Brain MRI Segmentation with SA-UNet

Sunidhi Ashtekar
9 min read · Mar 17, 2024

Image from Article

This article delves into the project that utilizes the Spatial Attention U-Net (SA-UNet) for brain MRI segmentation, aiming to improve medical diagnosis accuracy with the LGG-MRI-Segmentation dataset from Kaggle.

Brain MRI (Magnetic Resonance Imaging) uses detailed imaging to explore the brain's anatomy and potential issues, providing insights critical for diagnosis and treatment. Segmentation, crucial in medical imaging, divides images to analyze regions of interest, aiding in identifying abnormalities. There are two main types: semantic segmentation, which classifies each pixel without distinguishing objects of the same category, and instance segmentation, which differentiates between individual objects. These methods are vital for detailed analyses, such as identifying tumors in brain MRI scans.

Getting Started

For segmenting the brain MRI images, we have to follow these structured steps:

- Get the Dataset
- Environment setup
- Verify and Understand the Dataset
- Data Pre-processing
- Define Loss functions and Metrics
- Define Model
- Create Data Generator
- Model Training
- Model Evaluation and Inference

About the dataset

The "LGG-MRI-Segmentation" dataset, sourced from The Cancer Imaging Archive and part of The Cancer Genome Atlas, includes MRI images and genomic data from

110 patients with lower-grade gliomas, featuring FLAIR sequences.

Accompanied by detailed genomic and clinical information, this dataset enriches segmentation research by offering a comprehensive view of each patient's medical profile. It's crucial for developing precise segmentation models, enabling the exploration of the nexus between imaging and genomics.

	Patient	RNASeqCluster	MethylationCluster	miRNACluster	CNCluster	\
0	TCGA_CS_4941	2.0	4.0	2	2.0	
1	TCGA_CS_4942	1.0	5.0	2	1.0	
2	TCGA_CS_4943	1.0	5.0	2	1.0	
3	TCGA_CS_4944	NaN	5.0	2	1.0	
4	TCGA_CS_5393	4.0	5.0	2	1.0	

	RPPACluster	OncosignCluster	COCCluster	histological_type	\
0	NaN	3.0	2	1.0	
1	1.0	2.0	1	1.0	
2	2.0	2.0	1	1.0	
3	2.0	1.0	1	1.0	
4	2.0	3.0	1	1.0	

	neoplasm_histologic_grade	tumor_tissue_site	laterality	tumor_location	\
0	2.0	1.0	3.0	2.0	
1	2.0	1.0	3.0	2.0	
2	2.0	1.0	1.0	2.0	
3	1.0	1.0	3.0	6.0	
4	2.0	1.0	1.0	6.0	

	gender	age_at_initial_pathologic	race	ethnicity	death01
0	2.0	67.0	3.0	2.0	1.0
1	1.0	44.0	2.0	NaN	1.0
2	2.0	37.0	3.0	NaN	0.0
3	2.0	50.0	3.0	NaN	0.0
4	2.0	39.0	3.0	NaN	0.0

Environment Setup

A Google account is required to access Google Colab, a free cloud service providing necessary computational resources. Then mount the Google drive to utilize the dataset stored followed by installing and importing the necessary Python packages.

```
from google.colab import drive
drive.mount('/content/drive')

import os
```

```
import numpy as np
import pandas as pd
import cv2
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.models import Model, load_model, save_model
from tensorflow.keras.layers import Input, Conv2D, Conv2DTranspose, MaxPooling2
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.optimizers.legacy import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Understanding the Dataset

We'll explore the `data.csv` from our brain MRI dataset, crucial for understanding the images and labels, aiding in effective dataset preparation and analysis.

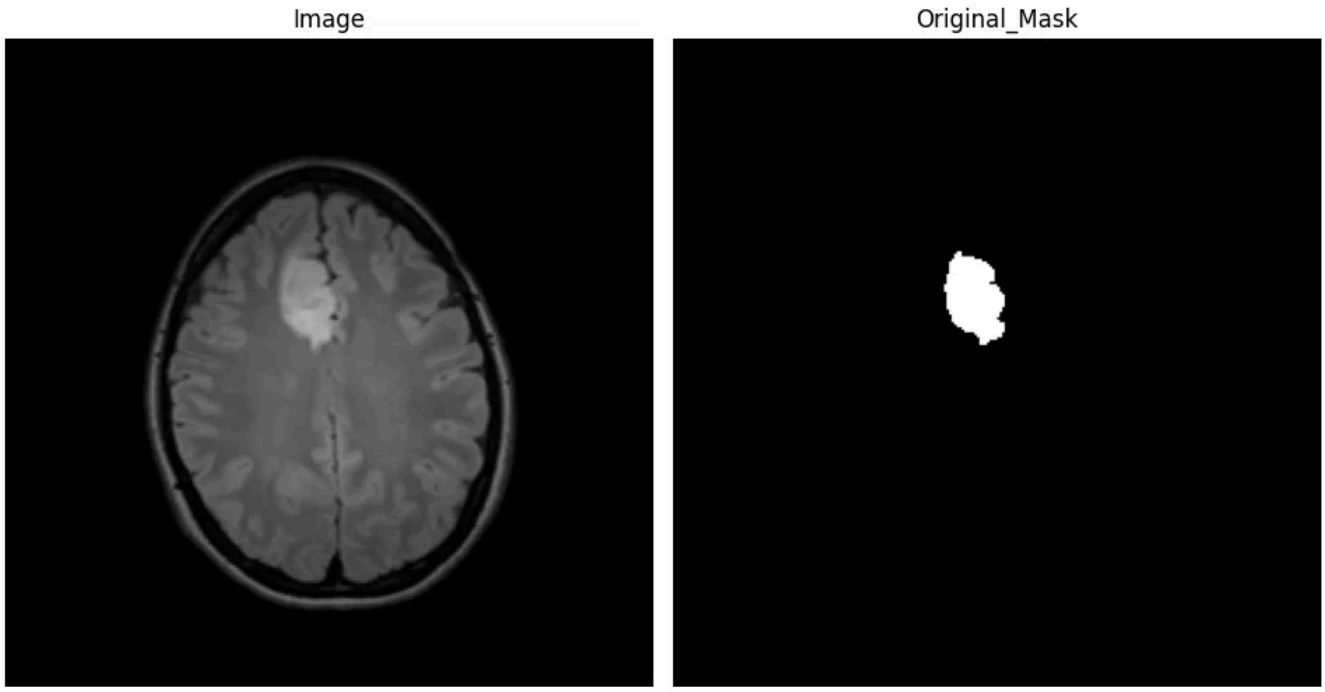
```
# Load the data.csv to understand the metadata
data_csv_path = '/content/drive/My Drive/archive/lgg-mri-segmentation/kaggle_3m
data_df = pd.read_csv(data_csv_path)

# Display the first few rows of the dataframe
print(data_df.head())
```

Data Visualization

We then visualize MRI images and their masks to understand mask application on the brain MRIs, crucial for assessing data quality and verifying our model's segmentation accuracy.

```
def load_images_with_masks(base_dir, patient_id, num_samples=50):
    patient_path = os.path.join(base_dir, patient_id)
    image_files = [f for f in os.listdir(patient_path) if not f.endswith('_mask
    image_files = sorted(image_files)[:num_samples]  # Limit the number of samp

    images = []
    masks = []

    for img_file in image_files:
        img_path = os.path.join(patient_path, img_file)
        mask_path = os.path.join(patient_path, img_file.replace('.tif', '_mask.

        img = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
        mask = cv2.imread(mask_path, cv2.IMREAD_UNCHANGED)

        if img is not None and mask is not None:
            images.append(img)
            masks.append(mask)

    return images, masks

def visualize_images_and_masks(images, masks):
    plt.figure(figsize=(10, 5 * len(images)))
    for i, (image, mask) in enumerate(zip(images, masks)):
        plt.subplot(len(images), 2, 2*i + 1)
        plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        plt.title('Image')
        plt.axis('off')

        plt.subplot(len(images), 2, 2*i + 2)
        plt.imshow(mask, cmap='gray')
        plt.title('Original_Mask')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

base_dir = '/content/drive/My Drive/archive/lgg-mri-segmentation/kaggle_3m/'
patient_id = 'TCGA_FG_A60K_20040224'

images, masks = load_images_with_masks(base_dir, patient_id)
visualize_images_and_masks(images, masks)

def count_total_images_and_masks(base_dir):
    total_images = 0
    total_masks = 0

    for root, dirs, files in os.walk(base_dir):
        for file in files:
            if file.endswith('.tif') and '_mask' not in file:
                total_images += 1
            elif '_mask.tif' in file:
```

```
                total_masks += 1

    return total_images, total_masks

base_dir = '/content/drive/My Drive/archive/lgg-mri-segmentation/kaggle_3m/'
total_images, total_masks = count_total_images_and_masks(base_dir)
print(f"Total number of images: {total_images}")
print(f"Total number of masks: {total_masks}")
```



Data Preprocessing

This step involves dividing our dataset into three distinct sets: training (85%), validation (10%), and test (5%).

```
def get_image_mask_pairs(base_dir):
    all_images = []
    all_masks = []

    for root, dirs, files in os.walk(base_dir):
        for file in sorted(files):
            if file.endswith('.tif') and not file.endswith('_mask.tif'):
                image_path = os.path.join(root, file)
                mask_path = os.path.join(root, file.replace('.tif', '_mask.tif'
```

```
                if os.path.exists(mask_path):  # Ensure the mask exists
                    all_images.append(image_path)
                    all_masks.append(mask_path)

    return all_images, all_masks

base_dir = '/content/drive/My Drive/archive/lgg-mri-segmentation/kaggle_3m/'
all_images, all_masks = get_image_mask_pairs(base_dir)

# Initial split: 85% for training, 15% for temp (to be split further into valic
train_images, temp_images, train_masks, temp_masks = train_test_split(all_image

# Split the temp data into validation and test sets
# Since temp is 15% of the whole, validation is 10% of the whole (about 2/3 of
validation_images, test_images, validation_masks, test_masks = train_test_split

print(f"Training set: {len(train_images)} images")
print(f"Validation set: {len(validation_images)} images")
print(f"Test set: {len(test_images)} images")

# Convert lists into DataFrames
df_train = pd.DataFrame({'image_path': train_images, 'mask_path': train_masks})
df_validation = pd.DataFrame({'image_path': validation_images, 'mask_path': val
df_test = pd.DataFrame({'image_path': test_images, 'mask_path': test_masks})
```

Setting Up the Training Data Generator

The generator enhances training through real-time data augmentation, like rotations and flips, boosting model robustness by mimicking various MRI conditions. It also manages efficient batch processing, essential for handling large datasets without memory overload.

```
def train_generator(
    data_frame,
    batch_size,
    augmentation_dict,
    image_color_mode="rgb",
    mask_color_mode="grayscale",
    image_save_prefix="image",
    mask_save_prefix="mask",
    save_to_dir=None,
    target_size=(256, 256),
    seed=1,
```

```
    ):
        image_datagen = ImageDataGenerator(**augmentation_dict)
        mask_datagen = ImageDataGenerator(**augmentation_dict)

        # Create generators for images and masks
        image_generator = image_datagen.flow_from_dataframe(
            data_frame,
            x_col="image_path",
            class_mode=None,
            color_mode=image_color_mode,
            target_size=target_size,
            batch_size=batch_size,
            save_to_dir=save_to_dir,
            save_prefix=image_save_prefix,
            seed=seed,
        )

        mask_generator = mask_datagen.flow_from_dataframe(
            data_frame,
            x_col="mask_path",
            class_mode=None,
            color_mode=mask_color_mode,
            target_size=target_size,
            batch_size=batch_size,
            save_to_dir=save_to_dir,
            save_prefix=mask_save_prefix,
            seed=seed,
        )

        train_gen = zip(image_generator, mask_generator)

        for (img, mask) in train_gen:
            img, mask = normalize_data(img, mask)
            yield (img, mask)
```

Data Normalization

Normalization scales pixel values to a standard range, typically 0 to 1, crucial for stabilizing training and enhancing model convergence in deep learning.

```
def normalize_data(img, mask):
    img = img / 255.0
    mask = mask / 255.0
```

```
    mask[mask > 0.5] = 1
    mask[mask <= 0.5] = 0
    return (img, mask)
```

Loss Function and Evaluation Metrics

We define the loss function and evaluation metrics for our model's training and evaluation. The Dice Coefficient Loss, based on the Dice coefficient, effectively measures overlap for segmentation tasks. Our metrics include accuracy, Intersection over Union (IoU), and the Dice coefficient.

```
def dice_coefficients(y_true, y_pred, smooth=100):
    y_true_flatten = K.flatten(y_true)
    y_pred_flatten = K.flatten(y_pred)

    intersection = K.sum(y_true_flatten * y_pred_flatten)
    union = K.sum(y_true_flatten) + K.sum(y_pred_flatten)
    return (2 * intersection + smooth) / (union + smooth)

def dice_coefficients_loss(y_true, y_pred, smooth=100):
    return -dice_coefficients(y_true, y_pred, smooth)

def iou(y_true, y_pred, smooth=100):
    intersection = K.sum(y_true * y_pred)
    sum = K.sum(y_true + y_pred)
    iou = (intersection + smooth) / (sum - intersection + smooth)
    return iou

def jaccard_distance(y_true, y_pred):
    y_true_flatten = K.flatten(y_true)
    y_pred_flatten = K.flatten(y_pred)
    return -iou(y_true_flatten, y_pred_flatten)
```

Defining Encoder and Decoder Blocks for the SA-UNet Model

Encoder Blocks: Each block comprises convolutional layers followed by dropout, batch normalization, ReLu and max pooling for downsampling.

Spatial Attention block: It applies both max pooling and average pooling across the channel dimension to capture different aspects of the spatial features. The outputs of these pooling operations are then concatenated and processed through a convolutional layer, creating a spatial attention map. This attention map, after being normalized by a sigmoid activation, is multiplied element-wise with the encoder output, refining the feature map that is fed into the decoder.

Decoder Blocks: Each decoder block starts with a transposed convolution for upsampling, followed by concatenation with the corresponding encoder output, and convolutional layers similar to those in the encoder.

```
def encoder_block(input_tensor, num_filters, dropout_rate=0.2):
    x = Conv2D(num_filters, (3, 3), padding='same')(input_tensor)
    x = Dropout(dropout_rate)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(num_filters, (3, 3), padding='same')(x)
    x = Dropout(dropout_rate)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    p = MaxPooling2D((2, 2))(x)
    return x, p

def spatial_attention_block(input_tensor):
    max_pool = tf.reduce_max(input_tensor, axis=-1, keepdims=True)
    avg_pool = tf.reduce_mean(input_tensor, axis=-1, keepdims=True)

    concat = tf.concat([max_pool, avg_pool], axis=-1)
    attention = Conv2D(1, (7, 7), padding='same', activation='sigmoid')(concat)
    output = Multiply()([input_tensor, attention])

    return output

def decoder_block(input_tensor, skip_features, num_filters, dropout_rate=0.2):
    x = Conv2DTranspose(num_filters, (3, 3), strides=(2, 2), padding='same')(ir
    x = concatenate([x, skip_features])

    x = Conv2D(num_filters, (3, 3), padding='same')(x)
    x = Dropout(dropout_rate)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
```

```
    x = Conv2D(num_filters, (3, 3), padding='same')(x)
    x = Dropout(dropout_rate)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    return x

img_width, img_height = 256, 256

def unet_model(input_shape = (img_width, img_height, 3)):
  inputs = Input(input_shape)

  # Encoder
  e1, p1 = encoder_block(inputs, 16, 0.2)
  e2, p2 = encoder_block(p1, 32, 0.2)
  e3, p3 = encoder_block(p2, 64, 0.2)
  e4, p4 = encoder_block(p3, 128, 0.2)

  # Spatial Attention
  sa = spatial_attention_block(p4)

  # Decoder
  d1 = decoder_block(sa, e4, 128, 0.2)
  d2 = decoder_block(d1, e3, 64, 0.2)
  d3 = decoder_block(d2, e2, 32, 0.2)
  d4 = decoder_block(d3, e1, 16, 0.2)

  # Output
  outputs = Conv2D(filters=1, kernel_size =(1, 1), activation='sigmoid')(d4)

  return Model(inputs=[inputs], outputs = [outputs])
```

Setting Up Training Parameters

Here, we define hyperparameters for model training. The model is trained for 50 epochs with some augmentations.

```
epochs = 50
batch_size = 32
lr = 1e-4

train_generator_args = dict(
    rotation_range = 0.25,
```

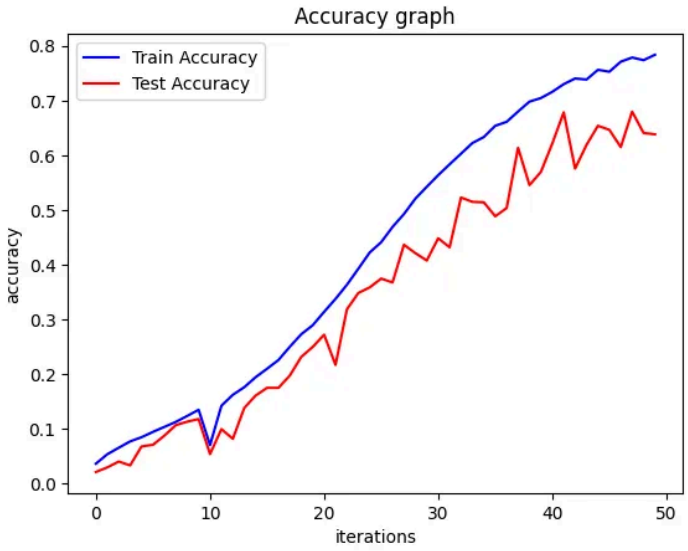
```
    width_shift_range = 0.05,
    height_shift_range=0.05,
    shear_range=0.05,
    zoom_range=0.05,
    horizontal_flip=True,
    fill_mode='nearest'
)

train_gen  = train_generator( df_train, batch_size, train_generator_args)
test_gen = train_generator(df_test, batch_size, dict() )

model = unet_model(input_shape =(256,256, 3))
optimizer = Adam(learning_rate = lr , beta_1 = 0.9, beta_2 = 0.999, epsilon = N

model.compile(optimizer=optimizer, loss=dice_coefficients_loss, metrics=["binar
callbacks = [
    ModelCheckpoint('unet_model.hdf5',verbose=1, save_best_only=True )
]

history = model.fit(
      train_gen,
      steps_per_epoch = len(df_train) / batch_size,
      epochs = epochs,
      callbacks = callbacks,
      validation_data = test_gen,
      validation_steps = len(df_validation) / batch_size
)
```

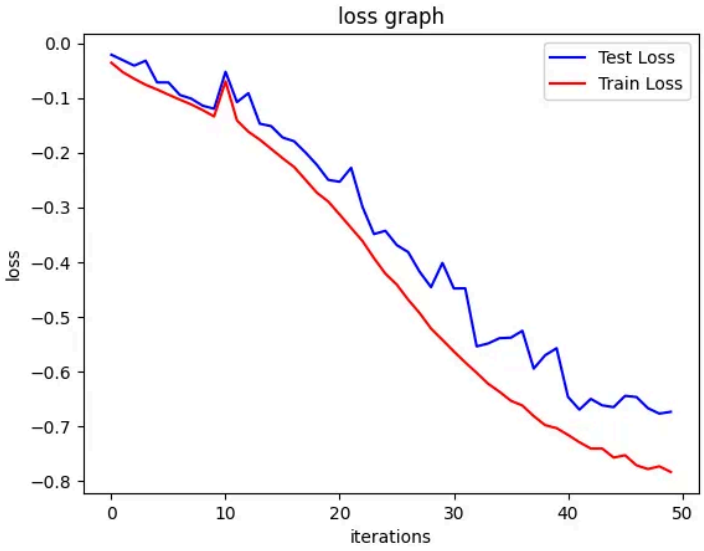
Visualize the Model and Graphs

Now the training is done we will see how the model has performed by tracking the progress across each epoch. The graphs for loss and accuracy will throw some light on it.

```
from tensorflow import keras
keras.utils.plot_model(model, to_file = "Model.png", show_shapes=True)

history_training = history.history

train_dice_coeff_list = history_training['dice_coefficients']
test_dice_coeff_list = history_training['val_dice_coefficients']

train_jaccard_list = history_training['iou']
```

```
test_jaccard_list = history_training['val_iou']

train_loss_list = history_training['loss']
test_loss_list = history_training['val_loss']


plt.plot(test_loss_list, 'b-', label='Test Loss')
plt.plot(train_loss_list, 'r-', label='Train Loss')

plt.xlabel('iterations')
plt.ylabel('loss')
plt.title('loss graph', fontsize=12)
plt.legend()
plt.show()
plt.savefig('Loss Graph')

plt.plot(train_dice_coeff_list, 'b-', label='Train Accuracy')
plt.plot(test_dice_coeff_list, 'r-', label = 'Test Accuracy')

plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.title('Accuracy graph', fontsize=12)
plt.legend()
plt.show()
plt.savefig('Accuracy Graph')
```





Model Evaluation

Here we will save the trained model parameters to a file and load them for evaluation. This will give us the metrics computed on test dataset.

```
model = load_model('unet_model.hdf5', custom_objects={'dice_coefficients_loss':

test_gen = train_generator(df_test, batch_size, dict(), target_size=(img_height
results = model.evaluate(test_gen, steps = len(df_test)/batch_size)
print('Test Loss', results[0])
print('Test IOU', results[1])
print('Test Dice Coeff', results[2])
```

Visualize Predictions

This piece of code does some post-processing to display the inference images.

```
for i in range(20):
    index = np.random.randint(0, len(df_test.index))
    img_path = df_test['image_path'].iloc[index]
    mask_path = df_test['mask_path'].iloc[index]

    # Read and preprocess the image
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img_resized = cv2.resize(img, (256, 256))
    img_normalized = img_resized / 255.0
    img_expanded = np.expand_dims(img_normalized, axis=0)

    # Generate the prediction
    pred_mask = model.predict(img_expanded)
    pred_mask_thresholded = (pred_mask.squeeze() > 0.5).astype(np.float32)

    # Read and preprocess the mask
    original_mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
    original_mask_resized = cv2.resize(original_mask, (256, 256))
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))  # Width, Height

    # Display the original image
    axs[0].imshow(img_resized)
    axs[0].set_title("Original Image")
    axs[0].axis('off')  # Hide axes ticks

    # Display the original mask
    axs[1].imshow(original_mask_resized, cmap='gray')
    axs[1].set_title("Original Mask")
    axs[1].axis('off')
```

```
    # Display the predicted mask
    axs[2].imshow(pred_mask_thresholded, cmap='gray')
    axs[2].set_title("Predicted Mask")
    axs[2].axis('off')

    plt.show()
```

Conclusion

I hope this guide has illuminated the path through brain MRI segmentation with SA-UNet, showing its effectiveness beyond traditional applications. By leveraging the LGG-MRI-Segmentation dataset, we've underlined the model's versatility in medical imaging, providing deep insights into brain anatomy and pathologies. This project underscores the transformative potential of deep learning in enhancing diagnostic precision.

For any inquiries or feedback, feel free to connect. Your experiences and challenges are invaluable to further refining and exploring the potentials of this technology.

References

For a deeper understanding of the methodologies and tools used in this project, consider exploring the following resources:

- Brain MRI Segmentation Repository
- Kaggle LGG MRI Segmentation Dataset
- SA-UNet Model
- SA-UNet Official Repository

This project stands as a testament to applying cutting-edge deep learning techniques in medical imaging, pushing the boundaries of what is possible in diagnostics and treatment planning.

Computer Vision | Deep Learning | Machine Learning | Segmentation

Edit profile

Written by Sunidhi Ashtekar

101 Followers

A Passionate Computer Vision & Machine Learning Engineer

More from Sunidhi Ashtekar