In [ ]:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.patches as mpatches
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure


plt.style.use('ggplot')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (12, 10)
plt.rcParams['axes.grid']=True
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green',
'xkcd:scarlet']


class Obstacle:
    FLAT, ROCK, HILL, MOUNTAIN = range(4)

def load_image(file_path):
    image = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
    return image

def tile_image(image, tile_size):
    rows, cols = image.shape
    tile_rows = rows // tile_size
    tile_cols = cols // tile_size

    tiles = []
    for i in range(tile_rows):
        for j in range(tile_cols):
            tile = image[i*tile_size:(i+1)*tile_size, j*tile_size:(j+1)*tile_size]
            tiles.append(tile)
    return tiles, tile_rows, tile_cols

def calculate_tile_statistics(tile):
    stddev_height = np.std(tile)
    mean_height = np.mean(tile)
    return stddev_height, mean_height
```

```python
def classify_tile(stddev, mean, thresholds):
    if stddev < thresholds['rock_stddev'] and mean < thresholds['rock_mean']:
        return Obstacle.FLAT
    elif thresholds['rock_stddev'] <= stddev < thresholds['hill_stddev']:
        return Obstacle.ROCK
    elif thresholds['hill_stddev'] <= stddev < thresholds['mountain_stddev']:
        return Obstacle.HILL
    else:
        return Obstacle.MOUNTAIN

def create_colored_overlay(image, tile_size, thresholds):
    tiles, tile_rows, tile_cols = tile_image(image, tile_size)
    overlay = np.zeros((image.shape[0], image.shape[1], 3), dtype=np.uint8)

    colormap = {
        Obstacle.FLAT: (0, 255, 0),
        Obstacle.ROCK: (255, 255, 0),
        Obstacle.HILL: (255, 0, 0),
        Obstacle.MOUNTAIN: (0, 0, 255)
    }

    for tile_idx, tile in enumerate(tiles):
        stddev, mean = calculate_tile_statistics(tile)
        tile_type = classify_tile(stddev, mean, thresholds)
        color = colormap[tile_type]

        row, col = divmod(tile_idx, tile_cols)
        overlay[row*tile_size:(row+1)*tile_size, col*tile_size:(col+1)*tile_size] =

    return overlay

def main(file_path, tile_size, thresholds, a1=0.7, a2=0.4):
    image = load_image(file_path)
    overlay = create_colored_overlay(image, tile_size, thresholds)
    # Blend the original image and the overlay
    blended = cv2.addWeighted(cv2.cvtColor(image, cv2.COLOR_GRAY2BGR), a1, overlay,

    # Display the result with a legend
    plt.figure(figsize=(10, 10), facecolor='black') # Set the figure's face color t
    plt.imshow(blended)
    plt.title(f"Obstacle Detection with Tile Size {tile_size}x{tile_size}", color='

    legend_labels = {
        'Flat': (0, 255, 0),
        'Rocks': (255, 255, 0),
        'Hills': (255, 0, 0),
        'Mountains': (0, 0, 255)
    }
    patches = [mpatches.Patch(color=np.array(color)/255, label=label) for label, co
    plt.legend(handles=patches, bbox_to_anchor=(1., 1), loc='upper left')

    plt.axis('off')
    plt.tight_layout(pad=2)  # This adjusts the padding around the plot area and be
    plt.show()
```

```python
thresholds = {
    'rock_stddev': 3,          # example values that classify tiles as rocks
    'rock_mean': 20,           # example mean height value
    'hill_stddev': 8,         # example values that classify tiles as hills
    'mountain_stddev': 15      # example values that classify tiles as mountains
}

def showframes_add(frames,ccmap= None,labels=[],label_font_size=8,nrows=0,ncols=0):
    """Sub function don't call it alone
    """

    plot_di = int(frames.shape[0]**0.5)

    if nrows ==0 or ncols ==0 :
        nrows, ncols = plot_di,plot_di

    fig, axes = plt.subplots(nrows=nrows, ncols=ncols)

    if len(labels)!=0:
        plt.subplots_adjust(left=0.1,bottom=0.02,right=0.9,top=0.99,wspace=0.3,hspa
        for ind ,ax in enumerate( axes.flat):

            im = ax.imshow(frames[int(ind)],cmap =ccmap) #### Plots the frame
            ax.set_title(f"lab: {labels[ind]}", fontstyle='italic',fontsize =label_

            ax.set_xticks([])#### Turn of Ticks
            ax.set_yticks([])#### Turn of Ticks




    else:
        plt.subplots_adjust(left=0.1,bottom=0.02,right=0.9,top=0.9,wspace=0.3,hspac
        for ind ,ax in enumerate( axes.flat):

            im = ax.imshow(frames[int(ind)],cmap =ccmap) #### Plots the frame
            ax.set_title(f"{ind}", fontstyle='italic',fontsize = label_font_size, p
            ax.set_xticks([])#### Turn of Ticks
            ax.set_yticks([])#### Turn of Ticks




    fig.subplots_adjust(right=0.85)
    cbar_ax = fig.add_axes([0.9, 0.15, 0.02, 0.7])
    fig.colorbar(im, cax=cbar_ax)




#     fig.tight_layout(pad=3.0)



def showframes(frames,typee= None,fig_s = (10,10),labels: list =[],label_font_size:
```

```
    """good for dealing with many frames with different image types like RGB, BGR,
    with some types "cmaps":
    gist_gray = g
    jet = c
    tab20b = t
    viridis = v
    cividis = d
    BGR = bgr
    or leave it and will do defult

    labels are used to make titles for each image like the model prediction for thi

    label_font_size takes int

    """
    plt.rcParams['figure.figsize'] = fig_s
    figure(figsize=fig_s, dpi=100)


    if typee=="g":
        showframes_add(frames,ccmap ='gist_gray',labels=labels,label_font_size= lab
    elif typee=="c" :
        showframes_add(frames,ccmap ='jet',labels=labels,label_font_size= label_fon
    elif typee=="t" :
        showframes_add(frames,ccmap ='tab20b',labels=labels,label_font_size= label_
    elif typee=="v" :
        showframes_add(frames,ccmap ='viridis',labels=labels,label_font_size= label
    elif typee=="d" :
        showframes_add(frames,ccmap ='cividis',labels=labels,label_font_size= label

    elif typee=="RGB"or typee=='rgb':
        showframes_add(frames[:,:,:,[2,1,0]],ccmap ='cividis',labels=labels,label_f
    else:
        showframes_add(frames,labels=labels,label_font_size= label_font_size,nrows=



    plt.show()


def index_to_position(index, tile_cols):
    """Convert a linear index to a 2D position (row, col)."""
    row = index // tile_cols
    col = index % tile_cols
    return row, col

def position_to_index(row, col, tile_cols):
    """Convert a 2D position (row, col) back to a linear index."""
    return row * tile_cols + col

def label_tiles(tiles, tile_rows, tile_cols):
    """Label tiles with their linear index based on their position."""
    labeled_tiles = []
    for index, tile in enumerate(tiles):
        row, col = index_to_position(index, tile_cols)
```

```python
        label = f"tile_{row}_{col}"
        labeled_tiles.append(label)
    return labeled_tiles

def extract_position_from_label(label):
    """Extract the row and column number from a tile label."""
    # Assume label format is 'tile_row_column'
    parts = label.split('_')
    if len(parts) == 3 and parts[0] == 'tile':
        row, col = int(parts[1]), int(parts[2])
        return row, col
    else:
        raise ValueError("Label does not match the expected format 'tile_row_col'")


import os

def save_labeled_tiles(image, tile_size, thresholds, storage_path):
    os.makedirs(storage_path, exist_ok=True)  # Ensure base storage path exists

    # Labels for the folders
    labels = {
        Obstacle.FLAT: "Flat",
        Obstacle.ROCK: "Rock",
        Obstacle.HILL: "Hill",
        Obstacle.MOUNTAIN: "Mountain"
    }

    # Create labeled folders
    for label in labels.values():
        os.makedirs(os.path.join(storage_path, label), exist_ok=True)

    # Process and save the tiles
    tiles, tile_rows, tile_cols = tile_image(image, tile_size)
    for tile_idx, tile in enumerate(tiles):
        stddev, mean = calculate_tile_statistics(tile)
        tile_type = classify_tile(stddev, mean, thresholds)
        label = labels[tile_type]
        label_folder = os.path.join(storage_path, label)

        # Save the tile image to the respective folder
        tile_filename = f"tile_{tile_idx}.png"
        tile_path = os.path.join(label_folder, tile_filename)
        cv2.imwrite(tile_path, tile)

tile_size=200
# Example usage
# Define thresholds, etc.
storage_path = 'data/combiled'  # Replace with your actual storage path
image = load_image('data/JEZ_ctx_B_soc_008_DTM_MOLAtopography_DeltaGeoid_20m_Eqc_la
save_labeled_tiles(image, tile_size, thresholds, storage_path)
```

In [ ]:

```python
import numpy as np
from pathlib import Path
from sklearn.model_selection import train_test_split
import cv2
```

```
import os


tile_size =50
image = load_image('data/JEZ_ctx_B_soc_008_DTM_MOLAtopography_DeltaGeoid_20m_Eqc_la

labels=label_tiles(*tile_image(image,tile_size))

pos_s = [extract_position_from_label(l) for l in labels]
print(pos_s[:5])
y = [position_to_index(*pos,tile_image(image,tile_size)[2]) for pos in pos_s]


data = np.array(tile_image(image,tile_size)[0])

labels=label_tiles(*tile_image(image,tile_size))
# showframes(data[:5],labels=y[:5],label_font_size=16)
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]
```

In [ ]:

```
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator

class Augmentor:
    def __init__(self, rotation_range=30, width_shift_range=0.1, height_shift_range
                 shear_range=0.2, zoom_range=0.2, horizontal_flip=True, noise_range
        self.rotation_range = rotation_range
        self.width_shift_range = width_shift_range
        self.height_shift_range = height_shift_range
        self.shear_range = shear_range
        self.zoom_range = zoom_range
        self.horizontal_flip = horizontal_flip
        self.noise_range = noise_range

        # Initialize ImageDataGenerator with the passed parameters
        self.datagen = ImageDataGenerator(
            rotation_range=self.rotation_range,
            width_shift_range=self.width_shift_range,
            height_shift_range=self.height_shift_range,
            shear_range=self.shear_range,
            zoom_range=self.zoom_range,
            horizontal_flip=self.horizontal_flip,
            preprocessing_function=self.add_noise
        )

    def add_noise(self, image):
        """Apply random noise to an image."""
        variance = np.random.uniform(0, self.noise_range) * (np.max(image) - np.min
        noise = np.random.normal(0, variance, image.shape)
        noisy_image = image + noise
        noisy_image = np.clip(noisy_image, 0, 255)
        return noisy_image.astype(image.dtype)

    def flow(self, x, y, batch_size=32):
        """Generate batches of augmented data."""
        return self.datagen.flow(x, y, batch_size=batch_size)
```

```python
# Assuming the Augmentor class is already defined as before
augmentor = Augmentor(rotation_range=350, width_shift_range=0.15, noise_range=0.04)

# Example dataset (replace data and y with your actual data)
# data = np.array([...])  # Your image data
# y = np.array([...])  # Your labels

# Factor to determine the number of augmented images per original image
augment_factor = 100  # Change this number based on how many augmented images you w

# Lists to store augmented images and labels
augmented_images = []
augmented_labels = []


# This 'for' loop will generate 'augment_factor' augmented images for each original
for i in range(len(data)):
    # Get a single image and label
    image = data[i]
    label = y[i]

    # Expand the image dimensions if necessary (add the channels dimension)
    if image.ndim == 2: # for grayscale images
        image = np.expand_dims(image, axis=-1)

    # Augment the image 'augment_factor' times
    for _ in range(augment_factor):
        # Use the 'flow' function from the Augmentor, which expects a batch
        # Ensure the image has four dimensions (batch_size, height, width, channels
        image_batch, label_batch = next(
            augmentor.flow(
                np.expand_dims(image, 0), # add the batch dimension
                np.expand_dims(label, 0), # add the batch dimension for the labels
                batch_size=1
            )
        )
        # Remove the batch dimension since we are processing one image at a time
        augmented_image = np.squeeze(image_batch, axis=0)
        augmented_label = np.squeeze(label_batch, axis=0)

        # Append the augmented images and labels
        augmented_images.append(augmented_image)
        augmented_labels.append(augmented_label)

# ...

# Convert the lists to Numpy arrays
augmented_images = np.array(augmented_images)
augmented_labels = np.array(augmented_labels)

# 'augmented_images' and 'augmented_labels' now contain the augmented dataset
# showframes(augmented_images[:5],labels=augmented_labels[:5])
```

```python
train_data, test_data, train_labels, test_labels = train_test_split(
        augmented_images, augmented_labels, test_size=0.25, random_state=42,shuffle
test_labels.shape
```

Out[ ]: (38000,)

In [ ]:
```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.callbacks import Callback
```

In [ ]:
```python
class EarlyStoppingByAccuracy(Callback):
    def __init__(self, monitor='accuracy', value=0.95, verbose=0):
        super(Callback, self).__init__()
        self.monitor = monitor
        self.value = value
        self.verbose = verbose

    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        current_accuracy = logs.get(self.monitor)
        if current_accuracy is None:
            raise ValueError(f"Metric `{self.monitor}` is not available. Available

        if current_accuracy >= self.value:
            if self.verbose > 0:
                print(f"Epoch {epoch}: early stopping with {self.monitor} = {curren
            self.model.stop_training = True
```

In [ ]:
```python
# Define the custom callback with the desired accuracy threshold
early_stopping_callback = EarlyStoppingByAccuracy(monitor='val_accuracy', value=0.7
```

In [ ]:
```python
# Step 1: Data Preparation

# Step 2: Model Architecture Design
model = models.Sequential([
    layers.Conv2D(16, (3, 3), activation='relu', input_shape=(tile_size, tile_size,
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dense(len(y), activation='softmax')
])

# Step 3: Model Training
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


history = model.fit(train_data, train_labels, epochs=100, validation_data=[test_dat

# Step 4: Evaluation and Validation
```

```
test_loss, test_acc = model.evaluate(test_data, test_labels)
print('Test accuracy:', test_acc)

# Step 5: Deployment and Testing
# Deploy and test the trained model in a real or simulated environment
```

```
Epoch 1/100
446/446 [==============================] - 13s 16ms/step - loss: 7.5760 - accuracy:
0.0023 - val_loss: 6.9779 - val_accuracy: 0.0048
Epoch 2/100
446/446 [==============================] - 7s 15ms/step - loss: 6.7869 - accuracy:
0.0073 - val_loss: 6.4476 - val_accuracy: 0.0137
Epoch 3/100
446/446 [==============================] - 6s 14ms/step - loss: 6.2842 - accuracy:
0.0193 - val_loss: 5.9664 - val_accuracy: 0.0308
Epoch 4/100
446/446 [==============================] - 7s 15ms/step - loss: 5.5753 - accuracy:
0.0450 - val_loss: 5.4367 - val_accuracy: 0.0473
Epoch 5/100
446/446 [==============================] - 6s 15ms/step - loss: 4.7113 - accuracy:
0.0850 - val_loss: 4.3412 - val_accuracy: 0.1085
Epoch 6/100
446/446 [==============================] - 7s 15ms/step - loss: 3.9994 - accuracy:
0.1428 - val_loss: 3.6603 - val_accuracy: 0.1696
Epoch 7/100
446/446 [==============================] - 7s 15ms/step - loss: 3.4653 - accuracy:
0.1979 - val_loss: 3.2609 - val_accuracy: 0.2204
Epoch 8/100
446/446 [==============================] - 7s 15ms/step - loss: 3.0742 - accuracy:
0.2488 - val_loss: 2.9106 - val_accuracy: 0.2764
Epoch 9/100
446/446 [==============================] - 7s 15ms/step - loss: 2.7104 - accuracy:
0.3064 - val_loss: 3.1206 - val_accuracy: 0.2294
Epoch 10/100
446/446 [==============================] - 7s 15ms/step - loss: 2.4459 - accuracy:
0.3552 - val_loss: 2.3232 - val_accuracy: 0.3721
Epoch 11/100
446/446 [==============================] - 7s 15ms/step - loss: 2.2338 - accuracy:
0.3980 - val_loss: 2.3253 - val_accuracy: 0.3648
Epoch 12/100
446/446 [==============================] - 7s 15ms/step - loss: 2.0255 - accuracy:
0.4416 - val_loss: 2.0753 - val_accuracy: 0.4165
Epoch 13/100
446/446 [==============================] - 7s 15ms/step - loss: 1.8975 - accuracy:
0.4758 - val_loss: 1.7875 - val_accuracy: 0.4900
Epoch 14/100
446/446 [==============================] - 7s 16ms/step - loss: 1.7145 - accuracy:
0.5120 - val_loss: 2.0208 - val_accuracy: 0.4479
Epoch 15/100
446/446 [==============================] - 7s 15ms/step - loss: 1.6311 - accuracy:
0.5303 - val_loss: 1.8015 - val_accuracy: 0.4954
Epoch 16/100
446/446 [==============================] - 7s 16ms/step - loss: 1.5645 - accuracy:
0.5569 - val_loss: 1.5606 - val_accuracy: 0.5536
Epoch 17/100
446/446 [==============================] - 7s 16ms/step - loss: 1.3911 - accuracy:
0.5911 - val_loss: 1.5254 - val_accuracy: 0.5615
Epoch 18/100
446/446 [==============================] - 7s 15ms/step - loss: 1.3154 - accuracy:
0.6120 - val_loss: 1.5037 - val_accuracy: 0.5565
Epoch 19/100
446/446 [==============================] - 7s 15ms/step - loss: 1.2729 - accuracy:
```

```
0.6201 - val_loss: 1.4754 - val_accuracy: 0.5715
Epoch 20/100
446/446 [==============================] - 7s 15ms/step - loss: 1.1840 - accuracy:
0.6456 - val_loss: 2.0806 - val_accuracy: 0.4921
Epoch 21/100
446/446 [==============================] - 7s 15ms/step - loss: 1.1355 - accuracy:
0.6579 - val_loss: 1.3548 - val_accuracy: 0.6032
Epoch 22/100
446/446 [==============================] - 7s 16ms/step - loss: 1.0502 - accuracy:
0.6832 - val_loss: 1.1399 - val_accuracy: 0.6672
Epoch 23/100
446/446 [==============================] - 7s 15ms/step - loss: 1.0018 - accuracy:
0.6951 - val_loss: 1.1654 - val_accuracy: 0.6629
Epoch 24/100
446/446 [==============================] - 7s 15ms/step - loss: 1.2202 - accuracy:
0.6697 - val_loss: 1.2519 - val_accuracy: 0.6351
Epoch 25/100
446/446 [==============================] - 7s 15ms/step - loss: 0.8974 - accuracy:
0.7268 - val_loss: 1.0423 - val_accuracy: 0.6939
Epoch 26/100
446/446 [==============================] - 7s 15ms/step - loss: 0.8640 - accuracy:
0.7344 - val_loss: 1.3299 - val_accuracy: 0.6253
Epoch 27/100
443/446 [============================>.] - ETA: 0s - loss: 1.0122 - accuracy: 0.7136
Epoch 26: early stopping with val_accuracy = 0.7139999866485596
446/446 [==============================] - 7s 15ms/step - loss: 1.0113 - accuracy:
0.7137 - val_loss: 0.9867 - val_accuracy: 0.7140
1188/1188 [==============================] - 5s 4ms/step - loss: 0.9867 - accuracy:
0.7140
Test accuracy: 0.7139999866485596
```

In [ ]:

```
# Save the model in HDF5 format
model.save('model/saved_model_acc_71.h5')
```