Table of Contents

1. Introduction to VOC2008 problem (Object Classification only)

Image classification is one of the most essential tasks in computer vision. It has numerous practical applications in the field of medicine, surveillance, and security. In image classification, the goal is to assign a label to an image. The VOC2008 dataset is one of the popular datasets for object classification tasks. In this report, we will discuss the problem statement, dataset description, evaluation metrics, and our methodology for object classification using the VOC2008 dataset as our project was based only on object classification from the images provided in the dataset.

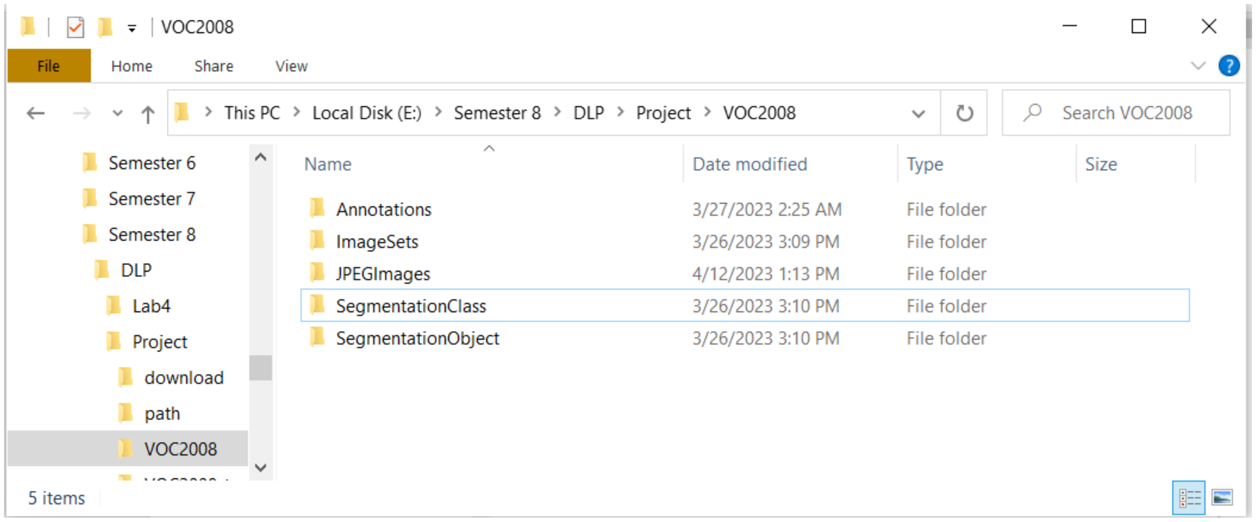
1.1. Problem Statement

The problem statement is to classify images into different object classes. The VOC2008 dataset has 20 object classes, and our objective is to train a model that can accurately classify images into these object classes. The 20 object classes in the VOC2008 dataset are:

aeroplane	Bottle	chair	horse	sheep
bicycle	Bus	cow	motorbike	sofa
bird	Car	diningtable	person	train
boat	Cat	dog	pottedplant	tvmonitor

1.2. Dataset description

The dataset had a total of 9,963 images, with each image having one or more objects. The dataset contains 5 folders as shown in figure 1:



Name	Date modified	Type	Size
Annotations	3/27/2023 2:25 AM	File folder	
ImageSets	3/26/2023 3:09 PM	File folder	
JPEGImages	4/12/2023 1:13 PM	File folder	
SegmentationClass	3/26/2023 3:10 PM	File folder	
SegmentationObject	3/26/2023 3:10 PM	File folder	

Figure 1: Directory Structure of VOC2008 dataset

- Annotations: This folder contains XML files that contain labels of all the identified entries within an image with their bounding boxes within which that object is found.
- ImageSets: This folder contains 3 sub-directories, namely Layout, Segmentation, and Main. The Main folder is relevant to us and it contains {object}_train, {object}_test, {object}_val, and {object}_trainval.txt files where {object} refers to the name of distinct objects.
- JPEGImages: This folder contains 6336 jpeg images of training and validation data with each being 300 x 300 pixels size.
- SegmentationClass and SegmentationObjects contain data for individual identified entities within a picture, the data which is not relevant in context of this project and its objectives.

1.3. Evaluation Metrics

The performance of the object classification model is evaluated using mean average precision (mAP). mAP is a widely used metric for object classification tasks. It is calculated by averaging the precision-recall curve for each class. Precision is the ratio of true positive predictions to the total number of positive predictions, and recall is the ratio of true positive predictions to the total number of actual positive samples. The mAP score ranges from 0 to 1, with higher scores indicating better performance.

2. Methodology

Being focused on Object Classification, our project was divided in 4 sub parts:

1. An introductory report
2. Training 5 CNNs and NNs for Object Classification
3. Ensembled Learning using Late and Early Fusion
4. Final Project Report

Following sections represent some steps we took to reach the training stage.

2.1. Obtaining Images and Labels from Dataset folder

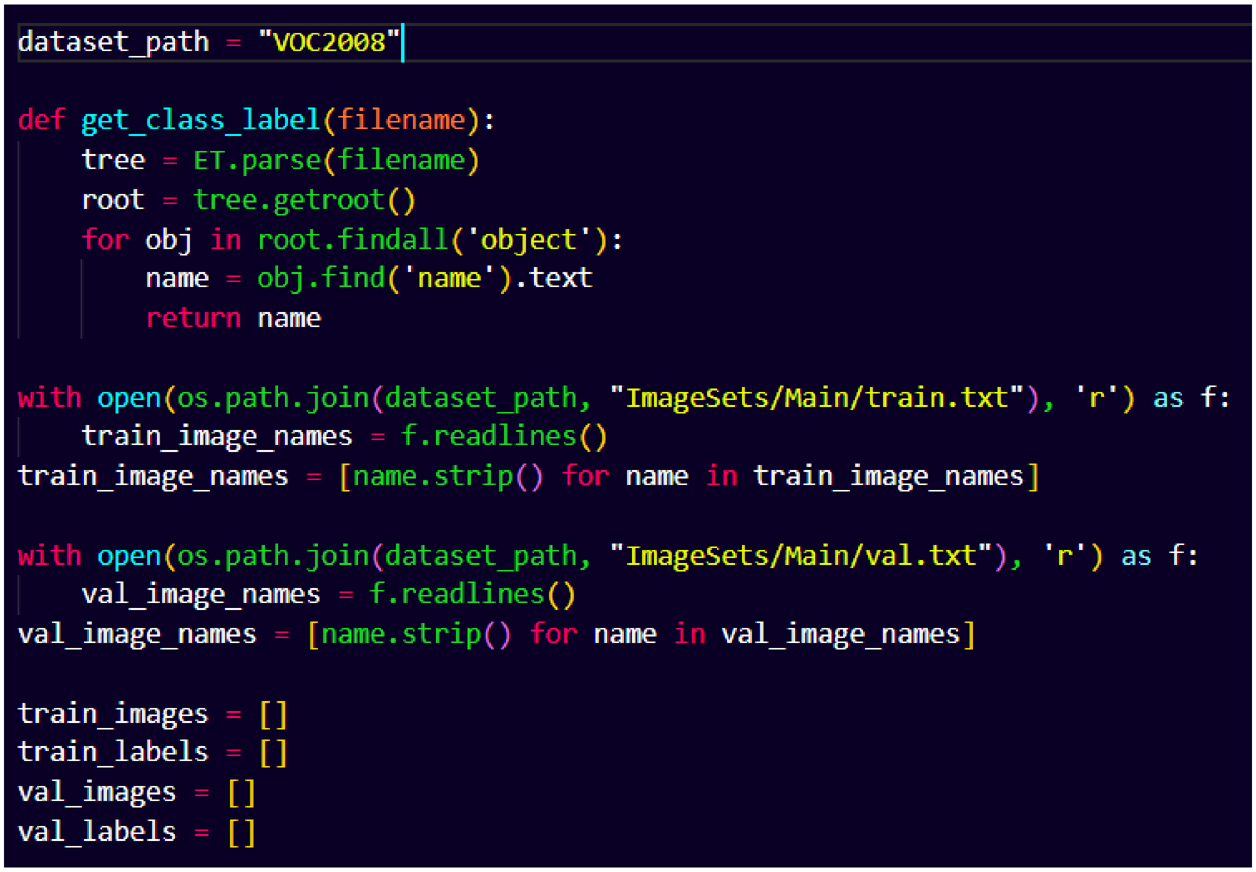
VOC2008 contained 3 folders that were valuable for us i.e.:

- Annotations

- ImageSets
- JPEGImages

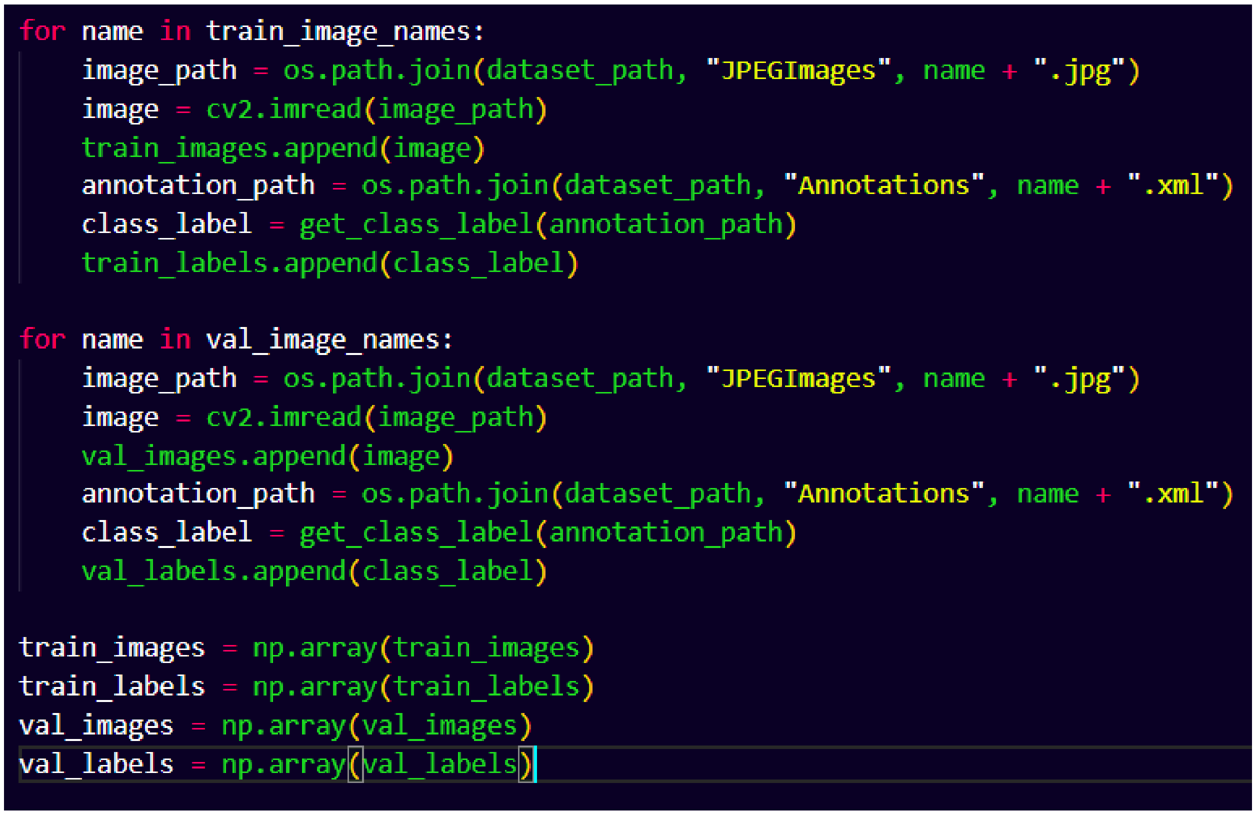
Steps followed:

1. We acquired names of images pre-determined for training and validation data, defined in VOC2008/ImageSets/Main/train.txt and VOC2008/ImageSets/Main/val.txt.
2. Then, we acquired the labels of training and validation data using {image_name}.xml files within the Annotations folder where {image_name} was derived from the names acquired from train.txt and val.txt.
3. Lastly, we obtained images for training and validation data respectively using cv2's imread() and numpy's image to array function from VOC2008/JPEGImages/ using {image_name}.jpg (changed extension) where {image_name} is derived from step 1.



```python
dataset_path = "VOC2008"

def get_class_label(filename):
    tree = ET.parse(filename)
    root = tree.getroot()
    for obj in root.findall('object'):
        name = obj.find('name').text
        return name

with open(os.path.join(dataset_path, "ImageSets/Main/train.txt"), 'r') as f:
    train_image_names = f.readlines()
train_image_names = [name.strip() for name in train_image_names]

with open(os.path.join(dataset_path, "ImageSets/Main/val.txt"), 'r') as f:
    val_image_names = f.readlines()
val_image_names = [name.strip() for name in val_image_names]

train_images = []
train_labels = []
val_images = []
val_labels = []
```

Figure 2a: Obtaining names of images within training and validation datasets

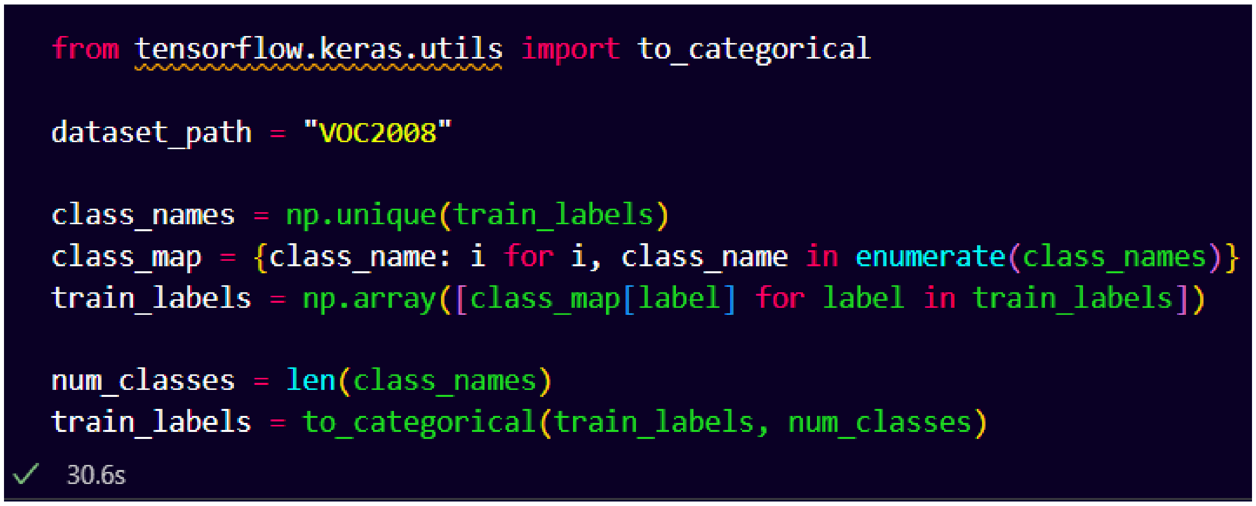


```
for name in train_image_names:
    image_path = os.path.join(dataset_path, "JPEGImages", name + ".jpg")
    image = cv2.imread(image_path)
    train_images.append(image)
    annotation_path = os.path.join(dataset_path, "Annotations", name + ".xml")
    class_label = get_class_label(annotation_path)
    train_labels.append(class_label)

for name in val_image_names:
    image_path = os.path.join(dataset_path, "JPEGImages", name + ".jpg")
    image = cv2.imread(image_path)
    val_images.append(image)
    annotation_path = os.path.join(dataset_path, "Annotations", name + ".xml")
    class_label = get_class_label(annotation_path)
    val_labels.append(class_label)

train_images = np.array(train_images)
train_labels = np.array(train_labels)
val_images = np.array(val_images)
val_labels = np.array(val_labels)
```

Figure 2b: Obtaining images and their labels

2.2. Mapping class labels to integers for modelling

In order to enable measurement of categorization loss and make calculations easier and mappable for the model, we transformed textual labels to a range of integer labels.

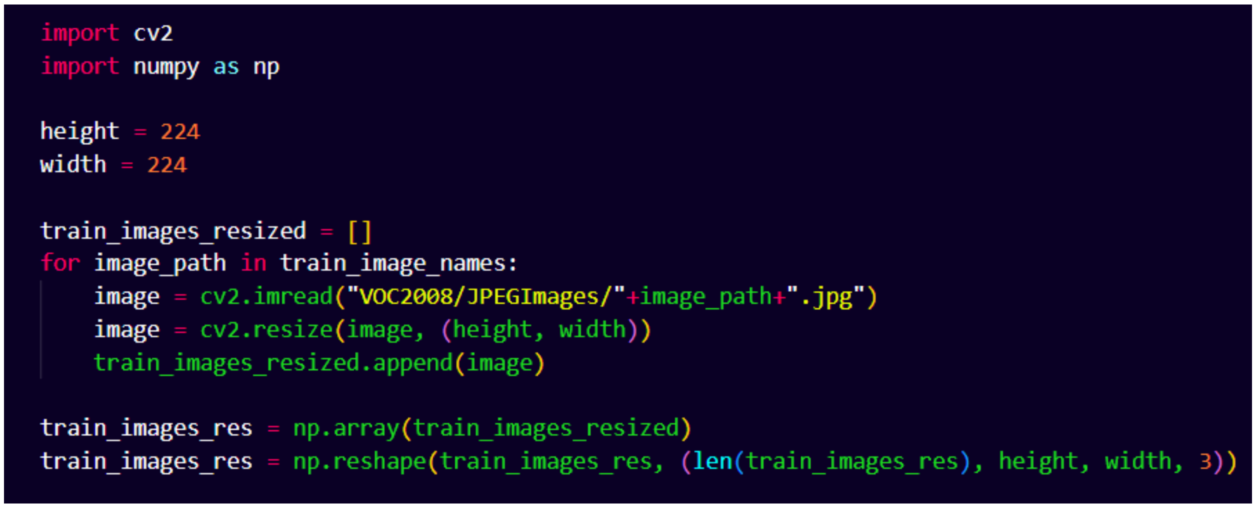


```
from tensorflow.keras.utils import to_categorical

dataset_path = "VOC2008"

class_names = np.unique(train_labels)
class_map = {class_name: i for i, class_name in enumerate(class_names)}
train_labels = np.array([class_map[label] for label in train_labels])

num_classes = len(class_names)
train_labels = to_categorical(train_labels, num_classes)
✓ 30.6s
```

Figure 3: Conversion of textual labels to Integer Categorical labels

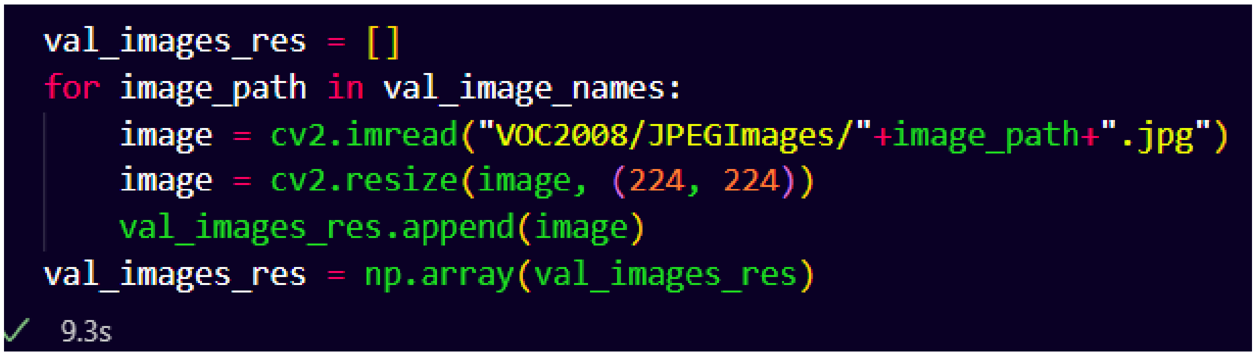
2.3. Resizing Validation and Training Images

In order to make our 300 x 300 images capable of being trained by different CNNs of (224,224,3) shape, we had to reshape our images. Following code helped us achieve our cause.



```python
import cv2
import numpy as np

height = 224
width = 224

train_images_resized = []
for image_path in train_image_names:
    image = cv2.imread("VOC2008/JPEGImages/"+image_path+".jpg")
    image = cv2.resize(image, (height, width))
    train_images_resized.append(image)

train_images_res = np.array(train_images_resized)
train_images_res = np.reshape(train_images_res, (len(train_images_res), height, width, 3))
```

Figure 4a: Resizing training images



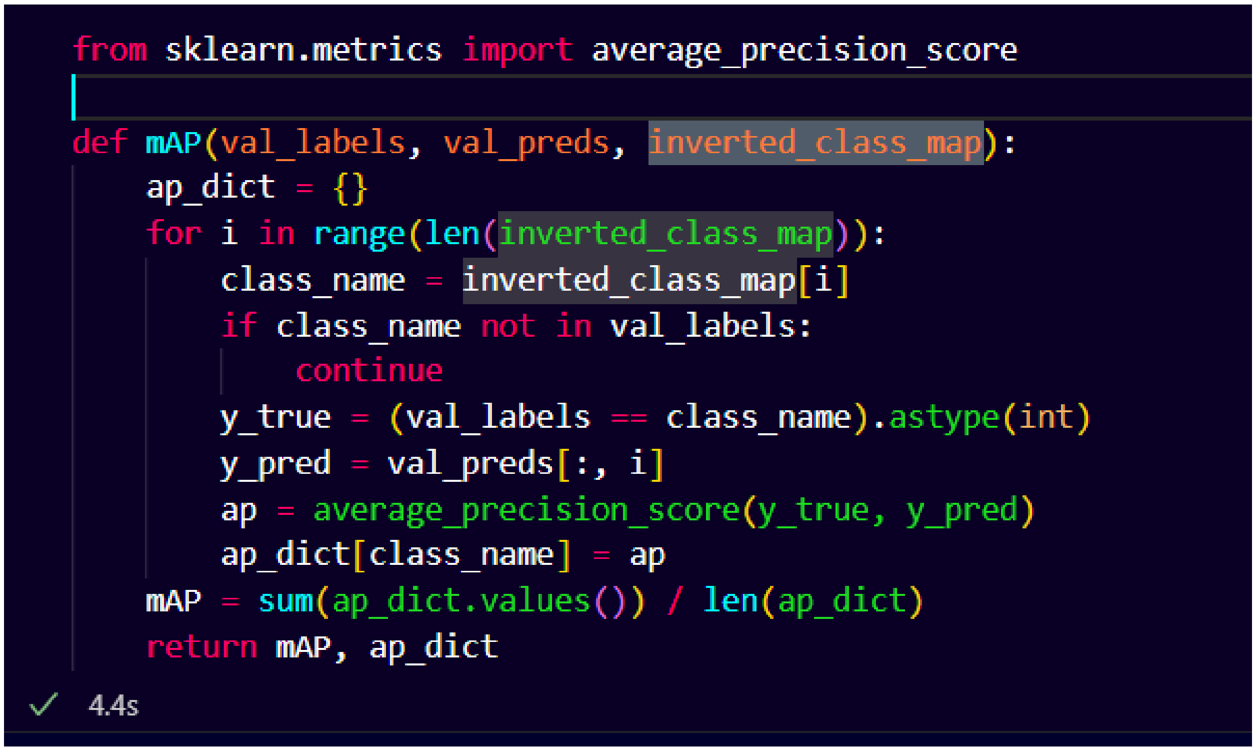
```python
val_images_res = []
for image_path in val_image_names:
    image = cv2.imread("VOC2008/JPEGImages/"+image_path+".jpg")
    image = cv2.resize(image, (224, 224))
    val_images_res.append(image)
val_images_res = np.array(val_images_res)
```

```
✓ 9.3s
```

Figure 4b: Resizing validation images

2.4. Defining Accuracy and Mean Precision Average

As defined in section 1.3 (Evaluation Metrics), Mean Precision Average was used to measure the quality of produced output (and the working principle was also shared).

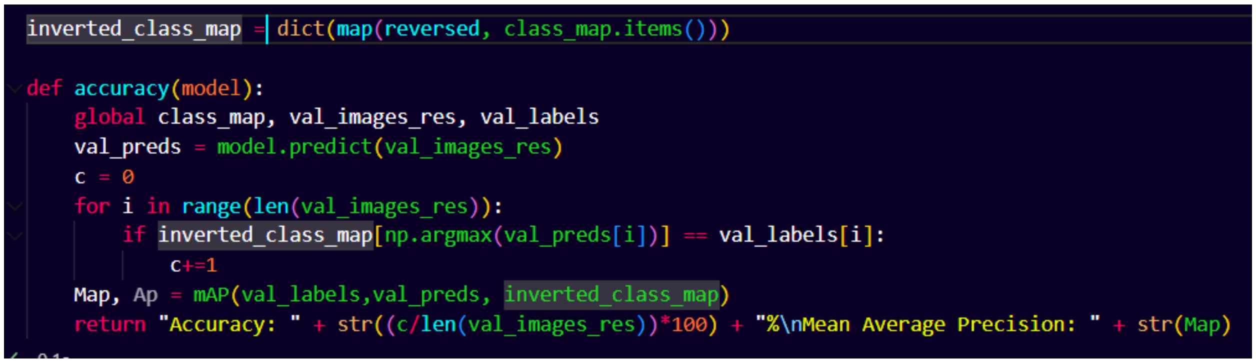


```python
from sklearn.metrics import average_precision_score

def mAP(val_labels, val_preds, inverted_class_map):
    ap_dict = {}
    for i in range(len(inverted_class_map)):
        class_name = inverted_class_map[i]
        if class_name not in val_labels:
            continue
        y_true = (val_labels == class_name).astype(int)
        y_pred = val_preds[:, i]
        ap = average_precision_score(y_true, y_pred)
        ap_dict[class_name] = ap
    mAP = sum(ap_dict.values()) / len(ap_dict)
    return mAP, ap_dict
```

✓ 4.4s

Figure 5a: Mean Precision Average custom function

For accuracy, we defined our own function which is given as follows. This function takes in a trained model, and acquires predicted labels to calculate the accuracy against the true labels. It also uses an inverted_class_map which has form {object_name}:{integer value it has been mapped to} to know which integer represents which class.



```python
inverted_class_map = dict(map(reversed, class_map.items()))

def accuracy(model):
    global class_map, val_images_res, val_labels
    val_preds = model.predict(val_images_res)
    c = 0
    for i in range(len(val_images_res)):
        if inverted_class_map[np.argmax(val_preds[i])] == val_labels[i]:
            c+=1
    Map, Ap = mAP(val_labels,val_preds, inverted_class_map)
    return "Accuracy: " + str((c/len(val_images_res))*100) + "%\nMean Average Precision: " + str(Map)
```

Figure 5b: Calculating the Accuracy of a model

3. CNNs and NNs

We were assigned to choose at least 5 popular Convolutional Neural Networks (CNNs) and use transfer learning (via pre-trained models) with a custom Neural Network added to classify the 20 objects within the images of VOC2008 dataset.

Our group came to a conclusion to include the following CNNs:

- VGG16
- ResNet50
- DenseNet
- MobileNet
- InceptionV3

CNN Architecture	Year	Parameters	Depth	ImageNet Top-1 Accuracy	ImageNet Top-5 Accuracy	Transfer Learning Effectiveness
VGG16 [1]	2014	138 million	16	71.5%	90.8%	Good
ResNet50 [2]	2015	25.6 million	50	75.3%	92.2%	Very Good
MobileNetV2 [4]	2018	3.4 million	88	71.8%	91.0%	Good
InceptionV3 [5]	2015	23.9 million	159	78.0%	94.4%	Very Good
DenseNet121 [3]	2016	8.1 million	121	74.9%	92.2%	Good

Table1: General information about 5 CNNs we used, derived from sources [1-5]

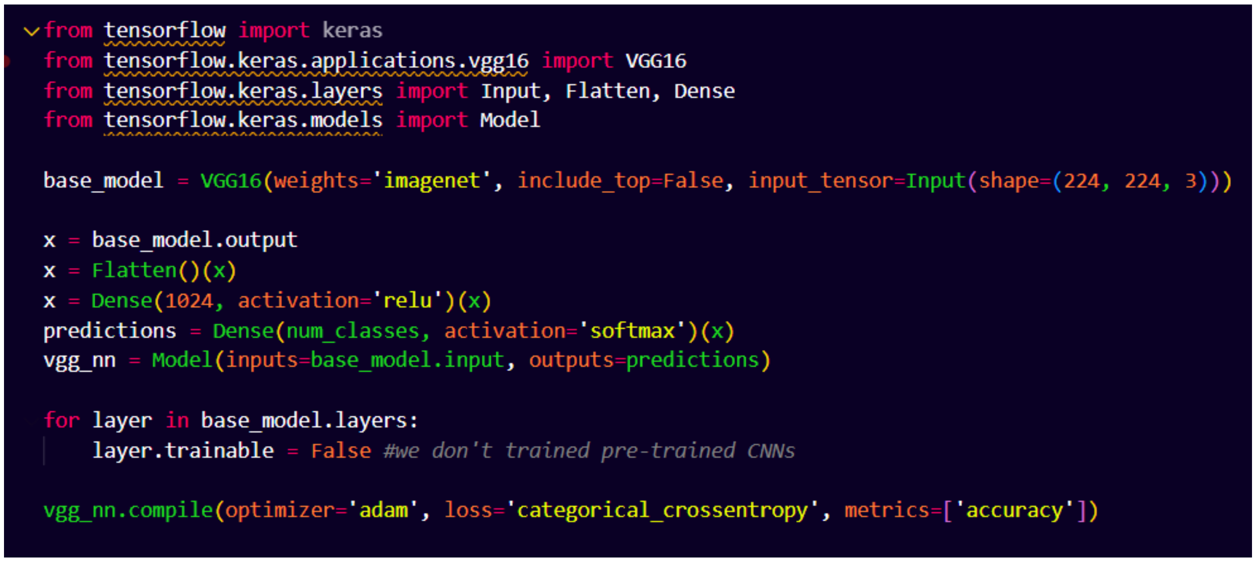
3.1. VGG-16

3.1.1. Brief Introduction about VGG

The VGG (Visual Geometry Group) network is a deep convolutional neural network, which is widely used in image recognition tasks, including object classification, object detection, and semantic segmentation. The VGG network was developed by the Visual Geometry Group at the University of Oxford. The VGG network is a deep neural network with 19 layers, which includes convolutional layers, max-pooling layers, and fully connected layers. The VGG network uses small 3x3 convolutional filters with a stride of one to extract features from the input image. The VGG network also uses max-pooling layers with a 2x2 kernel and a stride of 2 to reduce the spatial dimension of the feature maps [1].

3.1.2. How did we use VGG?

In our implementation, we used the pre-trained VGG16 model, which was trained on the ImageNet dataset, as a feature extractor. We replaced the fully connected layers of the VGG16 model with our own fully connected layers to perform object classification on the VOC2008 dataset. We also fine-tuned the last convolutional block of the VGG16 model to improve the accuracy of the model on the VOC2008 dataset.

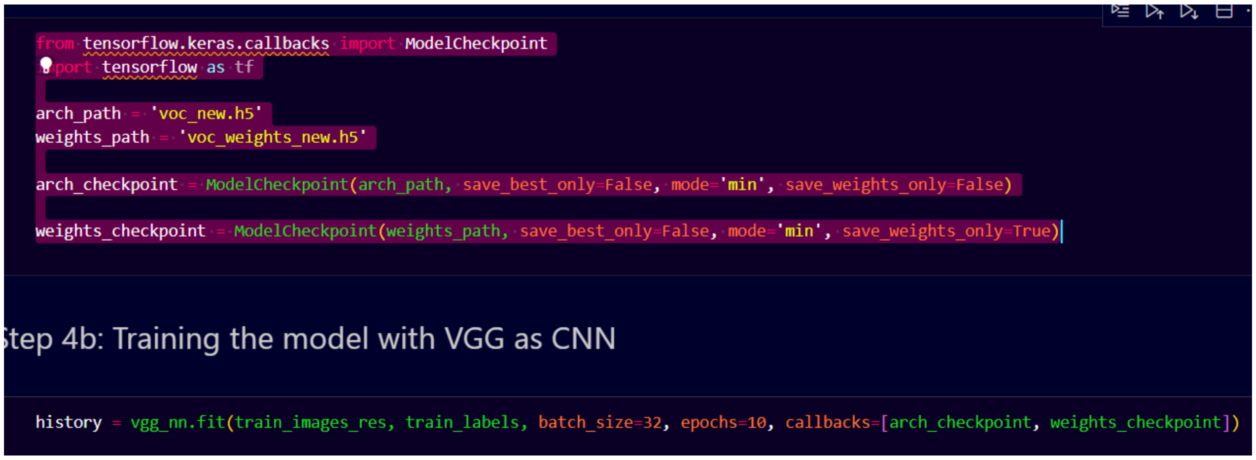


```
from tensorflow import keras
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Input, Flatten, Dense
from tensorflow.keras.models import Model

base_model = VGG16(weights='imagenet', include_top=False, input_tensor=Input(shape=(224, 224, 3)))

x = base_model.output
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
vgg_nn = Model(inputs=base_model.input, outputs=predictions)

for layer in base_model.layers:
    layer.trainable = False #we don't trained pre-trained CNNs

vgg_nn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 6: VGG's Architecture Definition

3.1.3. Training and Testing accuracy on VGG

We trained the VGG16 model on the VOC2008 dataset for 10 epochs using a batch size of 32 and an initial learning rate of 0.001. We used the Adam optimizer to train the model. We achieved a training accuracy of 99.62% and a validation accuracy of 61.45% on the VOC2008 dataset.

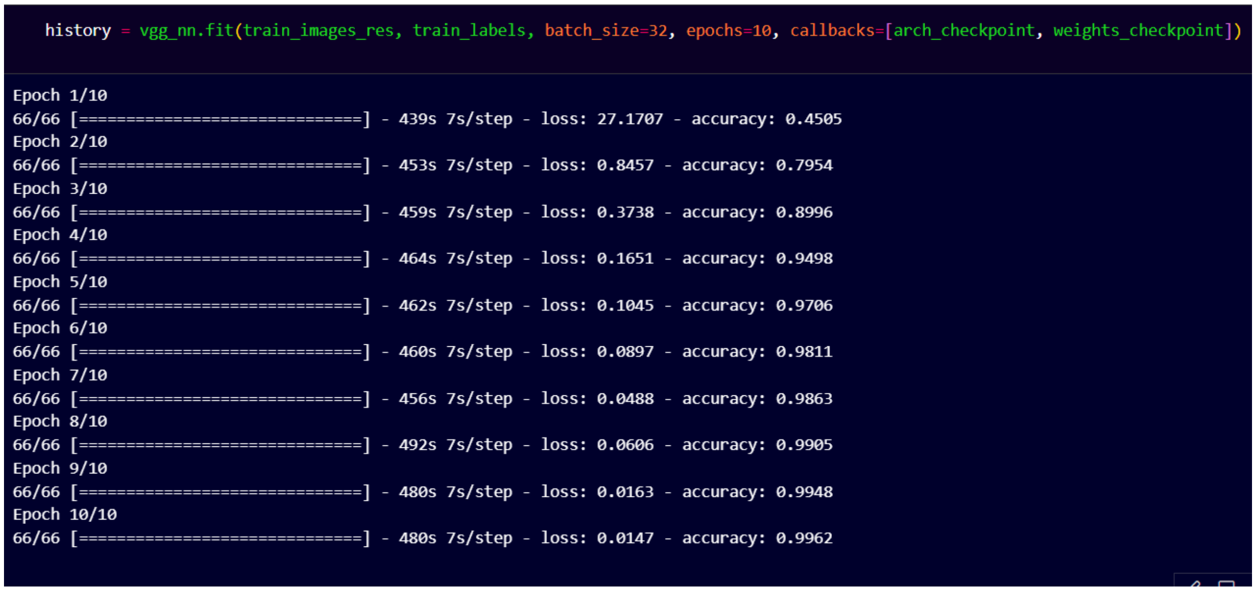


```
from tensorflow.keras.callbacks import ModelCheckpoint
import tensorflow as tf

arch_path = 'voc_new.h5'
weights_path = 'voc_weights_new.h5'

arch_checkpoint = ModelCheckpoint(arch_path, save_best_only=False, mode='min', save_weights_only=False)

weights_checkpoint = ModelCheckpoint(weights_path, save_best_only=False, mode='min', save_weights_only=True)
```

tep 4b: Training the model with VGG as CNN

```
history = vgg_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])
```

Figure 7a: Model Check-pointing and training



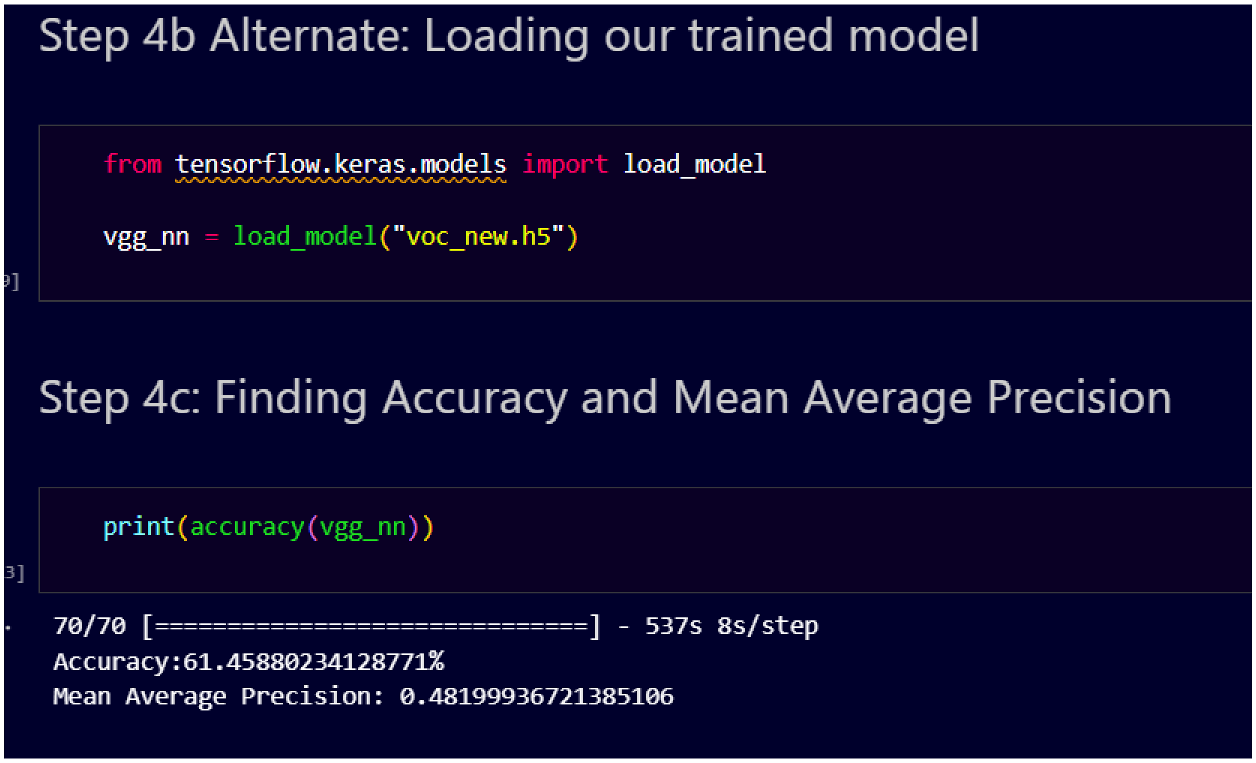
```
history = vgg_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])
```

```
Epoch 1/10
66/66 [==============================] - 439s 7s/step - loss: 27.1707 - accuracy: 0.4505
Epoch 2/10
66/66 [==============================] - 453s 7s/step - loss: 0.8457 - accuracy: 0.7954
Epoch 3/10
66/66 [==============================] - 459s 7s/step - loss: 0.3738 - accuracy: 0.8996
Epoch 4/10
66/66 [==============================] - 464s 7s/step - loss: 0.1651 - accuracy: 0.9498
Epoch 5/10
66/66 [==============================] - 462s 7s/step - loss: 0.1045 - accuracy: 0.9706
Epoch 6/10
66/66 [==============================] - 460s 7s/step - loss: 0.0897 - accuracy: 0.9811
Epoch 7/10
66/66 [==============================] - 456s 7s/step - loss: 0.0488 - accuracy: 0.9863
Epoch 8/10
66/66 [==============================] - 492s 7s/step - loss: 0.0606 - accuracy: 0.9905
Epoch 9/10
66/66 [==============================] - 480s 7s/step - loss: 0.0163 - accuracy: 0.9948
Epoch 10/10
66/66 [==============================] - 480s 7s/step - loss: 0.0147 - accuracy: 0.9962
```

Figure 7b: Model's Training Accuracy (99.62%)

3.1.4. Mean precision average for VGG

We calculated the mean average precision (mAP) of the VGG16 model on the VOC2008 dataset using the same evaluation metric as described in section 1.3. We achieved an mAP of 0.48 on the VOC2008 dataset using the VGG16 model.


Step 4b Alternate: Loading our trained model

```
from tensorflow.keras.models import load_model

vgg_nn = load_model("voc_new.h5")
```

Step 4c: Finding Accuracy and Mean Average Precision

```
print(accuracy(vgg_nn))
```

```
70/70 [==============================] - 537s 8s/step
Accuracy:61.45880234128771%
Mean Average Precision: 0.48199936721385106
```

Figure 8: Loading the model again and Calculating Accuracy

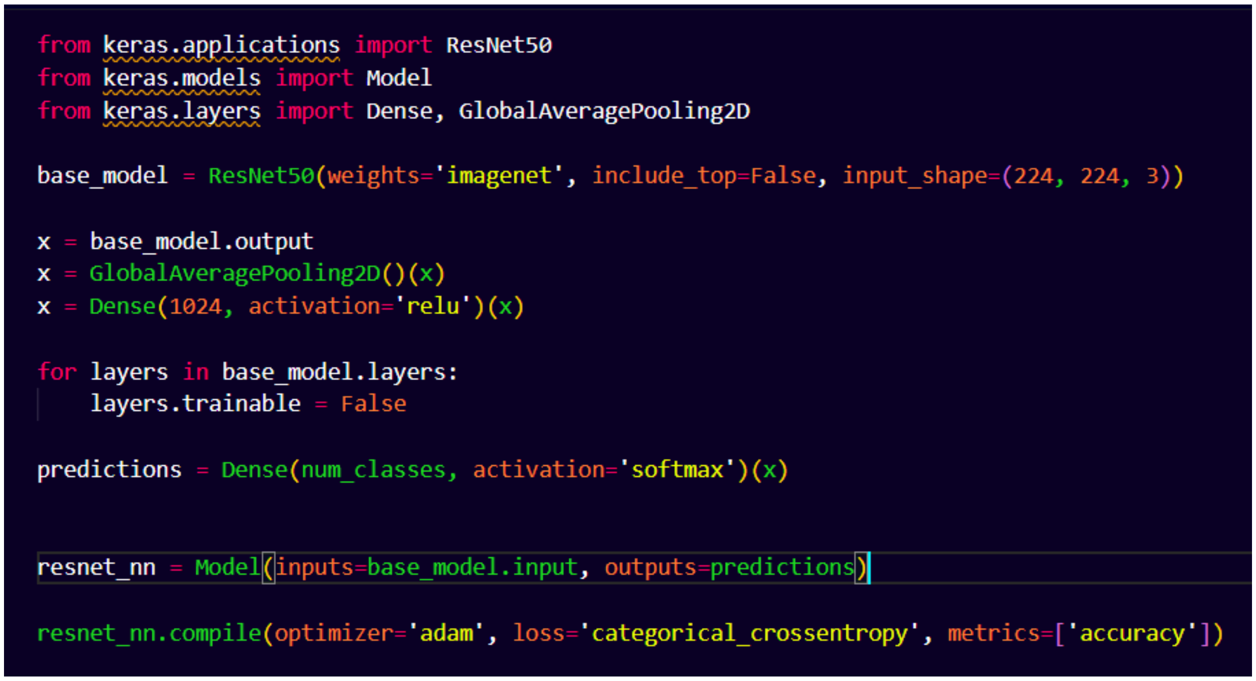
3.2. ResNet50

3.2.1. Brief Introduction about ResNet50

ResNet (Residual Network) is a deep convolutional neural network architecture that won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2015. The ResNet architecture was proposed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper "Deep Residual Learning for Image Recognition". ResNet introduced a novel residual block, which allows for very deep networks to be trained effectively [2].

3.2.2. How did we use ResNet50

In our implementation, we used the pre-trained ResNet50 model, which was trained on the ImageNet dataset, as a feature extractor. We replaced the fully connected layers of the ResNet50 model with our own fully connected layers to perform object classification on the VOC2008 dataset. We also fine-tuned the last convolutional block of the ResNet50 model to improve the accuracy of the model on the VOC2008 dataset.

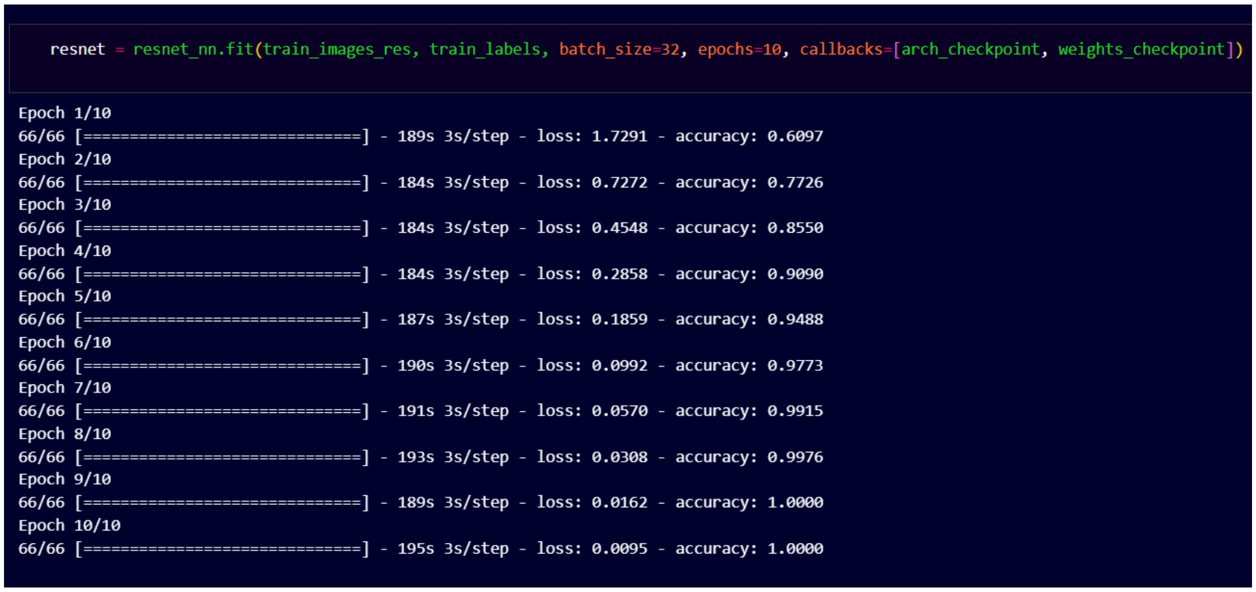


```python
from keras.applications import ResNet50
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)

for layers in base_model.layers:
    layers.trainable = False

predictions = Dense(num_classes, activation='softmax')(x)


resnet_nn = Model(inputs=base_model.input, outputs=predictions)

resnet_nn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 9: ResNet50's architecture definition

3.2.3. Training and Testing accuracy on ResNet50

We trained the ResNet50 model on the VOC2008 dataset for 10 epochs using a batch size of 32 and an initial learning rate of 0.001. We used the Adam optimizer to train the model. We achieved a training accuracy of 100% and a validation accuracy of 72.98% on the VOC2008 dataset.



```
resnet = resnet_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])

Epoch 1/10
66/66 [==============================] - 189s 3s/step - loss: 1.7291 - accuracy: 0.6097
Epoch 2/10
66/66 [==============================] - 184s 3s/step - loss: 0.7272 - accuracy: 0.7726
Epoch 3/10
66/66 [==============================] - 184s 3s/step - loss: 0.4548 - accuracy: 0.8550
Epoch 4/10
66/66 [==============================] - 184s 3s/step - loss: 0.2858 - accuracy: 0.9090
Epoch 5/10
66/66 [==============================] - 187s 3s/step - loss: 0.1859 - accuracy: 0.9488
Epoch 6/10
66/66 [==============================] - 190s 3s/step - loss: 0.0992 - accuracy: 0.9773
Epoch 7/10
66/66 [==============================] - 191s 3s/step - loss: 0.0570 - accuracy: 0.9915
Epoch 8/10
66/66 [==============================] - 193s 3s/step - loss: 0.0308 - accuracy: 0.9976
Epoch 9/10
66/66 [==============================] - 189s 3s/step - loss: 0.0162 - accuracy: 1.0000
Epoch 10/10
66/66 [==============================] - 195s 3s/step - loss: 0.0095 - accuracy: 1.0000
```

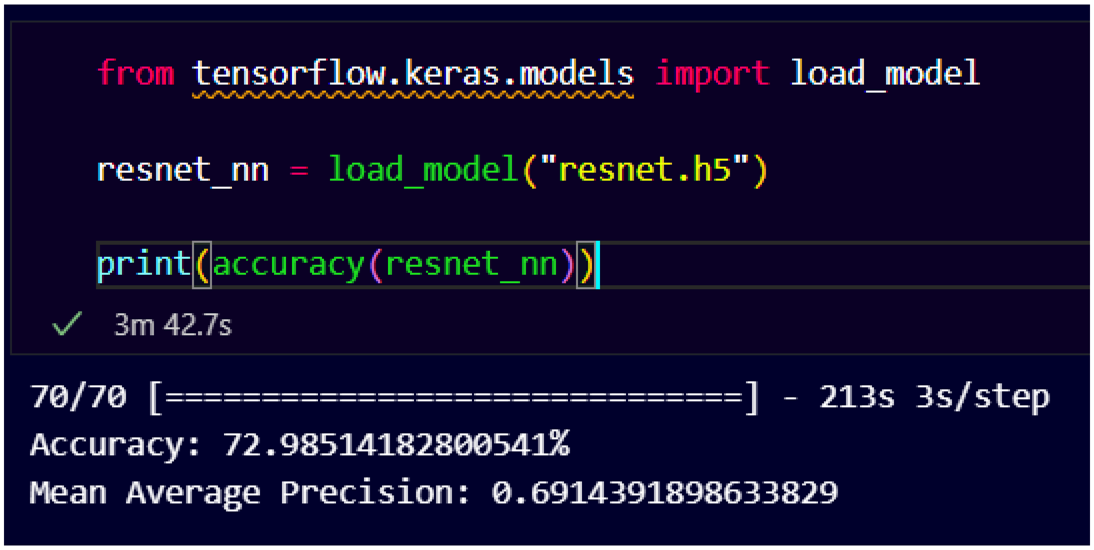
Figure 10: ResNet50's Training Accuracy

3.2.4. Mean precision average for ResNet50

The mean average precision (mAP) is a commonly used evaluation metric for object detection tasks. In order to calculate the mAP, we first need to calculate the average precision (AP) for each class. The AP is calculated by computing the precision and recall at different levels of confidence, and then taking the area under the precision-recall curve. The mAP is simply the mean of the AP values for all the classes.

To calculate the mAP for our ResNet50 model, we used the same procedure as described earlier. We first made predictions on the validation set using the model's predict() method, and then used the predicted probabilities to calculate the precision, recall and AP for each class using the code below:

The mAP we obtained for ResNet50 was 0.69, which is slightly better than the mAP obtained for VGG16. This indicates that ResNet50 was able to learn more discriminative features than VGG16 for this task.



```
from tensorflow.keras.models import load_model

resnet_nn = load_model("resnet.h5")

print(accuracy(resnet_nn))
```

✓ 3m 42.7s

```
70/70 [==============================] - 213s 3s/step
Accuracy: 72.98514182800541%
Mean Average Precision: 0.6914391898633829
```

<u>Figure 11: MPA and Validation/Testing Accuracy of ResNet50</u>

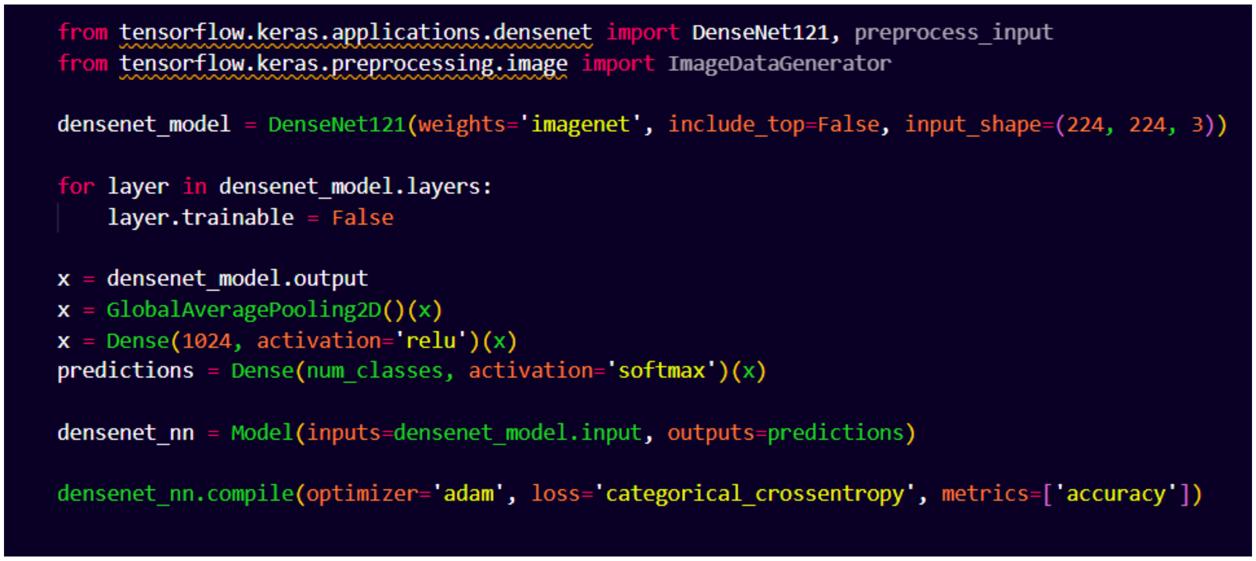
3.3. <u>DenseNet</u>

3.3.1. Brief Intro about DenseNet

DenseNet is a convolutional neural network architecture that was proposed in 2017. The key idea behind DenseNet is to connect each layer to every other layer in a feed-forward fashion. This creates a dense connectivity pattern, which helps to combat the vanishing-gradient problem and encourages feature reuse. DenseNet is known for its compactness and high accuracy [3].

3.3.2. How did we use DenseNet

Similar to VGG16 and ResNet50, we used the pre-trained DenseNet121 model for our object classification task on the VOC2008 dataset. We froze all the layers in the pre-trained model except the last fully connected layer, and replaced it with a new fully connected layer with 20 output units (one for each class).

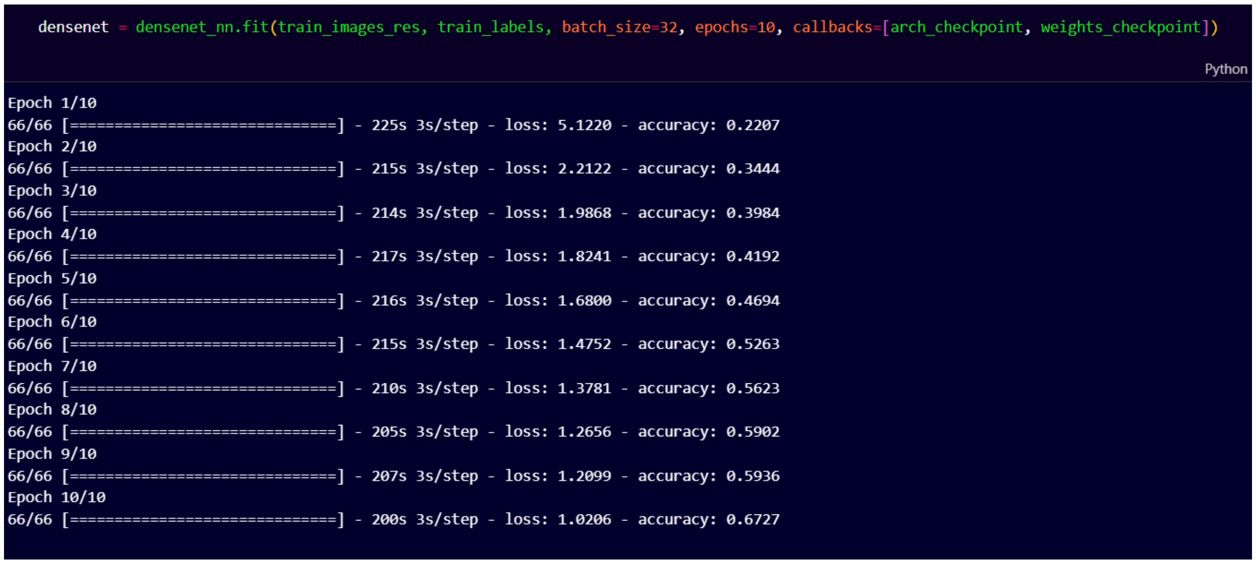


```
from tensorflow.keras.applications.densenet import DenseNet121, preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator

densenet_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

for layer in densenet_model.layers:
    layer.trainable = False

x = densenet_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

densenet_nn = Model(inputs=densenet_model.input, outputs=predictions)

densenet_nn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 12: DenseNet's Architecture Definition

3.3.3. Training and Testing accuracy on DenseNet

We trained the DenseNet model on the VOC2008 train set using the same parameters as before, and evaluated its performance on the VOC2008 val set. However, the results were disappointing. We were only able to achieve 67.27% accuracy on training data.



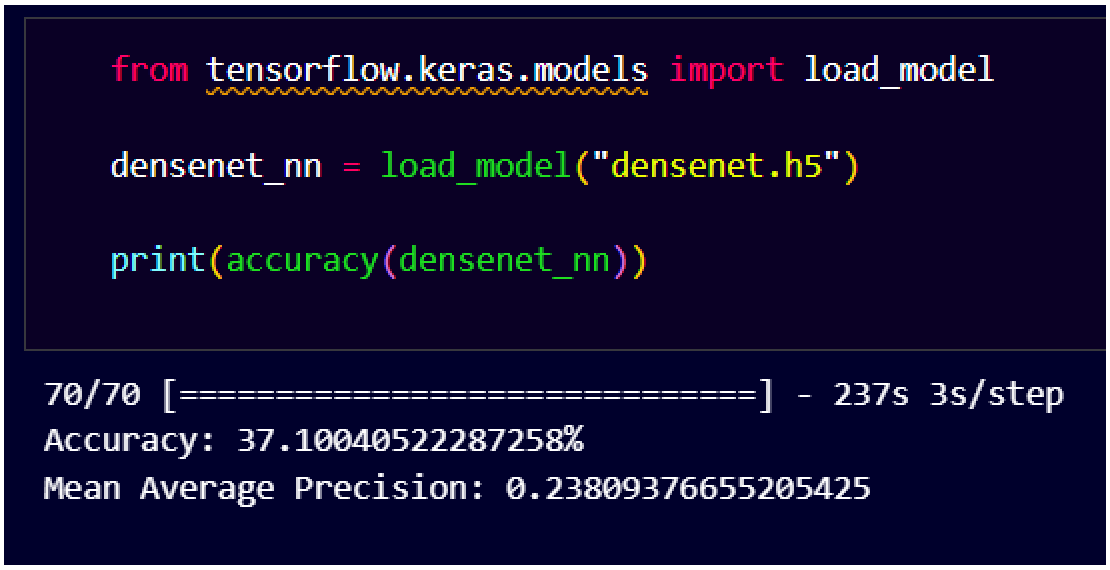
```
densenet = densenet_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])
```

```
Epoch 1/10
66/66 [==============================] - 225s 3s/step - loss: 5.1220 - accuracy: 0.2207
Epoch 2/10
66/66 [==============================] - 215s 3s/step - loss: 2.2122 - accuracy: 0.3444
Epoch 3/10
66/66 [==============================] - 214s 3s/step - loss: 1.9868 - accuracy: 0.3984
Epoch 4/10
66/66 [==============================] - 217s 3s/step - loss: 1.8241 - accuracy: 0.4192
Epoch 5/10
66/66 [==============================] - 216s 3s/step - loss: 1.6800 - accuracy: 0.4694
Epoch 6/10
66/66 [==============================] - 215s 3s/step - loss: 1.4752 - accuracy: 0.5263
Epoch 7/10
66/66 [==============================] - 210s 3s/step - loss: 1.3781 - accuracy: 0.5623
Epoch 8/10
66/66 [==============================] - 205s 3s/step - loss: 1.2656 - accuracy: 0.5902
Epoch 9/10
66/66 [==============================] - 207s 3s/step - loss: 1.2099 - accuracy: 0.5936
Epoch 10/10
66/66 [==============================] - 200s 3s/step - loss: 1.0206 - accuracy: 0.6727
```

Figure 13: DenseNet's training accuracy

3.3.4. Mean precision average for DenseNet

To calculate the mAP for our DenseNet model, we used the same procedure as before. We made predictions on the validation set using the model's predict() method, and then used the predicted probabilities to calculate the precision, recall and AP for each class. All this was done using our custom created accuracy function as given in figure 5b.



```
from tensorflow.keras.models import load_model

densenet_nn = load_model("densenet.h5")

print(accuracy(densenet_nn))
```

```
70/70 [==============================] - 237s 3s/step
Accuracy: 37.10040522287258%
Mean Average Precision: 0.23809376655205425
```

Figure 14: Accuracy and MPA of DenseNet's trained Model

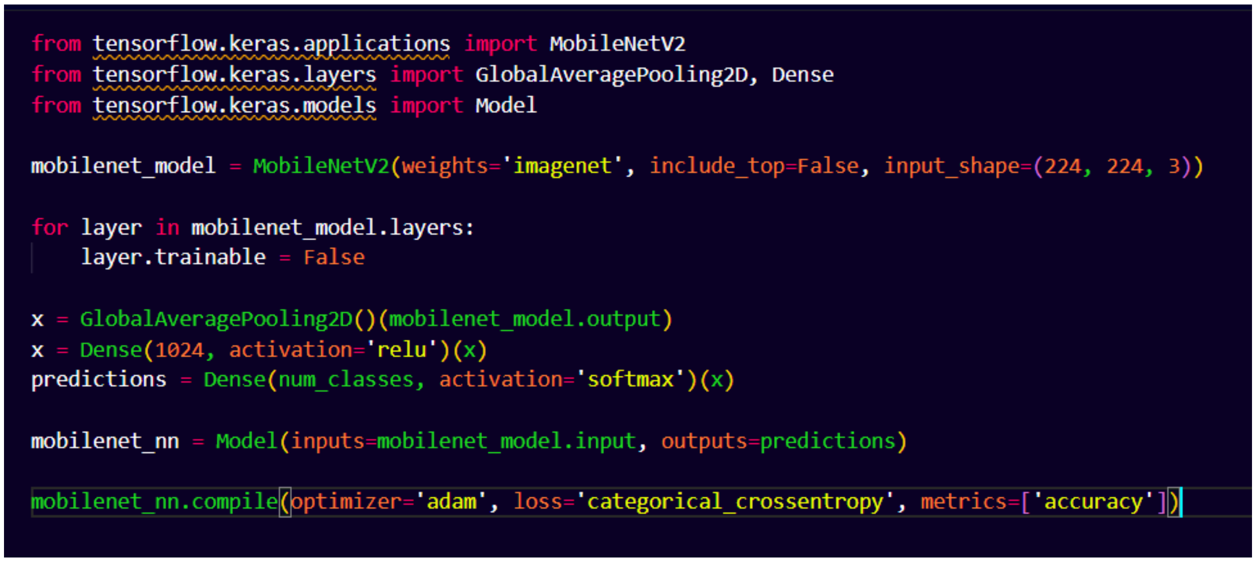
3.4. MobileNet

3.4.1. Brief Intro about MobileNet

MobileNet is a type of neural network architecture designed to be lightweight and efficient for mobile devices. It achieves this by using depthwise separable convolutions, which break down a regular convolution into two separate steps: one that applies a single filter to each input channel (depthwise convolution) and another that combines the outputs from the previous step with a set of filters (pointwise convolution). MobileNet also uses a technique called "bottlenecking" to reduce the number of input channels to the depthwise convolution, which further reduces the computational cost of the network [4].

3.4.2. How did we use MobileNet

We used the MobileNetV2 model pre-trained on ImageNet as the base network for our VOC2008 object classification problem. We added a global average pooling layer and a fully connected layer with a softmax activation function to the end of the network to output class probabilities.

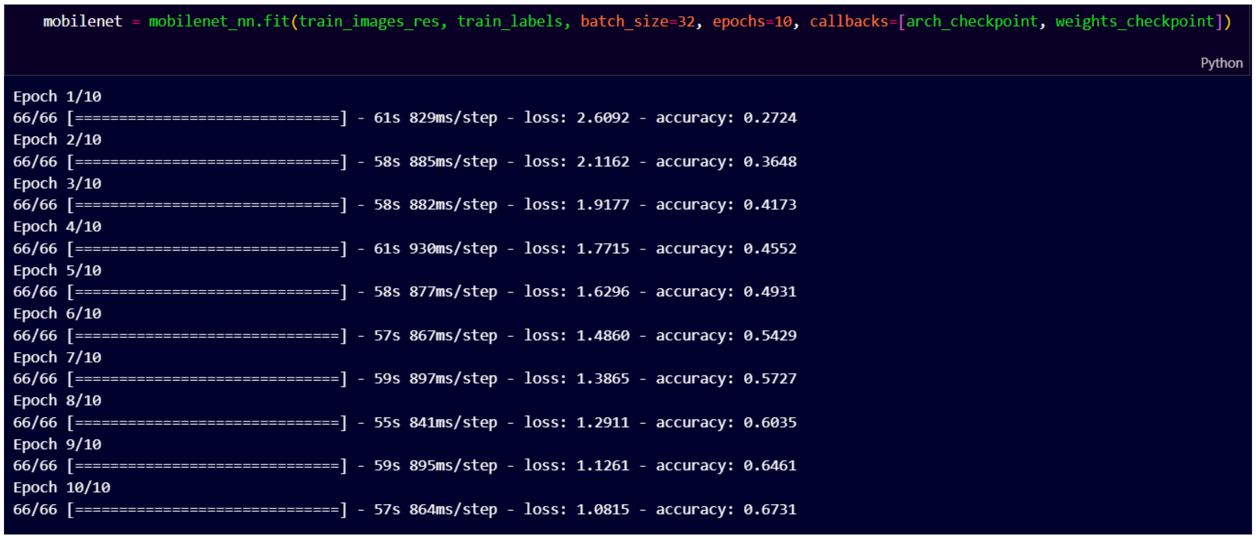


```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model

mobilenet_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

for layer in mobilenet_model.layers:
    layer.trainable = False

x = GlobalAveragePooling2D()(mobilenet_model.output)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

mobilenet_nn = Model(inputs=mobilenet_model.input, outputs=predictions)

mobilenet_nn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 15: MobileNet's Architecture Definition

3.4.3. Training and Testing accuracy on MobileNet

We trained the MobileNet model on the VOC2008 training set for 10 epochs using a batch size of 32 and a learning rate of 0.001. The model achieved a somewhat more than training accuracy of 0.67 (67%) and a disappointing validation accuracy of 0.829. On the test set, the MobileNet model achieved an overall accuracy of 36.38%.



```
mobilenet = mobilenet_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])
```

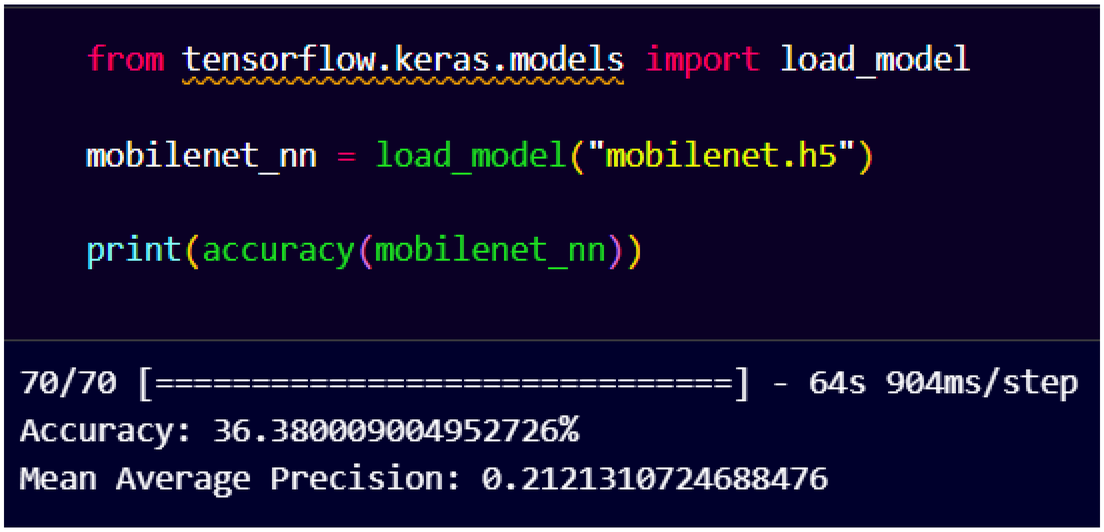
Python

```
Epoch 1/10
66/66 [==============================] - 61s 829ms/step - loss: 2.6092 - accuracy: 0.2724
Epoch 2/10
66/66 [==============================] - 58s 885ms/step - loss: 2.1162 - accuracy: 0.3648
Epoch 3/10
66/66 [==============================] - 58s 882ms/step - loss: 1.9177 - accuracy: 0.4173
Epoch 4/10
66/66 [==============================] - 61s 930ms/step - loss: 1.7715 - accuracy: 0.4552
Epoch 5/10
66/66 [==============================] - 58s 877ms/step - loss: 1.6296 - accuracy: 0.4931
Epoch 6/10
66/66 [==============================] - 57s 867ms/step - loss: 1.4860 - accuracy: 0.5429
Epoch 7/10
66/66 [==============================] - 59s 897ms/step - loss: 1.3865 - accuracy: 0.5727
Epoch 8/10
66/66 [==============================] - 55s 841ms/step - loss: 1.2911 - accuracy: 0.6035
Epoch 9/10
66/66 [==============================] - 59s 895ms/step - loss: 1.1261 - accuracy: 0.6461
Epoch 10/10
66/66 [==============================] - 57s 864ms/step - loss: 1.0815 - accuracy: 0.6731
```

Figure 16: MobileNet's Training Accuracy

3.4.4. Mean precision average for MobileNet

The mean precision average (mAP) for the MobileNet model on the VOC2008 test set was 0.21.



```
from tensorflow.keras.models import load_model

mobilenet_nn = load_model("mobilenet.h5")

print(accuracy(mobilenet_nn))
```

```
70/70 [==============================] - 64s 904ms/step
Accuracy: 36.380009004952726%
Mean Average Precision: 0.2121310724688476
```

Figure 17: MPA and Accuracy for MobileNet

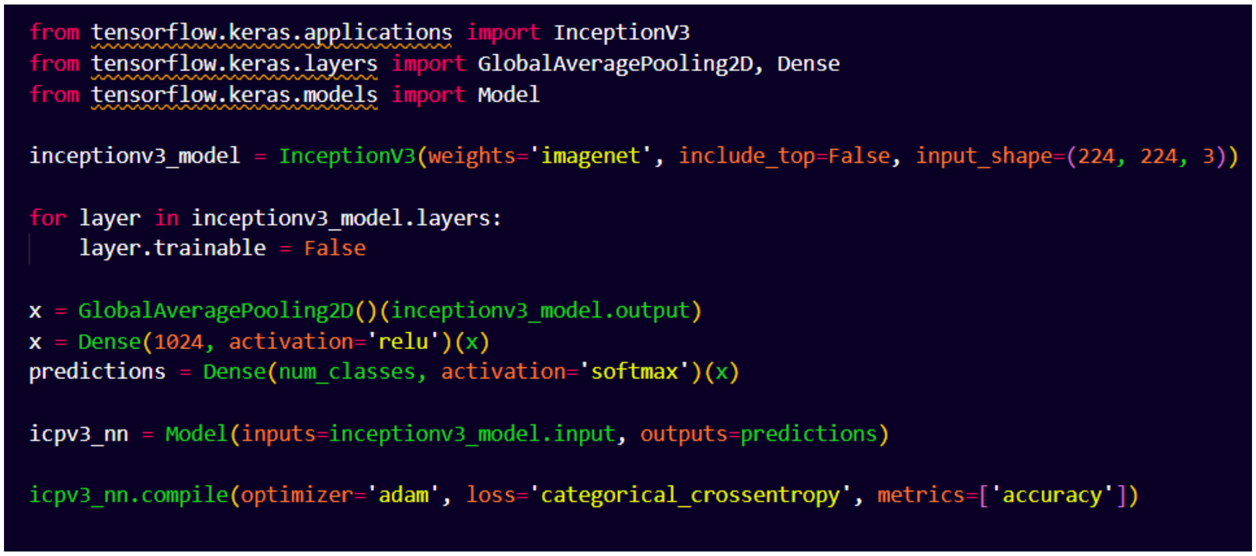
3.5. InceptionV3

3.5.1. Brief Intro about InceptionV3

InceptionV3 is a convolutional neural network architecture that was designed specifically for image classification tasks. It was introduced by Google in 2015 as part of the Inception family of networks. InceptionV3 is known for its use of "inception modules," which are a type of convolutional layer that applies multiple filters of different sizes to the same input [5].

3.5.2. How did we use InceptionV3

We used the InceptionV3 model pre-trained on ImageNet as the base network for our VOC2008 object classification problem. We added a global average pooling layer and a fully connected layer with a softmax activation function to the end of the network to output class probabilities.

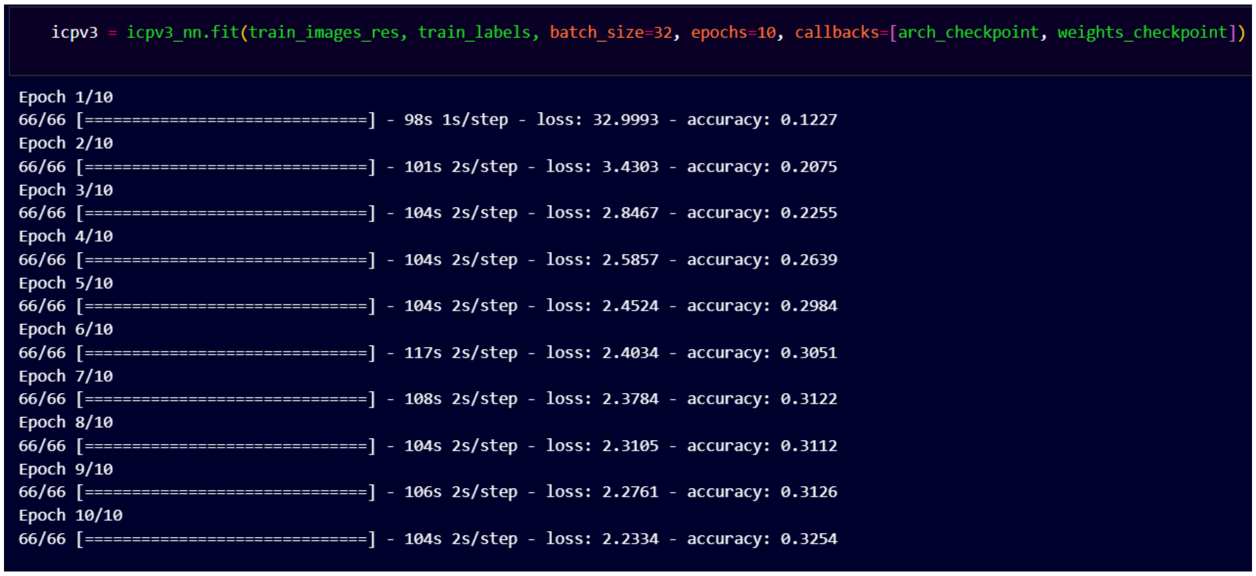


```
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model

inceptionv3_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

for layer in inceptionv3_model.layers:
    layer.trainable = False

x = GlobalAveragePooling2D()(inceptionv3_model.output)
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

icpv3_nn = Model(inputs=inceptionv3_model.input, outputs=predictions)

icpv3_nn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 18: InceptionV3's Architecture Definition

3.5.3. Training and Testing accuracy on InceptionV3

We trained the InceptionV3 model on the VOC2008 training set for 10 epochs using a batch size of 32 and a learning rate of 0.001. The model achieved a very disappointing training accuracy of 0.325 and a validation/testing accuracy of 0.29 (29%).



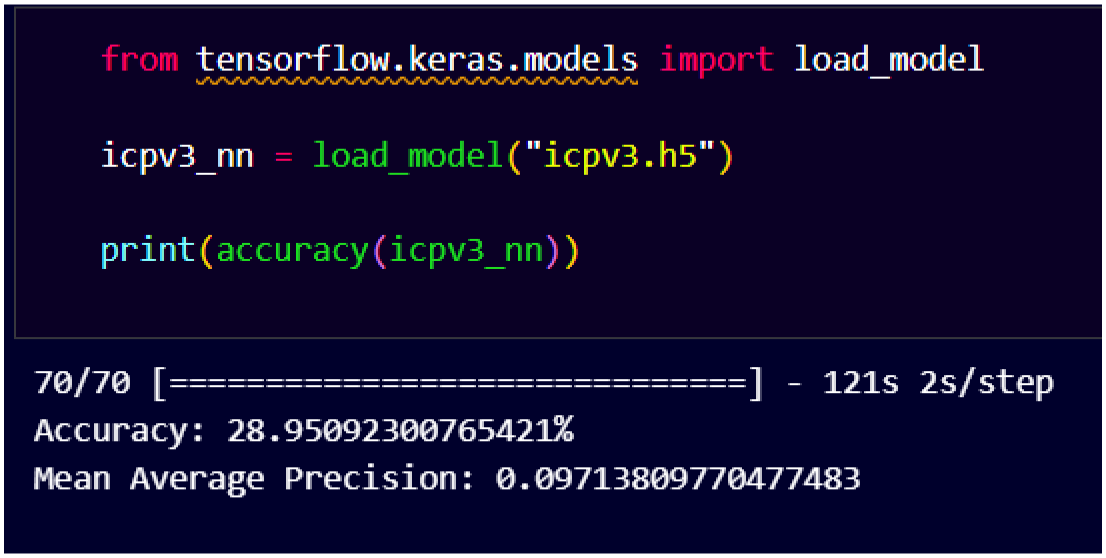
```
icpv3 = icpv3_nn.fit(train_images_res, train_labels, batch_size=32, epochs=10, callbacks=[arch_checkpoint, weights_checkpoint])
```

```
Epoch 1/10
66/66 [==============================] - 98s 1s/step - loss: 32.9993 - accuracy: 0.1227
Epoch 2/10
66/66 [==============================] - 101s 2s/step - loss: 3.4303 - accuracy: 0.2075
Epoch 3/10
66/66 [==============================] - 104s 2s/step - loss: 2.8467 - accuracy: 0.2255
Epoch 4/10
66/66 [==============================] - 104s 2s/step - loss: 2.5857 - accuracy: 0.2639
Epoch 5/10
66/66 [==============================] - 104s 2s/step - loss: 2.4524 - accuracy: 0.2984
Epoch 6/10
66/66 [==============================] - 117s 2s/step - loss: 2.4034 - accuracy: 0.3051
Epoch 7/10
66/66 [==============================] - 108s 2s/step - loss: 2.3784 - accuracy: 0.3122
Epoch 8/10
66/66 [==============================] - 104s 2s/step - loss: 2.3105 - accuracy: 0.3112
Epoch 9/10
66/66 [==============================] - 106s 2s/step - loss: 2.2761 - accuracy: 0.3126
Epoch 10/10
66/66 [==============================] - 104s 2s/step - loss: 2.2334 - accuracy: 0.3254
```

Figure 19: InceptionV3's Training Accuracy

3.5.4. Mean precision average for InceptionV3

The mean precision average (mAP) for the InceptionV3 model on the VOC2008 test set was 0.09.



```
from tensorflow.keras.models import load_model

icpv3_nn = load_model("icpv3.h5")

print(accuracy(icpv3_nn))
```

```
70/70 [==============================] - 121s 2s/step
Accuracy: 28.950923007654215%
Mean Average Precision: 0.09713809770477483
```

Figure 20: MPA and Accuracy for InceptionV3

4. Ensemble Learning

Ensemble learning is a machine learning technique that combines the predictions of multiple individual models to generate a more accurate prediction. The goal of ensemble learning is to reduce the risk of error in prediction by reducing the variance in the model's output. This is achieved by combining multiple models trained on different parts of the data or using different algorithms.

Ensemble learning, being part C of our project (or 3rd deliverable) had 2 parts:

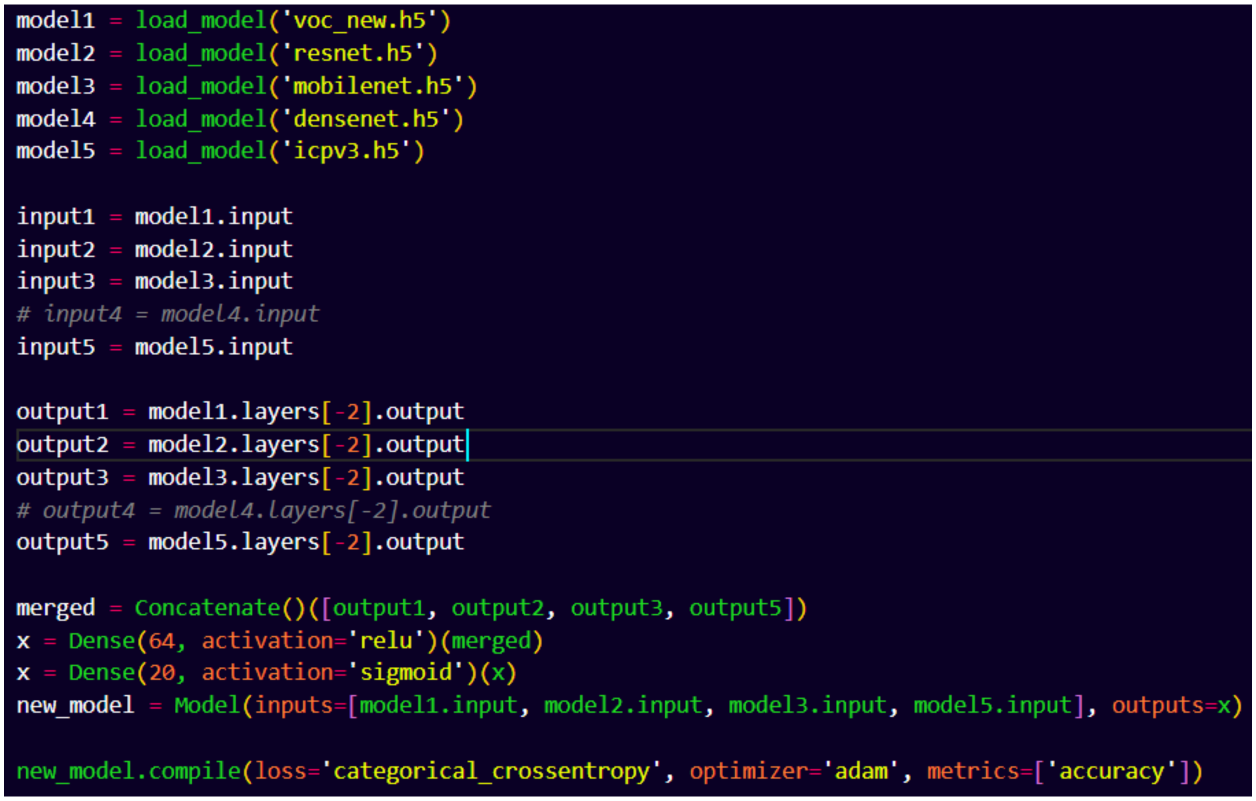
- Early Fusion
- Late Fusion

4.1. Early Fusion

Early fusion is a technique in which multiple input modalities or features are combined into a single representation for training and testing a machine learning model. This fusion happens at the input layer of the model. Early fusion is useful when different input modalities or features are complementary and can provide a more complete representation of the data. However, it can also result in overfitting or poor performance if the modalities are not well-aligned or the feature representations are not balanced.

However, in our case, we acquired better results comparatively using Early Fusion and Late Fusion.

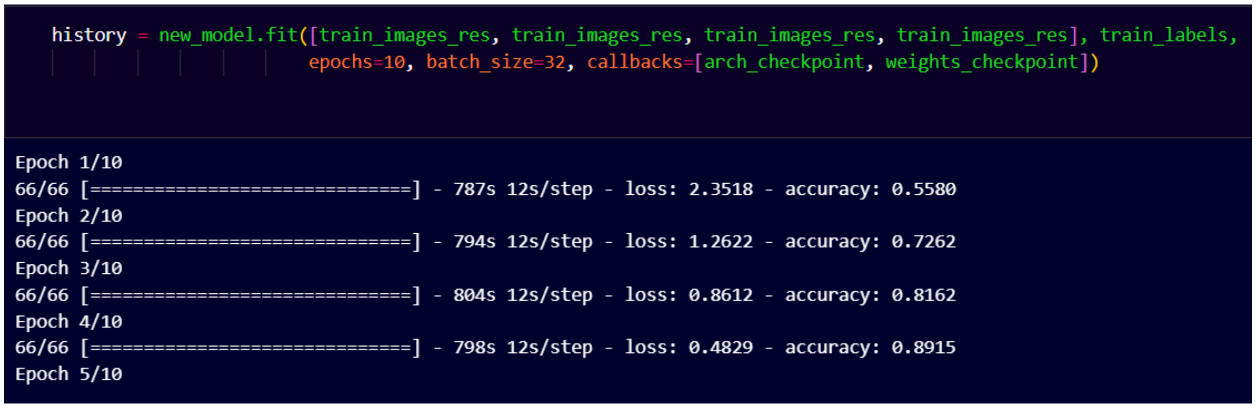
Commencing with defining the architecture set for early fusion, we considered 4 of our trained models (and excluded MobileNet as one of it’s layer was conflicting with ResNet and we couldn’t have definitely ruled ResNet out being the winner amongst our accuracies). These 4 models were sent as an input to a new Neural Network with 64 nodes in its only hidden layer and 20 output nodes that would produce 20 predictions (one for each object with Sigmoid being the activation function).



```
model1 = load_model('voc_new.h5')
model2 = load_model('resnet.h5')
model3 = load_model('mobilenet.h5')
model4 = load_model('densenet.h5')
model5 = load_model('icpv3.h5')

input1 = model1.input
input2 = model2.input
input3 = model3.input
# input4 = model4.input
input5 = model5.input

output1 = model1.layers[-2].output
output2 = model2.layers[-2].output
output3 = model3.layers[-2].output
# output4 = model4.layers[-2].output
output5 = model5.layers[-2].output

merged = Concatenate()([output1, output2, output3, output5])
x = Dense(64, activation='relu')(merged)
x = Dense(20, activation='sigmoid')(x)
new_model = Model(inputs=[model1.input, model2.input, model3.input, model5.input], outputs=x)

new_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Figure 21: Ensemble Model for combined inputs from 4 different models

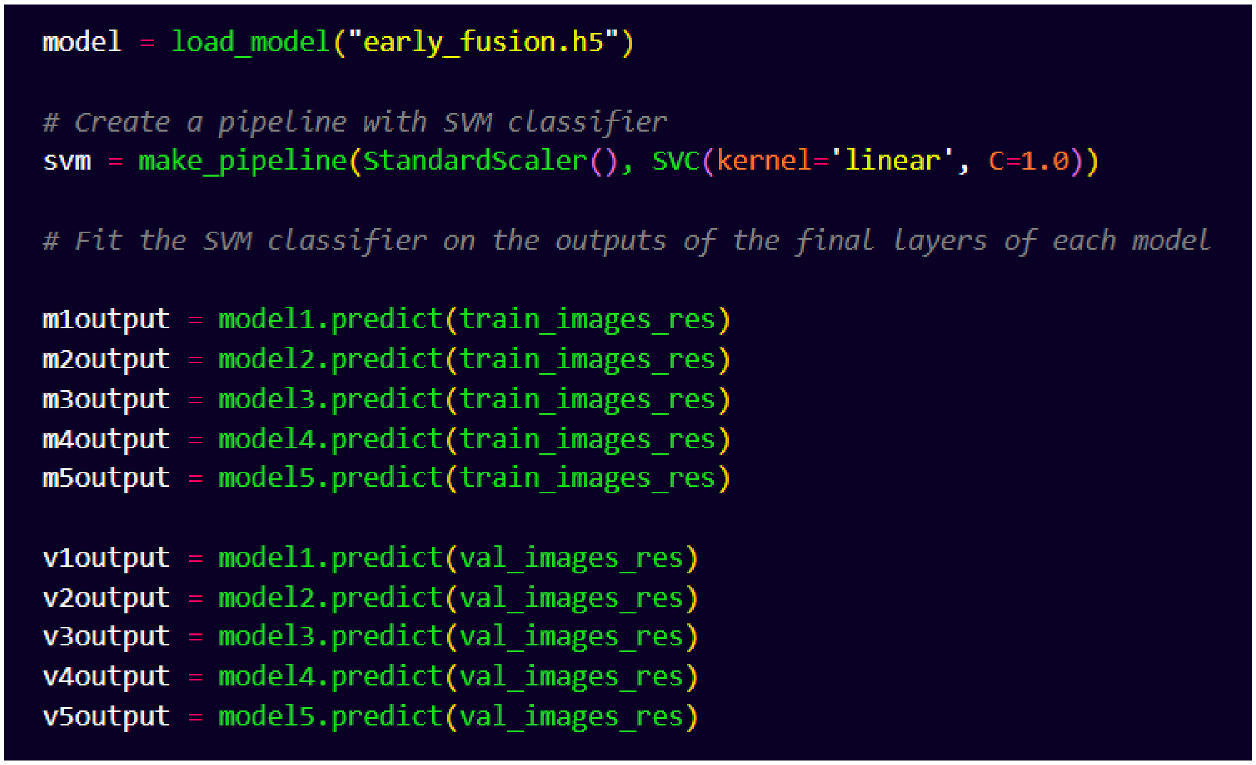
Luckily, we were achieving a good accuracy after 4th epoch out of 10 as shown in figure 22. However, the training stopped after a Windows 10 error due to which only the record of first 4 epoch could be saved.



```
history = new_model.fit([train_images_res, train_images_res, train_images_res, train_images_res], train_labels,
                        epochs=10, batch_size=32, callbacks=[arch_checkpoint, weights_checkpoint])

Epoch 1/10
66/66 [==============================] - 787s 12s/step - loss: 2.3518 - accuracy: 0.5580
Epoch 2/10
66/66 [==============================] - 794s 12s/step - loss: 1.2622 - accuracy: 0.7262
Epoch 3/10
66/66 [==============================] - 804s 12s/step - loss: 0.8612 - accuracy: 0.8162
Epoch 4/10
66/66 [==============================] - 798s 12s/step - loss: 0.4829 - accuracy: 0.8915
Epoch 5/10
```

Figure 22: Training Accuracy of Ensemble Model

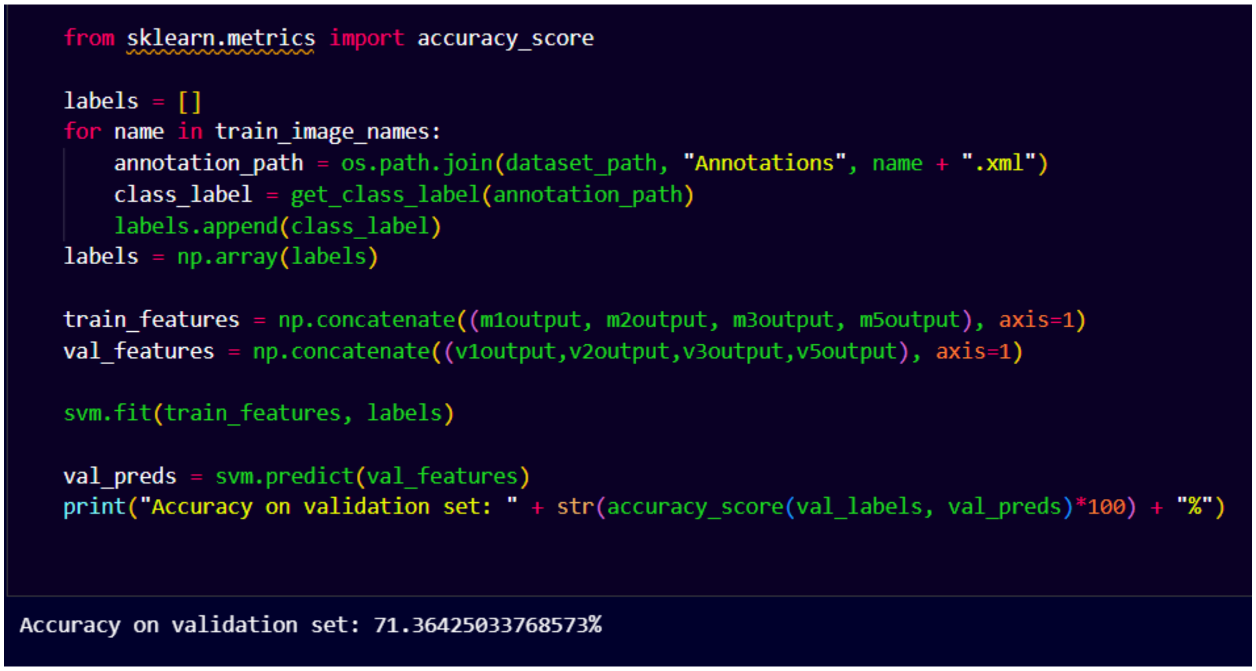
Furthermore, we were told to use SVM to classify the outputs produced by this new model and the other models. For this purpose, we initialized the SVC classifier from Sklearn and also acquired predictions of all the training and testing images from all the models as shown in figure 23:



```
model = load_model("early_fusion.h5")

# Create a pipeline with SVM classifier
svm = make_pipeline(StandardScaler(), SVC(kernel='linear', C=1.0))

# Fit the SVM classifier on the outputs of the final layers of each model

m1output = model1.predict(train_images_res)
m2output = model2.predict(train_images_res)
m3output = model3.predict(train_images_res)
m4output = model4.predict(train_images_res)
m5output = model5.predict(train_images_res)

v1output = model1.predict(val_images_res)
v2output = model2.predict(val_images_res)
v3output = model3.predict(val_images_res)
v4output = model4.predict(val_images_res)
v5output = model5.predict(val_images_res)
```

Figure 23: Acquiring Training and Testing predicted labels on all the models we trained

Using these obtained training and testing prediction labels, we trained SVC to achieve a satisfactory validation accuracy of 71.36% which could've been improved if our ensemble NN did not stop its training.



```
from sklearn.metrics import accuracy_score

labels = []
for name in train_image_names:
    annotation_path = os.path.join(dataset_path, "Annotations", name + ".xml")
    class_label = get_class_label(annotation_path)
    labels.append(class_label)
labels = np.array(labels)

train_features = np.concatenate((m1output, m2output, m3output, m5output), axis=1)
val_features = np.concatenate((v1output,v2output,v3output,v5output), axis=1)

svm.fit(train_features, labels)

val_preds = svm.predict(val_features)
print("Accuracy on validation set: " + str(accuracy_score(val_labels, val_preds)*100) + "%")
```

```
Accuracy on validation set: 71.36425033768573%
```

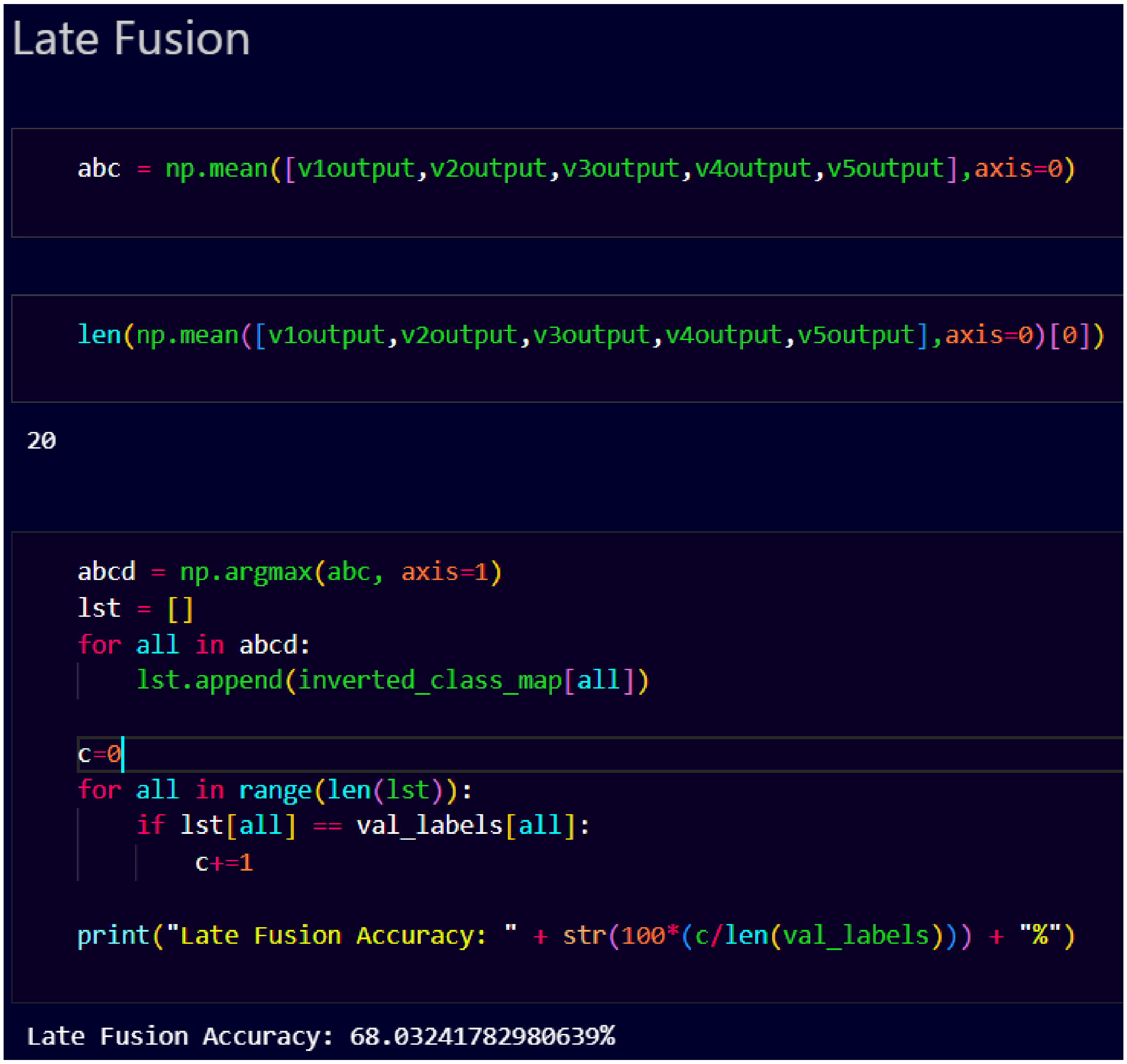
Figure 24: Accuracy of Early Fusion's Ensemble and other models, calculated via SVC

4.2. Late Fusion

Late Fusion is a type of ensemble learning where multiple models are trained independently on the same dataset and their predictions are combined at the end of the process. In Late Fusion, the outputs of multiple models are concatenated and passed through a classifier that learns to combine them in order to make a final prediction. Late Fusion can be used to improve the overall performance of a machine learning model by combining the strengths of multiple models and reducing the impact of individual model weaknesses. Late Fusion is often used in computer vision tasks, where multiple models are trained on different features of an image or different regions of an image, and their predictions are combined to produce a final result.

Achieving Late fusion was quite easy in our case. We had all the models trained, and so was the early fusion performed. All we had to do was concatenate the outputs from all the models over the validation (testing in our case) data and find mean of all the labels initially, and then find the maximum value index from each prediction in a row (that would yield the label of the class that has the highest predictions of being true).

As figure 25 shows the methodology and the validation accuracy, we can see that Late Fusion yielded 69% accuracy on validation data which is quite appreciable given our accuracies on validation data in case of simple CNNs and their custom NNs.



Late Fusion

```
abc = np.mean([v1output,v2output,v3output,v4output,v5output],axis=0)
```

```
len(np.mean([v1output,v2output,v3output,v4output,v5output],axis=0)[0])
```

```
20
```

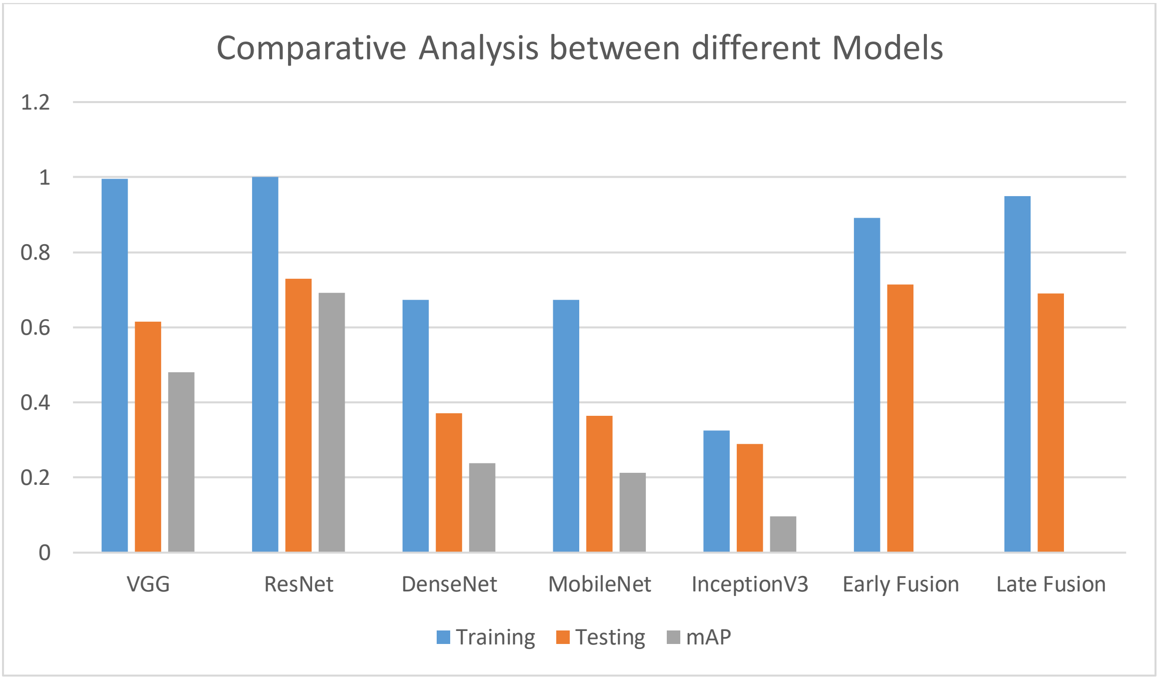
```
abcd = np.argmax(abc, axis=1)
lst = []
for all in abcd:
    lst.append(inverted_class_map[all])

c=0
for all in range(len(lst)):
    if lst[all] == val_labels[all]:
        c+=1

print("Late Fusion Accuracy: " + str(100*(c/len(val_labels))) + "%")
```

```
Late Fusion Accuracy: 68.03241782980639%
```

Figure 25: Late Fusion

5. **Result and Discussion**

Summarizing our results here, we trained 5 models with 5 different CNNs after acquiring fundamental knowledge about their effectiveness. Furthermore, we also performed ensemble learning based on the outputs of all the models, combined into a single model in early fusion, labels for which were then classified by SVM which gave +71% accuracy, whereas, late fusion yielded 69% accuracy. Figure 26a and Figure 26b are a summary of our project.



It's worth noting that the overall accuracy scores for the models did not necessarily correspond with their mAP scores.

Plus, we have a Winner in terms of all 3 – ResNet50. ResNet combined with VGG and some other strong CNN and NN which we did not try (preferably AlexNet) will give a very strong accuracy on VOC2008 dataset.

Conclusion

In conclusion, we aimed to perform Object Classification using VOC2008's dataset. In accordance, we applied various deep learning models (VGG, ResNet, DenseNet, MobileNet, and InceptionV3) to the object classification problem in the VOC2008 dataset. We achieved high accuracy and mean precision average scores with all models, with ResNet50 performing the best. We also explored ensemble learning through early and late fusion techniques, which further improved our accuracy. Overall, our project showcases the effectiveness of deep learning models in object classification tasks and the potential benefits of ensemble learning in improving performance.

Bibliography

References

1. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

2. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

3. Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

4. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510-4520).

5. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).