Chapter 1 :

Introduction

Chapter 2 :

conception

Chapter 3:

Implementation

3.1 Introduction :

This chapter presents the implementation of a mobile app leveraging AI to translate sign language, facilitating communication between sign language users and non-sign language speakers. It covers the selection and training of the AI model, integration into the app, user interface design, testing and evaluation, deployment considerations, and potential future enhancements. The chapter highlights the technical steps involved in developing a mobile app that provides real-time sign language translation on-the-go.

3.2 Artificial intelligence :

AI (Artificial Intelligence) is a field of computer science that focuses on developing intelligent systems capable of performing tasks that typically require human intelligence. It involves algorithms and models that enable machines to learn from data, reason, make decisions, and solve complex problems. AI has widespread applications across various domains, revolutionizing industries and driving advancements in areas such as robotics, natural language processing, computer vision, and more.

3.3 Software environment :

3.3.1 Visual Studio Code (VS Code) :

is a lightweight and fast code editor that supports multiple platforms. It offers extensive language support and a rich ecosystem of extensions

3.3.2 Google Colab:

Google Colab is an online platform for writing and running Python code collaboratively. It offers a Jupyter Notebook interface, free computing resources, pre-installed libraries, and Google Drive integration. It facilitates collaborative coding and is popular for tasks like machine learning and data analysis.

3.4 Programming languages :

3.4.1 Python : Python is chosen for machine learning due to its extensive libraries (e.g., **TensorFlow**, **PyTorch**) tailored for ML tasks. Its simplicity and readability make it easy to express complex ideas, while its interactive nature enables rapid prototyping. The broad community support, integration capabilities, and excellent data analysis and visualization tools further reinforce Python's suitability for ML. Additionally, Python's industry adoption ensures ample resources and opportunities for ML practitioners.

3.5 Frameworks & libraries :

3.5.1 Tensorflow :

TensorFlow is an open-source machine learning framework developed by Google that provides a comprehensive ecosystem for building and deploying machine learning models, utilizing computational graphs to represent operations and data flow. It offers high-level and lower-level APIs, supports distributed computing, and finds applications in diverse domains such as computer vision, natural language processing, and reinforcement learning.

3.5.2 Keras :

Keras is a high-level deep learning framework that runs on top of TensorFlow, providing a user-friendly interface for building and training neural networks with concise and intuitive code.

3.5.3 Sklearn :

scikit-learn (sklearn) is a widely-used open-source machine learning library for Python that offers a range of tools and algorithms for various tasks, such as classification, regression, clustering, and dimensionality reduction, providing a consistent and user-friendly API.

3.5.4 Matplotlib:

Matplotlib is a popular data visualization library for Python that provides a comprehensive set of tools for creating static, animated, and interactive visualizations, allowing users to generate high-quality plots, charts, and figures for data exploration and presentation.

3.5.5 Pandas :

pandas is a powerful open-source data manipulation and analysis library for Python, offering flexible data structures and functions to efficiently manipulate and analyze structured data, such as tabular data or time series, providing capabilities for data cleaning, preprocessing, filtering, aggregation, and more, making it a go-to tool for data manipulation tasks in data science and analysis workflows.

3.5.6 OpenCV :

pandas is a powerful open-source data manipulation and analysis library for Python, offering flexible data structures and functions to efficiently manipulate and analyze structured data, such as tabular data or time series, providing capabilities for data cleaning, preprocessing, filtering, aggregation, and more, making it a go-to tool for data manipulation tasks in data science and analysis workflows.

3.5.7 xml.etree.elementree:

xml.etree.ElementTree is a built-in Python library that provides a lightweight and efficient API for parsing and manipulating XML (eXtensible Markup Language) data, allowing users to easily read, write, and modify XML documents, extract information from XML structures, and create new XML elements and trees.

3.6 Dataset :

The American Sign Language Letters dataset is an object detection dataset of each ASL letter with a bounding box. David Lee, a data scientist focused on accessibility, curated and released the dataset for public use.

We got our dataset from kaggle , the dataset has 1584 images and 1584 annotation ,and two csv files (train,test).

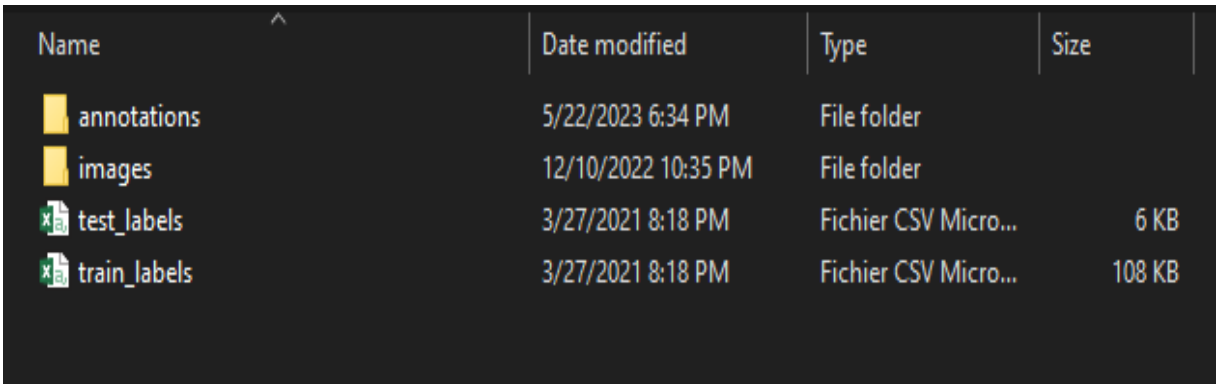


Figure 1.1: Dataset files

3.7 Used Method to detect and translate sign language :

In our project, we utilized object detection to recognize sign language, and we opted for the YOLO (You Only Look Once) algorithm due to its exceptional speed compared to other algorithms. By employing a single neural network, YOLO efficiently predicts bounding boxes and class probabilities for multiple

objects in real-time. This choice of algorithm was essential for achieving timely and accurate sign language recognition, making it suitable for applications where swift object detection is crucial.

To implement YOLO (You Only Look Once) algorithm, the following steps are typically involved:

- Intialisation
- HyperParameters
- Model Implementation
- Data Preprocessing
- Loss function Implementation
- Training
- Evaluation

3.7.1 Initialisation

In the initialization step, we start by importing the necessary libraries and configuring the training device. Additionally, we establish the connection between Google Colab and Google Drive to facilitate data access and storage during the training process.

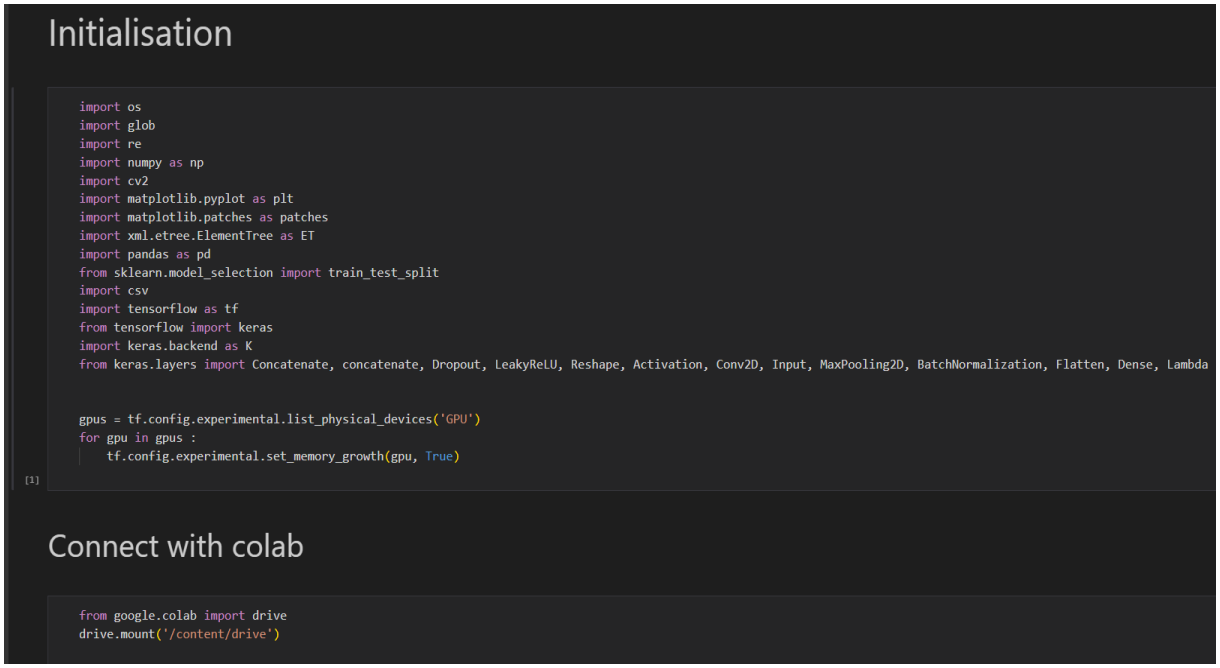


Figure 2. Initialisation

3.7.2 Hyperparameters :

During the hyperparameter tuning step, we focus on determining and fine-tuning the optimal parameters for our model to achieve the highest possible accuracy.

HYPERPRAMETERS

```
# Parameters

LABELS           = LABELS=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
IMAGE_H, IMAGE_W = 256, 256
GRID_H,  GRID_W  = 8,8 # GRID size = IMAGE size / 32
BOX              = 5
CLASS            = len(LABELS)
SCORE_THRESHOLD  = 0.5
IOU_THRESHOLD    = 0.45
ANCHORS          = [0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282, 3.52778, 9.77052, 9.16828]

TRAIN_BATCH_SIZE = 10
VAL_BATCH_SIZE   = 10
EPOCHS           = 250

columns=["path","width","height","label","xmin","ymin","xmax","ymax"]
LAMBDA_NOOBJECT  = 1
LAMBDA_OBJECT    = 5
LAMBDA_CLASS     = 1
LAMBDA_COORD     = 1


# Train and validation directory on colab
PATH='datasets/obd/hand sign dataset'
```

Python

Figure 3 Hyperparameters

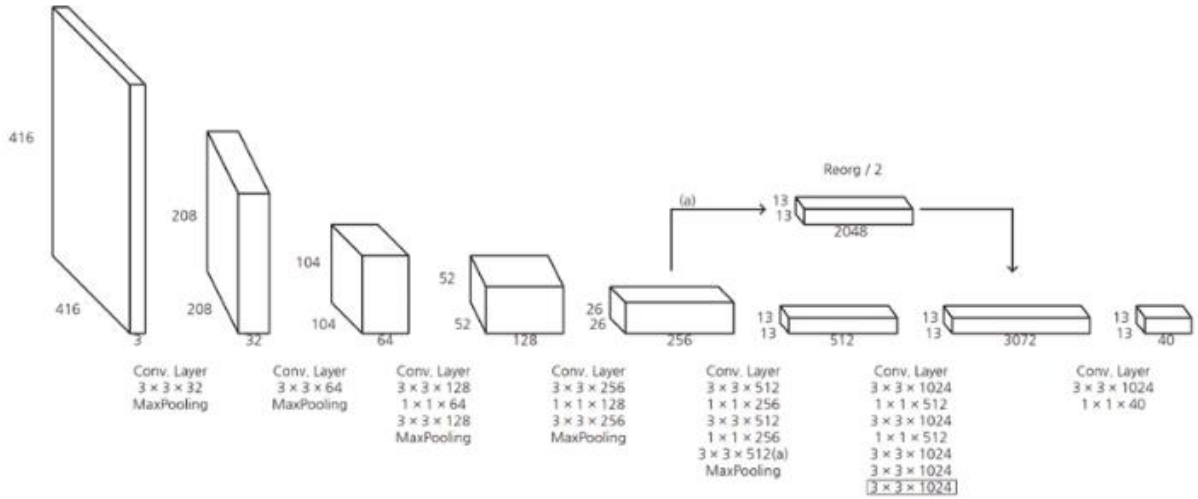
Note :

We adopted our hyperparameter values from the YOLO v2 paper for optimal configuration.

3.7.3 Build the Model :

The Build Model step involves implementing the **YOLO v2** model using the combination of **Keras** and **TensorFlow**. By leveraging the powerful capabilities of these frameworks, we construct the architecture of the YOLO v2 model, defining the layers, connections, and parameters required for object detection. This step establishes the foundation for subsequent training and inference stages.

3.7.3.1 Model architecture :



Here's a simplified breakdown of the architecture :

1. **Convolutional Layers:** yolo v2 has 23 convolutional layers. Some of these layers are followed by a max-pooling layer for downsampling. The network mainly uses 3x3 filters and employs 1x1 reduction layers to reduce the depth of the feature maps, which helps in controlling the computational complexity.

2. **Normalization:** Batch normalization is used after each convolutional layer, which speeds up learning and provides some level of regularization.
3. **Activation Function:** After each batch normalization, a Leaky ReLU activation function is used(alpha=0.1).
4. **MaxPooling2D:** A max pooling operation for spatial data, used to reduce the spatial dimensions (width, height) of the input volume.
5. **SpaceToDepth:** Rearranges blocks of spatial data (height and width) into depth. Used for the "passthrough" layer, which helps the network use fine-grained features.
6. **Pass-Through Layer:** This is a unique layer to the YOLO v2 architecture. It connects the output from an earlier layer (Layer 13) to a deeper layer in the network. This helps the network make use of finer-grained features for detecting smaller objects.
7. **Detection Layer:** The final layer uses a 1x1 convolution to combine the features from the previous layers and make predictions. For each grid cell, it predicts a fixed number of bounding boxes. Each prediction includes the box's coordinates (x, y, width, height), a confidence score, and class probabilities.
8. **Output**: The output is a tensor of shape SxSx155 (for 26 classes and 5 bounding boxes), which encodes the bounding box coordinates, confidence scores, and class probabilities.

```
# Layer 13
x = Conv2D(512, (3,3), strides=(1,1), padding='same', name='conv_13', use_bias=False)(x)
x = BatchNormalization(name='norm_13')(x)
x = LeakyReLU(alpha=0.1)(x)

skip_connection = x
```

Figure 4.conv layer with activation

```
# Layer 21
skip_connection = Conv2D(64, (1,1), strides=(1,1), padding='same', name='conv_21', use_bias=False)(skip_connection)
skip_connection = BatchNormalization(name='norm_21')(skip_connection)
skip_connection = LeakyReLU(alpha=0.1)(skip_connection)

skip_connection = SpaceToDepth(block_size=2)(skip_connection)

x = concatenate([skip_connection, x])
```

Figure 5.skip connection

```
# Layer 23
x = Conv2D(BOX * (4 + 1 + CLASS), (1,1), strides=(1,1), padding='same', name='conv_23')(x)
output = Reshape((GRID_W, GRID_H, BOX, 4 + 1 + CLASS))(x)

model = keras.models.Model(input_image, output)
```

Figure 6.outputLayer

3.7.3.2 Yolo Weights :

In our model, we have approximately 50 million parameters, which include the weights associated with the various layers and connections within the YOLO architecture.

```
...
Total params: 50,706,811
Trainable params: 50,686,139
Non-trainable params: 20,672
_________________________________________________________________
```

Figure 7. weights Number

To overcome the challenge of training YOLO v2 from scratch due to resource limitations, we utilized pre-trained weights obtained from the Darknet framework. By leveraging these pre-trained weights, which are learned from large-scale datasets, we can benefit from their feature extraction capabilities and enable faster convergence and better performance during the training process for our YOLO v2 model.

1.2 Read YOLO pretrained weigts

```
class WeightReader:
    def __init__(self, weight_file):
        self.offset = 4
        self.all_weights = np.fromfile(weight_file, dtype='float32')

    def read_bytes(self, size):
        self.offset = self.offset + size
        return self.all_weights[self.offset-size:self.offset]

    def reset(self):
        self.offset = 4
```

```
weight_reader = WeightReader('datasets/yolo.weights')
```

Figure 8.Weights reader

3.7.4 Data Preprocessing :

Data preprocessing in the context of YOLO v2 refers to the steps taken to prepare the dataset for training the model. This involves formatting the data into the required structure with labeled images and annotations, resizing the images to match the model's input size, assigning anchor boxes for object localization, normalizing the pixel values, and encoding the class labels. These preprocessing steps ensure that the data is appropriately organized and prepared to be fed into the YOLO v2 model during the training phase.

3.7.4.1 Load Data & Parse annotations (xml) :

In the Load Data and Parse Annotations step, we read the file names from a CSV using the **load_data** function. Then, we split the data into training and testing datasets using the **load_datasets** function. Subsequently, we parse the annotations from XML files using the **parse_annotation** function. This process involves extracting relevant information such as object labels and bounding box coordinates from the XML files, enabling us to associate annotations with the corresponding image files.

3.7.4.2 Build Data Pipeline :

Given the substantial size and memory demands of image files, it's not feasible to load all images into memory simultaneously. To circumvent this, we devise an efficient data pipeline using TensorFlow's Dataset API, In this pipeline we sequentially load image files and corresponding annotation files, processing them in manageable batches rather than a monolithic whole.

In each batch we apply a resizing operation to standardize the dimensions of all images. Following this, we implement a normalization procedure to standardize the pixel values of the images. This step not only helps the model to converge faster but also enhances the overall performance by ensuring all input features (have a similar data distribution)

After the model train on the batch we carefully each batch from the memory to maintain system efficiency and prepare for the next batch. This approach optimizes memory usage.

```python
def read_images(img_obj, true_boxes):
    image=tf.io.read_file(img_obj)
    image=tf.io.decode_jpeg(image)
    height,width,channels=image.shape
    #resize
    image=tf.image.resize(image,(IMAGE_H,IMAGE_W),tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    if channels==1:
        image=tf.image.grayscale_to_rgb(image)
    #normalize
    image=tf.cast(image,dtype=tf.float32)/255
    return image, true_boxes

def tfData(ann_paths, batch_size):


    images_names, bbox = parse_annotation(LABELS,ann_paths)
    dataset = tf.data.Dataset.from_tensor_slices((images_names, bbox))
    dataset = dataset.shuffle(len(images_names))
    dataset = dataset.repeat()
    dataset = dataset.map(read_images, num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(tf.data.AUTOTUNE)
    print('-------------------')
    print('Dataset:')
    print(f'Images count: {len(images_names)}')
    print(f'Step per epoch: {len(images_names) // batch_size}')
    print(f'Images per epoch: {batch_size * (len(images_names) // batch_size)}')

    return dataset
```

Figure 9.Data Pipeline

3.7.4.3 Process data to Yolo format

In **process_true_boxes**, takes ground truth bounding boxes and anchor box dimensions as input. For each bounding box, it finds the best matching anchor (the one with the highest Intersection over Union, IoU), scales and centers the bounding box in the grid cell corresponding to the center of the object, and then fills the appropriate cells in the detector mask and the array of matching true boxes. The function returns these arrays, along with the adjusted coordinates of the bounding boxes.

In **ground_truth_generator**, generates the ground truth data needed to train the model for each batch of images in a dataset. For each batch, it calls **process_true_boxes** to get the detector mask and matching true boxes for all images in the batch. Then, it converts the class labels into one-hot vectors. This data can be fed directly into a YOLO model for training.

3.7.5 Loss function Implementation :

During the Loss step, we focus on implementing the YOLO v2 loss function. This loss function is designed to measure the discrepancy between the predicted bounding box attributes (coordinates and objectness score) and the ground truth annotations

The loss function involve :

3.7.5.1 Adjust prediction output :

YOLO V2 loss function starts off by rescaling the prediction output.

- **Sigmoid :** to adjust Bounding Boxes and confidence score
- **Softmax :** for classification
- **exponential activation :** to adjust width and height

```
# grid coords tensor
coord_x = tf.cast(tf.reshape(tf.tile(tf.range(GRID_W), [GRID_H]), (1, GRID_H, GRID_W, 1, 1)), tf.float32)
coord_y = tf.transpose(coord_x, (0,2,1,3,4))
coords = tf.tile(tf.concat([coord_x,coord_y], -1), [y_pred.shape[0], 1, 1, 5, 1])

# coordinate loss
pred_xy = K.sigmoid(y_pred[:,:,:,:,0:2]) # adjust coords between 0 and 1
pred_xy = (pred_xy + coords) # add cell coord for comparaison with ground truth. New coords in grid cell unit

pred_wh = K.exp(y_pred[:,:,:,:,2:4]) * anchors # adjust width and height for comparaison with ground truth

# class loss
pred_box_class = y_pred[..., 5:]
true_box_class = tf.argmax(class_one_hot, -1)
class_loss = K.sparse_categorical_crossentropy(target=true_box_class, output=pred_box_class, from_logits=True)

# confidence loss
pred_conf = K.sigmoid(y_pred[...,4:5])
```

Figure 10. Adjust predections

3.7.5.2 Calculate loss (x, y, w, h) :

Now we are ready to calculate the loss specific to the bounding box parameters, with

$$\text{loss}_{i,j}^{xywh} = \frac{1}{N_{L^{obj}}} \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{i,j}^{\text{obj}} \left[(x_{i,j} - \hat{x}_{i,j})^2 + (y_{i,j} - \hat{y}_{i,j})^2 + \left(\sqrt{w}_{i,j} - \sqrt{\hat{w}}_{i,j} \right)^2 + \left(\sqrt{h}_{i,j} - \sqrt{\hat{h}}_{i,j} \right)^2 \right]$$

```
nb_detector_mask = K.sum(tf.cast(detector_mask > 0.0, tf.float32))
xy_loss = LAMBDA_COORD * K.sum(detector_mask * K.square(matching_true_boxes[...,:2] - pred_xy)) / (nb_detector_mask + 1e-6)
wh_loss = LAMBDA_COORD * K.sum(detector_mask * K.square(K.sqrt(matching_true_boxes[...,2:4]) -
                                                        K.sqrt(pred_wh))) / (nb_detector_mask + 1e-6)
coord_loss = xy_loss + wh_loss
```

Figure 11.loss Boundig boxes

3.7.5.3 Calclate Class loss and conf :

$$\text{loss}_{i,j}^{p} = -\frac{1}{N_{L^{obj}}} \lambda_{\text{class}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{i,j}^{\text{obj}} \sum_{c \in \text{class}} p_{i,j}^{c} \log(\hat{p}_{i,j}^{c})$$

```
# class loss
pred_box_class = y_pred[..., 5:]
true_box_class = tf.argmax(class_one_hot, -1)
class_loss = K.sparse_categorical_crossentropy(target=true_box_class, output=pred_box_class, from_logits=True)
class_loss = K.expand_dims(class_loss, -1) * detector_mask
class_loss = LAMBDA_CLASS * K.sum(class_loss) / (nb_detector_mask + 1e-6)
```

Figure 12.classification loss

3.7.5.4 Calculate confidence loss :

The rest of calculation is dedicated to evaluate confidence loss

$$\text{loss}_{i,j}^{c} = \lambda_{\text{obj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{i,j}^{\text{obj}} \left(IOU_{\text{prediuction}_{i,j}}^{\text{ground truth}_{i,j}} - \widehat{C}_{i,j} \right)^2 + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{i,j}^{\text{noobj}} \left(0 - \widehat{C}_{i,j} \right)$$

So, we nee first to calculate **Intersection over union** between ground truth bounding box and predicted bounding box

Than we need to calculate no object loss and object loss

```
# no object confidence loss
no_object_detection = K.cast(best_ious < 0.6, K.dtype(best_ious))
noobj_mask = no_object_detection * (1 - detector_mask)
nb_noobj_mask  = K.sum(tf.cast(noobj_mask  > 0.0, tf.float32))

noobject_loss =  LAMBDA_NOOBJECT * K.sum(noobj_mask * K.square(-pred_conf)) / (nb_noobj_mask + 1e-6)
# object confidence loss
object_loss = LAMBDA_OBJECT * K.sum(detector_mask * K.square(ious - pred_conf)) / (nb_detector_mask + 1e-6)
# total confidence loss
conf_loss = noobject_loss + object_loss
```

Figure 13 confidence loss

3.7.5.5 Calculate total loss :

Finally combine all the calculation above

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(x_i - \hat{x}_i\right)^2 + \left(y_i - \hat{y}_i\right)^2$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2 \qquad (3)$$

```
# total loss
loss = conf_loss + class_loss + coord_loss
sub_loss = [conf_loss, class_loss, coord_loss]



return loss, sub_loss
```

Figure 14.Total loss

3.7.6 Training :

During training, our YOLO V2 model learns from the provided data by adjusting its parameters through backpropagation and optimization algorithms.

The objective is to minimize the defined loss function, enabling the model to effectively detect and localize objects in new data.

To train the model, we follow a specific sequence of steps :

1. **feed forward :** we feed forward the input data through the YOLO V2 model, which generates predictions for object bounding boxes and class probabilities.
2. **Calculate Loss :** we calculate the loss by comparing the predicted values with the ground truth annotations.
3. **Backpropagation :** we compute the gradients of the model's parameters with respect to the loss using backpropagation. These gradients guide the optimization process, allowing us to update the model's parameters through gradient descent or other optimization algorithms , in our case we used Adam optimizer

```
# optimizer
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-5, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Figure 15.optimizer

```
# gradients
def grad(model, img, detector_mask, matching_true_boxes, class_one_hot, true_boxes, training=True):
    with tf.GradientTape() as tape:
        y_pred = model(img, training)
        loss, sub_loss = yolov2_loss(detector_mask, matching_true_boxes, class_one_hot, true_boxes, y_pred)
    return loss, sub_loss, tape.gradient(loss, model.trainable_variables)
```

Figure 16. calculate gradients

```
loss, _, grads = grad(model, img, detector_mask, matching_true_boxes, class_one_hot, true_boxes)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Figure 17. set new weights

we incorporate specific callbacks while training our model. These callbacks serve as checkpoints and provide additional functionality during the training process. In our case we have :

- **Save Model :** this callback automatically saves the model based on the validation loss, ensuring that the best model is saved for future use or further training.
- **CSV save :** callback that streams epoch results in csv file , facilitating future visualization and analysis of metrics.

```
#CSV save
csv_save(epoch,loss_avg,val_loss_avg)

# Save Model
if val_loss_avg < best_val_loss:
    save_best_weights(model, train_name, val_loss_avg)
    best_val_loss = val_loss_avg
```

Figure 17. callbacks

3.7.7 Evaluation :

After 300 epochs of training, our model achieved a best validation loss of **0.66** . The accompanying graph illustrates the progress of the loss, showcasing how it evolved with each epoch.

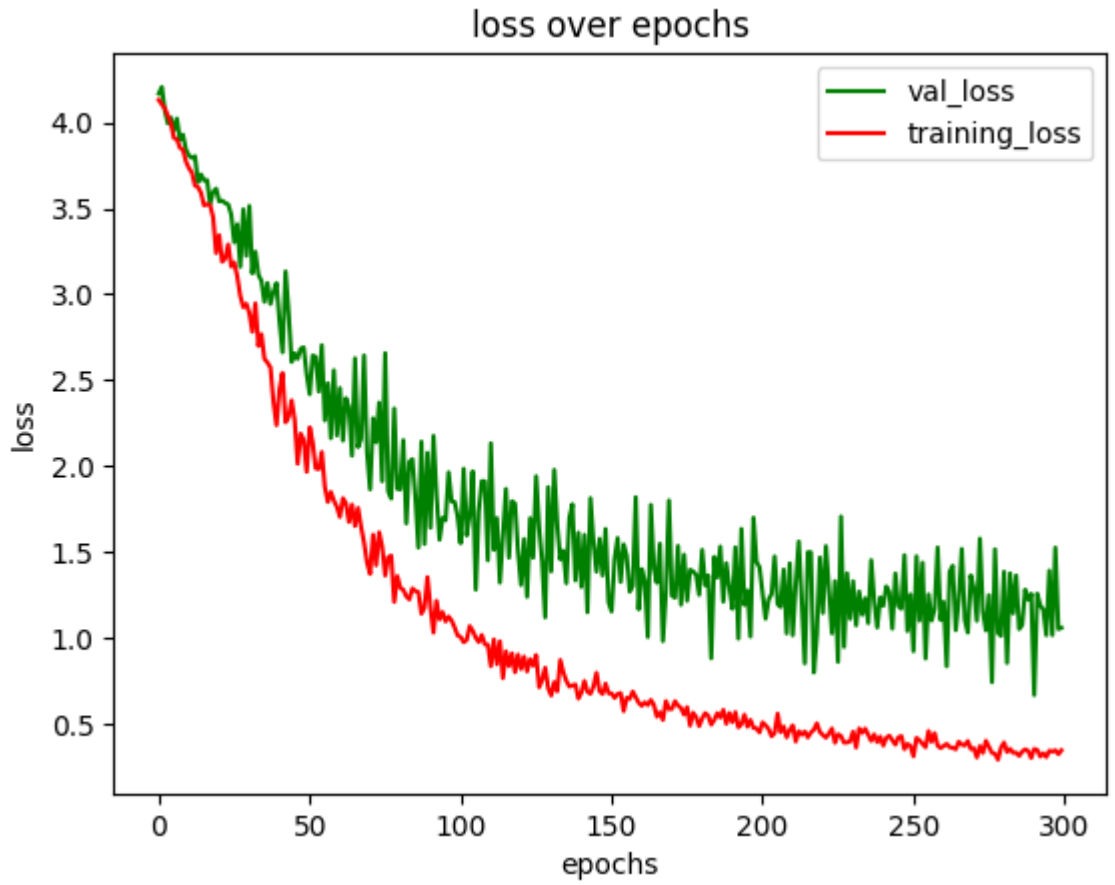
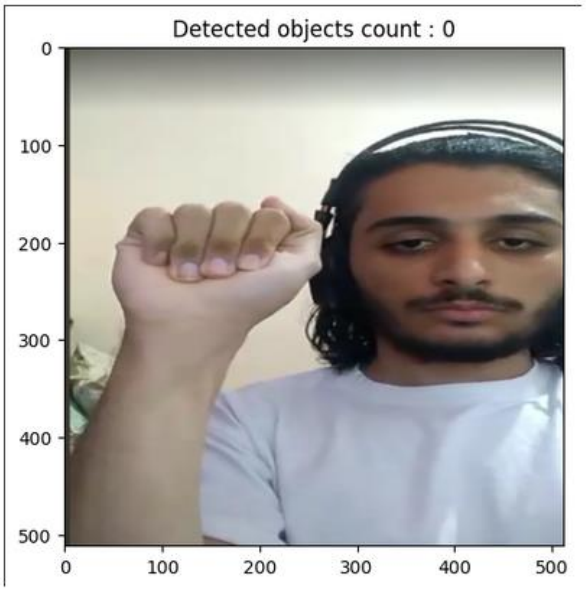


Figure 18. loss graph

This evaluation provides valuable insights into the model's performance and the effectiveness of the training process.

After testing the model on custom pictures, we obtained the following results:

Non-trained model



trained model