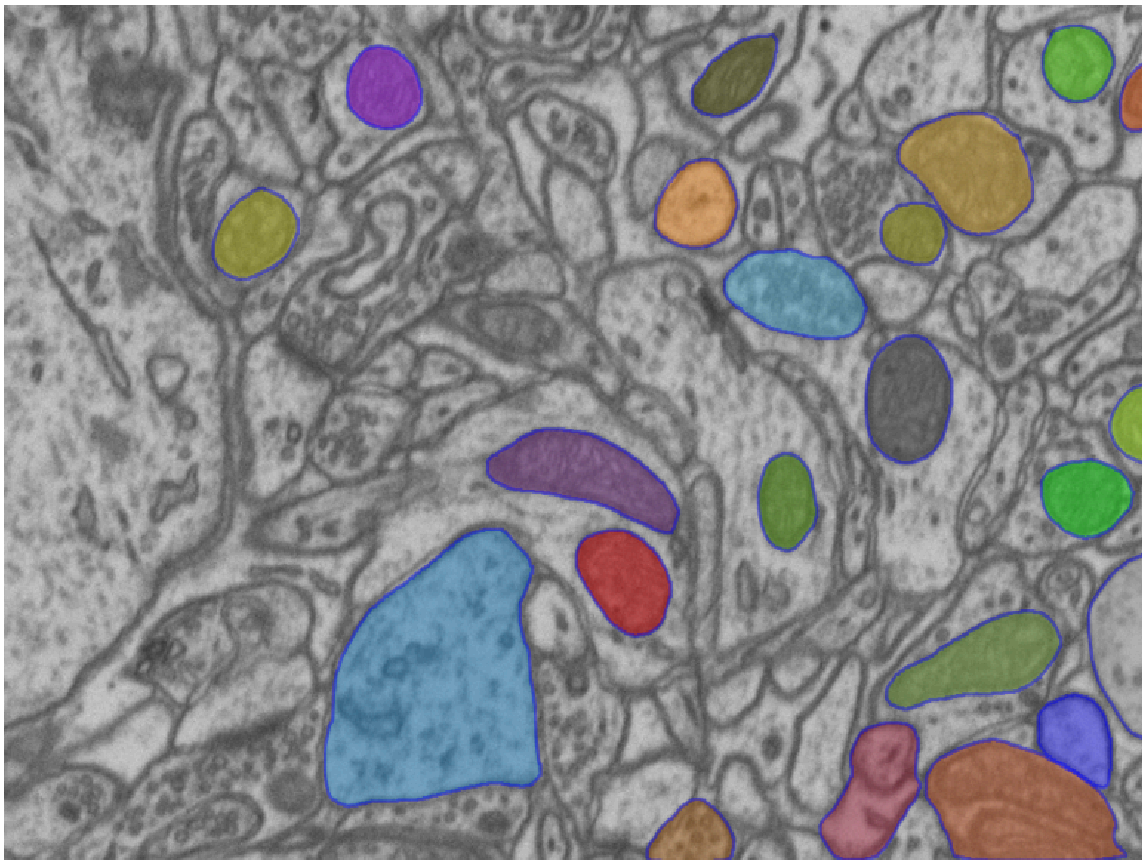
Computer Vision
MAZUST University


Picture segmented with segment-anything.com

Detection Mitochondria images Using SAM model (as end user) & U-Net custom model

Master: **Dr. Jeddisaravi**

Student: **Mohammd Mousapour** | mohammadmoosapoor4@gmail.com

Source Code

Table Of Content

Source Code

Abstract

Tracking mitochondria in cellular images is crucial for understanding their dynamics, morphology, and roles in diseases. Advanced segmentation models like U-Net and SAM are commonly used to enhance accuracy and efficiency in these tasks.

This project aims to develop a robust system for detecting and tracking objects in scientific images. The focus is on comparing U-Net and SAM models to determine which performs better in terms of accuracy and speed. Using machine vision techniques, we will apply masks to images, analyze the results, and compare their performance.

What Is Segment Anything

Website: Segment Anything

The Segment Anything Model (SAM) is an instance segmentation model developed by Meta Research and released in April 2023. A versatile segmentation model that leverages transformer-based architectures for accurate and adaptable segmentation across different image domains. SAM can segment various objects within an image without needing retraining for each specific object type.

It trained over 11 billion segmentation masks from millions of images.

It is designed to take human prompts, in the form of points, bounding boxes or even a text prompt describing what should be segmented. Also, it can auto segment the image, which gives us all objects and masks on that image.

Source Code

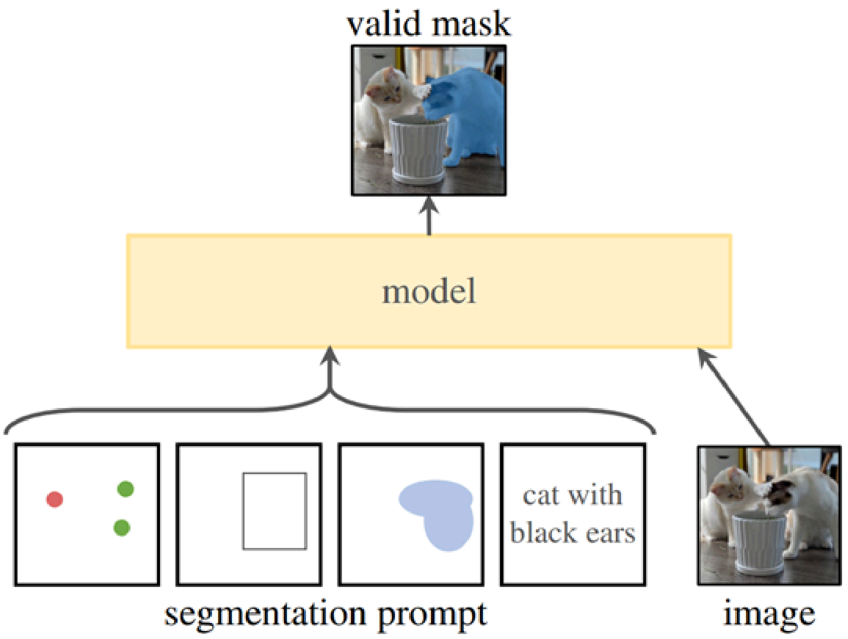
How Does SAM Work

SAM works by first encoding the image into a high-dimensional vector representation. The prompt is encoded into a separate vector representation.

The two vector representations are then combined and passed to a mask decoder, which the object specified by the prompt.

The image encoder is a vision transformer (VIT-H) model, which is a large language (a language model can be used for image analysis tasks by first encoding the image into a text representation) model that has been pre-trained on a massive dataset of images.

The prompt encoder is a simple text encoder that converts the input prompt into a vector representation.



The mask decoder is a lightweight transformer model that predicts the object mask from the image and prompt embeddings.

Why Do We Use SAM

Our primary goal is to detect mitochondria. Additionally, we need to provide the model with reference points or inputs. There are two possible approaches to achieve this:

1. Manual Point Selection: In this approach, the user manually selects the points and provides them to the model. However, this method is neither practical nor efficient.
2. Automated Point Generation: A separate model generates the points automatically. Every time we input an image, the model identifies the points and feeds them into SAM for detection.

This second approach is more scalable and efficient for automating the process of mitochondria detection.

What Is The U-Net Model

U-Net is a convolutional neural network architecture that was originally developed for biomedical image segmentation. It is particularly well-suited for tasks that involve dense prediction, where the goal is to assign a class label to every pixel in an input image. The U-Net architecture has proven to be very effective for a variety of image segmentation tasks, not just in the biomedical domain but also in areas like autonomous driving, satellite imagery analysis, and general computer vision. Its ability to capture both local and global information in an image makes it a powerful and versatile model.

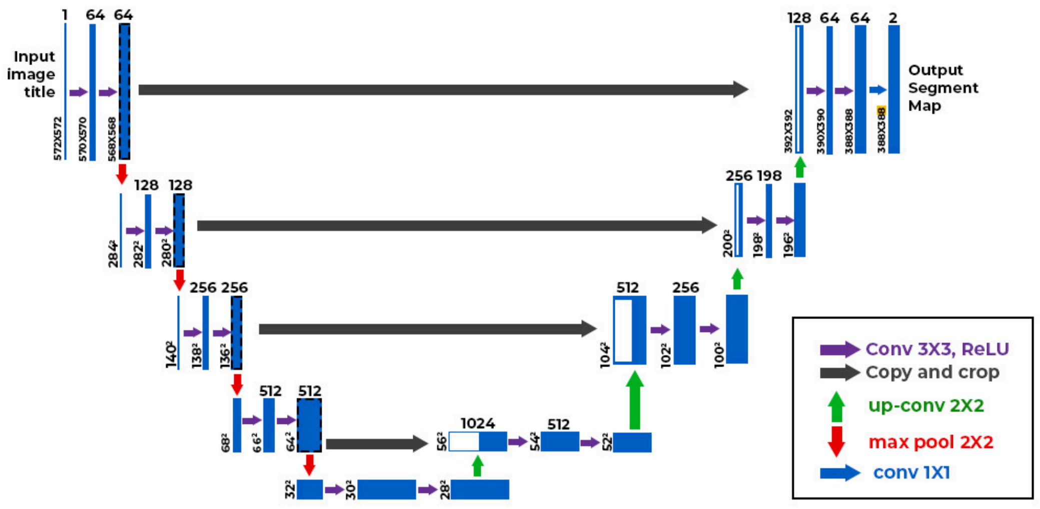
How Does U-Net Model Work

U-Net has a distinctive "U" shaped architecture, consisting of two main parts:

1. Contracting Path (Downsampling): Similar to a typical convolutional network, this path reduces image dimensions while increasing feature maps.
2. Expansive Path (Upsampling): This path reverses the contracting path operation and uses skip connections to pass feature maps from the contracting path to the expansive path, helping with precise localization and learning representations for segmentation.

The U-Net model processes images as follows:

- Input Image
- Encoder (Contracting Path)
- Decoder (Expansive Path)
- Skip Connections
- Output Segmentation map



The U-Net model captures both local and global information for image segmentation. Its binary outputs (white for objects, black for background) are ideal for creating precise masks used as ground truth labels, enabling detailed analysis and improving segmentation or tracking performance, especially in scientific tasks like mitochondria tracking.

If you wants to know more about U-Net model, use this article: U-Net Architecture Explained - GeeksforGeeks

Source Code

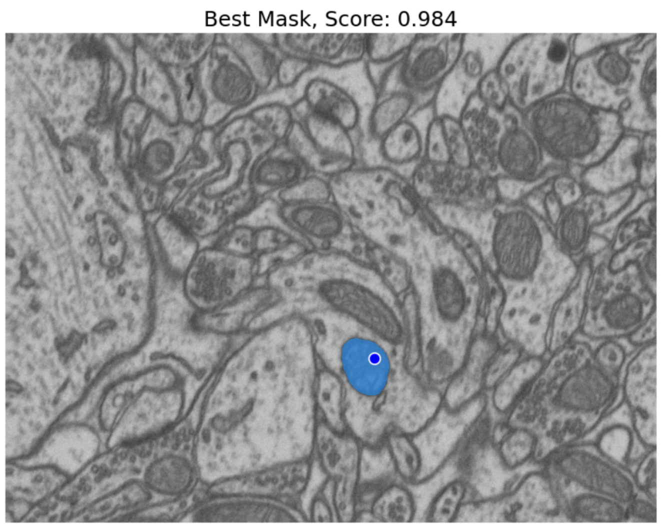
Applications

The use of mitochondrial tracking and detection technology, employing models like U-Net, provides significant support to medical practitioners across various medical domains.

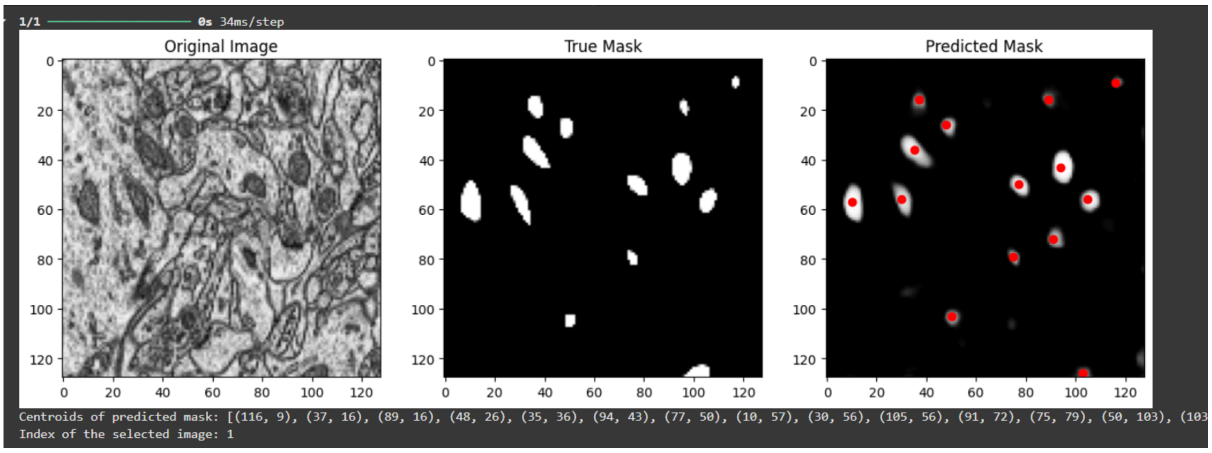
1. Disease Diagnosis Linked to Mitochondria: Precise identification of mitochondria aids in early detection of mitochondrial-related diseases, enabling timely and appropriate treatment.
2. Therapy Monitoring: Mitochondrial detection allows healthcare professionals to monitor the efficacy of different treatments, ensuring continuous improvement in patient care.
3. Research and Drug Development: A deeper understanding of mitochondrial status and function supports medical research, fostering the development of novel drugs and therapeutic approaches.

By utilizing mitochondrial detection through models like U-Net, medical practitioners can enhance disease diagnosis, treatment, and cellular health monitoring effectively.

Results

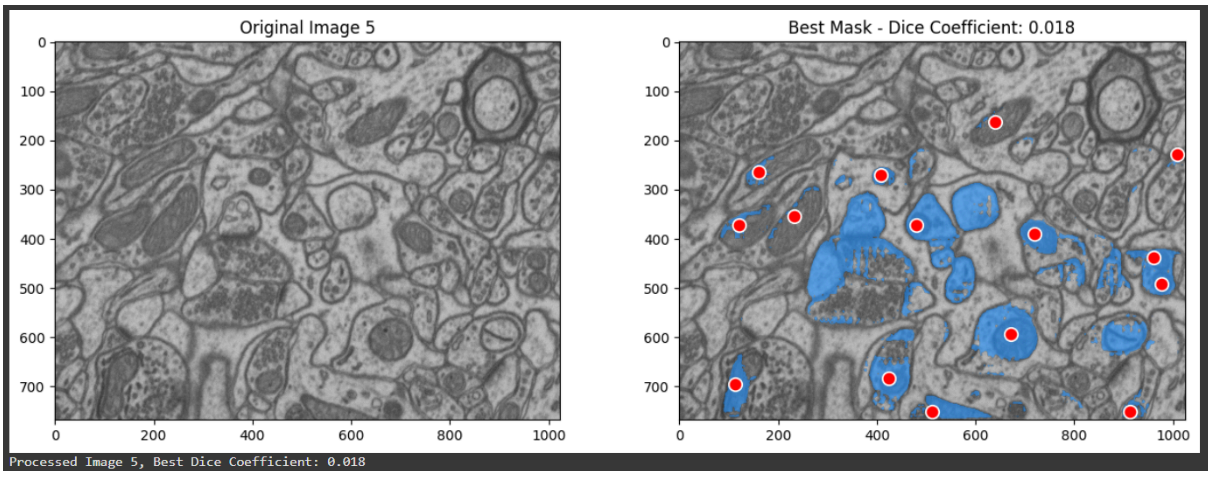


Source Code

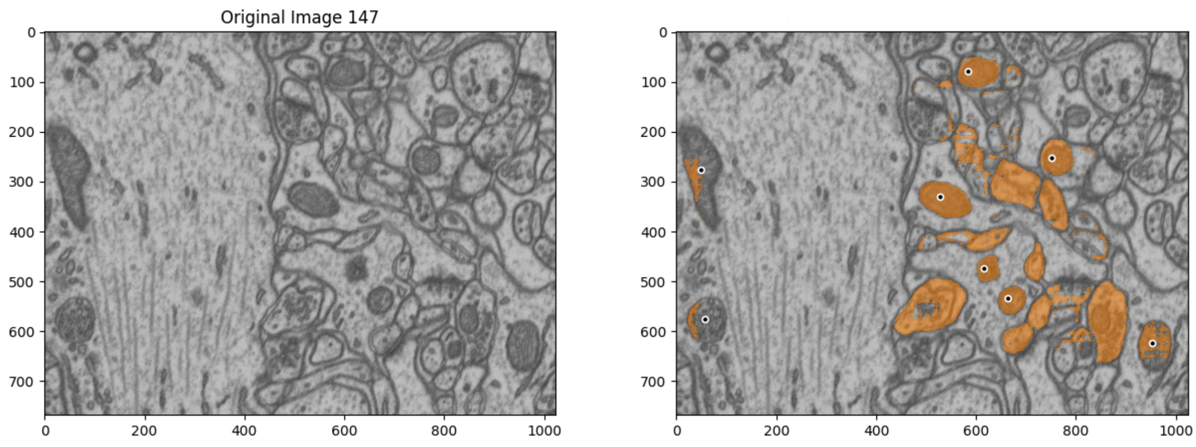


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
```

```
Centroids of predicted mask: [(116, 9), (37, 16), (89, 16), (48, 26), (35, 36), (94, 43), (77, 50), (10, 57), (30, 56), (105, 56), (91, 72), (75, 79), (50, 103), (103
Index of the selected image: 1
```



```
Processed Image 5, Best Dice Coefficient: 0.018
```



Source Code

Code Review (SAM)

In this section, we have some static code structures that are utilized in both approaches.
Our dataset is a **tiff** file, so we have to unpack this dataset first, we'll do this with tifffile library.

Main Functions & Single Point

To load pictures from the dataset, use `tiff.imread` and give that the path of the dataset.

To display an image, we prompt the user to provide the number of the picture. After that, with the `select_and_show_image` class, we can display the image that we want.

To display the point selected by the user on the current image, we use the following function:

```
def show_points(coords, labels, ax, marker_size=375):
    pos_points = coords[labels == 1]
    ax.scatter(pos_points[:, 0], pos_points[:, 1], color='red',
marker='.', s=marker_size, edgecolor='white', linewidth=1.25)
```

The `show_mask` function overlays a semi-transparent mask on an image. It takes in the mask, a matplotlib axis (`ax`), and an optional `random_color` flag to use a random color or a default blue color. The mask is resized to match the image dimensions and blended with the chosen color before being displayed on the axis.

```
def show_mask(mask, ax, random_color=False):
    if random_color:
        color = np.concatenate([np.random.random(3), np.array([0.6])],
axis=0)
    else:
        color = np.array([30/255, 144/255, 255/255, 0.6])
    h, w = mask.shape[-2:]
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    ax.imshow(mask_image)
```

The we have to set our model, our device and more settings (these settings are available in Meta official repository)

```
sam_checkpoint = 'path-to-sam-predictor'
device = 'cuda'
model_type = 'default'
sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)
predictor = SamPredictor(sam)
predictor.set_image(the_img)
```

You can get this SAM predictor from this website: [sam_vit_h_4b8939.pth · abhishek/StableSAM at main]()

Also you will need CUDA for this project, so install it from here: [CUDA Toolkit 12.6 Update 3 Downloads | NVIDIA Developer]()

After completing these steps, we need to obtain the mask from the SAM model. For this, we use the following lines of code:

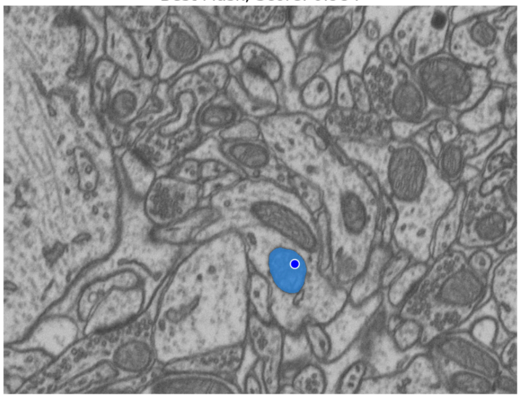
```
masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    multimask_output=True)

best_index = np.argmax(scores)

plt.figure(figsize=(10, 10))
plt.imshow(the_img)
show_mask(masks[best_index], plt.gca())
show_points(input_point, input_label, plt.gca())
plt.title(f"Best Mask, Score: {scores[best_index]:.3f}", fontsize=18)
plt.axis('off')
plt.show()
```

And here is an example result:

Best Mask, Score: 0.984



Now, with these elements in place, if we have the center of each mitochondrion in the image, we can easily detect and display all mitochondria in a single step using the SAM model.

To obtain the centers of each mitochondrion, we train a model. The model is provided with the image and its corresponding mask during training, and it generates binary masks for the mitochondria. Using **computer vision** techniques, we can extract the center points from these masks. Finally, these center points, along with the image index, are saved in a file. For this training, we use the **U-Net** model.

Source Code

Code Review (U-Net)

In this section, we don't have a pre-trained model anymore. First, we need to train the U-Net model using the datasets, build our custom model, and then proceed to test it.

Resize The Images

The key point here is the lack of a GPU. Training this model using the original quality of the dataset requires a significant amount of GPU resources, which are not available. Therefore, before feeding the images into the model, we resize them.

```python
def resize_images(images, new_size=(128, 128)):
    resized_images = np.zeros((images.shape[0], new_size[0], new_size[1],
images.shape[3]))
    for i in range(images.shape[0]):
        resized_images[i] = tf.image.resize(images[i], new_size)
    return resized_images
```

Train U-Net Model

Now it's time to proceed with the step-by-step implementation of the training process for the U-Net model.

Input Layer

```python
inputs = layers.Input(input_size)
```

This line defines the input layer for the image. The input size `(128, 128, 1)` specifies:

- `128x128`: The width and height of the image.
- `1`: The number of channels (grayscale image). For a color image, this would be `3`

This part of the code pertains to the **contracting path** of the U-Net model, which is responsible for **feature extraction** from the input images.

Block 1: Extracting Low-Level Features

First Convolution

```
c1 = layers.Conv2D(64, (3, 3), activation='relu',
padding='same')(inputs)
```

- `Conv2D`: Applies 64 filters of size `(3, 3)` to the input image.
- **Filters**: These filters try to detect basic patterns such as edges, corners, or simple textures in the image.
- `activation='relu'`: The ReLU activation function sets negative values to zero while leaving positive values unchanged. This helps speed up learning and avoids gradient issues.
- `padding='same'`: Ensures that the output dimensions are the same as the input by padding the edges of the image with zeros.

Second Convolution

```
c1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c1)
```

- Performs the same operation on the output of the first convolution layer.
- Using consecutive convolutional layers allows the model to learn more complex features from the data.

Max-Pooling

```
p1 = layers.MaxPooling2D((2, 2))(c1)
```

- `MaxPooling2D`: Reduces the spatial dimensions by half (from `128x128` to `64x64`) using a `(2, 2)` filter with stride 2.
- In each 2x2 region, the maximum value is selected. This helps preserve important features while discarding less significant ones.
- **Purpose**: To reduce the size of the data for computational efficiency while retaining critical information.

Block 2: Extracting Mid-Level Features

Convolution Layers

```
c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
c2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c2)
```

- The number of filters increases to 128, indicating a greater depth in the network. This allows the model to capture more complex features.
- The spatial dimensions remain 64x64, while the number of channels increases to 128.
- Each convolution refines the representation of patterns from the previous layer.

Max-Pooling

```
p2 = layers.MaxPooling2D((2, 2))(c2)
```

The spatial dimensions are reduced again, now to 32x32.

Block 3: Extracting Deeper Features

Convolution Layers

```
c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
c3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c3)
```

- The number of filters increases to 256, enabling the model to learn even more intricate patterns and structures.
- The dimensions are still 32x32, but the number of channels is now 256.

Max-Pooling

```
p3 = layers.MaxPooling2D((2, 2))(c3)
```

The output dimensions are reduced to 16x16.

Block 4: Capturing the Most Abstract Features

Convolution Layers

```
c4 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(p3)
c4 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c4)
```

- The number of filters increases to 512. At this stage, the network focuses on highly abstract and complex features.

- These features are often key to understanding the overall structure and details of the input data.

Max-Pooling

```
p4 = layers.MaxPooling2D((2, 2))(c4)
```

The spatial dimensions are further reduced to `8x8`, while the number of channels remains at 512. This output is sent to the **bottleneck** stage.

Bottleneck Section

The **bottleneck** section is a critical part of the U-Net architecture. It acts as the **bridge** between the contracting path (encoder) and the expansive path (decoder). At this stage, the network has the smallest spatial dimensions but the largest number of filters, enabling it to focus on the most **abstract and high-level features** of the input.

First Convolution

```
c5 = layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(p4)
```

- **`Conv2D`**: Applies 1024 filters, each of size `(3, 3)`, to the input from the previous max-pooling layer (`p4`).
- **Filters**: At this depth, filters detect highly abstract features that may represent the overall structure, patterns, or regions within the input.
- **ReLU Activation**: Keeps positive values and zeroes out negatives, ensuring non-linearity and preventing gradient issues.
- **Padding**: With `padding='same'`, the spatial dimensions of the output remain the same (`8x8`).

Second Convolution

```
c5 = layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(c5)
```

- A second convolutional layer is applied to further refine and consolidate the high-level features extracted by the first layer.
- This step ensures that the model captures complex relationships between abstract patterns.

Source Code

Expansive Path

The **expansive path** of the U-Net model is where the network reconstructs the image from the abstract features captured in the bottleneck. This part of the network mirrors the structure of the encoder but in reverse order, allowing the model to upsample the features and recover spatial information that was lost during the downsampling process.

u6

```
u6 = layers.Conv2DTranspose(512, (2, 2), strides=(2, 2),
padding='same')(c5)
u6 = layers.concatenate([u6, c4])
```

- **`Conv2DTranspose`**: Upsamples the feature map by a factor of 2 (doubling the spatial dimensions). The size `(2, 2)` refers to the kernel size used for the transpose convolution operation.
- **Strides**: `(2, 2)` specifies the stride for the upsampling operation, effectively increasing the spatial dimensions.
- **Padding**: `'same'` maintains the size of the output feature map.
- **Concatenate**: Merges the output from the transpose convolution (`u6`) with the corresponding feature map from the contracting path (`c4`). This concatenation adds low-level details back into the feature map to assist with feature recovery.

c6

```
c6 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(u6)
c6 = layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c6)
```

- **Two Convolutional Layers**: Two `Conv2D` layers with `(3, 3)` kernel size to refine the merged feature map (`u6`). The `3x3` kernels allow the model to learn complex patterns and relationships between features.
- **Activation**: Each convolutional layer uses ReLU activation to introduce non-linearity and prevent gradient vanishing/explosion issues.

The **expansive path** of the U-Net model is designed to reconstruct the original image from the abstract features captured in the bottleneck. Here's a concise summary:

1. **Upsampling with `Conv2DTranspose`**: Each layer upsamples the previous feature map by a factor of 2, doubling the spatial dimensions.
2. **Concatenation**: The upsampled features are concatenated with the corresponding features from the contracting path, allowing integration of high-level semantic features.
3. **Convolutional Layers**: Two convolutional layers with ReLU activation refine the concatenated features in each step.
4. **Final Layer**: The final layer outputs a feature map (`c9`) which is used for segmentation, combining detailed low-level features with abstract high-level features from the network.

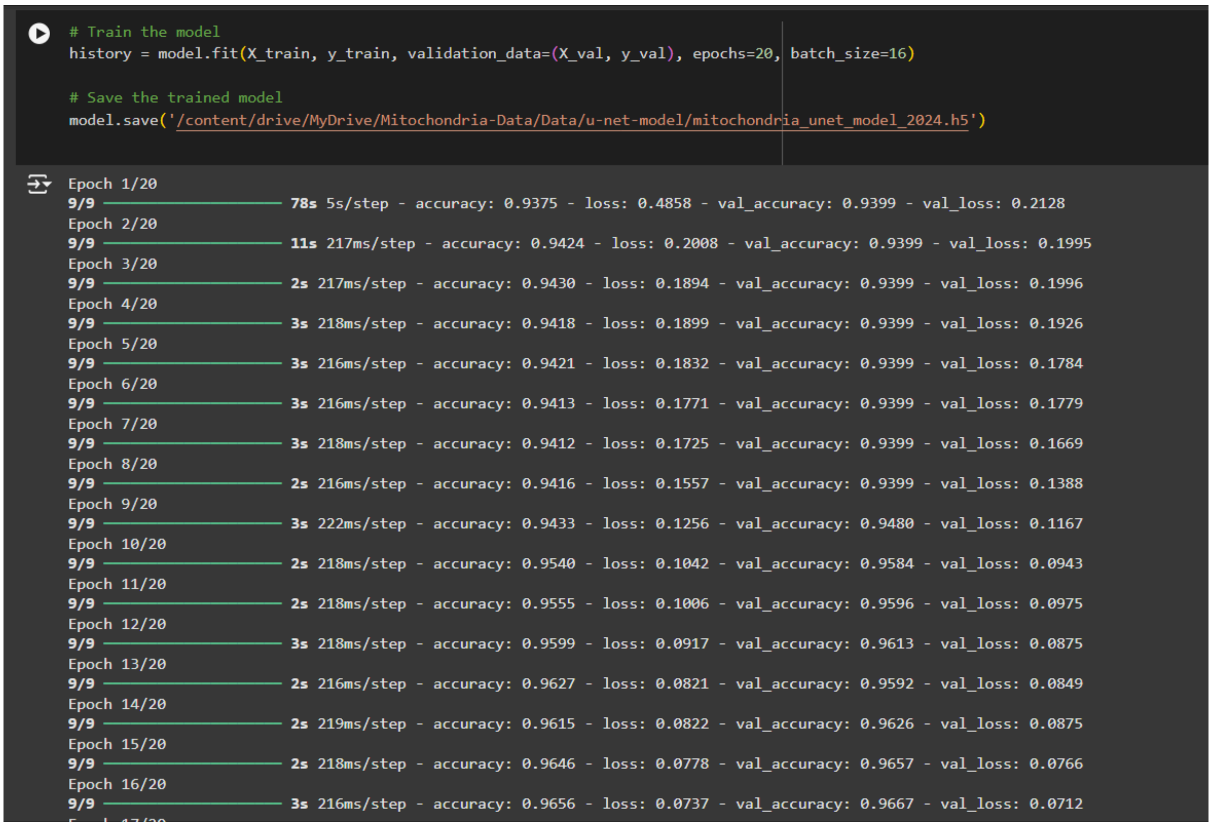
```
model = unet_model()
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

model.summary()
```

- **`unet_model()`**: Creates a U-Net model with specified input size and layers for encoding and decoding hierarchical features.
- **`model.compile()`**: Sets up the model with the Adam optimizer, binary cross-entropy loss function, and accuracy as the evaluation metric.
- **`model.summary()`**: Prints a summary of the model, including layers, their sizes, and total number of parameters.

To start learning the model, we set 20 epochs. It takes about **40 minutes** to train on **Google Colab**, also requiring **14 GB of the GPU** and 8.7 GB of the device's CPU.

Source Code



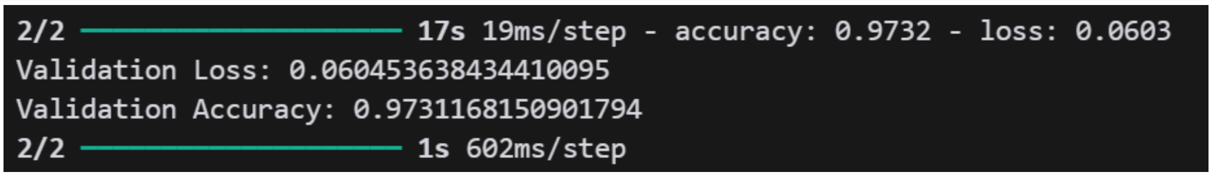
```
# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20, batch_size=16)

# Save the trained model
model.save('/content/drive/MyDrive/Mitochondria-Data/Data/u-net-model/mitochondria_unet_model_2024.h5')
```

```
Epoch 1/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 78s 5s/step - accuracy: 0.9375 - loss: 0.4858 - val_accuracy: 0.9399 - val_loss: 0.2128
Epoch 2/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 11s 217ms/step - accuracy: 0.9424 - loss: 0.2008 - val_accuracy: 0.9399 - val_loss: 0.1995
Epoch 3/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 217ms/step - accuracy: 0.9430 - loss: 0.1894 - val_accuracy: 0.9399 - val_loss: 0.1996
Epoch 4/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 218ms/step - accuracy: 0.9418 - loss: 0.1899 - val_accuracy: 0.9399 - val_loss: 0.1926
Epoch 5/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 216ms/step - accuracy: 0.9421 - loss: 0.1832 - val_accuracy: 0.9399 - val_loss: 0.1784
Epoch 6/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 216ms/step - accuracy: 0.9413 - loss: 0.1771 - val_accuracy: 0.9399 - val_loss: 0.1779
Epoch 7/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 218ms/step - accuracy: 0.9412 - loss: 0.1725 - val_accuracy: 0.9399 - val_loss: 0.1669
Epoch 8/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 216ms/step - accuracy: 0.9416 - loss: 0.1557 - val_accuracy: 0.9399 - val_loss: 0.1388
Epoch 9/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 222ms/step - accuracy: 0.9433 - loss: 0.1256 - val_accuracy: 0.9480 - val_loss: 0.1167
Epoch 10/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 218ms/step - accuracy: 0.9540 - loss: 0.1042 - val_accuracy: 0.9584 - val_loss: 0.0943
Epoch 11/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 218ms/step - accuracy: 0.9555 - loss: 0.1006 - val_accuracy: 0.9596 - val_loss: 0.0975
Epoch 12/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 218ms/step - accuracy: 0.9599 - loss: 0.0917 - val_accuracy: 0.9613 - val_loss: 0.0875
Epoch 13/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 216ms/step - accuracy: 0.9627 - loss: 0.0821 - val_accuracy: 0.9592 - val_loss: 0.0849
Epoch 14/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 219ms/step - accuracy: 0.9615 - loss: 0.0822 - val_accuracy: 0.9626 - val_loss: 0.0875
Epoch 15/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 2s 218ms/step - accuracy: 0.9646 - loss: 0.0778 - val_accuracy: 0.9657 - val_loss: 0.0766
Epoch 16/20
9/9 ━━━━━━━━━━━━━━━━━━━━ 3s 216ms/step - accuracy: 0.9656 - loss: 0.0737 - val_accuracy: 0.9667 - val_loss: 0.0712
```

After evaluating the U-Net model on the validation set, we print these data to see what accuracy model can achieve.

```
loss, accuracy = model.evaluate(X_val, y_val)
print(f"Validation Loss: {loss}")
print(f"Validation Accuracy: {accuracy}")
```

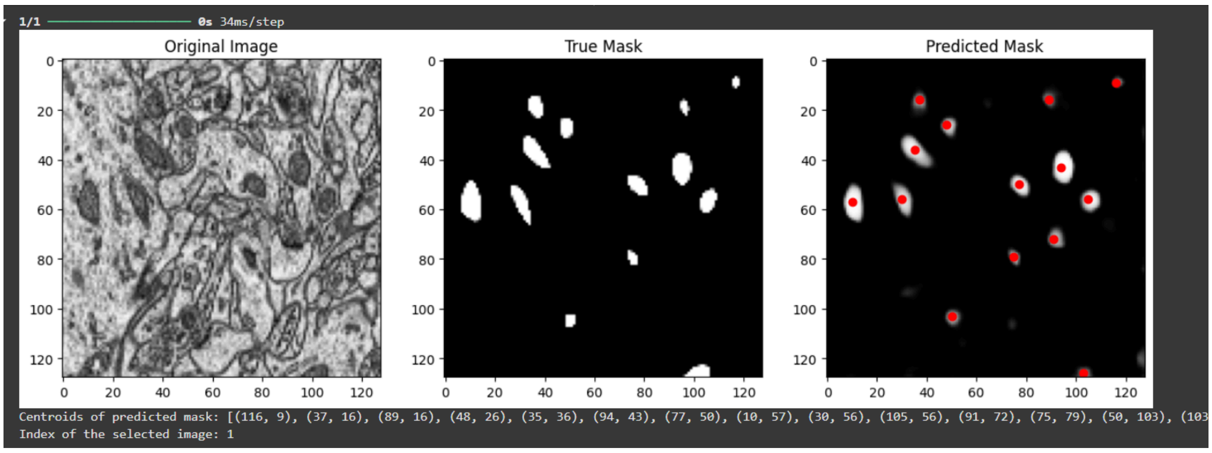


```
2/2 ━━━━━━━━━━━━━━━━━━━━ 17s 19ms/step - accuracy: 0.9732 - loss: 0.0603
Validation Loss: 0.060453638434410095
Validation Accuracy: 0.9731168150901794
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 602ms/step
```

Also we need to get the center of our masks, so we use the `find_centroids` Class, this class identifies the centroid positions of objects in a binary mask. It filters the mask using a threshold (0.5) and labels the connected regions. Then, it calculates the mean position of points in each labeled region to determine the centroid and returns these positions as a list.

Result of the U-Net model be like this:

Source Code



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
```

```
Centroids of predicted mask: [(116, 9), (37, 16), (89, 16), (48, 26), (35, 36), (94, 43), (77, 50), (10, 57), (30, 56), (105, 56), (91, 72), (75, 79), (50, 103), (103
Index of the selected image: 1
```

Resources

- Electron Microscopy Dataset – CVLAB - EPFL
- Segment Anything
- GitHub - facebookresearch/segment-anything
- Automated segmentation and tracking of mitochondria in live-cell time-lapse images - PubMed
- U-Net: Convolutional Networks for Biomedical Image Segmentation
- Meta AI's Segment Anything Model (SAM) Explained: The Ultimate Guide
- 204 - U-Net for semantic segmentation of mitochondria
- How to Use the Segment Anything Model (SAM)
- U-Net , Semantic Segmentation
- Learn How to Train U-Net On Your Dataset | by Sukriti Paul | Coinmonks | Medium
- U-Net , Semantic Segmentation