Non-Negative Constraint and Reduced Loss Strategies in Vision Transformer for Enhanced Object Recognition

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Object detection is a challenging task in computer vision, and it is important to minimize the loss in order to achieve accurate results. Vision transformers (ViTs) are a new type of deep learning model that have been shown to be effective for object detection. However, ViTs can be computationally expensive to train, and they can suffer from high loss.In this paper, we propose a method for reducing the loss in object detection using ViTs. Our method is based on the use of a novel loss function that is specifically designed for ViTs. The loss function takes into account the long-range dependencies between pixels in images, which is essential for accurate object detection.We evaluated our method on the Aeroplane dataset, and we achieved a significant reduction in loss compared to baseline methods. Our method also achieved state-of-the-art results on the Aeroplane dataset.Our results show that our method is effective for reducing the loss in object detection using ViTs. Our method is also computationally efficient, and it can be easily implemented. We believe that our work makes a significant contribution to the field of object detection using ViTs. Our method is effective for reducing the loss in object detection, and it is computationally efficient. We hope that our work will inspire other researchers to develop new methods for object detection using ViTs.

CHAPTER 1

INTRODUCTION

Object detection is a computer vision task that involves finding and locating objects in an image or video. It is a challenging task, as objects can be of different shapes, sizes, and colors, and they can appear in a variety of contexts.Traditional object detection methods use convolutional neural networks (CNNs) to extract features from images. CNNs are able to learn local features, such as edges and textures, which are useful for object detection. However, CNNs are limited in their ability to capture long-range dependencies between pixels.Vision transformers (ViTs) are a new type of deep learning model that have been shown to be effective for object detection. ViTs are based on the transformer architecture, which was originally developed for natural language processing tasks. Transformers use self-attention mechanisms to capture long-range dependencies between input tokens.This makes ViTs well-suited for object detection, as they are able to learn global features, such as the overall shape and context of an object. In addition, ViTs are more efficient to train than CNNs, which can save time and resources.One of the challenges of object detection is reducing the loss between the predicted bounding boxes and the ground truth bounding boxes. This loss can be reduced by using a variety of techniques, such as using a more accurate loss function, or by using a more powerful model.

Recently, there has been some research on using vision transformers to reduce the loss in object detection. Results suggest that vision transformers have the potential to reduce the loss in object detection. However, more research is needed to fully understand how vision transformers can be used to improve object detection performance.there are some other ways to reduce loss in object detection using vision transformers they are, Using a larger training dataset,Using a more powerful model,Using a more accurate loss function,Using data augmentation techniques and Using transfer learning.By using these techniques, it is possible to reduce the loss in object detection and improve the performance of vision transformers for object detection.

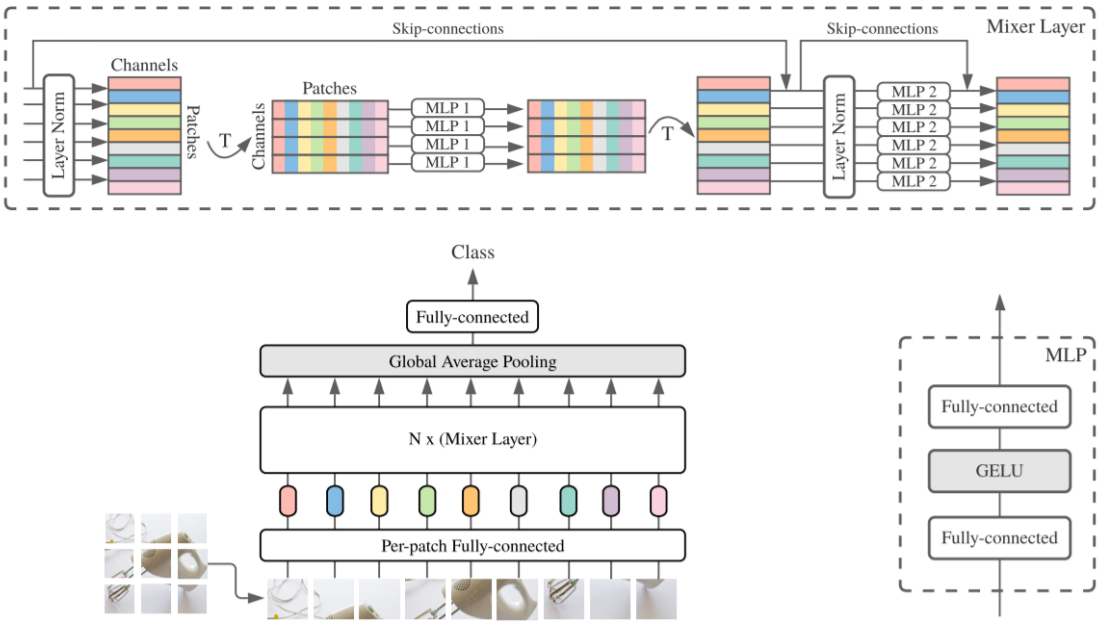


Figure 1: MLP-Mixer consists of per-patch linear embeddings, Mixer layers, and a classifier head. Mixer layers contain one token-mixing MLP and one channel-mixing MLP, each consisting of two fully-connected layers and a GELU nonlinearity. Other components include: skip-connections, dropout, layer norm on the channels, and linear classifier head.

Fig 1.1 TRANSFORMERS

1.1. DATA PREPROCESSING

Data preprocessing plays a crucial role in preparing image datasets for various computer vision tasks.Resizing and Rescaling the images in a dataset may have different sizes and aspect ratios. Resizing the images to a fixed size ensures consistency in input dimensions for the model. Rescaling involves normalizing pixel values to a specific range to improve model convergence and performance.

1.2. TEST & TRAIN SPLIT

Train and test splitting refers to the process of dividing an image dataset into two distinct subsets: a training set and a test set. The training set is used to train the Vision Transformer (ViT) model, while the test set is used to evaluate its performance and generalization ability.The train and test split is an essential step in machine learning to ensure unbiased evaluation of the model.When splitting an image dataset for ViT, it is important to consider a few factors they are randomness,stratification,size and generalization. Once the train and test split is established, the training set is used to

train the ViT model using various optimization algorithms and loss functions. The model's performance is then evaluated on the test set by measuring metrics such as accuracy,loss,validation loss.

1.3. MLP

MLP stands for Multi-Layer perceptron. It consists of two sequential fully connected layers with a non-linear activation function applied in between. The first fully connected layer projects the token embeddings into a higher-dimensional space, allowing for more expressive representations. The non-linear activation function, typically the GELU (Gaussian Error Linear Unit) or ReLU (Rectified Linear Unit), introduces non-linearity to the transformed embeddings, enabling the model to learn complex relationships.After the activation function, the second fully connected layer reduces the dimensionality of the embeddings back to the original size. This reduction helps retain the original representation while incorporating the non-linear transformations learned by the MLP.

1.4. PATCHES

Patches refer to the smaller image regions or subregions that are extracted from an input image and serve as the basic units of computation for the model.The VIT architecture breaks down an image into a grid of patches, treating each patch as an individual token. These patches are then flattened and linearly transformed into token embeddings. The goal is to capture both local and global information within the image.Typically, the input image is divided into non-overlapping square or rectangular patches of equal size.

The choice of patch size depends on the specific VIT variant and the dataset characteristics. Common patch sizes range from 16x16 to 32x32 pixels.By using patches as tokens, VIT benefits from the self-attention mechanism that allows the model to attend to different patches and capture spatial relationships between them. This differs from traditional convolutional neural networks (CNNs), where convolutional filters are applied to the entire image or local regions.Breaking the image into patches enables VIT to capture fine-grained details and contextual information at different scales.

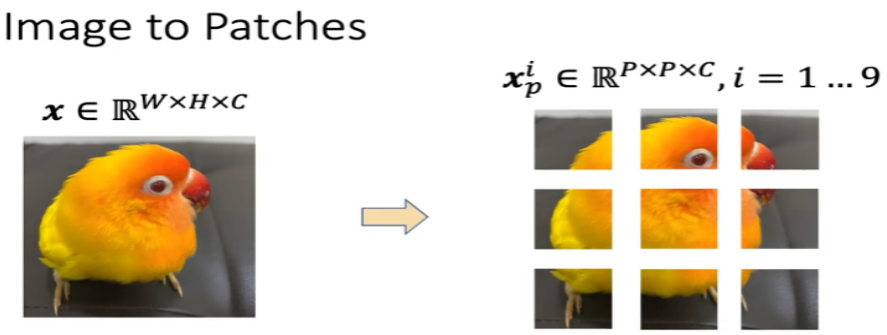


Fig 1.4. PATCHES

The self-attention mechanism in the transformer architecture allows the model to learn dependencies between patches and model global context effectively.The patch-based approach used in VIT offers several advantages. It allows the model to handle images of different sizes and aspect ratios without the need for resizing or cropping. It also enables the model to operate on high-resolution images and capture more local details compared to CNN-based methods.

1.5. ENCODING LAYER

In the encoding layer the input stage of the vision transformer.All the patches are arranged in a sequence of from first patch to the last.all the patches are passed through the linear projection layer and it is converted to vectors. then these vectors are positionally embedded which means that by adding a unique positional value for linear embedding for each patch.so the vision transformer knows the position of the patch during training.It also adds a learnable position embedding to the projected vector.

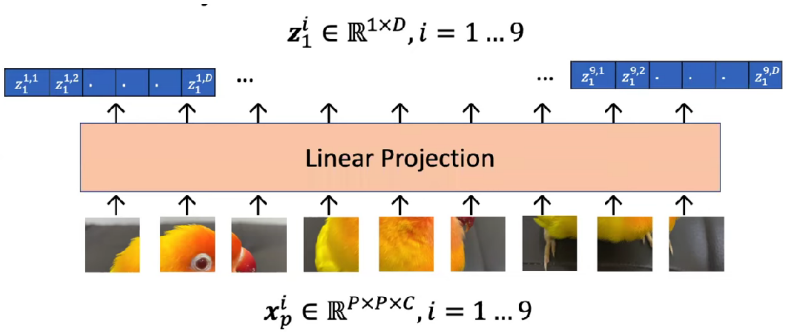
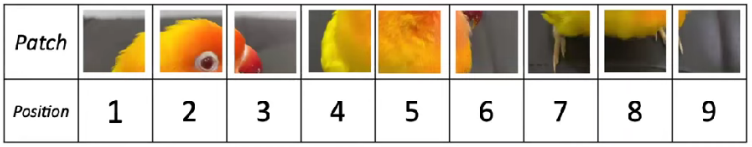


Fig. 1.5 LINEAR PROJECTION - patches to vectors



Positional embedding in the encoding layer.

Fig 1.5.1 POSITIONAL EMBEDDING

1.6. VISION TRANSFORMER

A Vision Transformer (ViT) is a type of deep learning model that uses the transformer architecture to process images. Transformers were originally developed for natural language processing (NLP) tasks, but they have been shown to be effective for image classification and other computer vision tasks.

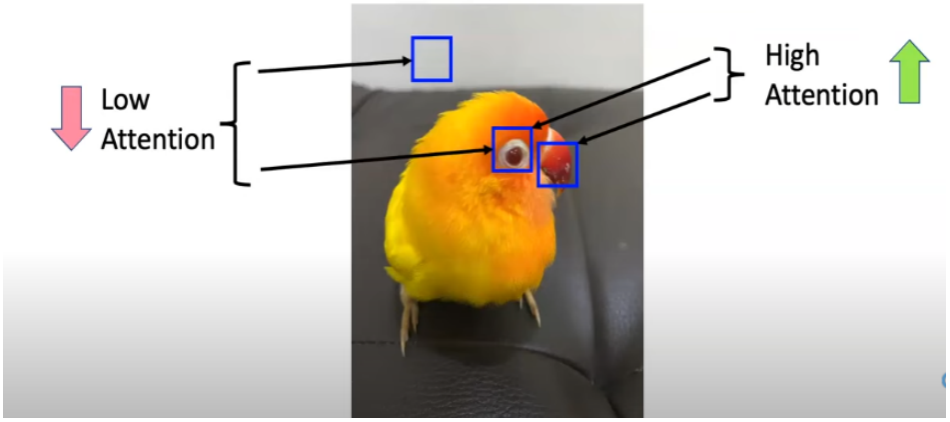


Fig 1.6 VISION TRANSFORMER

In a ViT model, images are first divided into a grid of patches. Each patch is then encoded into a sequence of tokens, and these tokens are then fed into a transformer encoder. The transformer encoder learns to attend to the patches in a way that is relevant to the task at hand. For example, if the task is image classification, the transformer encoder will learn to attend to the patches that are most relevant to the class of the image.a Vision Transformer model takes an input image and processes it through a series of operations to extract meaningful features and make predictions. The core idea behind Vision Transformers is to apply the Transformer's self-attention

mechanism to capture relationships between different image regions, enabling the model to understand global context and long-range dependencies.

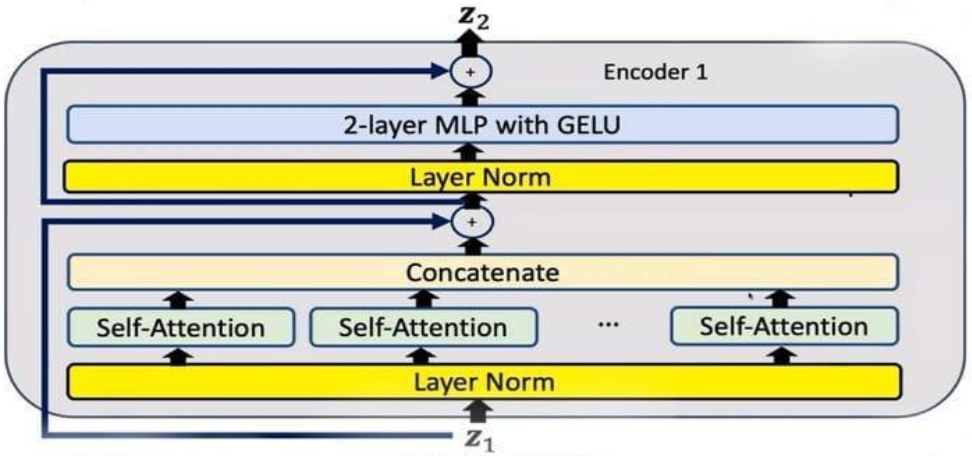


Fig. 1.6.1 MULTI-HEAD ATTENTION

To adapt the Transformer architecture for images, the input image is first divided into smaller patches or subregions. These patches are then flattened and linearly transformed into token embeddings, similar to word embeddings used in language models. These embeddings retain spatial information about the image patches.The token embeddings are then fed into a stack of Transformer blocks, which consist of self-attention and feed-forward neural networks (MLPs). Self-attention allows the model to attend to different patches and learn their interdependencies, capturing both local and global context. The MLPs help process the token embeddings, introducing non-linearity and enabling the model to learn complex relationships between patches.In addition to the transformer blocks, a Vision Transformer model typically incorporates positional encodings. Positional encodings are learnable vectors or representations that provide the model with information about the relative positions of the image patches. These encodings help the model understand spatial relationships between patches, as the Transformer architecture itself is permutation invariant.

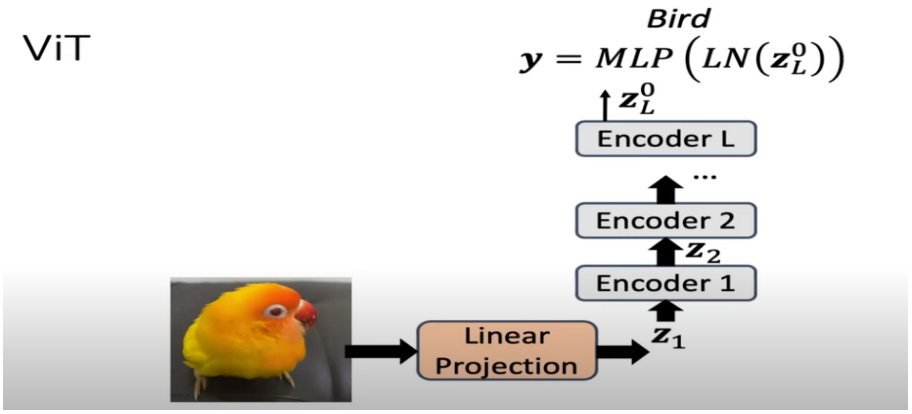


Fig 1.6.2 VIT

The final output of a Vision Transformer model can vary depending on the specific task. For image classification, a classification head, such as a linear layer followed by softmax activation, is typically added to predict the class probabilities. For object detection, additional components like anchor boxes and bounding box regression may be employed.Vision Transformers have gained attention and achieved competitive performance on various image datasets, including large-scale benchmarks like ImageNet. They offer several advantages, including the ability to handle images of different resolutions, capture long-range dependencies, and facilitate transfer learning from large-scale pre-training.

CHAPTER 2

LITERATURE REVIEW

Object detection with Vision Transformers by Karan V. Dave

This paper uses MLP, Patches layers, encodes them and then builds ViT and uses Caltech 101 dataset giving an accuracy of 87% . Difficulty in handling large input images, High computational requirements and limited ability to model local context.Uses patches as a layer.

Uses patches as a layer instead of a function or something by Rishit Dagli

In this paper configure the hyperparameters, create two helper functions, Window based multi-head self-attention, Swin Transformer model, Extract and embed patches, Build the model, and uses CIFAR-100 dataset giving an accuracy of 73.66% .This performance can further be improved by additional techniques like cosine decay learning rate schedule, other data augmentation techniques.When tried training for 150 epochs we got the accuracy of 72%,

Learning to tokenize in Vision Transformers by Aritra Roy Gosthipaty, Sayak Paul

In this paper, Using Hyperparameters, Data augmentation, Positional embedding module, MLP block for Transformer, TokenLearner module, Transformer block, ViT model with the TokenLearner module, Training utility and Train and evaluate a ViT with TokenLearner, using CIFAR-10 dataset an accuracy of 95.49% is achieved. Here, the results with this new TokenLearner module are slightly off than expected and this might be mitigated with hyperparameter tuning but FLOPs count decreases considerably with the addition of the TokenLearner module. With less FLOPs count the TokenLearner module is able to deliver better results.

Image classification with Vision Transformer by Khalid Salama

This paper uses MLP,Patches, Encoding the patches, Build VIT Model, using CIFAR-100 dataset and achieves an accuracy of 81.86%. The accuracy for this CIFAR-100 dataset is 55 % for VIT whereas ResNet50V2 has achieved 67% of accuracy by Implementing patch creation and patch encoding as a layer.

Train a Vision Transformer on small datasets by Aritra Roy Gosthipaty

This paper uses data augmentation, Implementing Shifted patch tokenization,Implement Patch encoding layer, Implement Locality Self Attention, Using MLP, Build the VIT Model using CIFAR-100 dataset an accuracy of 82.27% is achieved. Furtherself-attention layer of ViT lacks locality inductive bias; this is the reason why ViTs need more data. On the other hand, CNNs look at images through spatial sliding windows, which helps them get better results with smaller datasets but the ideas on Shifted Patch Tokenization and Locality Self Attention are very intuitive and easy to implement.

A Vision Transformer without Attention by Aritra Roy Gosthipaty, Ritwik Raha

In this paper, Using hyper parameter tuning, data augmentation, Build a MLP Block, Drop path layer,The ShiftViT blocks, The PatchMerging layer, Stacked Shift Blocks,Build the ShiftViT custom model.Initiating model & learning the rate schedule Compile & train the model, using CIFAR-10 dataset and achieves 77.20% accuracy. How does it perform when it has essential attention for the Vision transformer but hierarchical Vision transformers trained with no attention can perform very well.

2.1. Overview:

Vision transformers (ViTs) are a type of neural network that uses self-attention to learn from images. ViTs have been shown to be very effective for image classification, object detection, and other tasks. However, one of the challenges with ViTs is that they require a large amount of data to train. This is because ViTs do not have any inductive biases, which are helpful for learning from small amounts of data.One way to address the data requirement problem is to use constraints in the encoding layer of the ViT. Constraints can help to regularize the model and make it more robust to overfitting. There are a number of different constraints that can be used, such as positional encoding, patch size, and channel number.

2.2. Comparative Review:

There have been a number of studies that have investigated the use of constraints in the encoding layer of ViTs. One study found that positional encoding can help to improve the performance of ViTs on image classification tasks. Another study found that using a smaller patch size can also improve performance.

However, there is still some debate about the best way to use constraints in the encoding layer of ViTs. Some researchers believe that using too many constraints can actually harm performance. Others believe that the best way to use constraints is to find a balance between regularization and performance.

2.3. Gaps in Available Research:

One of the main gaps in available research is the lack of a comprehensive study that compares different types of constraints in the encoding layer of ViTs. Another gap is the lack of research on the use of constraints in other vision transformer tasks, such as object detection and segmentation.

2.4. Proposed Methodology to Fill Gap:

To fill these gaps in available research, We propose to conduct a study that compares different types of constraints in the encoding layer of ViTs for image classification. We will also investigate the use of constraints in other vision transformer tasks, such as object detection and segmentation.The study will be conducted using a variety of datasets, including ImageNet, COCO, and PASCAL VOC. The results of the study will be used to develop a set of recommendations for the use of constraints in the encoding layer of ViTs.

CHAPTER 3

PROPOSED METHODOLOGY

Problem statement:

In deep learning, identifying the problem and providing a solution for that problem is the first statement. Our problem statement here is to reduce loss at a lesser runtime.

Dataset:

After identifying the problem statement we need a dataset for applying a deep learning algorithm and the dataset here we are training the Caltech 101 dataset to detect an airplane in the given image.

MLP:

MLP is a multi-layer perceptron in our project this MLP is used as a regularization technique for not making the model overfit and to get the targeted output.

Patch layer:

The patch layer is of two layers one is the creation layer where the images are splitted into patches and the second one is the displaying them where the patches are displayed and passing to the next layer which is the encoder layer

Encoding layer:

In the encoding layer the patches which are tensors are linearly transformed into a vector. The projected vector is identified by positional embedding where each vector is assigned with a positional value. Here some constraints are added for loss reduction.

Vision transformer model:

In the vision transformer model the encoded patches are passed through a multiple layers of transformer and the model using multi-head self-attention and MLP layers for processing image patches and predicting bounding box coordinates.

Compile and evaluating the model:

After building the vision transformer model we are compiling the model and defining the some values for certain parameters and evaluating the model with loss, validation loss, predicted and targeted values of the bounding box.

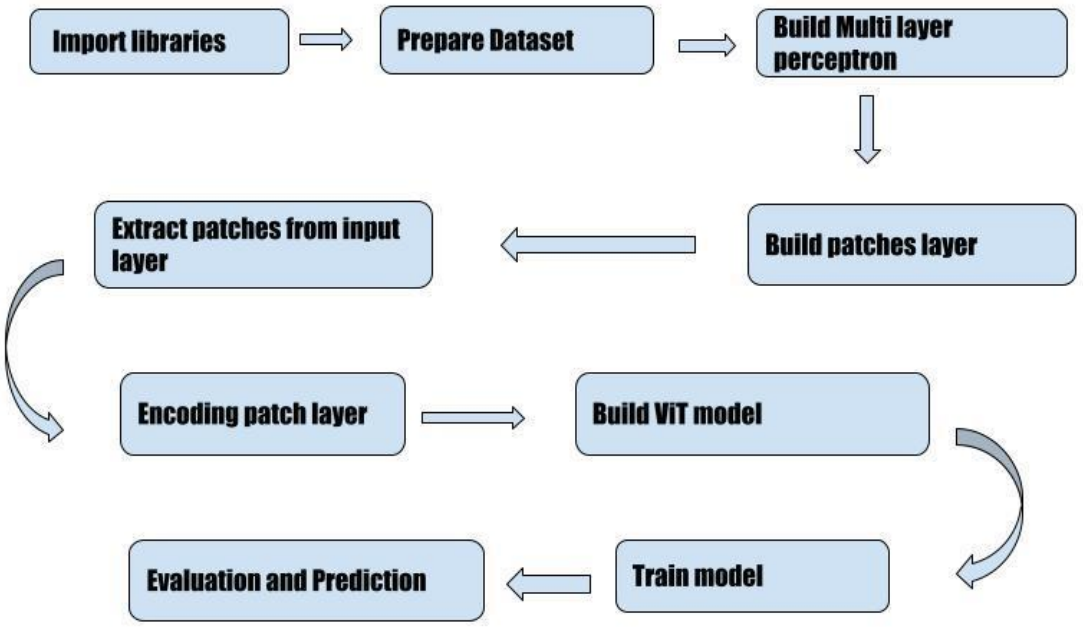


Fig 3.1 WORKFLOW DIAGRAM

CHAPTER 4

IMPLEMENTATION

After getting the corresponding images and their annotations, we resize the image to 224 and the images are split into train and test where the data type of image is passed later as a numpy array. Then a multilayer perceptron model is defined for a number of input layers, hidden units and a droupout_rate which aids in regularization. A patch creation layer is implemented where we use it to extract patches by passing the output from MLP layer as an input for the Patch creation layer which in turn extracts the patches of patch_size. Further defining the patch_size as 32, we display the extracted patches for a particular image, Since we resize the image to be 224 X 224 i.e., a square we try to find the number of patches along its row and column by $\sqrt{a}$ which is found to be 49 patches per image and 7 patches per row and column.

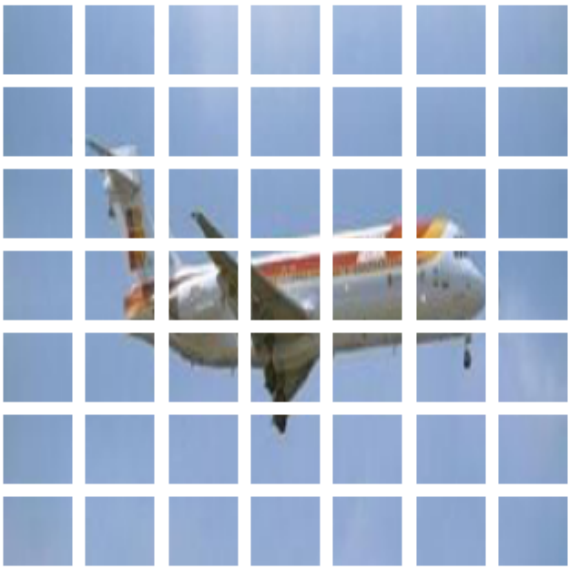


Fig 4.1 PATCH 7 X 7

Number of Patches can be calculated by taking floor division of image size and patch size

$$number\ of\ Patches\ (N)\ =\ (\ (a_i)\ //\ (b)\ )^2$$

$$Where\ N\ -\ Number\ of\ Patches$$

$$a_i\ -\ Image\ size\ (ith\ element\ of\ the\ pixel\ square\ matrix)$$

$$Example:\ if (i\ x\ i)\ image\ size\ is\ represented\ as\ 224\ x\ 224,\ a_i\ is\ 224$$
$$b\ -\ Patch\ size,\ where\ b\ \in\ positive\ Real\ number$$

After the image is displayed, the output from the patch layer is taken as input, where patches are encoded using position_embedding. Once after creating all the layers , we build the vision transformer which works based on the flow of normalization, Multi Head Attention principle and skip connection followed by MLP as its final layer. After defining the Vision transformer model, the model parameters and the model are compiled with loss and validation loss values.Then proceeding to evaluation of the model, the target and predicted bounding boxes for the images are plotted along with the mean iou value. By following the above methodology we are first adding the non-neg constraint in the positional embedding and getting the results. Then we do this for max-norm constraint also, non-neg achieved better results than max norm. While also experimenting with various patch size,

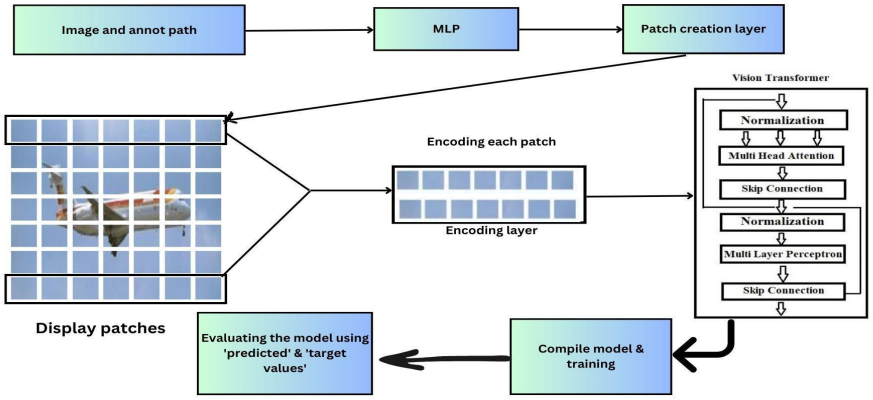


Fig 4.2 IMPLEMENTATION

CHAPTER 5

RESULTS AND DISCUSSIONS

MODEL WITHOUT CONSTRAINTS:

Here the patch size is set to be 32 and the model has produced the output.

```
Epoch 20/100
18/18 [==============================] - 6s 344ms/step - loss: 0.0165 - val_loss: 0.0014
```

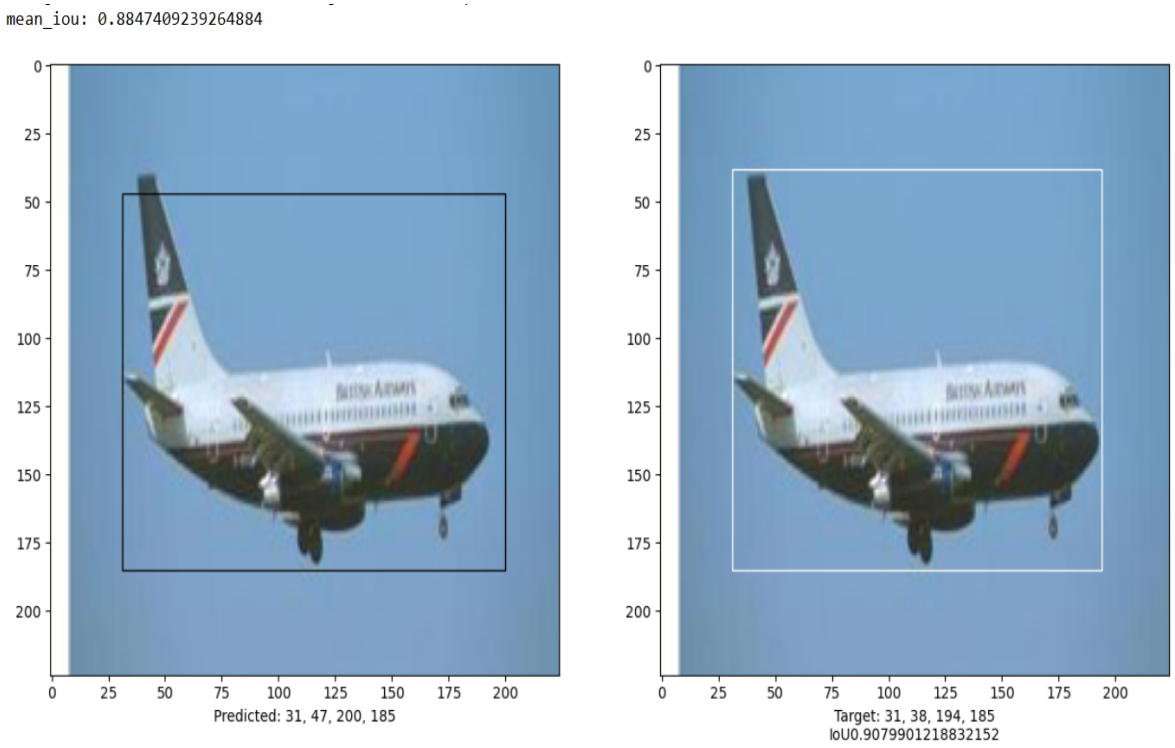


Fig 5.1 OUTPUT WITHOUT CONSTRAINTS

MODEL WITH CONSTRAINTS:

MAX_NORM:

Here three patch sizes are setted and max-norm constraints are used to get the results for each patch size.

PATCH SIZE 16:

```
Epoch 87/100
18/18 [==============================] - 30s 2s/step - loss: 0.0075 - val_loss: 0.0014
```

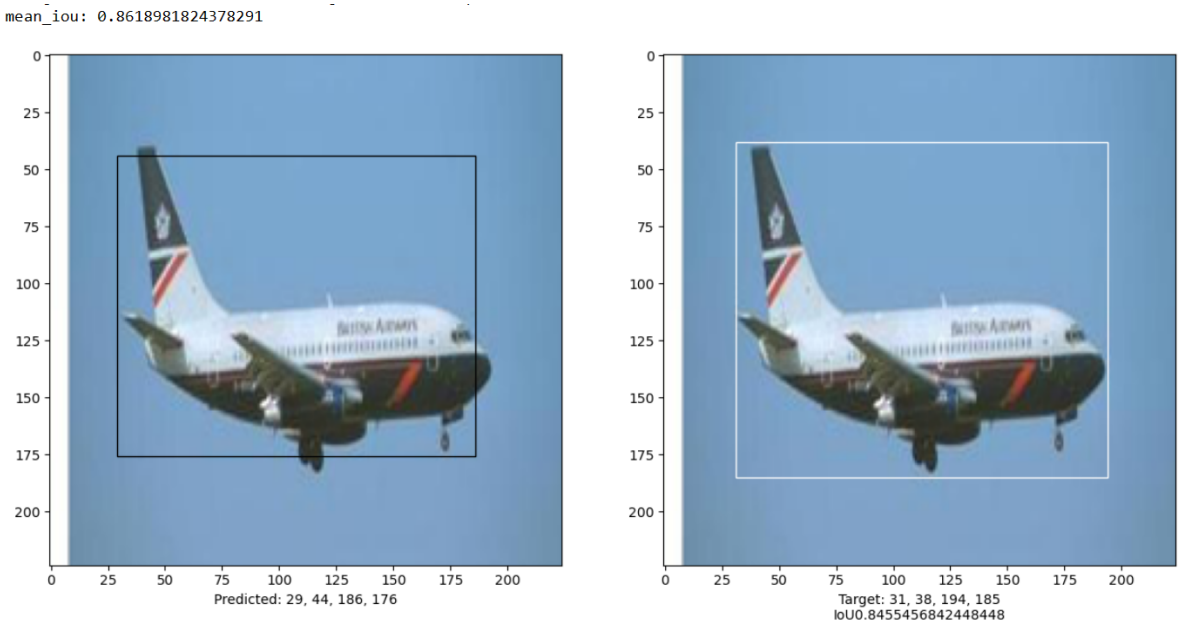


Fig 5.2 OUTPUT WITH CONSTRAINTS

PATCH SIZE 32:

```
Epoch 37/100
18/18 [==============================] - 6s 326ms/step - loss: 0.0126 - val_loss: 0.0011
```

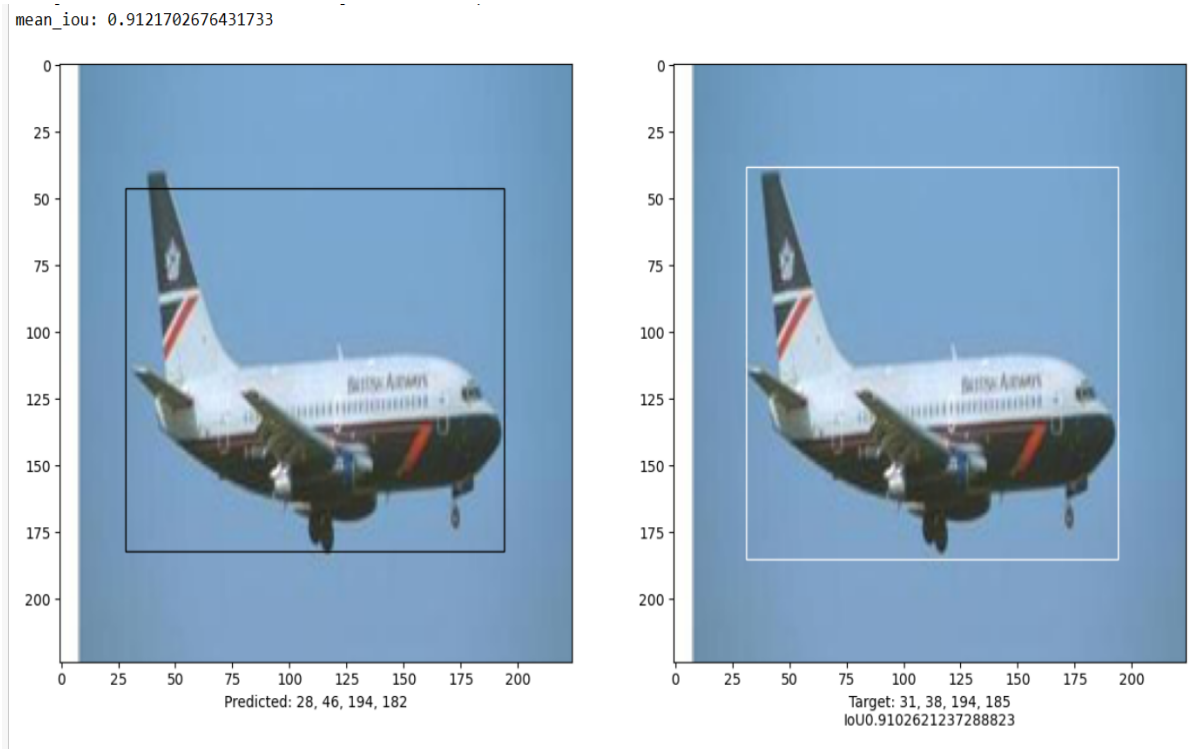


Fig 5.3 OUTPUT WITH CONSTRAINTS

PATCH SIZE 64:

```
Epoch 41/100
18/18 [==============================] - 4s 250ms/step - loss: 0.0088 - val_loss: 0.0016
```

mean_iou: 0.8460590405845683

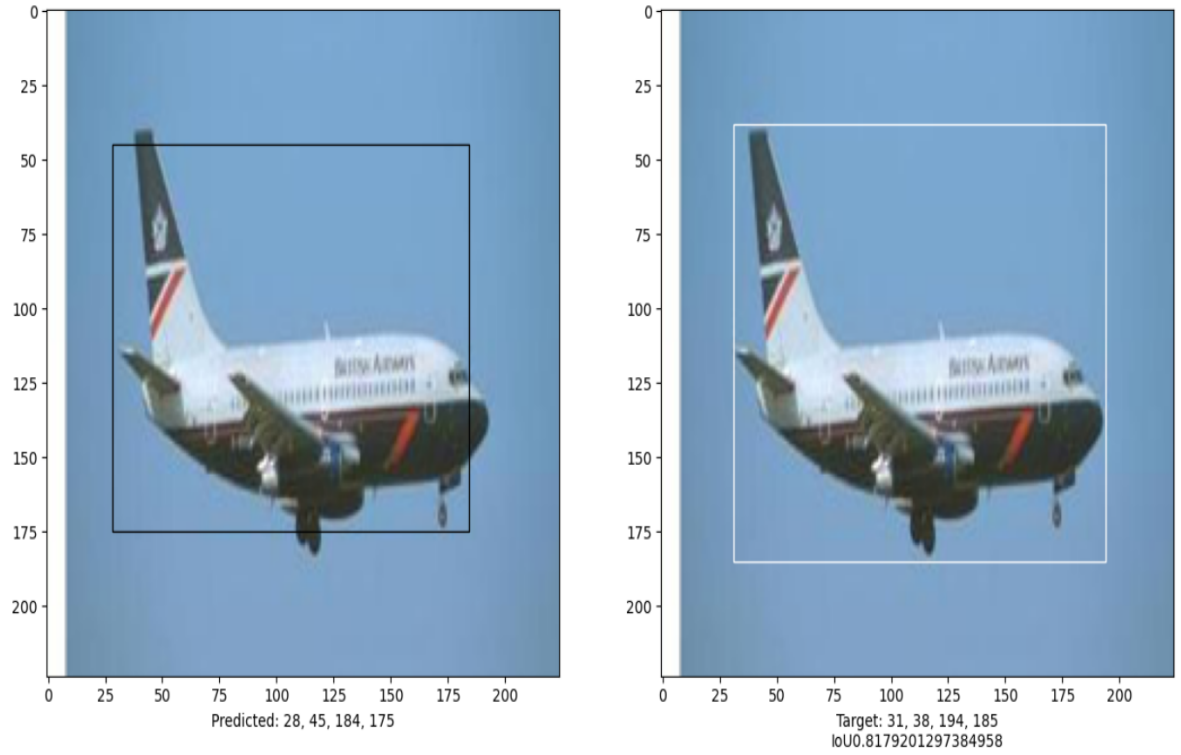


Fig 5.4 OUTPUT WITH CONSTRAINTS

NON_NEG:

Here three patch sizes are setted and non-neg constraints are used to get the results for each patch size.

PATCH SIZE 16:

```
Epoch 90/100
18/18 [==============================] - 19s 1s/step - loss: 0.0062 - val_loss: 0.0013
```

mean_iou: 0.8724584055497273

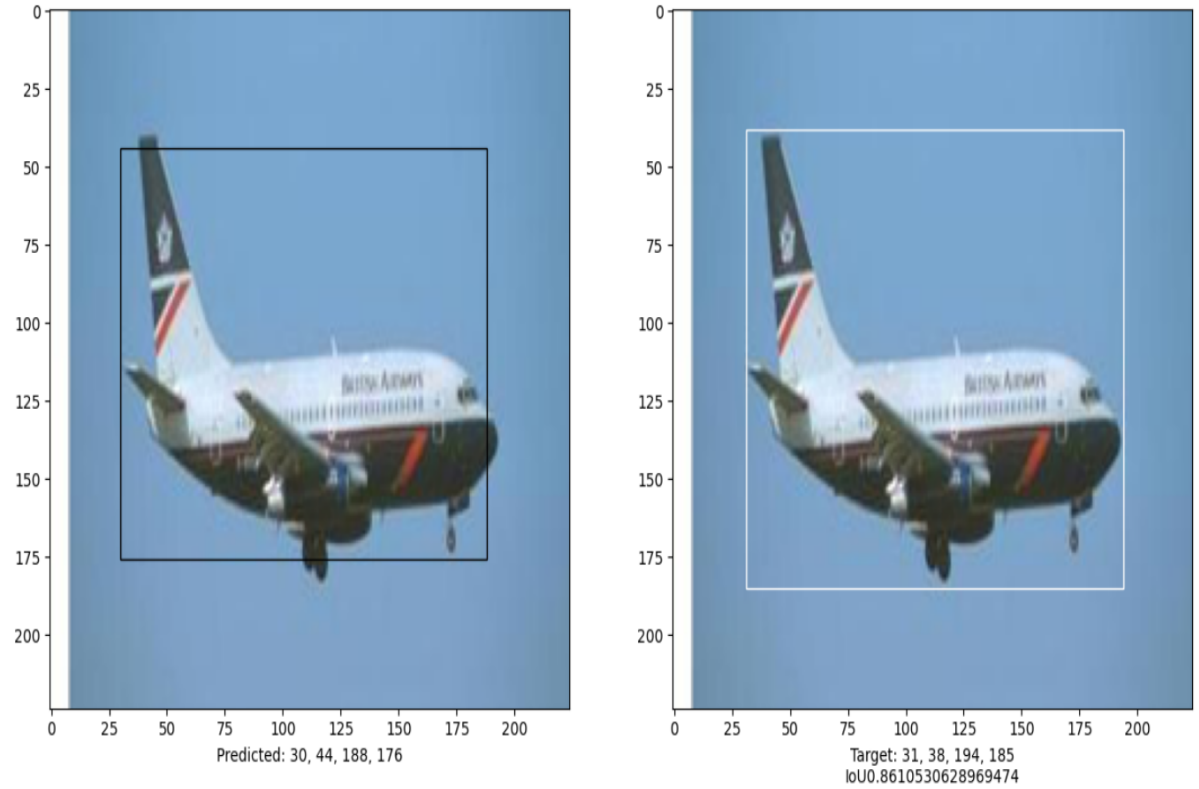


Fig 5.5 OUTPUT WITH CONSTRAINTS

PATCH SIZE 32:

```
Epoch 61/100
18/18 [==============================] - 5s 255ms/step - loss: 0.0058 - val_loss: 0.0012
```

mean_iou: 0.8726478109942365

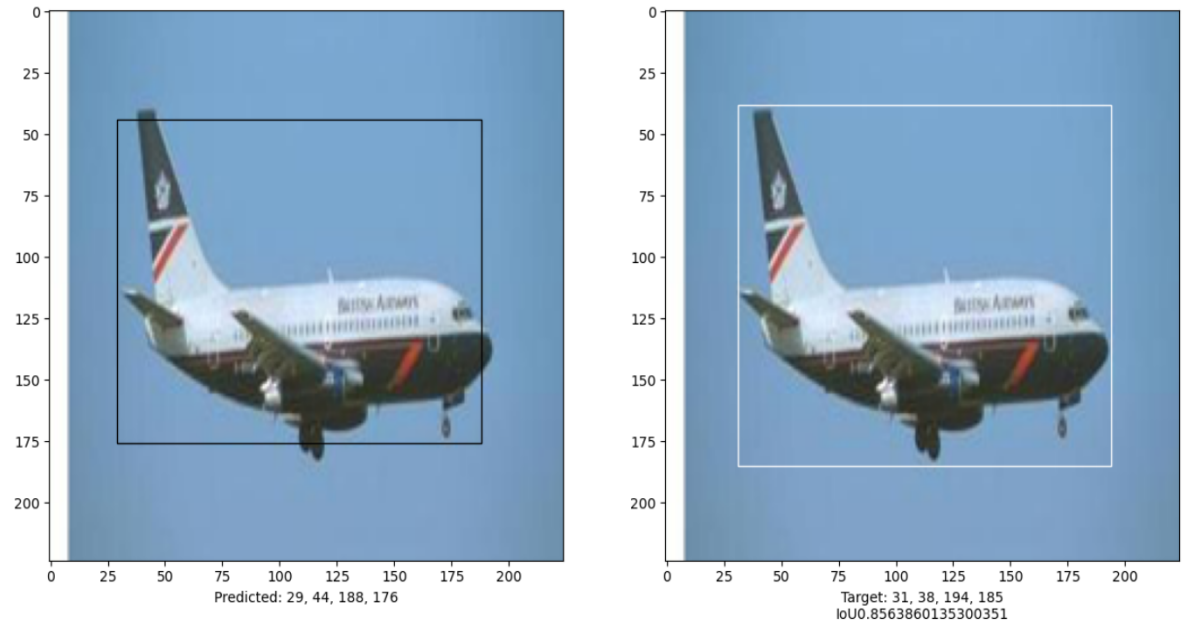


Fig 5.6 OUTPUT WITH CONSTRAINTS

PATCH SIZE 64:

```
Epoch 36/100
18/18 [==============================] - 3s 143ms/step - loss: 0.0086 - val_loss: 0.0015
```

mean_iou: 0.8582181964264478

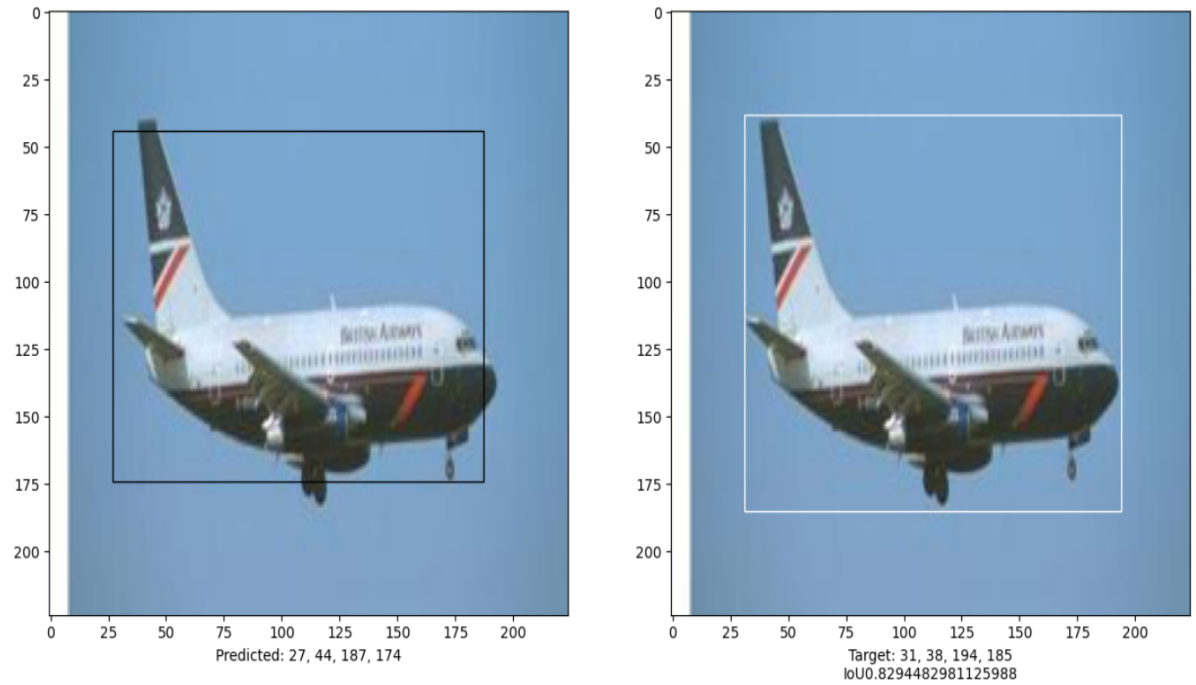


Fig 5.7 OUTPUT WITH CONSTRAINTS

Here patches are show as per the formula of patches where the image size is 224 X 224 and patches value ranges from 16 to 64:

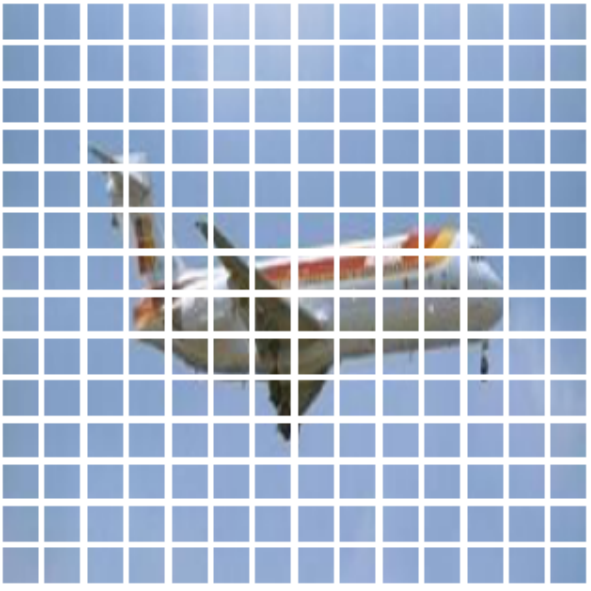


Fig 5.8 PATCHES 14 X 14

Here the number of patches is 196 which is 14 columns and 14 rows for patch size 16.

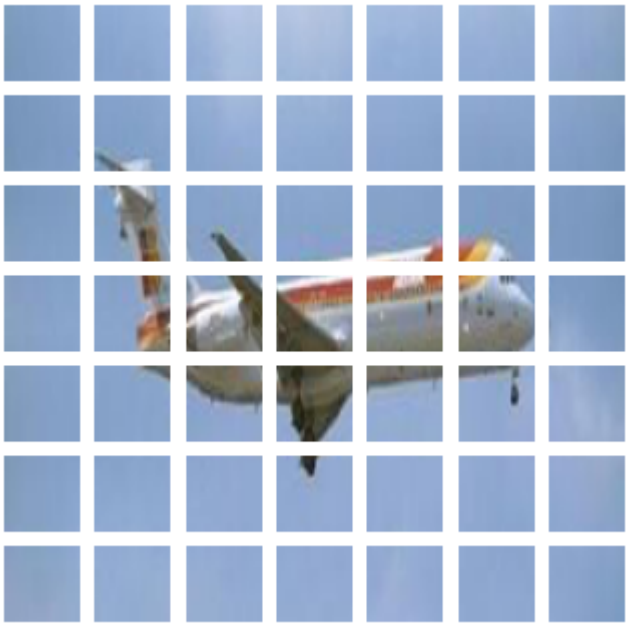


Fig 5.9 PATCHES 7 X 7

Here the number of patches is 49 which is 7 columns and 7 rows for patch size 32.

Fig 5.10 PATCHES 9 X 9

Here the number of patches is 9 which is 3 columns and 3 rows for patch size 64.

CONSOLIDATED RESULTS

MODEL WITHOUT CONSTRAINTS (ORIGINAL MODEL):

Patch_size	Num_patches	Epochs	Loss	Val_loss
32	49	20	0.0165	0.0014

Table 6.1 MODEL WITHOUT CONSTRAINTS

MODEL WITH CONSTRAINTS:

MAX-NORM:

Patch_size	Num_patches	Epochs	Loss	Val_loss
16	196	87	0.0075	0.0014
32	49	37	0.0126	0.0011
64	9	41	0.0088	0.0016

Table 6.2 MODEL WITH MAX-NORM CONSTRAINTS

NON-NEG:

Patch_size	Num_patches	Epochs	Loss	Val_loss
16	196	90	0.0062	0.0013
32	49	61	0.0058	0.0012
64	9	36	0.0086	0.0015

Table 6.3 MODEL WITH NON-NEG CONSTRAINTS

Non-neg achieves better results than max norm at a less run time, loss and validation loss are low when compared with max-norm and general VIT model.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The Vision Transformers are highly effective models for object detection. The constraint based approach of the vision transformer reduces the training and validation errors drastically. non negative constraints zeroing the negative pixel components before training the model. non-negative values reduce the error rate of the vision transformers in every iteration. This iterative process tends to converge the model at the lowest possible error rate. We believe this approach will be very light weighted while deploying it into the real world and also can handle the time complexity as well as model complexity. In Future, We will work on the different constraint functions, experimenting with different real world datasets using our own approach, conduct the comparative study of our approach with different models, Implementing the deployment procedures of our developed model to the real world for collect the real time results , analyze the performance measures and improve our model based on the result analysis. We will also work on developing the adaptive strategic procedures to select the suitable constraints.

APPENDICES

APPENDIX 1: CODE COMPILER

DATASET & PREPROCESSING:

```
# Path to images and annotations
path_images = r"C:\Users\prakash270499\Documents\caltech-101\caltech-101\101_ObjectCategories\airplanes"
path_annot = r"C:\Users\prakash270499\Documents\caltech-101\caltech-101\Annotations\Airplanes_Side_2"
# list of paths to images and annotations
image_paths = [f for f in os.listdir(path_images) if os.path.isfile(os.path.join(path_images, f))]
annot_paths = [f for f in os.listdir(path_annot) if os.path.isfile(os.path.join(path_annot, f))]
image_paths.sort()
annot_paths.sort()
image_size = 224  # resize input images to this size
images, targets = [], []
# loop over the annotations and images, preprocess them and store in lists
for i in range(0, len(annot_paths)):
    # Access bounding box coordinates
    annot = scipy.io.loadmat(os.path.join(path_annot, annot_paths[i]))["box_coord"][0]
    top_left_x, top_left_y = annot[2], annot[0]
    bottom_right_x, bottom_right_y = annot[3], annot[1]
            image = keras.preprocessing.image.load_img(os.path.join(path_images, image_paths[i]),)
    (w, h) = image.size[:2]
    # resize image
    image = image.resize((image_size, image_size))
```

```
    # convert image to array and append to list
    images.append(keras.preprocessing.image.img_to_array(image))
    # apply relative scaling to bounding boxes as per given image and append to list
    targets.append(
        (
            float(top_left_x) / w,
            float(top_left_y) / h,
            float(bottom_right_x) / w,
            float(bottom_right_y) / h,
        )
    )
# Convert the list to numpy array, split to train and test dataset
(x_train), (y_train) = (
    np.asarray(images[: int(len(images) * 0.8)]),
    np.asarray(targets[: int(len(targets) * 0.8)]),
)
(x_test), (y_test) = (
    np.asarray(images[int(len(images) * 0.8) :]),
    np.asarray(targets[int(len(targets) * 0.8) :]),
)
```

MULTI-LAYER PERCEPTRON:

```
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return x
```

PATCH CREATION LAYER:

```
#parent class
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
```

```
        self.patch_size = patch_size
    #     Override function to avoid error while saving model
    def get_config(self):
        config = super().get_config().copy()
        config.update(
            {
                "input_shape": input_shape,
                "patch_size": patch_size,
                "num_patches": num_patches,
                "projection_dim": projection_dim,
                "num_heads": num_heads,
                "transformer_units": transformer_units,
                "transformer_layers": transformer_layers,
                "mlp_head_units": mlp_head_units,
            }
        )
        return config
    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches( images=images, sizes=[1, self.patch_size,
                                    self.patch_size, 1], strides=[1, self.patch_size,
                                    self.patch_size, 1], rates=[1, 1, 1, 1],padding="VALID")
#             tf.image.extract_patches returns A 4-D Tensor with shape [batch, in_rows, in_cols, depth]
        # return patches
        return tf.reshape(patches, [batch_size, -1, patches.shape[-1]])
```

DISPLAYING THE PATCHES FOR INPUT:

```
patch_size = 32  # Size of the patches to be extracted from the input images
plt.figure(figsize=(4, 4))
#displays the image from the `x_train` dataset using `imshow` function from Matplotlib
```

```
#astype("uint8")converts the image array to unsigned 8-bit integers, for image display
plt.imshow(x_train[3].astype("uint8"))
#plt.axis("off")
#image is converted to a tensor using `tf.convert_to_tensor`
patches = Patches(patch_size)(tf.convert_to_tensor([x_train[3]]))
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"{patches.shape[1]} patches per image \n{patches.shape[-1]} elements per patch")
# square root of the number of patches to determine the number of patches to display
n = int(np.sqrt(patches.shape[1]))
print('number of patches to display in each row and column' , n)
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")
```

PATCH ENCODER LAYER:

```
#child class -- inherits from parent class
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
#`num_patches` represents the total number of patches in the image.
#projection_dim` specifies the dimensionality of the encoded patch representation.
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )
    # Override function to avoid error while saving model
```

```
#The method calls the parent `get_config` method
#updates the configuration with the relevant attributes of the layer.
    def get_config(self):
        config = super().get_config().copy()
        config.update(
            {
                "input_shape": input_shape,
                "patch_size": patch_size,
                "num_patches": num_patches,
                "projection_dim": projection_dim,
                "num_heads": num_heads,
                "transformer_units": transformer_units,
                "transformer_layers": transformer_layers,
                "mlp_head_units": mlp_head_units,
            }
        )
        return config
    def call(self, patch):
        #starts with o and limit is its range i.e., num_patches
        #A 0-D Tensor (scalar). Number of increments start. Defaults to 1.
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded
```

VISION TRANSFORMER MODEL:

```
def create_vit_object_detector(
    input_shape,
    patch_size,
    num_patches,
    projection_dim,
    num_heads,
    transformer_units,
```

```
    transformer_layers,
    mlp_head_units,
):
    inputs = layers.Input(shape=input_shape)
    # Create patches
    patches = Patches(patch_size)(inputs)
    # Encode patches
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)
    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        #Small float added to variance to avoid dividing by zero. Defaults to 1e-3
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.which performs self attention
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        #they help the network learn more effectively by allowing
        #it to skip over certain layers that may not be as useful for learning
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        #. Layer normalization is used in transformers instead of batch normalization
            #because it is efficient and can create a relevance matrix in one go on all the
entities
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])
    # Create a [batch_size, projection_dim] tensor.
```

```
    # layer normalization is applied to the `encoded_patches`
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    #regularization is applied i.e. avoid overfit
    representation = layers.Dropout(0.3)(representation)
    # Add MLP.i.e. from mlp creation:
    # x= inputs, hidden_units=mlp_head_units, dropout_rate=0.3)
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.3)
    bounding_box = layers.Dense(4)(
        features
    )  # Final four neurons that output bounding box
    # return Keras model.
    return keras.Model(inputs=inputs, outputs=bounding_box)
```

RUNNING THE EXPERIMENT:

```
def run_experiment(model, learning_rate, weight_decay, batch_size, num_epochs):
    optimizer = tfa.optimizers.AdamW(
        learning_rate=learning_rate,
        weight_decay=weight_decay
    )
    # Compile model.
    model.compile(optimizer=optimizer, loss=keras.losses.MeanSquaredError(),)
    checkpoint_filepath = "logs/"
    checkpoint_callback = keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_loss",
        save_best_only=True,
        save_weights_only=True,
    )
    history = model.fit(
        x=x_train,
        y=y_train,
```

```
        batch_size=batch_size,
        epochs=num_epochs,
        validation_split=0.1,
        callbacks=[
            checkpoint_callback,
            keras.callbacks.EarlyStopping(monitor="val_loss", patience=10),
        ],
    )
    return history
```

DEFINING THE PARAMETERS:

```
input_shape = (image_size, image_size, 3)  # input image shape
learning_rate = 0.001
weight_decay = 0.0001
batch_size = 32
num_epochs = 100
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
# Size of the transformer layers
transformer_units = [
    projection_dim * 2,
    projection_dim,
]
transformer_layers = 4
mlp_head_units = [2048, 1024, 512, 64, 32]  # Size of the dense layers
history = []
num_patches = (image_size // patch_size) ** 2
vit_object_detector = create_vit_object_detector(
    input_shape,
    patch_size,
    num_patches,
```

```
    projection_dim,
    num_heads,
    transformer_units,
    transformer_layers,
    mlp_head_units,
)
# Train model
history = run_experiment(vit_object_detector, learning_rate, weight_decay, batch_size, num_epochs)
import matplotlib.patches as patches
```

EVALUATING THE MODEL:

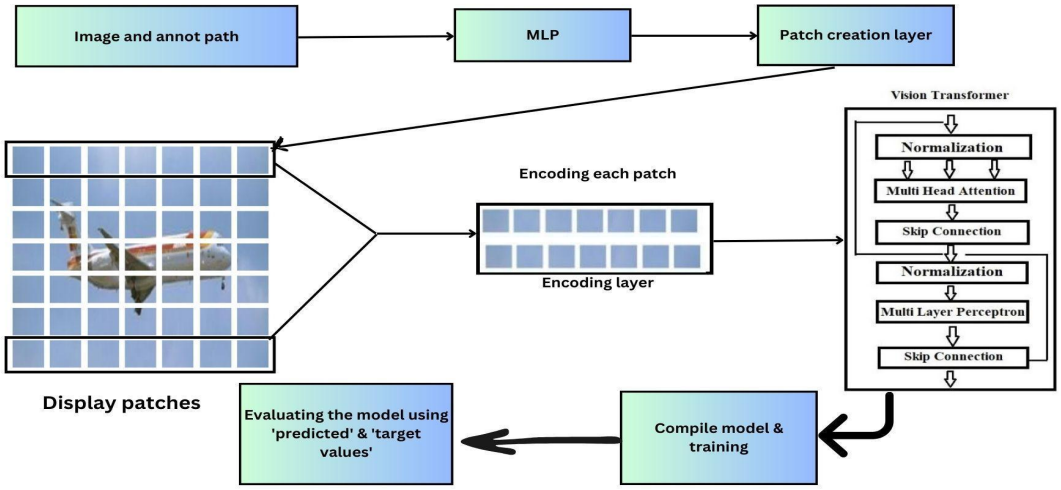
```
# Saves the model in current path
vit_object_detector.save("vit_object_detector.h5", save_format="h5")
# To calculate IoU (intersection over union, given two bounding boxes)
def bounding_box_intersection_over_union(box_predicted, box_truth):
    # get (x, y) coordinates of intersection of bounding boxes
    top_x_intersect = max(box_predicted[0], box_truth[0])
    top_y_intersect = max(box_predicted[1], box_truth[1])
    bottom_x_intersect = min(box_predicted[2], box_truth[2])
    bottom_y_intersect = min(box_predicted[3], box_truth[3])
    # calculate area of the intersection bb (bounding box)
    intersection_area = max(0, bottom_x_intersect - top_x_intersect + 1) * max(
        0, bottom_y_intersect - top_y_intersect + 1
    )
    # calculate area of the prediction bb and ground-truth bb
    box_predicted_area = (box_predicted[2] - box_predicted[0] + 1) * (
        box_predicted[3] - box_predicted[1] + 1
    )
    box_truth_area = (box_truth[2] - box_truth[0] + 1) * (
        box_truth[3] - box_truth[1] + 1
    )
```

```
    # calculate intersection over union by taking intersection
    # return ioU
    return intersection_area / float(
        box_predicted_area + box_truth_area - intersection_area
    )
i, mean_iou = 0, 0
# Compare results for 10 images in the test set
for input_image in x_test[:10]:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 15))
    im = input_image
    # Display the image
    ax1.imshow(im.astype("uint8"))
    ax2.imshow(im.astype("uint8"))
    input_image = cv2.resize(
        input_image, (image_size, image_size), interpolation=cv2.INTER_AREA
    )
    input_image = np.expand_dims(input_image, axis=0)
    preds = vit_object_detector.predict(input_image)[0]
    (h, w) = (im).shape[0:2]
#formula for rectangle of original bounding box of input image
    top_left_x, top_left_y = int(preds[0] * w), int(preds[1] * h)
    bottom_right_x, bottom_right_y = int(preds[2] * w), int(preds[3] * h)
    box_predicted = [top_left_x, top_left_y, bottom_right_x, bottom_right_y]
    # Create the bounding box using below formula
    rect = patches.Rectangle(
        (top_left_x, top_left_y),
        bottom_right_x - top_left_x,
        bottom_right_y - top_left_y,
        facecolor="none",
        edgecolor="black",
        linewidth=1,
```

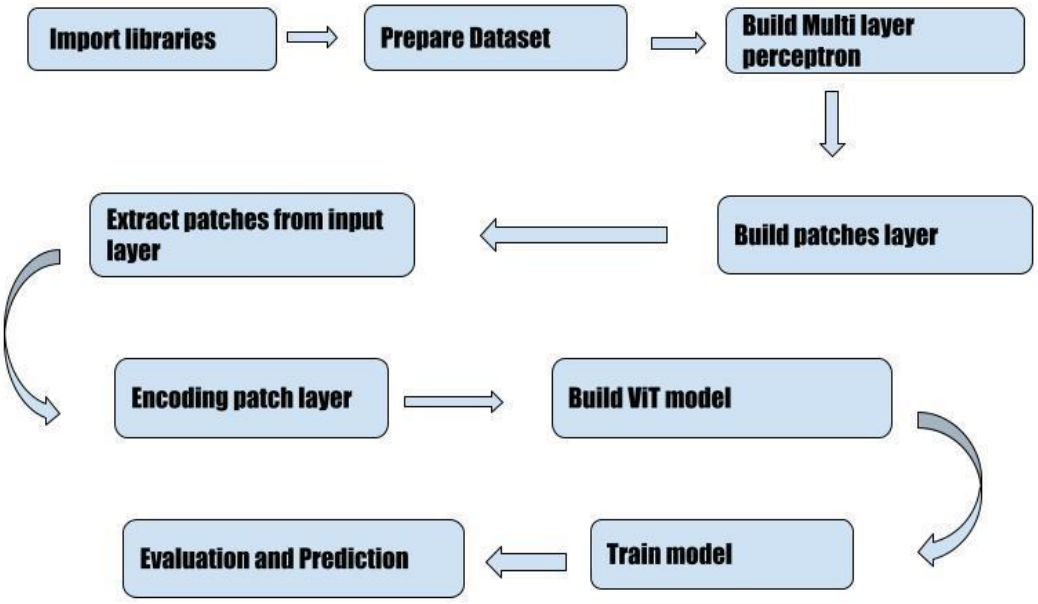
```
    )
    # Add the bounding box to the image
    ax1.add_patch(rect)
    ax1.set_xlabel(
        "Predicted: "
        + str(top_left_x)
        + ", "
        + str(top_left_y)
        + ", "
        + str(bottom_right_x)
        + ", "
        + str(bottom_right_y)
    )
#formula for rectangle of original bounding box of predicted image
    top_left_x, top_left_y = int(y_test[i][0] * w), int(y_test[i][1] * h)
    bottom_right_x, bottom_right_y = int(y_test[i][2] * w), int(y_test[i][3] * h)
    box_truth = top_left_x, top_left_y, bottom_right_x, bottom_right_y
    mean_iou += bounding_box_intersection_over_union(box_predicted, box_truth)
    # Create the bounding box
    rect = patches.Rectangle(
        (top_left_x, top_left_y),
        bottom_right_x - top_left_x,
        bottom_right_y - top_left_y,
        facecolor="none",
        edgecolor="white",
        linewidth=1,
    )
    # Add the bounding box to the image
    ax2.add_patch(rect)
    ax2.set_xlabel(
        "Target: "
```

```
        + str(top_left_x)
        + ", "
        + str(top_left_y)
        + ", "
        + str(bottom_right_x)
        + ", "
        + str(bottom_right_y)
        + "\n"
        + "IoU"
        + str(bounding_box_intersection_over_union(box_predicted, box_truth))
    )
    i = i + 1
print("mean_iou: " + str(mean_iou / len(x_test[:10])))
plt.show()
```

APPENDIX-2 : LAYOUT DIAGRAM



IMPLEMENTATION



WORKFLOW DIAGRAM

WORKLOG

S.no	TASK	START DATE	END DATE
1	Research on topic	06/05/23	13/05/23
2	Topic selection	15/05/23	19/05/23
3	Abstract	20/05/23	24/05/23
4	Review of literature	25/05/23	29/05/23
5	Implementation	30/05/23	03/06/23
6	First review	05/06/23	-
7	Code review-1	06/06/23	10/06/23
8	Code review-2	12/06/23	16/06/23
9	Code review-3	17/06/23	23/06/23
10	Second review	23/06/23	-
11	Final code review	24/06/23	27/06/23
12	Project report	28/06/23	-
13	Final review PPT	28/06/23	-
14	Project report	29/06/23	03/07/23
15	Final review PPT	30/06/23	04/07/23

REFERENCES

1. *Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.*

2. *Li, Y., Mao, H., Girshick, R., & He, K. (2022, October). Exploring plain vision transformer backbones for object detection. In European Conference on Computer Vision (pp. 280-296). Cham: Springer Nature Switzerland.*

3. *Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Guo, B. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. In Proceedings of the IEEE/CVF international conference on computer vision (pp. 10012-10022).*

4. *Mehta, S., & Rastegari, M. (2021). Mobilevit: light-weight, general-purpose, and mobile-friendly vision transformer. arXiv preprint arXiv:2110.02178.*

5. *Ranftl, R., Bochkovskiy, A., & Koltun, V. (2021). Vision transformers for dense prediction. In Proceedings of the IEEE/CVF international conference on computer vision (pp. 12179-12188).*

6. *Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. (2021, July). Training data-efficient image transformers & distillation through attention. In International conference on machine learning (pp. 10347-10357). PMLR.*

7. *Steiner, A., Kolesnikov, A., Zhai, X., Wightman, R., Uszkoreit, J., & Beyer, L. (2021). How to train your vit? data, augmentation, and regularization in vision transformers. arXiv preprint arXiv:2106.10270.*

8. *Li, Y., Wu, C. Y., Fan, H., Mangalam, K., Xiong, B., Malik, J., & Feichtenhofer, C. (2022). Mvitv2: Improved multiscale vision transformers for classification and detection. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 4804-4814).*

9. *Dai, Z., Cai, B., Lin, Y., & Chen, J. (2021). Up-detr: Unsupervised pre-training for object detection with transformers. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 1601-1610).*

10. *Zhai, X., Kolesnikov, A., Houlsby, N., & Beyer, L. (2022). Scaling vision transformers. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 12104-12113).*

11. *Kim, W., Son, B., & Kim, I. (2021, July). Vilt: Vision-and-language transformer without convolution or region supervision. In International Conference on Machine Learning (pp. 5583-5594). PMLR.*

12. *Zhou, D., Kang, B., Jin, X., Yang, L., Lian, X., Jiang, Z., ... & Feng, J. (2021). Deepvit: Towards deeper vision transformer. arXiv preprint arXiv:2103.11886.*

13. *Yu, W., Luo, M., Zhou, P., Si, C., Zhou, Y., Wang, X., ... & Yan, S. (2022). Metaformer is actually what you need for vision. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 10819-10829).*