Notebook

December 15, 2024

0.0.1 CSCI 6364 - Machine Learning

Written by: Huru Algayeva

[31]:
```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
import rasterio
import requests
from bs4 import BeautifulSoup
import urllib.request
from tqdm import tqdm
from sklearn.metrics import f1_score
from urllib.parse import urljoin
import matplotlib.pyplot as plt
from tensorflow.keras.metrics import MeanIoU
from sklearn.metrics import precision_score, recall_score, f1_score
```

0.1 `DataLoader` Class Explanation

I created the `DataLoader` class to help me fetch satellite and map images from specific URLs. It organizes the files into folders and gives me the paths to use later. It's my tool for automating this whole process.

0.1.1 1. Initialization

When I set up the class, I give it: - A link to satellite images. - A link to map images. - A folder where I want everything to be saved.

If the folder doesn't exist, I make sure it gets created automatically. That way, I don't have to worry about missing directories or errors.

0.1.2 2. Getting TIFF File Links

I wrote a function to scan a webpage and pull out only the links that lead to `.tiff` or `.tif` files. If the links are incomplete, I fix them by adding the base part of the URL so they work properly.

This function makes sure I get all the TIFF files I need from the page.

0.1.3 3. Downloading TIFF Files

When I download a file, I decide where to save it by organizing it into folders. For example, satellite images go into one folder, and map images go into another.

Before downloading, I check if the file already exists. If it does, I skip downloading it again. Once the file is saved, I know exactly where it is, and I can use it later.

0.1.4 4. Loading the Dataset

I use this function to handle everything at once: 1. I first grab all the links for the satellite and map images. 2. I download the files, one by one, and watch the progress using a bar that shows how much is done. 3. At the end, I have two lists: one for satellite images and one for map images. They're sorted and ready for whatever I want to do next.

[24]:
```
class DataLoader:
    def __init__(self, sat_url, map_url, save_dir):
        self.sat_url = sat_url
        self.map_url = map_url
        self.save_dir = save_dir
        os.makedirs(save_dir, exist_ok=True)

    def get_tiff_links(self, url):
        """Get TIFF file links from the webpage."""
        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')
        tiff_links = [link.get('href') for link in soup.find_all('a')
                      if link.get('href', '').lower().endswith(('.tiff', '.
 ↪tif'))]
        base_url = '/'.join(url.split('/')[:-1])
        return [urljoin(base_url + '/', link) for link in tiff_links]

    def download_tiff(self, url, subdir):
        """Download a TIFF file."""
        save_path = os.path.join(self.save_dir, subdir, url.split('/')[-1])
        os.makedirs(os.path.dirname(save_path), exist_ok=True)

        if not os.path.exists(save_path):
            urllib.request.urlretrieve(url, save_path)
        return save_path

    def load_dataset(self):
        """Download and load both satellite and map images."""
        # Download images
```

```
        sat_links = self.get_tiff_links(self.sat_url)
        map_links = self.get_tiff_links(self.map_url)

        print("Downloading satellite images...")
        sat_paths = [self.download_tiff(url, 'sat') for url in tqdm(sat_links)]
        print("Downloading map images...")
        map_paths = [self.download_tiff(url, 'map') for url in tqdm(map_links)]

        return sorted(sat_paths), sorted(map_paths)
```

0.2 `TiffGenerator` Class

I use `TiffGenerator` to process satellite and map images in batches. It reads, preprocesses, and optionally augments the data, making it ready for training models. It integrates well with TensorFlow.

0.2.1 1. Initialization

I provide: - Image paths (`sat_paths`, `map_paths`), - Batch size, - Image size (resized to 384x384 by default), - An option for augmentation.

0.2.2 2. Batch Count

The class automatically calculates the number of batches based on the number of images and batch size.

0.2.3 3. Reading TIFF Files

I load TIFF files, rearrange dimensions, handle grayscale images, and resize them to my chosen size.

0.2.4 4. Augmentation

When enabled, I randomly flip images and masks horizontally or vertically to add variety and improve model generalization.

0.2.5 5. Getting Batches

For each batch: - I load and preprocess satellite images and map masks. - Normalize images (0 to 1) and binarize masks (threshold-based).

[25]:
```
class TiffGenerator(tf.keras.utils.Sequence):
    def __init__(self, sat_paths, map_paths, batch_size=8, img_size=(384, 384),␣
 ↪augment=False):
        self.sat_paths = sat_paths
        self.map_paths = map_paths
        self.batch_size = batch_size
        self.img_size = img_size
        self.augment = augment

    def __len__(self):
        return len(self.sat_paths) // self.batch_size

    def read_tiff(self, path):
        """Read and preprocess TIFF file."""
        with rasterio.open(path) as src:
            image = src.read()
            image = np.transpose(image, (1, 2, 0))
            # Converting to numpy array before resizing
            image = np.array(image)
            if len(image.shape) == 2:
                image = np.expand_dims(image, axis=-1)
            image = tf.image.resize(image, self.img_size)
            return image.numpy()  # Convert back to numpy array

    def augment_data(self, image, mask):
        """Apply data augmentation."""
        if tf.random.uniform([]) > 0.5:
            image = tf.image.flip_left_right(image)
            mask = tf.image.flip_left_right(mask)
        if tf.random.uniform([]) > 0.5:
            image = tf.image.flip_up_down(image)
            mask = tf.image.flip_up_down(mask)
        return image.numpy(), mask.numpy()  # Converting back to numpy arrays

    def __getitem__(self, idx):
        batch_sat_paths = self.sat_paths[idx * self.batch_size:(idx + 1) * self.
 ↪batch_size]
        batch_map_paths = self.map_paths[idx * self.batch_size:(idx + 1) * self.
 ↪batch_size]

        batch_images = []
        batch_masks = []

        for sat_path, map_path in zip(batch_sat_paths, batch_map_paths):
            # Load and preprocess images
            image = self.read_tiff(sat_path)
            image = image / 255.0  # Normalization part
```

```
            mask = self.read_tiff(map_path)
            mask = np.where(mask > 128, 1, 0).astype(np.float32)

            if self.augment:
                image, mask = self.augment_data(tf.convert_to_tensor(image), tf.
↪convert_to_tensor(mask))

            batch_images.append(image)
            batch_masks.append(mask)

        return np.array(batch_images), np.array(batch_masks)
```

[1]:
```
def create_unet(input_shape=(384, 384, 3)):
    """Creation of U-Net model architecture."""
    # I started with the input layer where I defined the input shape of the␣
↪model
    inputs = layers.Input(input_shape)

    # --- Encoder ---
    # I used two convolutional layers with ReLU activation to extract features
    conv1 = layers.Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = layers.Conv2D(64, 3, activation='relu', padding='same')(conv1)
    # I downsampled the features using MaxPooling to reduce spatial dimensions
    pool1 = layers.MaxPooling2D(pool_size=(2, 2))(conv1)

    # I repeated the process for the second block of the encoder
    conv2 = layers.Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv2 = layers.Conv2D(128, 3, activation='relu', padding='same')(conv2)
    pool2 = layers.MaxPooling2D(pool_size=(2, 2))(conv2)

    # I added another block, increasing the filters for more feature extraction
    conv3 = layers.Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv3 = layers.Conv2D(256, 3, activation='relu', padding='same')(conv3)
    pool3 = layers.MaxPooling2D(pool_size=(2, 2))(conv3)

    # --- Bridge ---
    # I created the bottleneck to connect the encoder and decoder
    # This layer captured the most complex features
    conv4 = layers.Conv2D(512, 3, activation='relu', padding='same')(pool3)
    conv4 = layers.Conv2D(512, 3, activation='relu', padding='same')(conv4)
    # I added a Dropout layer to prevent overfitting
    conv4 = layers.Dropout(0.5)(conv4)

    # --- Decoder ---
    # I started upsampling the features back to the original size
    # First, I used Conv2DTranspose to increase spatial dimensions
```

```
    up5 = layers.concatenate([layers.Conv2DTranspose(256, 2, strides=(2, 2),␣
↪padding='same')(conv4), conv3], axis=3)
    # I refined the features using convolutional layers
    conv5 = layers.Conv2D(256, 3, activation='relu', padding='same')(up5)
    conv5 = layers.Conv2D(256, 3, activation='relu', padding='same')(conv5)

    # I repeated the same process for the next upsampling block
    up6 = layers.concatenate([layers.Conv2DTranspose(128, 2, strides=(2, 2),␣
↪padding='same')(conv5), conv2], axis=3)
    conv6 = layers.Conv2D(128, 3, activation='relu', padding='same')(up6)
    conv6 = layers.Conv2D(128, 3, activation='relu', padding='same')(conv6)

    # I processed the final upsampling block
    up7 = layers.concatenate([layers.Conv2DTranspose(64, 2, strides=(2, 2),␣
↪padding='same')(conv6), conv1], axis=3)
    conv7 = layers.Conv2D(64, 3, activation='relu', padding='same')(up7)
    conv7 = layers.Conv2D(64, 3, activation='relu', padding='same')(conv7)

    # I created the output layer with a single filter and sigmoid activation
    # This ensured the output was a binary mask
    outputs = layers.Conv2D(1, 1, activation='sigmoid')(conv7)

    # I built the U-Net model by connecting inputs and outputs
    model = models.Model(inputs=inputs, outputs=outputs)
    return model
```

[27]:
```
def dice_loss(y_true, y_pred):
    """Calculation of Dice loss."""
    smooth = 1e-6
    y_true_f = tf.keras.backend.flatten(y_true)
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true_f * y_pred_f)
    return 1 - (2. * intersection + smooth) / (tf.keras.backend.sum(y_true_f) +␣
↪tf.keras.backend.sum(y_pred_f) + smooth)
```

[28]:
```
def combined_loss(y_true, y_pred):
    """Combining binary crossentropy and dice loss."""
    bce = tf.keras.losses.binary_crossentropy(y_true, y_pred)
    dice = dice_loss(y_true, y_pred)
    return bce + dice
```

[6]:
```
# Dataset URLs
train_urls = {
    'sat': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/train/sat/index.
↪html',
    'map': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/train/map/index.
↪html'
```

```
}
valid_urls = {
    'sat': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/valid/sat/index.
 ↪html',
    'map': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/valid/map/index.
 ↪html'
}
test_urls = {
    'sat': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/test/sat/index.
 ↪html',
    'map': 'https://www.cs.toronto.edu/~vmnih/data/mass_roads/test/map/index.
 ↪html'
}
```

[7]:
```
print("Loading training data...")
train_loader = DataLoader(train_urls['sat'], train_urls['map'], 'data/train')
train_sat_paths, train_map_paths = train_loader.load_dataset()

print("Loading validation data...")
valid_loader = DataLoader(valid_urls['sat'], valid_urls['map'], 'data/valid')
valid_sat_paths, valid_map_paths = valid_loader.load_dataset()
```

```
Loading training data…
Downloading satellite images…
100%|      | 1108/1108 [13:17<00:00,  1.39it/s]
Downloading map images…
100%|      | 1108/1108 [11:04<00:00,  1.67it/s]
Loading validation data…
Downloading satellite images…
100%|      | 14/14 [00:10<00:00,  1.34it/s]
Downloading map images…
100%|      | 14/14 [00:08<00:00,  1.73it/s]
```

[8]:
```
# Instead of loading all the data into memory at once (which can be huge for␣
 ↪satellite and map images),
# I used generators to load and preprocess the data in batches. This keeps␣
 ↪memory usage low and speeds up training.
train_gen = TiffGenerator(train_sat_paths, train_map_paths, batch_size=8,␣
 ↪augment=True)
valid_gen = TiffGenerator(valid_sat_paths, valid_map_paths, batch_size=8,␣
 ↪augment=False)
```

[9]:
```
model = create_unet()
model.compile(
    optimizer='adam',
    loss=combined_loss,
    metrics=['accuracy', tf.keras.metrics.IoU(num_classes=2,␣
 ↪target_class_ids=[1])]
)
```

[10]:
```
callbacks = [
    # I added a ModelCheckpoint callback to save the best model during training
    # It monitored the 'val_iou_score' metric and saved the model when it␣
 ↪improved
    tf.keras.callbacks.ModelCheckpoint(
        'best_model.keras',
        monitor='val_iou_score',  # The metric used for monitoring
        mode='max',  # I set this to save the model when the IoU score was␣
 ↪maximized
        save_best_only=True,
        verbose=1
    ),
    # I included an EarlyStopping callback to stop training if performance␣
 ↪stopped improving
    # It monitored 'val_iou_score' and restored the best weights after stopping
    tf.keras.callbacks.EarlyStopping(
        monitor='val_iou_score',  # The metric used for monitoring
        mode='max',  # I specified maximizing the IoU score
        patience=10,  # Training stopped if there were no improvements for 10␣
 ↪epochs
        restore_best_weights=True,
        verbose=1
    ),
    # I used ReduceLROnPlateau to reduce the learning rate when the loss␣
 ↪plateaued
    # It monitored 'val_loss' and decreased the learning rate by a factor of 0.
 ↪5 after 5 epochs without improvement
    tf.keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss',  # The metric used for monitoring
        factor=0.5,  # The factor by which the learning rate was reduced
        patience=5,  # Number of epochs without improvement before reduction
        min_lr=1e-6,  # The minimum learning rate
        mode='min',  # I set this to minimize the loss
        verbose=1
    )
]
```

[11]:
```
# Training model
history = model.fit(
    train_gen,
    validation_data=valid_gen,
    epochs=20,
    callbacks=callbacks)

# Save=ing the final model
model.save('final_model.keras')
```

```
/usr/local/lib/python3.10/dist-packages/rasterio/__init__.py:356:
NotGeoreferencedWarning: Dataset has no geotransform, gcps, or rpcs. The
identity matrix will be returned.
  dataset = DatasetReader(path, driver=driver, sharing=sharing, **kwargs)

Epoch 1/20

/usr/local/lib/python3.10/dist-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()

138/138             100s 457ms/step -
accuracy: 0.9140 - io_u: 0.0000e+00 - loss: 1.2581 - val_accuracy: 0.9257 -
val_io_u: 0.0000e+00 - val_loss: 1.0525 - learning_rate: 0.0010
Epoch 2/20

/usr/local/lib/python3.10/dist-
packages/keras/src/callbacks/model_checkpoint.py:206: UserWarning: Can save best
model only with val_iou_score available, skipping.
  self._save_model(epoch=epoch, batch=None, logs=logs)
/usr/local/lib/python3.10/dist-
packages/keras/src/callbacks/early_stopping.py:155: UserWarning: Early stopping
conditioned on metric `val_iou_score` which is not available. Available metrics
are: accuracy,io_u,loss,val_accuracy,val_io_u,val_loss
  current = self.get_monitor_value(logs)

138/138             60s 418ms/step -
accuracy: 0.9407 - io_u: 0.0000e+00 - loss: 1.0243 - val_accuracy: 0.8943 -
val_io_u: 0.0000e+00 - val_loss: 0.9702 - learning_rate: 0.0010
Epoch 3/20
138/138             60s 419ms/step -
accuracy: 0.9265 - io_u: 0.0000e+00 - loss: 0.9730 - val_accuracy: 0.8805 -
val_io_u: 0.0000e+00 - val_loss: 0.9643 - learning_rate: 0.0010
Epoch 4/20
138/138             60s 416ms/step -
accuracy: 0.9351 - io_u: 0.0000e+00 - loss: 0.9277 - val_accuracy: 0.9227 -
```

```
val_io_u: 0.0000e+00 - val_loss: 0.8901 - learning_rate: 0.0010
Epoch 5/20
138/138             60s 415ms/step -
accuracy: 0.9383 - io_u: 0.0000e+00 - loss: 0.8981 - val_accuracy: 0.9269 -
val_io_u: 0.0000e+00 - val_loss: 0.9667 - learning_rate: 0.0010
Epoch 6/20
138/138             60s 416ms/step -
accuracy: 0.9461 - io_u: 0.0000e+00 - loss: 0.8319 - val_accuracy: 0.9280 -
val_io_u: 0.0000e+00 - val_loss: 0.9603 - learning_rate: 0.0010
Epoch 7/20
138/138             60s 414ms/step -
accuracy: 0.9508 - io_u: 0.0000e+00 - loss: 0.7971 - val_accuracy: 0.9358 -
val_io_u: 0.0000e+00 - val_loss: 0.8815 - learning_rate: 0.0010
Epoch 8/20
138/138             61s 418ms/step -
accuracy: 0.9567 - io_u: 0.0000e+00 - loss: 0.7213 - val_accuracy: 0.9368 -
val_io_u: 0.0000e+00 - val_loss: 0.7113 - learning_rate: 0.0010
Epoch 9/20
138/138             60s 415ms/step -
accuracy: 0.9602 - io_u: 0.0000e+00 - loss: 0.6563 - val_accuracy: 0.9456 -
val_io_u: 0.0000e+00 - val_loss: 0.6476 - learning_rate: 0.0010
Epoch 10/20
138/138             60s 412ms/step -
accuracy: 0.9627 - io_u: 0.0000e+00 - loss: 0.6119 - val_accuracy: 0.9461 -
val_io_u: 0.0000e+00 - val_loss: 0.6576 - learning_rate: 0.0010
Epoch 11/20
138/138             60s 413ms/step -
accuracy: 0.9629 - io_u: 0.0000e+00 - loss: 0.6140 - val_accuracy: 0.9484 -
val_io_u: 0.0000e+00 - val_loss: 0.6275 - learning_rate: 0.0010
Epoch 12/20
138/138             60s 415ms/step -
accuracy: 0.9620 - io_u: 0.0000e+00 - loss: 0.5964 - val_accuracy: 0.9496 -
val_io_u: 0.0000e+00 - val_loss: 0.5561 - learning_rate: 0.0010
Epoch 13/20
138/138             60s 413ms/step -
accuracy: 0.9625 - io_u: 0.0000e+00 - loss: 0.5945 - val_accuracy: 0.9494 -
val_io_u: 0.0000e+00 - val_loss: 0.5957 - learning_rate: 0.0010
Epoch 14/20
138/138             59s 411ms/step -
accuracy: 0.9664 - io_u: 0.0000e+00 - loss: 0.5678 - val_accuracy: 0.9532 -
val_io_u: 0.0000e+00 - val_loss: 0.5601 - learning_rate: 0.0010
Epoch 15/20
138/138             60s 412ms/step -
accuracy: 0.9684 - io_u: 0.0000e+00 - loss: 0.5330 - val_accuracy: 0.9499 -
val_io_u: 0.0000e+00 - val_loss: 0.5413 - learning_rate: 0.0010
Epoch 16/20
138/138             60s 414ms/step -
accuracy: 0.9670 - io_u: 0.0000e+00 - loss: 0.5496 - val_accuracy: 0.9503 -
```

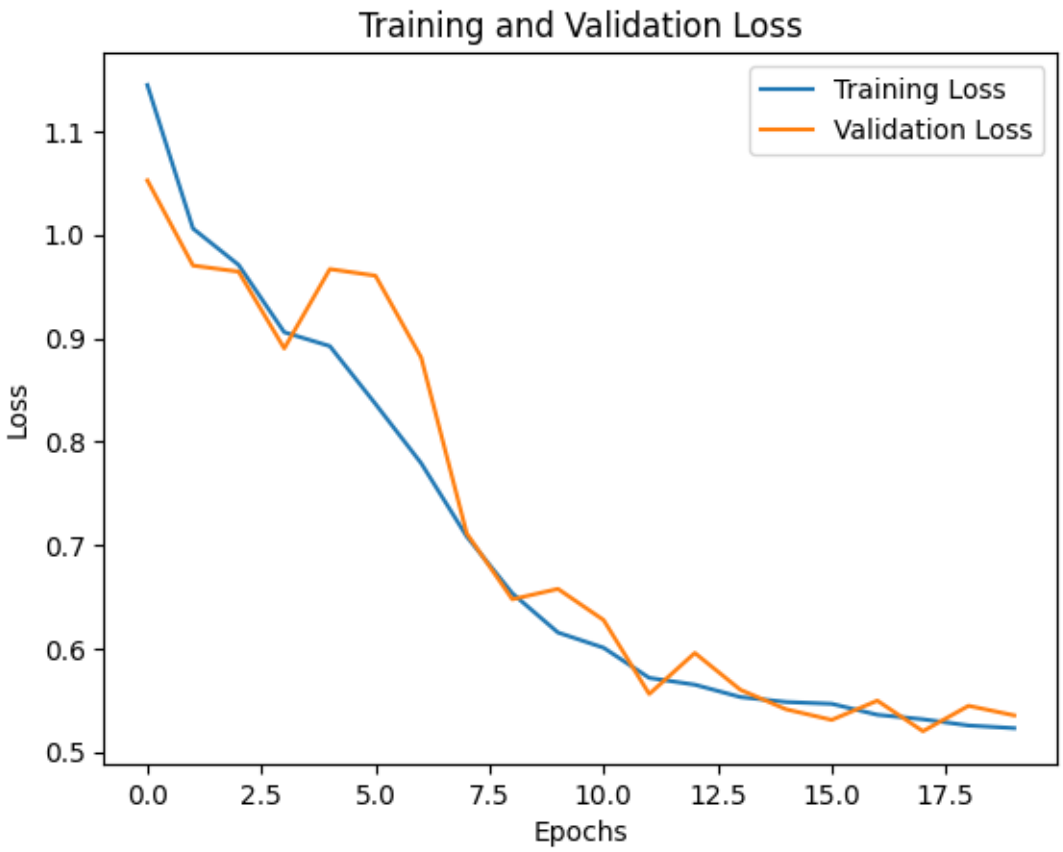
```
val_io_u: 0.0000e+00 - val_loss: 0.5310 - learning_rate: 0.0010
Epoch 17/20
138/138             60s 414ms/step -
accuracy: 0.9683 - io_u: 0.0000e+00 - loss: 0.5296 - val_accuracy: 0.9531 -
val_io_u: 0.0000e+00 - val_loss: 0.5496 - learning_rate: 0.0010
Epoch 18/20
138/138             60s 413ms/step -
accuracy: 0.9680 - io_u: 0.0000e+00 - loss: 0.5499 - val_accuracy: 0.9514 -
val_io_u: 0.0000e+00 - val_loss: 0.5201 - learning_rate: 0.0010
Epoch 19/20
138/138             60s 413ms/step -
accuracy: 0.9670 - io_u: 0.0000e+00 - loss: 0.5170 - val_accuracy: 0.9536 -
val_io_u: 0.0000e+00 - val_loss: 0.5447 - learning_rate: 0.0010
Epoch 20/20
138/138             60s 412ms/step -
accuracy: 0.9700 - io_u: 0.0000e+00 - loss: 0.5175 - val_accuracy: 0.9520 -
val_io_u: 0.0000e+00 - val_loss: 0.5354 - learning_rate: 0.0010
```

[21]:
```
print("\nLoading test data...")
test_loader = DataLoader(test_urls['sat'], test_urls['map'], 'data/test')
test_sat_paths, test_map_paths = test_loader.load_dataset()
test_gen = TiffGenerator(test_sat_paths, test_map_paths, batch_size=8,␣
 ↪augment=False)
```

```
Loading test data…
Downloading satellite images…
100%|      | 49/49 [00:00<00:00, 26795.42it/s]
Downloading map images…
100%|      | 49/49 [00:00<00:00, 58171.78it/s]
```

[13]:
```
def plot_loss(history):
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()
    plt.show()
```

[15]:
```
plot_loss(history)
```



[32]:
```
test_images, test_masks = next(iter(test_gen))

predictions = model.predict(test_images)
thresholded_preds = (predictions > 0.5).astype(np.uint8)

iou_metric = MeanIoU(num_classes=2)
iou_metric.update_state(test_masks.flatten(), thresholded_preds.flatten())
print(f"Test IoU: {iou_metric.result().numpy():.4f}")
```

```
/usr/local/lib/python3.10/dist-packages/rasterio/__init__.py:356:
NotGeoreferencedWarning: Dataset has no geotransform, gcps, or rpcs. The
identity matrix will be returned.
  dataset = DatasetReader(path, driver=driver, sharing=sharing, **kwargs)

1/1             0s 28ms/step
Test IoU: 0.7470
```

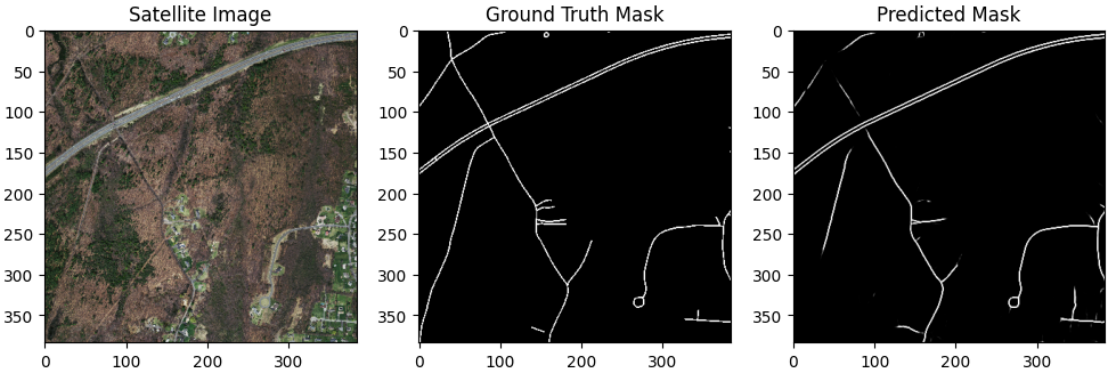
[23]:
```
y_true = test_masks.flatten()
y_pred = thresholded_preds.flatten()

precision = precision_score(y_true, y_pred)
```

```
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
```

```
Precision: 0.6609
Recall: 0.7100
F1 Score: 0.6846
```

[18]:
```
sat_img, map_gt = next(iter(test_gen))
map_pred = model.predict(sat_img)

plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.title("Satellite Image")
plt.imshow(sat_img[0].squeeze(), cmap='gray')
plt.subplot(1, 3, 2)
plt.title("Ground Truth Mask")
plt.imshow(map_gt[0].squeeze(), cmap='gray')
plt.subplot(1, 3, 3)
plt.title("Predicted Mask")
plt.imshow(map_pred[0].squeeze(), cmap='gray')
plt.show()
```

```
1/1             0s 454ms/step
```



This notebook was converted with convert.ploomber.io