list-test-1-2

May 10, 2025

[1]:

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)
print("GPUs Available:", tf.config.list_physical_devices('GPU'))
```

```
2025-05-07 16:19:38.892618: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

```
TensorFlow version: 2.13.0
GPUs Available: [PhysicalDevice(name='/physical_device:GPU:0',
device_type='GPU'), PhysicalDevice(name='/physical_device:GPU:1',
device_type='GPU'), PhysicalDevice(name='/physical_device:GPU:2',
device_type='GPU'), PhysicalDevice(name='/physical_device:GPU:3',
device_type='GPU')]
```

[2]:

```
import tensorflow as tf

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        # Select GPU with index 1 (i.e., the second GPU)
        tf.config.set_visible_devices(gpus[3], 'GPU')
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(f"Using GPUs: {logical_gpus}")
    except RuntimeError as e:
        # Visible devices must be set before GPUs have been initialized
        print(e)
else:
    print("No GPUs found.")
```

```
Using GPUs: [LogicalDevice(name='/device:GPU:0', device_type='GPU')]
```

```
2025-05-07 16:19:41.915259: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 30942 MB memory:  -> device:
3, name: Tesla V100-DGXS-32GB, pci bus id: 0000:0f:00.0, compute capability: 7.0
```

[ ]:

```

```

[ ]:

```

```

[3]:

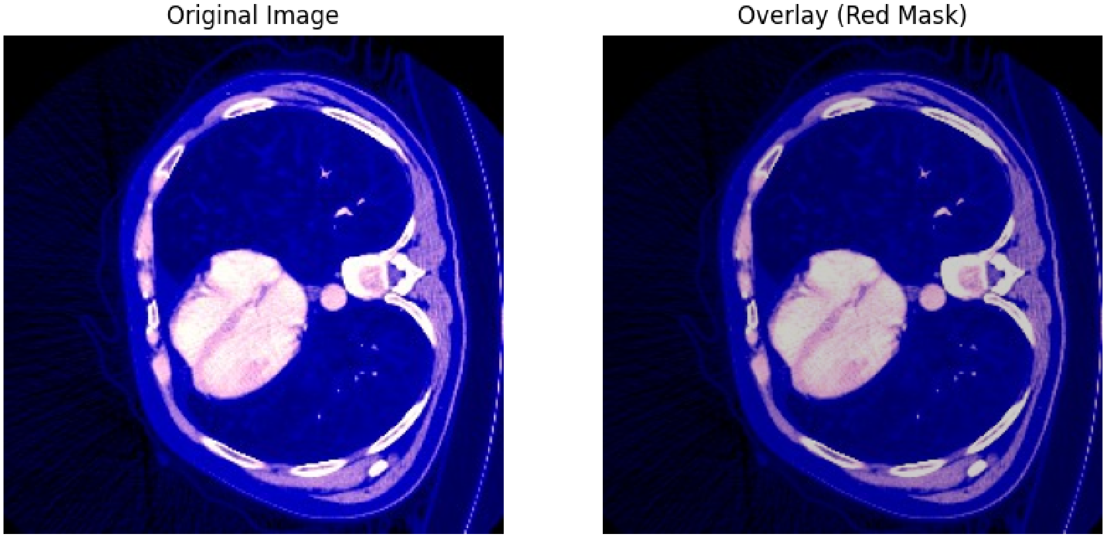
```python
import os
import cv2
import numpy as np
import math
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.utils import Sequence
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Conv2DTranspose, concatenate, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping

# ==== 1. Paths ====
image_dir = '/raid/lits_dataset/train_images/train_images'
mask_dir  = '/raid/lits_dataset/train_masks/train_masks'

# ==== 2. Load all valid image-mask pairs ====
all_filenames = [
    f for f in os.listdir(image_dir)
    if f.lower().endswith(('.png', '.jpg', '.jpeg')) and os.path.exists(os.path.join(mask_dir, f))
]

valid_filenames = [
    f for f in all_filenames
    if cv2.imread(os.path.join(mask_dir, f), cv2.IMREAD_GRAYSCALE) is not None
]

# Optional: Print number of positive masks
tumor_count = 0
for f in valid_filenames:
    mask = cv2.imread(os.path.join(mask_dir, f), cv2.IMREAD_GRAYSCALE)
    if np.any(mask > 127):
        tumor_count += 1
print(f"{tumor_count}/{len(valid_filenames)} images contain tumors")

# ==== 3. Train/Val Split ====
np.random.seed(42)
indices = np.arange(len(valid_filenames))
np.random.shuffle(indices)
split_idx = int(0.8 * len(indices))
train_files = [valid_filenames[i] for i in indices[:split_idx]]
val_files   = [valid_filenames[i] for i in indices[split_idx:]]
```

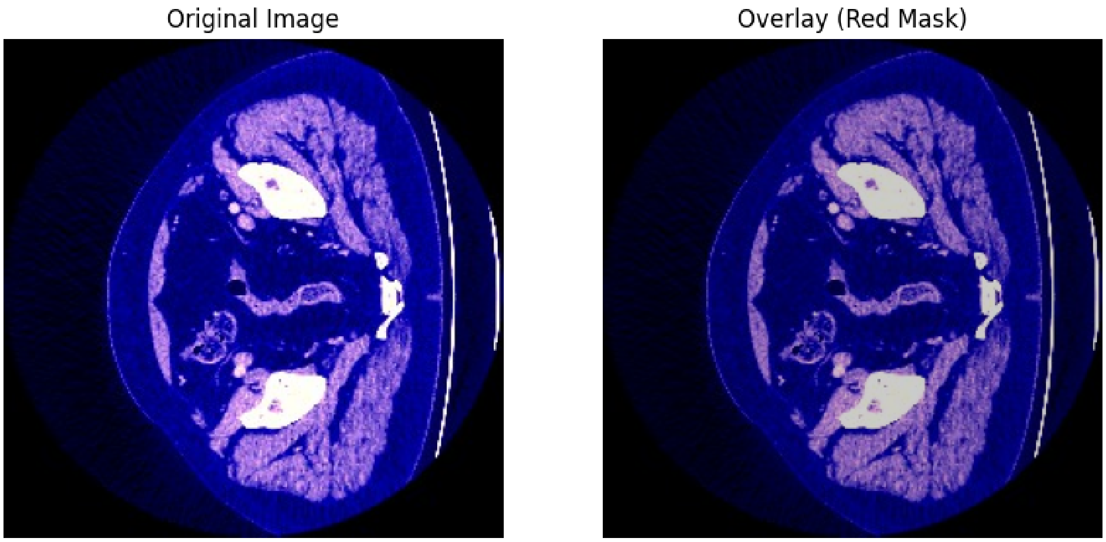
```
0/58638 images contain tumors
```

[4]:

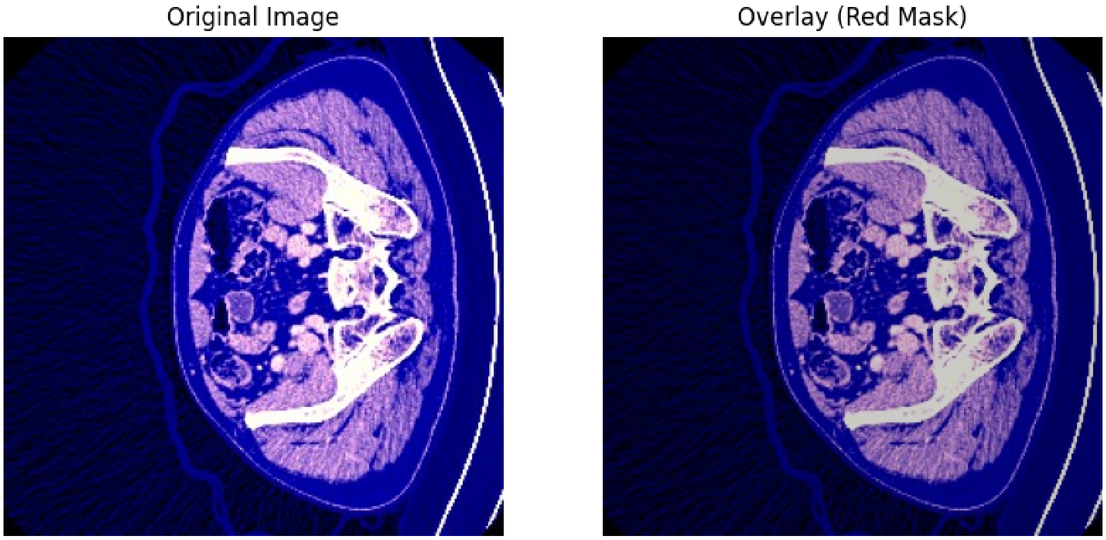
```python
# If in a notebook, ensure inline plotting
%matplotlib inline

import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from random import sample

def overlay_mask_on_image(img, mask, color=(0, 0, 255), alpha=0.5):
    """
    Overlay a single-channel mask onto a BGR image in red.

    Args:
        img   : H×W×3 uint8 BGR image.
        mask  : H×W uint8 mask (0–255).
        color : BGR tuple for the mask overlay (now red = (0,0,255)).
        alpha : float opacity of the mask overlay [0.0–1.0].
    Returns:
        overlaid : H×W×3 uint8 image.
    """
    mask_color = np.zeros_like(img)
    mask_color[mask > 0] = color
    return cv2.addWeighted(mask_color, alpha, img, 0.8, 0)


# 1. Configure these to point to your folders:
image_dir = '/raid/lits_dataset/train_images/train_images'
mask_dir  = '/raid/lits_dataset/train_masks/train_masks'
image_size = (256, 256)

# 2. Find all filenames present in both folders:
all_filenames = [
    f for f in os.listdir(image_dir)
    if f.lower().endswith(('.png', '.jpg', '.jpeg'))
    and os.path.exists(os.path.join(mask_dir, f))
]

print(f"Found {len(all_filenames)} image-mask pairs.")

# 3. Visualize a random subset (up to 5)
n_samples = min(5, len(all_filenames))
for fname in sample(all_filenames, n_samples):
    # Load and resize
    img  = cv2.imread(os.path.join(image_dir, fname))
    img  = cv2.resize(img, image_size)
    mask = cv2.imread(os.path.join(mask_dir,  fname), cv2.IMREAD_GRAYSCALE)
    mask = cv2.resize(mask, image_size)

    # Create overlay in red
    overlay = overlay_mask_on_image(img, mask, color=(0, 0, 255), alpha=0.5)

    # Convert BGR→RGB for matplotlib
    img_rgb     = cv2.cvtColor(img,     cv2.COLOR_BGR2RGB)
    overlay_rgb = cv2.cvtColor(overlay, cv2.COLOR_BGR2RGB)

    # Plot original and overlay
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.title('Original Image')
    plt.imshow(img_rgb)
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title('Overlay (Red Mask)')
    plt.imshow(overlay_rgb)
    plt.axis('off')

    plt.show()
```

```
Found 58638 image-mask pairs.
```

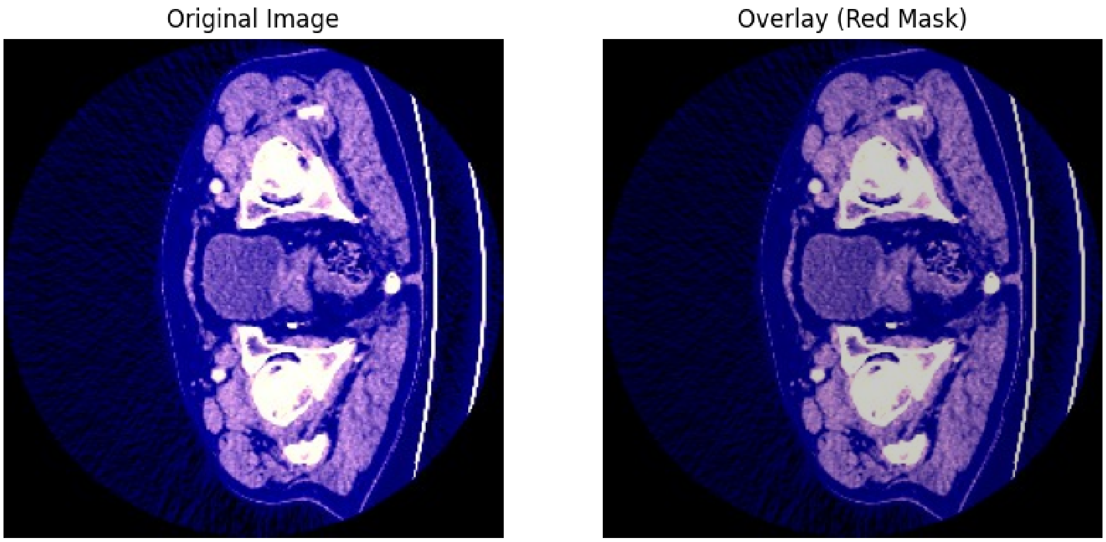


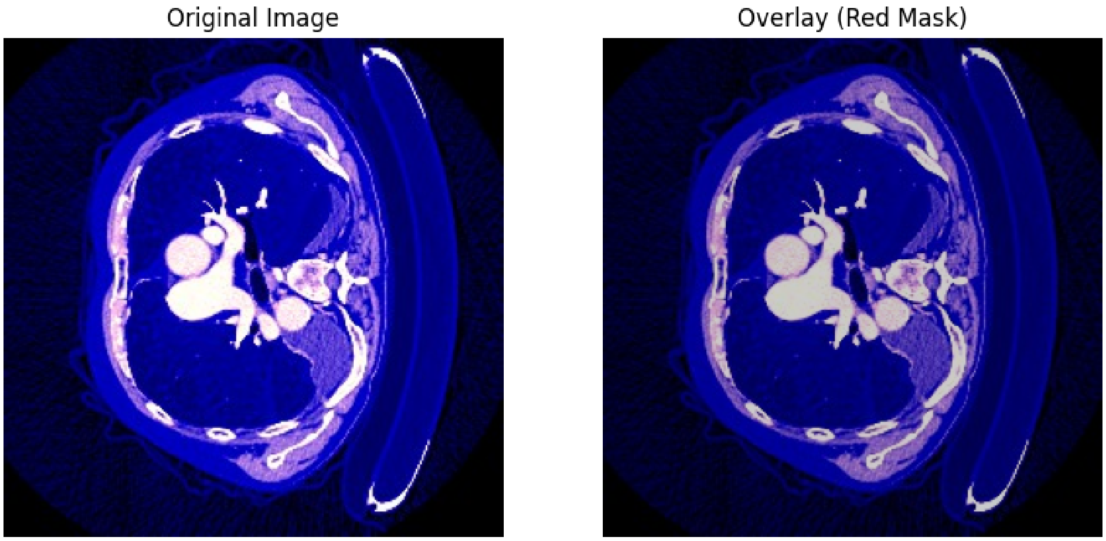

Original Image

Overlay (Red Mask)


Original Image

Overlay (Red Mask)





[23]:

```
import os
import cv2
import numpy as np
from tensorflow.keras.utils import Sequence
from sklearn.model_selection import train_test_split

# ------------------------------------------------------------------------
# Your existing DataGenerator class
# ------------------------------------------------------------------------
class SegmentationDataGenerator(Sequence):
    def __init__(self, image_dir, mask_dir, file_list, batch_size=16,
                 image_size=(256,256), shuffle=True, augment_fn=None):
        """
        Args:
            image_dir   : path to folder with input images
            mask_dir    : path to folder with corresponding masks
            file_list   : list of filenames (basename) to include
            batch_size  : number of samples per batch
            image_size  : (height, width) to resize to
            shuffle     : whether to shuffle at epoch end
            augment_fn  : optional function(img, mask) -> (img, mask)
        """
        self.image_dir  = image_dir
        self.mask_dir   = mask_dir
        self.file_list  = list(file_list)
        self.batch_size = batch_size
        self.image_size = image_size
        self.shuffle    = shuffle
        self.augment_fn = augment_fn
        self.on_epoch_end()

    def __len__(self):
        # number of batches per epoch
        return int(np.ceil(len(self.file_list) / self.batch_size))

    def __getitem__(self, idx):
        # build batch indexes
        batch_files = self.file_list[idx * self.batch_size : (idx+1) * self.
↪batch_size]

        imgs, masks = [], []
        for fname in batch_files:
            # load & resize
            img  = cv2.imread(os.path.join(self.image_dir, fname), cv2.
↪IMREAD_COLOR)
            mask = cv2.imread(os.path.join(self.mask_dir,  fname), cv2.
↪IMREAD_GRAYSCALE)
            img  = cv2.resize(img,  self.image_size)
            mask = cv2.resize(mask, self.image_size)

            # normalize image
            img  = img.astype(np.float32) / 255.0
            # binarize mask
            mask = (mask > 0).astype(np.float32)

            # optional augmentation
            if self.augment_fn is not None:
                img, mask = self.augment_fn(img, mask)

            imgs.append(img)
            masks.append(mask[..., np.newaxis])  # add channel dim

        X = np.stack(imgs, axis=0)
        y = np.stack(masks, axis=0)
        return X, y

    def on_epoch_end(self):
        if self.shuffle:
            np.random.shuffle(self.file_list)


# -------------------------------------------------------------------------
# 1) Gather and split your filenames
# -------------------------------------------------------------------------
image_dir = '/raid/lits_dataset/train_images/train_images'
mask_dir  = '/raid/lits_dataset/train_masks/train_masks'

# get all matching image-mask names
all_filenames = [
    f for f in os.listdir(image_dir)
    if f.lower().endswith(('.png', '.jpg', '.jpeg'))
       and os.path.exists(os.path.join(mask_dir, f))
]

# split into train/val
train_files, val_files = train_test_split(
    all_filenames,
    test_size=0.2,
    random_state=42,
    shuffle=True
)

print(f"Train samples: {len(train_files)}, Validation samples:␣
 ↪{len(val_files)}")


# -------------------------------------------------------------------------
# 2) (Optional) Define a simple augmentation function
# -------------------------------------------------------------------------
def simple_augment(img, mask):
    # example: random horizontal flip
    if np.random.rand() < 0.5:
        img = np.fliplr(img)
        mask = np.fliplr(mask)
    return img, mask


# ---------------------------------------------------------------------------
# 3) Instantiate your train and validation generators
# ---------------------------------------------------------------------------
train_gen = SegmentationDataGenerator(
    image_dir=image_dir,
    mask_dir=mask_dir,
    file_list=train_files,
    batch_size=16,
    image_size=(256,256),
    shuffle=True,
    augment_fn=simple_augment
)

val_gen = SegmentationDataGenerator(
    image_dir=image_dir,
    mask_dir=mask_dir,
    file_list=val_files,
    batch_size=16,
    image_size=(256,256),
    shuffle=False,       # usually no shuffle for validation
    augment_fn=None      # no augmentation on validation
)
```

```
Train samples: 46910, Validation samples: 11728
```

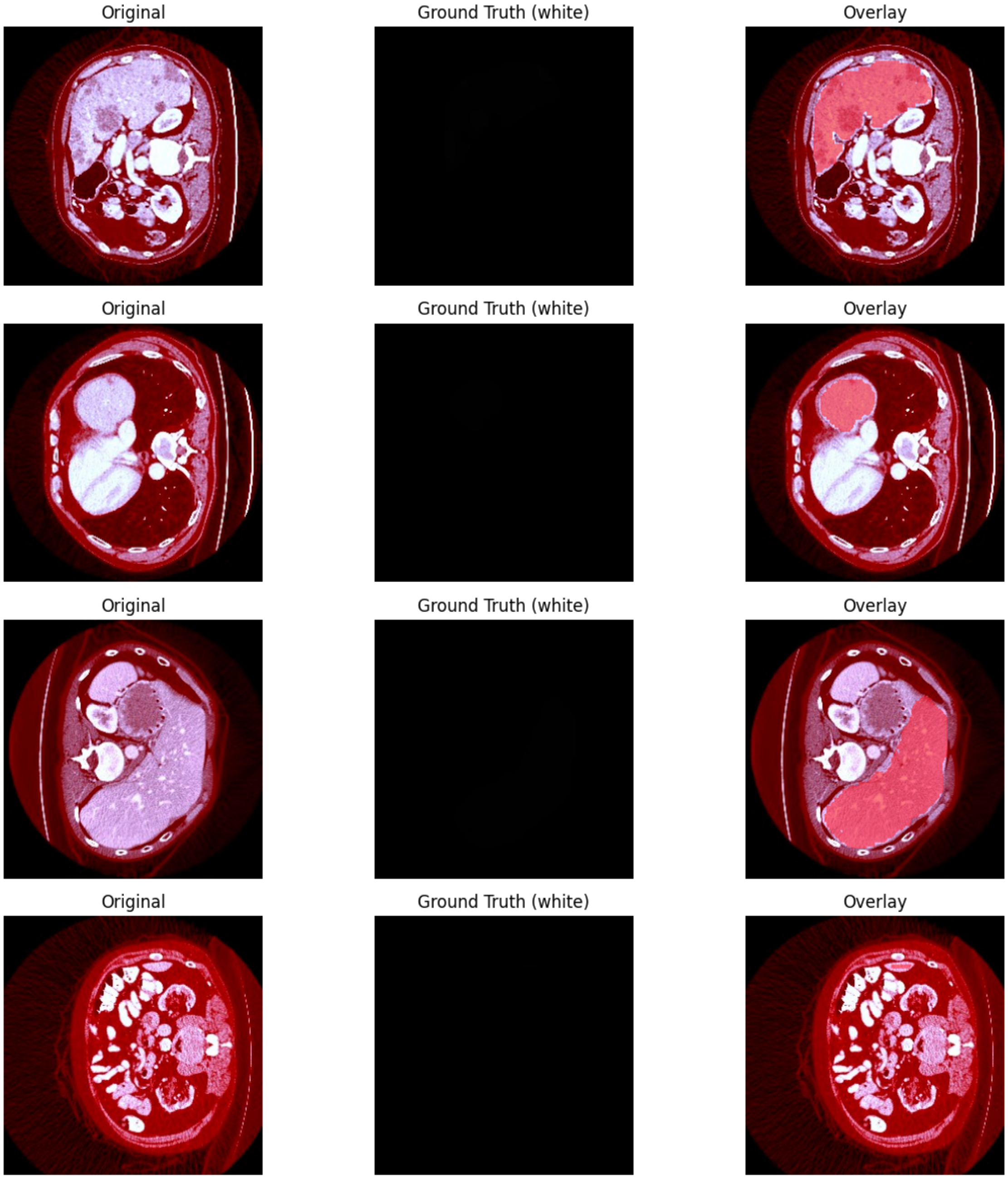
[ ]:

```

```

[19]:

```
import numpy as np
import matplotlib.pyplot as plt

def overlay_mask_on_image(img, mask, color=(1, 0, 0), alpha=0.5):
    overlay = img.copy()
    for c in range(3):
        overlay[..., c] = np.where(
            mask > 0,
            img[..., c] * (1 - alpha) + alpha * color[c],
            img[..., c]
        )
    return overlay

X_val, y_val = val_gen[0]
n = min(4, X_val.shape[0])
plt.figure(figsize=(12, 3 * n))
```

```
for i in range(n):
    img  = X_val[i]
    mask = y_val[i, ..., 0]   # binary 0/1

    overlay = overlay_mask_on_image(img, mask)

    # Original
    ax = plt.subplot(n, 3, 3*i + 1)
    ax.imshow(img)
    ax.set_title("Original")
    ax.axis('off')

    # Ground truth (white)
    ax = plt.subplot(n, 3, 3*i + 2)
    #-- choose one--#
    ax.imshow(mask, cmap='gray', vmin=0, vmax=1, interpolation='nearest')
    # OR:
    # gt = np.zeros((*mask.shape, 3), dtype=np.float32)
    # gt[mask>0] = 1.0
    # ax.imshow(gt)
    ax.set_title("Ground Truth (white)")
    ax.axis('off')

    # Overlay
    ax = plt.subplot(n, 3, 3*i + 3)
    ax.imshow(overlay)
    ax.set_title("Overlay")
    ax.axis('off')

plt.tight_layout()
plt.show()
```

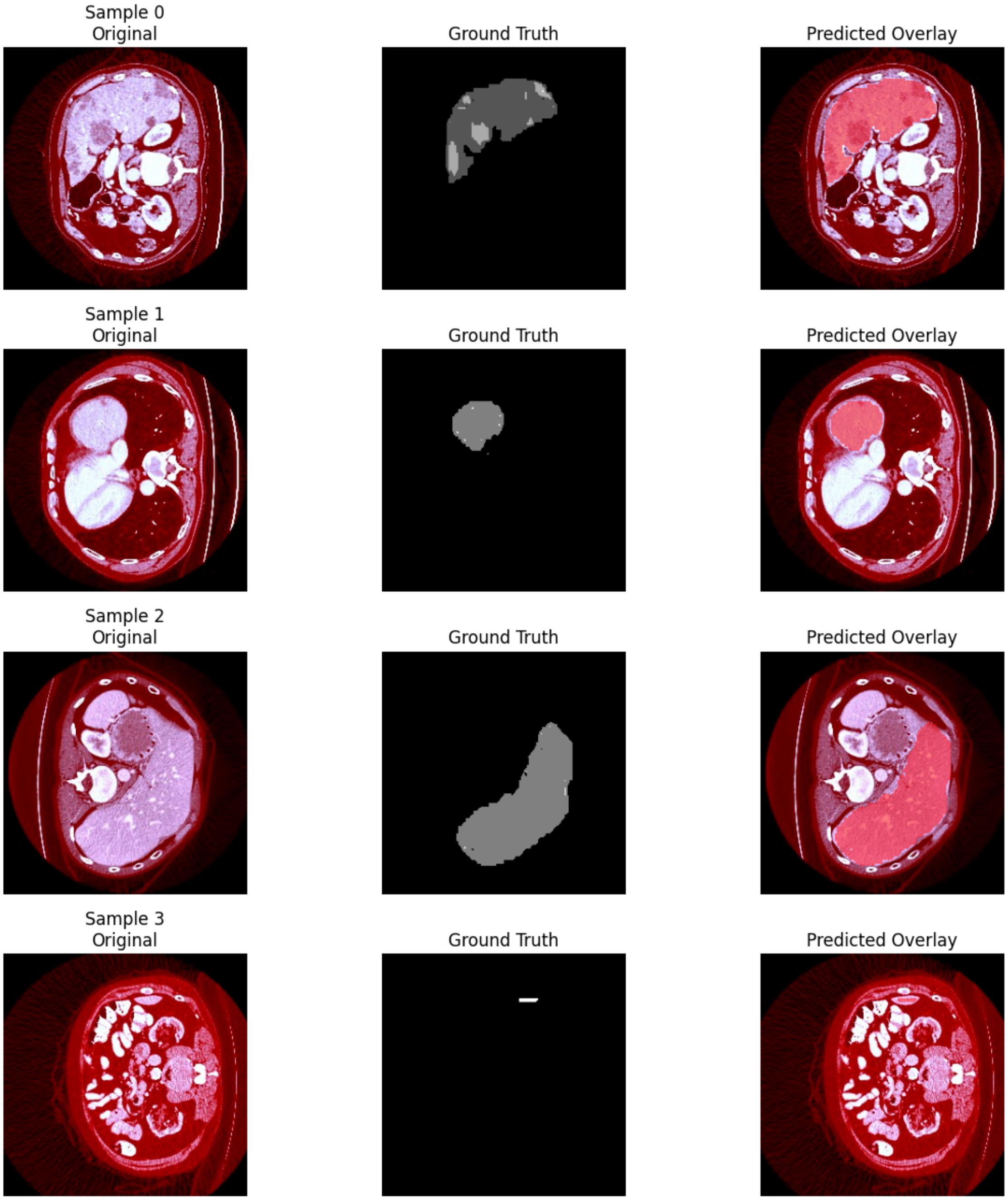


[ ]:

```
from tensorflow.keras.utils import Sequence
import os, cv2
import numpy as np

class SegmentationDataGenerator(Sequence):
    def __init__(self, image_dir, mask_dir, batch_size, image_size):
        self.image_dir  = image_dir
        self.mask_dir   = mask_dir
        self.batch_size = batch_size
        self.image_size = image_size

        # automatically grab all filenames
        self.filenames = sorted(os.listdir(self.image_dir))
        # optional: filter to known extensions
        self.filenames = [f for f in self.filenames if f.lower().endswith(('.
↪png','.jpg','.jpeg'))]

        assert len(self.filenames) > 0, "No images found in " + image_dir

    def __len__(self):
        return int(np.ceil(len(self.filenames) / self.batch_size))

    def __getitem__(self, idx):
        batch_files = self.filenames[idx * self.batch_size : (idx + 1) * self.
↪batch_size]
        images = []
        masks  = []

        # DEBUG: print what we're about to load
        print("Loading batch:", batch_files)

        for fname in batch_files:
            img_path  = os.path.join(self.image_dir, fname)
            mask_path = os.path.join(self.mask_dir,  fname)
            # Now both img_path and mask_path are guaranteed strings
            img  = cv2.imread(img_path)
            mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
            img  = cv2.resize(img, self.image_size)
            mask = cv2.resize(mask, self.image_size)
            images.append(img)
            masks.append(mask[..., np.newaxis])

        return np.array(images), np.array(masks)
```

[16]:

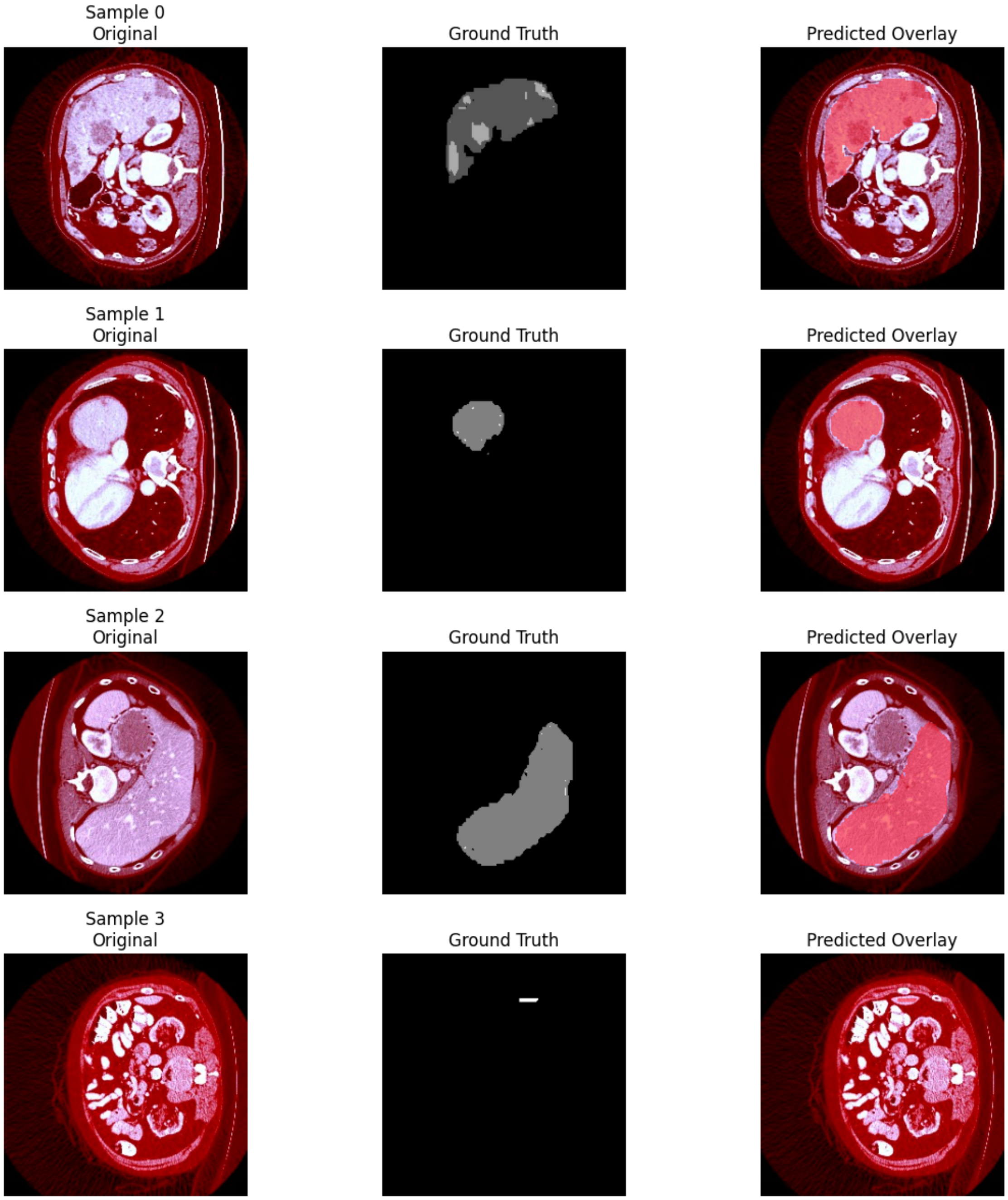
```
import cv2
import matplotlib.pyplot as plt

# helper to overlay mask onto image
def overlay_mask_on_image(img, mask, color=(1, 0, 0), alpha=0.5):
    """
    Overlay a single-channel mask onto an RGB image.

    Args:
        img   : H×W×3 float image in [0,1] RGB
        mask  : H×W binary mask (0 or 1)
        color : RGB tuple for the mask overlay (default red)
        alpha : float opacity of the mask overlay [0.0-1.0]
    Returns:
        overlaid : H×W×3 float image
    """
    overlay = img.copy()
    for c in range(3):
        overlay[..., c] = np.where(mask > 0,
                                   img[..., c] * (1 - alpha) + alpha * color[c],
                                   img[..., c])
    return overlay

# 1) Pull a batch from val_gen
X_val, y_val = val_gen[0]   # shapes: (batch, H, W, 3) and (batch, H, W, 1)

# 2) Visualize the first 4 samples
n = min(4, X_val.shape[0])
plt.figure(figsize=(12, 3 * n))

for i in range(n):
    img  = X_val[i]            # float RGB in [0,1]
    mask = y_val[i, ..., 0]    # binary mask 0/1

    overlay = overlay_mask_on_image(img, mask)

    # Original
    ax = plt.subplot(n, 3, 3*i + 1)
    ax.imshow(img)
    ax.set_title(f"Sample {i}\nOriginal")
    ax.axis('off')

    # Ground truth mask
    ax = plt.subplot(n, 3, 3*i + 2)
    ax.imshow(mask, cmap='gray')
    ax.set_title("Ground Truth")
    ax.axis('off')

    # Overlay
    ax = plt.subplot(n, 3, 3*i + 3)
    ax.imshow(overlay)
    ax.set_title("Predicted Overlay")
    ax.axis('off')

plt.tight_layout()
plt.show()
```



[22]:

```
import cv2
import matplotlib.pyplot as plt

# helper to overlay mask onto image
def overlay_mask_on_image(img, mask, color=(1, 0, 0), alpha=0.5):
    """
    Overlay a single-channel mask onto an RGB image.
    Args:
        img   : H×W×3 float image in [0,1] RGB
        mask  : H×W binary mask (0 or 1)
        color : RGB tuple for the mask overlay (default red)
        alpha : float opacity of the mask overlay [0.0-1.0]
    Returns:
        overlaid : H×W×3 float image
    """
    overlay = img.copy()
    for c in range(3):
        overlay[..., c] = np.where(mask > 0,
                                   img[..., c] * (1 - alpha) + alpha * color[c],
                                   img[..., c])
    return overlay

# 1) Pull a batch from val_gen
X_val, y_val = val_gen[0]   # shapes: (batch, H, W, 3) and (batch, H, W, 1)

# 2) Visualize the first 4 samples
n = min(4, X_val.shape[0])
plt.figure(figsize=(12, 3 * n))

for i in range(n):
    img  = X_val[i]            # float RGB in [0,1]
    mask = y_val[i, ..., 0]    # binary mask 0/1

    overlay = overlay_mask_on_image(img, mask)

    # Original
    ax = plt.subplot(n, 3, 3*i + 1)
    ax.imshow(img)
    ax.set_title(f"Sample {i}\nOriginal")
    ax.axis('off')

    # Ground truth mask
    ax = plt.subplot(n, 3, 3*i + 2)
    ax.imshow(mask, cmap='gray')
    ax.set_title("Ground Truth")
    ax.axis('off')

    # Overlay
    ax = plt.subplot(n, 3, 3*i + 3)
    ax.imshow(overlay)
    ax.set_title("Predicted Overlay")
    ax.axis('off')
```

```
plt.tight_layout()
plt.show()
```



[ ]:

[ ]: