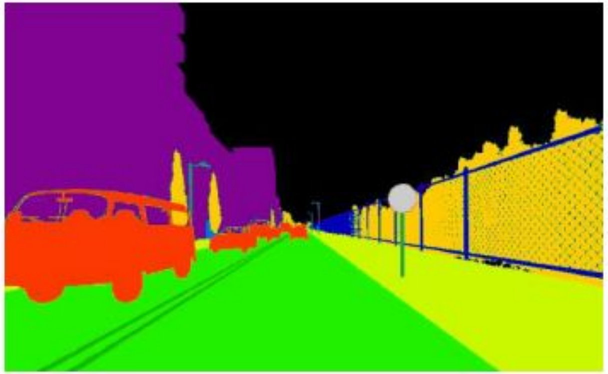
Image Segmentation Project

Anushka Gupta
Kashish Aggarwal
Mahua Singh
Mahek Chaudhary

SEMANTIC SEGMENTATION

Input image



Semantic Segmentation

Semantic segmentation is a computer vision task that involves-

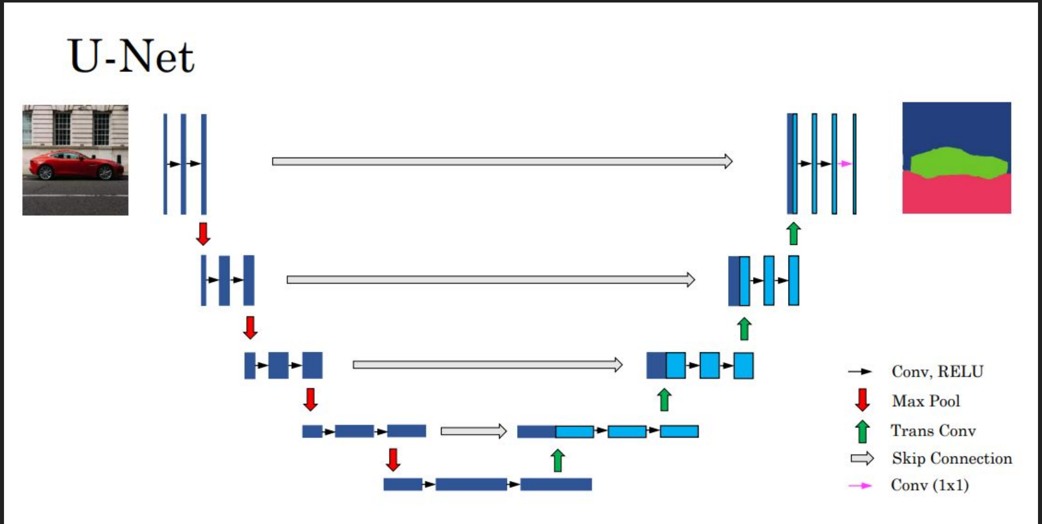
- Classifying each pixel in an image to a specific object category or class
- Labeling of objects within an image, enabling applications like object recognition,advanced image analysis and interpretation.

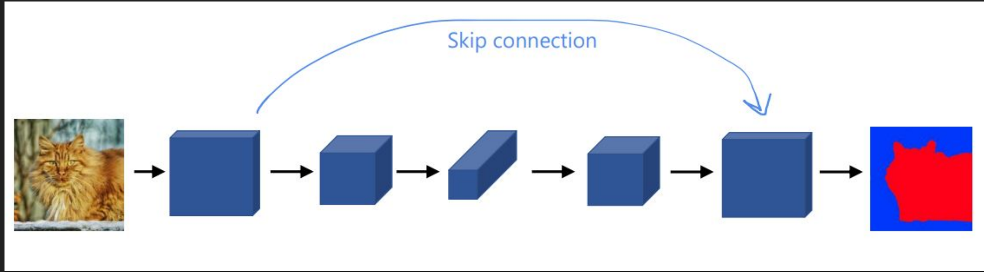
What is U- Net?

➔ Contracting path
- Channels increase
- Resolution decreases
- Gradually generate higher level features

➔ Expansive path
- Transpose convolutions
- Channels decrease
- Resolution increases
- More precise localisation

➔ Skip connection by concatenation
- Merge high-level and low-level information.
- Restore spatial resolution.
- Enable multi-scale feature integration.





ABOUT THE DATASET

This dataset has 2975 training images files and 500 validation image files. Each image file is 256x512 pixels, and each file is a composite with the original photo on the left half of the image, alongside the labeled image (output of semantic segmentation) on the right half.

Sample image in our dataset

LIBRARIES USED

- Tensorflow
- MatPlotLib
- NumPy
- CV2

```
[ ] #!pip install tensorflow
    import os
    import numpy as np
    import matplotlib.pyplot as plt
    from matplotlib.pyplot import imshow
    import tensorflow as tf
    import cv2
    from tensorflow.keras.layers import Input
    from tensorflow.keras.layers import Conv2D
    from tensorflow.keras.layers import concatenate
    from tensorflow.keras.layers import Conv2DTranspose
    from tensorflow.keras.layers import MaxPooling2D
    from tensorflow.keras.layers import Dropout
    from tensorflow.keras.callbacks import ModelCheckpoint
    %matplotlib inline
```

APPROACH

- We have mask for an image, where different color represents different classes.
- For each mask, we need to know what pixel belongs to what class. For example, if a car is present, all pixels in the car belong to class car and would have id as 26 (see Label). Hence, we will give a mask which is of shape(256x256x3) (depth dimension for color) and receive an array of shape(256x256) where each pixel will contains values from 0 to NUM_CLASSES -1 ,i.e, a single class.
- So we decided to choose the label, which were the closest distance (pixel wise) to our chosen pixel. Distance between two pixels can be said as the square root of norm between their difference.

```
labels = [
    #       name                     id       color
    Label(  'unlabeled'            ,  0 ,  , (  0,  0,  0) ),
    Label(  'ego vehicle'          ,  1 ,  , (  0,  0,  0) ),
    Label(  'rectification border' ,  2 ,  , (  0,  0,  0) ),
    Label(  'out of roi'           ,  3 ,  , (  0,  0,  0) ),
    Label(  'static'               ,  4 ,  , (  0,  0,  0) ),
    Label(  'dynamic'              ,  5 ,  , (111, 74,  0) ),
    Label(  'ground'               ,  6 ,  , ( 81,  0, 81) ),
    Label(  'road'                 ,  7 ,  , (128, 64,128) ),
    Label(  'sidewalk'             ,  8 ,  , (244, 35,232) ),
    Label(  'parking'              ,  9 ,  , (250,170,160) ),
    Label(  'rail track'           , 10 ,  , (230,150,140) ),
    Label(  'building'             , 11 ,
```

APPROACH

- First a distance array and category array is initialized. Category is our output and distance stores distances of each pixel with their category (we have to minimize this). Initially distance should be infinite.
- For each item in Id2Color, I find the distance of every pixel in mask with label pixels and store in an array dist.
- Then I need to find values in distance which are larger, these can be replaced with smaller values,i.e., dist. This can be done using boolean masking and using np.where().

```
def FindLabels(mask, mapping):
    distance = np.full([mask.shape[0], mask.shape[1]], 99999)
    category = np.full([mask.shape[0], mask.shape[1]], None)

    for id, color in mapping.items():
        dist = np.sqrt(np.linalg.norm(mask - color.reshape([1,1,-1]), axis=-1))
        condition = distance > dist
        distance = np.where(condition, dist, distance)
        category = np.where(condition, id, category)

    return category
```

APPROACH

- Non-masked images are stored in array X.
- Y stores the mask.
- Y_mask_enc stores the encoded masks (stores the classes of each pixel value)

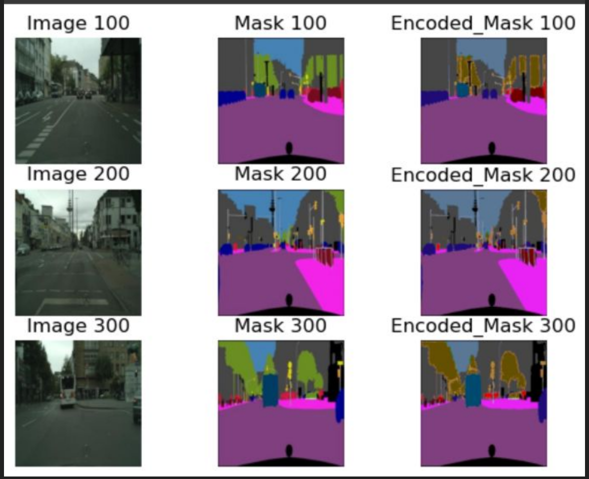
```python
def pre_process_images(num_images,path):
    X = np.empty((num_images ,SIZE[0] ,SIZE[1] ,SIZE[2]), dtype=np.uint8)
    Y_mask = np.empty((num_images ,SIZE[0] ,SIZE[1] ,SIZE[2]), dtype=np.uint8)
    Y_mask_enc = np.empty((num_images ,SIZE[0] ,SIZE[1]), dtype=np.uint8)

    for i, image in enumerate(os.listdir(path)):
        img = cv2.imread(os.path.join(path,image))

        X[i] = img[:,:256,:]
        Y_mask[i] = img[:,256:,:]
        Y_mask_enc[i] = FindLabels(Y_mask[i] , Id2Color)

    print(X.shape)
    print(Y_mask.shape)
    print(Y_mask_enc.shape)

    return X,Y_mask,Y_mask_enc
```

APPROACH

- The function below converts the mask with labels back to images by using Id2Color to map back label_id to its color.

```
def color_enc_mask(mask_enc, mapping):
    color_enc = np.zeros(SIZE)
    for i in range(SIZE[0]):
        for j in range(SIZE[1]):
            color_enc[i,j,:] = mapping[mask_enc[i,j]]
            color_enc = color_enc.astype('uint8')
    return color_enc
```

- Finally we get something like this



U-NET MODEL

- Downsampling Block

```
def down_sampling_block(prev_layer,filters,maxPool=True,drop_prob=0):
    conv = Conv2D(filters=filters,
                  kernel_size=KERNEL,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(prev_layer)

    conv = Conv2D(filters=filters,
                  kernel_size=KERNEL,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conv)
    if drop_prob > 0:
        conv = Dropout(drop_prob)(conv)

    if maxPool:
        next_layer = MaxPooling2D(pool_size=(2,2))(conv)
    else:
        next_layer = conv

    skip_connection = conv

    return next_layer,skip_connection
```

U-NET MODEL

- Up Sampling Block

```
def up_sampling_block(skip_connection,prev_layer,filters):
    upper = Conv2DTranspose(filters=filters,
                            kernel_size=KERNEL,
                            strides=(2,2),
                            padding='same')(prev_layer)
    conc = concatenate([upper,skip_connection],axis=3)

    conv = Conv2D(filters=filters,
                  kernel_size=KERNEL,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conc)
    conv = Conv2D(filters=filters,
                  kernel_size=KERNEL,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conv)
    return conv
```

U-NET MODEL

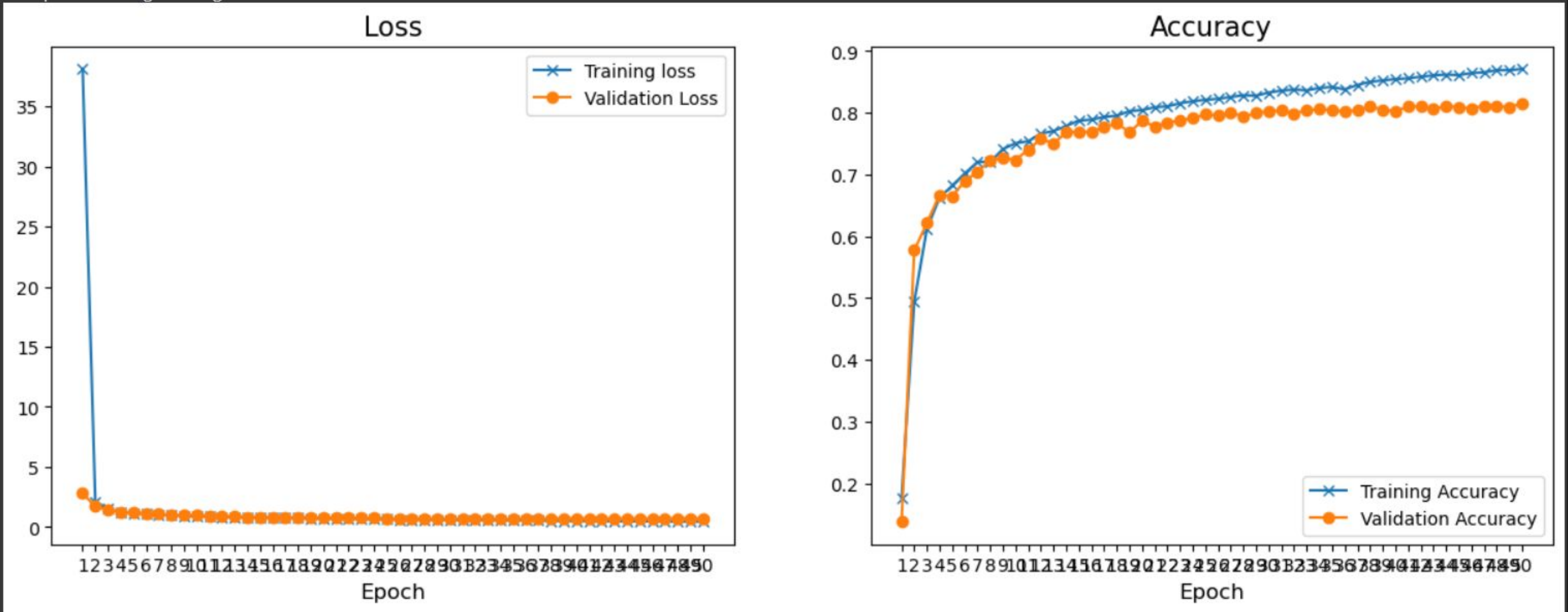
- The final U-NET MODEL

```
def U_Net_Model(filters,size,classes=N_CLASSES):
    #Input layer
    d0 = Input(size)

    #Downsampling Region
    d1, d1_skip = down_sampling_block(d0,filters)
    d2, d2_skip = down_sampling_block(d1,filters*2)
    d3, d3_skip = down_sampling_block(d2,filters*4)
    d4, d4_skip = down_sampling_block(d3,filters*8,drop_prob = 0.1)
    d5, d5_skip = down_sampling_block(d4,filters*16,drop_prob = 0.3)

    #Bottleneck
    b0, _ = down_sampling_block(d5,filters*32,maxPool = False,drop_prob = 0.3)

    #Upsampling Region
    u5 = up_sampling_block(d5_skip,b0,filters*16)
    u4 = up_sampling_block(d4_skip,u5,filters*8)
    u3 = up_sampling_block(d3_skip,u4,filters*4)
    u2 = up_sampling_block(d2_skip,u3,filters*2)
    u1 = up_sampling_block(d1_skip,u2,filters)

    #Output Layer
    u0 = Conv2D(filters=filters,
                kernel_size=KERNEL,
                activation='relu',
                padding='same',
                kernel_initializer='he_normal')(u1)
    u0 = Conv2D(classes,kernel_size=1 , padding='same')(u0)

    model = tf.keras.Model(inputs=d0, outputs=u0)

    return model
```

U-NET MODEL

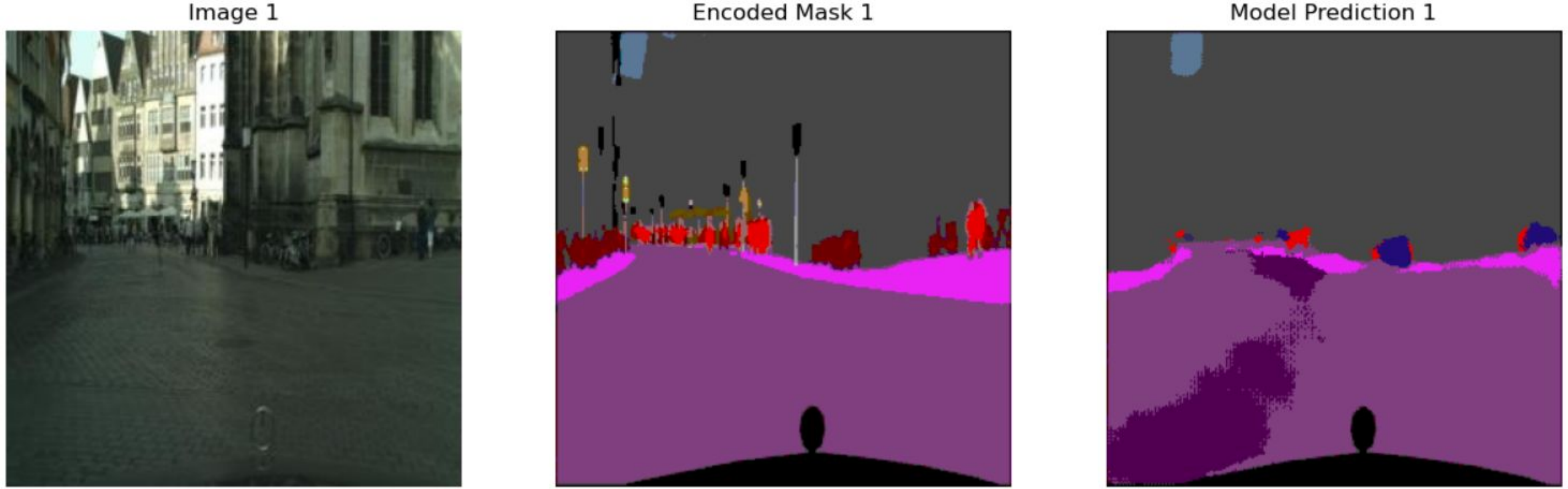
- OPTIMIZER: Adam optimizer for training the model. Adam is a popular gradient-based optimization algorithm.
- LOSS: The loss function we are using is **SparseCategoricalCrossentropy**, which is typically used for multi-class classification problems where the target values are integers representing class labels.
- Setting **from_logits=True** implies that the model's output is considered as logits, and the loss function will apply a softmax operation internally before computing the loss.
- METRICS: You are monitoring the accuracy metric during training. The model will compute and display accuracy as a training metric.

```
UNet.compile(optimizer='adam',
             loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
             metrics=['accuracy'])

model_checkpoint = ModelCheckpoint(CHECKPOINT_PATH,
                                   monitor='val_accuracy',
                                   save_best_only=True,
                                   save_weights_only=True,
                                   verbose=1,
                                   mode = 'max')
```



Loss
Training loss
Validation Loss
Epoch
Accuracy
Training Accuracy
Validation Accuracy
Epoch

RESULTS



RESULTS