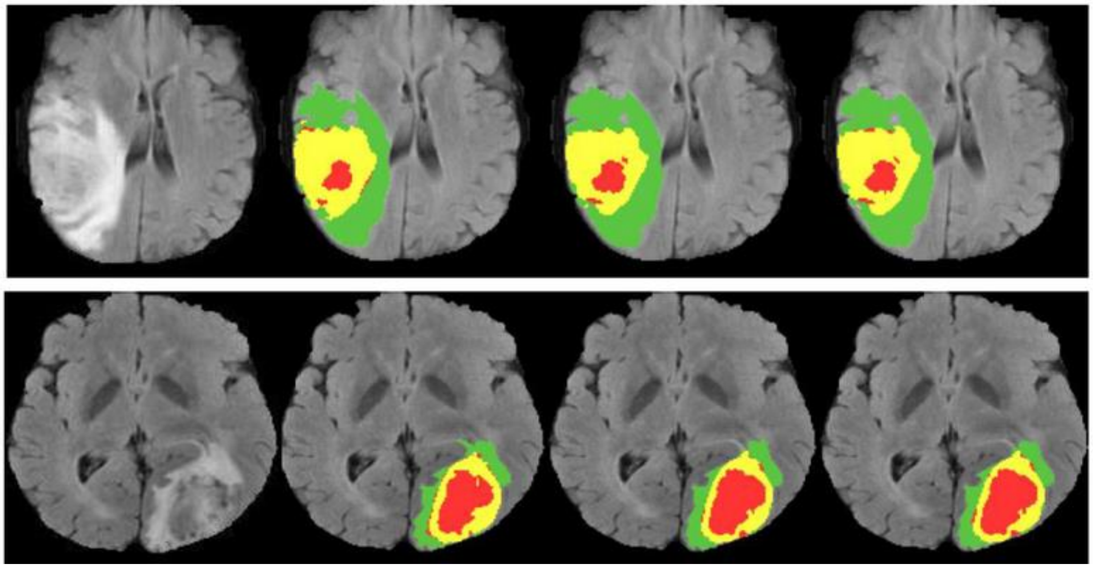
Brain Tumor Segmentation using Vision Transformer (ViT)

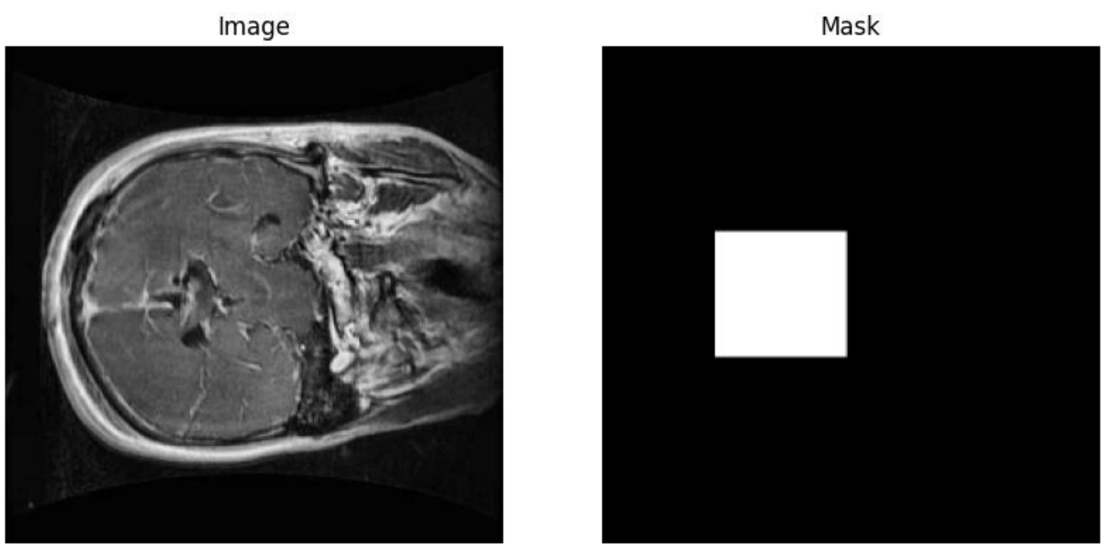


```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras import backend as K

from pycocotools.coco import COCO

train_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/train'
val_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/valid'
test_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/test'

train_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/train/_annotations.coco.json'
test_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/test/_annotations.coco.json'
val_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/valid/_annotations.coco.json'

train_coco = COCO(train_annotation_file)
val_coco = COCO(val_annotation_file)
test_coco = COCO(test_annotation_file)
```

```
loading annotations into memory...
Done (t=0.01s)
creating index...
index created!
loading annotations into memory...
Done (t=0.01s)
creating index...
index created!
loading annotations into memory...
Done (t=0.00s)
creating index...
index created!
```

```
def load_image_and_mask(coco, image_dir, image_id):
    image_info = coco.loadImgs(image_id)[0]
    image_path = os.path.join(image_dir, image_info['file_name'])
    image = Image.open(image_path)
    image = np.array(image)

    ann_ids = coco.getAnnIds(imgIds=image_id)
    anns = coco.loadAnns(ann_ids)
    mask = np.zeros((image_info['height'], image_info['width']))
    for ann in anns:
        mask = np.maximum(mask, coco.annToMask(ann))

    return image, mask

def create_tf_dataset(coco, image_dir, image_ids):
    def generator():
        for image_id in image_ids:
            yield load_image_and_mask(coco, image_dir, image_id)

    return tf.data.Dataset.from_generator(generator,

output_signature=(tf.TensorSpec(shape=(None, None, 3), dtype=tf.uint8),

tf.TensorSpec(shape=(None, None), dtype=tf.uint8)))

train_dataset = create_tf_dataset(train_coco, train_dir,
train_coco.getImgIds())
val_dataset = create_tf_dataset(val_coco, val_dir, val_coco.getImgIds())
test_dataset = create_tf_dataset(test_coco, test_dir, test_coco.getImgIds())

def preprocess(image, mask):

    image = tf.image.resize(image, (256, 256))

    mask = tf.expand_dims(mask, axis=-1)
    mask = tf.image.resize(mask, (256, 256))
```

```
    image = tf.cast(image, tf.float32) / 255.0

    return image, mask

train_dataset = train_dataset.map(preprocess)
val_dataset = val_dataset.map(preprocess)
test_dataset = test_dataset.map(preprocess)

def visualize_dataset(dataset, num_samples=5):
    for i, (image, mask) in enumerate(dataset.take(num_samples)):
        plt.figure(figsize=(10, 5))

        plt.subplot(1, 2, 1)
        plt.imshow(image.numpy())
        plt.title("Image")
        plt.axis("off")

        plt.subplot(1, 2, 2)
        plt.imshow(mask.numpy().squeeze(), cmap="gray")
        plt.title("Mask")
        plt.axis("off")

        plt.show()

visualize_dataset(train_dataset)
visualize_dataset(val_dataset)
```



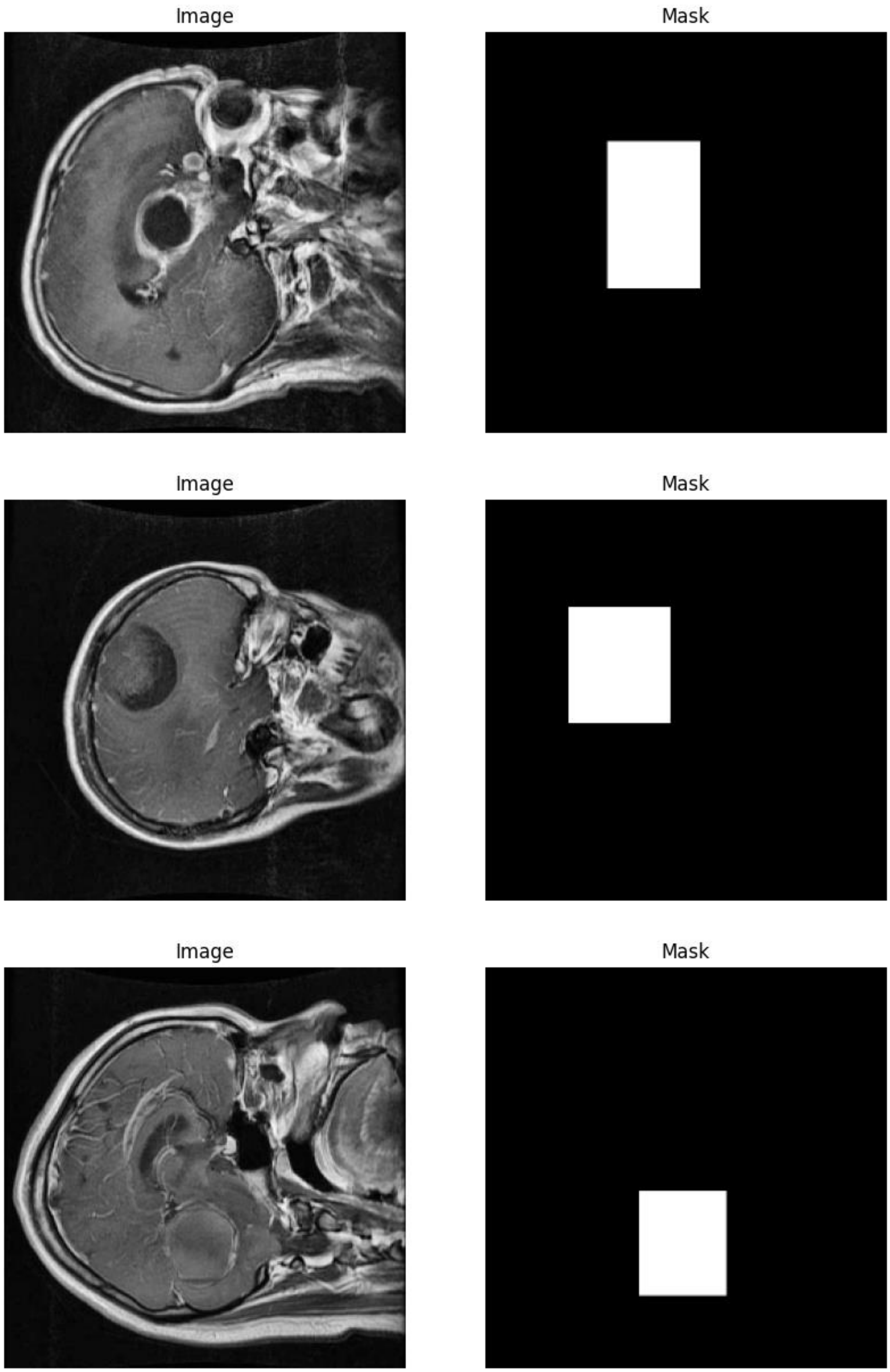


Image
Mask
Image
Mask
Image
Mask

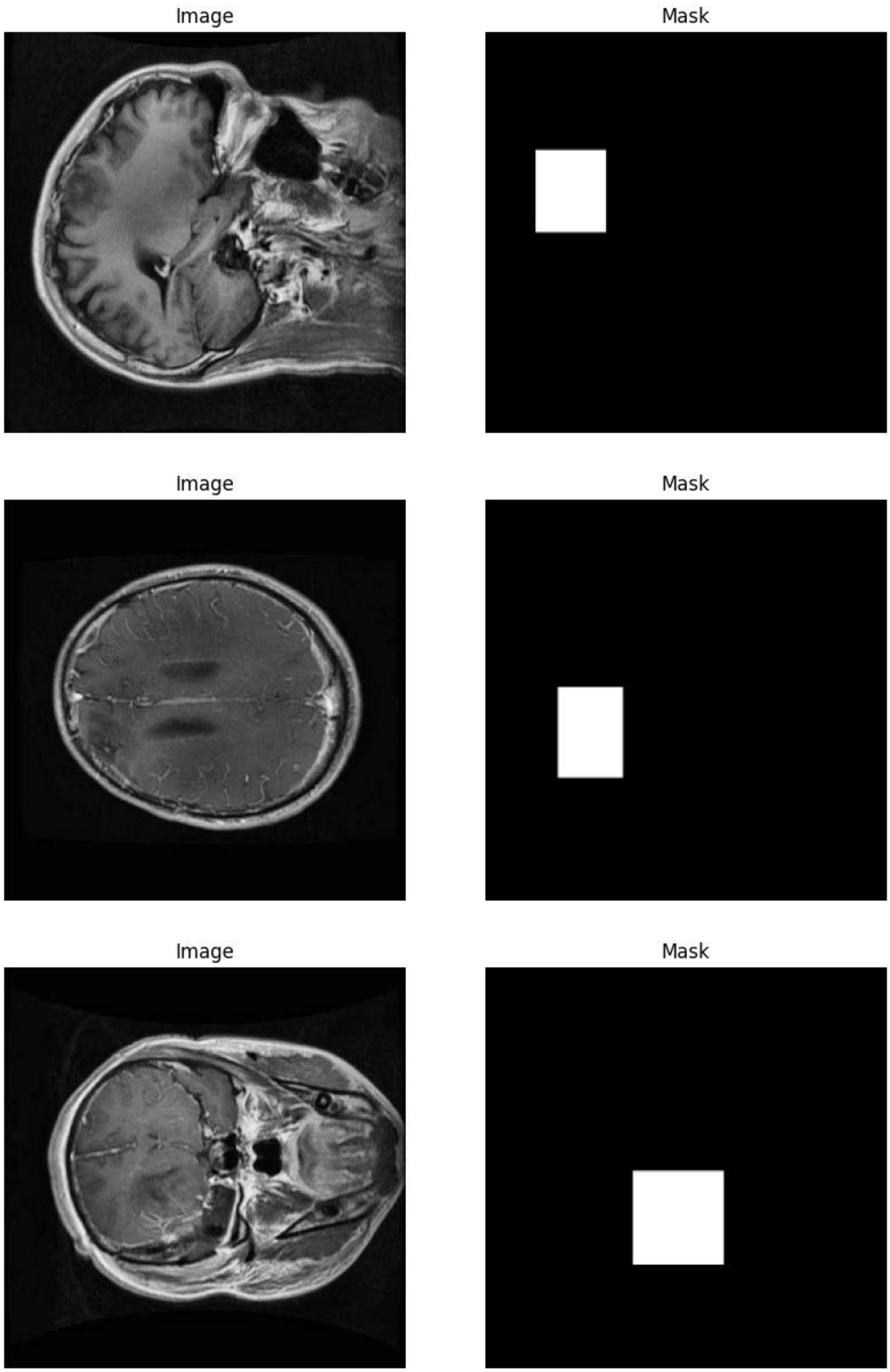


Image
Mask
Image
Mask
Image
Mask

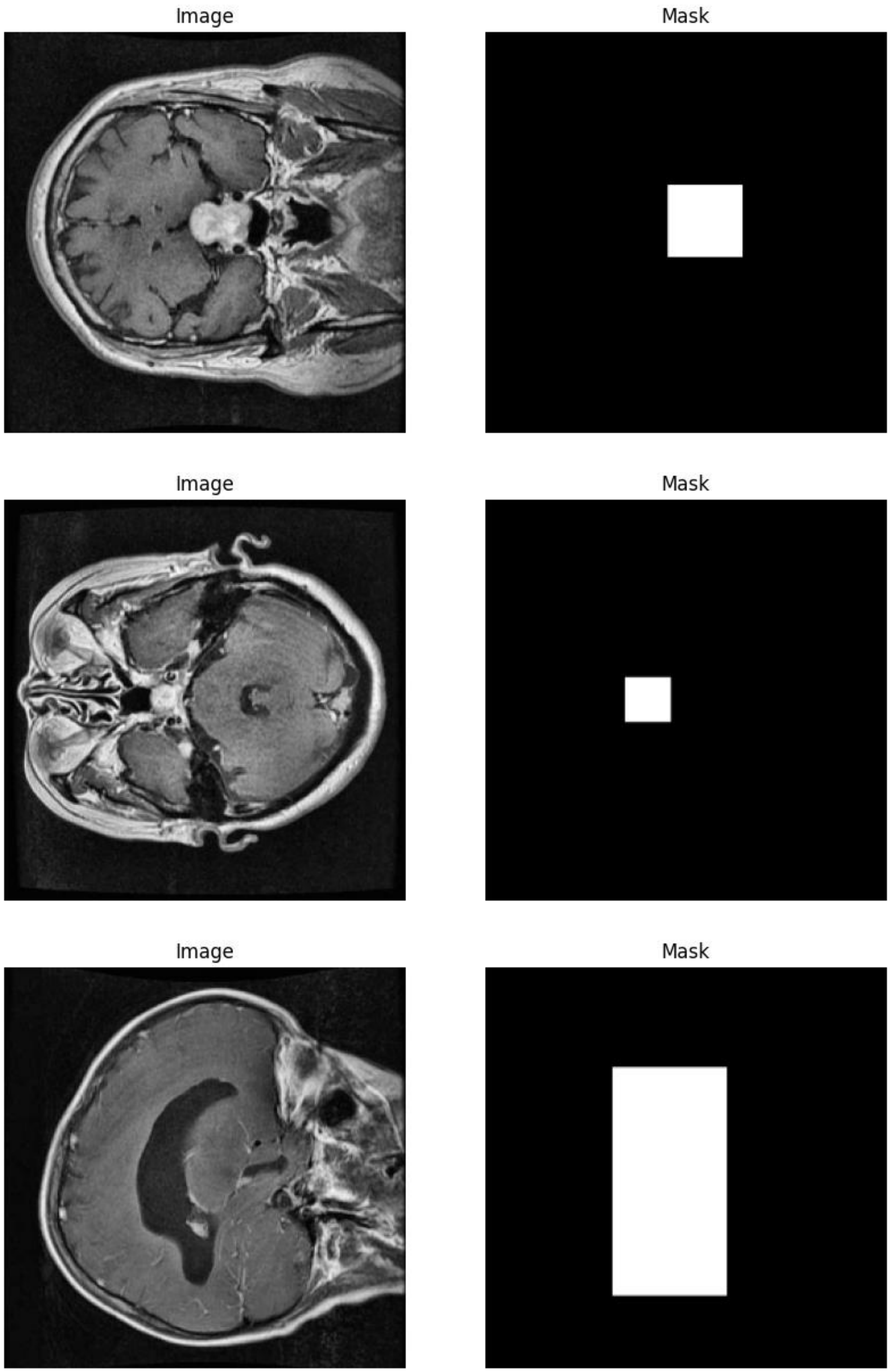


Image
Mask
Image
Mask
Image
Mask

```python
class PatchEmbedding(layers.Layer):
    def __init__(self, patch_size, embed_dim):
        super(PatchEmbedding, self).__init__()
        self.patch_size = patch_size
        self.embed_dim = embed_dim
        self.conv = layers.Conv2D(embed_dim, kernel_size=patch_size,
strides=patch_size, padding="valid")

    def call(self, images):
        x = self.conv(images)
        x = tf.reshape(x, [tf.shape(x)[0], -1, self.embed_dim])
        return x

class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, mlp_dim, dropout_rate=0.1):
        super(TransformerBlock, self).__init__()
        self.attention = layers.MultiHeadAttention(num_heads=num_heads,
key_dim=embed_dim)
        self.dropout1 = layers.Dropout(dropout_rate)
        self.norm1 = layers.LayerNormalization(epsilon=1e-6)


        self.mlp = tf.keras.Sequential([
            layers.Dense(mlp_dim, activation="gelu"),
            layers.Dropout(dropout_rate),
            layers.Dense(embed_dim),
            layers.Dropout(dropout_rate),
        ])
        self.norm2 = layers.LayerNormalization(epsilon=1e-6)

    def call(self, inputs):

        attn_output = self.attention(inputs, inputs)
        attn_output = self.dropout1(attn_output)
        out1 = self.norm1(inputs + attn_output)


        mlp_output = self.mlp(out1)
        return self.norm2(out1 + mlp_output)

def upsampling_block(inputs, filters):

    conv = layers.Conv2D(filters, kernel_size=(3, 3), padding="same")
    batch_norm = layers.BatchNormalization()
    activation = layers.Activation("relu")
    upsample = layers.UpSampling2D(size=(2, 2), interpolation="bilinear")

    x = upsample(inputs)
    x = conv(x)
```

```
    x = batch_norm(x)
    x = activation(x)
    return x

class VisionTransformer(tf.keras.Model):
    def __init__(self, image_size, patch_size, embed_dim, num_heads,
num_blocks, mlp_dim, decoder_filters, dropout_rate=0.1):
        super(VisionTransformer, self).__init__()

        self.patch_embed = PatchEmbedding(patch_size, embed_dim)
        height, width, _ = image_size
        self.num_patches = (height // patch_size) * (width // patch_size)


        self.pos_embed = self.add_weight(name="pos_embed", shape=(1,
self.num_patches, embed_dim),

initializer=tf.initializers.RandomNormal(stddev=0.02), trainable=True)

        self.dropout = layers.Dropout(dropout_rate)
        self.transformer_blocks = [TransformerBlock(embed_dim, num_heads,
mlp_dim, dropout_rate) for _ in range(num_blocks)]
        self.norm = layers.LayerNormalization(epsilon=1e-6)


        self.upsample = layers.UpSampling2D(size=(2, 2),
interpolation="bilinear")
        self.conv_layers = [layers.Conv2D(filters, kernel_size=(3, 3),
padding="same") for filters in decoder_filters]
        self.batch_norm_layers = [layers.BatchNormalization() for _ in
decoder_filters]
        self.activation_layers = [layers.Activation("relu") for _ in
decoder_filters]

        self.final_conv = layers.Conv2D(1, (1, 1), activation="sigmoid")

    def upsampling_block(self, inputs, idx):

        x = self.upsample(inputs)
        x = self.conv_layers[idx](x)
        x = self.batch_norm_layers[idx](x)
        x = self.activation_layers[idx](x)
        return x

    def call(self, images):
        batch_size = tf.shape(images)[0]

        patches = self.patch_embed(images)
        patches += self.pos_embed
```

```
        patches = self.dropout(patches)

        for block in self.transformer_blocks:
            patches = block(patches)
        patches = self.norm(patches)


        height, width = 256, 256
        num_channels = patches.shape[-1]


        x = tf.reshape(patches, [batch_size, height // 16, width // 16,
num_channels])
        print("Shape after transformer and reshaping:", x.shape)


        for i in range(len(self.conv_layers)):
            x = self.upsampling_block(x, i)
            print(f"Shape after {i+1}th upsampling:", x.shape)


        x = self.final_conv(x)
        print("Shape after final convolution:", x.shape)

        return x

vit_model = VisionTransformer(
    image_size=(256, 256, 3),
    patch_size=16,
    embed_dim=128,
    num_heads=8,
    num_blocks=4,
    mlp_dim=256,
    decoder_filters=[128, 64, 32, 16],
    dropout_rate=0.1
)

initial_learning_rate = 1e-4
optimizer = tf.keras.optimizers.Adam(learning_rate=initial_learning_rate)

def dice_coef(y_true, y_pred, smooth=1e-6):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f)
+ smooth)

def dice_loss(y_true, y_pred):
    return 1 - dice_coef(y_true, y_pred)
```

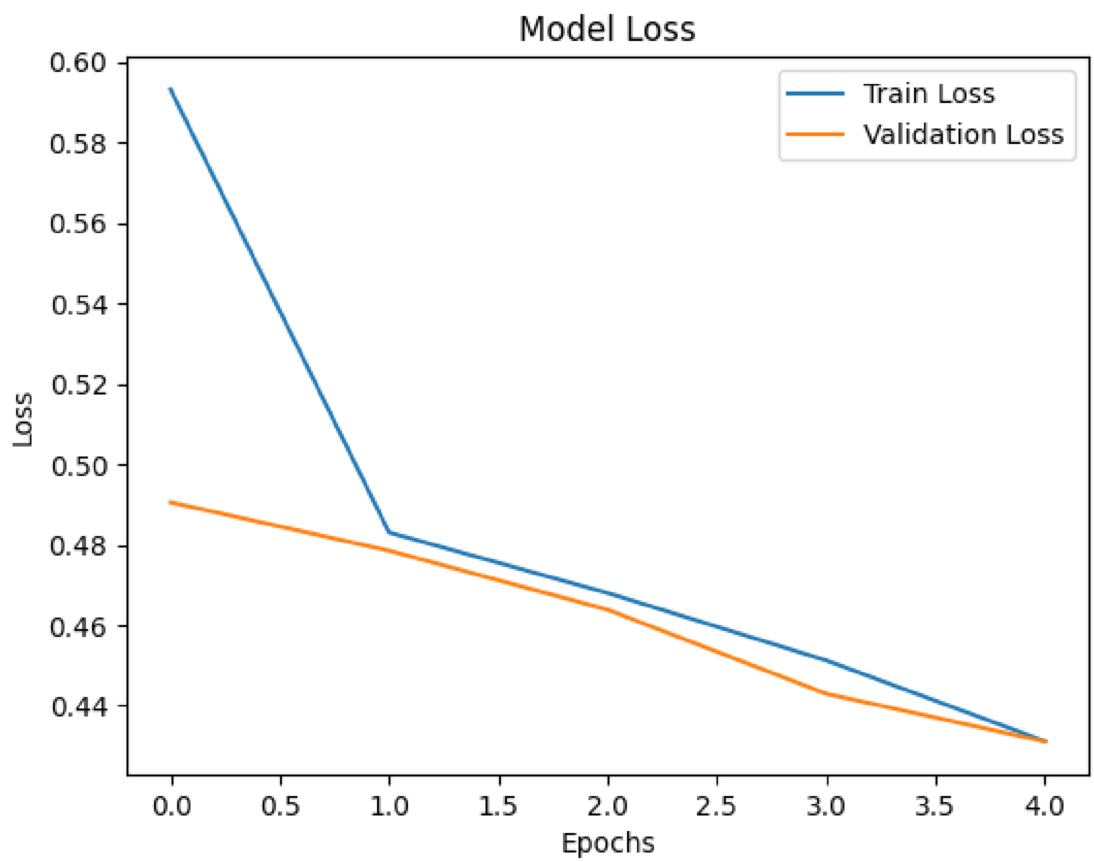
```
def combined_loss(y_true, y_pred):
    dice = dice_loss(y_true, y_pred)
    bce = tf.keras.losses.binary_crossentropy(y_true, y_pred)
    return 0.6 * dice + 0.4 * bce

vit_model.compile(optimizer=optimizer,
                  loss=combined_loss,
                  metrics=["accuracy", dice_coef])

callbacks = [
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=5, verbose=1),
    EarlyStopping(monitor="val_loss", patience=10, verbose=1)
]

history = vit_model.fit(train_dataset.batch(8),
              validation_data=val_dataset.batch(8),
              epochs=5,
              callbacks=callbacks)

Epoch 1/5
Shape after transformer and reshaping: (None, 16, 16, 128)
Shape after 1th upsampling: (None, 32, 32, 128)
Shape after 2th upsampling: (None, 64, 64, 64)
Shape after 3th upsampling: (None, 128, 128, 32)
Shape after 4th upsampling: (None, 256, 256, 16)
Shape after final convolution: (None, 256, 256, 1)
Shape after transformer and reshaping: (None, 16, 16, 128)
Shape after 1th upsampling: (None, 32, 32, 128)
Shape after 2th upsampling: (None, 64, 64, 64)
Shape after 3th upsampling: (None, 128, 128, 32)
Shape after 4th upsampling: (None, 256, 256, 16)
Shape after final convolution: (None, 256, 256, 1)
Shape after transformer and reshaping: (None, 16, 16, 128)
Shape after 1th upsampling: (None, 32, 32, 128)
Shape after 2th upsampling: (None, 64, 64, 64)
Shape after 3th upsampling: (None, 128, 128, 32)
Shape after 4th upsampling: (None, 256, 256, 16)
Shape after final convolution: (None, 256, 256, 1)
    188/Unknown 41s 110ms/step - accuracy: 0.9009 - dice_coef: 0.0827 - loss:
0.6674Shape after transformer and reshaping: (None, 16, 16, 128)
Shape after 1th upsampling: (None, 32, 32, 128)
Shape after 2th upsampling: (None, 64, 64, 64)
Shape after 3th upsampling: (None, 128, 128, 32)
Shape after 4th upsampling: (None, 256, 256, 16)
Shape after final convolution: (None, 256, 256, 1)
188/188 ━━━━━━━━━━━━━━━━━━━━49s 154ms/step - accuracy: 0.9011 - dice_coef:
0.0830 - loss: 0.6670 - val_accuracy: 0.9298 - val_dice_coef: 0.2861 -
val_loss: 0.4905 - learning_rate: 1.0000e-04
Epoch 2/5
188/188 ━━━━━━━━━━━━━━━━━━━━20s 107ms/step - accuracy: 0.9268 - dice_coef:
```

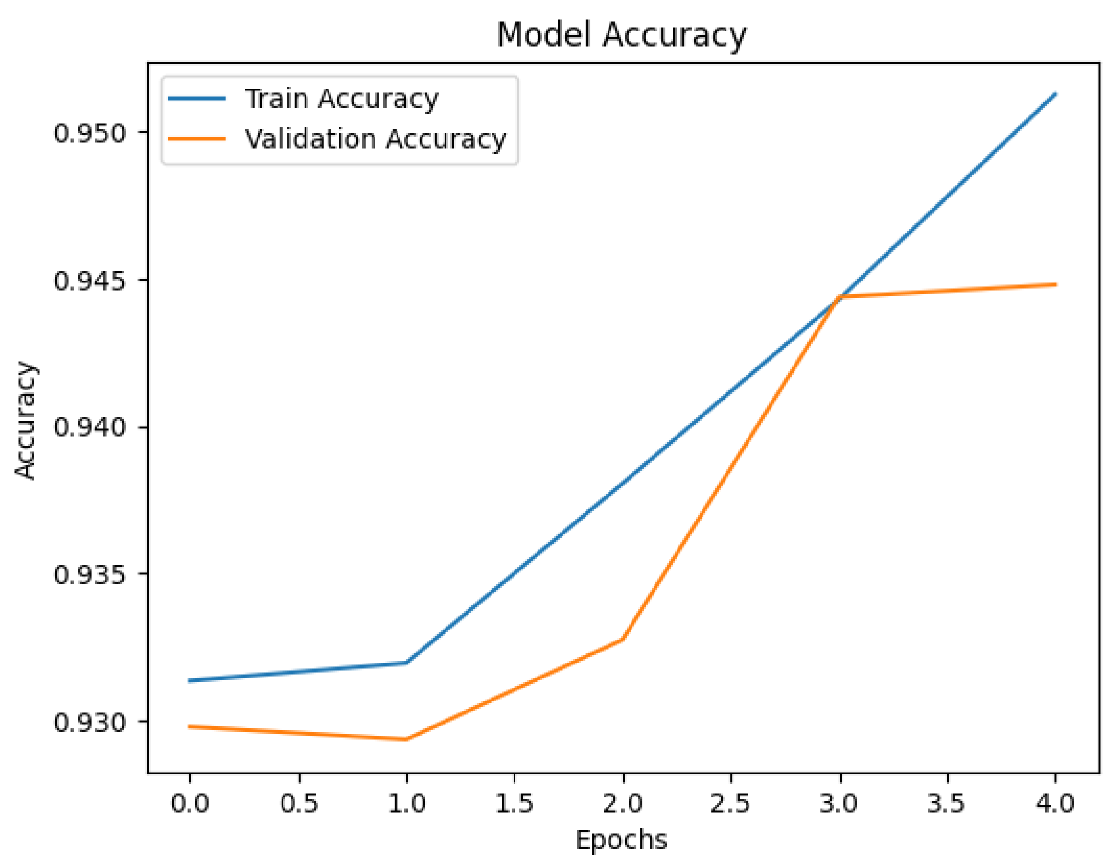
```
0.2911 - loss: 0.4922 - val_accuracy: 0.9294 - val_dice_coef: 0.3214 -
val_loss: 0.4785 - learning_rate: 1.0000e-04
Epoch 3/5
188/188 ━━━━━━━━━━━━━━━━━━━━21s 111ms/step - accuracy: 0.9321 - dice_coef:
0.3101 - loss: 0.4796 - val_accuracy: 0.9328 - val_dice_coef: 0.3371 -
val_loss: 0.4638 - learning_rate: 1.0000e-04
Epoch 4/5
188/188 ━━━━━━━━━━━━━━━━━━━━16s 86ms/step - accuracy: 0.9374 - dice_coef:
0.3278 - loss: 0.4667 - val_accuracy: 0.9444 - val_dice_coef: 0.3777 -
val_loss: 0.4430 - learning_rate: 1.0000e-04
Epoch 5/5
188/188 ━━━━━━━━━━━━━━━━━━━━18s 98ms/step - accuracy: 0.9469 - dice_coef:
0.3569 - loss: 0.4468 - val_accuracy: 0.9448 - val_dice_coef: 0.3931 -
val_loss: 0.4311 - learning_rate: 1.0000e-04
```

```
test_loss, test_accuracy, test_dice_coef =
vit_model.evaluate(test_dataset.batch(8))
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Test Dice Coefficient: {test_dice_coef}")
```

```
27/27 ━━━━━━━━━━━━━━━━━━━━4s 140ms/step - accuracy: 0.9425 - dice_coef:
0.3842 - loss: 0.4405
Test Loss: 0.4380626678466797
Test Accuracy: 0.9441211223602295
Test Dice Coefficient: 0.38198113441467285
```

```
# Plot loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```


Model Loss
Train Loss
Validation Loss
Loss
Epochs
0.60
0.58
0.56
0.54
0.52
0.50
0.48
0.46
0.44
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Model Accuracy
Train Accuracy
Validation Accuracy
Accuracy
Epochs
0.950
0.945
0.940
0.935
0.930
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0