Segmenting the Right-of-Way of San Francisco

Using VGG16-UNET on RGB Satellite Imagery

MUSA 650 Final Project – *Alexander Nelms*

Methods

Since I wanted to use image segmentation on a large RGB dataset, I needed to pick a model and architecture that will:

1. be specific enough to consistently detect urban design patterns
2. be generalized enough to not overfit on
   - *street-like* shapes
   - chaotic assortments of cars, trees, & such
   - potentially unclassified or incorrect RoW
3. not be too process intensive

The model and architecture I ended up using is an odd VGG16 and UNet hybrid that I found from a journal by Pravitasari et al. and in a related article by Gazali. Both are using this architecture to perform segmenetation of MRI scans to find brain tumors. Although the use case is different, the concept *and model* are similar.

In [ ]:
```python
from google.colab import drive
drive.mount('/content/drive')

import os
import pandas as pd
import json
import numpy as np
import dask.dataframe as dd

PROJ_DIR = 'drive/MyDrive/Penn/MUSA-650/FinalProject'

%cd {PROJ_DIR}

#DATA_DIR = PROJ_DIR + '/' + 'data'
DATA_DIR = 'data'

IMAGE_PATH = DATA_DIR + '/clean/' + 'sf_naip_images.parquet'
MASK_PATH = DATA_DIR + '/clean/' + 'sf_naip_mask.parquet'
INFO_PATH = DATA_DIR + '/clean/' + 'sf_naip_info.parquet'

!python -m pip install 'fsspec>=0.3.3'
```

In [ ]:
```python
# CLEAN DATAFRAMES AND TURN THEM TO ARRAYS
def df_to_array(arr_df):
  # import csv of image data
  ## pre-flattened
  cols = [col for col in list(arr_df) if 'Unnamed' not in col]
  arr_df = arr_df[cols]

  # list of shape index colnames
  icols = [col for col in list(arr_df) if 'ass' not in col]
  # make array out of rgb dataframe
  #image_arr = np.stack(arr_df[icols].apply(list, axis=1))
  image_arr = arr_df[icols].to_numpy(dtype='int')
  print('Shape: {}'.format(image_arr.shape))

  return image_arr

# UNFLATTEN FUNCTION
def unflatten(array, new_shape):
  # shape initial
  n_samples = array.shape[0]
  flat_shape = array.shape[1]

  new_shape = [n_samples] + list(new_shape)

  return array.reshape(new_shape)

# SHAPES
img_shape = (128, 128, 4)
mask_shape = (128, 128, 1)
rgb_shape = (128,128,3)

# IMPORT IMAGE ARRAYS
img_dd = dd.read_parquet(
             IMAGE_PATH,
             blocksize=1000000,
             sample=655360,
             dtype='int'
             )
img_arr = df_to_array(img_dd.compute(scheduler='processes'))

# IMPORT ROW MASKS
mask_arr = df_to_array(pd.read_parquet(MASK_PATH))

# IMPORT ADDITIONAL INFO
info_df = pd.read_parquet(INFO_PATH)
```

Train / Test Split

When preparing the data, I perform a 75 / 25 split on the data. I can afford 25% for training as I have a large dataset.

In [ ]:
```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential, Model
from tensorflow.keras.utils import plot_model
from tensorflow.keras.optimizers import Adam, RMSprop
import keras

# load vgg model
from keras.applications.vgg16 import VGG16, preprocess_input

# SPLIT GREY DATAFRAME INTO 50/50
## WITH STRATIFICATION ON THE CLASSES COL
X_train, X_test, y_train, y_test = train_test_split(
    img_arr, mask_arr,
    test_size=0.75, random_state=420
    )
## Scale the data
scalar = MinMaxScaler()
scalar.fit(X_train)
#X_train = scalar.transform(X_train)
#X_test = scalar.transform(X_test)
X_train = unflatten(X_train, img_shape)[:,:,:,:3]
#X_train = preprocess_input(X_train)
X_test = unflatten(X_test, img_shape)[:,:,:,:3]
#X_test = preprocess_input(X_test)
y_train = unflatten(y_train, mask_shape)
y_test = unflatten(y_test, mask_shape)

input_shape = X_train.shape[1:]
print('X train: {}'. format(X_train.shape))
print('Y train: {}'. format(y_train.shape))
```

```
X train: (4702, 128, 128, 3)
Y train: (4702, 128, 128, 1)
```

Model

As mentioned, I am basing my architecture off of an article by Gazali [source]. I found this architecture very impactfull as U-Net provides a helpful path for semantic segmentation. By narrowing the shape of the data then bringing it back out to the same shape, it is able to pool the connections into buckets. When the image is resized back to its original shape, it is then able to take with the classification for each pixel

Import VGG16 Model

In [ ]:
```python
# load the model
vgg16 = VGG16(
    include_top = False,
    input_shape = rgb_shape,
    weights='imagenet'
)
vgg16.summary()
```

In [ ]:
```python
import tensorflow as tf
from tensorflow.keras.layers import \
  Conv2D, Conv2DTranspose, MaxPooling2D, \
  Dense, Activation, Dropout, \
  Flatten, Input, Concatenate, \
  BatchNormalization
from tensorflow.keras.initializers import RandomNormal

def conv_block(inputs,num_filters):
  x = Conv2D(num_filters,3,padding='same')(inputs)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = Conv2D(num_filters,3,padding='same')(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  return x

def define_decoder(inputs,skip_layer,num_filters):
  init = RandomNormal(stddev=0.02)
  x = Conv2DTranspose(
        num_filters,(2,2),
        strides=(2,2),
        padding='same',
        kernel_initializer=init
      )(inputs)
  g = Concatenate()([x,skip_layer])
  g = conv_block(g, num_filters)
  return g

def vgg16_unet(input_shape):
  inputs = Input(shape=input_shape)
  vgg16 = VGG16(include_top=False,weights='imagenet',input_tensor=inputs)
  # We will extract encoder layers based on their output shape from vgg16 model
  s1 = vgg16.get_layer('block1_conv2').output
  s2 = vgg16.get_layer('block2_conv2').output
  s3 = vgg16.get_layer('block3_conv3').output
  s4 = vgg16.get_layer('block4_conv3').output    # bottleneck/bridege layer from vgg16
  b1 = vgg16.get_layer('block5_conv3').output #32

  # Decoder Block
  d1 = define_decoder(b1,s4,512)
  d2 = define_decoder(d1,s3,256)
  d3 = define_decoder(d2,s2,128)
  d4 = define_decoder(d3,s1,64)  #output layer
  outputs = Conv2D(1,1, padding='same', activation='sigmoid')(d4)
  model = Model(inputs,outputs)

  return model
```

Import Metrics

In [ ]:
```python
from tensorflow.math import reduce_sum

# Dice Loss
## a measure of overlap between two samples
## ranges from 0 to 1
## a Dice coefficient of 1 denotes perfect and complete overlap
smooth = 1e-15
def dice_coef(y_true,y_pred):
  y_true = Flatten()(y_true)
  y_pred = Flatten()(y_pred)
  intersection = reduce_sum(y_true*y_pred)
  return (2. * intersection + smooth) / (reduce_sum(y_true) + reduce_sum(y_pred))

def dice_loss(y_true,y_pred):
  return 1.0 - dice_coef(y_true,y_pred)

# Intersection-Over-Union
## a common evaluation metric for semantic image segmentation
## true_positive / (true_positive + false_positive + false_negative)
def iou(y_true,y_pred):
  def f(y_true,y_pred):
    intersection = (y_true*y_pred).sum()
    union = y_true.sum() + y_pred.sum() - intersection
    x = (intersection + 1e-15) / (union + 1e-15)
    x = x.astype(np.float32)
    return x
  return tf.numpy_function(f,[y_true,y_pred],tf.float32)
```

Compile Model

In [ ]:
```python
from tensorflow.keras.metrics import Precision, Recall

model = vgg16_unet(rgb_shape)
model.compile(
    loss=dice_loss,
    optimizer=Adam(lr = .001),
    metrics=[dice_coef, iou, Recall(), Precision()]
    )
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105: UserWarning: The `lr` argument is deprec
ated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

In [45]:
```python
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 128, 128, 3  0           []
                                )]

 block1_conv1 (Conv2D)          (None, 128, 128, 64  1792        ['input_2[0][0]']
                                )

 block1_conv2 (Conv2D)          (None, 128, 128, 64  36928       ['block1_conv1[0][0]']
                                )

 block1_pool (MaxPooling2D)     (None, 64, 64, 64)   0           ['block1_conv2[0][0]']

 block2_conv1 (Conv2D)          (None, 64, 64, 128)  73856       ['block1_pool[0][0]']

 block2_conv2 (Conv2D)          (None, 64, 64, 128)  147584      ['block2_conv1[0][0]']

 block2_pool (MaxPooling2D)     (None, 32, 32, 128)  0           ['block2_conv2[0][0]']

 block3_conv1 (Conv2D)          (None, 32, 32, 256)  295168      ['block2_pool[0][0]']

 block3_conv2 (Conv2D)          (None, 32, 32, 256)  590080      ['block3_conv1[0][0]']

 block3_conv3 (Conv2D)          (None, 32, 32, 256)  590080      ['block3_conv2[0][0]']

 block3_pool (MaxPooling2D)     (None, 16, 16, 256)  0           ['block3_conv3[0][0]']

 block4_conv1 (Conv2D)          (None, 16, 16, 512)  1180160     ['block3_pool[0][0]']

 block4_conv2 (Conv2D)          (None, 16, 16, 512)  2359808     ['block4_conv1[0][0]']

 block4_conv3 (Conv2D)          (None, 16, 16, 512)  2359808     ['block4_conv2[0][0]']

 block4_pool (MaxPooling2D)     (None, 8, 8, 512)    0           ['block4_conv3[0][0]']

 block5_conv1 (Conv2D)          (None, 8, 8, 512)    2359808     ['block4_pool[0][0]']

 block5_conv2 (Conv2D)          (None, 8, 8, 512)    2359808     ['block5_conv1[0][0]']

 block5_conv3 (Conv2D)          (None, 8, 8, 512)    2359808     ['block5_conv2[0][0]']

 conv2d_transpose (Conv2DTransp  (None, 16, 16, 512)  1049088    ['block5_conv3[0][0]']
 ose)

 concatenate (Concatenate)      (None, 16, 16, 1024  0           ['conv2d_transpose[0][0]',
                                )                                 'block4_conv3[0][0]']

 conv2d (Conv2D)                (None, 16, 16, 512)  4719104     ['concatenate[0][0]']

 batch_normalization (BatchNorm  (None, 16, 16, 512)  2048       ['conv2d[0][0]']
 alization)

 activation (Activation)        (None, 16, 16, 512)  0           ['batch_normalization[0][0]']

 conv2d_1 (Conv2D)              (None, 16, 16, 512)  2359808     ['activation[0][0]']

 batch_normalization_1 (BatchNo  (None, 16, 16, 512)  2048       ['conv2d_1[0][0]']
 rmalization)

 activation_1 (Activation)      (None, 16, 16, 512)  0           ['batch_normalization_1[0][0]']

 conv2d_transpose_1 (Conv2DTran  (None, 32, 32, 256)  524544     ['activation_1[0][0]']
 spose)

 concatenate_1 (Concatenate)    (None, 32, 32, 512)  0           ['conv2d_transpose_1[0][0]',
                                                                  'block3_conv3[0][0]']

 conv2d_2 (Conv2D)              (None, 32, 32, 256)  1179904     ['concatenate_1[0][0]']

 batch_normalization_2 (BatchNo  (None, 32, 32, 256)  1024       ['conv2d_2[0][0]']
 rmalization)

 activation_2 (Activation)      (None, 32, 32, 256)  0           ['batch_normalization_2[0][0]']

 conv2d_3 (Conv2D)              (None, 32, 32, 256)  590080      ['activation_2[0][0]']

 batch_normalization_3 (BatchNo  (None, 32, 32, 256)  1024       ['conv2d_3[0][0]']
 rmalization)

 activation_3 (Activation)      (None, 32, 32, 256)  0           ['batch_normalization_3[0][0]']

 conv2d_transpose_2 (Conv2DTran  (None, 64, 64, 128)  131200     ['activation_3[0][0]']
 spose)

 concatenate_2 (Concatenate)    (None, 64, 64, 256)  0           ['conv2d_transpose_2[0][0]',
                                                                  'block2_conv2[0][0]']

 conv2d_4 (Conv2D)              (None, 64, 64, 128)  295040      ['concatenate_2[0][0]']

 batch_normalization_4 (BatchNo  (None, 64, 64, 128)  512        ['conv2d_4[0][0]']
 rmalization)

 activation_4 (Activation)      (None, 64, 64, 128)  0           ['batch_normalization_4[0][0]']

 conv2d_5 (Conv2D)              (None, 64, 64, 128)  147584      ['activation_4[0][0]']

 batch_normalization_5 (BatchNo  (None, 64, 64, 128)  512        ['conv2d_5[0][0]']
 rmalization)

 activation_5 (Activation)      (None, 64, 64, 128)  0           ['batch_normalization_5[0][0]']

 conv2d_transpose_3 (Conv2DTran  (None, 128, 128, 64  32832      ['activation_5[0][0]']
 spose)                         )

 concatenate_3 (Concatenate)    (None, 128, 128, 12  0           ['conv2d_transpose_3[0][0]',
                                8)                                'block1_conv2[0][0]']

 conv2d_6 (Conv2D)              (None, 128, 128, 64  73792       ['concatenate_3[0][0]']
                                )

 batch_normalization_6 (BatchNo  (None, 128, 128, 64  256        ['conv2d_6[0][0]']
 rmalization)                   )

 activation_6 (Activation)      (None, 128, 128, 64  0           ['batch_normalization_6[0][0]']
                                )

 conv2d_7 (Conv2D)              (None, 128, 128, 64  36928       ['activation_6[0][0]']
                                )

 batch_normalization_7 (BatchNo  (None, 128, 128, 64  256        ['conv2d_7[0][0]']
 rmalization)                   )

 activation_7 (Activation)      (None, 128, 128, 64  0           ['batch_normalization_7[0][0]']
                                )

 conv2d_8 (Conv2D)              (None, 128, 128, 1)  65          ['activation_7[0][0]']

==================================================================================================
Total params: 25,862,337
Trainable params: 25,858,497
Non-trainable params: 3,840
__________________________________________________________________________________________________
```

Fit Model

In [ ]:
```python
tf.config.run_functions_eagerly(True)
```

In [ ]:
```python
EPOCHS = 25
#train_steps = len(X_train)//batch_size
#test_steps = len(X_test)//batch_size

history = model.fit(
  X_train, tf.cast(y_train, tf.float32),
  epochs=EPOCHS,
  validation_data=(X_test, tf.cast(y_test, tf.float32))#,
  #steps_per_epoch=train_steps,
  #validation_steps=test_steps
  )

# convert the history.history dict to a pandas DataFrame:
```

```
Epoch 1/25
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/structured_function.py:265: UserWarning: Even
though the `tf.config.experimental_run_functions_eagerly` option is set, this option does not apply to tf.data
functions. To force eager execution of tf.data functions, please use `tf.data.experimental.enable_debug_mode()
`.
  "Even though the `tf.config.experimental_run_functions_eagerly` "
147/147 [==============================] - 4432s 30s/step - loss: 0.2445 - dice_coef: 0.7555 - iou: 0.6095 - re
call: 0.7806 - precision: 0.7519 - val_loss: 0.2305 - val_dice_coef: 0.7695 - val_iou: 0.6267 - val_recall: 0.7
769 - val_precision: 0.7932
Epoch 2/25
147/147 [==============================] - 4356s 30s/step - loss: 0.1956 - dice_coef: 0.8044 - iou: 0.6744 - re
call: 0.8102 - precision: 0.8070 - val_loss: 0.2085 - val_dice_coef: 0.7915 - val_iou: 0.6565 - val_recall: 0.7
171 - val_precision: 0.8895
Epoch 3/25
147/147 [==============================] - 4371s 30s/step - loss: 0.1679 - dice_coef: 0.8321 - iou: 0.7138 - re
call: 0.8293 - precision: 0.8399 - val_loss: 0.2650 - val_dice_coef: 0.7350 - val_iou: 0.5825 - val_recall: 0.7
972 - val_precision: 0.6862
Epoch 4/25
147/147 [==============================] - 4364s 30s/step - loss: 0.1549 - dice_coef: 0.8452 - iou: 0.7333 - re
call: 0.8379 - precision: 0.8568 - val_loss: 0.1587 - val_dice_coef: 0.8413 - val_iou: 0.7273 - val_recall: 0.8
707 - val_precision: 0.8171
Epoch 5/25
147/147 [==============================] - 4375s 30s/step - loss: 0.1415 - dice_coef: 0.8585 - iou: 0.7533 - re
call: 0.8518 - precision: 0.8683 - val_loss: 0.1509 - val_dice_coef: 0.8491 - val_iou: 0.7390 - val_recall: 0.8
249 - val_precision: 0.8779
Epoch 6/25
147/147 [==============================] - 4359s 30s/step - loss: 0.1326 - dice_coef: 0.8674 - iou: 0.7668 - re
call: 0.8569 - precision: 0.8805 - val_loss: 0.1719 - val_dice_coef: 0.8281 - val_iou: 0.7085 - val_recall: 0.7
606 - val_precision: 0.9123
Epoch 7/25
147/147 [==============================] - 4357s 30s/step - loss: 0.1257 - dice_coef: 0.8743 - iou: 0.7778 - re
call: 0.8650 - precision: 0.8859 - val_loss: 0.1410 - val_dice_coef: 0.8590 - val_iou: 0.7542 - val_recall: 0.8
116 - val_precision: 0.9149
Epoch 8/25
147/147 [==============================] - 4358s 30s/step - loss: 0.1252 - dice_coef: 0.8747 - iou: 0.7787 - re
call: 0.8652 - precision: 0.8887 - val_loss: 0.1368 - val_dice_coef: 0.8632 - val_iou: 0.7606 - val_recall: 0.8
405 - val_precision: 0.8896
Epoch 9/25
 58/147 [==========>...................] - ETA: 26:17 - loss: 0.1217 - dice_coef: 0.8783 - iou: 0.7838 - recal
l: 0.8728 - precision: 0.8867
```

In [ ]:
```python
import datetime

now = datetime.datetime.now()
dt_string = now.strftime("%Y%m%d_%H%M")

model_path = os.path.join(DATA_DIR, 'models', 'vgg16_model_{}.keras'.format(dt_string))
model.save(model_path)

hist_df = pd.DataFrame(history.history)
history_path = os.path.join(DATA_DIR, 'models', 'vgg16_history_{}.csv'.format(dt_string))
with open(history_path, mode='w') as f:
    hist_df.to_csv(f)
```

Results

Accuracy

Instead of accuracy, I used two different metrics: A Dice Coefficient and an Intersection over Union.

Dice Loss is able to measure the overlap between two samples. In my case, that's the observed Right-of-Way Mask/Training set and my predicted RoW. The coefficient closer to one means there is a perfect overlap.

Looking at the middle plot of the figure below, the testing Dice Cofficent (blue) was able to rise to .85 by the 5th epoch. Which is amazing. The Validation was more inconsistent, but eventually rose to .9. The inconsistency could mean that I don't have that generalized of a model. I could use a dropout layer to add more variety to my dataset.

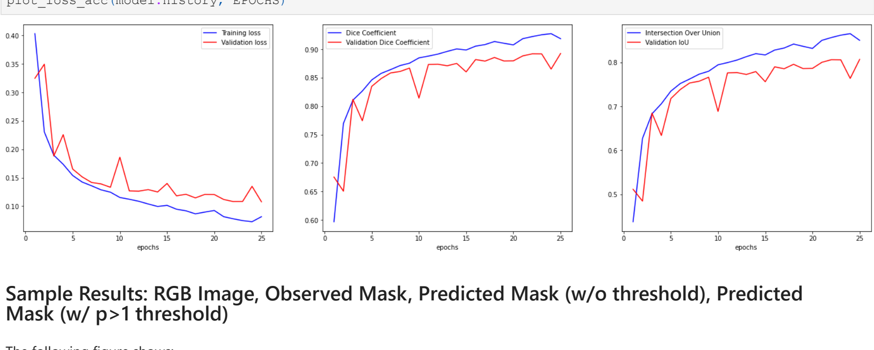
Intersection-Over-Union is a common evaluation metric for semantic image segmentation. It also measures the overlap but uses bounding boxes instead.

It is helpful to look at both the Dice Loss and the IoU as they give different assessments. The Dice Loss primarily looks at the overlap, regardless of the shape. But might not be the best at getting a larger context of the shape. The IoU metric can help understand the larger positioning.

The IoU metric, based on the figure below, took longer to rise than the dice coefficient. This is likely due to a number of reasons. *Missing* RoW polygon masks could lower the metrics as the model will still predict a road. The IoU metric will be affected more by this missing mask as it compares more area.

Regardless, both metrics got fairly close to .9, which suggests that the model is fairly accurate.

In [39]:
```python
def plot_loss_acc(fitted_model, num_epoch = EPOCHS):
  import matplotlib.pyplot as plt
  fig, ax = plt.subplots(1,3, figsize=[24,6])
  ax[0].plot(range(1, num_epoch+1), fitted_model.history['loss'], c='blue', label='Training loss')
  ax[0].plot(range(1, num_epoch+1), fitted_model.history['val_loss'], c='red', label='Validation loss')
  ax[0].legend()
  ax[0].set_xlabel('epochs')

  ax[1].plot(range(1, num_epoch+1), fitted_model.history['dice_coef'], c='blue', label='Dice Coefficient')
  ax[1].plot(range(1, num_epoch+1), fitted_model.history['val_dice_coef'], c='red', label='Validation Dice Coef
  ax[1].legend()
  ax[1].set_xlabel('epochs')

  ax[2].plot(range(1, num_epoch+1), fitted_model.history['iou'], c='blue', label='Intersection Over Union')
  ax[2].plot(range(1, num_epoch+1), fitted_model.history['val_iou'], c='red', label='Validation IoU')
  ax[2].legend()
  ax[2].set_xlabel('epochs')

plot_loss_acc(model.history, EPOCHS)
```



Sample Results: RGB Image, Observed Mask, Predicted Mask (w/o threshold), Predicted Mask (w/ p>1 threshold)

The following figure shows:

1. the observed satellite image,
2. the observed RoW mask
3. the Predicted Mask
4. the Predicted Mask (probabilites > 1)

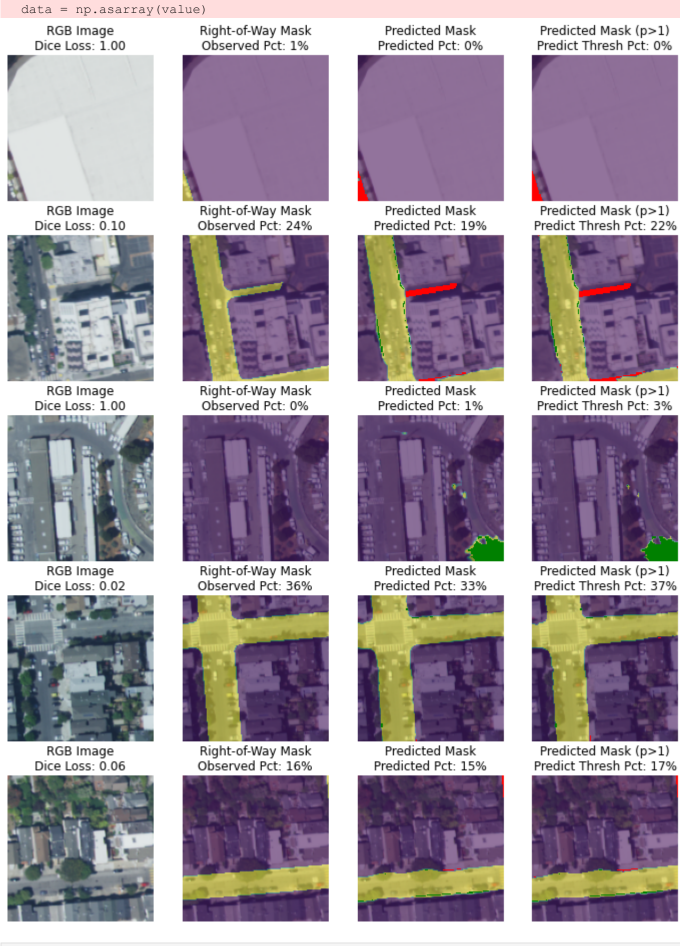
Yellow Areas means that the predicted results *did* predict the mask area

Red areas means that the predicted results *didn't* predict the mask area.

Green areas means that the predicted results predicted *too much* area

In [41]:
```python
import matplotlib.pyplot as plt
import random
from skimage.io import imread
import matplotlib
from matplotlib import colors

# counts array equal to focus value
def count_focus(array, value=1):
    return len(array[array==value])
# get percent that the array is equal to value
## used for filter masks
def get_pct_arr(focus_array, value=1):
    return np.array([count_focus(arr, value=value)/arr.size for arr in focus_array])

smooth = 1e-15
def dice_coef_arr(y_true,y_pred):
  y_true = np.array(y_true).flatten()
  y_pred = np.array(y_pred).flatten()
  intersection = reduce_sum(y_true*y_pred)
  return (2. * intersection + smooth) / (reduce_sum(y_true) + reduce_sum(y_pred))

def dice_loss_arr(y_true,y_pred):
  return 1.0 - dice_coef(y_true,y_pred)

def arr_to_binary(arr, cutoff=1.0, buffer=.01, good=1, bad=0):
  arr[arr>=cutoff-buffer] = good
  arr[arr<cutoff-buffer] = bad
  return arr

def get_masked_cmap(arr, thresh = 1.0, cmap='red'):
  masked_array = np.ma.masked_where(arr < thresh, arr)
  cmap = colors.ListedColormap([cmap])
  return masked_array, cmap

cols, rows = 4, 5
axes = []
[[axes.append((c,r)) for r in range(rows)] for c  in range(cols)]
axis_count = len(axes)
fig, ax = plt.subplots(rows, cols, figsize=[12, 16])

for row, idx in enumerate(random.sample(range(len(X_test)), rows)):

    focus_image, observed_mask = X_test[idx:idx+1,:,:,:], y_test[idx:idx+1,:,:,:]

    predicted_mask = model.predict(tf.cast(focus_image, tf.float32))

    predicted_mask_thresh = arr_to_binary(predicted_mask.copy(), buffer=0.01)
    dice_loss = dice_loss_arr(tf.cast(observed_mask, tf.float32), predicted_mask_thresh)

    pct_observed = get_pct_arr(observed_mask)[0]
    pct_predicted = get_pct_arr(predicted_mask)[0]
    pct_predicted_thresh = get_pct_arr(predicted_mask_thresh)[0]

    ax[row, 0].imshow(focus_image[0])
    ax[row, 0].set_title('RGB Image' + '\n' + 'Dice Loss: {:.2f}'.format(dice_loss))

    ax[row, 1].imshow(focus_image[0])
    ax[row, 1].imshow(observed_mask.reshape(128,128), alpha=.5)
    ax[row, 1].set_title('Right-of-Way Mask' + '\n' + 'Observed Pct: {0:.0%}'.format(pct_observed))



    ax[row, 2].imshow(focus_image[0])
    ax[row, 2].imshow(predicted_mask.reshape(128,128), alpha=.5)

    diff_mask = arr_to_binary(predicted_mask - observed_mask, buffer=0.01, bad=np.nan).reshape(128,128)
    diff_mask, cmap = get_masked_cmap(diff_mask, cmap='green')
    ax[row, 2].imshow(diff_mask, cmap=cmap)
    diff_mask  = arr_to_binary(observed_mask - predicted_mask, buffer=0.01, bad=np.nan).reshape(128,128)
    diff_mask, cmap = get_masked_cmap(diff_mask, cmap='red')
    ax[row, 2].imshow(diff_mask, cmap=cmap)
    #ax[row, 2].set_title('Pct Roads: {0:.1%}'.format(pct_roads) + '\n' + 'Pct Diff: {0:.1%}'.format(pct_diff))
    ax[row, 2].set_title('Predicted Mask' + '\n' + 'Predicted Pct: {0:.0%}'.format(pct_predicted))



    ax[row, 3].imshow(focus_image[0])
    ax[row, 3].imshow(predicted_mask_thresh.reshape(128,128), alpha=.5)

    diff_mask = arr_to_binary(predicted_mask_thresh - observed_mask, buffer=0.01, bad=np.nan).reshape(128,128)
    diff_mask, cmap = get_masked_cmap(diff_mask, cmap='green')
    ax[row, 3].imshow(diff_mask, cmap=cmap)
    diff_mask  = arr_to_binary(observed_mask - predicted_mask_thresh, buffer=0.01, bad=np.nan).reshape(128,128)
    diff_mask, cmap = get_masked_cmap(diff_mask, cmap='red')
    ax[row, 3].imshow(diff_mask, cmap=cmap)
    #ax[row, 3].imshow(predicted_mask_thresh.reshape(128,128), alpha=.5)
    ax[row, 3].set_title('Predicted Mask (p>1)' + '\n' + 'Predict Thresh Pct: {0:.0%}'.format(pct_predicted_thr

    for col in range(cols):
        ax[row, col].set_axis_off()
fig.patch.set_facecolor('#FFFFFF')
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/structured_function.py:265: UserWarning: Even
though the `tf.config.experimental_run_functions_eagerly` option is set, this option does not apply to tf.data
functions. To force eager execution of tf.data functions, please use `tf.data.experimental.enable_debug_mode()
`.
  "Even though the `tf.config.experimental_run_functions_eagerly` "
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:452: UserWarning: Warning: converting a masked eleme
nt to nan.
  dv = np.float64(self.norm.vmax) - np.float64(self.norm.vmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:459: UserWarning: Warning: converting a masked eleme
nt to nan.
  a_min = np.float64(newmin)
/usr/local/lib/python3.7/dist-packages/matplotlib/image.py:464: UserWarning: Warning: converting a masked eleme
nt to nan.
  a_max = np.float64(newmax)
<string>:6: UserWarning: Warning: converting a masked element to nan.
/usr/local/lib/python3.7/dist-packages/matplotlib/colors.py:993: UserWarning: Warning: converting a masked elem
ent to nan.
  data = np.asarray(value)
```



In [ ]:
```python
arr = arr_to_binary(observed_mask - predicted_mask_thresh, buffer=0, bad=0).reshape(128,128)
thresh=1
np.ma.masked_where(arr < thresh, arr)
```

In [ ]:
```python
diff_mask[~np.isnan(diff_mask)]
```

In [ ]:
```python
smooth = 1e-15
def dice_coef(y_true,y_pred):
  y_true = np.array(y_true).flatten()
  y_pred = np.array(y_pred).flatten()
  intersection = reduce_sum(y_true*y_pred)
  return (2. * intersection + smooth) / (reduce_sum(y_true) + reduce_sum(y_pred))

def dice_loss(y_true,y_pred):
  return 1.0 - dice_coef(y_true,y_pred)

dice_loss(tf.cast(focus_image, tf.float32), predicted_image)
```

In [ ]:

In [ ]:

Discuss

Overall, I am fairly happy with the results. The metrics suggest that my model was fairly accurate. But the figure samples show that the predictions aren't fine-tuned. Although the predictions are in the same ball park, they won't have clean shapes

Things I'd change

Returning to the issue of 256x256 vs 128x128, I wish I had chosen the larger images as it could have improved the accuracy while removing some samples with the viewable right-of-way cut by the crop.

In general, I feel that most of the inaccuracy comes from the irregularities of the Right-of-Way polygon dataset. Since the polygons don't follow a consistent set of rules when defining the RoW, the model will also be inconsistent when defining sidewalks, parking lots, and other concrete surfaces.