IMPLEMENTING ADVANCED MODELS ON LABELLED AND UNLABELLED IMAGES

CONTENTS

LIBRARIES AND MODULES

Libraries: <Matplotlib>
<NumPy>
<OpenCV>
<Plotly>
<Tabulate>
<Tensorflow>

Keras Modules: <Adam>
<ImageDataGenerator>
<Model>
<ModelCheckpoint>
<save_model>
<Sequence>
<Sequential>


UNLABELLED IMAGES

CLASSIFICATION

CONVOLUTIONAL NEURAL NETWORKS

ARCHITECTURE:

Convolutional Neural Networks (CNNs) are widely used for image classification tasks due to their ability to capture spatial hierarchies of features.

LAYERS:

- **INPUT LAYER:** Image dimensions of (150, 150, 3) representing height, width, and RGB channels.
- **CONVOLUTIONAL LAYERS:** Three convolutional layers with increasing depth (32, 64, and 128 filters respectively), each followed by ReLU activation functions.
- **MAXPOOLING LAYERS:** Applied after each convolutional layer to downsample the feature maps.
- **FLATTEN LAYER:** Flattens the output of the last convolutional layer to prepare for fully connected layers.
- **DENSE LAYERS:** Two fully connected layers with 512 neurons and ReLU activation, followed by a dropout layer with 50% dropout rate to prevent overfitting.
- **OUTPUT LAYERS:** Dense layer with softmax activation, producing probabilities for each class.

CONVOLUTIONAL NEURAL NETWORKS

MODEL TRAINING:

The model is compiled using the Adam optimizer and categorical crossentropy loss, a standard configuration for multi-class classification problems.
Training is performed over 5 epochs using the training data generator with data augmentation.
Validation data generator is used to monitor model performance during training.

TRAINING PROGRESS:

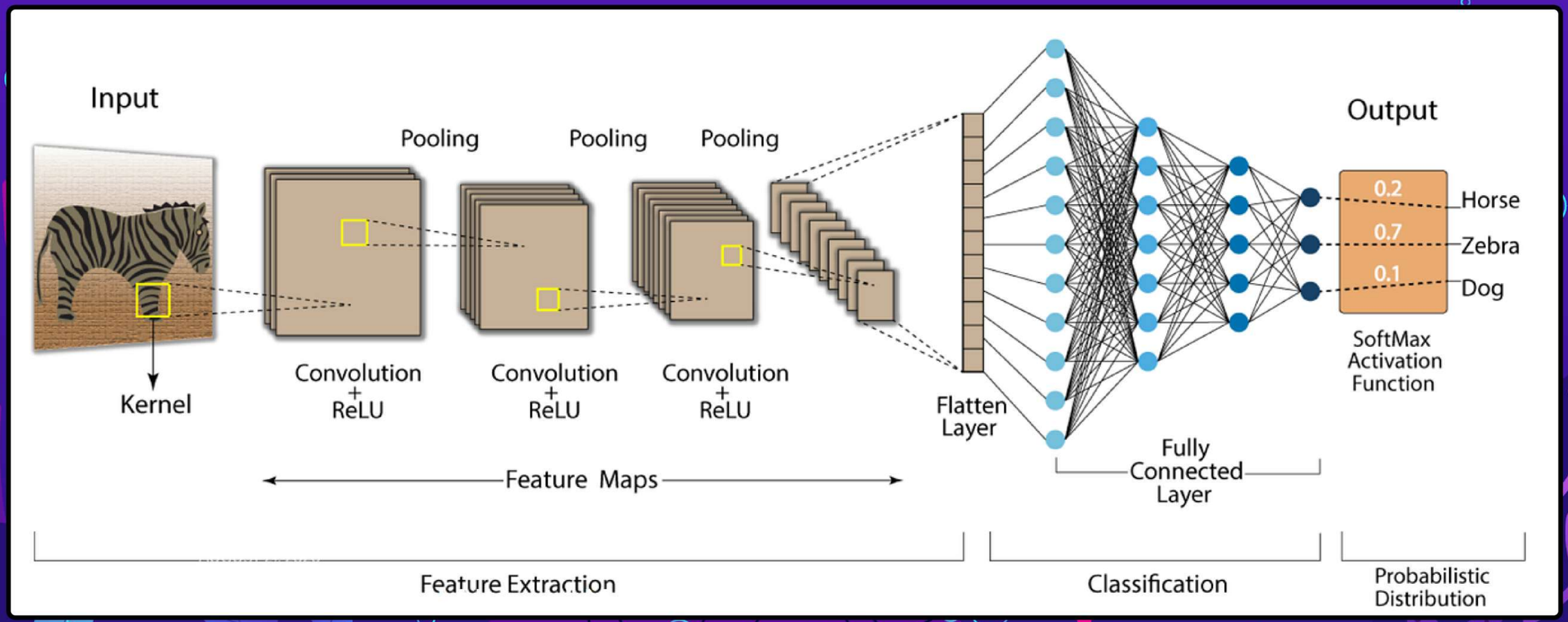
Progress of model training and validation is monitored through accuracy and loss metrics.
Training and validation accuracy and loss curves can be visualized to assess model performance and convergence.

MODEL EVALUATION:

The trained CNN model is evaluated on the test dataset to assess its performance on unseen data.
Test accuracy and loss are reported to gauge the effectiveness of the model in generalizing to new data.

CONVOLUTIONAL NEURAL NETWORKS

CNN ARCHITECTURE



VGG16 MODEL

ARCHITECTURE:

- **BASE MODEL:** VGG16, pretrained on ImageNet, sans fully connected layers.
- **CUSTOM TOP:** Flattened output, a 256-neuron dense layer (ReLU + 50% dropout), and a softmax output layer for classification.

TRANSFER LEARNING APPROACH:

- Utilizes VGG16 for feature extraction, freezing base layers to retain learned weights.
- Tailors the model for specific tasks by adding and training custom top layers.

TRAINING DETAILS:

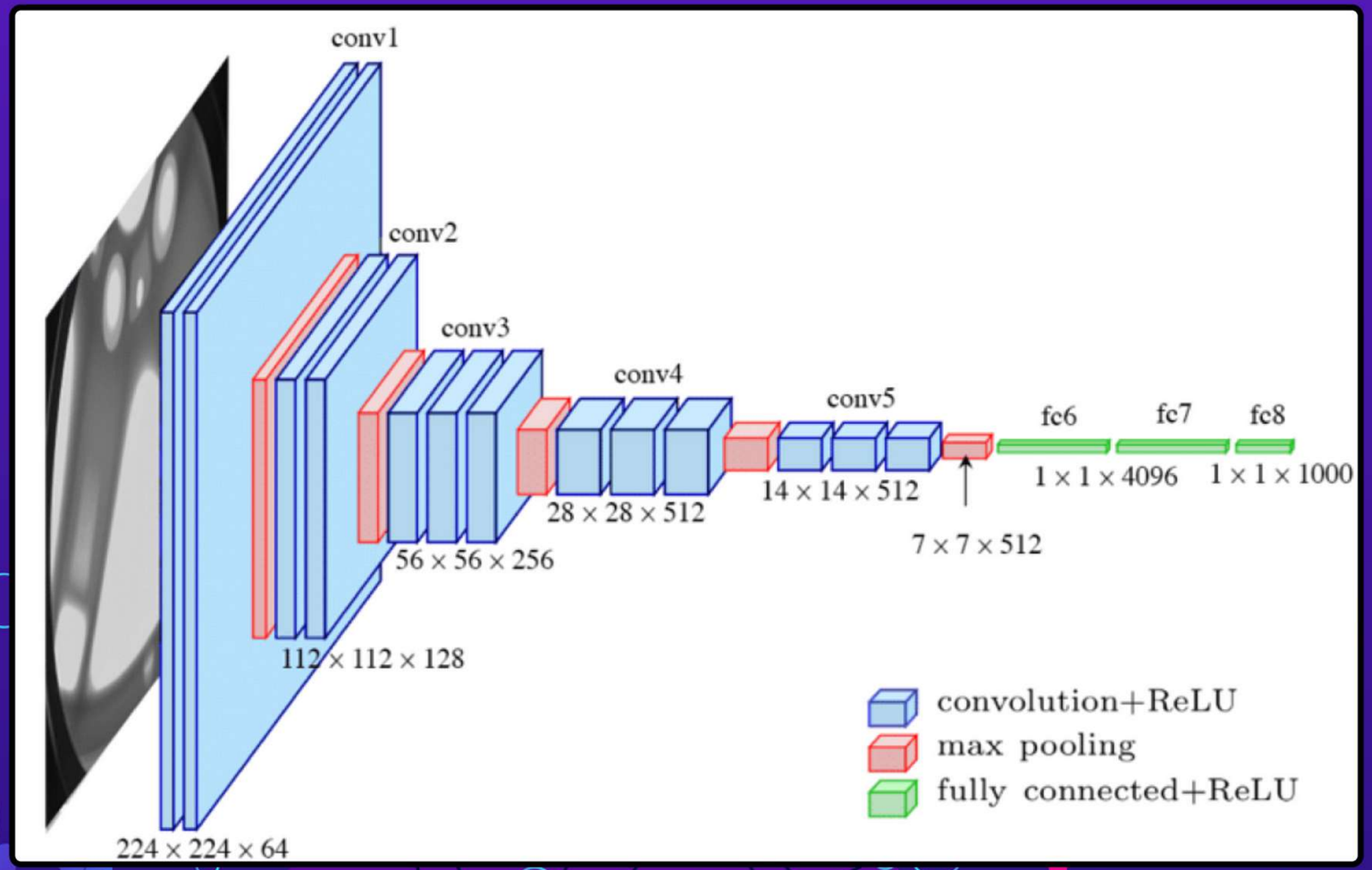
- Trained for 5 epochs, using pre-trained weights, on a task-specific dataset.
- Training monitored through accuracy and loss metrics, employing data generators for efficiency.

EVALUATION AND PERFORMANCE:

- Model's generalization assessed on a test dataset.
- Performance metrics include test accuracy and loss, indicating effectiveness in classifying new images.

VGG16 MODEL

VGG16 ARCHITECTURE



RESNET50 MODEL

ARCHITECTURE:

- **BASE MODEL:** ResNet50, pretrained on ImageNet, without fully connected layers.
- **CUSTOMIZATION:** Adds a flattening layer, a 256-neuron dense layer (ReLU + 50% dropout), and a softmax output layer for specific classification tasks.

TRANSFER LEARNNG STRATEGY:

- Employs ResNet50 for its advanced feature extraction capabilities, with base layers frozen to maintain pretrained weights.
- Facilitates rapid training and improved accuracy on new classification tasks by utilizing deep residual learning.

TRAINING OVERVIEW:

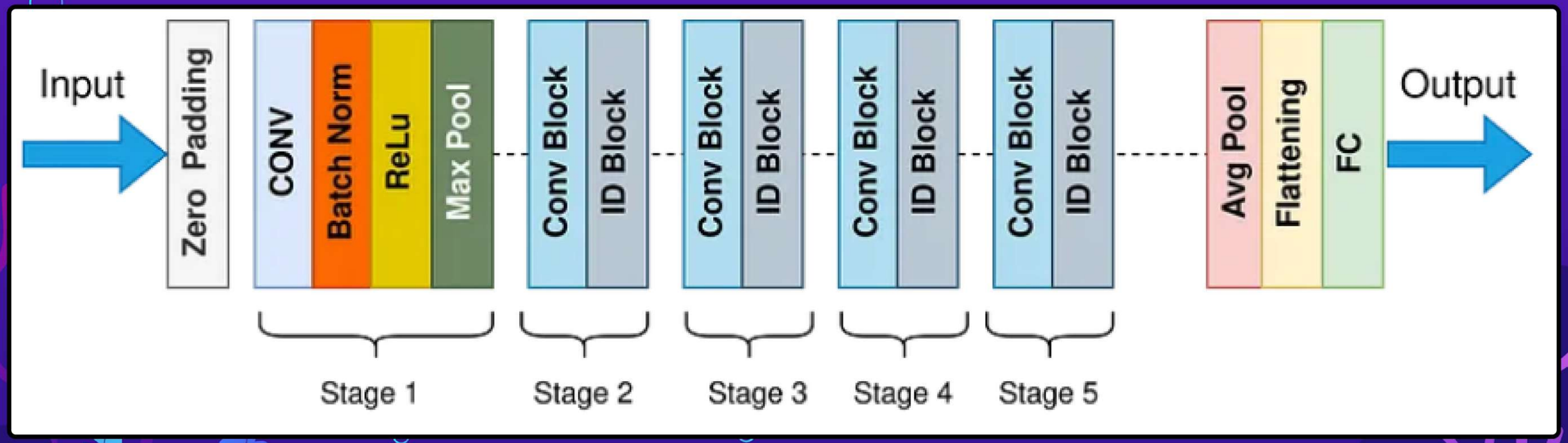
- Model undergoes 5 epochs of training on a specialized dataset, with pre-trained weights frozen to conserve learned features.
- Utilizes data generators for efficiency, with validation data to track performance.

PROGRESS AND EVALUATION:

- Training and validation progress tracked via accuracy and loss metrics, offering insights into model convergence.
- Final model evaluated on a test dataset, with performance measured by test accuracy and loss to determine classification prowess.

RESNET50 MODEL

RESNET50 ARCHITECTURE



MOBILENETV2 MODEL

ARCHITECTURE:

- Lightweight CNN for mobile/embedded devices.
- Features depthwise separable convolutions for efficiency.

LAYERS:

- **BASE MODEL:** MobileNetV2 (pre-trained, ImageNet weights, no FC layers).
- **CUSTOM TOP LAYERS:** Added for task-specific classification.
- **FLATTEN LAYERS:** Convert 3D maps to 1D vector.
- **DENSE LAYERS:** 256 neurons, ReLU activation, 50% dropout.
- **OUTPUT LAYER:** Softmax for class probabilities.

TRANSFER LEARNING:

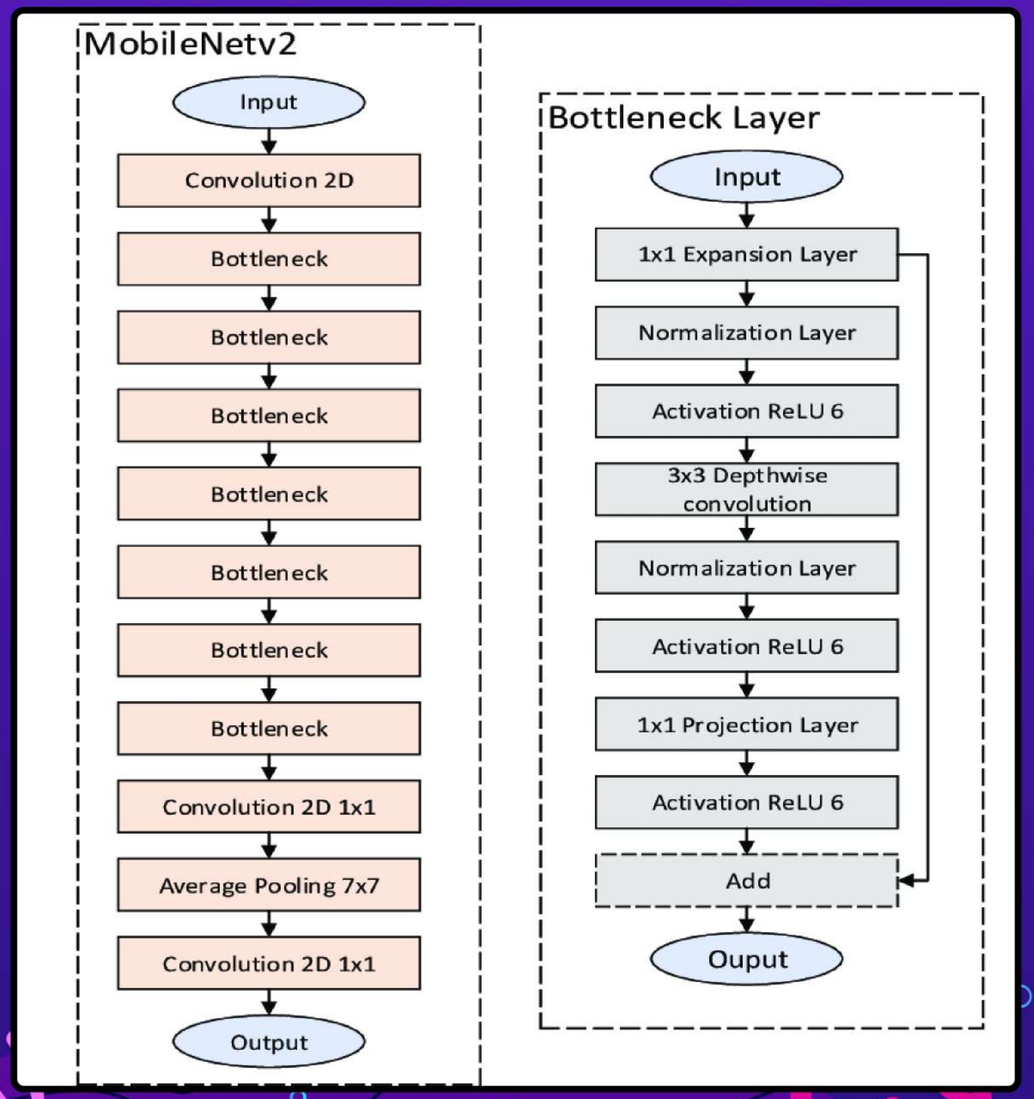
- Utilize pre-trained weights as feature extractor.
- Freeze base model weights for feature preservation.

MODEL TRAINING AND EVALUATION:

- Train MobileNetV2 (frozen) on dataset for 5 epochs.
- Monitor performance with validation data.
- Assess generalization on test dataset.
- Report test accuracy and loss.

MOBILENETV2 MODEL

MOBILENETV2 ARCHITECTURE

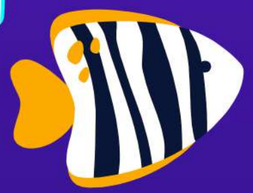


CNN MODELS

ACCURACY/LOSS OF DIFFERENT CNN MODELS

Model	Loss	Accuracy
CNN	0	1
VGG16	0	1
ResNet50	0	1
MobileNetV2	0	1

DETECTION



OPENCV AND YOLO

OpenCV Model Training and Evaluation

- **Data Preparation:** Gather unlabeled(should be labelled)image data for training and testing.
- **Feature Extraction:** Use OpenCV functions for feature extraction, such as edge detection or image descriptors.
- **Model Training:** Train machine learning models (e.g., SVM, KNN) on extracted features.
- **Evaluation Metrics:** Evaluate model performance using metrics like accuracy, precision, recall, and F1 score.
- **Results Analysis:** Analyze model performance and adjust parameters or features for optimization.

YOLO Model Training and Evaluation

- **Data Preparation:** Collect image and annotation paths for training, validation, and testing datasets.
- **Data Generator:** Implement YOLODataGenerator class for loading and preprocessing data in batches.
- **Model Training:** Build and compile YOLO model with Adam optimizer and sparse categorical crossentropy loss.
- **Training Details:** Train model for 5 epochs using YOLODataGenerator, monitor validation data during training.
- **Model Evaluation:** Evaluate model performance on validation and test datasets using evaluate method.
- **Evaluation Metrics:** Assess model performance through loss value on respective datasets.
- **Results Analysis:** Monitor training and validation metrics, evaluate model's object detection effectiveness on unseen data.

Layer (type)	Output Shape	Param #
conv2d_258 (Conv2D)	(None, 222, 222, 16)	448
max_pooling2d_64 (MaxPooling2D)	(None, 111, 111, 16)	0
conv2d_259 (Conv2D)	(None, 109, 109, 32)	4,640
max_pooling2d_65 (MaxPooling2D)	(None, 54, 54, 32)	0
conv2d_260 (Conv2D)	(None, 52, 52, 64)	18,496
max_pooling2d_66 (MaxPooling2D)	(None, 26, 26, 64)	0
flatten_14 (Flatten)	(None, 43264)	0
dense_28 (Dense)	(None, 128)	5,537,920
dense_29 (Dense)	(None, 1)	129

Total params: 5,561,633 (21.22 MB)

Trainable params: 5,561,633 (21.22 MB)

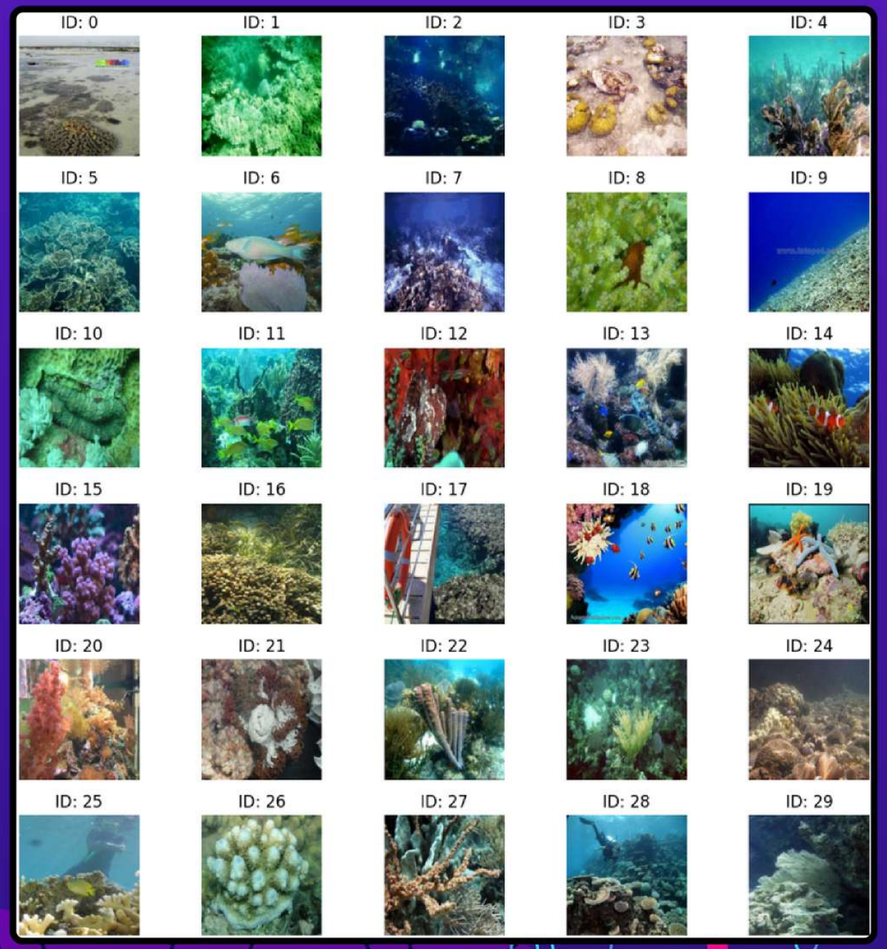
Non-trainable params: 0 (0.00 B)

Detection using YOLO is unsuccessful because of the absence of class labels in the dataset. Detection isn't possible without mapping images with numbers.

IMAGE MAPPING

Training Dataset (50 Images)

Testing Dataset (30 Images)

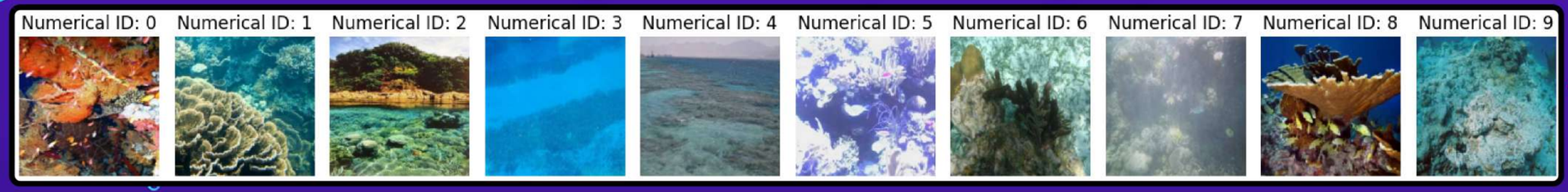


Validation Dataset (15 Images)

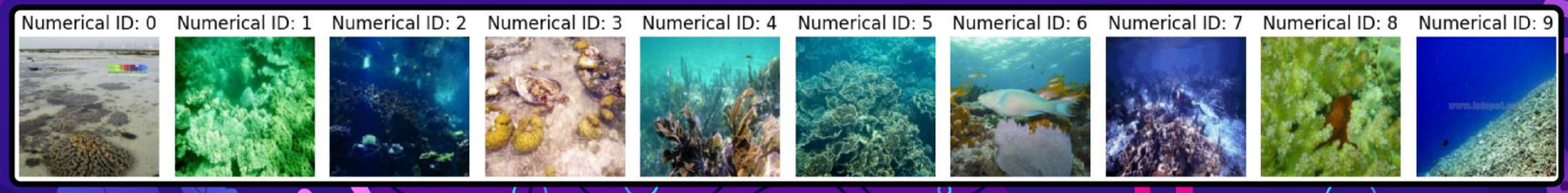
IMAGE MAPPING

Training Dataset (10 Images)

Testing Dataset (10 Images)



Validation Dataset (10 Images)



SEGMENTATION

SEGNET MODEL

SegNet is a convolutional neural network architecture designed for semantic segmentation tasks. The model consists of an encoder-decoder structure with skip connections to capture both local and global features.

ENCODER ARCHITECTURE:
The encoder comprises several convolutional blocks, each consisting of convolutional layers followed by max-pooling layers.
Feature maps are progressively downsampled to capture hierarchical representations of the input image.

DECODER ARCHITECTURE:
The decoder upsamples the feature maps using upsampling layers and reconstructs the segmented image.
Upsampling is performed using convolutional layers to retain spatial information.

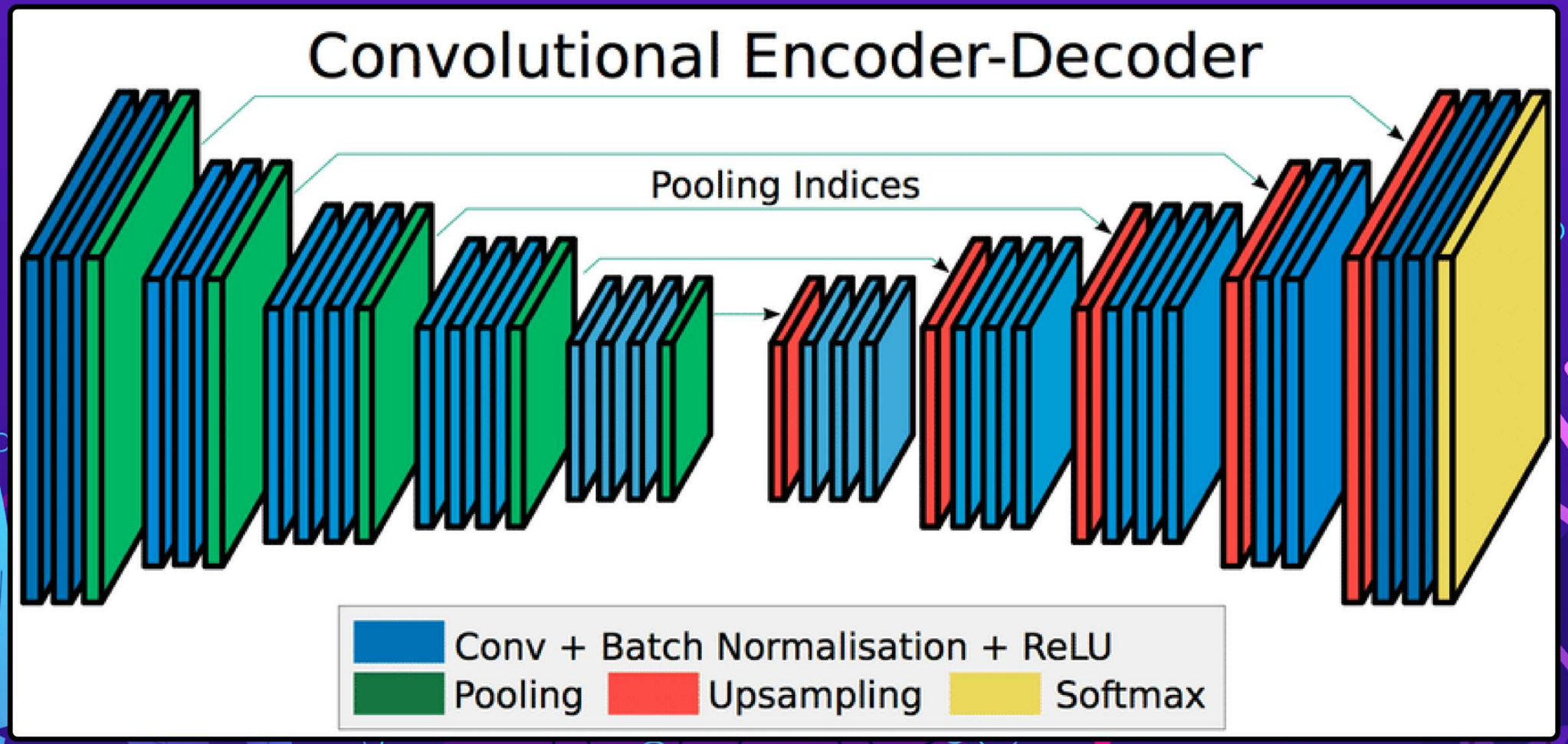
SKIP CONNECTIONS:
Skip connections are established between corresponding encoder and decoder layers to preserve fine-grained details during upsampling.

OUTPUT LAYER:
The final output layer employs a softmax activation function to generate class probabilities for each pixel.

MODEL COMPILATION:
The SegNet model is compiled using the Adam optimizer and sparse categorical crossentropy loss function, suitable for segmentation tasks.
Accuracy is used as a metric to monitor model performance during training.

SEGNET MODEL

SEGNET ARCHITECTURE



PARAMETERS

Layer (type)	Output Shape	Param #
input_layer_37 (InputLayer)	(None, 416, 416, 3)	0
conv2d_285 (Conv2D)	(None, 416, 416, 64)	1,792
batch_normalization_38 (BatchNormalization)	(None, 416, 416, 64)	256
conv2d_286 (Conv2D)	(None, 416, 416, 64)	36,928
batch_normalization_39 (BatchNormalization)	(None, 416, 416, 64)	256
max_pooling2d_74 (MaxPooling2D)	(None, 208, 208, 64)	0
conv2d_287 (Conv2D)	(None, 208, 208, 128)	73,856
batch_normalization_40 (BatchNormalization)	(None, 208, 208, 128)	512
conv2d_288 (Conv2D)	(None, 208, 208, 128)	147,584
batch_normalization_41 (BatchNormalization)	(None, 208, 208, 128)	512
max_pooling2d_75 (MaxPooling2D)	(None, 104, 104, 128)	0
conv2d_289 (Conv2D)	(None, 104, 104, 256)	295,168
batch_normalization_42 (BatchNormalization)	(None, 104, 104, 256)	1,024

Total params: 17,058,561 (65.07 MB)

Trainable params: 17,048,321 (65.03 MB)

Non-trainable params: 10,240 (40.00 KB)

FCN MODEL

Fully Convolutional Networks (FCNs) are a type of neural network architecture used for semantic segmentation tasks, where the goal is to assign each pixel in an image to a specific class label. FCNs have gained popularity due to their ability to efficiently process and segment images of arbitrary sizes.

ENCODER ARCHITECTURE:
The encoder consists of several convolutional and max-pooling layers to extract hierarchical features from the input image. Each convolutional layer is followed by a rectified linear unit (ReLU) activation function to introduce non-linearity.

DECODER ARCHITECTURE:
The decoder upsamples the feature maps to generate dense pixel-wise predictions. Upsampling is performed using upsampling layers followed by convolutional layers.

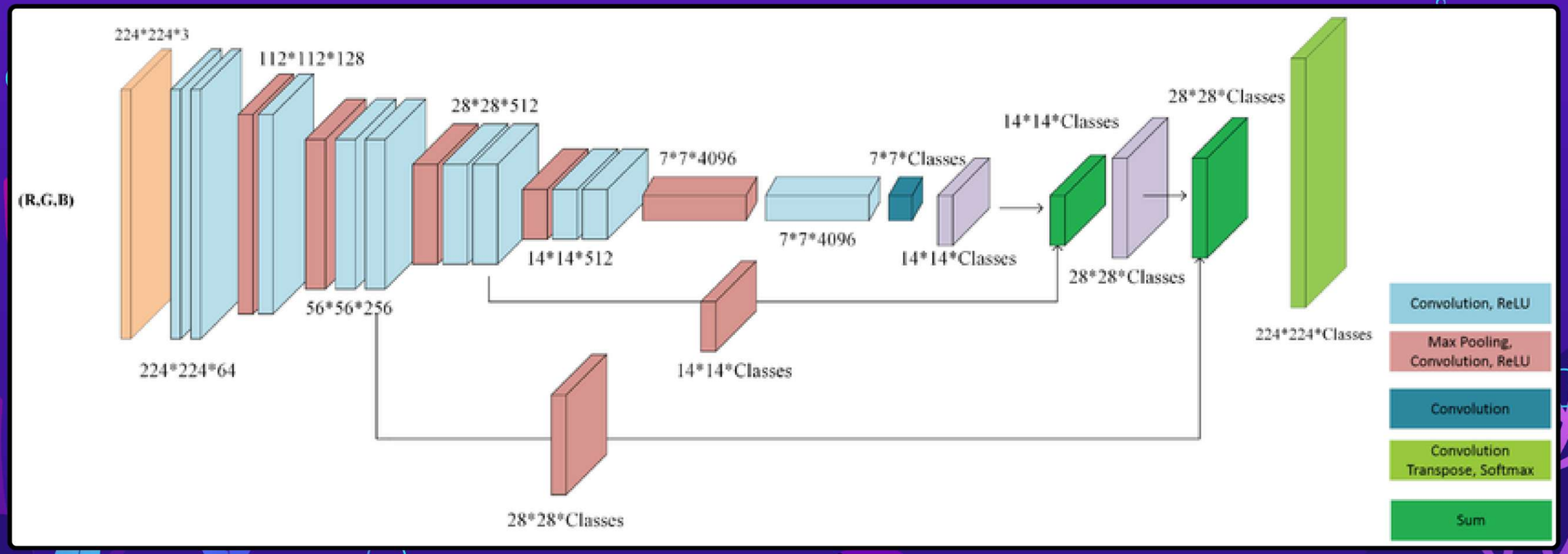
SKIP CONNECTIONS:
Skip connections are often used to combine low-level and high-level features, enabling the model to capture both fine details and global context.

OUTPUT LAYER:
The output layer produces class probabilities for each pixel using a softmax activation function. The number of output channels is determined by the number of classes in the segmentation task.

MODEL CREATION:
The FCN model is created using the Keras functional API, which allows for flexible model architectures.
The model takes an input image of arbitrary size and outputs a segmentation map with the same spatial dimensions as the input.

FCN MODEL

FCN ARCHITECTURE



PARAMETERS

Layer (type)	Output Shape	Param #
input_layer_38 (InputLayer)	(None, 416, 416, 3)	0
conv2d_305 (Conv2D)	(None, 416, 416, 64)	1,792
conv2d_306 (Conv2D)	(None, 416, 416, 64)	36,928
max_pooling2d_78 (MaxPooling2D)	(None, 208, 208, 64)	0
conv2d_307 (Conv2D)	(None, 208, 208, 128)	73,856
conv2d_308 (Conv2D)	(None, 208, 208, 128)	147,584
max_pooling2d_79 (MaxPooling2D)	(None, 104, 104, 128)	0
conv2d_309 (Conv2D)	(None, 104, 104, 256)	295,168
conv2d_310 (Conv2D)	(None, 104, 104, 256)	590,080
max_pooling2d_80 (MaxPooling2D)	(None, 52, 52, 256)	0
up_sampling2d_56 (UpSampling2D)	(None, 104, 104, 256)	0
conv2d_311 (Conv2D)	(None, 104, 104, 256)	590,080
conv2d_312 (Conv2D)	(None, 104, 104, 256)	590,080
up_sampling2d_57 (UpSampling2D)	(None, 208, 208, 256)	0

Total params: 2,878,977 (10.98 MB)

Trainable params: 2,878,977 (10.98 MB)

Non-trainable params: 0 (0.00 B)

ENET MODEL

EfficientNet is a family of convolutional neural network architectures that have achieved state-of-the-art performance with significantly fewer parameters compared to traditional models. ENet utilizes the EfficientNet backbone for feature extraction and adds a segmentation head for semantic segmentation tasks.

ENET BACKBONE:
The backbone of ENet consists of multiple convolutional layers organized in a hierarchical manner. These layers are responsible for extracting high-level features from input images while maintaining computational efficiency.

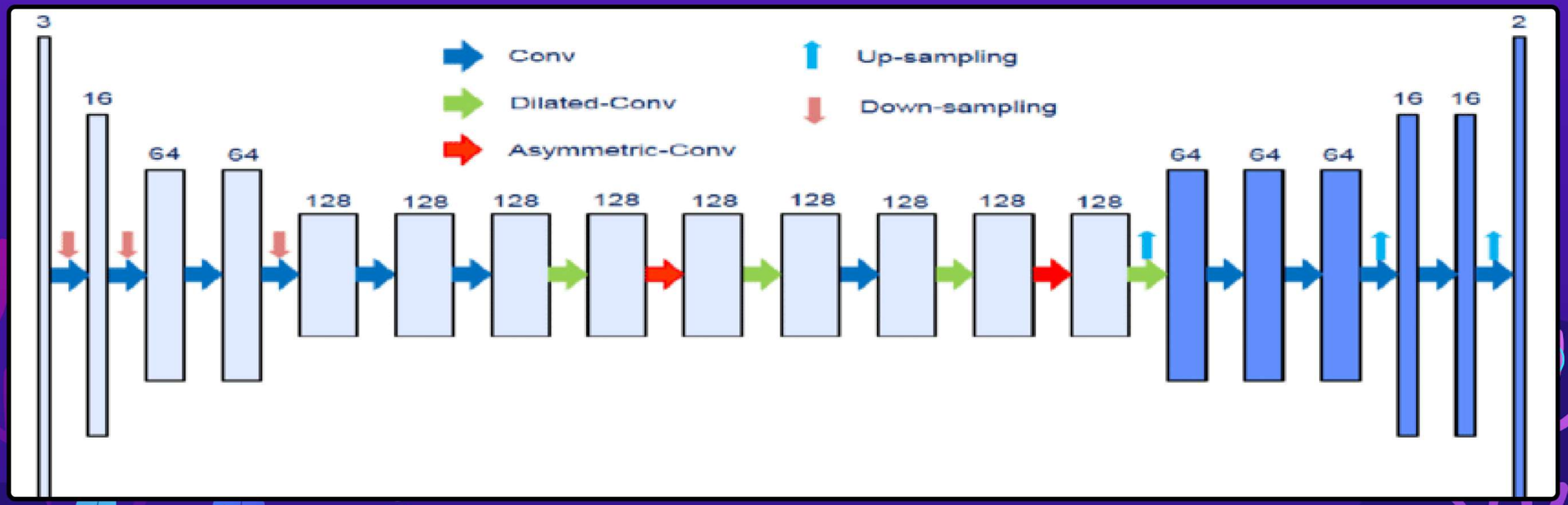
SEGMENTATION HEAD:
The segmentation head is added on top of the EfficientNet backbone. It typically consists of a single convolutional layer followed by an upsampling layer. The convolutional layer produces pixel-wise class probabilities, and the upsampling layer increases the resolution of the segmentation map to match the input image size.

FREEZING PRE-TRAINED WEIGHTS:
In ENet, the weights of the pre-trained EfficientNet backbone are frozen during training to prevent them from being updated. This helps retain the learned features from the ImageNet dataset and accelerates training for segmentation tasks.

MODEL CREATION:
The ENet model is created by combining the pre-trained EfficientNet backbone with a segmentation head using the Keras functional API.
The model takes an input image of arbitrary size and outputs a segmentation map with the same dimensions as the input, where each pixel is assigned a class label.

ENET MODEL

ENET ARCHITECTURE



PARAMETERS

Layer (type)	Output Shape	Param #	Connected to
input_layer_39 (InputLayer)	(None, 416, 416, 3)	0	-
rescaling_4 (Rescaling)	(None, 416, 416, 3)	0	input_layer_39[0…
normalization_2 (Normalization)	(None, 416, 416, 3)	7	rescaling_4[0][0]
rescaling_5 (Rescaling)	(None, 416, 416, 3)	0	normalization_2[…
stem_conv_pad (ZeroPadding2D)	(None, 417, 417, 3)	0	rescaling_5[0][0]
stem_conv (Conv2D)	(None, 208, 208, 32)	864	stem_conv_pad[0]…
stem_bn (BatchNormalizatio…	(None, 208, 208, 32)	128	stem_conv[0][0]
stem_activation (Activation)	(None, 208, 208, 32)	0	stem_bn[0][0]
block1a_dwconv (DepthwiseConv2D)	(None, 208, 208, 32)	288	stem_activation[…
block1a_bn (BatchNormalizatio…	(None, 208, 208, 32)	128	block1a_dwconv[0…

Total params: 4,050,852 (15.45 MB)

Trainable params: 1,281 (5.00 KB)

Non-trainable params: 4,049,571 (15.45 MB)

U-NET MODEL

U-Net is a convolutional neural network architecture designed for biomedical image segmentation tasks. It was developed by Olaf Ronneberger, Philipp Fischer, and Thomas Brox and is widely used in medical image analysis due to its effectiveness in segmenting anatomical structures.

ENCODER-DECODER ARCHITECTURE:
U-Net follows an encoder-decoder architecture where the encoder captures context information from the input image through a series of convolutional and pooling layers, while the decoder gradually recovers spatial information by upsampling feature maps.

SKIP CONNECTIONS:
U-Net introduces skip connections between corresponding encoder and decoder layers to mitigate information loss during downsampling and upsampling. These connections concatenate feature maps from the encoder to the decoder, enabling precise localization of object boundaries.

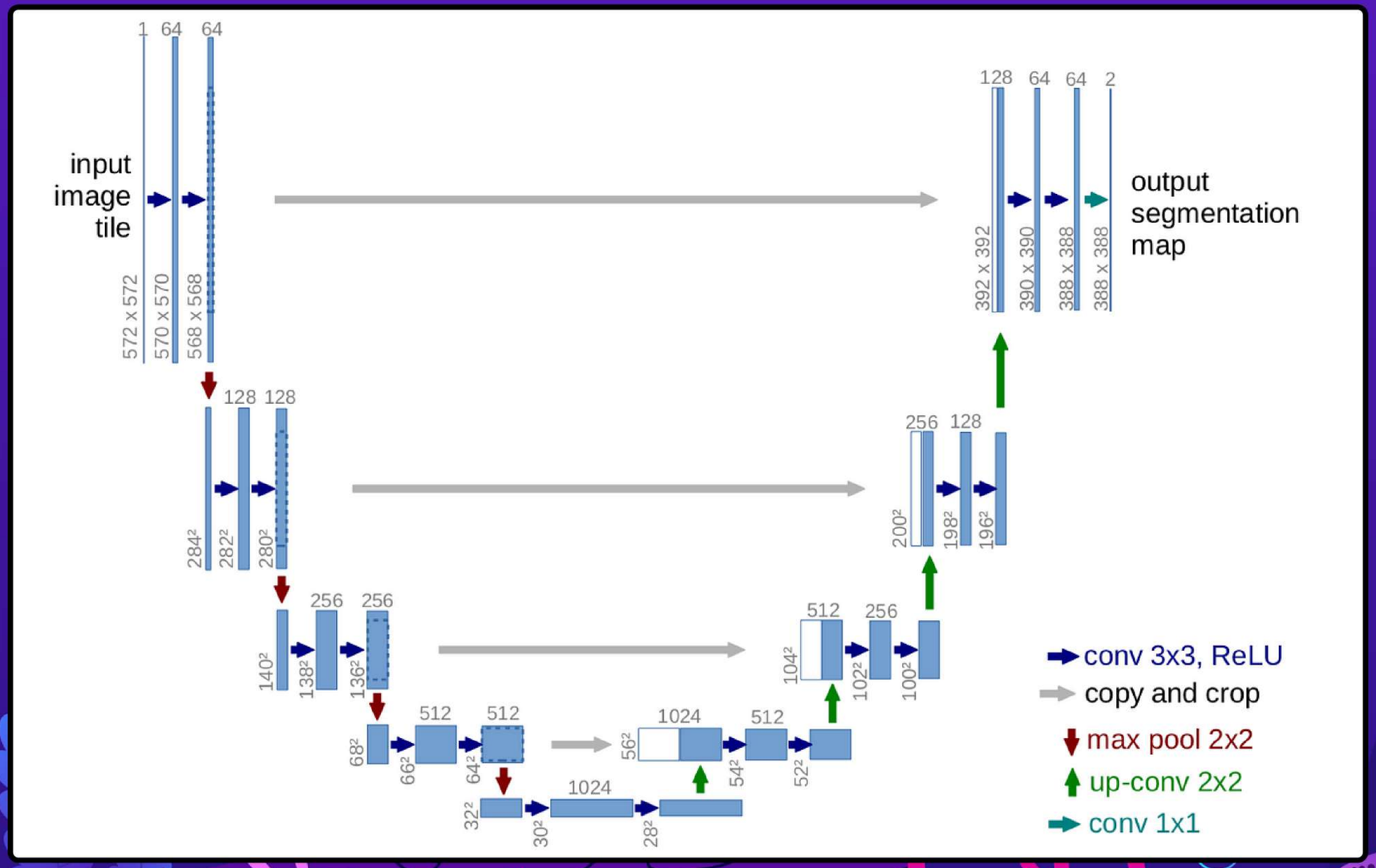
SYMMETRIC DESIGN:
The encoder and decoder paths in U-Net are symmetric, allowing for efficient feature extraction and precise localization of objects in the output segmentation map.

FINAL LAYER:
The final layer of U-Net typically consists of a convolutional layer with softmax activation to produce pixel-wise class probabilities for each class in the segmentation task.

MODEL CREATION:
The U-Net model uses convolutional and pooling layers in the encoder, transposed convolutional layers in the decoder, and skip connections between corresponding layers. It takes an input image and produces a segmentation map of the same size, assigning a class label to each pixel.

U-NET MODEL

U-NET ARCHITECTURE



PARAMETERS

Layer (type)	Output Shape	Param #	Connected to
input_layer_40 (InputLayer)	(None, 416, 416, 3)	0	-
conv2d_319 (Conv2D)	(None, 416, 416, 64)	1,792	input_layer_40[0…
conv2d_320 (Conv2D)	(None, 416, 416, 64)	36,928	conv2d_319[0][0]
max_pooling2d_81 (MaxPooling2D)	(None, 208, 208, 64)	0	conv2d_320[0][0]
conv2d_321 (Conv2D)	(None, 208, 208, 128)	73,856	max_pooling2d_81…
conv2d_322 (Conv2D)	(None, 208, 208, 128)	147,584	conv2d_321[0][0]
max_pooling2d_82 (MaxPooling2D)	(None, 104, 104, 128)	0	conv2d_322[0][0]
conv2d_323 (Conv2D)	(None, 104, 104, 256)	295,168	max_pooling2d_82…
conv2d_324 (Conv2D)	(None, 104, 104, 256)	590,080	conv2d_323[0][0]
max_pooling2d_83 (MaxPooling2D)	(None, 52, 52, 256)	0	conv2d_324[0][0]

Total params: 31,031,745 (118.38 MB)

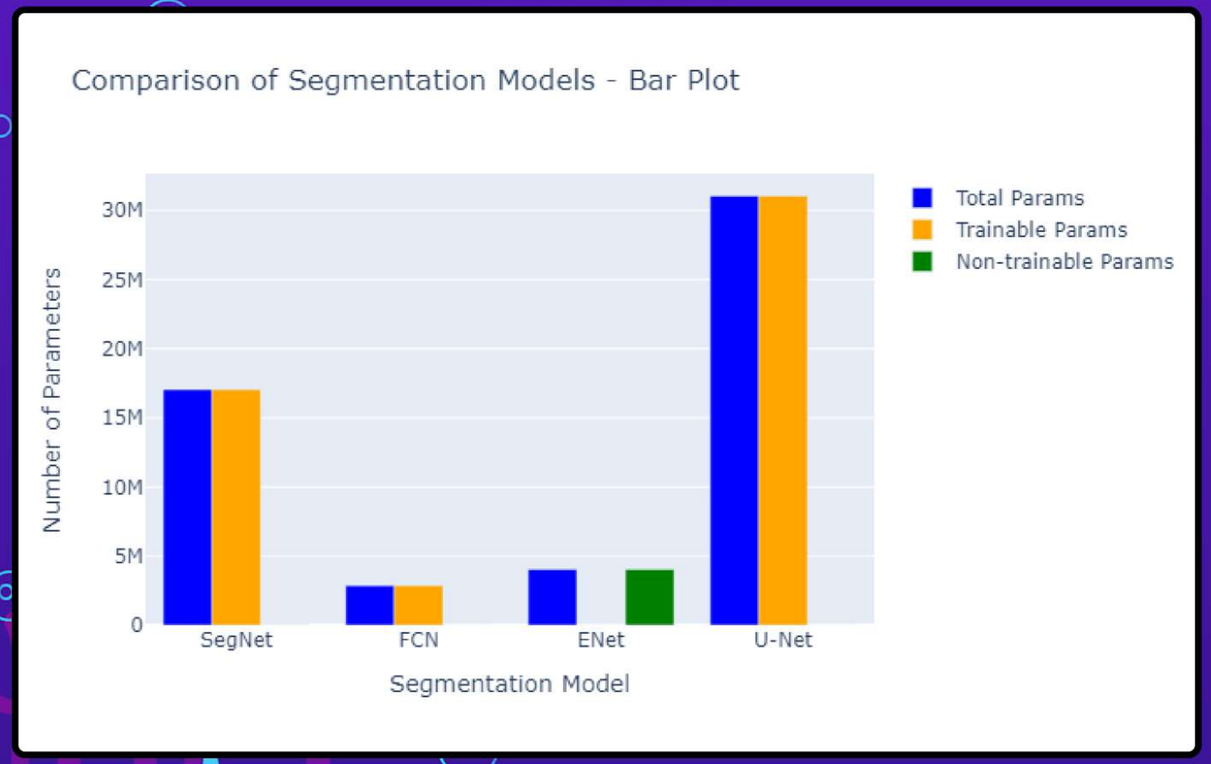
Trainable params: 31,031,745 (118.38 MB)

Non-trainable params: 0 (0.00 B)

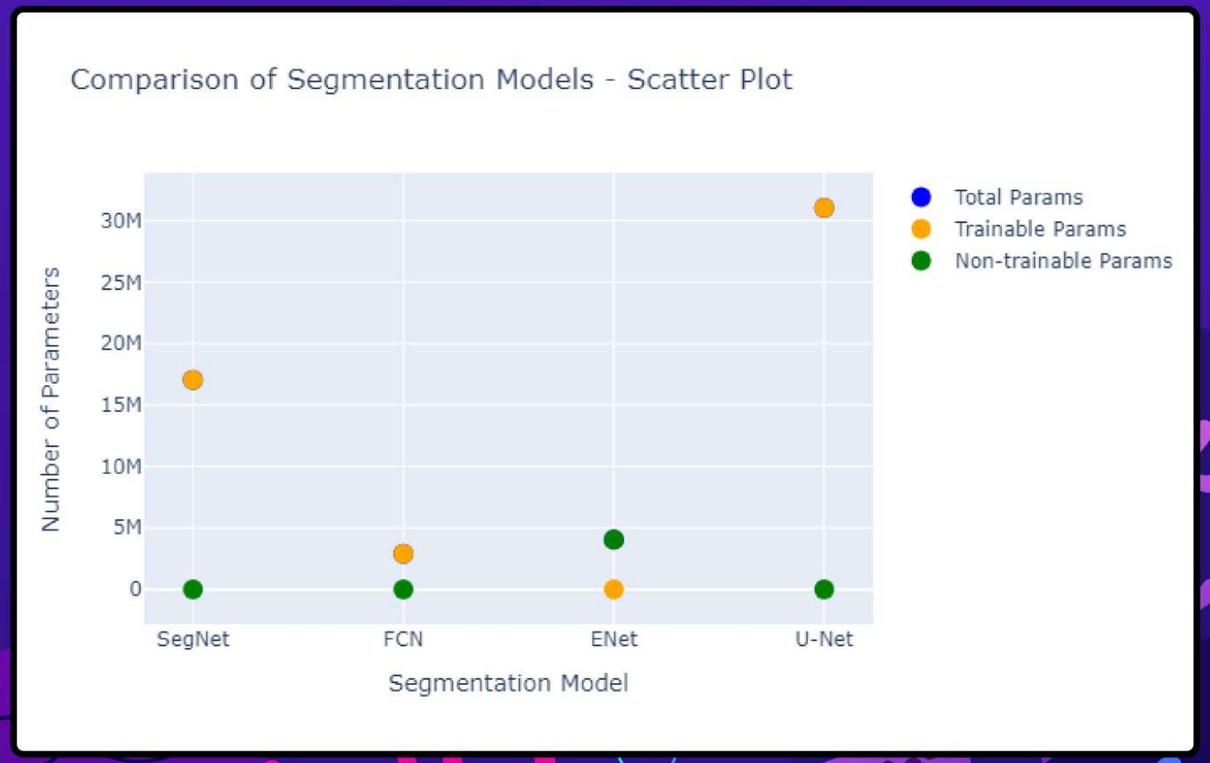
VISUALIZATION

GRAPHICAL VISUALIZATION

Bar Plot



Scatter Plot



LABELLED IMAGES

CLASSIFICATION

CNN MODEL USING FUNCTIONAL API

USING CIFAR-10 IMAGES

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class.

CNN MODEL USING FUNCTIONAL API

INPUT LAYER: Input shape from x_train[0].
CONVOLUTIONAL LAYERS: Two sets of 32 filters, (3, 3) kernel, ReLU activation, 'same' padding, followed by batch normalization.
MAXPOOLING LAYERS: (2, 2) pool size after each convolutional layer.
ADDITIONAL CONVOLUTIONAL LAYERS: 64 and 128 filters, (3, 3) kernel, ReLU activation, 'same' padding, batch normalization.
FLATTEN LAYER: Flattens output for dense layers.
DROPOUT LAYERS: Dropout rate of 0.2 before and after the dense layer.
DENSE LAYERS: Hidden layer with 1024 units, ReLU activation.
OUTPUT LAYERS: Dense layer with softmax activation for class probabilities (K = number of classes).
MODEL COMPILATION: Optimizer, loss function, and metrics not specified in the code snippet.
MODEL SUMMARY: Total number of parameters and trainable parameters printed using model.summary().

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 32, 32, 3)	0
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 1024)	2,098,176
dropout_1 (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 10)	10,250

Total params: 2,397,226 (9.14 MB)

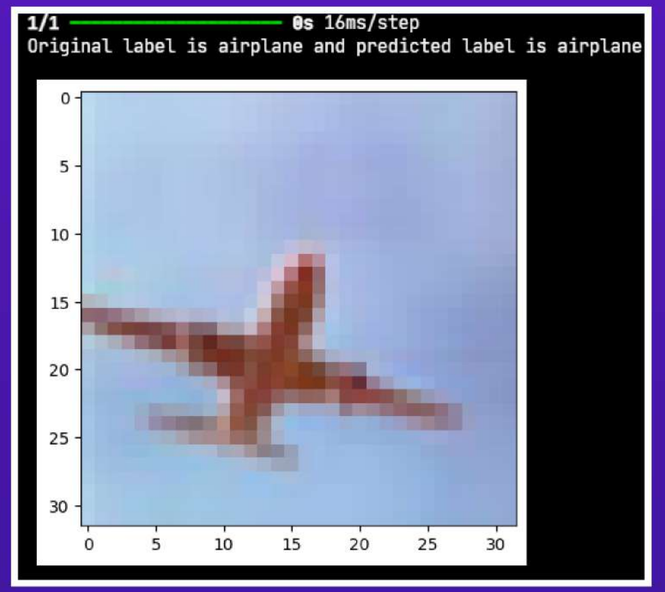
Trainable params: 2,396,330 (9.14 MB)

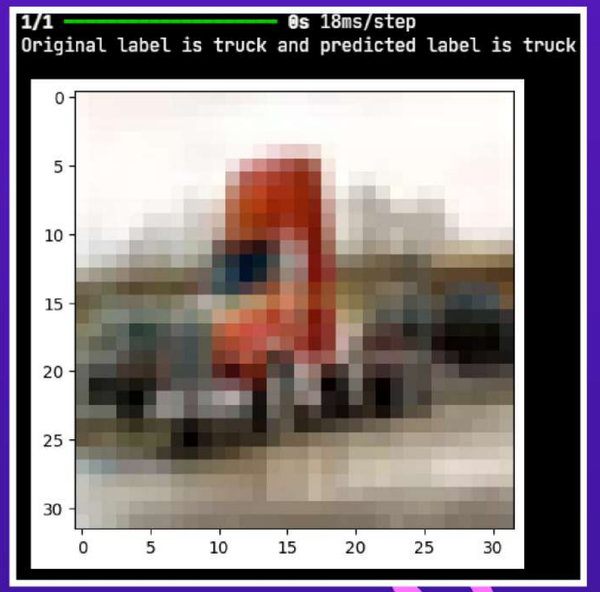
Non-trainable params: 896 (3.50 KB)

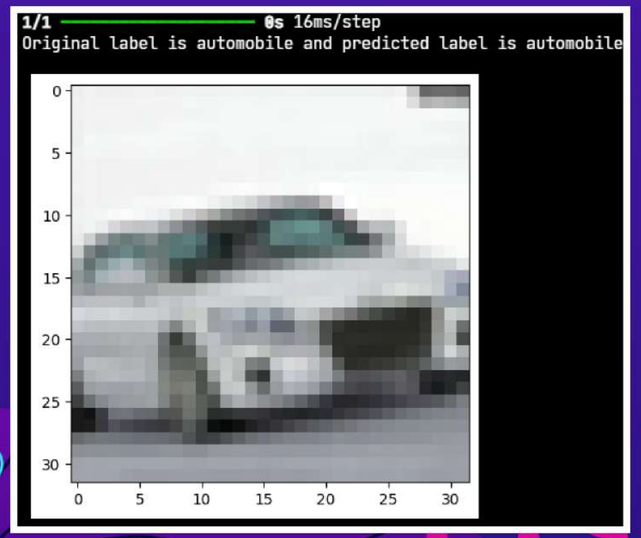
CNN MODEL USING FUNCTIONAL API

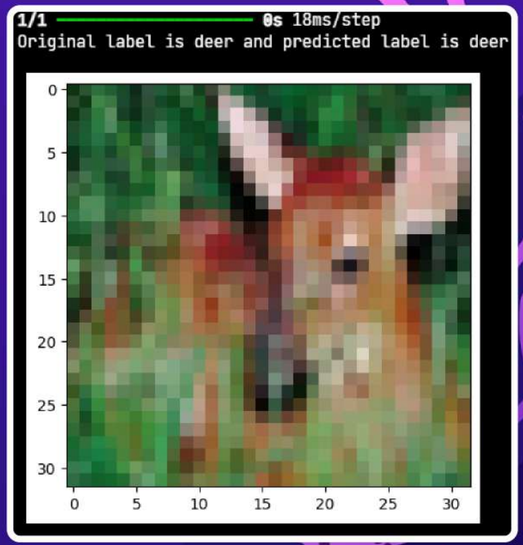
```
labels = '''airplane automobile bird cat deer dog
frog horse ship truck'''.split()

# select the image from our test dataset
image_number = 0

# display the image
plt.imshow(x_test[image_number])

# load the image in an array
n = np.array(x_test[image_number])

# reshape it
p = n.reshape(1, 32, 32, 3)

# pass in the network for prediction and
# save the predicted label
predicted_label = labels[model.predict(p).argmax()]

# load the original label
original_label = labels[y_test[image_number]]

# display the result
print("Original label is {} and predicted label is
{}".format(original_label, predicted_label))
```

Correct Predictions





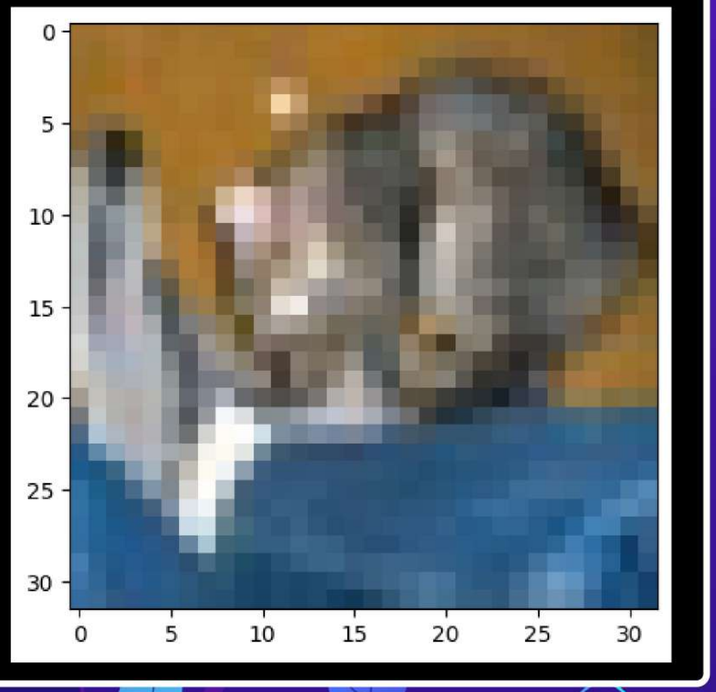




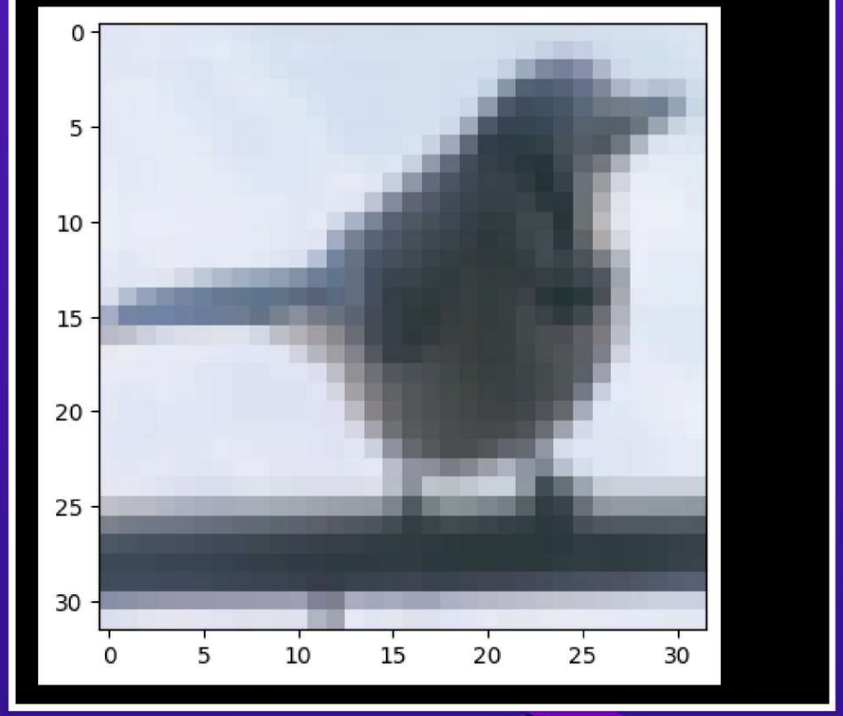
CNN MODEL USING FUNCTIONAL API

Incorrect Predictions

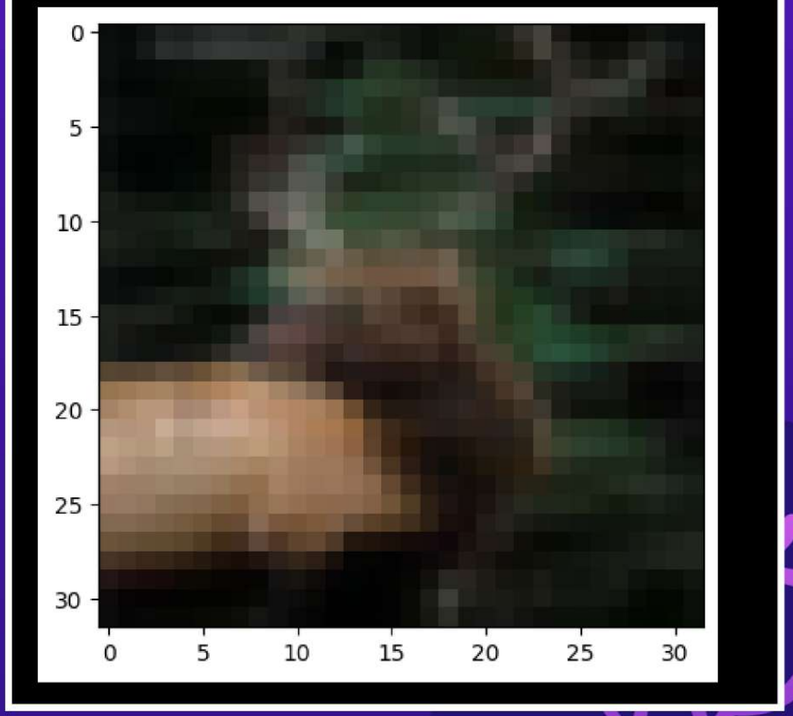
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step
Original label is cat and predicted label is ship



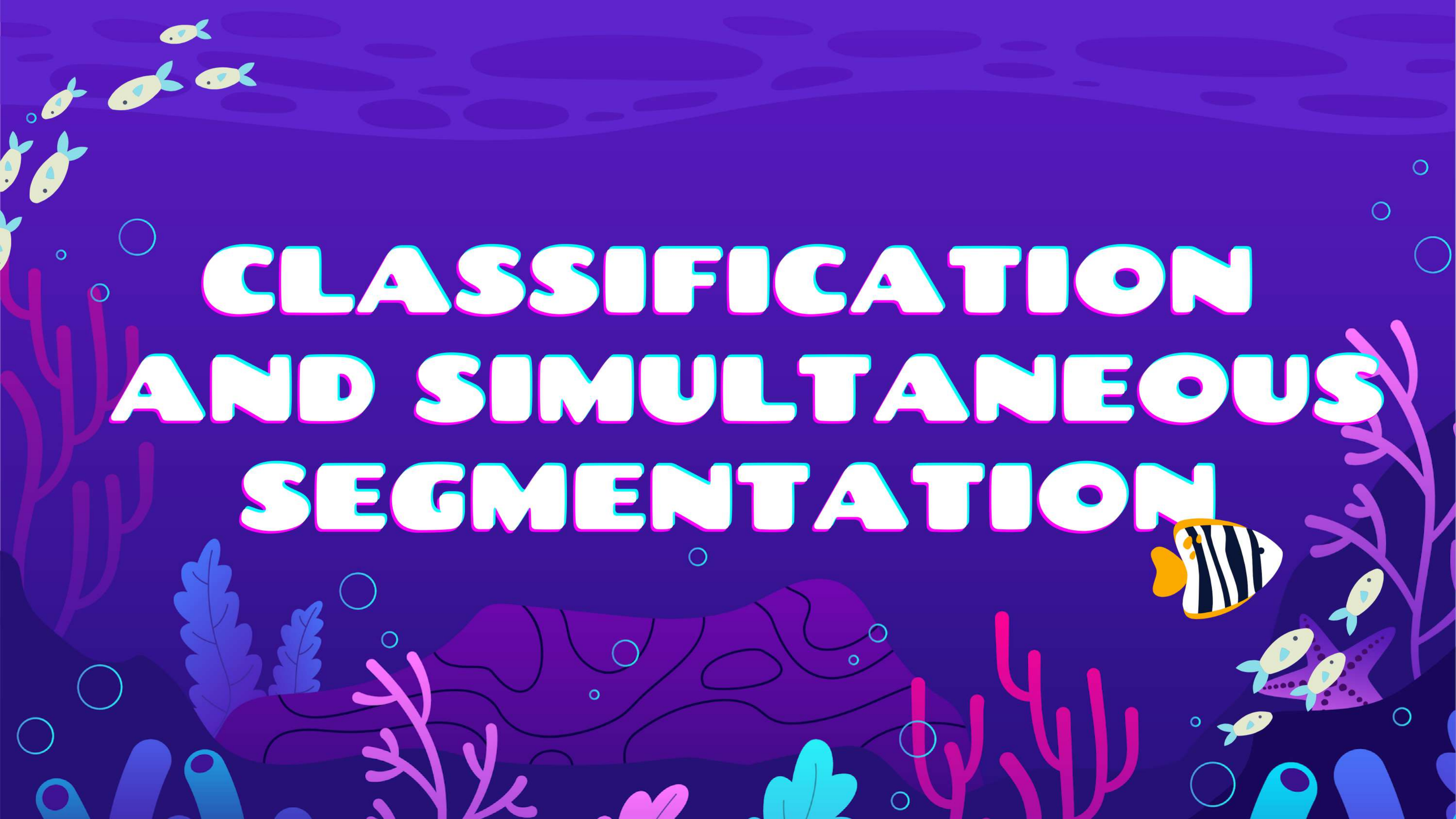
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Original label is bird and predicted label is airplane



1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step
Original label is deer and predicted label is frog



CLASSIFICATION AND SIMULTANEOUS SEGMENTATION



BASIC CNN TO CLASSIFY

```
# Define CNN model
model = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.Flatten(),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])
```

```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - accuracy: 0.3281 - loss: 1.7967 - val_accuracy: 0.5353 - val_loss: 1.3019
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.5635 - loss: 1.2219 - val_accuracy: 0.5879 - val_loss: 1.1560
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.6318 - loss: 1.0496 - val_accuracy: 0.6213 - val_loss: 1.0882
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.6724 - loss: 0.9409 - val_accuracy: 0.6643 - val_loss: 0.9583
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.6999 - loss: 0.8634 - val_accuracy: 0.6830 - val_loss: 0.9219
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.7160 - loss: 0.8063 - val_accuracy: 0.6983 - val_loss: 0.8614
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.7384 - loss: 0.7531 - val_accuracy: 0.6999 - val_loss: 0.8711
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.7555 - loss: 0.6997 - val_accuracy: 0.7198 - val_loss: 0.8217
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.7662 - loss: 0.6627 - val_accuracy: 0.7180 - val_loss: 0.8536
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.7798 - loss: 0.6243 - val_accuracy: 0.7108 - val_loss: 0.8920
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7126 - loss: 0.8817
Test accuracy: 0.7107999920845032
```

U-NET MODEL FOR SEGMENTATION

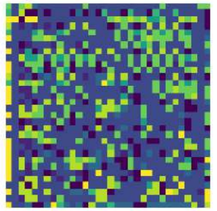
```
# Define U-Net model for segmentation
def unet_model():
  inputs = layers.Input((32, 32, 3))
  conv1 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
  conv1 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
  pool1 = layers.MaxPooling2D(pool_size=(2, 2))(conv1)

  conv2 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
  conv2 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
  pool2 = layers.MaxPooling2D(pool_size=(2, 2))(conv2)

  conv3 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)
  conv3 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)
  pool3 = layers.MaxPooling2D(pool_size=(2, 2))(conv3)

  up4 = layers.concatenate([layers.Conv2DTranspose(64, (2, 2), strides=(2, 2),
padding='same')(conv3), conv2], axis=3)
  conv4 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(up4)
  conv4 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(conv4)

  up5 = layers.concatenate([layers.Conv2DTranspose(32, (2, 2), strides=(2, 2),
padding='same')(conv4), conv1], axis=3)
  conv5 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(up5)
  conv5 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(conv5)

  outputs = layers.Conv2D(10, (1, 1), activation='softmax')(conv5)

  model = models.Model(inputs=[inputs], outputs=[outputs])
  return model
```

```
Epoch 1/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 4s 110ms/step - accuracy: 0.1000 - loss: 2.3034 - val_accuracy: 0.0997 - val_loss: 2.3026
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step - accuracy: 0.1002 - loss: 2.3026 - val_accuracy: 0.1009 - val_loss: 2.3026
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 90ms/step - accuracy: 0.1011 - loss: 2.3026 - val_accuracy: 0.1003 - val_loss: 2.3026
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 86ms/step - accuracy: 0.1015 - loss: 2.3025 - val_accuracy: 0.1011 - val_loss: 2.3026
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 85ms/step - accuracy: 0.1016 - loss: 2.3025 - val_accuracy: 0.1012 - val_loss: 2.3026
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 90ms/step - accuracy: 0.1027 - loss: 2.3025 - val_accuracy: 0.1007 - val_loss: 2.3026
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 87ms/step - accuracy: 0.1033 - loss: 2.3024 - val_accuracy: 0.1008 - val_loss: 2.3026
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 84ms/step - accuracy: 0.1043 - loss: 2.3023 - val_accuracy: 0.1006 - val_loss: 2.3026
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 86ms/step - accuracy: 0.1051 - loss: 2.3022 - val_accuracy: 0.0998 - val_loss: 2.3027
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 92ms/step - accuracy: 0.1062 - loss: 2.3020 - val_accuracy: 0.1015 - val_loss: 2.3027
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.1014 - loss: 2.3027
Test accuracy: 0.10146484524011612
```

PREDICTION

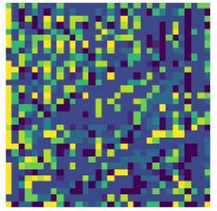
True Label: 3
Predicted Label: 3

Segmentation Mask



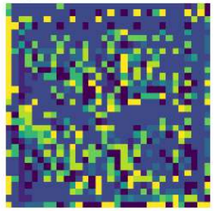
True Label: 2
Predicted Label: 2

Segmentation Mask



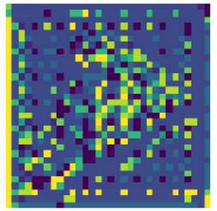
True Label: 1
Predicted Label: 1

Segmentation Mask

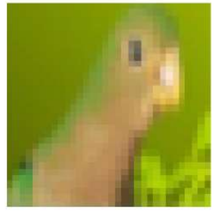


True Label: 0
Predicted Label: 0

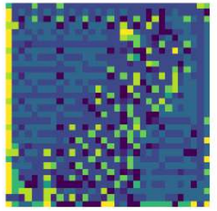
Segmentation Mask



True Label: 2
Predicted Label: 2

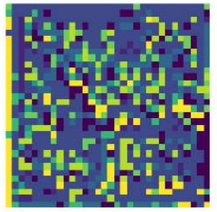


Segmentation Mask



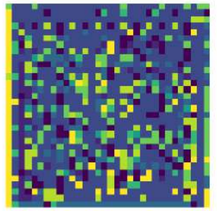
True Label: 7
Predicted Label: 7

Segmentation Mask



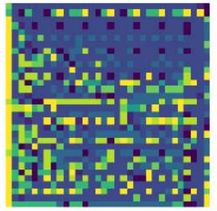
True Label: 6
Predicted Label: 6

Segmentation Mask



True Label: 8
Predicted Label: 8

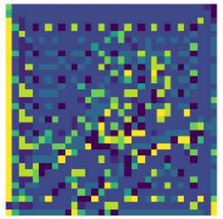
Segmentation Mask



PREDICTION

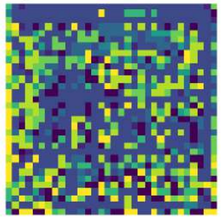
True Label: 2
Predicted Label: 2

Segmentation Mask



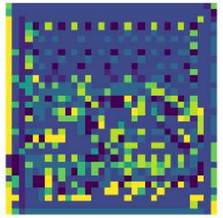
True Label: 9
Predicted Label: 9

Segmentation Mask



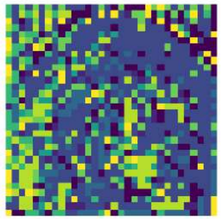
True Label: 1
Predicted Label: 1

Segmentation Mask



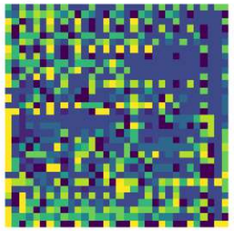
True Label: 5
Predicted Label: 5

Segmentation Mask



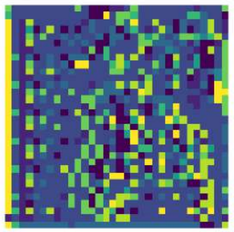
True Label: 1
Predicted Label: 1

Segmentation Mask



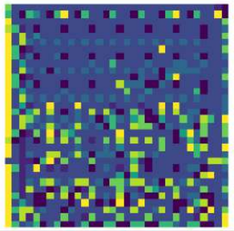
True Label: 5
Predicted Label: 7

Segmentation Mask



True Label: 4
Predicted Label: 4

Segmentation Mask



CONCLUSION

IAM-LUI: Implementing Advanced Models on Labelled and Unlabelled Images

In the realm of marine conservation and ecological research, the Implementing Advanced Models on Labelled and Unlabelled Images (IAM-LUI) project emerges as a pioneering endeavor to understand and protect one of the Earth's most vital ecosystems: coral reefs. Throughout our journey, we delved into the intricate world of coral reef images, employing state-of-the-art machine learning models and data processing techniques to extract invaluable insights.

Our expedition began with meticulous data collection, curating a comprehensive dataset rich in diverse coral reef images. Leveraging this dataset, we embarked on an exploration of cutting-edge models, each tailored to decipher the complex attributes embedded within these images.

The SegNet, PSPNet, FCN, ENet, and U-Net architectures stood as stalwart companions throughout our quest, each offering unique perspectives and capabilities in unraveling the mysteries concealed within coral reef imagery. From semantic segmentation to scene parsing and instance segmentation, these models bestowed upon us the power to discern intricate details, paving the way for a deeper understanding of coral reef ecosystems.

As we ventured deeper into the depths of our dataset, the YOLO model emerged as a beacon of hope for object detection tasks, enabling us to identify and localize key elements within coral reef images with remarkable precision.

Our odyssey was not without its challenges. From intricacies in data preprocessing to the fine-tuning of model hyperparameters, we encountered obstacles at every turn. Yet, through perseverance and ingenuity, we overcame these hurdles, emerging stronger and more knowledgeable with each stride.

In the end, IAM-LUI stands not only as a testament to our dedication to marine conservation but also as a testament to the transformative power of technology in safeguarding our planet's most precious ecosystems. As we forge ahead, let IAM-LUI serve as a beacon of hope and inspiration for future endeavors in the noble pursuit of understanding and preserving coral reefs for generations to come.

PRESENTED BY

<JAGANNATH MONDAL> <2105546>

<RIDDHI GHOSH> <2105566>

<SAI SANKET BAL> <21051164>

<SAMBIT KUMAR NAYAK> <21052786>

<SNEHASIS NAYAK> <21051768>