SATELLITE IMAGERY

USING U-NET

CONTENT

DATA COLLECTION

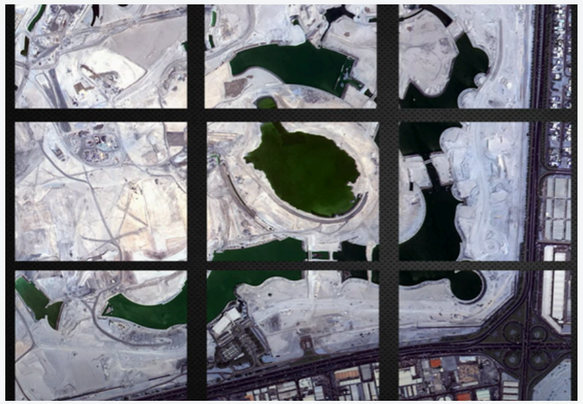
Dataset - Semantic segmentation of aerial imagery

The dataset consists of aerial imagery of Dubai obtained by MBRSC satellites and annotated with pixel-wise semantic segmentation in 6 classes. The total volume of the dataset is 72 images grouped into 6 larger tiles. The classes are:

1. **Building: #3C1098**
2. **Land (unpaved area): #8429F6**
3. **Road: #6EC1E4**
4. **Vegetation: #FEDD3A**
5. **Water: #E2A929**
6. **Unlabeled: #9B9B9B**

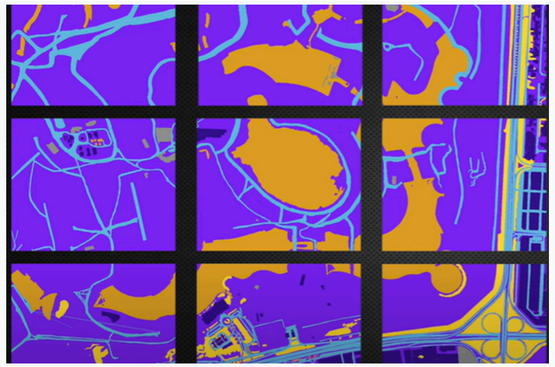
DATA PRE-PROCESSING

Tile



- Each satellite tile has its mask image which has colour labels for landmarks.
- Each satellite tile is further divided into 2x2, 3x3 or 4x4 images and in sometimes 1x3 or 4x5 images.
- Each satellite tile mask stay same with the size of further smaller size of tile images.

Mask



Tile and Mask processing

- If **image_type** is '**masks**', the loaded image is converted from BGR to RGB
- Decide the patch size i.e. 256x256 or 512x512.
- Split all the images into patch size and then convert it into **numpy array**.
- The **patchify** function to extract patches from an image.

Normalizing Images

- **Accessing Patched Image**: individual_patched_image = patched_images[i, j, :, :]: This line retrieves a specific patch (i, j) from the patched_images array. This patch represents a part of the original image.
- **Reshaping for Min-Max Scaling**: individual_patched_image.reshape(-1, individual_patched_image.shape[-1]): The reshape function is used to flatten the patch into a 2D array, where each row corresponds to a pixel and each column corresponds to a color channel (assuming RGB with individual_patched_image.shape[-1] being 3).
- **Applying Min-Max Scaling**: minmaxscaler.fit_transform(...): The fit_transform method of minmaxscaler is applied to the flattened patch. This function scales each feature (pixel value) to a specified range (usually [0, 1]) using the minimum and maximum values encountered in the data.
- **Reshaping Back to Image Patch**: reshape(individual_patched_image.shape): After scaling, the flattened array is reshaped back into the original patch shape ((image_patch_size, image_patch_size, 3)), assuming RGB color channels.

- For 'images', each patched image is processed by a **minmaxscaler** to normalize pixel values and then appended to image_dataset.
- For 'masks', patched masks are directly appended to mask_dataset.

MASK LABEL PROCESSING

The Python function **rgb_to_label**(label) converts an RGB image into a labeled segmentation mask based on predefined color classes (class_water, class_land, class_road, class_building, class_vegetation, class_unlabeled).

label_segment is initialized as a NumPy array of zeros with the same shape as the input label. The dtype=np.uint8 specifies that the array will contain unsigned 8-bit integers (values ranging from 0 to 255).

CREATING AN OUTPUT ARRAY

The function assigns integer labels (0, 1, 2, 3, 4, 5) to different regions in the input RGB image based on the predefined color classes

ASSIGNING LABELS BASED ON RGB VALUES

After assigning labels based on RGB values, the function converts label_segment to a grayscale image by selecting only the first channel ([:, :, 0]) of the array. This results in a 2D array of integers representing different label values.

CONVERTING TO GRAYSCALE

IMPLEMENTING THE DEEP LEARNING MODEL

1) Train Test Split

```
master_trianing_dataset = image_dataset

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(master_trianing_dataset, labels_categorical_dataset, test_size=0.15, random_state=100)
```

The train_test_split function from scikit-learn is used for this purpose. The split is done randomly, with **85%** of the data reserved for training (X_train and y_train), and 15% reserved for testing (X_test and y_test).
The random_state parameter ensures reproducibility by fixing the random seed.

2) U NET MODEL

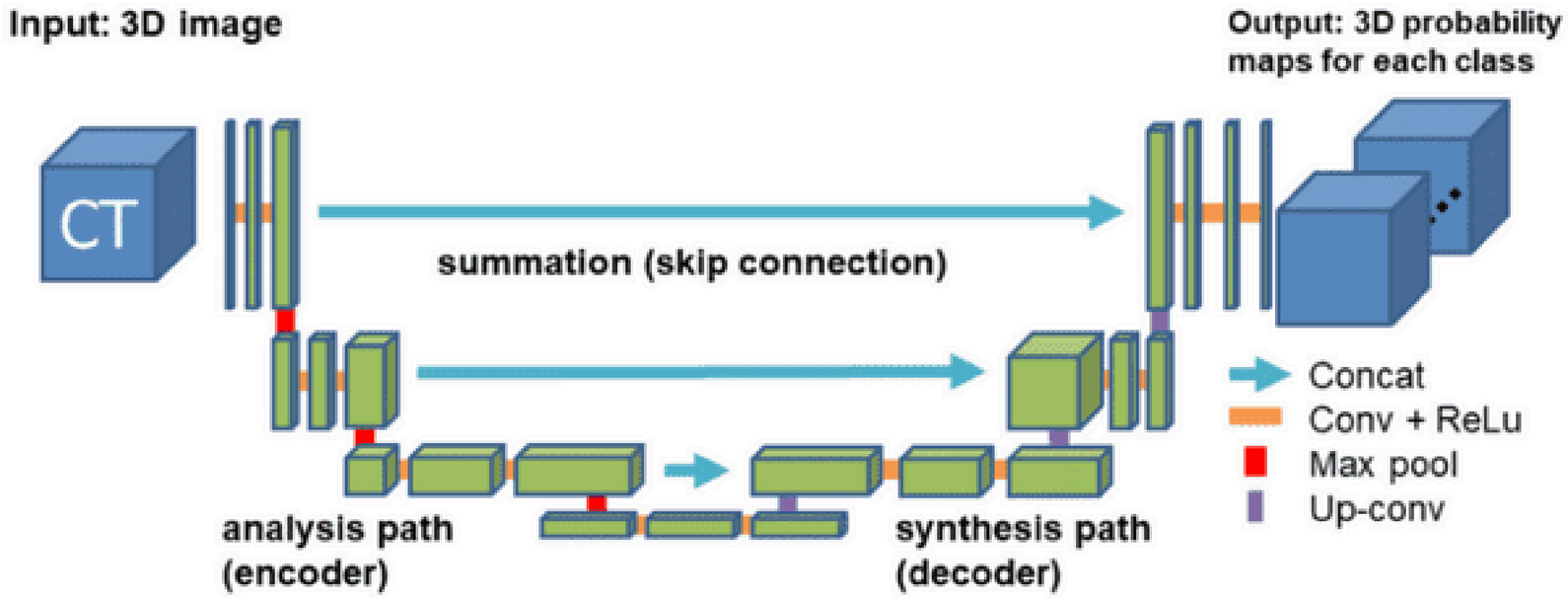
U-Net is a **convolutional neural network** architecture designed for biomedical image segmentation tasks, particularly in the field of medical image analysis. ."
The U-Net architecture is characterized by its symmetric encoder-decoder structure with skip connections.

1. **Encoder (Contracting Path)**
The encoder consists of a series of convolutional and max-pooling layers. These layers progressively reduce the spatial dimensions of the input image while increasing the number of feature channels. This process helps extract hierarchical features from the input image.

2. **Decoder (Expansive Path):**
The decoder is a symmetric counterpart to the encoder. It consists of a series of upsampling and convolutional layers. The upsampling layers gradually increase the spatial dimensions of the feature maps back to the original image size. Additionally, skip connections are introduced in the decoder to concatenate feature maps from the corresponding layers in the encoder.

3. **Skip Connections:**
Skip connections are connections that skip one or more layers in the network. In U-Net, skip connections concatenate feature maps from the encoder directly with the corresponding feature maps in the decoder. This allows the decoder to access both low-level and high-level features, facilitating more accurate segmentation results.

4. **Output Layer:**
The output layer of U-Net typically consists of a convolutional layer with softmax activation, which produces pixel-wise classification probabilities for each class.



```python
def multi_unet_model(n_classes=5, image_height=256, image_width=256, image_channels=1):

  inputs = Input((image_height, image_width, image_channels))

  source_input = inputs

  c1 = Conv2D(16, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(source_input)
  c1 = Dropout(0.2)(c1)
  c1 = Conv2D(16, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(c1)
  p1 = MaxPooling2D((2,2))(c1)

  c2 = Conv2D(32, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(p1)
  c2 = Dropout(0.2)(c2)
  c2 = Conv2D(32, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(c2)
  p2 = MaxPooling2D((2,2))(c2)

  c3 = Conv2D(64, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(p2)
  c3 = Dropout(0.2)(c3)
  c3 = Conv2D(64, (3,3), activation="relu", kernel_initializer="he_normal", padding="same")(c3)
  p3 = MaxPooling2D((2,2))(c3)
```

This function, multi_unet_model, creates a U-Net convolutional neural network architecture for multi-class semantic segmentation.

3) Generating Loss Function

dice loss > Focal Loss > Total Loss

Total Loss = (Dice loss + (1*Focal Loss))

```
dice_loss = sm.losses.DiceLoss(class_weights = weights)

focal_loss = sm.losses.CategoricalFocalLoss()

total_loss = dice_loss + (1 * focal_loss)
```

1) **Dice Loss:** Dice loss is a similarity coefficient-based loss function commonly used for segmentation tasks. It measures the overlap between the predicted segmentation and the ground truth. The Dice coefficient is defined as

2×intersection / total number of pixels in both masks

where the intersection is the number of overlapping pixels between the predicted and ground truth masks.
The Dice loss is computed as 1-Dice coefficient.

2) **Focal Loss:** Focal loss is a loss function designed to address class imbalance in classification tasks. It assigns higher weights to hard-to-classify examples (those with low probability predictions) and lower weights to well-classified examples. This helps in focusing training on hard examples, leading to improved performance, especially when dealing with heavily imbalanced datasets.

4) MODEL COMPILATION

```
model.compile(optimizer="adam", loss=total_loss, metrics=metrics)
```

The model is complied using the Adam optimizer, which is a popular choice for training neural networks. It's an adaptive learning rate optimization algorithm that combines ideas from RMSprop and momentum. Adam adjusts the learning rates for each parameter individually, allowing for faster convergence and better performance.

```
model_history = model.fit(X_train, y_train,
                          batch_size=16,
                          verbose=1,
                          epochs=100,
                          validation_data=(X_test, y_test),
                          shuffle=False)
```

MODEL VISUALIZATION WITH KERAS

- Visualizing a Keras model can be done using plot_model function from tensorflow.keras.utils
- The plot_model function will generate a PNG image file named "satellite_model_plot.png" that visualizes the model architecture, including the shapes of input and output tensors and the names of the layers.

```
[ ]  from keras.utils.vis_utils import plot_model
```

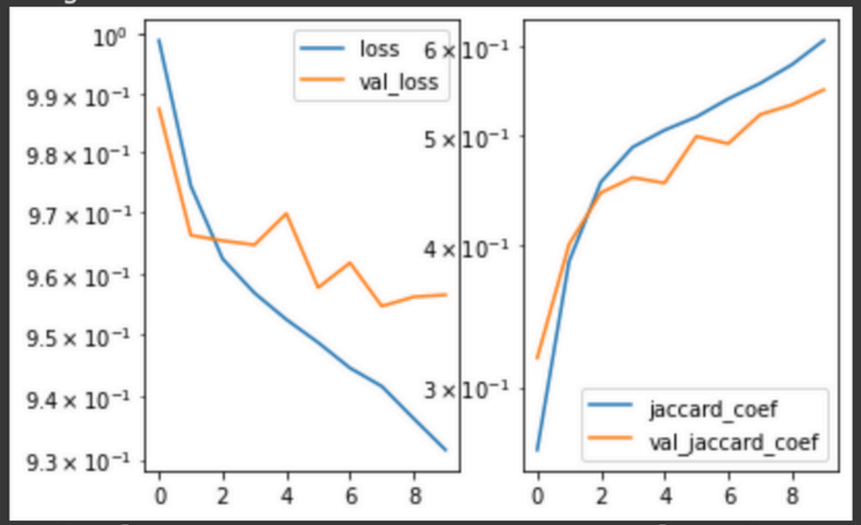
```
import keras
from IPython.display import clear_output

%matplotlib inline
```

```
plot_model(model, to_file="satellite_model_plot.png", show_shapes=True, show_layer_names=True)
```

Now the **PlotLoss** callback class is defined to plot the loss after each epoch during training. It plots the loss values stored in the **self.losses** list. Then, an instance of this callback is created (**plot_loss**) and passed to the **fit()** function as a part of the **callbacks** list.

```
model_history = model.fit(X_train, y_train,
                          batch_size=16,
                          verbose=1,
                          epochs=10,
                          validation_data=(X_test, y_test),
                          callbacks=[plot_loss],
                          shuffle=False)
```

```
<Figure size 1008x576 with 0 Axes>
```




```
51/51 [==============================] - 10s 191ms/step - loss: 0.9315 - accuracy
```

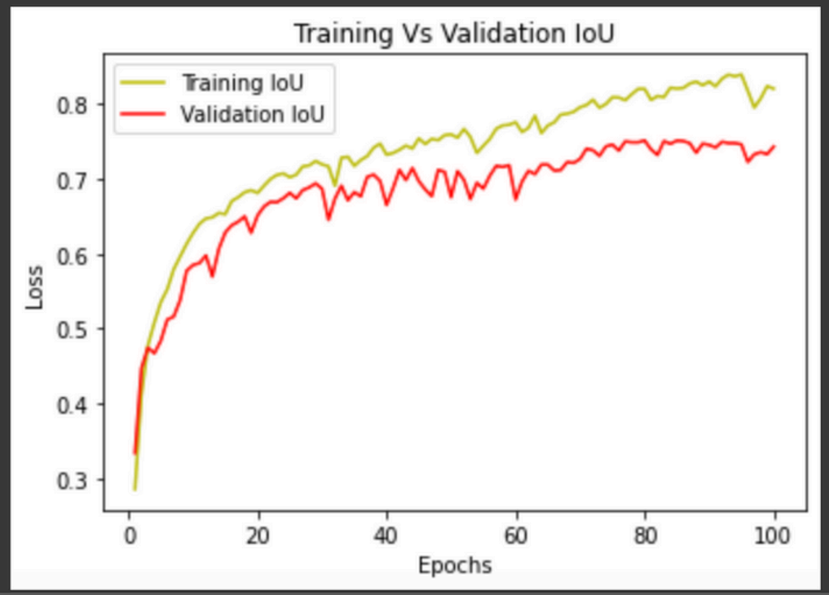
Now we will plot the training and validation loss from a history object (history_a). .

This code will plot the training loss (in yellow) and the validation loss (in red) across epochs. History_a contains the necessary history data from the model training process.

```
loss = history_a.history['loss']
val_loss = history_a.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y', label="Training Loss")
plt.plot(epochs, val_loss, 'r', label="Validation Loss")
plt.title("Training Vs Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

Now plot the training and validation Jaccard coefficient (IoU) from a history object (**history_a**) This code will plot the training IoU (Jaccard coefficient) in yellow and the validation IoU in red across epochs. **history_a** contains the necessary history data including 'jaccard_coef' and 'val_jaccard_coef'.

```
jaccard_coef = history_a.history['jaccard_coef']
val_jaccard_coef = history_a.history['val_jaccard_coef']

epochs = range(1, len(jaccard_coef) + 1)
plt.plot(epochs, jaccard_coef, 'y', label="Training IoU")
plt.plot(epochs, val_jaccard_coef, 'r', label="Validation IoU")
plt.title("Training Vs Validation IoU")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



COMPARING PREDICTION RESULTS

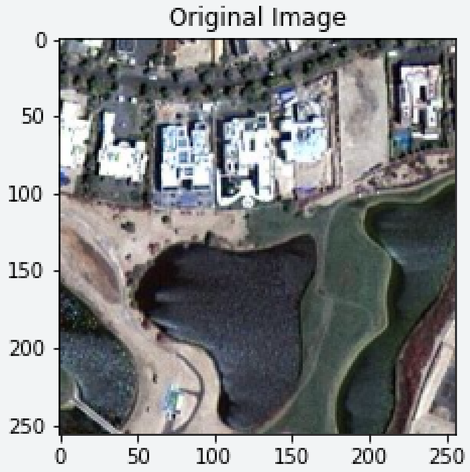
```
[ ]  import random
```

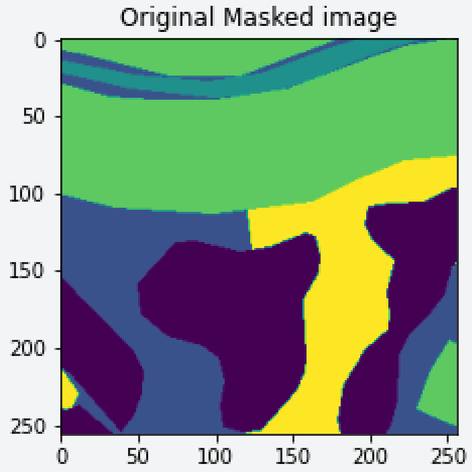
```
[ ]  test_image_number = random.randint(0, len(X_test))

     test_image = X_test[test_image_number]
     ground_truth_image = y_test_argmax[test_image_number]

     test_image_input = np.expand_dims(test_image, 0)

     prediction = model.predict(test_image_input)
     predicted_image = np.argmax(prediction, axis=3)
     predicted_image = predicted_image[0,:,:]
```

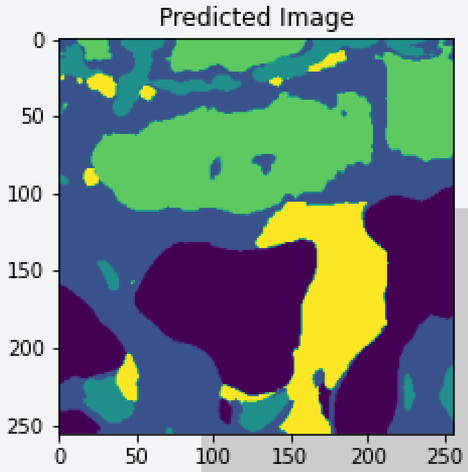
```
1/1 [==============================] - 0s 19ms/step
```

```
plt.figure(figsize=(14,8))
plt.subplot(231)
plt.title("Original Image")
plt.imshow(test_image)
plt.subplot(232)
plt.title("Original Masked image")
plt.imshow(ground_truth_image)
plt.subplot(233)
plt.title("Predicted Image")
plt.imshow(predicted_image)
```

This code will create a subplot with three images horizontally arranged: the original image, the original masked image, and the predicted image.







CONCLUSION

Key Findings:

1. **Identification of Objects:** Successfully identified various objects within satellite images, including buildings, roads, vegetation, and water bodies.
2. **Optimization Techniques:** Implemented advanced optimization techniques to enhance the model's performance, including data augmentation, transfer learning, and fine-tuning.
3. **Challenges Overcome:** Overcame challenges such as data variability, limited labeled data availability, and computational resource constraints.

Future Directions:

1. **Continuous Improvement:** Plan to further enhance the model's accuracy and robustness through ongoing research and development efforts.
2. **Expansion of Applications:** Explore additional applications and domains where satellite image detection can provide valuable insights and support decision-making processes.
3. **Collaborative Initiatives:** Seek opportunities for collaboration with domain experts, governmental organizations, and research institutions to leverage satellite image detection for societal benefit.

THANK YOU