474974513
      }
     }
     depth: 256
     num_layers_before_predictor: 4
     kernel_size: 3
     class_prediction_bias_init: -4.599999904632568
    }
}
```

Box predictors are classes that take a high level
image feature map as input and produce two predictions,
(1) a tensor encoding box locations, and
(2) a tensor encoding classes for each box.
These components are passed directly to loss functions
in our detection models.
These modules are separated from the main model since the
same
few box predictor architectures are shared across many
models.

pipeline.config : post_processing

model config

Configuration proto for non-max-suppression operation on a batch of detections.

```
post_processing {
    batch_non_max_suppression {
      score_threshold: 9.99999993922529e-09
      iou_threshold: 0.6000000238418579
      max_detections_per_class: 100
      max_total_detections: 100
      use_static_shapes: false
    }
   score_converter: SIGMOID
  }
```

Depth Scalar threshold for score (low scoring boxes are removed).

Scalar threshold for IOU (boxes that have high IOU overlap with previously selected boxes are removed).

Maximum number of detections to retain per class.

Maximum number of detections to retain across all classes

Whether to use the implementation of NMS that guarantees static shapes.

Enum to specify how to convert the detection scores.

- IDENTITY = 0; // Input scores equals output scores.
- SIGMOID = 1; // Applies a sigmoid on input scores.
- SOFTMAX = 2;// Applies a softmax on input scores

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

pipeline.config : batch_size,data_augmentation_option

train_config Configuring Detection Model training jobs

```
train_config {
 batch_size: 8
 data_augmentation_options {
  random_horizontal_flip {
  }
 }
 data_augmentation_options {
  random_crop_image {
   min_object_covered: 0.0
   min_aspect_ratio: 0.75
   max_aspect_ratio: 3.0
   min_area: 0.75
   max_area: 1.0
   overlap_thresh: 0.0
  }
 }
```

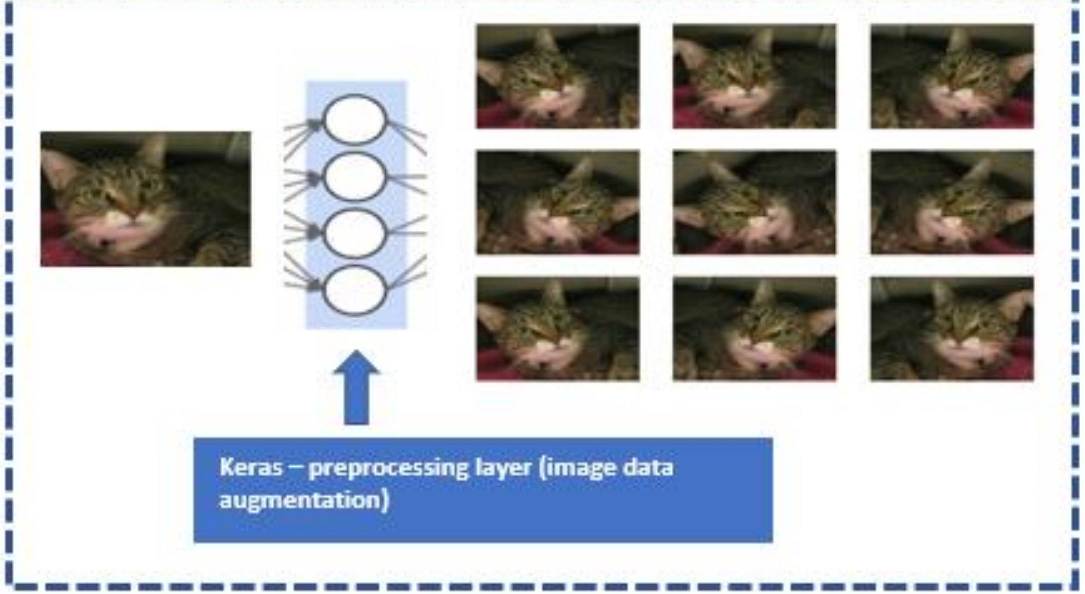
Batch size to use for training

The **data_augmentation_options** in train_config can be used to specify how training data can be modified. This field is **optional**.

Possible Image Augmentation

- normalize_image
- random_horizontal_flip
- random_pixel_value_scale
- random_image_scale
- random_rgb_to_gray
- random_adjust_brightness
- random_adjust_contrast
- random_adjust_hue
- random_adjust_saturation
- random_distort_color
- random_jitter_boxes
- random_crop_image
- random_pad_image
- random_crop_pad_image
- random_crop_to_aspect_ratio
- random_black_patches
- random_resize_method
- scale_boxes_to_pixel_coordinates
- resize_image
- subtract_channel_mean
- ssd_random_crop
- ssd_random_crop_pad
- ssd_random_crop_fixed_aspect_ratio

Image data augmentation is a technique to increase the diversity of training set by applying random (but realistic) transformations such as image rotation.



Picture Credit : google search

pipeline.config: fine_tune_checkpoint,num_steps,startup_delay_steps,fine_tune_checkpoint_type

testing_config

Path of the checkpoint of pre-trained model

```
fine_tune_checkpoint: "/content/gdrive/MyDrive/CustomObjectDetection/workspace/training_demo/pre-trained-models/ssd_resnet101_v1_fpn_640x640_coco17_tpu-8/checkpoint/ckpt-0"
  num_steps: 200000
  startup_delay_steps: 0.0
  replicas_to_aggregate: 8
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  fine_tune_checkpoint_type: "detection"
  use_bfloat16: false
  fine_tune_checkpoint_version: V2
}
```

This option Number of steps to train the Detection Model for. If 0, will train the model indefinitely.

Number of training steps between replica startup.This flag must be set to 0 if sync_replicas is set to true.

Number of replicas to aggregate before making parameter updates

Maximum number of boxes used during training.

This option controls how variables are restored from the (pre-trained)
This option is typically used when we **want to use a pre-trained detection model and train on a new dataset**

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

- The TensorFlow Object Detection API accepts inputs in the TFRecord file format.
- Users must specify :
  - locations of both the training and evaluation files.
  - location of a label map, which define the mapping between a class id and class name.
- The label map should be identical between training and evaluation datasets.

pipeline.config: label_map_path, tf_record_input_reader

Path to **LabelMap pbtxt** file specifying the mapping from string labels to integer ids

```
train_input_reader {
  label_map_path: "/content/gdrive/MyDrive/CustomObjectDetection/workspace/training_demo/annotations/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "/content/gdrive/MyDrive/CustomObjectDetection/workspace/training_demo/annotations/train.record"
  }
}
```

Path to **train.record** dataset

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

pipeline.config: eval_config

The TensorFlow Object Detection API currently supports three evaluation protocols, that can be configured in Eval Config by setting metrics_set to the corresponding value :

- pascal_voc_detection_metrics
- weighted_pascal_voc_detection_metrics
- pascal_voc_instance_segmentation_metrics
- coco_detection_metrics

The parameter metrics_set indicates which metrics to run during evaluation.

```
eval_config {
 metrics_set: "coco_detection_metrics"
 }
```

The COCO metrics are the official detection metrics used to score the COCO competition and are similar to Pascal VOC metrics but have a slightly different implementation and report additional statistics such as mAP at IOU thresholds of .5:.95, and precision/recall statistics for small, medium, and large objects.

A skeleton configuration file is shown below

```
model {
  (... Add model config here...)
}

train_config : {
  (... Add train_config here...)
}

train_input_reader: {
  (... Add train_input configuration here...)
}

eval_config: {
  (... Add eval_config here...)
}

eval_input_reader: {
  (... Add eval_input configuration here...)
}
```

pipeline.config: label_map_path, tf_record_input_reader

Path to **LabelMap pbtxt** file specifying the mapping from string labels to integer ids

```
eval_input_reader {
  label_map_path:
"/content/gdrive/MyDrive/CustomObjectDetection/workspace/training_demo/annotations/labelmap.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "/content/gdrive/MyDrive/CustomObjectDetection/workspace/training_demo/annotations/test.record"
  }
}
```

Path to **test.record** dataset

Model Development – model_main_tf2.py

```
model_lib_v2.train_loop(
    pipeline_config_path=FLAGS.pipeline_config_path,
    model_dir=FLAGS.model_dir,
    train_steps=FLAGS.num_train_steps,
    use_tpu=FLAGS.use_tpu,
    checkpoint_every_n=FLAGS.checkpoint_every_n,
    record_summaries=FLAGS.record_summaries)
```

This method:

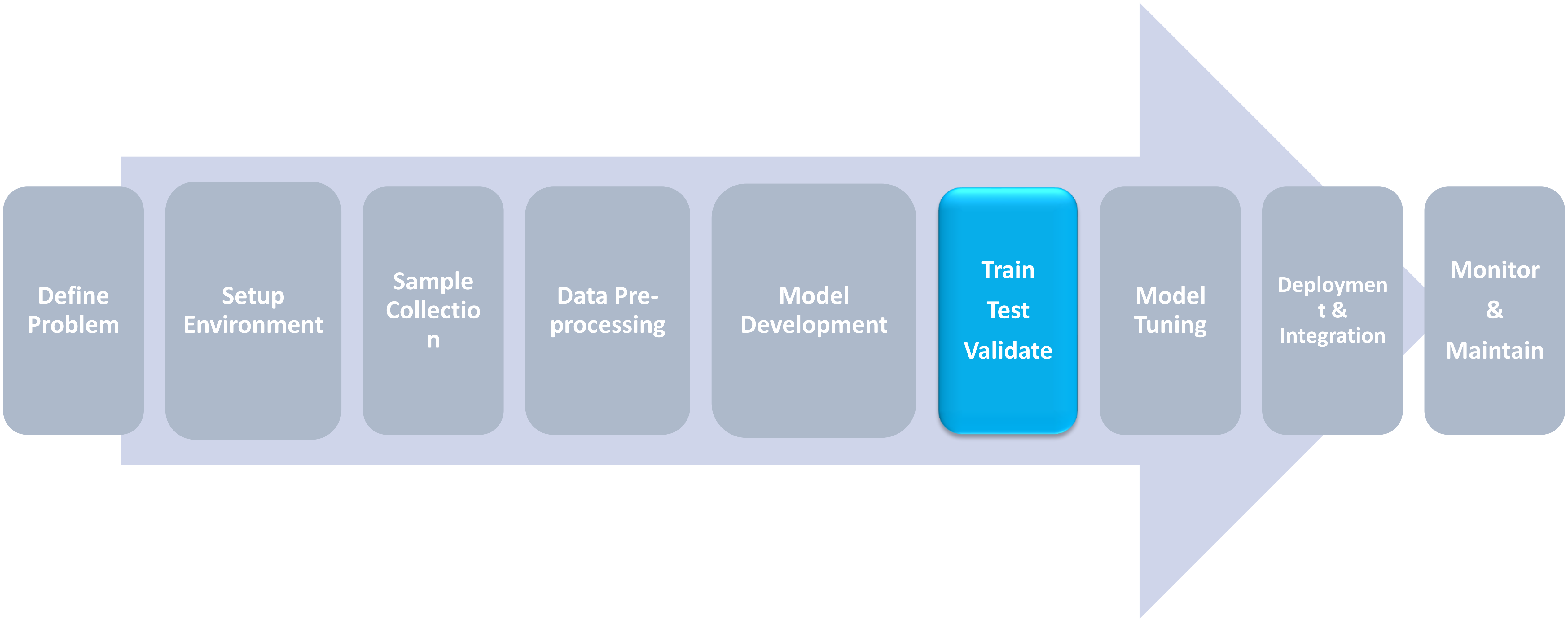
- Processes the pipeline configs
- Builds the model & optimizer
- Gets the training input data
- Loads a fine-tuning detection or classification checkpoint if requested
- Loops over the train data, executing distributed training steps inside tf.functions.
- Checkpoints the model every `checkpoint_every_n` training steps.
- Logs the training metrics as TensorBoard summaries.

Args:

pipeline_config_path: A path to a pipeline config file.
model_dir: The directory to save checkpoints and summaries to.
config_override: A pipeline_pb2.TrainEvalPipelineConfig text proto to override the config from `pipeline_config_path`.
train_steps: Number of training steps. If None, the number of training steps is set from the `TrainConfig` proto.
use_tpu: Boolean, whether training and evaluation should run on TPU.
save_final_config: Whether to save final config (obtained after applying overrides) to `model_dir`.
checkpoint_every_n: Checkpoint every n training steps.
checkpoint_max_to_keep: int, the number of most recent checkpoints to keep in the model directory.
record_summaries: Boolean, whether or not to record summaries defined by the model or the training pipeline. This does not impact the summaries of the loss values which are always recorded. Examples of summaries that are controlled by this flag include:
- Image summaries of training images.
- Intermediate tensors which maybe logged by meta architectures.

performance_summary_exporter: function for exporting performance metrics.
num_steps_per_iteration: int, The number of training steps to perform in each iteration.

https://github.com/tensorflow/models/blob/beec3163b3aaacf6cb488fb843d75f66e367e984/research/object_detection/model_lib_v2.py#L206


Define Problem
Setup Environment
Sample Collectio n
Data Pre-processing
Model Development
Train Test Validate
Model Tuning
Deploymen t & Integration
Monitor & Maintain

Training the model

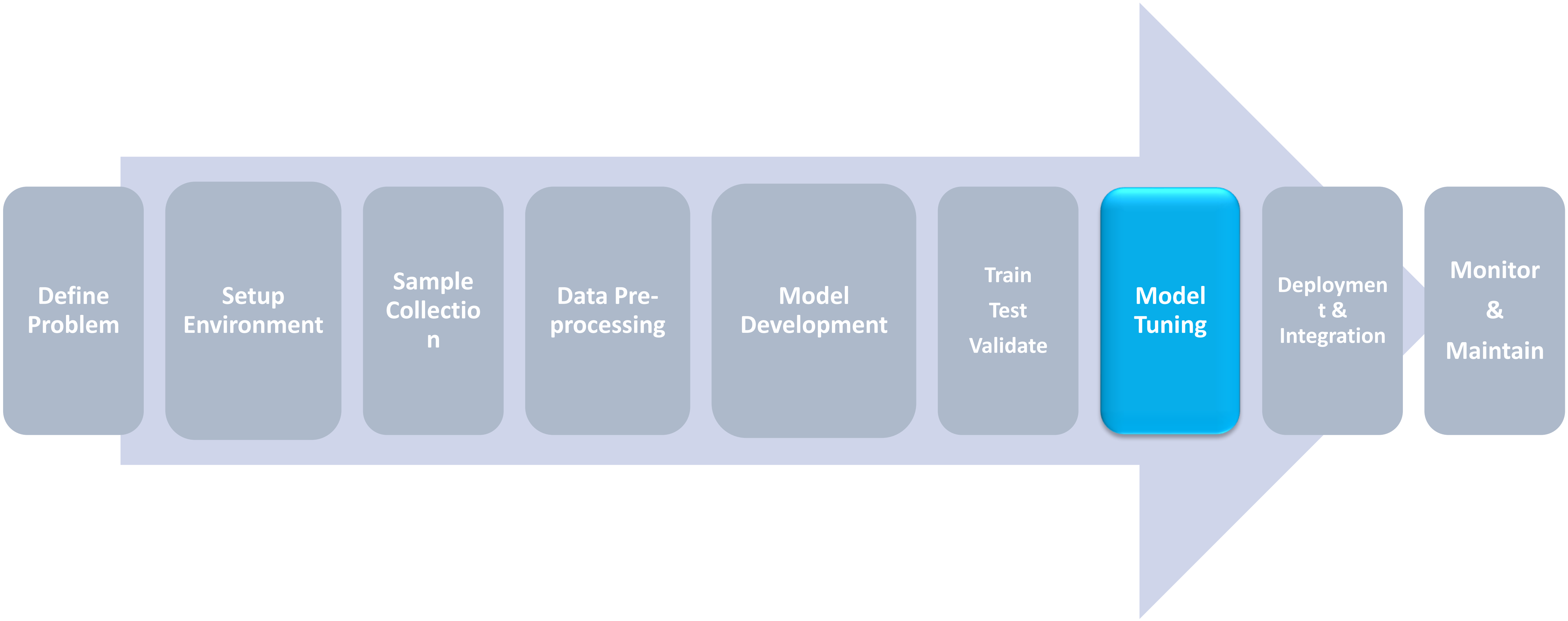
Copy **model_main_tf2.py** from
/content/gdrive/MyDrive/CustomObjectDetection/models/research/object_detection/
to our workspace
/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection

```
cd /content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection
```

```
cp /content/gdrive/MyDrive/CustomObjectDetection/models/research/object_detection/model_main_tf2.py .
```

Finally run the training from our workspace

```
!python model_main_tf2.py --
model_dir=/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/models/my_ssd_res
net50_v1_fpn --
pipeline_config_path=/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/models
/my_ssd_resnet50_v1_fpn/pipeline.config
```


Define Problem
Setup Environment
Sample Collectio n
Data Pre-processing
Model Development
Train
Test
Validate
Model Tuning
Deploymen t & Integration
Monitor
&
Maintain

Tunning the model

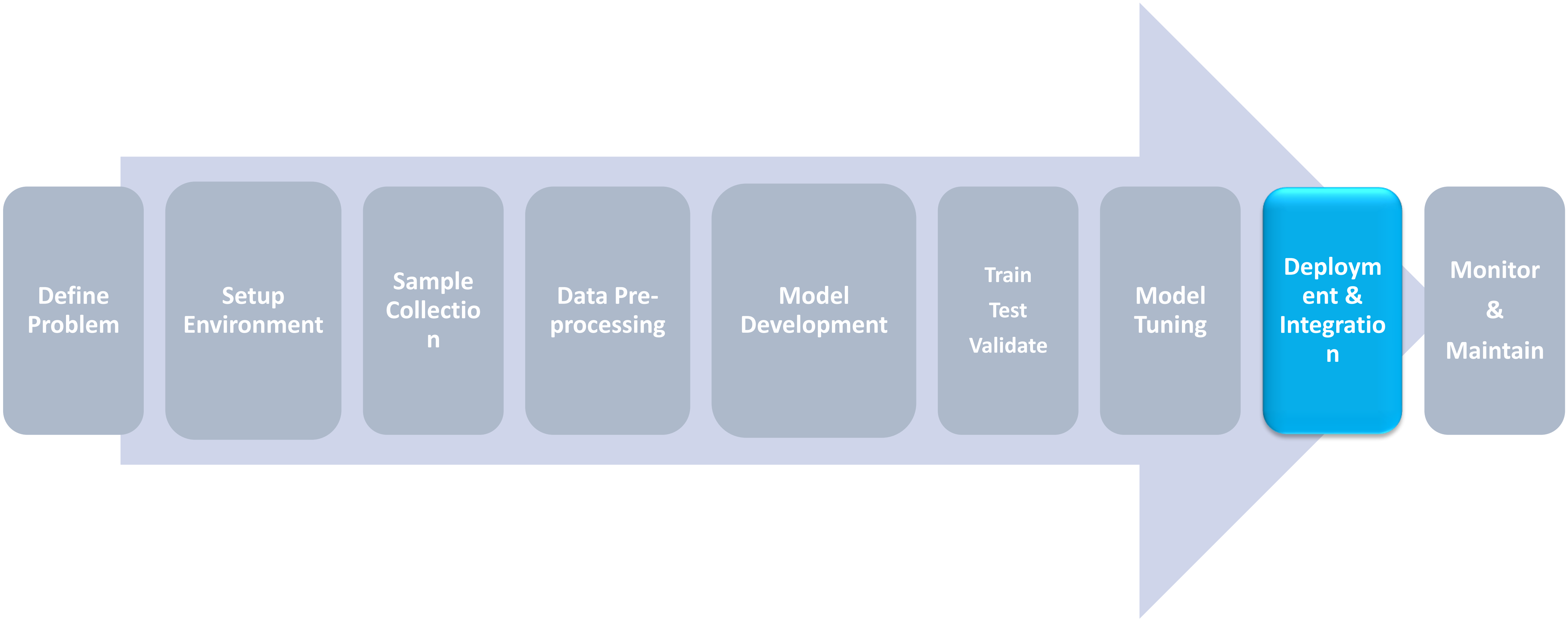
Most of the hyperparameters are already tunned because our base model is a pre-trained model.

But we still have two important hyperparameters which can be tunned in pipeline.config

```
batch_size: 8          # Increase/Decrease this value depending on the available
                       memory (Higher values require more memory and vice-versa)
num_steps: 2000
```

num_steps sets how many trainings steps we are going to use based on the batch_size.
A batch_size of 8 and num_steps set to 2000 will be equal to processing 16000 images in total.

pipeline.config path :
/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/models/my_ssd_resnet50_v1_fpn/pipeline.config


Define Problem
Setup Environment
Sample Collectio n
Data Pre-processing
Model Development
Train
Test
Validate
Model Tuning
Deploym ent & Integratio n
Monitor
&
Maintain

Deploy trained finger counting detection model

Install TensorFlow & TF Object Detection API

Configure the path of trained Model & corresponding labelmap.pbtxt

```
# Path of the trained model directory. Set this path till my_model folder
TRAINED_MODEL_DIR = '/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/exported-models/my_model'

# Path of labelmap.pbtxt file
LABELMAP_PATH = '/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/annotations/labelmap.pbtxt'
```

Loading labelmap.pbtxt file

```
# Loading Label map(labelmap.pbtxt)

category_index = label_map_util.create_category_index_from_labelmap(LABELMAP_PATH, use_display_name=True)
```

Deploy trained finger counting detection model

Loading the saved(trained) model from exported directory

There are two sets of APIs available:

High level : tf.keras.models.load_model (if the model was saved as Keras model)

Low level : tf.saved_model.load

tf.saved_model.load(
model_dir, tags=None, options=None
)

Since it is an API that is on the lower level (and hence has a wider range of use cases), it returns an **object** that contain **functions** that can be used to do inference.

The loaded object may contain multiple functions, each associated with a key. The "**serving_default**" is the default key for the inference function with a saved Keras model. To do an inference with this function:

```
# Loading saved model and building detection function
detect_fn = tf.saved_model.load(SAVED_MODEL_PATH)
```

Running our trained model to detect finger counting

Loading an image for detection

```
# Loading an image for detection
IMAGE_PATHS = '/content/gdrive/MyDrive/CustomObjectDetection/workspace/finger_count_detection/images/test/20211027_112622.jpg'
image = cv2.imread(IMAGE_PATHS)
```

Convert this image into tensors

```
# The input needs to be a tensor, converting it using `tf.convert_to_tensor`.
input_tensor = tf.convert_to_tensor(image)
```

Running our trained model to detect finger counting

Draw bounding boxes on an image with formatted scores and label names

```
visualize_boxes_and_labels_on_image_array(
    image,
    boxes,
    classes,
    scores,
    category_index,
    instance_masks=None,
    instance_boundaries=None,
    keypoints=None,
    use_normalized_coordinates=False,
    max_boxes_to_draw=20,
    min_score_thresh=.5,
    agnostic_mode=False,
    line_thickness=4,
    groundtruth_box_visualization_color='black',
    skip_scores=False,
    skip_labels=False):
```

This function groups boxes that correspond to the same location and creates a display string for each detection and overlays these on the image.

Note that this function modifies the image in place, and returns that same image.

Args:

- **image**: uint8 numpy array with shape (img_height, img_width, 3)
- **boxes**: a numpy array of shape [N, 4]
- **classes**: a numpy array of shape [N]. Note that class indices are 1-based, and match the keys in the label map.
- **scores**: a numpy array of shape [N] or None. If scores=None, then this function assumes that the boxes to be plotted are groundtruth boxes and plot all boxes as black with no classes or scores.
- **category_index**: a dict containing category dictionaries (each holding category index \`id\` and category name \`name\`) keyed by category indices.
- **instance_masks**: a numpy array of shape [N, image_height, image_width] with values ranging between 0 and 1, can be None.
- **instance_boundaries**: a numpy array of shape [N, image_height, image_width] with values ranging between 0 and 1, can be None.
- **keypoints**: a numpy array of shape [N, num_keypoints, 2], can be None
- **use_normalized_coordinates**: whether boxes is to be interpreted as normalized coordinates or not.
- **max_boxes_to_draw**: maximum number of boxes to visualize. If None, draw all boxes.
- **min_score_thresh**: minimum score threshold for a box to be visualized
- **agnostic_mode**: boolean (default: False) controlling whether to evaluate in class-agnostic mode or not. This mode will display scores but ignore classes.
- **line_thickness**: integer (default: 4) controlling line width of the boxes.
- **groundtruth_box_visualization_color**: box color for visualizing groundtruth boxes
- **skip_scores**: whether to skip score when drawing a single detection
- **skip_labels**: whether to skip label when drawing a single detection

Returns: uint8 numpy array with shape (img_height, img_width, 3) with overlaid boxes

```
visualize_boxes_and_labels_on_image_array(
      image_with_detections,
      detections['detection_boxes'],
      detections['detection_classes'],
      detections['detection_scores'],
      category_index,
      use_normalized_coordinates=True,
      max_boxes_to_draw=200,
      min_score_thresh=0.5,
      agnostic_mode=False)
```

Running our trained model to detect finger counting

Overlay labeled boxes on an image with formatted scores and label names

```
visualize_boxes_and_labels_on_image_array(
    image,
    boxes,
    classes,
    scores,
    category_index,
    instance_masks=None,
    instance_boundaries=None,
    keypoints=None,
    use_normalized_coordinates=False,
    max_boxes_to_draw=20,
    min_score_thresh=.5,
    agnostic_mode=False,
    line_thickness=4,
    groundtruth_box_visualization_color='black',
    skip_scores=False,
    skip_labels=False):
```

This function groups boxes that correspond to the same location and creates a display string for each detection and overlays these on the image.

Note that this function modifies the image in place, and returns that same image.

```
viz_utils.visualize_boxes_and_labels_on_image_array(
      image_with_detections,
      detections['detection_boxes'],
      detections['detection_classes'],
      detections['detection_scores'],
      category_index,
      use_normalized_coordinates=True,
      max_boxes_to_draw=200,
      min_score_thresh=0.5,
      agnostic_mode=False)
```

Downloads

Download complete notebook from my google colab

https://colab.research.google.com/drive/1b9zZRc_Rs9N4GwoSn_BNmE392L4hkPYO?authuser=1#scrollTo=pNVwlSCq9pr1

GitHub

https://github.com/datasciencechampion/CustomObjectDetection

References

White papers

ModelPerformanceWhitepaper_Google_For_ObjectDetection

SSD(Single_Shot_MultiBox_Detector)_whitepaper

A Survey of Modern Deep Learning based Object

TensorFlow GitHub for Object Detection Model

https://github.com/tensorflow/models/tree/master/research/object_detection

TensorFlow 2 Object Detection API tutorial

https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/index.html

Thank You

Send you feedback, queries to:

Pramod Singh
datasciencechampion@gmail.com