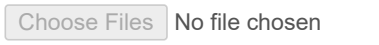
Satellite Image Land Cover Segmentation

Cameron Braatz - 12.05.2024

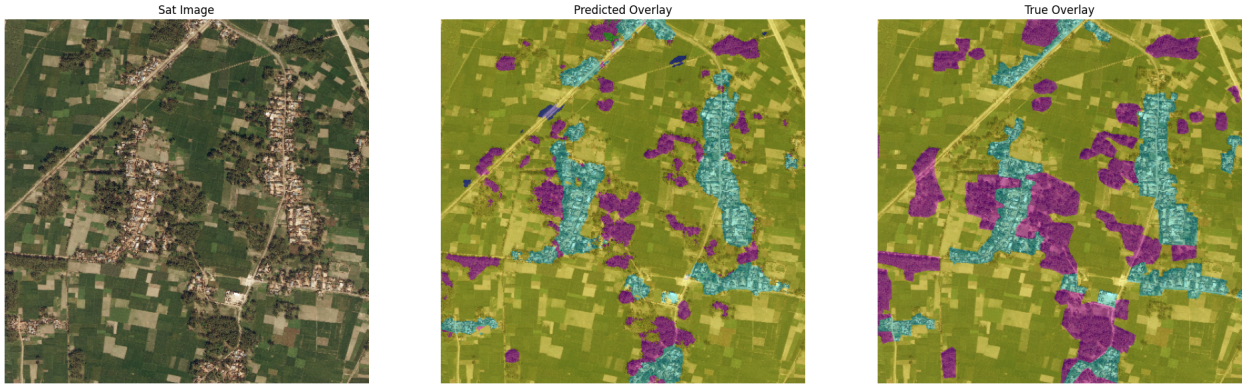
CU-Boulder - Machine Learning

In [71]:
```
from google.colab import files
from IPython.display import display
from IPython.display import Image as Im

files.upload()
display(Im('/content/prediction_thumbnail.png'))
```



Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving prediction_thumbnail.png to prediction_thumbnail (1).png
```



Overview

The following notebook prepares and trains a machine learning model to perform multi-class segmentation to predict land cover types from a set of satellite images. This dataset focuses on the detection of urban, agricultural, rangeland, forest, water, barrn and unknown land covers.

Traditionally, this task was accomplished through time-intensive manual inspection of aerial/satellite imagery. Experts performing this analysis were trained to identify and differentiate the 'shape, size, pattern, color, tone, texture, shadows, geographic context and association' of the land covers classes exhibited in the landscape (USGS).

Data Source

The dataset used for this project, DeepGlobe Land Cover Classification Dataset, was sourced from Kaggle. The dataset is comprised of:

- 803 training images and masks
- 171 validation images
- 172 test images
- `metadata.csv` image/mask index file
- `classes.csv` land cover class encoding

Outline

The basic structure of the following notebook is as follows:

- Exploratory Data Analysis
  - Accessing the Data
  - Splitting Train/Test Sets
  - Visual Inspection
- Image Preprocessing
  - Testing the `Image` class
  - Applying Augmentations
- Mask Preprocessing
  - Testing the `Mask` class
  - Visualizing Class Distribution
  - Applying Augmentations
- Data Preparation
  - Image/Mask Processing
  - Analyzing Training Distribution
  - Model Training
- Reflection

Setting up the Environment

The first section entails importing all of the libraries, downloading/unpacking the data remotely from Kaggle. This process ensures our runtime environment has what it needs to train and test our model.

In [2]:
```
# data manipulation...
import numpy as np
import pandas as pd

# visualization
import matplotlib.pyplot as plt
import seaborn as sns
import cv2 as cv
from IPython.display import display
from tqdm.keras import TqdmCallback
from PIL import Image as PILImage

# tensor flow keras/cnn...
from tensorflow.keras.models import Sequential
from tensorflow.keras import regularizers
from tensorflow.keras.layers import Conv2D, Dropout, Dense, AveragePooling2D, MaxPooling2D, Flatte

# vision transformers
from transformers.models.vit.feature_extraction_vit import ViTFeatureExtractor
from transformers import ViTForImageClassification, TrainingArguments, Trainer

# saving models and data
import joblib

# OS for navigation and environment set up
import os

# tensorflow
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

# splitting datasets
from sklearn.model_selection import train_test_split
```

```
# random
import random

# progress bar
from tqdm import tqdm

# model import

from keras.models import load_model

# warnings
import warnings
warnings.filterwarnings('ignore', category=DeprecationWarning)
```

In [3]:
```
files.upload() # kaggle.json...
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

Out[3]:
```
{'kaggle.json': b'{"username":"cambraatz","key":"30961088bd1792504b5acdc44a1b48e4"}'}
```

In [4]:
```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
#!kaggle datasets list
```

In [5]:
```
!kaggle datasets download -d balraj98/deepglobe-land-cover-classification-dataset
```

```
Dataset URL: https://www.kaggle.com/datasets/balraj98/deepglobe-land-cover-classification-dataset
License(s): other
Downloading deepglobe-land-cover-classification-dataset.zip to /content
100% 2.73G/2.74G [00:17<00:00, 164MB/s]
100% 2.74G/2.74G [00:17<00:00, 165MB/s]
```

In [76]:
```
!unzip deepglobe-land-cover-classification-dataset.zip > /dev/null
```

```
replace class_dict.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename: N
```

Exploratory Data Analysis (EDA)

Accessing the Data

The process below involves opening the two CSV files that provide insights into the data we will be exploring:

`metadata.csv` contains unique image ID's shared by a satellite image and mask image pair, as well as thier respective 'split' classification. The 'images' in our case are the relative file paths to the real image and mask files.

In [7]:
```
''' base_dir = "/kaggle/input/deepglobe-land-cover-classification-dataset" '''

# read in metadata and preview...
metadata = pd.read_csv("metadata.csv") #os.path.join(base_dir, "metadata.csv"))
display(metadata.sample(5))

# read in classes and preview...
classes = pd.read_csv("class_dict.csv") #os.path.join(base_dir, "class_dict.csv"))
display(classes)
```

	image_id	split	sat_image_path	mask_path
957	905310	valid	valid/905310_sat.jpg	NaN
581	733758	train	train/733758_sat.jpg	train/733758_mask.png
912	671274	valid	valid/671274_sat.jpg	NaN
240	354033	train	train/354033_sat.jpg	train/354033_mask.png
88	194156	train	train/194156_sat.jpg	train/194156_mask.png

	name	r	g	b
0	urban_land	0	255	255
1	agriculture_land	255	255	0
2	rangeland	255	0	255
3	forest_land	0	255	0
4	water	0	0	255
5	barren_land	255	255	255
6	unknown	0	0	0

`class_dict.csv` contains the mask color encoding, which encodes 7 unique land cover types and their corresponding color blocking. Meaning that each land cover type has a unique permutation of full-saturation (R,G,B) values, see below:

Land Cover	Color Name	RGB Value	Color Block
Urban	Cyan	rgb(0,255,255)	==========
Agricultural	Yellow	rgb(255,255,0)	==========
Range	Magenta	rgb(255,0,255)	==========
Forested	Green	rgb(0,255,0)	==========
Water	Blue	rgb(0,0,255)	==========
Barren	White	rgb(255,255,255)	==========

Data Format

We see above that we are dealing with satellite images and masks stored using the `.jpg` and `.png` image file types. During the preprocessing stage, these image files will be broken into pixel arrays to access the RGB tuples encoding each pixel.

Images are dimensioned 2448px by 2448px, when accessed as an array the satellite images will be an array sized 2448x2448x3. This added depth of three encodes the RGB values; which in the case of satellite imagery will range $0 < R, G, B < 255$. Recall that the mask files should be color blocked according to the RGB mapping seen in the classes dataframe above, ideally we should be seeing the range of RGB values for masking cells to be $R, G, B = 0 \; || \; R, G, B = 255$.

Splitting Train/Test sets

As prefaced above, the `metadata.csv` file contains predefined training and testing splits. Immediately of note, the testing data split *does not include masks*. Unfortunately this makes it difficult to test our model's performance.

We can keep this data, predict on it and assessment model performance by visual inspection; or we can split the training data into a training and validation set where we can explicitly test its accuracy.

In [8]:
```
# split metadata (ie: file paths) into training files...
train_data_paths = metadata[metadata['split'] == 'train'].drop(['split'], axis=1)
print("train_data_paths size: ", train_data_paths.shape)
display(train_data_paths.head())

# and testing files...
test_data_paths = metadata[metadata['split'] == 'test'].drop(['split'], axis=1)
print("test_data_paths size: ", test_data_paths.shape)
display(test_data_paths.head())
```

```
train_data_paths size:  (803, 3)
```

	image_id	sat_image_path	mask_path
0	100694	train/100694_sat.jpg	train/100694_mask.png
1	102122	train/102122_sat.jpg	train/102122_mask.png
2	10233	train/10233_sat.jpg	train/10233_mask.png
3	103665	train/103665_sat.jpg	train/103665_mask.png
4	103730	train/103730_sat.jpg	train/103730_mask.png

```
test_data_paths size:  (172, 3)
```

	image_id	sat_image_path	mask_path
974	100877	test/100877_sat.jpg	NaN
975	103215	test/103215_sat.jpg	NaN
976	103742	test/103742_sat.jpg	NaN
977	110224	test/110224_sat.jpg	NaN
978	112946	test/112946_sat.jpg	NaN

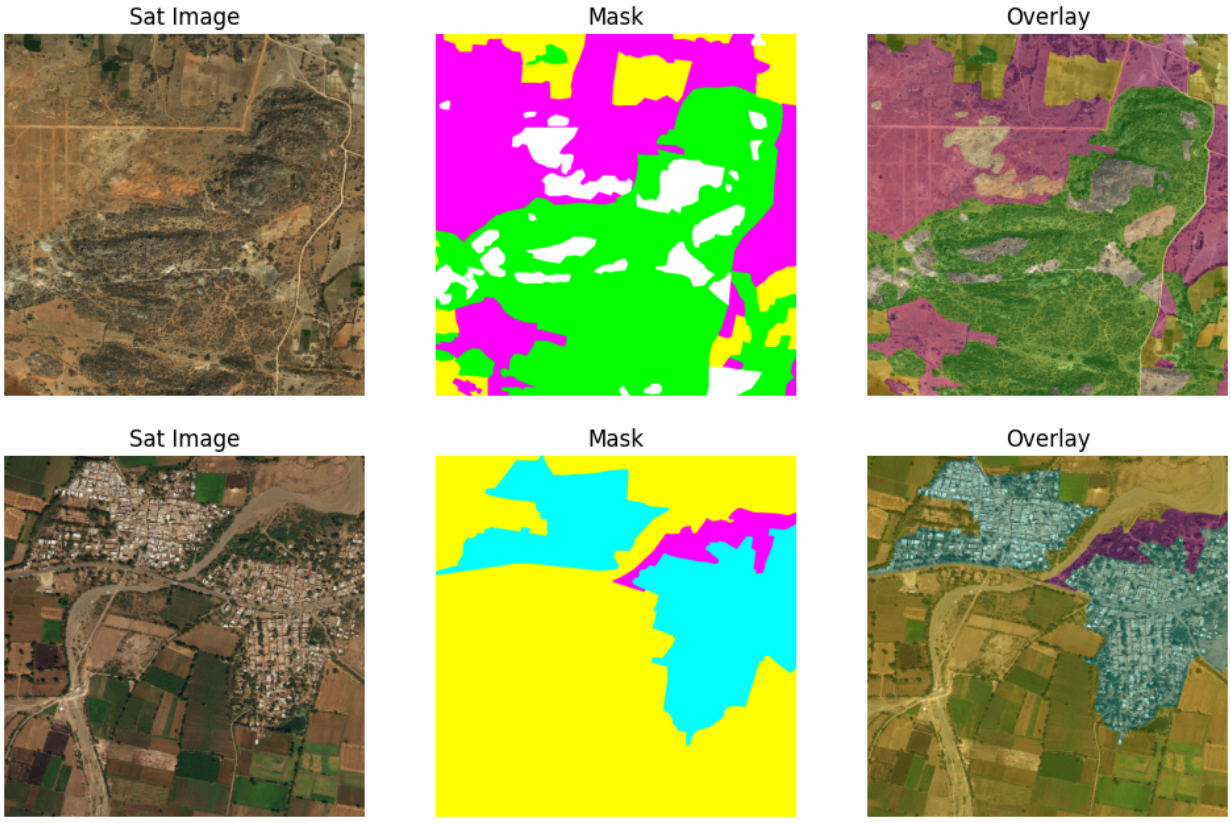
Visual Inspection

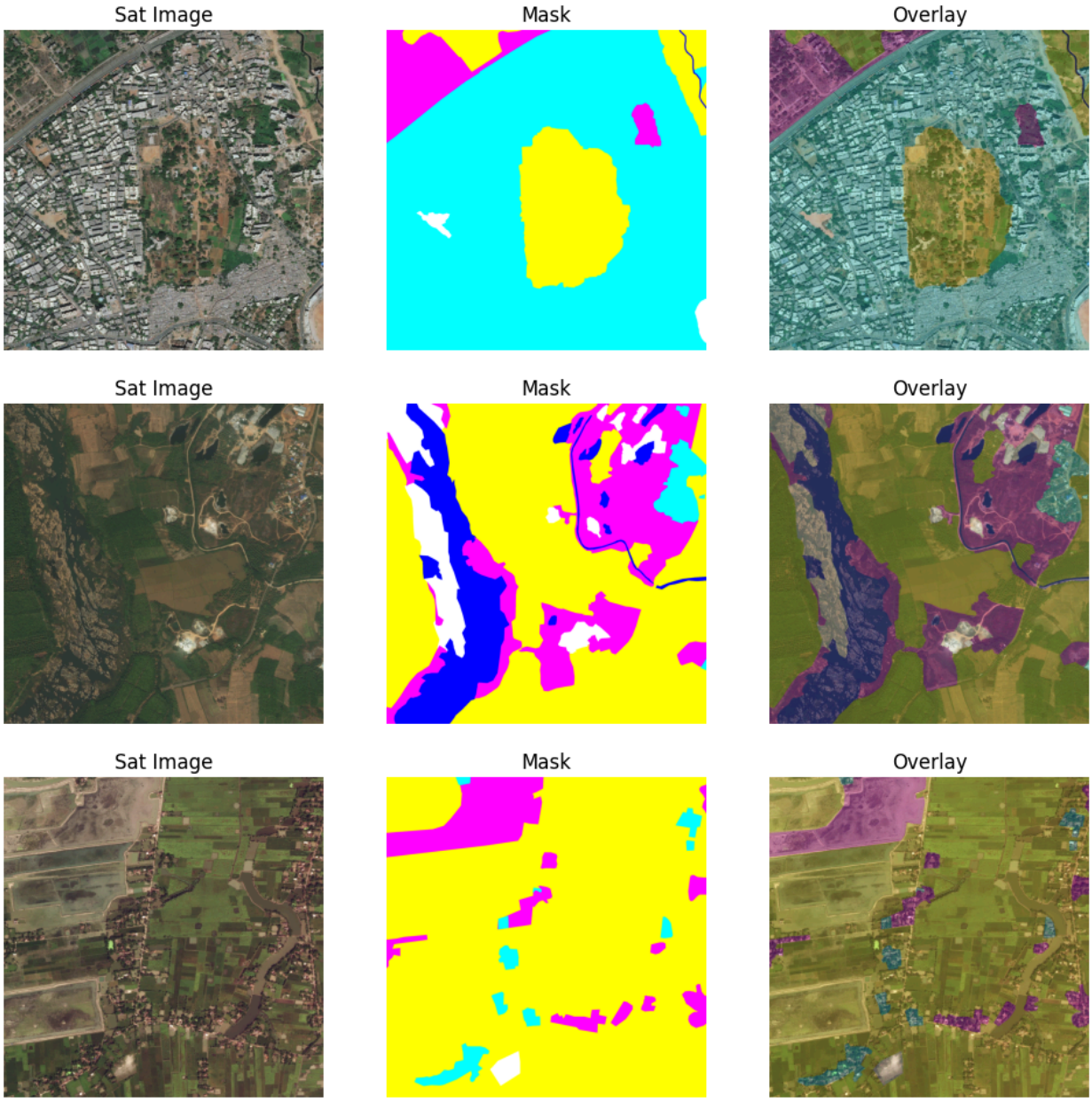
For now, we are only interested in getting more familiar with the type of training we will need to do. The plotting function below allows for direct visual comparison between the input images, the color blocked land cover mask and how they encode one another.

In [9]:
```
# helper to plot satellite, mask and overlay in sequence...
def plot_overlay(index_img=0, data_path=train_data_paths):
  # load the image and mask using our training data...
  img = plt.imread(data_path['sat_image_path'][index_img])
  mask = plt.imread(data_path['mask_path'][index_img])

  # create a figure and axes...
```

```
  fig, axs = plt.subplots(1,3, figsize=(12,10))

  # display the satellite image...
  axs[0].imshow(img)
  axs[0].set_title("Sat Image")
  axs[0].axis('off')

  # display the mask...
  axs[1].imshow(mask)
  axs[1].set_title("Mask")
  axs[1].axis('off')

  # display the mask over the sat image...
  axs[2].imshow(img)
  axs[2].imshow(mask, alpha=0.2)
  axs[2].set_title("Overlay")
  axs[2].axis('off')

  # show the plot...
  plt.show()

img = plt.imread(train_data_paths['sat_image_path'][0])
print(f"standard image/mask shape: {img.shape}")

cherry_picks = [161,348,482,423,305]

for i in cherry_picks:
  plot_overlay(index_img=i)
```

```
standard image/mask shape: (2448, 2448, 3)
```





It might be helpful to get acquainted with the land covers the model will aim to classify. Refer to these images to get familiar with the particular patterns, textures, colors, etc. exhibited by each class.

Please note that the images selected for visualization were hand-selected for their overwhelming proportion of each land cover (excluding *unknown*), and are not necessarily indicative of the average image samples. While the images below are outliers, they are quite indicative of the overall composition of the training images.

In [10]:
```
land_covers = {
    "urban_land": 79,
    "agriculture_land": 101,
    "rangeland": 169,
    "forest_land": 74,
    "water": 290,
    "barren_land": 115,
    "unknown (ie:cloud)": 329,
  }

for key, val in land_covers.items():
```

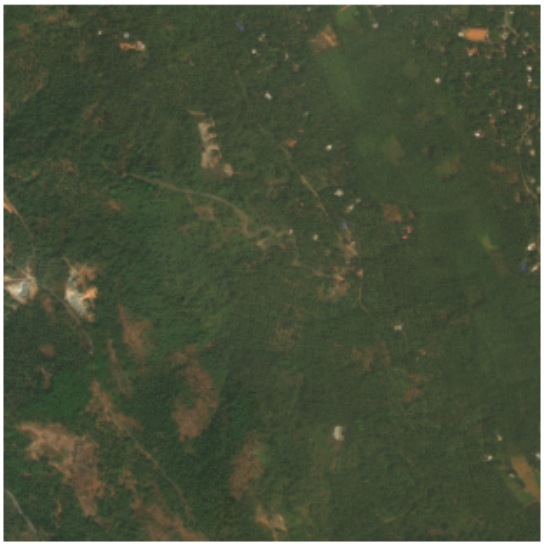
```
img = plt.imread(train_data_paths['sat_image_path'][val])

plt.figure(figsize=(4,4))
plt.imshow(img)
plt.title(key)
plt.axis('off')
plt.show()
```

urban_land

agriculture_land

rangeland

forest_land



water

barren_land

unknown (ie:cloud)

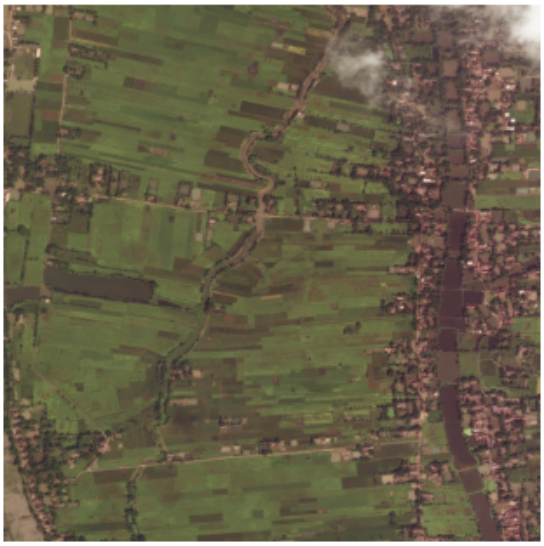


Image Pre-Processing

We are sticking to a object-oriented approach to managing our data resources and start by defining the `Image` class definition. Meant to generate a new object for each image processed, pertinent information is held contained within for easy access throughout the course of training, testing and visualization.

Our standard image processing functions include loading, resizing and plotting. Given our images are 2448px by 2448px it was in our best interest to downsize the images for the sake of our study, training our model on full size images would undoubtedly be computationally taxing. At least the first few iterations will work from images 25% of the original size as a *proof of concept*.

Additional class functions, `subdivide_image` and `reconstruct_image`, perform what is known as a tiling procedure. Used primarily with large images or small object detection, a high resolution image is broken into a grid of smaller images that are easier to process. These smaller image chunks are run through the image classifier and aggregated back to form a composite prediction.

In [11]:
```
# image class definition and helper functions...
class Image:
  # initialize class definition...
  def __init__(self, image_path, image_shape=(2448, 2448, 3)):
    self.image_path = image_path
    self.image = self._load_image(image_path)
    self.image_shape = image_shape
    self.tile_dim = int(image_shape[0] / 9)  # default tile dim is based on image shape
    self.tiles = None
    self.shape = self.image.shape

    # ensure TensorFlow uses GPU (if available)...
    self.device = '/GPU:0' if tf.config.list_physical_devices('GPU') else '/CPU:0'

  # load in our image...
  def _load_image(self,img_path):
    return plt.imread(img_path)

  # resize image using tf techniques...
  def resize_image(self, target_size=(512, 512)):
    with tf.device(self.device):
        image_tensor = tf.convert_to_tensor(self.image, dtype=tf.float32)
        resized_image = tf.image.resize(image_tensor, target_size)
        self.image = resized_image.numpy().astype(np.uint8)  # Convert back to NumPy for consisten
    self.image_shape = target_size

  # procedure to divide the image into tiles...
  def subdivide_image(self, tile_shape=None, show_image=False):
    tile_shape = tile_shape or (self.tile_dim, self.tile_dim)
    image_shape = self.image.shape
    n_rows = image_shape[0] // tile_shape[0]
    n_cols = image_shape[1] // tile_shape[1]

    sub_images = []
    if show_image:
      fig, ax = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=(6, 6))

    for i in range(n_rows):
      for j in range(n_cols):
        sub_image = self.image[i * tile_shape[0]:(i + 1) * tile_shape[0], j * tile_shape[1]:(j + 1
        sub_images.append(sub_image)
        if show_image:
            ax[i, j].imshow(sub_image)
            ax[i, j].axis('off')

    if show_image:
      fig.suptitle('Subdivided Images')
      plt.show()

    self.tiles = np.array(sub_images)
    return self.tiles

  # recompile the tiles into the original image format...
  def reconstruct_image(self, cropped_images=None, tile_shape=None, image_shape=None, show_image=F
    cropped_images = cropped_images or self.tiles
    tile_shape = tile_shape or (self.tile_dim, self.tile_dim)
    image_shape = image_shape or self.image_shape
    new_image = np.zeros(image_shape)
    n_rows = image_shape[0] // tile_shape[0]
    n_cols = image_shape[1] // tile_shape[1]

    for i in range(n_rows):
      for j in range(n_cols):
        sub_image = cropped_images[i * n_rows + j]
```

```
            new_image[i * tile_shape[0]:(i + 1) * tile_shape[0], j * tile_shape[1]:(j + 1) * tile_shap

        if show_image:
          plt.figure(figsize=(6, 6))
          plt.axis('off')
          plt.title('Re-Constructed Image')
          plt.imshow(new_image)
          plt.show()

        return new_image

      def augment_image(self, image, augmentations):
        for aug in augmentations:
            image = aug(image)  # Apply each augmentation
        self.image = image  # Update self.image with the augmented version
        return image  # Optional: Return augmented image if needed

      # plot the image for visual review...
      def plot_image(self, title):
        plt.figure(figsize=(6,6))
        plt.imshow(self.image)
        plt.title(title)
        plt.axis('off')
        plt.show()
```

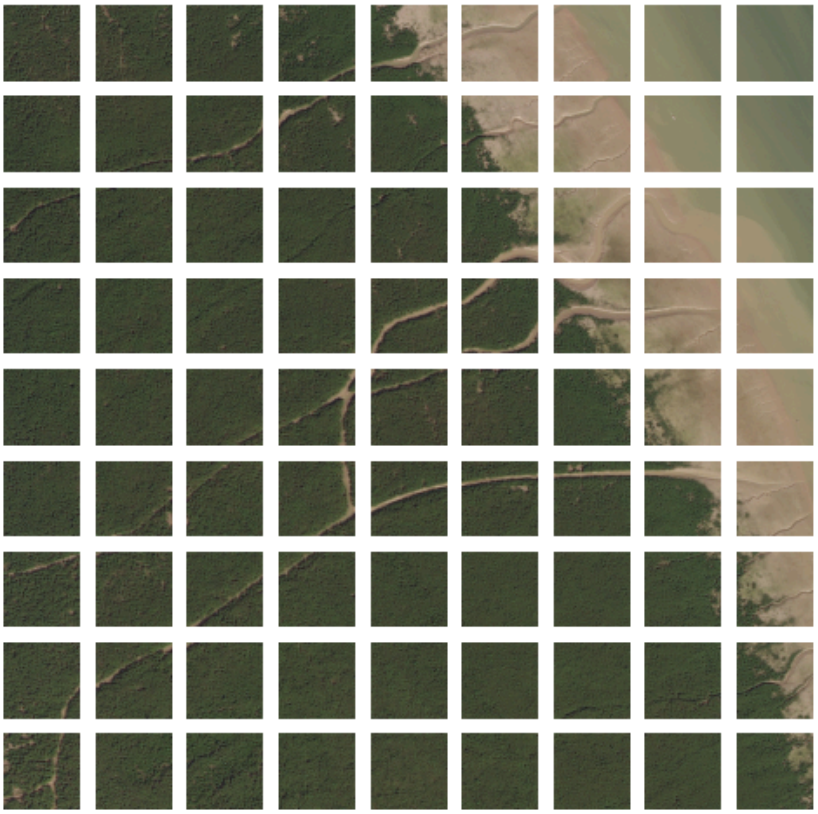
Testing the `Image` class

An arbitrary image is chosen to test our class definition to be sure it can successfully pre-process an image. Particularly we want to be sure we can split our image into a grid of sub-images, and then recompile it again.

In [12]:
```
# define global image dimensions, likely need to downsize for training...
IMAGE_SHAPE = (2448,2448,3)

# sample arbitrary training image and initialize as Image object...
sample_path = train_data_paths['sat_image_path'][771]
image_obj = Image(sample_path, image_shape=IMAGE_SHAPE)

# demonstrate image tiling...
image_obj.plot_image('Original Image')
tiles = image_obj.subdivide_image(show_image=True)
reconstructed_image = image_obj.reconstruct_image(show_image=True)
```

Original Image

Subdivided Images



Applying Augmentations

Out of the box TensorFlow image augmentations can also be applied to our images to increase the diversity of our training set.

It is important to note that whatever transformations applied to the satellite image, must also be performed on the mask image as well. Additionally, the mask files must be protected from augmentations like brightness and contrast changes that will impact the color mapping.

In [13]:
```
# define a set of custom augmentations...
augmentations = [
  lambda x: tf.image.random_flip_left_right(x),
  lambda x: tf.image.random_flip_up_down(x),
  lambda x: tf.image.rot90(x, k=np.random.randint(1, 4)),
  lambda x: tf.image.random_brightness(x, max_delta=0.2),
  lambda x: tf.image.random_contrast(x, lower=0.8, upper=1.2)
]

# apply and visualize augmented image...
image_obj.augment_image(image_obj.image, augmentations)
image_obj.plot_image('Augmented Image')
```

Ultimately I was able to tile my images, but the additional processing and aggregation proved to be logistically and computationally taxing with this data. While many agree that this approach is effective, I hypothesize that much of the land cover detail may still be perceived on a macro-level.

Instead, I opted to simplify and train the model with whole image processing on scaled down images, 512px x 512px, in lieu of processing tiled, full-size images at 2448px x 2448px dimensionality. This significantly speeds up pre-processing and training times, but also operations from here on out.

Mask Pre-Processing

Just as with the image class definition, we define the `Mask` object below to organize the data pertaining to individual land cover masks. It also has the standard image processing function like loading, resizing and plotting...however, it also holds the logic for one hot encoding the land cover masks so that our model can process them.

In order to correctly one hot encode the masks, we first need to normalize their color values. The data source indicates that not all pixel RGB values will be pure, ie: not exclusively 0 or 255. By normalizing the RGB array, we first ensure it's values match the intended range before binarizing with a threshold of 128 per the source's recommendation. Any red, green or blue values < 128 are assigned a 0 value and any values ≥ 128 are assigned 255.

In [14]:
```
# mask class definition and helper functions...
class Mask:
  # initialize class object...
  def __init__(self, mask_path, mask_shape=(2448, 2448, 3)):
    self.mask_path = mask_path
    self.class_colors = { # can this be removed to lighten it up???
        (0, 255, 255): "urban_land",
```

```
        (255, 255, 0): "agriculture_land",
        (255, 0, 255): "rangeland",
        (0, 255, 0): "forest_land",
        (0, 0, 255): "water",
        (255, 255, 255): "barren_land",
        (0, 0, 0): "unknown",
    }
    self.color_indices = { # can this be removed to lighten it up???
        (0, 255, 255): 0,  # urban_land
        (255, 255, 0): 1,  # agriculture_land
        (255, 0, 255): 2,  # rangeland
        (0, 255, 0): 3,    # forest_land
        (0, 0, 255): 4,    # water
        (255, 255, 255): 5, # barren_land
        (0, 0, 0): 6       # unknown
    }

    # mandatory processing...
    self.mask = self.load_mask(mask_path)
    self.normalized_mask = self.normalize_mask(self.mask)
    self.unique_colors = self.find_unique_colors()
    self.ohe = self.one_hot_encode()
    self.classes = self.map_colors_to_classes()

    # optional processing...
    self.class_counts = None
    self.shape = self.mask.shape
    self.mask_shape = mask_shape
    self.tile_dim = int(mask_shape[0] / 9)  # Default tile dim is based on image shape
    self.tiles = None

  # load the mask image file as np array...
  def load_mask(self,mask_path):
    return plt.imread(mask_path)

  # resize the mask to a given size...
  def resize_mask(self, target_size=(512, 512)):
    self.normalized_mask = cv.resize(self.normalized_mask, target_size, interpolation=cv.INTER_ARE
    self.ohe = self.one_hot_encode()

  # normalize the mask to range [0-255] and binarize to threshold of 128...
  def normalize_mask(self,mask):
    # ensure proper range...
    if mask.max() <= 1.0:
        mask = (mask * 255).astype(np.uint8)
    else:
        mask = mask.astype(np.uint8)

    # binarize the mask RGB values...
    mask[mask >= 128] = 255
    mask[mask < 128] = 0

    return mask

  # find unique colors from mask for validation...
  def find_unique_colors(self):
    return np.unique(self.normalized_mask.reshape(-1, 3), axis=0)

  # map the unique colors to their classes...
  def map_colors_to_classes(self):
    found_classes = {}
    for color in self.unique_colors:
        color_tuple = tuple(color)
        found_classes[color_tuple] = self.class_colors.get(color_tuple, "unknown")
```

```
    return found_classes

  # one hot encode each pixel to a color index...
  def one_hot_encode(self):
    # initialize blank mask array...
    height, width, _ = self.normalized_mask.shape
    index_mask = np.zeros((height, width), dtype=np.int32)

    # map one hot encodings into array for training...
    for color, idx in self.color_indices.items():
        match = np.all(self.normalized_mask == np.array(color), axis=-1)
        index_mask[match] = idx

    # attempt to bring more GPU optimization to this...
    index_mask_tensor = tf.convert_to_tensor(index_mask, dtype=tf.int32)
    one_hot_mask = tf.one_hot(index_mask_tensor, len(self.color_indices))

    # ensure output is still in numpy format...
    return one_hot_mask.numpy()

  # divide mask into tiles and store subimages...
  def subdivide_mask(self, tile_shape=None, show_mask=False):
    tile_shape = tile_shape or (self.tile_dim, self.tile_dim)
    mask_shape = self.mask.shape
    n_rows = mask_shape[0] // tile_shape[0]
    n_cols = mask_shape[1] // tile_shape[1]

    sub_masks = []
    if show_mask:
        fig, ax = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=(6, 6))

    for i in range(n_rows):
        for j in range(n_cols):
            sub_mask = self.mask[i * tile_shape[0]:(i + 1) * tile_shape[0], j * tile_shape[1]:(j +
            sub_masks.append(sub_mask)
            if show_mask:
                ax[i, j].imshow(sub_mask, cmap='gray')
                ax[i, j].axis('off')

    if show_mask:
        fig.suptitle('Subdivided Masks')
        plt.show()

    self.tiles = np.array(sub_masks)
    return self.tiles

  # reconstruct a mask from its tiles...
  def reconstruct_mask(self, cropped_masks=None, tile_shape=None, mask_shape=None, show_mask=False
    cropped_masks = cropped_masks or self.tiles
    tile_shape = tile_shape or (self.tile_dim, self.tile_dim)
    mask_shape = mask_shape or self.mask_shape
    new_mask = np.zeros(mask_shape)
    n_rows = mask_shape[0] // tile_shape[0]
    n_cols = mask_shape[1] // tile_shape[1]

    for i in range(n_rows):
      for j in range(n_cols):
        sub_mask = cropped_masks[i * n_rows + j]
        new_mask[i * tile_shape[0]:(i + 1) * tile_shape[0], j * tile_shape[1]:(j + 1) * tile_shape

    if show_mask:
      plt.figure(figsize=(6, 6))
      plt.axis('off')
      plt.title('Re-Constructed Mask')
```

```
      plt.imshow(new_mask, cmap='gray')
      plt.show()

    return new_mask

  # apply augmentations to the mask...
  def augment_mask(self, mask, augmentations):
    for aug in augmentations:
        mask = aug(mask)
    self.mask = mask
    return mask

  def plot_mask(self, title, overlay=None, size=(6,6)):
    plt.figure(figsize=size)
    if (overlay):
      img = plt.imread(overlay)
      plt.imshow(img)
      plt.imshow(self.mask, alpha=0.2)
    else:
      plt.imshow(self.mask)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

Testing the `Mask` class

Once defined, we can test the classes ability to process a single mask file. Just as with the images, a new `Mask` object will be generated for each mask file processed...making it a priority to keep the object lightweight.

The contained `.calculate_class_distribution` and `.plot_class_histogram` class functions were removed and converted to standalone functions (see below) to be called only during preliminary testing. Additionally, this slims down our `Mask`'s and protects them from excessive and unnecessary computation. See below for their logic and implementation.

Our preliminary testing shows that the class processes the land cover mask, normalizing the RGB colors into clean 0's and 255's to be one hot encoded. The results of this process can be seen below, where unique colors and classes were identified and a preview of the encoding are logged.

In [15]:
```
# iterate tensors to gather pixel counts for each class...
def calculate_class_distribution(mask, class_colors):
    class_counts = {name: 0 for name in class_colors.values()}
    normalized_mask_tensor = tf.convert_to_tensor(mask.normalized_mask, dtype=tf.uint8)

    for color, class_name in class_colors.items():
        color_tensor = tf.constant(color, dtype=tf.uint8)
        match = tf.reduce_all(tf.equal(normalized_mask_tensor, color_tensor), axis=-1)

        # Count occurrences on GPU
        count = tf.reduce_sum(tf.cast(match, tf.int32)).numpy()
        class_counts[class_name] += count

    return class_counts

# visualize the class histogram...
def plot_class_histogram(class_counts, class_colors):
    # prepare colors and labels for the bars...
    bar_colors = [np.array(color) / 255.0 for color in class_colors.keys()]
    bar_labels = list(class_colors.values())
```

```
    bar_values = [class_counts[name] for name in bar_labels]

    plt.bar(bar_labels, bar_values, color=bar_colors)
    plt.xlabel("Classes")
    plt.ylabel("Pixel Count")
    plt.title("Class Distribution in Mask")
    plt.xticks(rotation=60)
    plt.show()
```

In [20]:
```
# load in mask and create Mask object...
mask_path = train_data_paths['mask_path'][161]
mask = Mask(mask_path)

# demonstrate class functions...
print(f"Unique colors:\n{mask.find_unique_colors()}\n")
print(f"Unique classes:\n{mask.classes}\n")
print(f"One hot encoded mask (first row):\n{mask.ohe[0]}\n")
```

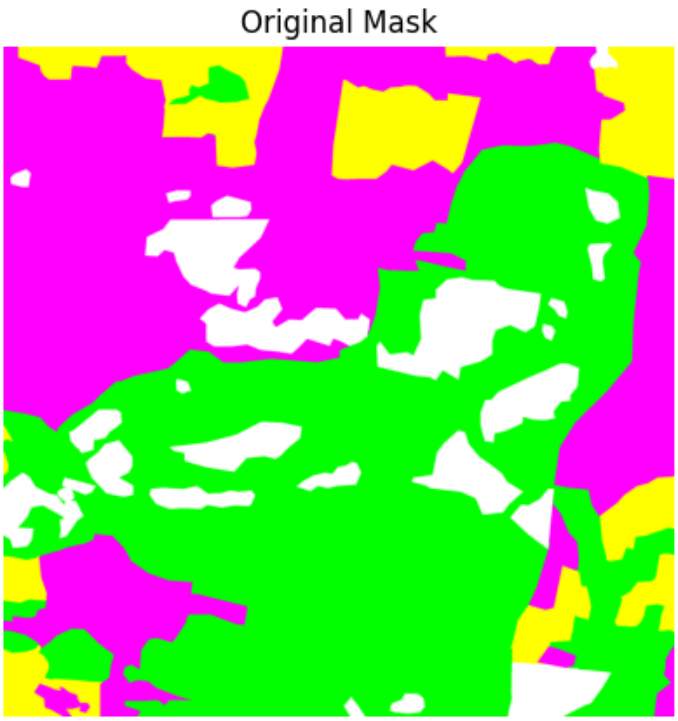
```
Unique colors:
[[  0 255   0]
 [255   0 255]
 [255 255   0]
 [255 255 255]]

Unique classes:
{(0, 255, 0): 'forest_land', (255, 0, 255): 'rangeland', (255, 255, 0): 'agriculture_land', (255, 2
55, 255): 'barren_land'}

One hot encoded mask (first row):
[[0. 0. 1. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 ...
 [0. 1. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]]
```

Visualizing Class Distribution

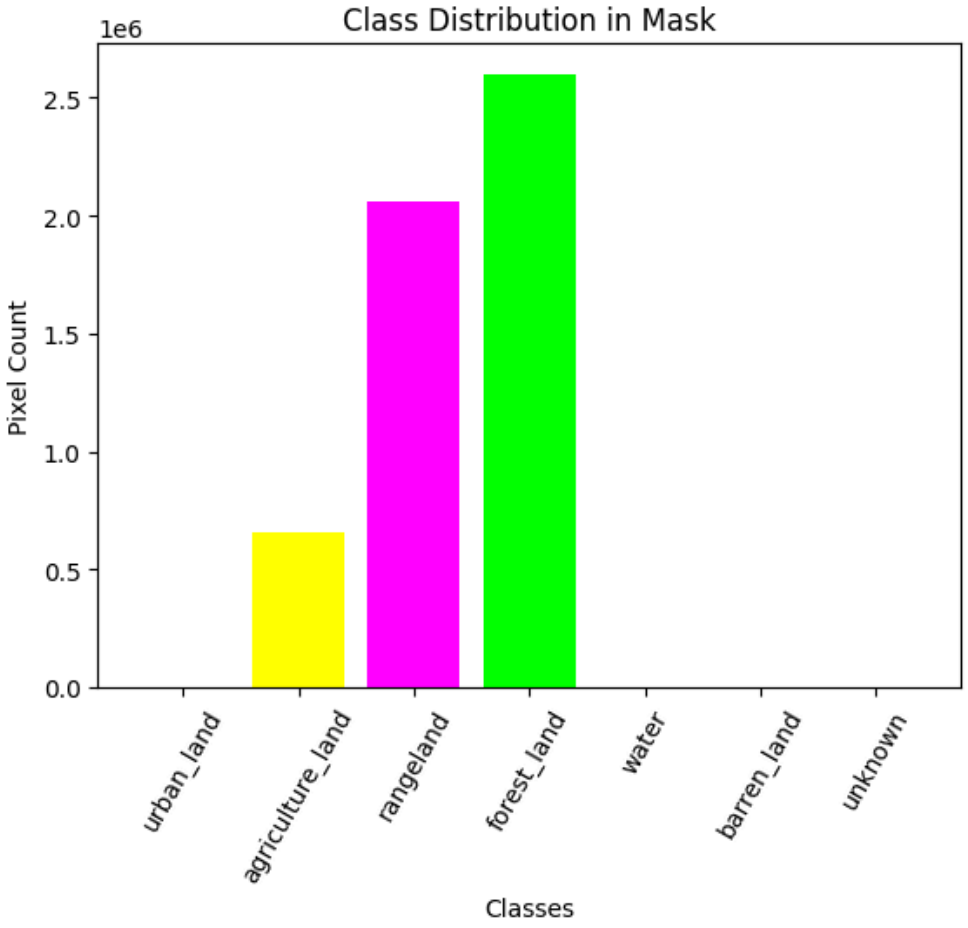
While the class appears to be working just fine, it is best to do a visual inspection to avoid getting lost in the abstraction. The aforementioned plotting functions, `.calculate_class_distribution` and `.plot_class_histogram`, are called to compare the original mask to the perceived mask.

Ensuring that the set of perceived colors matches those seen in the mask and the relative distribution of each color is a good first step. Before moving on, I also ensure that my one hot encoded class distribution matches the color class distribution.

In [21]:
```
# visualize sample class distribution...
mask.plot_mask('Original Mask',size=(5,5))

# derive the class counts for plotting...
class_counts = calculate_class_distribution(mask, mask.class_colors)
print(f"Class Distribution:\n{class_counts}\n")

plot_class_histogram(class_counts, mask.class_colors)
```



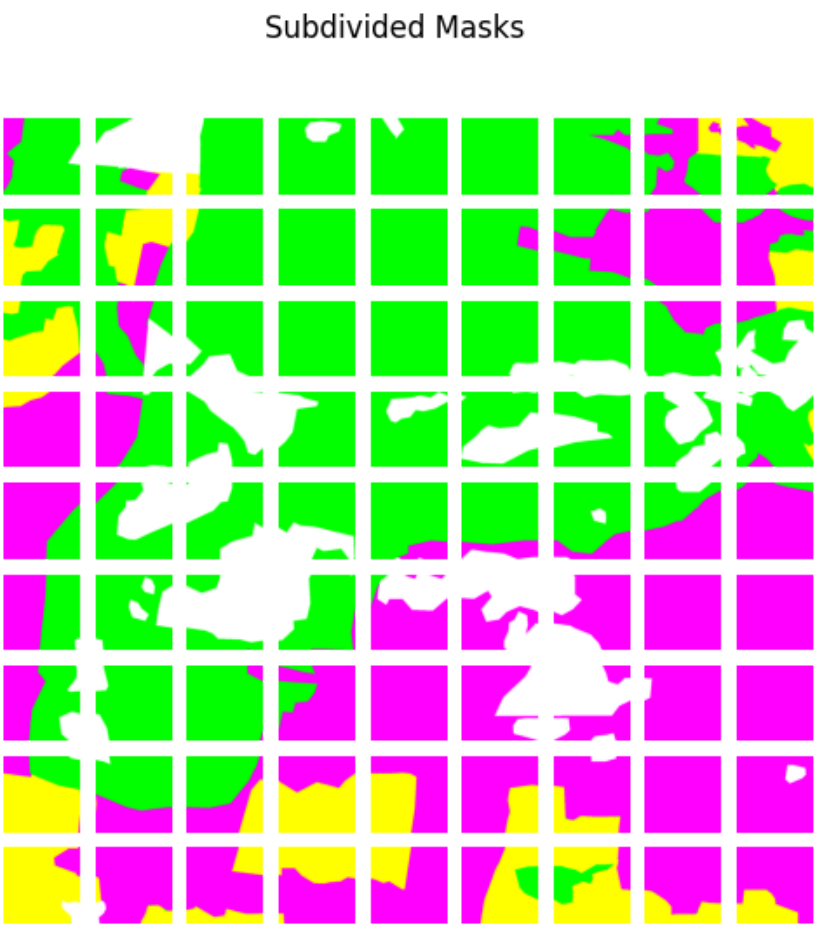

```
Class Distribution:
{'urban_land': 0, 'agriculture_land': 655965, 'rangeland': 2057871, 'forest_land': 2598728, 'wate
r': 0, 'barren_land': 680140, 'unknown': 0}
```



Applying Augmentations

Just as was done to the `Image` class, the mask images are also tested for their ability to be augmented. Flexibility in its implementation allows for easier model-tuning down the road.

In [22]:
```
# do not manipulate brightness/contrast...
augmentations = [
  lambda x: tf.image.random_flip_left_right(x),
  lambda x: tf.image.random_flip_up_down(x),
  lambda x: tf.image.rot90(x, k=np.random.randint(1, 4)),  # Random 90-degree rotations
  #lambda x: tf.image.random_brightness(x, max_delta=0.2),
  #lambda x: tf.image.random_contrast(x, lower=0.8, upper=1.2)
]

# apply augmentations...
mask.augment_mask(mask.mask, augmentations)
mask.plot_mask('Augmented Mask')

# demonstrate image tiling...
tiles = mask.subdivide_mask(show_mask=True)
reconstructed_mask = mask.reconstruct_mask(show_mask=True)
```


Subdivided Masks


Re-Constructed Mask

With our classes, `Image` and `Mask`, defined and individually tested, we are ready to move on to macro-data processing.

Data Preparation

The code below references the initial splitting of our raw `metadata` into the predefined train/test sets. As eluded to then, this test data contains *unlabeled* satellite images (ie: no masks).

```
# split metadata (ie: file paths) into training files...
train_data_paths = metadata[metadata['split'] == 'train'].drop(['split'], axis=1)
print("train_data_paths size: ", train_data_paths.shape) # 803
display(train_data_paths.head())

# and testing files...
test_data_paths = metadata[metadata['split'] == 'test'].drop(['split'], axis=1)
print("test_data_paths size: ", test_data_paths.shape) # 172
display(test_data_paths.head())
```

Given there is no way to evaluate the model's performance with this data, we will be splitting our training dataset into a training and evaluation set with an 80/20 split. A cursory look at the training data verifies we are dealing with a list of file paths for training satellite images, matching in length to the original predefined train/test sets.

The four datasets; `X_train, X_test, y_train, y_test`, are converted to PANDAS dataframes for quick manipulation before they are assessed to be consistently shaped.

In [23]:
```
# isolate the training image/mask paths...
train_image_paths = train_data_paths['sat_image_path']
train_mask_paths = train_data_paths['mask_path']
print(f"Training set: {len(train_image_paths)} images, {len(train_mask_paths)} masks")

# isolate the testing image/mask paths...
val_image_paths = test_data_paths['sat_image_path']
val_mask_paths = test_data_paths['mask_path']
print(f"Evaluation set: {len(val_image_paths)} images, {len(val_mask_paths)} masks")
```

```
Training set: 803 images, 803 masks
Evaluation set: 172 images, 172 masks
```

In [24]:
```
# split the training data (w/ masks) into train/test splits for validation...
X_train, X_test, y_train, y_test = train_test_split(train_image_paths, train_mask_paths, test_size

# gather images/masks and ensure PANDAS format...
X_train = pd.Series(X_train.tolist())
X_test = pd.Series(X_test.tolist())
y_train = pd.Series(y_train.tolist())
y_test = pd.Series(y_test.tolist())

# verify equality consistent shaping...
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# ensure valid image(X)/mask paths(y)...
print("X_train:")
display(X_train.head(3))
print("X_test:")
```

```
display(X_test.head(3))
print("y_train:")
display(y_train.head(3))
print("y_test:")
display(y_test.head(3))
```

```
X_train shape: (642,)
X_test shape: (161,)
y_train shape: (642,)
y_test shape: (161,)
X_train:
```

	0
0	train/586222_sat.jpg
1	train/736869_sat.jpg
2	train/599743_sat.jpg

dtype: object

```
X_test:
```

	0
0	train/315352_sat.jpg
1	train/427774_sat.jpg
2	train/28689_sat.jpg

dtype: object

```
y_train:
```

	0
0	train/586222_mask.png
1	train/736869_mask.png
2	train/599743_mask.png

dtype: object

```
y_test:
```

	0
0	train/315352_mask.png
1	train/427774_mask.png
2	train/28689_mask.png

dtype: object

Image/Mask Processing

The training/validation sets are defined and we can proceed to iteratively process them using the predefined methods tested above. Previously tested in individual samples, the following procedures are applied to the entired dataset.

The functions below assign each image and mask to an instance of `Image` and `Mask` respectively. Options for training pre-processing include resizing, tiling and augmentation. One of the most effective approaches had been training the model on whole images, reduced in size ny 25%. Augmentation and tiling approaches are also being explored to see which approach trains the model most efficiently.

In [25]:
```
# preprocess all images and masks...
def preprocess_data(X_paths, y_paths, target_size=(512, 512), resize=False, tiling=False):
  # initialize arrays for populating...
  images = []
  masks = []

  #img_tiles = []
  #mask_tiles = []

  # iterate each image/mask pair, process each and package for training...
  for img_path, mask_path in tqdm(zip(X_paths, y_paths), total=len(X_paths), desc="Processing imag
    # generate objects...
    img = Image(img_path)
    mask = Mask(mask_path)

    # generate full-size tiles...
    if tiling:
      img.subdivide_image()
      mask.subdivide_mask()

    # tile full image before resizing, allowing training on either set...
    if resize:
      img.resize_image(target_size)
      mask.resize_mask(target_size)

    # append image and masks to arrays...
    images.append(img.image)
    masks.append(mask.ohe)

    # append tiles to arrays...
    #img_tiles.append(img.tiles)
    #mask_tiles.append(mask.tiles)

  return np.array(images), np.array(masks)#, np.array(img_tiles), np.array(mask_tiles)

# save generated numpy arrays to zip files...
def save_np_arrays(images, masks, save_dir_images, save_dir_masks):
  # generate local directories if none exist...
  os.makedirs(save_dir_images, exist_ok=True)
  os.makedirs(save_dir_masks, exist_ok=True)

  # save images as npz compressed files...
  np.savez_compressed(os.path.join(save_dir_images, save_dir_images), images=images)
  print(f"Compressing images to {save_dir_images}...")
  np.savez_compressed(os.path.join(save_dir_masks, save_dir_masks), masks=masks)
  print(f"Compressing masks to {save_dir_masks}...")
```

Optional Data Upload

The data processing pipeline is computationally taxing and can suffer from I/O bottle necking. Loading pre-processed data saves time and allows the notebook to be run with less hardware dependency.

If desired, images and mask pairs processed from previous sessions can be loaded in; assuming input files are in `.npz` format (NumPy Zip). When conducting test analysis alone, skip to uploading `Xtest_img` and `ytest_mask` to avoid unnecessarily loading in training data.

In [26]:
```
xtrain_img = files.upload()
ytrain_mask = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving X_train_images.npz to X_train_images.npz
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving y_train_masks.npz to y_train_masks.npz
```

In [28]:
```
# if both training sets were loaded in, swap in old...
if xtrain_img and ytrain_mask:
  print("importing training data...")

  # swap in existing training images...
  data = np.load("/content/X_train_images.npz")
  X_train_images = data['images']

  # swap in existing training masks...
  data = np.load("/content/y_train_masks.npz")
  y_train_masks = data['masks']

# process a new set for local storage...
else:
  print("incomplete or no data found, processing new batch...")

  # define training preprocessing routine here...
  X_train_images, y_train_masks = preprocess_data(X_train, y_train, target_size=(512, 512), resize
  save_np_arrays(X_train_images, y_train_masks, "X_train_images", "y_train_masks")

  # output the size of each set and typical image shape...
  print(f"number of train images: {len(X_train_images)}, sample image shape: {X_train_images[0].sh
  print(f"number of train masks: {len(y_train_masks)}, sample mask shape: {y_train_masks[0].shape}

print("import complete.")
```

```
importing training data...
import complete.
```

In [29]:
```
xtest_img = files.upload()
ytest_mask = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving X_test_images.npz to X_test_images.npz
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving y_test_masks.npz to y_test_masks.npz
```

In [30]:
```
# if both test sets were loaded in, swap in old...
if xtest_img and ytest_mask:
  print("importing test data...")

  # swap in existing training images...
  data = np.load("/content/X_test_images.npz")
  X_test_images = data['images']

  # swap in existing training masks...
  data = np.load("/content/y_test_masks.npz")
  y_test_masks = data['masks']

# process a new set for local storage...
```

```
else:
  print("incomplete or no data found, processing new batch...")

  # define training preprocessing routine here...
  X_test_images, y_test_masks = preprocess_data(X_test, y_test, target_size=(512, 512), resize=Tru
  save_np_arrays(X_test_images, y_test_masks, "X_test_images", "y_test_masks")

  # output the size of each set and typical image shape...
  print(f"number of test images: {len(X_test_images)}, sample image shape: {X_test_images[0].shape
  print(f"number of test masks: {len(y_test_masks)}, sample mask shape: {y_test_masks[0].shape}")

print("import complete.")
```

```
importing test data...
import complete.
```

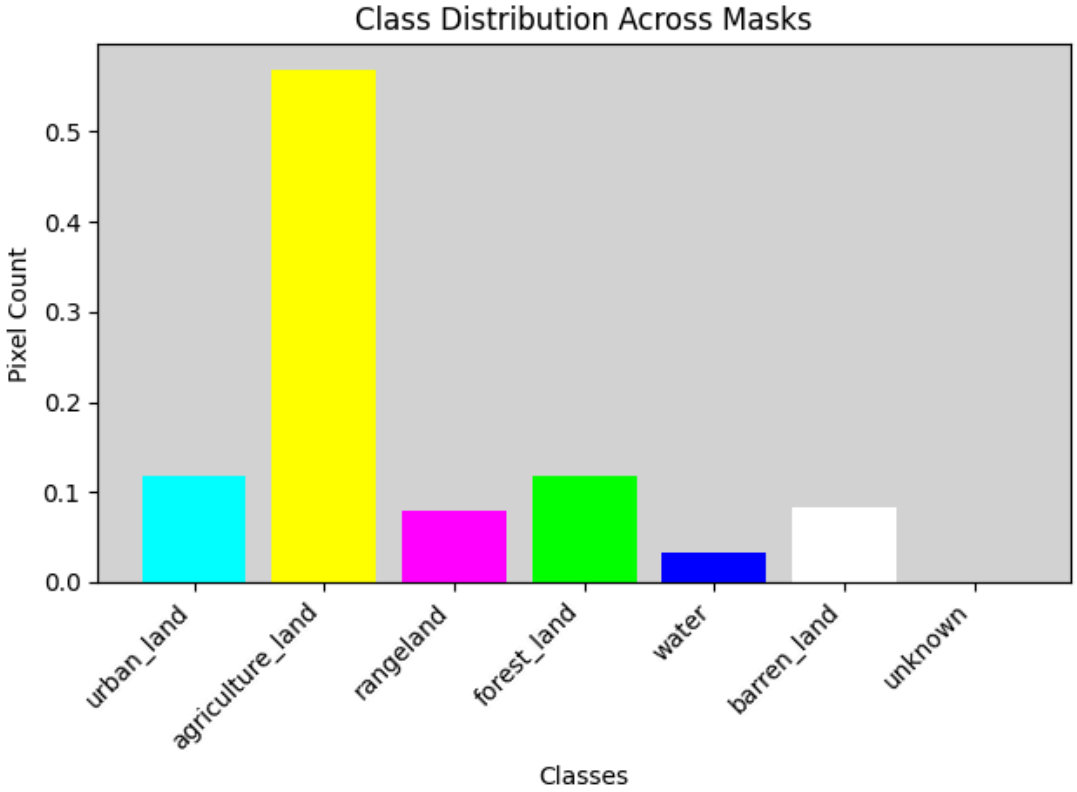
Analyzing Training Distribution

Before jumping into model training, were taking a look at the distribution of land cover across out training data. The `count_class_pixels` helper derives the grand total pixel count of each land cover class to be used in the `plot_class_pixel_counts_with_colors`

In [31]:
```
# define RGB -> class mapping...
class_colors = {
  (0, 255, 255): "urban_land",
  (255, 255, 0): "agriculture_land",
  (255, 0, 255): "rangeland",
  (0, 255, 0): "forest_land",
  (0, 0, 255): "water",
  (255, 255, 255): "barren_land",
  (0, 0, 0): "unknown",
}
# define RGB -> one hot encoding indices mapping...
color_indices = {
  (0, 255, 255): 0,  # urban_land
  (255, 255, 0): 1,  # agriculture_land
  (255, 0, 255): 2,  # rangeland
  (0, 255, 0): 3,    # forest_land
  (0, 0, 255): 4,    # water
  (255, 255, 255): 5, # barren_land
  (0, 0, 0): 6       # unknown
}

# find total class distribution across all training masks...
def count_class_pixels(masks):
  # stack masks into a single array of shape (N, H, W, num_classes)....
  masks_array = np.stack(masks, axis=0)

  # grab total number of classes...
  num_classes = masks_array.shape[-1]

  # initialize a list to count the occurrences of each class...
  class_counts = np.zeros(num_classes, dtype=int)

  # iterate over each pixel in each mask...
  for mask in masks_array:
    for pixel in mask.reshape(-1, num_classes):  # Reshape to (H*W, num_classes)
      class_index = np.argmax(pixel)  # grab the index of the class for each pixel...
      class_counts[class_index] += 1  # increment the count of class...

  return class_counts
```

```
# plot total class count distribution, color match to mapping...
def plot_class_pixel_counts(pixel_counts, class_colors, title):
  # extract colors and labels from the dictionary...
  bar_colors = [np.array(color) / 255.0 for color in class_colors.keys()]
  bar_labels = list(class_colors.values())

  # set up figure with background color for white...
  ax = plt.gca()
  ax.set_facecolor("#D3D3D3")

  # plot distribution proportions...
  plt.bar(bar_labels, pixel_counts/np.sum(pixel_counts), color=bar_colors)
  plt.xlabel("Classes")
  plt.ylabel("Pixel Count")
  plt.title("Class Distribution Across Masks")
  plt.xticks(rotation=45, ha="right")
  plt.tight_layout()
  plt.show()

# gather class distribution...
class_pixel_counts = count_class_pixels(y_train_masks)
print("Class pixel counts:", class_pixel_counts)

# print total number of pixels counted...
total_pixels_counted = np.sum(class_pixel_counts)
print("Total number of pixels counted:", total_pixels_counted)

# plot class distribution...
plot_class_pixel_counts(class_pixel_counts, class_colors, "Class Distribution Across Masks")

# log proportion of agriculture for reference...
print(f"Proportion of agriculture: {class_pixel_counts[1]/total_pixels_counted}")
```

```
Class pixel counts: [19739903 95652562 13418455 19864234  5378020 14142750   100524]
Total number of pixels counted: 168296448
```




```
Proportion of agriculture: 0.5683575805473922
```

The training data shows that land cover is overwhelmingly comprised of agricultural land, with all other classes following significantly behind. Given such a strong representation of a single class, I want to be sure the model does not form a strategy to predict only agricultural land.

It may be wise to weight our classes in the case one of the classes, like `agricultural_land`, when one/a few are being overrepresented. Our World in Data finds that:

- Agricultural land comprises ~37% of all land cover
- Forest land comprises ~31% of all land cover

Agricultural land can be found in around 57% of the total pixels in training, which is *significantly greater* than the global average. This alone is not enough to intervene just yet, so long as our model continues to learn well past this threshold, we can assume it is learning to identify other land cover types.

While this particular dataset is composed of satellite imagery from around the globe, many of its images focusing on rural and semi-rural areas in Africa, South America and parts of Asia. These communities tend to lean more agricultural in nature, making the increased proportion of `agricultural_land` acceptable until proven otherwise.

Model Training

The first few cells involve establishing and fitting the model. If you plan to load a model at this point, skip past model fitting and uncomment the calls to load in a previous model.

Model Architecture

The model leverages a U-Net model implementation using TensorFlow/Keras for our image segmentation task. According to my research, this type of model is typically used for biology, medical and other fine image segmentation applications.

It is said to perform well with capturing both local (micro) and global (macro) features. The definition of the U-Net architecture is laid out in the following structure:

- Input
  - Establishes input size to the shape of our training images
  - Expects input of shape: (Width, Height, (R,G,B))
- Encoder / Contracting Path
  - Builds up the layers of convolutional layers/filters
  - The number of filters increases the depth of the image/feature map accordingly
  - Extracts spatial features; edges, textures, patterns, etc.
- Bottleneck
  - 'Lowest point' of the U-Net model used to extract deep features
  - Final translation to a max depth of 256 in the current model
- Decoder / Expansive Path
  - Aims to recover information from downsampling (encoding)
  - Integrates insights learned in each level into a combined feature map
- Output
  - Returns the overall feature map depth back to `num_classes` (7)
  - Ensures prediction output matches standard land cover masks
  - Softmax is preferred for pixel-wise class probabilities

Model Definition

Declared using a dynamic learning rate; Adam (Adaptive Moment Estimation) prioritizes momentum or gradients formed in previous iterations. This aims to facilitate progress and accelerated convergence.

As the model trains, the learning rate is adjusted automatically to prevent overfitting. The small initial rate prevents forming strong opinions too quickly.

The loss function is categorical crossentropy using accuracy as the evaluation metric. This loss function is well-suited to multi-class segmentation where the model predicts pixel-by-pixel. It measures the difference between predicted probabilities and true class labels, penalizing incorrect predictions on a graduated scale according to the model's confidence.

In [60]:
```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from sklearn.utils import class_weight

def unet_model(input_shape, num_classes):
  # defines the input shape to expect (H,W,RGB)...
  inputs = layers.Input(input_shape)

  # encoder; contracting path translates map to (W,H,64)...
  c1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
  c1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c1)
  p1 = layers.MaxPooling2D((2, 2))(c1)

  # encoder; contracting path translates map to (W,H,128)...
  c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
  c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c2)
  p2 = layers.MaxPooling2D((2, 2))(c2)

  # bottleneck; max depth (W,H,256)...
  c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
  c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c3)

  # decoder; expansive path translates map to (W,H,128)...
  u4 = layers.UpSampling2D((2, 2))(c3)
  u4 = layers.concatenate([u4, c2], axis=-1)
  c4 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(u4)
  c4 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c4)

  # decoder; expansive path translates map to (W,H,64)...
  u5 = layers.UpSampling2D((2, 2))(c4)
  u5 = layers.concatenate([u5, c1], axis=-1)
  c5 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(u5)
  c5 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c5)

  # returns output to (W,H,7), with softmax activation...
  outputs = layers.Conv2D(num_classes, (1, 1), activation='softmax')(c5)

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

# Define input shape and number of classes
input_shape = X_train_images[0].shape
num_classes = len(class_colors)

# initialize the model...
model = unet_model(input_shape, num_classes)
```

```
optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)

# compile the model...
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# plot summary...
model.summary()
```

Model: "functional_2"

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 512, 512, 3)	0	-
conv2d_22 (Conv2D)	(None, 512, 512, 64)	1,792	input_layer_2[0][0]
conv2d_23 (Conv2D)	(None, 512, 512, 64)	36,928	conv2d_22[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 256, 256, 64)	0	conv2d_23[0][0]
conv2d_24 (Conv2D)	(None, 256, 256, 128)	73,856	max_pooling2d_4[0][
conv2d_25 (Conv2D)	(None, 256, 256, 128)	147,584	conv2d_24[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 128, 128, 128)	0	conv2d_25[0][0]
conv2d_26 (Conv2D)	(None, 128, 128, 256)	295,168	max_pooling2d_5[0][
conv2d_27 (Conv2D)	(None, 128, 128, 256)	590,080	conv2d_26[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 256, 256, 256)	0	conv2d_27[0][0]
concatenate_4 (Concatenate)	(None, 256, 256, 384)	0	up_sampling2d_4[0][conv2d_25[0][0]
conv2d_28 (Conv2D)	(None, 256, 256, 128)	442,496	concatenate_4[0][0]
conv2d_29 (Conv2D)	(None, 256, 256, 128)	147,584	conv2d_28[0][0]
up_sampling2d_5 (UpSampling2D)	(None, 512, 512, 128)	0	conv2d_29[0][0]
concatenate_5 (Concatenate)	(None, 512, 512, 192)	0	up_sampling2d_5[0][conv2d_23[0][0]
conv2d_30 (Conv2D)	(None, 512, 512, 64)	110,656	concatenate_5[0][0]
conv2d_31 (Conv2D)	(None, 512, 512, 64)	36,928	conv2d_30[0][0]
conv2d_32 (Conv2D)	(None, 512, 512, 7)	455	conv2d_31[0][0]

Total params: 1,883,527 (7.19 MB)

Trainable params: 1,883,527 (7.19 MB)

Non-trainable params: 0 (0.00 B)

Applying Augmentations

To apply augmentations on the fly (ie: between training epochs), we package our previous augmentation strategy within a helper that packages our training and test data and applies augmentations to increase the diversity of our training set.

Either comment this section out and move forward with the original dataset, or uncomment and train our model using the dynamic image augmentation strategy. Just as before, this process is computationally taxing and can simply be left to TensorFlow in-built methods to integrate with our model efficiently.

In [61]:
```
''' implementation of tensor dataset for augmentations between epochs... '''
# apply same augmentations seen earlier...
def augment(image, mask):
  # random horizontal flip...
  h_flip = tf.random.uniform([], 0, 1) > 0.5
  image = tf.cond(h_flip, lambda: tf.image.flip_left_right(image), lambda: image)
  mask = tf.cond(h_flip, lambda: tf.image.flip_left_right(mask), lambda: mask)
  #image = tf.image.random_flip_left_right(image)
  #mask = tf.image.random_flip_left_right(mask)

  # random vertical flip...
  v_flip = tf.random.uniform([], 0, 1) > 0.5
  image = tf.cond(v_flip, lambda: tf.image.flip_up_down(image), lambda: image)
  mask = tf.cond(v_flip, lambda: tf.image.flip_up_down(mask), lambda: mask)
  #image = tf.image.random_flip_up_down(image)
  #mask = tf.image.random_flip_up_down(mask)

  # random 90-degree rotation...
  k = tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32)
  image = tf.image.rot90(image, k=k)
  mask = tf.image.rot90(mask, k=k)

  # random brightness/contrast (images only)...
  image = tf.image.random_brightness(image, max_delta=0.2)
  image = tf.image.random_contrast(image, lower=0.8, upper=1.2)

  return image, mask

# package into tensorflow dataset...
def create_dataset(X, y, target_size=(512, 512), batch_size=8, apply_augment=False):
  def preprocess(img, mask):
    # resize images and masks to the target size...
    img = tf.image.resize(img, target_size)
    mask = tf.image.resize(mask, target_size)
    return img, mask

  dataset = tf.data.Dataset.from_tensor_slices((X, y))
  dataset = dataset.map(lambda img, mask: preprocess(img, mask))

  if augment:
    dataset = dataset.map(augment, num_parallel_calls=tf.data.AUTOTUNE)

  dataset = dataset.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)
  return dataset
```

Model Parameters

Now that the model architecture is defined, the model can finally be trained/fit to our data. To prevent our model from overfitting, we leverage early stopping and another level of dynamic learning rate adjustments.

While the `Adam` implementation of adaptive learning rates applied directly to each parameter internally, the `ReduceLROnPlateau` implementation looks at our model's learning at a macro scale. The former is

reassessed parameter by parameter, increasing and decreasing learning rates according to it's confidence and progress rates. The latter acts globally, ensuring slower learning rates once learning progress slows. This action aims to hone in on fine adjustments as our model begins to converge.

A second model fitting approach was also defined in an attempt to apply TensorFlow augmentations between training epochs. Initial testing showed diminished performance, further exploration of this approach will be explored (time willing).

In [62]:
```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import ReduceLROnPlateau

# leverage early stopping and dynamic larning rates to prevent overfitting...
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-6)

''' model fitting for self-defined image processing... '''
"""history = model.fit(
  X_train_images,
  y_train_masks,
  epochs=100,
  batch_size=8,
  validation_data=(X_test_images, y_test_masks),
  callbacks=[early_stopping, lr_scheduler]
)"""

''' model fitting for tensorflow image processing... '''
# generate datasets...
train_dataset = create_dataset(X_train_images, y_train_masks, apply_augment=True)
val_dataset = create_dataset(X_test_images, y_test_masks, apply_augment=False)

history = model.fit(
  train_dataset,
  epochs=100,
  batch_size=8,
  validation_data=val_dataset,
  callbacks=[early_stopping, lr_scheduler]
)

# check training history for accuracy and loss...
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

plt.legend()
plt.show()
```

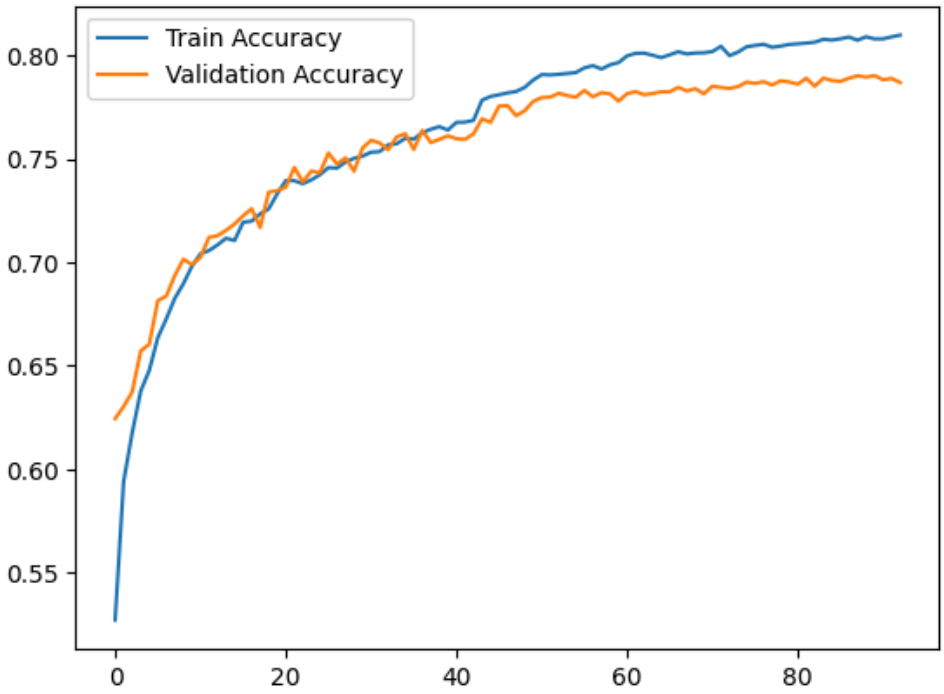
```
Epoch 1/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 23s 186ms/step - accuracy: 0.4510 - loss: 2.3665 - val_accuracy: 0.6245
- val_loss: 1.2122 - learning_rate: 1.0000e-04
Epoch 2/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.5658 - loss: 1.3167 - val_accuracy: 0.6304
- val_loss: 1.1540 - learning_rate: 1.0000e-04
Epoch 3/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.5918 - loss: 1.2048 - val_accuracy: 0.6374
- val_loss: 1.1261 - learning_rate: 1.0000e-04
Epoch 4/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6132 - loss: 1.1787 - val_accuracy: 0.6572
- val_loss: 1.0518 - learning_rate: 1.0000e-04
Epoch 5/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6258 - loss: 1.1359 - val_accuracy: 0.6604
- val_loss: 1.0274 - learning_rate: 1.0000e-04
Epoch 6/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6433 - loss: 1.0904 - val_accuracy: 0.6815
- val_loss: 0.9897 - learning_rate: 1.0000e-04
Epoch 7/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6541 - loss: 1.0702 - val_accuracy: 0.6837
- val_loss: 0.9850 - learning_rate: 1.0000e-04
Epoch 8/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6702 - loss: 1.0201 - val_accuracy: 0.6937
- val_loss: 0.9506 - learning_rate: 1.0000e-04
Epoch 9/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6769 - loss: 0.9888 - val_accuracy: 0.7015
- val_loss: 0.9210 - learning_rate: 1.0000e-04
Epoch 10/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6873 - loss: 0.9672 - val_accuracy: 0.6989
- val_loss: 0.9163 - learning_rate: 1.0000e-04
Epoch 11/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.6924 - loss: 0.9518 - val_accuracy: 0.7024
- val_loss: 0.8950 - learning_rate: 1.0000e-04
Epoch 12/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.6956 - loss: 0.9397 - val_accuracy: 0.7121
- val_loss: 0.8713 - learning_rate: 1.0000e-04
Epoch 13/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7010 - loss: 0.9157 - val_accuracy: 0.7129
- val_loss: 0.8780 - learning_rate: 1.0000e-04
Epoch 14/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7026 - loss: 0.9124 - val_accuracy: 0.7155
- val_loss: 0.8800 - learning_rate: 1.0000e-04
Epoch 15/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7020 - loss: 0.9163 - val_accuracy: 0.7185
- val_loss: 0.8617 - learning_rate: 1.0000e-04
Epoch 16/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7071 - loss: 0.8914 - val_accuracy: 0.7225
- val_loss: 0.8407 - learning_rate: 1.0000e-04
Epoch 17/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7114 - loss: 0.8967 - val_accuracy: 0.7259
- val_loss: 0.8479 - learning_rate: 1.0000e-04
Epoch 18/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7192 - loss: 0.8616 - val_accuracy: 0.7169
- val_loss: 0.8487 - learning_rate: 1.0000e-04
Epoch 19/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7157 - loss: 0.8666 - val_accuracy: 0.7339
- val_loss: 0.8152 - learning_rate: 1.0000e-04
Epoch 20/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7285 - loss: 0.8402 - val_accuracy: 0.7346
- val_loss: 0.8067 - learning_rate: 1.0000e-04
Epoch 21/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7361 - loss: 0.8166 - val_accuracy: 0.7363
- val_loss: 0.8137 - learning_rate: 1.0000e-04
Epoch 22/100
```

```
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7332 - loss: 0.8160 - val_accuracy: 0.7458
- val_loss: 0.7976 - learning_rate: 1.0000e-04
Epoch 23/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7289 - loss: 0.8318 - val_accuracy: 0.7389
- val_loss: 0.7978 - learning_rate: 1.0000e-04
Epoch 24/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7334 - loss: 0.8091 - val_accuracy: 0.7441
- val_loss: 0.7860 - learning_rate: 1.0000e-04
Epoch 25/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7344 - loss: 0.8063 - val_accuracy: 0.7433
- val_loss: 0.7746 - learning_rate: 1.0000e-04
Epoch 26/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7415 - loss: 0.7827 - val_accuracy: 0.7528
- val_loss: 0.7723 - learning_rate: 1.0000e-04
Epoch 27/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7412 - loss: 0.7813 - val_accuracy: 0.7475
- val_loss: 0.7841 - learning_rate: 1.0000e-04
Epoch 28/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7423 - loss: 0.7837 - val_accuracy: 0.7505
- val_loss: 0.7668 - learning_rate: 1.0000e-04
Epoch 29/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7438 - loss: 0.7692 - val_accuracy: 0.7440
- val_loss: 0.7925 - learning_rate: 1.0000e-04
Epoch 30/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7449 - loss: 0.7704 - val_accuracy: 0.7553
- val_loss: 0.7551 - learning_rate: 1.0000e-04
Epoch 31/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7465 - loss: 0.7658 - val_accuracy: 0.7589
- val_loss: 0.7487 - learning_rate: 1.0000e-04
Epoch 32/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7496 - loss: 0.7546 - val_accuracy: 0.7577
- val_loss: 0.7579 - learning_rate: 1.0000e-04
Epoch 33/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7511 - loss: 0.7461 - val_accuracy: 0.7544
- val_loss: 0.7382 - learning_rate: 1.0000e-04
Epoch 34/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7510 - loss: 0.7458 - val_accuracy: 0.7606
- val_loss: 0.7382 - learning_rate: 1.0000e-04
Epoch 35/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7573 - loss: 0.7357 - val_accuracy: 0.7622
- val_loss: 0.7255 - learning_rate: 1.0000e-04
Epoch 36/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7548 - loss: 0.7388 - val_accuracy: 0.7545
- val_loss: 0.7439 - learning_rate: 1.0000e-04
Epoch 37/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7574 - loss: 0.7230 - val_accuracy: 0.7637
- val_loss: 0.7194 - learning_rate: 1.0000e-04
Epoch 38/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7613 - loss: 0.7186 - val_accuracy: 0.7579
- val_loss: 0.7455 - learning_rate: 1.0000e-04
Epoch 39/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7608 - loss: 0.7167 - val_accuracy: 0.7594
- val_loss: 0.7376 - learning_rate: 1.0000e-04
Epoch 40/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7601 - loss: 0.7193 - val_accuracy: 0.7611
- val_loss: 0.7185 - learning_rate: 1.0000e-04
Epoch 41/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.7659 - loss: 0.6952 - val_accuracy: 0.7597
- val_loss: 0.7301 - learning_rate: 1.0000e-04
Epoch 42/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7636 - loss: 0.7070 - val_accuracy: 0.7594
- val_loss: 0.7369 - learning_rate: 1.0000e-04
Epoch 43/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7662 - loss: 0.6961 - val_accuracy: 0.7621
```

```
- val_loss: 0.7205 - learning_rate: 1.0000e-04
Epoch 44/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7754 - loss: 0.6722 - val_accuracy: 0.7693
- val_loss: 0.6953 - learning_rate: 5.0000e-05
Epoch 45/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7797 - loss: 0.6634 - val_accuracy: 0.7676
- val_loss: 0.6930 - learning_rate: 5.0000e-05
Epoch 46/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7789 - loss: 0.6619 - val_accuracy: 0.7755
- val_loss: 0.6726 - learning_rate: 5.0000e-05
Epoch 47/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7805 - loss: 0.6523 - val_accuracy: 0.7757
- val_loss: 0.6728 - learning_rate: 5.0000e-05
Epoch 48/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7803 - loss: 0.6508 - val_accuracy: 0.7708
- val_loss: 0.6861 - learning_rate: 5.0000e-05
Epoch 49/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7839 - loss: 0.6462 - val_accuracy: 0.7733
- val_loss: 0.6789 - learning_rate: 5.0000e-05
Epoch 50/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7855 - loss: 0.6346 - val_accuracy: 0.7778
- val_loss: 0.6548 - learning_rate: 2.5000e-05
Epoch 51/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7909 - loss: 0.6176 - val_accuracy: 0.7796
- val_loss: 0.6501 - learning_rate: 2.5000e-05
Epoch 52/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7912 - loss: 0.6175 - val_accuracy: 0.7798
- val_loss: 0.6499 - learning_rate: 2.5000e-05
Epoch 53/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7922 - loss: 0.6131 - val_accuracy: 0.7817
- val_loss: 0.6424 - learning_rate: 2.5000e-05
Epoch 54/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7917 - loss: 0.6137 - val_accuracy: 0.7805
- val_loss: 0.6528 - learning_rate: 2.5000e-05
Epoch 55/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7917 - loss: 0.6145 - val_accuracy: 0.7798
- val_loss: 0.6484 - learning_rate: 2.5000e-05
Epoch 56/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7948 - loss: 0.6061 - val_accuracy: 0.7831
- val_loss: 0.6406 - learning_rate: 2.5000e-05
Epoch 57/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7962 - loss: 0.6019 - val_accuracy: 0.7800
- val_loss: 0.6496 - learning_rate: 2.5000e-05
Epoch 58/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7953 - loss: 0.6019 - val_accuracy: 0.7820
- val_loss: 0.6444 - learning_rate: 2.5000e-05
Epoch 59/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7956 - loss: 0.6033 - val_accuracy: 0.7814
- val_loss: 0.6427 - learning_rate: 2.5000e-05
Epoch 60/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7946 - loss: 0.5985 - val_accuracy: 0.7779
- val_loss: 0.6422 - learning_rate: 1.2500e-05
Epoch 61/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7991 - loss: 0.5880 - val_accuracy: 0.7815
- val_loss: 0.6402 - learning_rate: 1.2500e-05
Epoch 62/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8009 - loss: 0.5852 - val_accuracy: 0.7825
- val_loss: 0.6365 - learning_rate: 1.2500e-05
Epoch 63/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.8011 - loss: 0.5839 - val_accuracy: 0.7810
- val_loss: 0.6394 - learning_rate: 1.2500e-05
Epoch 64/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8004 - loss: 0.5826 - val_accuracy: 0.7816
- val_loss: 0.6362 - learning_rate: 1.2500e-05
```

```
Epoch 65/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7999 - loss: 0.5866 - val_accuracy: 0.7824
- val_loss: 0.6379 - learning_rate: 1.2500e-05
Epoch 66/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8023 - loss: 0.5813 - val_accuracy: 0.7824
- val_loss: 0.6310 - learning_rate: 1.2500e-05
Epoch 67/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8032 - loss: 0.5781 - val_accuracy: 0.7846
- val_loss: 0.6306 - learning_rate: 1.2500e-05
Epoch 68/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.7998 - loss: 0.5859 - val_accuracy: 0.7828
- val_loss: 0.6329 - learning_rate: 1.2500e-05
Epoch 69/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8035 - loss: 0.5778 - val_accuracy: 0.7839
- val_loss: 0.6294 - learning_rate: 1.2500e-05
Epoch 70/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8020 - loss: 0.5797 - val_accuracy: 0.7814
- val_loss: 0.6355 - learning_rate: 1.2500e-05
Epoch 71/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8043 - loss: 0.5753 - val_accuracy: 0.7851
- val_loss: 0.6247 - learning_rate: 1.2500e-05
Epoch 72/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 134ms/step - accuracy: 0.8053 - loss: 0.5722 - val_accuracy: 0.7845
- val_loss: 0.6282 - learning_rate: 1.2500e-05
Epoch 73/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8000 - loss: 0.5835 - val_accuracy: 0.7840
- val_loss: 0.6284 - learning_rate: 1.2500e-05
Epoch 74/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8014 - loss: 0.5817 - val_accuracy: 0.7849
- val_loss: 0.6252 - learning_rate: 1.2500e-05
Epoch 75/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8042 - loss: 0.5734 - val_accuracy: 0.7871
- val_loss: 0.6197 - learning_rate: 6.2500e-06
Epoch 76/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8057 - loss: 0.5687 - val_accuracy: 0.7864
- val_loss: 0.6212 - learning_rate: 6.2500e-06
Epoch 77/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8064 - loss: 0.5691 - val_accuracy: 0.7873
- val_loss: 0.6192 - learning_rate: 6.2500e-06
Epoch 78/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8045 - loss: 0.5696 - val_accuracy: 0.7857
- val_loss: 0.6215 - learning_rate: 6.2500e-06
Epoch 79/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8066 - loss: 0.5678 - val_accuracy: 0.7877
- val_loss: 0.6153 - learning_rate: 6.2500e-06
Epoch 80/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8049 - loss: 0.5713 - val_accuracy: 0.7871
- val_loss: 0.6193 - learning_rate: 6.2500e-06
Epoch 81/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8075 - loss: 0.5640 - val_accuracy: 0.7861
- val_loss: 0.6225 - learning_rate: 6.2500e-06
Epoch 82/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8081 - loss: 0.5616 - val_accuracy: 0.7889
- val_loss: 0.6159 - learning_rate: 6.2500e-06
Epoch 83/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8070 - loss: 0.5641 - val_accuracy: 0.7851
- val_loss: 0.6255 - learning_rate: 3.1250e-06
Epoch 84/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8093 - loss: 0.5588 - val_accuracy: 0.7890
- val_loss: 0.6106 - learning_rate: 3.1250e-06
Epoch 85/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8093 - loss: 0.5590 - val_accuracy: 0.7879
- val_loss: 0.6177 - learning_rate: 3.1250e-06
Epoch 86/100
```

```
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8095 - loss: 0.5593 - val_accuracy: 0.7874
- val_loss: 0.6163 - learning_rate: 3.1250e-06
Epoch 87/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8100 - loss: 0.5556 - val_accuracy: 0.7890
- val_loss: 0.6158 - learning_rate: 3.1250e-06
Epoch 88/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8088 - loss: 0.5608 - val_accuracy: 0.7901
- val_loss: 0.6104 - learning_rate: 1.5625e-06
Epoch 89/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8105 - loss: 0.5563 - val_accuracy: 0.7895
- val_loss: 0.6112 - learning_rate: 1.5625e-06
Epoch 90/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8094 - loss: 0.5593 - val_accuracy: 0.7902
- val_loss: 0.6122 - learning_rate: 1.5625e-06
Epoch 91/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8090 - loss: 0.5593 - val_accuracy: 0.7882
- val_loss: 0.6178 - learning_rate: 1.5625e-06
Epoch 92/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8097 - loss: 0.5580 - val_accuracy: 0.7888
- val_loss: 0.6128 - learning_rate: 1.0000e-06
Epoch 93/100
81/81 ━━━━━━━━━━━━━━━━━━━━ 11s 135ms/step - accuracy: 0.8108 - loss: 0.5533 - val_accuracy: 0.7869
- val_loss: 0.6215 - learning_rate: 1.0000e-06
```



Model Prediction

Optionally **save** and/or **load** a trained/fit model for forming prediction masks. Uncomment the combination of lines that suits your current needs before proceeding to generate predictions on our test images.

In [63]:
```
os.makedirs("/content/models", exist_ok=True)
model.save("/content/models/standard_1209_0115.keras")
#files.upload()
#model = load_model("/content/standard_1208_0915.keras")

# make predictions...
predictions = model.predict(X_test_images)
print("predictions shape:", predictions.shape)
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 180ms/step
predictions shape: (161, 512, 512, 7)
```

Visualizing the Predictions

The first function below, `convert_predictions_to_colors`, gathers the predicted color mask for each pixel, applying the inverse mapping to reassign the RGB tuple value to each pixel, rather than the one hot encoded indexing.

The second, `plot_overlay_with_predictions`, accepts an image index, test image, and truth mask and plots an array of images for each prediction; the original satellite image, the model's predicted land cover mask overlay, and the true land cover mask overlay.

In [64]:
```
# mapping ohe indexing to RBG vals...
inverse_mapping = {
  0: (0, 255, 255),  # urban_land
  1: (255, 255, 0),  # agriculture_land
  2: (255, 0, 255),  # rangeland
  3: (0, 255, 0),    # forest_land
  4: (0, 0, 255),    # water
  5: (255, 255, 255), # barren_land
  6: (0, 0, 0)       # unknown
}

# decode ohe predictions to color mask coloration...
def convert_predictions_to_colors(predictions, mapping, verbose=True):
  # collect predicted class index [0-7] for each pixel (2D array)...
  predicted_classes = np.argmax(predictions, axis=-1)  # Shape: (256, 256)

  # place mapping colors into a list in their respective index...
  color_list = np.array([mapping[i] for i in sorted(mapping.keys())])

  if verbose:
    print("predicted_classes shape:", predicted_classes.shape)
    print("color_list:", np.array(color_list))

  # np.take replaces each class index to its color in list...
  color_predictions = np.take(color_list, predicted_classes, axis=0)
  return color_predictions

# plot array of images/mask overlays for visual comparison...
def plot_overlay_with_predictions(index_img=0, images=X_test_images, true_masks=y_test, prediction
  # load the true mask...
  img = plt.imread(true_masks[index_img])

  # resize image to match predicted mask, if it was downsized...
  if img.shape[:2] != predictions[index_img].shape[:2]:
      img = cv.resize(img, (predictions[index_img].shape[1], predictions[index_img].shape[0]))

  # get the predicted color map from predictions...
  predicted_colors = convert_predictions_to_colors(predictions[index_img], inverse_mapping, verbos

  # create a figure and axe...
  fig, axs = plt.subplots(1, 3, figsize=(24, 8))

  # plot satellite image...
  axs[0].imshow(images[index_img])
  axs[0].set_title(f"Sat Image [{index_img}]")
  axs[0].axis('off')

  # plot overlay of predicted mask on satellite image...
  axs[1].imshow(images[index_img])
```

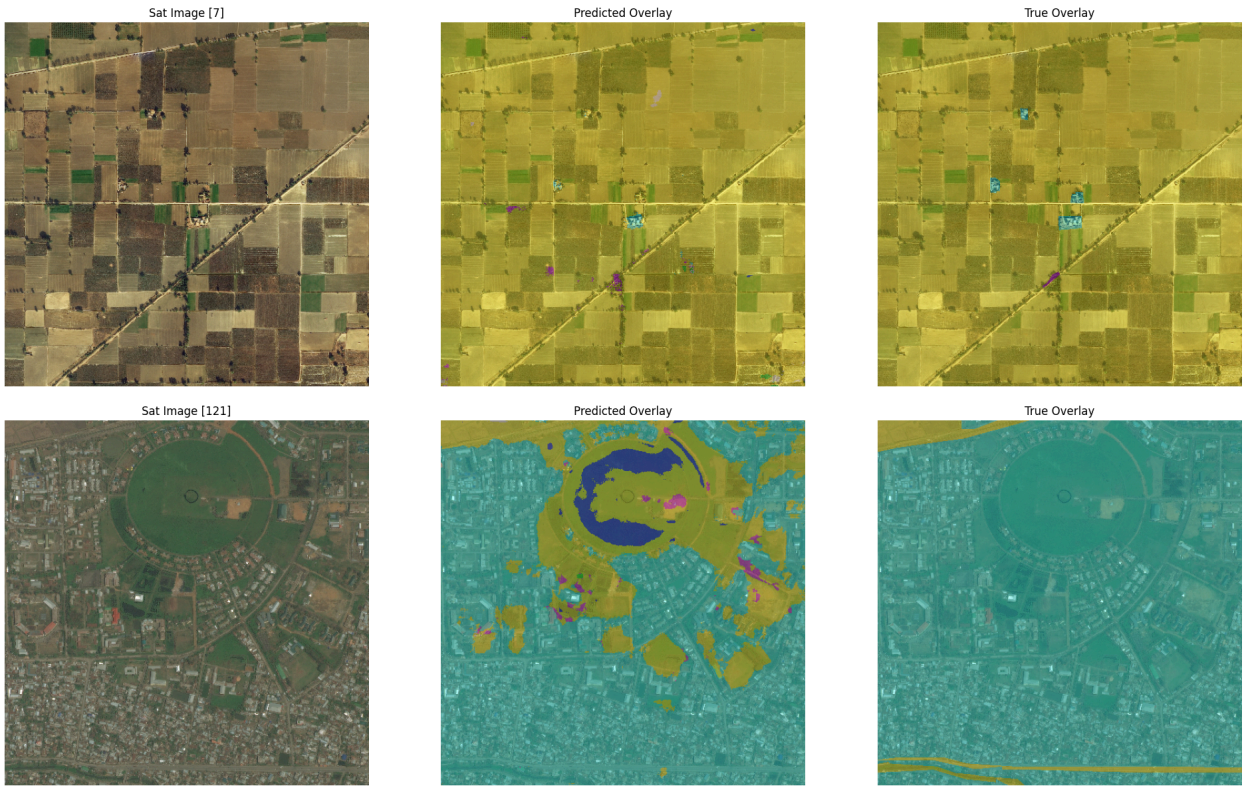
```
  axs[1].imshow(predicted_colors, alpha=0.3)  # Adjust alpha to control transparency of the overla
  axs[1].set_title("Predicted Overlay")
  axs[1].axis('off')

  # plot overlay of true mask on satellite image...
  axs[2].imshow(images[index_img])
  axs[2].imshow(img, alpha=0.3)
  axs[2].set_title("True Overlay")
  axs[2].axis('off')

  # show the plot...
  plt.show()
```

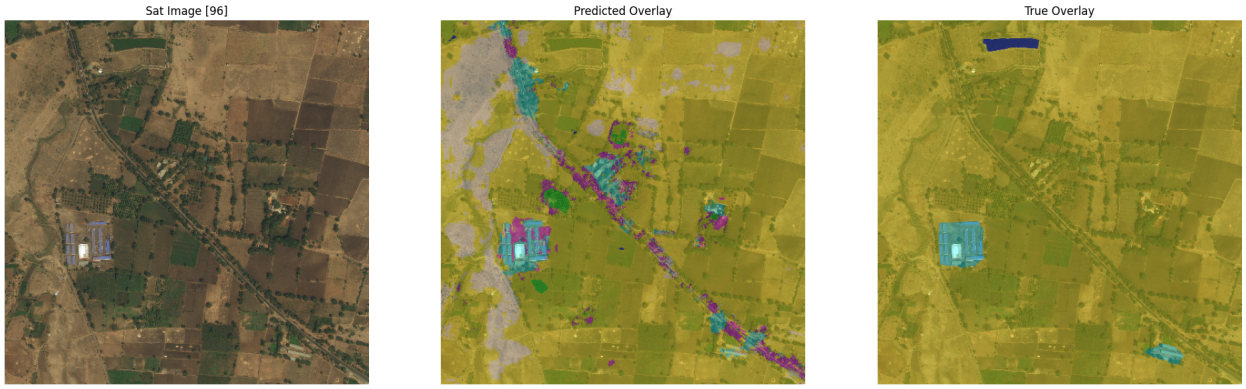
The model predictions are converted to prediction masks of a valid shape 166 masks of the correct dimensions. The `color_list` shows the RGB values that were used to map the predictions.

In [65]:
```
# convert predicted ohe masks to RGB masks...
predicted_mask = convert_predictions_to_colors(predictions, inverse_mapping, verbose=True)
```

```
predicted_classes shape: (161, 512, 512)
color_list: [[  0 255 255]
 [255 255   0]
 [255   0 255]
 [  0 255   0]
 [  0   0 255]
 [255 255 255]
 [  0   0   0]]
```

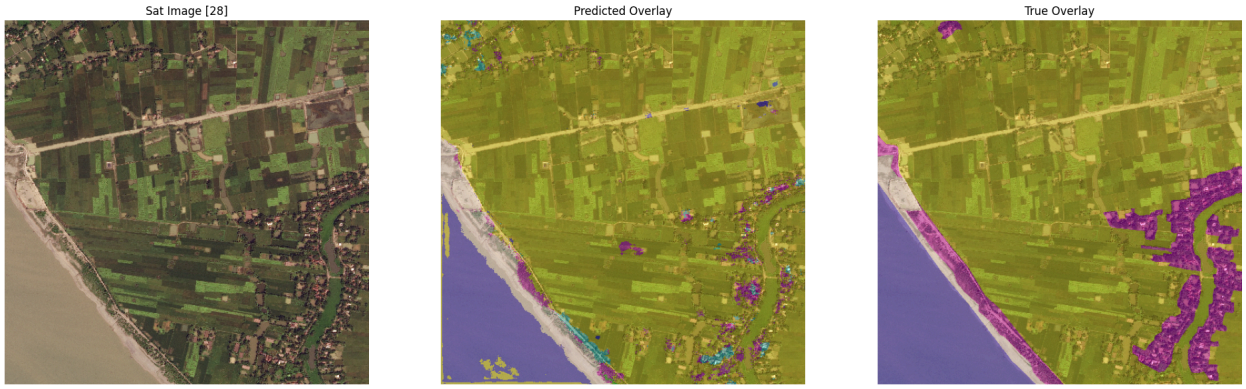
In [66]:
```
for i in random.sample(range(len(X_test_images)), 3):
  plot_overlay_with_predictions(index_img=i, images=X_test_images, predictions=predictions, true_m
```



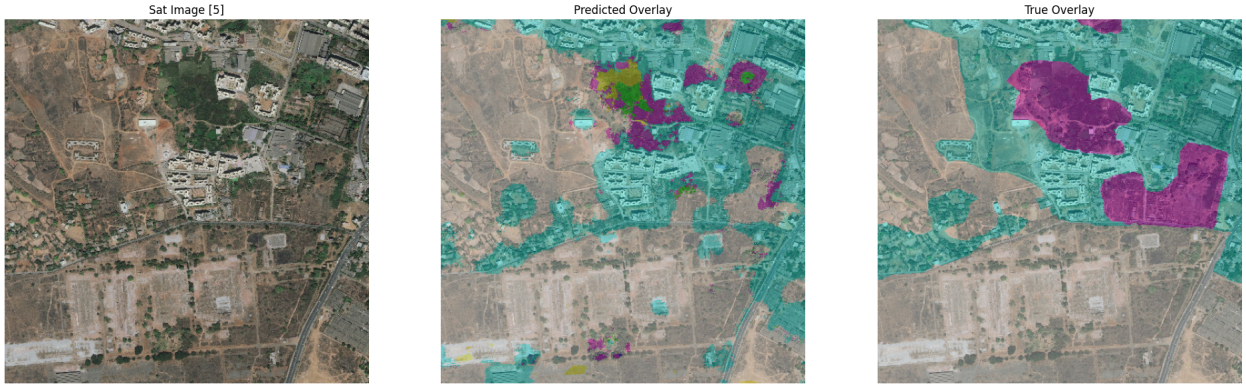


In [67]:
```
cherry_picks = {
  "(+) more discerning of urban/range relationship": 28,
  "(+/-) more dicerning of urban boundary, possibly misclassifying rangeland": 5,
  "(-) pixel-wise prediction may be too targeted": 68,
  "(-) confused rocky terrain as urban construction": 15,
  "(-) confused with diverse water conditions": 103,
  "(-) confused with water segmentation in general": 20,
  "(+/-) over specificity/lack of detail in truth mask": 149,
  "(+/-) detects forested islands while missing the river": 141
}

for key, val in cherry_picks.items():
  print(key)
  plot_overlay_with_predictions(index_img=val, images=X_test_images, predictions=predictions, true
```

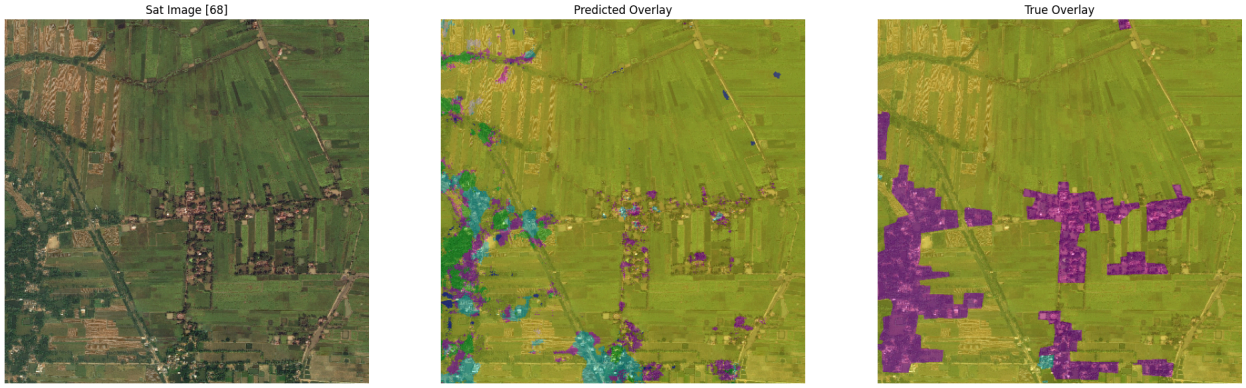
```
(+) more discerning of urban/range relationship
```



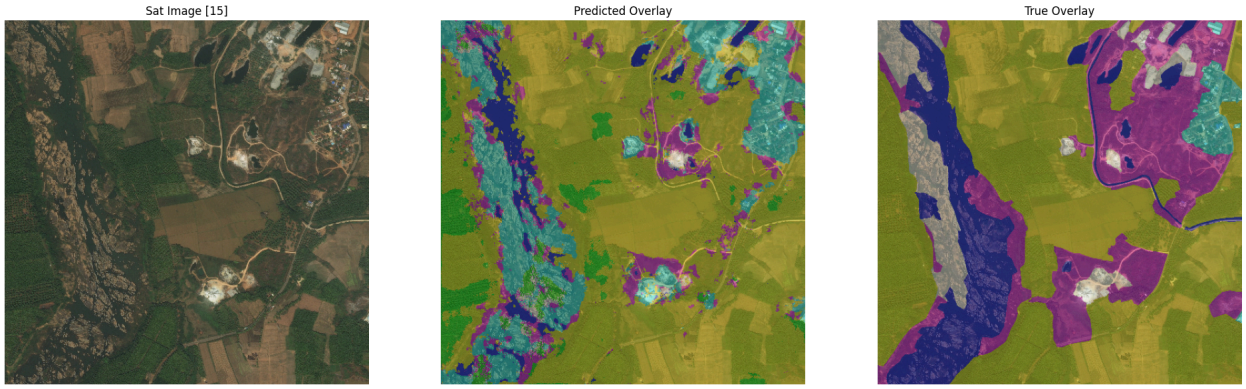

```
(+/-) more dicerning of urban boundary, possibly misclassifying rangeland
```



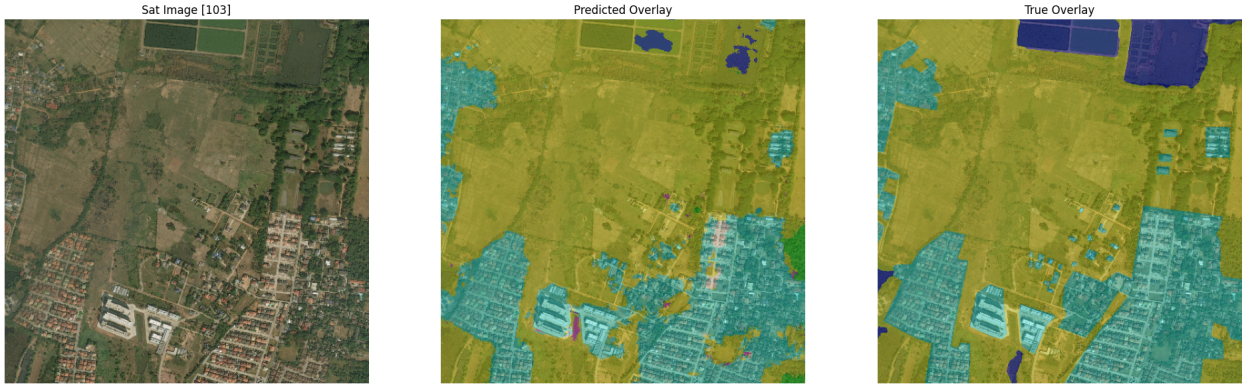

```
(-) pixel-wise prediction may be too targeted
```



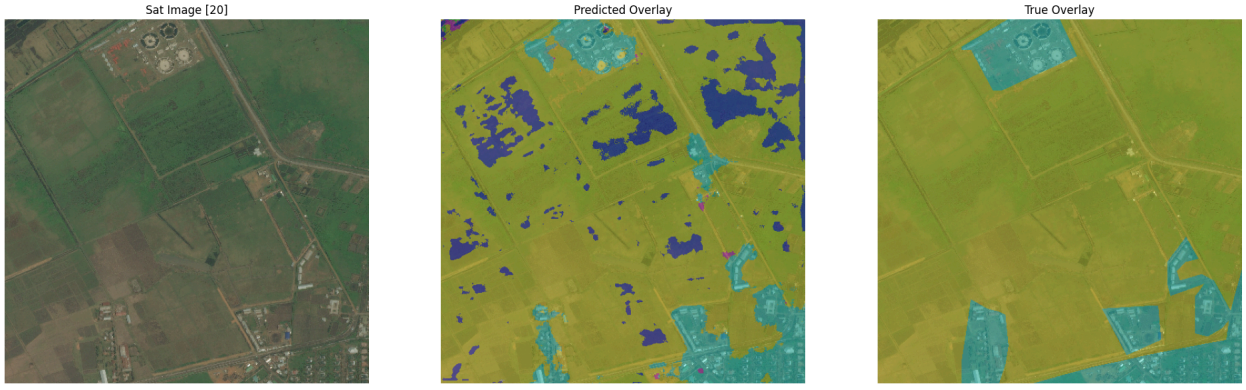
(-) confused rocky terrain as urban construction



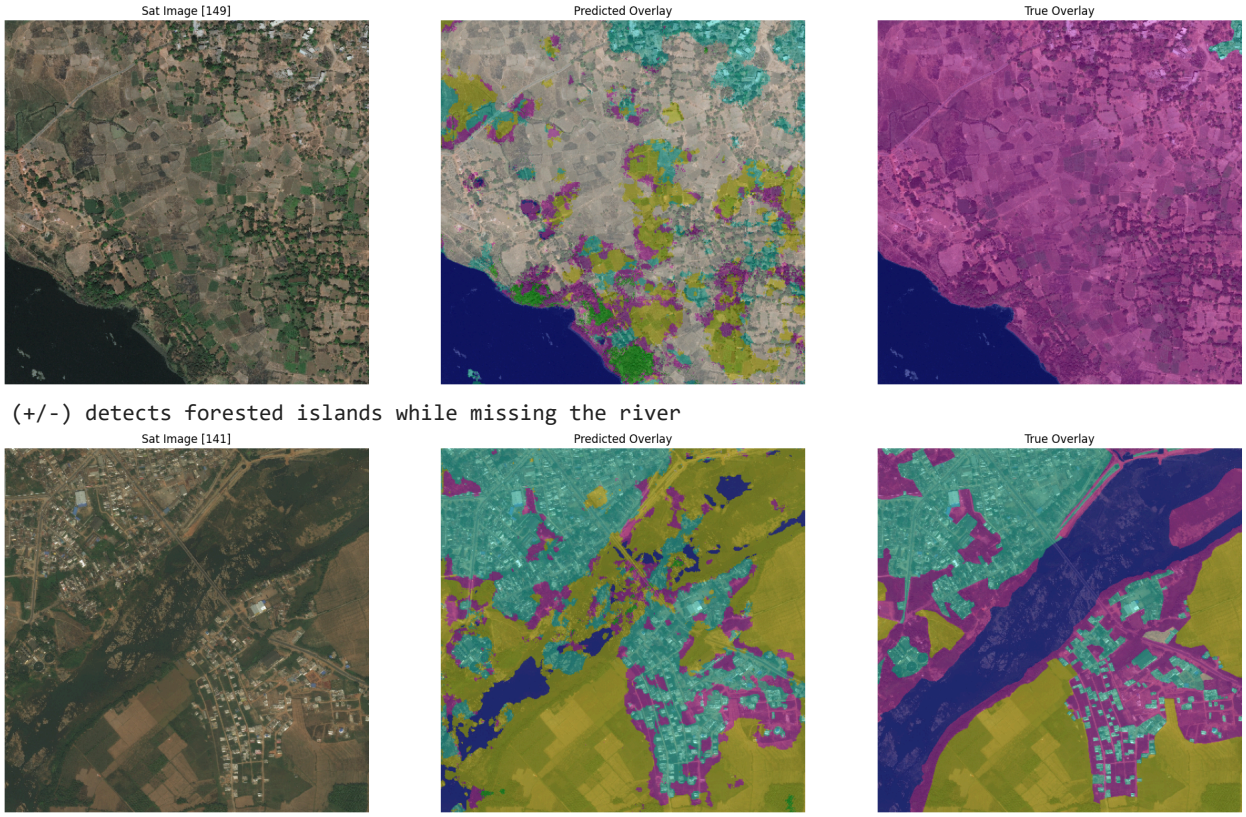
(-) confused with diverse water conditions



(-) confused with water segmentation in general



(+/-) over specificity/lack of detail in truth mask




```
(+/-) detects forested islands while missing the river
```


Reflections

Overall the model designed and trained in this notebook provide a promising proof of concept for an effective land cover classification model. Using 25% downsampled images and limited preprocessing/augmentation were able to achieve around 80% accuracy on our validation images.

Once proven, the model was expanded to incorporate random augmentations on the training data to increase the diversity of training images. While this new training approach did not make considerable progress in terms of accuracy, it smoothed the overall training/validation prediction significantly.

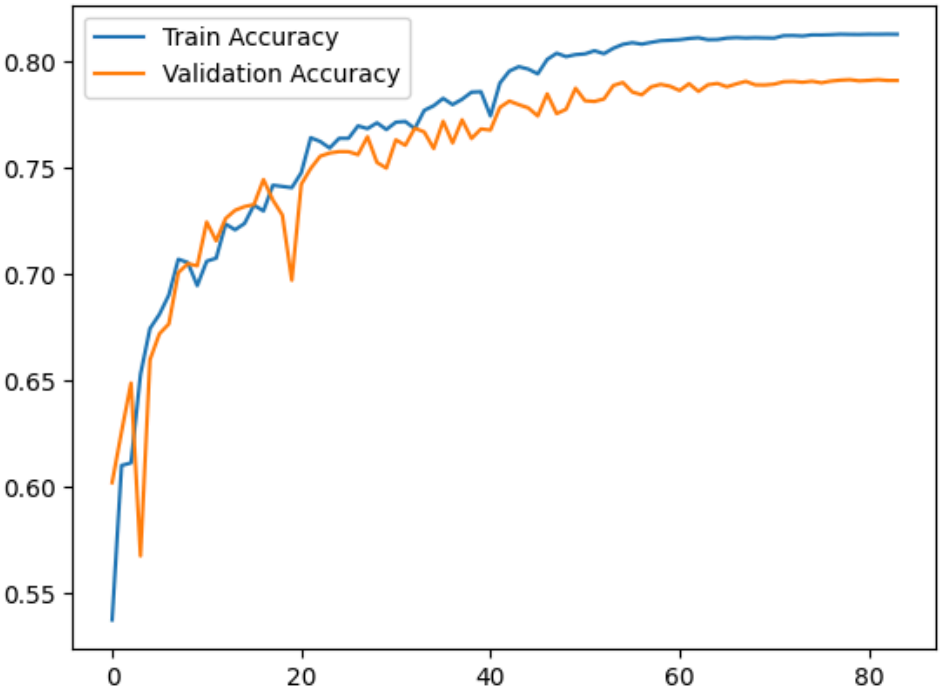
I suspect the impact of this approach is downplayed on our dataset that is relatively uniform across all image exposure, shape, hue, etc. I would argue that this new model would be significantly better suited at handling satellite imagery which exhibits more variability.

In [73]:
```
files.upload() # standard training...
files.upload() # augmented training...

print("Standard Model Training (no augmentation)")
display(Im('/content/performance.png'))

print("Augmented Model Training")
display(Im('/content/performance_aug.png'))
```

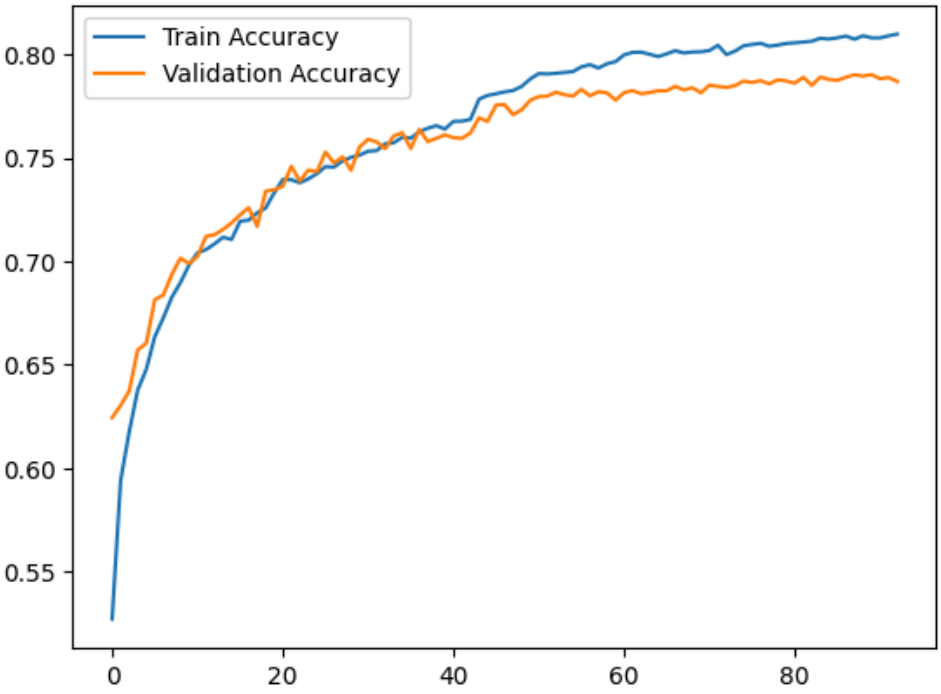
Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Standard Model Training (no augmentation)
```




```
Augmented Model Training
```



Prediction Analysis

Overall the model seemed to work fairly well on simple data, predicting urban land cover quite well. It struggled in a few understandable areas:

- Water Bounday Detection
- Urban/Forest Edge Detection

Comparing the model predictions to the truth masks, it is clear that the model struggles most with segmenting water correctly. Misclassifying situations where water was both present and absent. Also evident is the diverse spectrum of water color, clarity and hue.

The model also seems to frequently misclassify urban edge conditions to be range land, with a similar issue arising with forest edges and/or savannah conditions. The source defines **rangeland** to be "any green land that is not classified as forest or farmland". In other words, occupying the transition zones between urban/agriculture, urban/forest and agriculture/forest.

In both cases the error is reasonably justifiable given the flexible decision boundary between land cover types. Image augmentation and additional layers of data, such as LiDar, to enrich the visual data and ideally improve overall model performance.

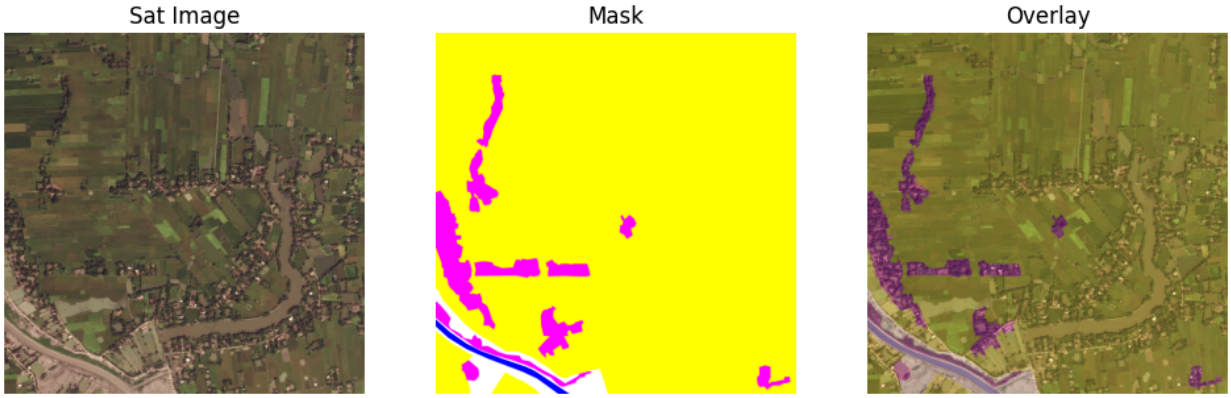
Training Error

In the process of hand selecting images for presentation, I encountered several cases of seemingly flawed land masking throughout the training and testing set. Many of the images found had errors with water detection and in many cases omitted many boundary cases that the trained model struggled to correctly identify.

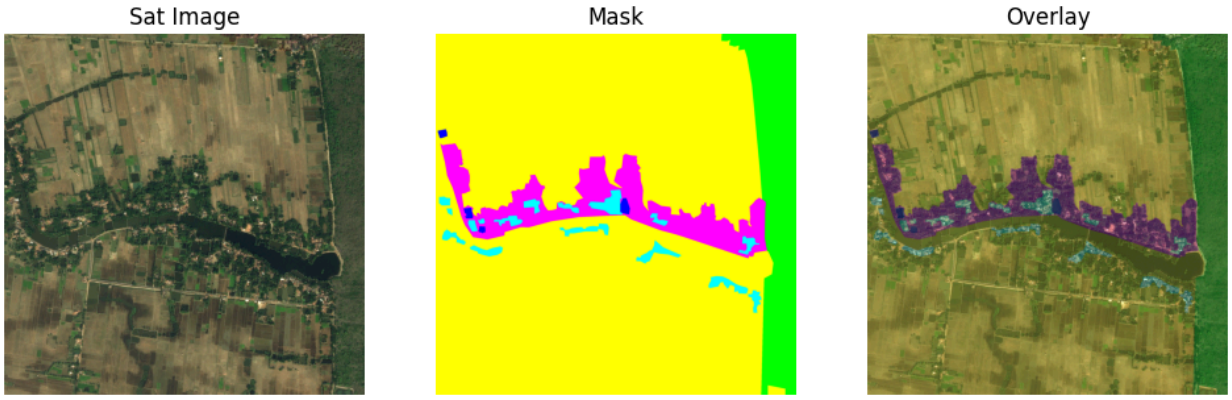
The original dataset's masks were generated by professional annotators who were able to leverage domain knowledge to outperform many automated processes, "particularly in distinguishing subtle variations in land cover types like rangelands and barren lands."

In [74]:
```
flagged_masks = [346,296,338,63,380,510]
flagged = {
  510: "Large river and human dwellings misclassified as agriculture",
  346: "Large river misclassified  as agriculture",
  338: "Human dwellings misclassified as rangeland",
  63: "Sinous river misclassified as agriculture",
  380: "Airport tarmac and adjacent barren land misclassified as agriculture",
  296: "Possible over generalization of forest/range land",
}

#for i in random.sample(range(len(train_data_paths)), 20):
for key,val in flagged.items():
  print(f"{val}")
  plot_overlay(index_img=key)
```

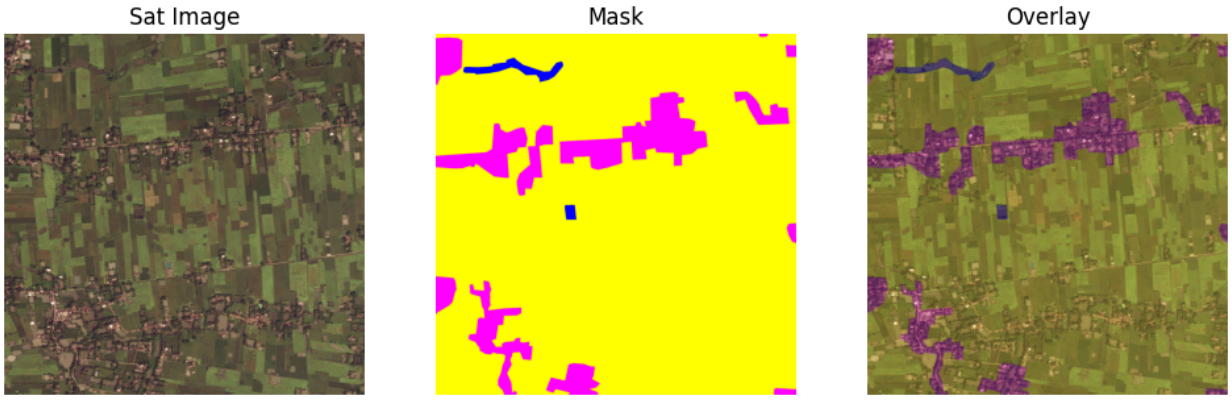
```
Large river and human dwellings misclassified as agriculture
```



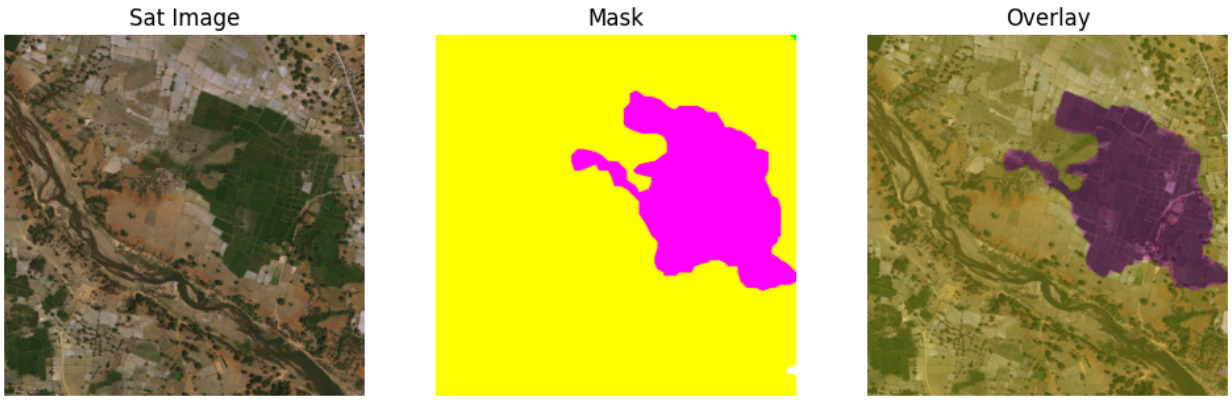

```
Large river misclassified  as agriculture
```



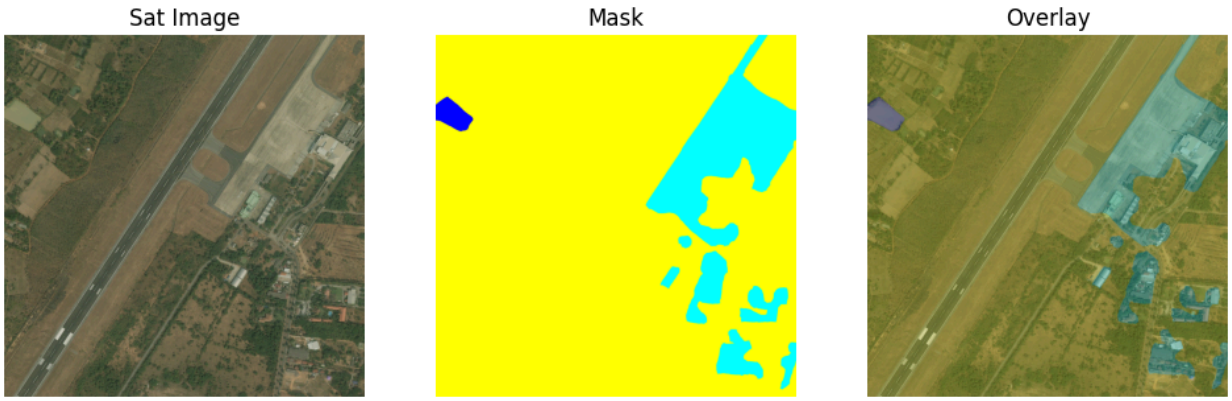
Human dwellings misclassified as rangeland



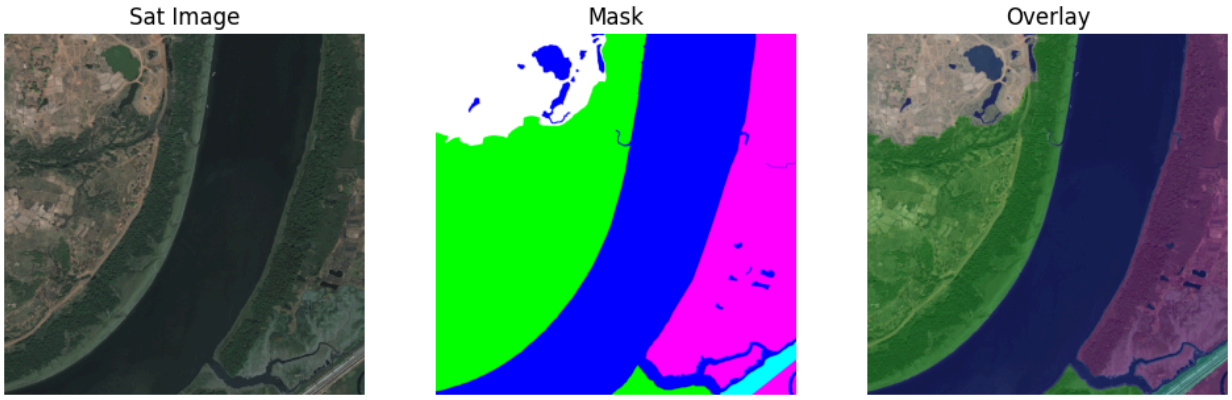
Sinous river misclassified as agriculture



Airport tarmac and adjacent barren land misclassified as agriculture



Possible over generalization of forest/range land



As eluded to during our EDA process, the training data can sometimes appear to misclassify and/or confuse land cover types. While the source asserts that the land covers were conducted by a professional to ensure greater accuracy than automated means, this also opens the door for human-error.

I am not debating the validity of the training data, nor is that part of the scope of this project. I mean to only shed light on classification differences between the dataset and my personal experience tapping into my extensive time spent practicing landscape architecture.

The dataset is small enough that I was hesitant to cull these samples from consideration, but the issues are prevalent enough for consideration in assessing our model's performance. Especially when one considers the areas of contention seem to correspond to the areas of confusion in the model's performance.

Outcome

In brief, the U-Net model exceeded my expectations in many aspects of this land cover segmentation task. The model proved reliably effective and classifying visually distinct land cover types, while expectedly struggling in areas that benefit most from domain familiarity.

More advanced logic and/or imaging tools (ie: LiDar) would provide a new depth of information that would target quantifiable differences between difficult to distinguish areas.