Object Detection on Pascal Dataset using EfficientNetB0

Human beings collect information through their senses to understand the universe and then analyze it. Among these senses, the sense of sight undoubtedly plays an important role. In the field of artificial intelligence, the computer vision technique first emerged in 1966 under the "Summer Vision Project" project in the examination of data carrying visual information such as images and videos. Later, it has been developed with advanced machine learning and deep learning algorithms since 1990.(Zou, Z., 2023) One of the most challenging aspects of traditional computer vision techniques was that the edges, corners or colors in the image had to be defined manually. This problem-solving process was quite long and tiring. Since the algorithms used were specialized algorithms for certain tasks, they could not analyze every image with the same quality. To give examples of traditional computer vision monitoring algorithms, these are SIFT, support vector machines and histogram of oriented gradients. When it comes to advanced object oriented techniques, the most useful and useful algorithms are undoubtedly deep learning networks, or CNNs. One of the biggest advantages in the development of advanced methods was that a lot of data was collected on platforms such as ImageNet. This made it easier to create more deep learning algorithms from the collected image data and test their success. This provided more accuracy and allowed more objects to be examined. In addition, thanks to end-to-end learning, the image was analyzed with a single model from input to output and the process was shortened

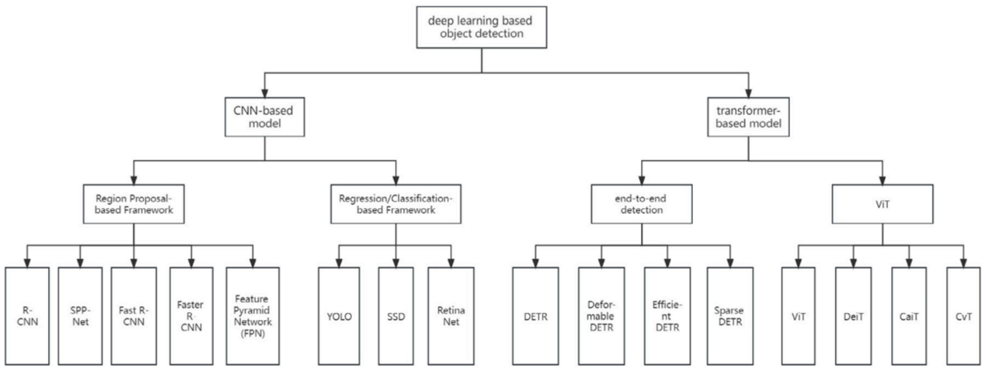
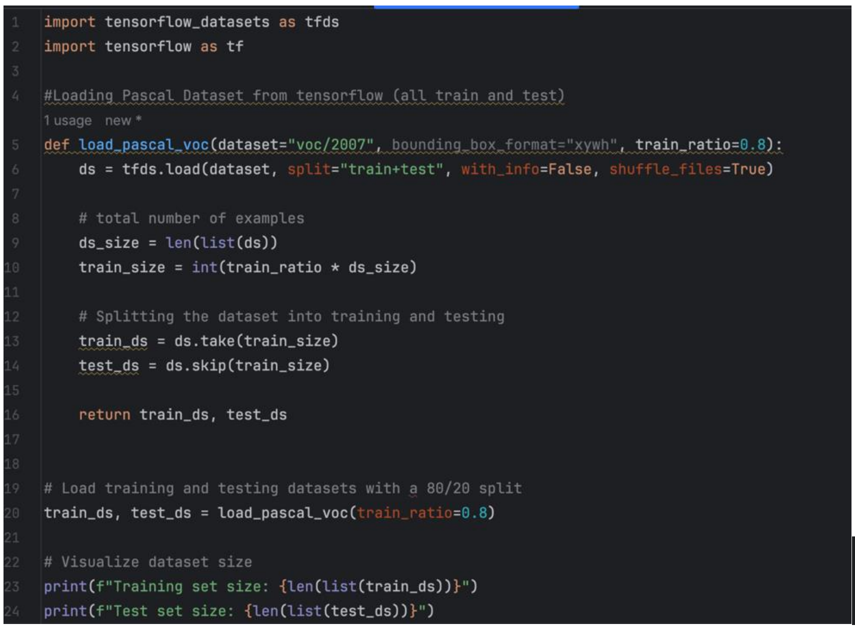


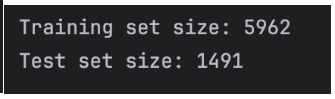
Fig. 1. Classification diagram of deep learning-based object detection.

Advanced computer vision models have entered people's lives in many areas of daily life because they can analyze many data instantly and efficiently. For example, they have been included in crucial areas such as energy usage and efficiency, climate change strategies, forest fire prevention, self-driving cars, waste reduction and medical MRI images and have started to achieve efficient work. Two of the most advanced and most used algorithms are Faster R-CNN and YOLO architectures.(Hubel D. H. and Wiesel T. N.) Faster R-CNN, as its name suggests, is an advanced algorithm of R-CNN. What makes it different is that it has a Region Proposal Network (RPN) in its structure. The network has a very high accuracy rate in detecting the locations and features of different objects. In this way, forests can be monitored with drones and information about biodiversity can be obtained. The algorithm is quite functional when it is desired to make observations and analysis for object identification or animal diversity depending on tree species. When we look at the other algorithm, YOLO, YOLO is a real-time object detection model that divides the image into a grid and analyzes each section separately. Thanks to its end-to-end structure, it has become quite fast and functional for algorithms with real-time environmental functions. It is possible for YOLOv8 to process hundreds of frames per second (FPS).

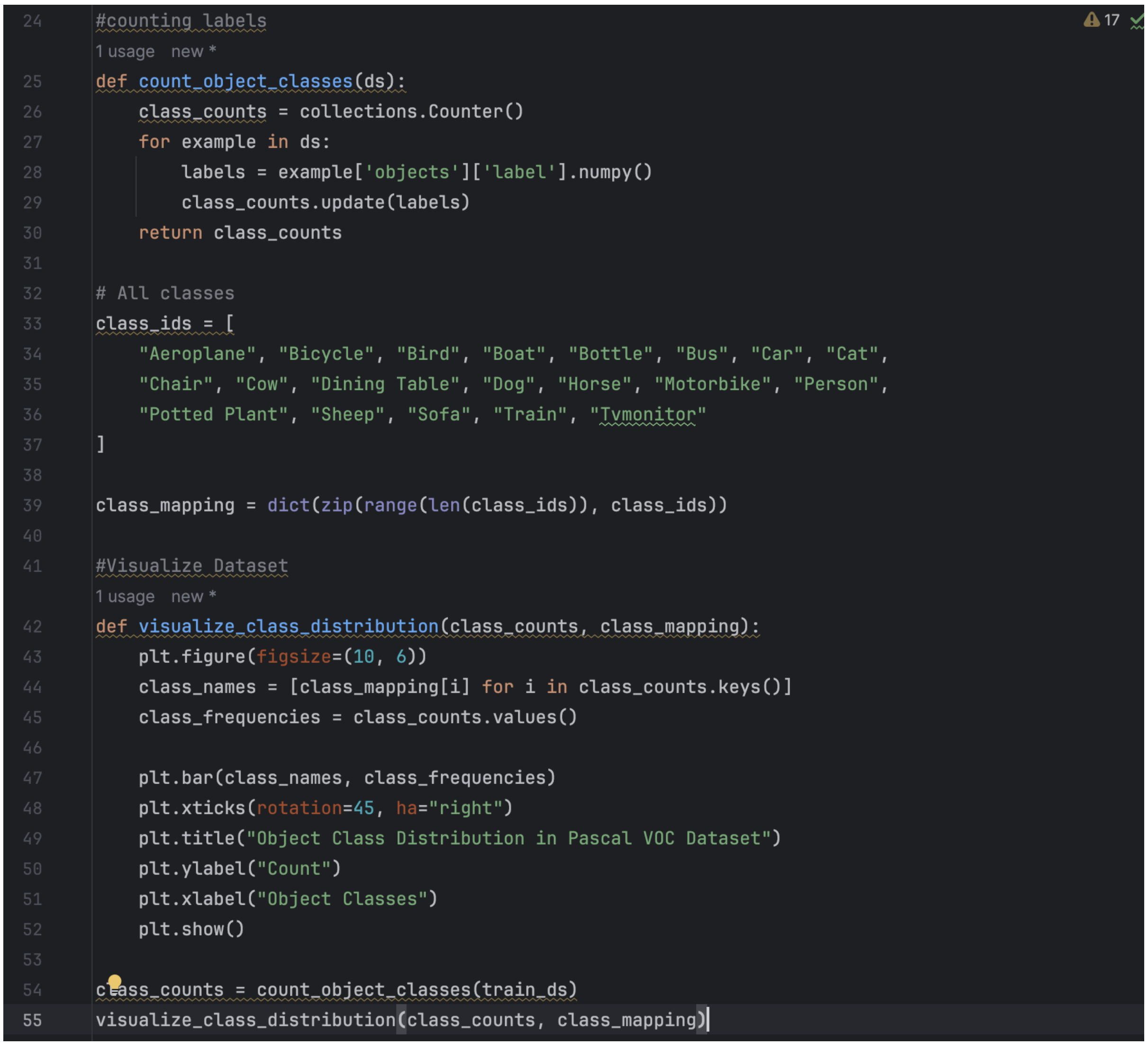
Firstly, in this project, Pascal 2007 image dataset which tensorflow library has are used. Tensorflow and tensorflow datasets are imported because it will be needed to create new neural networks to train. Furthermore, Pascal 2007 dataset are loaded through load_pascal_voc function. It's bounding boxes format are set as "xywh" means there are four coordinates. Also, both train and test datas are loaded and shuffled without any information of dataset. After that process, dataset is measured as list. It gives the how many elements in the dataset. Below that, train size is set as a calculation of 0,8 (train ratio) multiply the dataset size. Furthermore, train and test sets are splitted. First, train set are splitted. Then, remain datas are splitted as test. Also, these splitted datasets which are train and test are set as train ratio equals 0,8. Therefore as shown below, training set size is 5962 and test set size is 1491.



```python
import tensorflow_datasets as tfds
import tensorflow as tf

#Loading Pascal Dataset from tensorflow (all train and test)
1 usage  new *
def load_pascal_voc(dataset="voc/2007", bounding_box_format="xywh", train_ratio=0.8):
    ds = tfds.load(dataset, split="train+test", with_info=False, shuffle_files=True)

    # total number of examples
    ds_size = len(list(ds))
    train_size = int(train_ratio * ds_size)

    # Splitting the dataset into training and testing
    train_ds = ds.take(train_size)
    test_ds = ds.skip(train_size)

    return train_ds, test_ds


# Load training and testing datasets with a 80/20 split
train_ds, test_ds = load_pascal_voc(train_ratio=0.8)

# Visualize dataset size
print(f"Training set size: {len(list(train_ds))}")
print(f"Test set size: {len(list(test_ds))}")
```

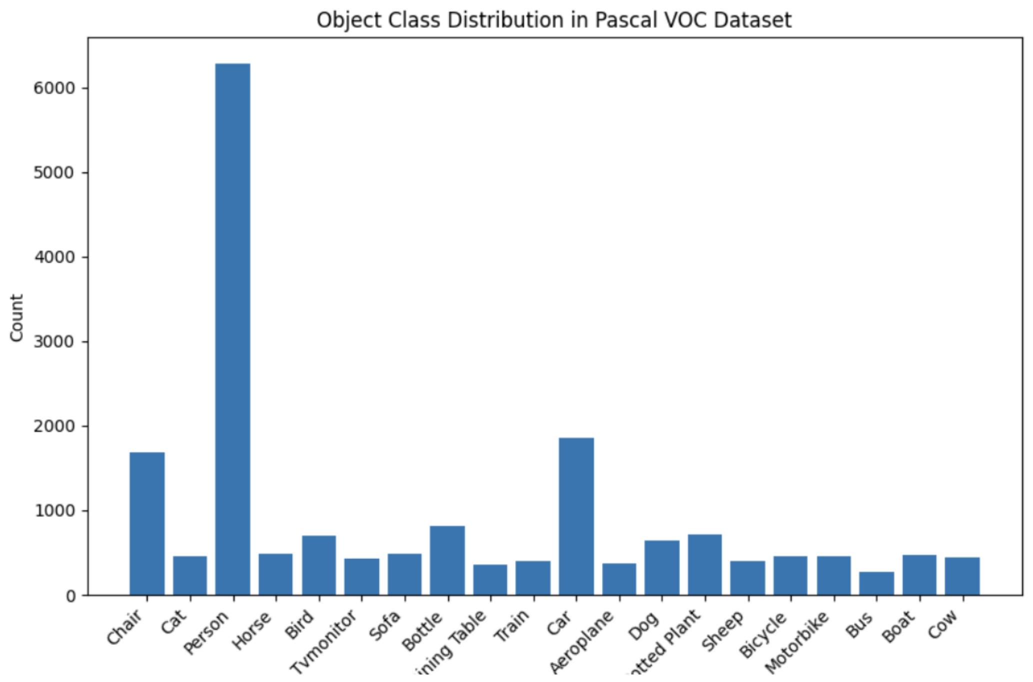


```
Training set size: 5962
Test set size: 1491
```

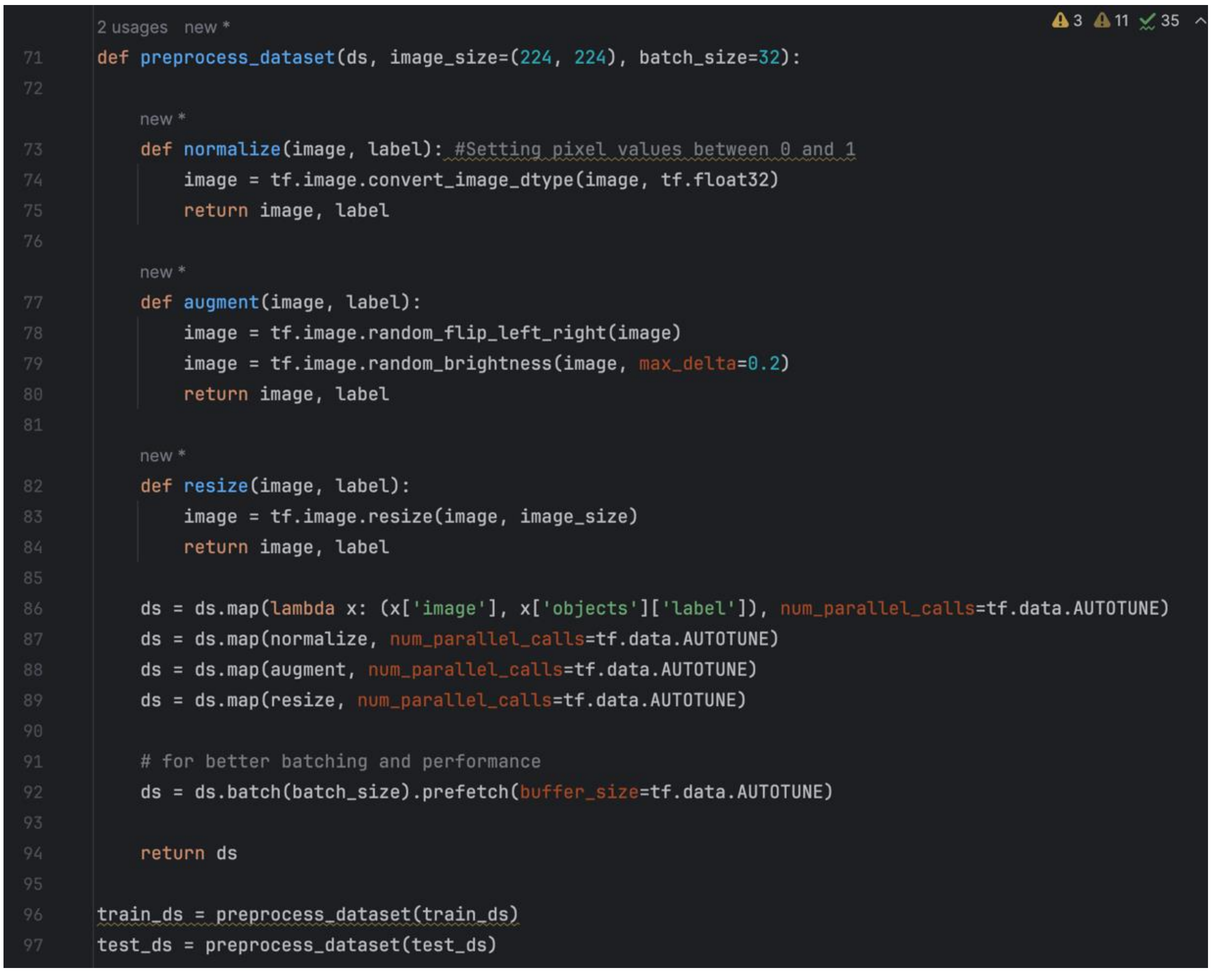


```python
#counting labels
1 usage  new *
def count_object_classes(ds):
    class_counts = collections.Counter()
    for example in ds:
        labels = example['objects']['label'].numpy()
        class_counts.update(labels)
    return class_counts

# All classes
class_ids = [
    "Aeroplane", "Bicycle", "Bird", "Boat", "Bottle", "Bus", "Car", "Cat",
    "Chair", "Cow", "Dining Table", "Dog", "Horse", "Motorbike", "Person",
    "Potted Plant", "Sheep", "Sofa", "Train", "Tvmonitor"
]

class_mapping = dict(zip(range(len(class_ids)), class_ids))

#Visualize Dataset
1 usage  new *
def visualize_class_distribution(class_counts, class_mapping):
    plt.figure(figsize=(10, 6))
    class_names = [class_mapping[i] for i in class_counts.keys()]
    class_frequencies = class_counts.values()

    plt.bar(class_names, class_frequencies)
    plt.xticks(rotation=45, ha="right")
    plt.title("Object Class Distribution in Pascal VOC Dataset")
    plt.ylabel("Count")
    plt.xlabel("Object Classes")
    plt.show()

class_counts = count_object_classes(train_ds)
visualize_class_distribution(class_counts, class_mapping)
```

Secondly, class count means that how many object can be seen in train dataset through count_oject_classes function. Then, all class ids are set. Through class mapping, it gives index for each classes such as 0,1,2.. sequentially. Furthermore, for the visualization, visualize class distribution function is used. First, the graphic size is set as 10x6. Secondly, these class indexed are converted from numbers to strings. Moreover, class frequencies are calculated through class count values. Furthermore, bar graphic became ready to visualize. The names are set as names and frequencies. It gives the how many times are there in the dataset from each class. Plt.xticks make image rotates at a 45-degree angle to provide a clearer view of the model. The result can be shown as a bar graph below. As it can be seen, how many times can be seen from which class. Moreover, there are mostly images from the human class, followed by car and chair.



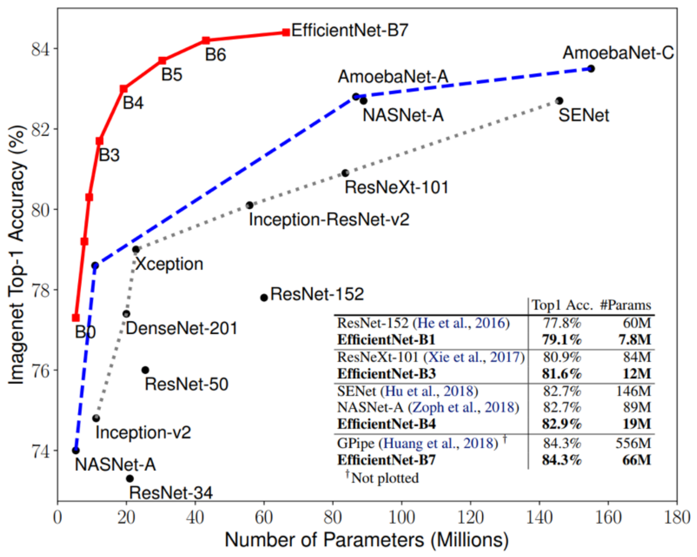
Preprocessing is applied to ensure that the model can generalize better and analyze faster. It provides the data to be put into a format suitable for the model. In this way, easier and faster analysis can be done on the data. Normalization, augmentation and resize preprocessing operations were performed in this project. First, while normalization is performed, the images are converted to float32 type. This ensures that the pixel value in the image is between 0 and 1. The large pixel values of the model make it difficult for the model to update its weights. Therefore, the pixel values are reduced to small values. The augmentation step increases the depth of the model with various variations during the training process of the model. The model sees different data and prevents overfitting. In this example, the images are randomly rotated on the horizontal axis and their brightness is changed randomly. This prevents the model from analyzing the images in the same way and allows it to be trained on different images. Finally, the resize preprocessing process is performed. In this process, it is important that the images that will enter the model are the same size. Otherwise, the system will give an error. The important thing here is to perform a resize operation suitable for the model used. The image size was selected as 224x224 pixels. Afterwards, the lambda function ensures that the model is trained only on the "image" and "label" elements. Afterwards, additional codes were written to optimize the CPU and GPU resources.



```
2 usages   new *
def preprocess_dataset(ds, image_size=(224, 224), batch_size=32):

    new *
    def normalize(image, label): #Setting pixel values between 0 and 1
        image = tf.image.convert_image_dtype(image, tf.float32)
        return image, label

    new *
    def augment(image, label):
        image = tf.image.random_flip_left_right(image)
        image = tf.image.random_brightness(image, max_delta=0.2)
        return image, label

    new *
    def resize(image, label):
        image = tf.image.resize(image, image_size)
        return image, label

    ds = ds.map(lambda x: (x['image'], x['objects']['label']), num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(normalize, num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(augment, num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(resize, num_parallel_calls=tf.data.AUTOTUNE)

    # for better batching and performance
    ds = ds.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)

    return ds

train_ds = preprocess_dataset(train_ds)
test_ds = preprocess_dataset(test_ds)
```

Model Implementation

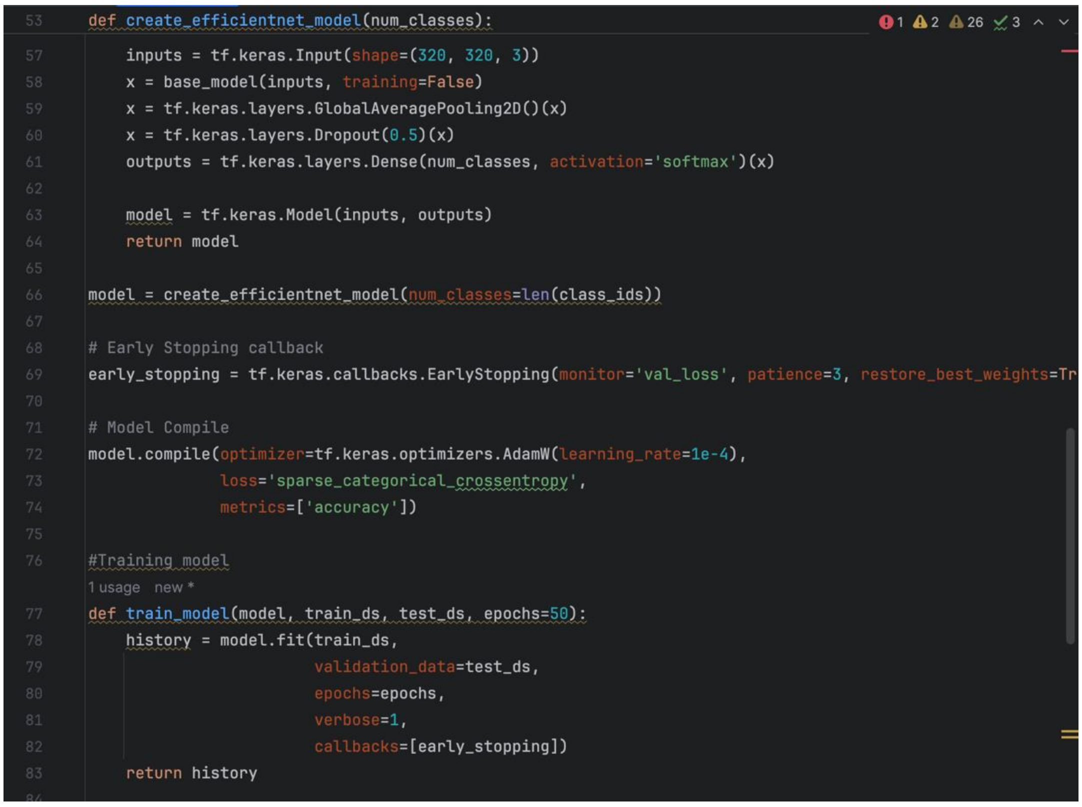
EfficientNet is chosen among one-stage architectures for this project. It applies a technique known as compound coefficient to scale models in a simple yet efficient way. Instead of randomly scaling width, depth, or resolution, compound scaling scales each dimension equally by a predetermined set of scaling coefficients. It provides the model to be efficient by distributing the depth, width and resolution in a balanced way with the compounding scale method.(Tan, M. and Le, Q., 2019) Since the dataset studied is well-labeled and small, it was decided to choose an architecture that provides a better solution with less computation. In addition, it is very effective compared to manual systems with an accuracy rate of 84 percent in ImageNet. In the Efficient model, the data augmentation process with the use of augmentation increases the accuracy of the model. It uses the parameters correctly, has better generalization ability and is an architecture that avoids overfitting. Therefore, EfficientNet was chosen in this study on the pascal dataset.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

† Not plotted

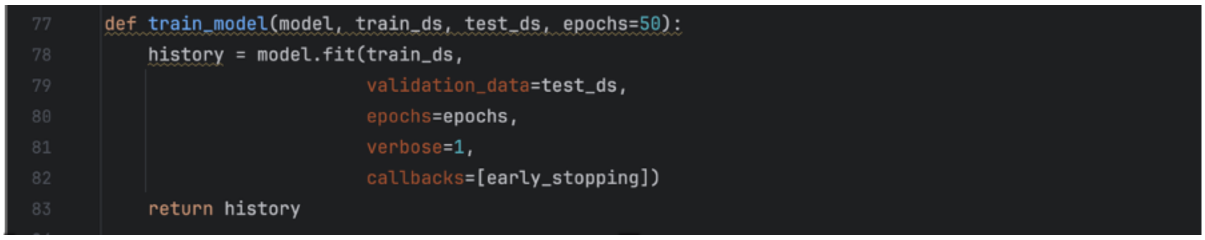
Tan, M., & Le, Q. v. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. *36th International Conference on Machine Learning, ICML 2019, 2019-June.*



```
def create_efficientnet_model(num_classes):
    inputs = tf.keras.Input(shape=(320, 320, 3))
    x = base_model(inputs, training=False)
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dropout(0.5)(x)
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

    model = tf.keras.Model(inputs, outputs)
    return model

model = create_efficientnet_model(num_classes=len(class_ids))

# Early Stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=Tr

# Model Compile
model.compile(optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#Training model
1 usage   new *
def train_model(model, train_ds, test_ds, epochs=50):
    history = model.fit(train_ds,
                        validation_data=test_ds,
                        epochs=epochs,
                        verbose=1,
                        callbacks=[early_stopping])
    return history
```

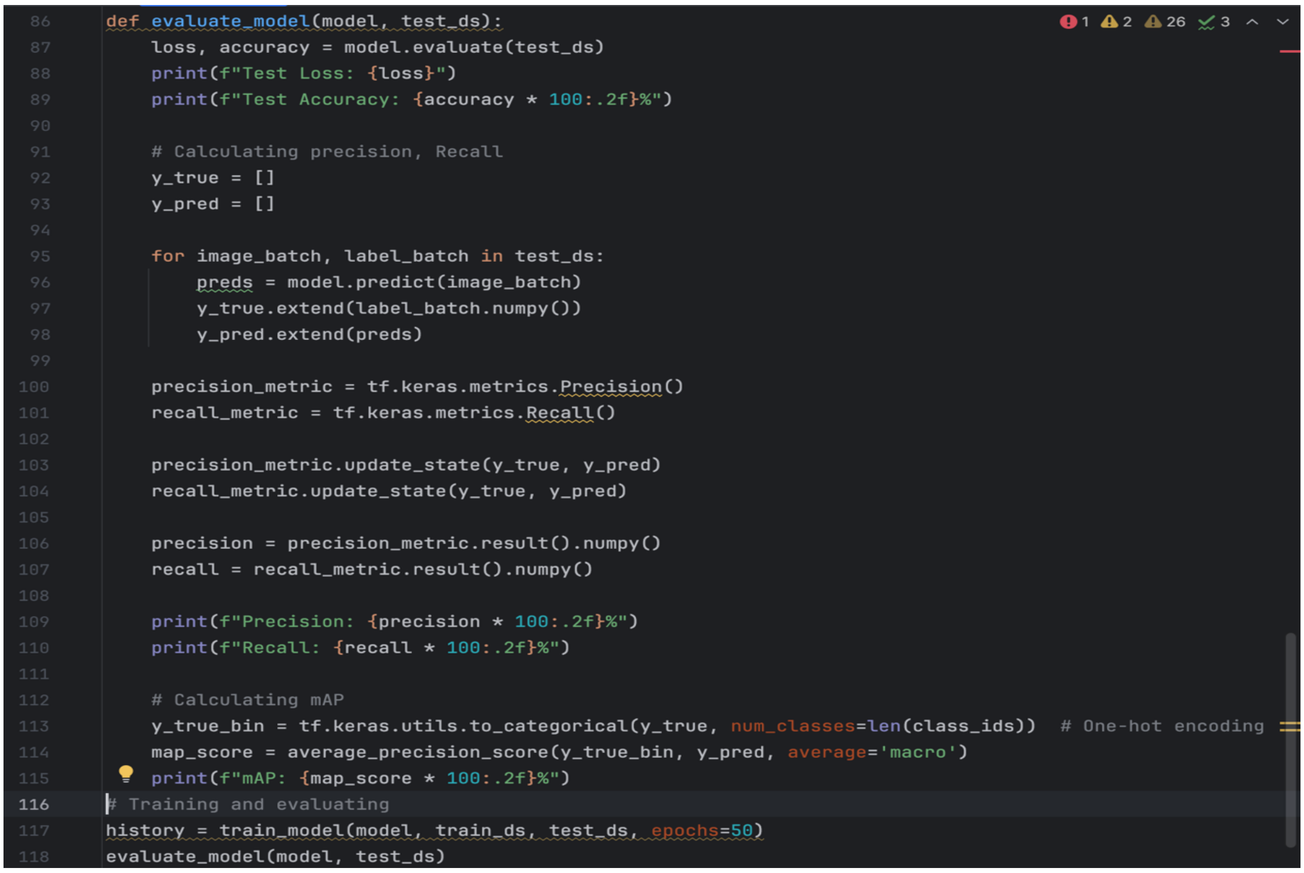
In this part of the code, an architecture that implements the Efficient model is shown. The inputs of the model are set to receive 3-channel data input at 320x320 resolution. The aim that the inputs and outputs are equal to each other. The EfficientNetB0 model is used as the base model. Other advanced models of EfficientNet are also designed based on this B0 model. In order to provide transer learning, only the last layers of the model can be learned. It compresses the feature maps into a single vector using global average pooling. Then, the dropout method randomly inactivates half of the neurons and prevents the model from overfitting. The dense layer is the last classification layer of the model. It gives the probability distribution using the softmax activation function. Then, the model is called according to the number of classes. Moreover, if the verification loss does not increase for 3 epochs, the model is stopped thanks to early stopping. In the model compile section, the adamw optimizer has been added so that the model parameters do not grow too much and the model can work more efficiently. Sparse categorical crossentropy is the classification loss function. It is used when the labels are integers. Finally, the accuracy metric is used to measure the performance of the model.

Model Training and Evaluation



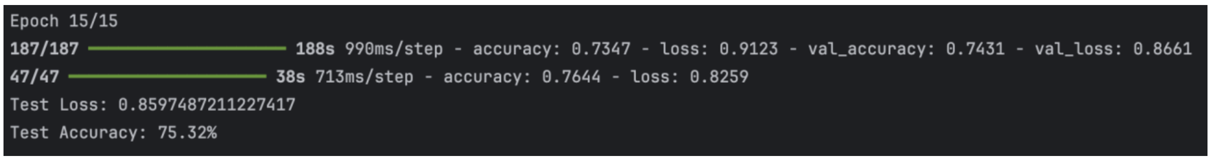
```
def train_model(model, train_ds, test_ds, epochs=50):
    history = model.fit(train_ds,
                        validation_data=test_ds,
                        epochs=epochs,
                        verbose=1,
                        callbacks=[early_stopping])
    return history
```

This section is utilized to train the model using training data and to monitor its performance using validation data. The EfficientNet model is trained for a specified number of epochs using the train_model function, which is set to 50 by default. The model is initially trained using the training dataset (train_ds). The validation dataset (test_ds) is used to evaluate the model's performance at the conclusion of each epoch during training. This procedure is essential for the purpose of assessing the model's overall performance and preventing it from overfitting during training. At the conclusion of each epoch, the model.fit() function computes the loss and accuracy on the validation set and trains the model with training data. The finest weights are restored by the early stopping callback function, which terminates training early when the validation loss does not improve. Consequently, a history object is returned, which enables analysis of the model's training process and the subsequent changes in metrics such as accuracy and loss.



```
def evaluate_model(model, test_ds):
    loss, accuracy = model.evaluate(test_ds)
    print(f"Test Loss: {loss}")
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

    # Calculating precision, Recall
    y_true = []
    y_pred = []

    for image_batch, label_batch in test_ds:
        preds = model.predict(image_batch)
        y_true.extend(label_batch.numpy())
        y_pred.extend(preds)

    precision_metric = tf.keras.metrics.Precision()
    recall_metric = tf.keras.metrics.Recall()

    precision_metric.update_state(y_true, y_pred)
    recall_metric.update_state(y_true, y_pred)

    precision = precision_metric.result().numpy()
    recall = recall_metric.result().numpy()

    print(f"Precision: {precision * 100:.2f}%")
    print(f"Recall: {recall * 100:.2f}%")

    # Calculating mAP
    y_true_bin = tf.keras.utils.to_categorical(y_true, num_classes=len(class_ids))  # One-hot encoding
    map_score = average_precision_score(y_true_bin, y_pred, average='macro')
    print(f"mAP: {map_score * 100:.2f}%")
# Training and evaluating
history = train_model(model, train_ds, test_ds, epochs=50)
evaluate_model(model, test_ds)
```

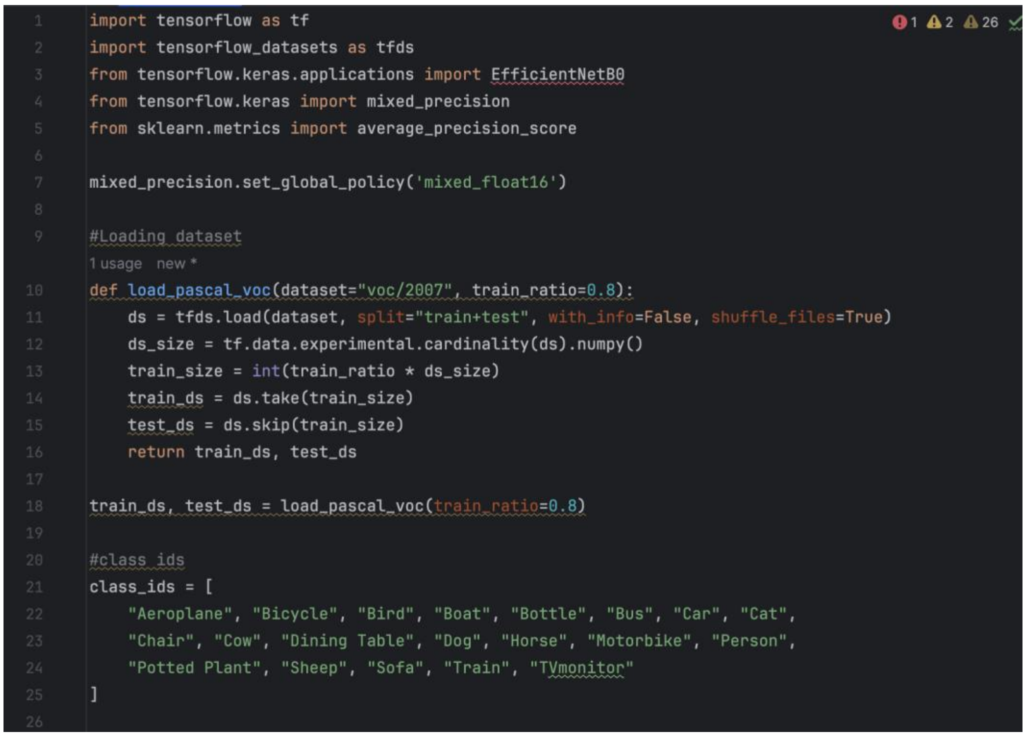
The last part of the model was intended to calculate the model's accuracy, precision, recall and mAP metrics. The Evaluate model function first calculates the model loss and accuracy values. The model makes predictions for each dataset with the test set cycle and these predictions are recorded in y_pred. Ready-made metrics are used to calculate precision and recall. Finally, one-hot-coding is done to create the mAp metric. The average_precision_score function is used and the probabilities predicted by the model are compared with the real labels and the mAP value is calculated.



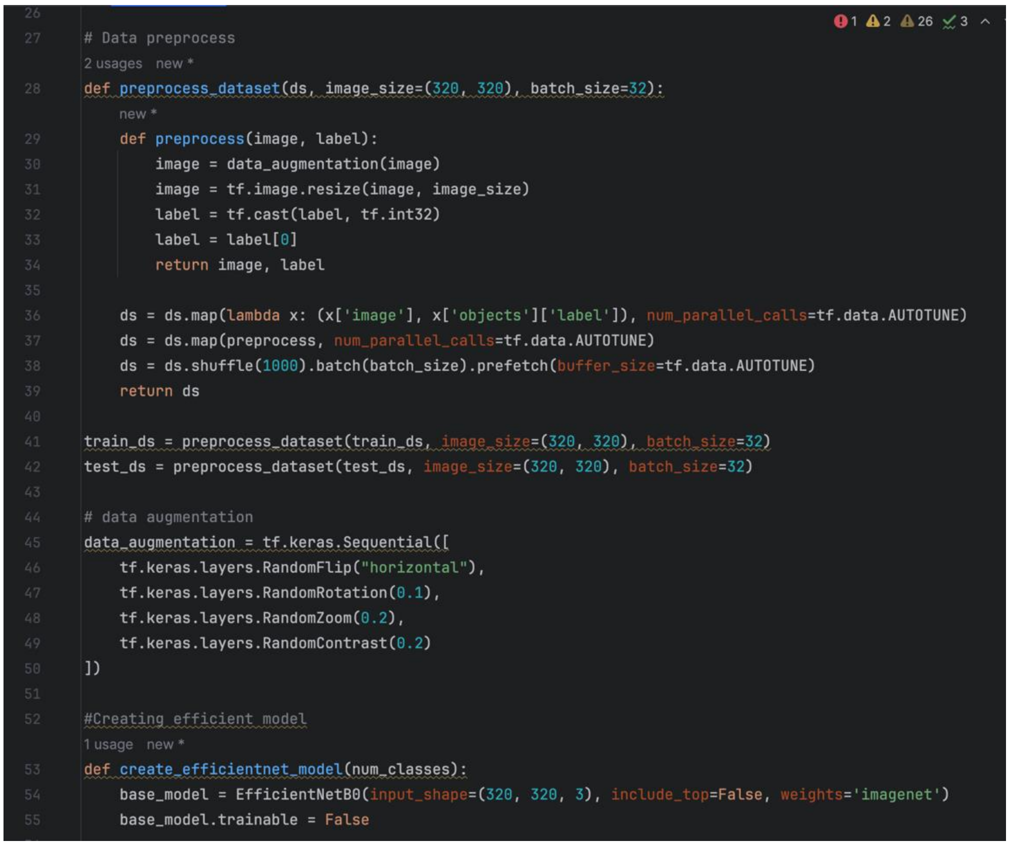
```
Epoch 15/15
187/187 ━━━━━━━━━━━━━━━━━━━━ 188s 990ms/step - accuracy: 0.7347 - loss: 0.9123 - val_accuracy: 0.7431 - val_loss: 0.8661
47/47 ━━━━━━━━━━━━━━━━━━━━ 38s 713ms/step - accuracy: 0.7644 - loss: 0.8259
Test Loss: 0.8597487211227417
Test Accuracy: 75.32%
```

This model gave 75 percent accuracy. When we look at other models that are particularly effective in object detection, for example, Faster R-CNN gives an average accuracy of 75 to 80 percent. The YOLO model is between 80-85 percent. Studies conducted so far show that EfficientNet has an accuracy of 82-82 percent on the Pascal dataset. In line with this information, it can be stated that the EfficientNetB0 model used in this project gives high accuracy. Of course, an increase of 7-8 percent can be achieved. For example, more advanced B1, B2, B3 models can be used. Overfitting of the model can be prevented by using data augmentation techniques and its performance can be increased. Another option is to increase the accuracy by playing with hyperparameters. As a result, slightly lower results were obtained than other advanced model results. However, performance can be increased with the necessary improvements.

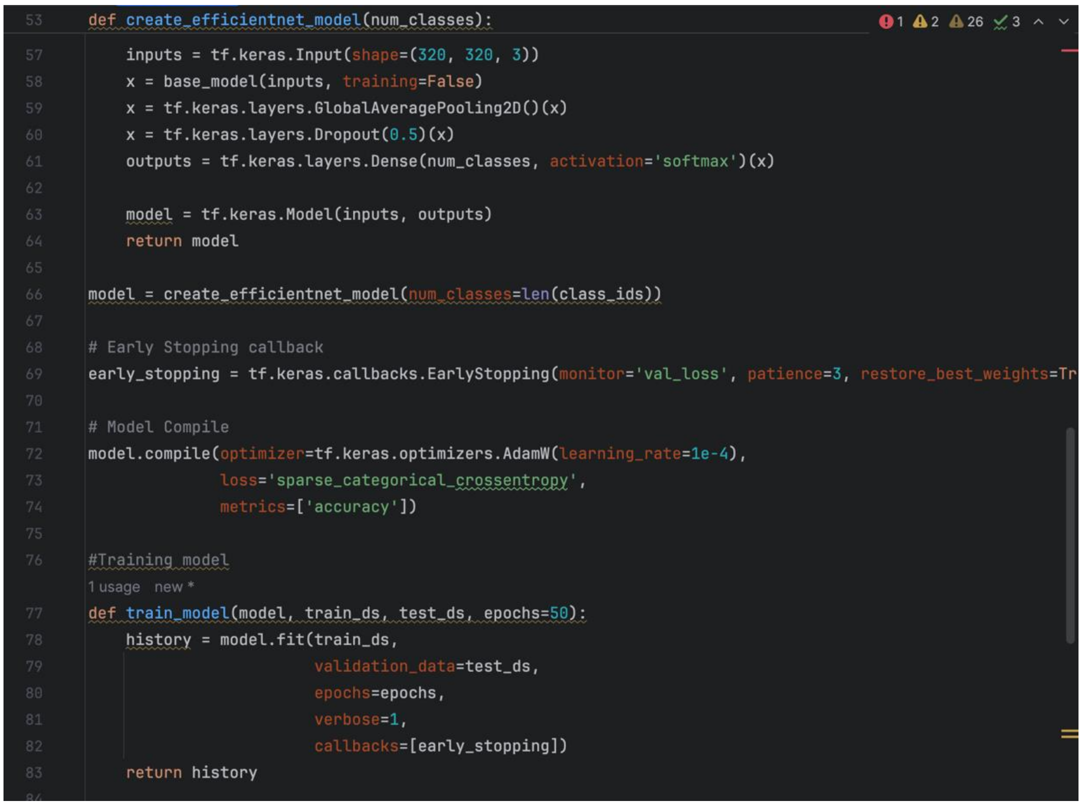
Practical Application



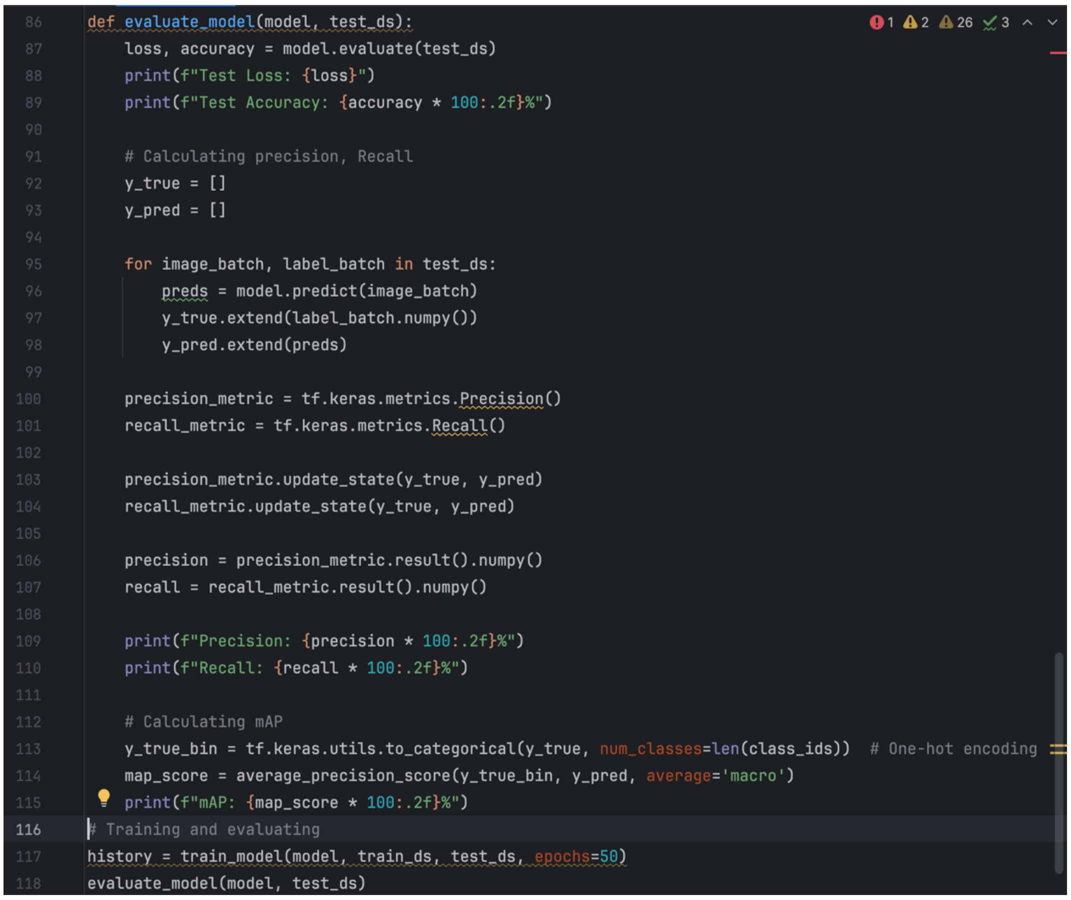
```python
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras import mixed_precision
from sklearn.metrics import average_precision_score

mixed_precision.set_global_policy('mixed_float16')

#Loading dataset
def load_pascal_voc(dataset="voc/2007", train_ratio=0.8):
    ds = tfds.load(dataset, split="train+test", with_info=False, shuffle_files=True)
    ds_size = tf.data.experimental.cardinality(ds).numpy()
    train_size = int(train_ratio * ds_size)
    train_ds = ds.take(train_size)
    test_ds = ds.skip(train_size)
    return train_ds, test_ds

train_ds, test_ds = load_pascal_voc(train_ratio=0.8)

#class ids
class_ids = [
    "Aeroplane", "Bicycle", "Bird", "Boat", "Bottle", "Bus", "Car", "Cat",
    "Chair", "Cow", "Dining Table", "Dog", "Horse", "Motorbike", "Person",
    "Potted Plant", "Sheep", "Sofa", "Train", "TVmonitor"
]
```



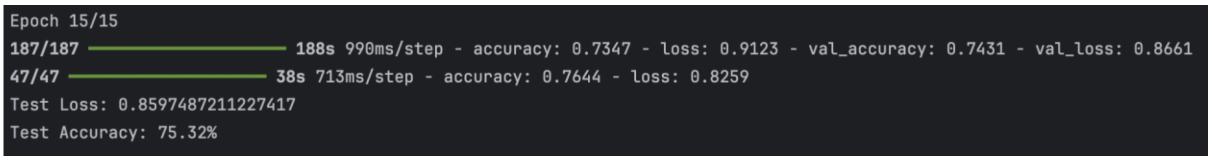
```
# Data preprocess
2 usages   new *
def preprocess_dataset(ds, image_size=(320, 320), batch_size=32):
    new *
    def preprocess(image, label):
        image = data_augmentation(image)
        image = tf.image.resize(image, image_size)
        label = tf.cast(label, tf.int32)
        label = label[0]
        return image, label

    ds = ds.map(lambda x: (x['image'], x['objects']['label']), num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.shuffle(1000).batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)
    return ds

train_ds = preprocess_dataset(train_ds, image_size=(320, 320), batch_size=32)
test_ds = preprocess_dataset(test_ds, image_size=(320, 320), batch_size=32)

# data augmentation
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.2),
    tf.keras.layers.RandomContrast(0.2)
])

#Creating efficient model
1 usage   new *
def create_efficientnet_model(num_classes):
    base_model = EfficientNetB0(input_shape=(320, 320, 3), include_top=False, weights='imagenet')
    base_model.trainable = False
```



```
def create_efficientnet_model(num_classes):

    inputs = tf.keras.Input(shape=(320, 320, 3))
    x = base_model(inputs, training=False)
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x = tf.keras.layers.Dropout(0.5)(x)
    outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

    model = tf.keras.Model(inputs, outputs)
    return model

model = create_efficientnet_model(num_classes=len(class_ids))

# Early Stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=Tr

# Model Compile
model.compile(optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#Training model
1 usage   new *
def train_model(model, train_ds, test_ds, epochs=50):
    history = model.fit(train_ds,
                        validation_data=test_ds,
                        epochs=epochs,
                        verbose=1,
                        callbacks=[early_stopping])
    return history
```



```python
def evaluate_model(model, test_ds):
    loss, accuracy = model.evaluate(test_ds)
    print(f"Test Loss: {loss}")
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

    # Calculating precision, Recall
    y_true = []
    y_pred = []

    for image_batch, label_batch in test_ds:
        preds = model.predict(image_batch)
        y_true.extend(label_batch.numpy())
        y_pred.extend(preds)

    precision_metric = tf.keras.metrics.Precision()
    recall_metric = tf.keras.metrics.Recall()

    precision_metric.update_state(y_true, y_pred)
    recall_metric.update_state(y_true, y_pred)

    precision = precision_metric.result().numpy()
    recall = recall_metric.result().numpy()

    print(f"Precision: {precision * 100:.2f}%")
    print(f"Recall: {recall * 100:.2f}%")

    # Calculating mAP
    y_true_bin = tf.keras.utils.to_categorical(y_true, num_classes=len(class_ids))  # One-hot encoding
    map_score = average_precision_score(y_true_bin, y_pred, average='macro')
    print(f"mAP: {map_score * 100:.2f}%")
# Training and evaluating
history = train_model(model, train_ds, test_ds, epochs=50)
evaluate_model(model, test_ds)
```



```
Epoch 15/15
187/187 ━━━━━━━━━━━━━━━━━━━━ 188s 990ms/step - accuracy: 0.7347 - loss: 0.9123 - val_accuracy: 0.7431 - val_loss: 0.8661
47/47 ━━━━━━━━━━━━━━━━━━━━ 38s 713ms/step - accuracy: 0.7644 - loss: 0.8259
Test Loss: 0.8597487211227417
Test Accuracy: 75.32%
```

Computer vision artificial intelligence models are seen in many applications in daily life. The most important of these are transportation and healthcare fields. It has become a necessary factor for the safety and efficiency of society in areas such as autonomous vehicles and traffic optimization. Especially today, self-driving cars provide high accuracy results to prevent traffic accidents. Another important area where it is used in real life is healthcare. It has started to take place in vital points such as X-ray, MRI, cancer detection. In fact, it has been observed that radiology images can be analyzed at the same rate as doctors today. With the Brain-Connectum project launched in 2015, for the first time, information about the brain began to be obtained by examining human MRIs thanks to artificial intelligence.(Shekouh, 2024)It is used to detect diseases such as internal bleeding or tumors that doctors cannot see with the naked eye. Another important development in the field of health is that breast cancer can be detected 5 years before it matures.

Conclusion and Future Scope

This Efficient model, which has been developed and has a 75 percent accuracy rate, can be applied in many real applications in the field of object detection in daily life. For example, these are security and monitoring systems, autonomous vehicles, medical imaging areas. It can be used to analyze and optimize traffic or detect theft. In addition, object detection is quite useful in autonomous vehicles. It can be used without detecting pedestrians, other vehicles or traffic signs. Another important area where it can be used is medical imaging. The model can be useful in the detection of tumors, cysts and other anomalies.
The Pascal dataset has approximately 10,000 images and 20 classes. Training the model with a larger dataset instead of this dataset can improve performance. More advanced algorithms can be added to make the model better able to recognize small or overlapping objects. More advanced algorithms can be added to make the model better able to recognize small or overlapping objects.

References

- Hubel, D. H., and Wiesel, T. N. (1962) ‘Receptive fields, binocular interaction and functional architecture in the cat's visual cortex’, *The Journal of Physiology*, 160, pp. 106–154.

- Shekouh, D., Sadat Kaboli, H., Ghaffarzadeh-Esfahani, M., Khayamdar, M., Hamedani, Z., Oraee-Yazdani, S., Zali, A., and Amanzadeh, E. (2024) ‘Artificial intelligence role in advancement of human brain connectome studies’, *Frontiers in Neuroinformatics*, 18, p. 1399931.

- Tan, M. and Le, Q. (2019) ‘EfficientNet: Rethinking model scaling for convolutional neural networks’, *36th International Conference on Machine Learning (ICML)*, June 2019.

- Zou, Z., Chen, K., Shi, Z., Guo, Y. and Ye, J. (2023) ‘Object detection in 20 years: A survey’, *Proceedings of the IEEE*, 111(3), pp. 257-276.