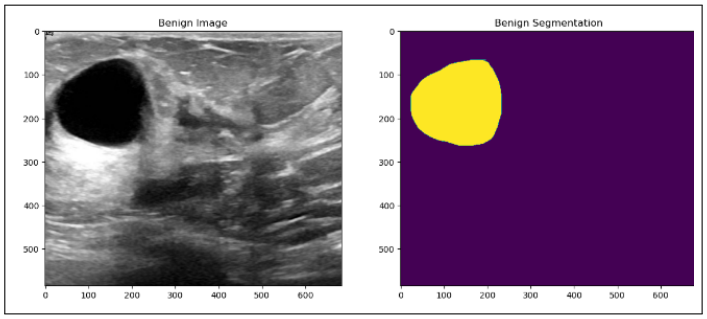
Breast Cancer Detection Using U-Net

Ayoub Bilel

October 2024



Abstract

This project was undertaken independently to deepen my knowledge of deep learning, specifically in the field of object detection. Object detection has numerous applications across various domains, including medicine, autonomous vehicles, manufacturing, and damage detection. Through this project, I developed a breast detection model using U-Net, aiming to contribute to advancements in medical imaging and enhance early detection capabilities in breast cancer diagnosis.

Contents

1 Introduction

Breast cancer remains one of the most common cancers affecting women globally, making early detection crucial for effective treatment and improved patient outcomes. The World Health Organization emphasizes that early diagnosis greatly increases the chances of successful treatment, highlighting the need for innovative approaches in medical imaging and diagnostics. Traditional methods like mammography can face challenges such as false positives and subjective interpretation, driving a growing interest in leveraging advanced technologies—especially deep learning—to enhance diagnostic accuracy and efficiency.

Deep learning, a subset of artificial intelligence, has already shown remarkable success in fields like computer vision and medical imaging. Among the many deep learning architectures, the U-Net model stands out as a powerful tool for image segmentation, especially in biomedical applications. Its encoder-decoder structure allows it to capture both the big picture and detailed features, making it well-suited for identifying and outlining tumors in breast images.

This project began as a personal initiative to deepen my understanding of deep learning and its applications in object detection. The main goal was to develop a breast tumor detection model using the U-Net architecture, with the hope of advancing automated diagnostic tools in healthcare. By training on a dataset of annotated breast images, the model aims to improve early detection capabilities and serve as a reliable tool for medical professionals.

In this report, I'll go over the methods used to build the U-Net model, including details on the dataset, preprocessing steps, and model performance. I'll also discuss the results and explore ways to improve the model, such as using larger datasets and more powerful computational resources.

2 Data Description

2.1 Dataset Overview

```
def pair_images_and_masks(path, category):
    images = glob.glob(os.path.join(path, f"{category}/*.png"))

    image_files = [f for f in images if "_mask" not in f]
    mask_files = [f.replace(".png", "_mask.png") for f in image_files]


    paired_data = [(img, msk) for img, msk in zip(image_files, mask_files) if os.path.exists(msk)]

    print(f"Found {len(paired_data)} pairs in {category}")
    return paired_data


normal_data = pair_images_and_masks('C:/Users/bilel/Downloads/archive/Dataset', 'normal')
malignant_data = pair_images_and_masks('C:/Users/bilel/Downloads/archive/Dataset', 'malignant')
benign_data = pair_images_and_masks('C:/Users/bilel/Downloads/archive/Dataset', 'benign')


all_data = normal_data + malignant_data + benign_data
```

Output: Found 133 pairs in normal
Found 210 pairs in malignant
Found 437 pairs in benign
Our dataset has 3 classes: healthy, represented by normal, and two types of cancer, malignant and benign.

```
def load_image_and_mask(image_path, mask_path):
    image = Image.open(image_path).convert("RGB")
```

```
    mask = Image.open(mask_path).convert("L")  # Convert mask to grayscale
    return np.array(image), np.array(mask)
image_path_normal, mask_path_normal = normal_data[1]
image_path_benign, mask_path_benign = benign_data[1]
image_path_malignant, mask_path_malignant = malignant_data[1]
image_normal, mask_normal=load_image_and_mask(image_path_normal, mask_path_normal)
image_benign, mask_benign=load_image_and_mask(image_path_benign, mask_path_benign)
image_malignant, mask_malignant = load_image_and_mask(image_path_malignant, mask_path_malignant)

print(f"Noraml image shape: {image_normal.shape}, Normal mask shape: {mask_normal.shape}")
print(f"Benign image shape: {image_benign.shape}, Benign mask shape: {mask_benign.shape}")
print(f"Malignant image shape: {image_malignant.shape}, Malignant mask shape: {mask_malignant.shape}")
```

Output: Noraml image shape: (485, 501, 3), Normal mask shape: (485, 501) Benign image shape: (585, 683, 3), Benign mask shape: (585, 683) Malignant image shape: (393, 462, 3), Malignant mask shape: (393, 462)

```
def find_smallest_image_dimensions(paired_data):
    smallest_dimensions = None

    for img, _ in paired_data:
        with Image.open(img) as image:
            width, height = image.size
            if smallest_dimensions is None or (width * height) < (smallest_dimensions[0] * smallest_dimensions[1]):
                smallest_dimensions = (width, height)

    return smallest_dimensions
smallest_dimensions = find_smallest_image_dimensions(all_data)
print(f"The smallest image dimensions are: {smallest_dimensions}")
```

Output: The smallest image dimensions are: (190, 335)
In this section, I analyzed the shapes of the smallest image and several randomly selected images from the dataset to determine an optimal image size that would enhance learning performance without compromising data quality."

```
fig, arr = plt.subplots(1, 2, figsize=(14, 10))
arr[0].imshow(image_normal)
arr[0].set_title('Normal Image')
arr[1].imshow(mask_normal)
arr[1].set_title('Normal Segmentation')
fig, arr = plt.subplots(1, 2, figsize=(14, 10))
arr[0].imshow(image_benign)
arr[0].set_title('Benign Image')
arr[1].imshow(mask_benign)
arr[1].set_title('Benign Segmentation')
fig, arr = plt.subplots(1, 2, figsize=(14, 10))
arr[0].imshow(image_malignant)
arr[0].set_title('Malignant Image')
arr[1].imshow(mask_malignant)
arr[1].set_title('Malignant Segmentation')
```

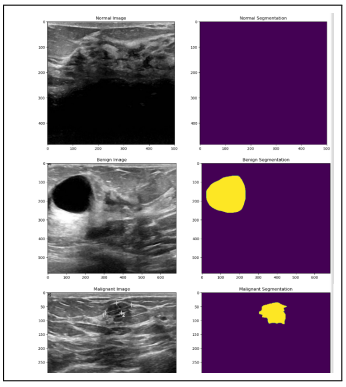


Figure 1: Types of cancer and their masks

These are samples from all the data types available in the dataset.

2.2 Data Preprocessing

```
    def preprocess_image(image_path, mask_path,
        target_size=(256, 256)):
    try:
        image = tf.io.read_file(image_path)
        mask = tf.io.read_file(mask_path)

        image = tf.io.decode_image(image, channels=3,
            expand_animations=False)
        mask = tf.io.decode_image(mask, channels=1,
            expand_animations=False)
        if image.shape is None or image.shape[0] == 0 or
            image.shape[1] == 0:
            print(f"Error: Image has undefined or zero
                dimensions: {image.shape}")
            return None, None

        if mask.shape is None or mask.shape[0] == 0 or
            mask.shape[1] == 0:
            print(f"Error: Mask has undefined or zero
                dimensions: {mask.shape}")
            return None, None
        if image.shape[-1] != 3:
            print(f"Error: Image does not have 3
                channels (found {image.shape[-1]}).")
            return None, None
        image = tf.image.convert_image_dtype(image, tf.
            float32)
        img_height, img_width = tf.shape(image)[0], tf.
            shape(image)[1]
        aspect_ratio = img_width / img_height
        target_width, target_height = target_size
        if aspect_ratio > 1:
            new_height = target_height
            new_width = tf.cast(new_height *
                aspect_ratio, tf.int32)
        else:
            new_width = target_width
```

```
            new_height = tf.cast(new_width /
                aspect_ratio, tf.int32)
        image = tf.image.resize(image, [new_height,
            new_width], method='nearest')
        mask = tf.image.resize(mask, [new_height,
            new_width], method='nearest')
        image = tf.image.resize_with_crop_or_pad(image,
            target_height, target_width)
        mask = tf.image.resize_with_crop_or_pad(mask,
            target_height, target_width)
        mask = tf.cast(tf.math.reduce_max(mask, axis=-1,
            keepdims=True) > 0, tf.float32)  # Ensure
            binary mask
        return image, mask

    except Exception as e:
        print(f"Error during preprocessing: {str(e)}")
        return None, None

# Convert the image and mask paths to a TensorFlow
    dataset
image_paths, mask_paths = zip(*all_data)  # Unzip into
    two separate lists of image and mask paths
dataset = tf.data.Dataset.from_tensor_slices((list(
    image_paths), list(mask_paths)))
image_ds = dataset.map(lambda img_path, mask_path:
    preprocess_image(img_path, mask_path))
# Filter out None values returned by preprocess_image
image_ds = image_ds.filter(lambda img, mask: img is not
    None and mask is not None)

print(image_ds)
```

The preprocess-image function was created to load images and their masks, while checking for errors concerning color channels, dimensions and image format .After the checks, the function calculates the aspect ratio of the image, resizing it proportionally by adjusting either its width or height as needed. The function then resizes both the image and mask to maintain the calculated aspect ratio, followed by a final resize with cropping or padding to fit a target size,which is 256x256 pixels. Additionally, the function converts the mask

into a binary format, ensuring only two distinct values are present, which is essential for segmentation tasks. Finally, if all processing steps are successful, the function returns the prepared image and mask as TensorFlow tensors.

3 Methodology

3.1 Model Architecture

3.1.1 Overview of U-Net

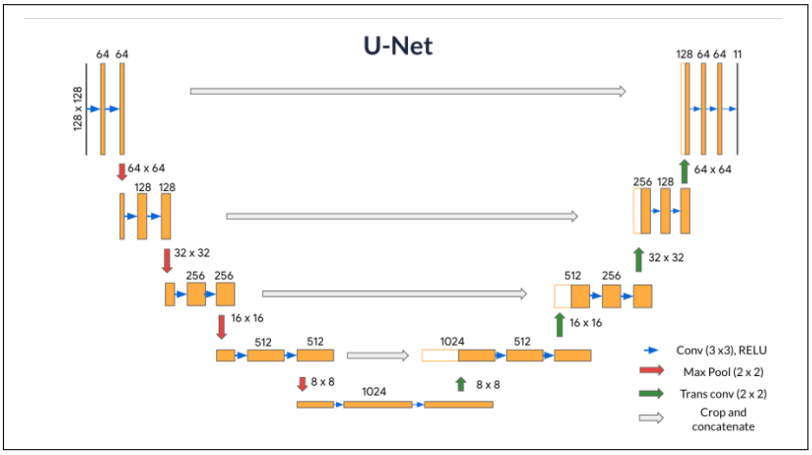


Figure 2: Example of a U-Net Architecture

This above is an example of a U-Net model that I will be using something similar to it. The code below shows the implementation of the model.

3.1.2 Model Implementation

Conv-block:

```
def conv_block(inputs=None, n_filters=32, dropout_prob
    =0, max_pooling=True):
      conv = Conv2D(n_filters,
                    3,
                    activation='relu',
                    padding='same',
```

```
                    kernel_initializer='he_normal')(inputs
                        )
    conv = Conv2D(n_filters,
                    3,
                    activation='relu',
                    padding='same',

                    kernel_initializer='he_normal')(conv)
    if dropout_prob > 0:
        conv = Dropout(dropout_prob)(conv)
    if max_pooling:
        next_layer = MaxPooling2D(pool_size=(2, 2))(conv
            )
    else:
        next_layer = conv
    skip_connection = conv
    return next_layer, skip_connection
```

Upsampling-block:

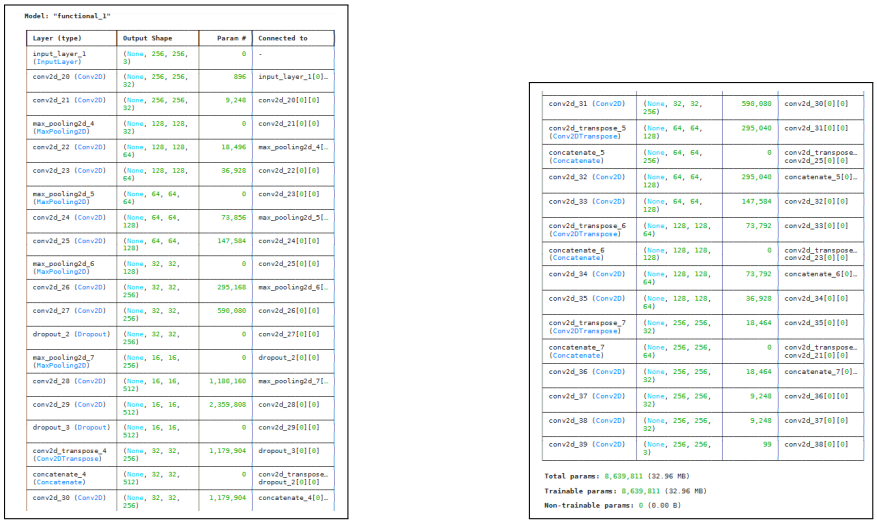
```
def upsampling_block(expansive_input, contractive_input,
    n_filters=32):
    up = Conv2DTranspose(
                    n_filters,
                    (3,3),
                    strides=(2,2),
                    padding='same')(expansive_input)
    merge = concatenate([up, contractive_input], axis=3)
    conv = Conv2D(n_filters,
                    3,
                    activation='relu',
                    padding='same',
                    kernel_initializer='he_normal')(merge)
    conv = Conv2D(n_filters,
                    3,
                    activation='relu',
                    padding='same',
                    kernel_initializer='he_normal')(conv)
    return conv
```

Unet-model:

```
1
2  def unet_model(input_size=(256, 256, 3), n_filters=32,
       n_classes=3):
3      inputs = Input(input_size)
4      cblock1 = conv_block(inputs, n_filters)
5      cblock2 = conv_block(cblock1[0], n_filters * 2)
6      cblock3 = conv_block(cblock2[0], n_filters * 4)
7      cblock4 = conv_block(cblock3[0], n_filters * 8,
           dropout_prob=0.3)
8      cblock5 = conv_block(cblock4[0], n_filters * 16,
           dropout_prob=0.3, max_pooling=False)
9      ublock6 = upsampling_block(cblock5[0], cblock4[1],
           n_filters * 8)
10     ublock7 = upsampling_block(ublock6, cblock3[1],
           n_filters * 4)
11     ublock8 = upsampling_block(ublock7, cblock2[1],
           n_filters * 2)
12     ublock9 = upsampling_block(ublock8, cblock1[1],
           n_filters)
13     conv9 = Conv2D(n_filters,
14                    3,
15                    activation='relu',
16                    padding='same',
17                    kernel_initializer='he_normal')(
                          ublock9)
18     conv10 = Conv2D(n_classes, 1, padding='same')(conv9)
19     model = tf.keras.Model(inputs=inputs, outputs=conv10
           )
20     return model
```

3.1.3 Model summary:

```
1  img_height = 256
2  img_width = 256
3  num_channels = 3
4  unet = unet_model((img_height, img_width, num_channels))
5  unet.summary()
```



Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 256, 256, 3)	0	-
conv2d_20 (Conv2D)	(None, 256, 256, 32)	896	input_layer_1[0]…
conv2d_21 (Conv2D)	(None, 256, 256, 32)	9,248	conv2d_20[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 128, 128, 32)	0	conv2d_21[0][0]
conv2d_22 (Conv2D)	(None, 128, 128, 64)	18,496	max_pooling2d_4[…
conv2d_23 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_22[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 64, 64, 64)	0	conv2d_23[0][0]
conv2d_24 (Conv2D)	(None, 64, 64, 128)	73,856	max_pooling2d_5[…
conv2d_25 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_24[0][0]
max_pooling2d_6 (MaxPooling2D)	(None, 32, 32, 128)	0	conv2d_25[0][0]
conv2d_26 (Conv2D)	(None, 32, 32, 256)	295,168	max_pooling2d_6[…
conv2d_27 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_26[0][0]
dropout_2 (Dropout)	(None, 32, 32, 256)	0	conv2d_27[0][0]
max_pooling2d_7 (MaxPooling2D)	(None, 16, 16, 256)	0	dropout_2[0][0]
conv2d_28 (Conv2D)	(None, 16, 16, 512)	1,180,160	max_pooling2d_7[…
conv2d_29 (Conv2D)	(None, 16, 16, 512)	2,359,808	conv2d_28[0][0]
dropout_3 (Dropout)	(None, 16, 16, 512)	0	conv2d_29[0][0]
conv2d_transpose_4 (Conv2DTranspose)	(None, 32, 32, 256)	1,179,904	dropout_3[0][0]
concatenate_4 (Concatenate)	(None, 32, 32, 512)	0	conv2d_transpose… dropout_2[0][0]
conv2d_30 (Conv2D)	(None, 32, 32, 256)	1,179,904	concatenate_4[0]…
conv2d_31 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_30[0][0]
conv2d_transpose_5 (Conv2DTranspose)	(None, 64, 64, 128)	295,040	conv2d_31[0][0]
concatenate_5 (Concatenate)	(None, 64, 64, 256)	0	conv2d_transpose… conv2d_25[0][0]
conv2d_32 (Conv2D)	(None, 64, 64, 128)	295,040	concatenate_5[0]…
conv2d_33 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_32[0][0]
conv2d_transpose_6 (Conv2DTranspose)	(None, 128, 128, 64)	73,792	conv2d_33[0][0]
concatenate_6 (Concatenate)	(None, 128, 128, 128)	0	conv2d_transpose… conv2d_23[0][0]
conv2d_34 (Conv2D)	(None, 128, 128, 64)	73,792	concatenate_6[0]…
conv2d_35 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_34[0][0]
conv2d_transpose_7 (Conv2DTranspose)	(None, 256, 256, 32)	18,464	conv2d_35[0][0]
concatenate_7 (Concatenate)	(None, 256, 256, 64)	0	conv2d_transpose… conv2d_21[0][0]
conv2d_36 (Conv2D)	(None, 256, 256, 32)	18,464	concatenate_7[0]…
conv2d_37 (Conv2D)	(None, 256, 256, 32)	9,248	conv2d_36[0][0]
conv2d_38 (Conv2D)	(None, 256, 256, 32)	9,248	conv2d_37[0][0]
conv2d_39 (Conv2D)	(None, 256, 256, 3)	99	conv2d_38[0][0]

Total params: 8,639,811 (32.96 MB)

Trainable params: 8,639,811 (32.96 MB)

Non-trainable params: 0 (0.00 B)

Figure 3: model Summary

We got 8,639,811 parameters to train, the number of parameters can be way larger than we got if we want to train a much deeper model or with a different image size.

3.2 Training Process

3.2.1 Loss Function

```
unet.compile(optimizer='adam',
                loss=tf.keras.losses.
                    SparseCategoricalCrossentropy(
                    from_logits=True),
                metrics=['accuracy'])
```

Used ADAM (Adaptive Moment Estimation) optimizer because it is efficient and can adjust learning rates for each parameter which is useful in U-Nets as different parts of the network require different learning rates due to varying gradients.

3.2.2 Training

```
EPOCHS = 30
VAL_SUBSPLITS = 5
BUFFER_SIZE = 780
BATCH_SIZE = 16
train_dataset = image_ds.cache().shuffle(BUFFER_SIZE).
    batch(BATCH_SIZE)
print(image_ds.element_spec)
model_history = unet.fit(train_dataset, epochs=EPOCHS)
```

I set the `BUFFER_SIZE` to 780, equal to the total number of image-mask pairs in the dataset, as each pair counts as a single training instance. For the `VAL_SUBSPLITS`, I chose 5 to achieve an 80/20 train/validation split. I also set the `BATCH_SIZE` to 16 to optimize training while maintaining compatibility with my PC's GPU performance.

```
Epoch 20/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 226s 5s/step - accuracy: 0.9339 - loss: 0.1837
Epoch 21/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 237s 5s/step - accuracy: 0.9337 - loss: 0.1886
Epoch 22/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 232s 5s/step - accuracy: 0.9388 - loss: 0.1729
Epoch 23/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 232s 5s/step - accuracy: 0.9304 - loss: 0.1878
Epoch 24/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 234s 5s/step - accuracy: 0.9357 - loss: 0.1759
Epoch 25/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 230s 5s/step - accuracy: 0.9398 - loss: 0.1665
Epoch 26/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 227s 5s/step - accuracy: 0.9385 - loss: 0.1702
Epoch 27/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 225s 5s/step - accuracy: 0.9398 - loss: 0.1642
Epoch 28/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 227s 5s/step - accuracy: 0.9425 - loss: 0.1621
Epoch 29/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 223s 5s/step - accuracy: 0.9482 - loss: 0.1504
Epoch 30/30
49/49 ━━━━━━━━━━━━━━━━━━━━ 227s 5s/step - accuracy: 0.9454 - loss: 0.1501
```

Figure 4: Accuracy and loss developemnt within the last 10 Epochs

4 Results

4.1 Performance Metric

```
plt.plot(model_history.history["accuracy"])
```

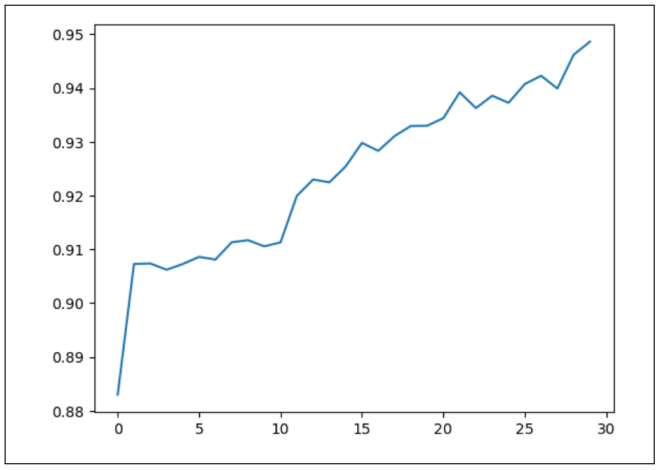


Figure 5: Performance metric

After 30 epochs, we attained an accuracy of **94.54%** and a loss of **0.1501**, which is quite good considering the small dataset and the limited number of epochs used.

4.2 Visualizations

```
def create_mask(pred_mask):
    pred_mask = tf.nn.softmax(pred_mask, axis=-1)
    pred_mask = tf.argmax(pred_mask, axis=-1)
    pred_mask = pred_mask[..., tf.newaxis]
    return pred_mask[0]
```

```
def show_predictions(dataset=None, num=1):
    if dataset:
        for image, mask in dataset.take(num):
```

```
            pred_mask = unet.predict(image)
            display([image[0], mask[0], create_mask(
                pred_mask)])
    else:
        display([sample_image, sample_mask,
             create_mask(unet.predict(sample_image[tf.
                 newaxis, ...]))])
show_predictions(train_dataset, 6)
```

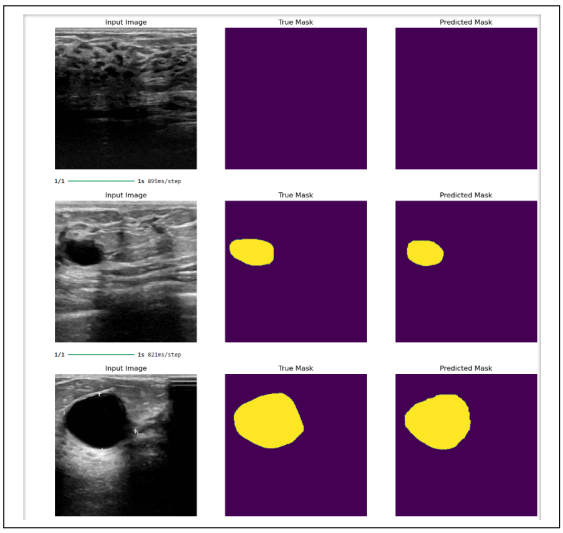


Figure 6: Results

5 Discussion

5.1 Interpretation of Results

The U-Net model did an impressive job at segmenting breast ultrasound images, hitting a solid accuracy of 94.54% and keeping the loss low at 0.1501. These numbers reflect that the model isn't just memorizing patterns—it's genuinely learning to distinguish between different tissue types, which is crucial for accurate

detection.

Looking at the images, the predicted masks line up closely with the true masks. The model captures the shape and location of the regions it's supposed to detect, which is especially challenging given the irregular shapes in medical images like these. This alignment between the predictions and the actual areas is promising because it suggests the model could really help radiologists pinpoint suspicious areas more accurately.

There are some small discrepancies in a few masks where the contours don't perfectly match up, which shows there's still room for improvement. More data or some fine-tuning might help refine these edges. But overall, the model's performance suggests that U-Net architectures like this one could play a valuable role in supporting early breast cancer detection, potentially giving doctors another reliable tool in the diagnostic process.

5.2 Limitations

While the U-Net model performed well, there are a few limitations to consider. First, the model's accuracy could likely improve with access to a larger and more diverse dataset. Right now, it's trained on a limited number of images, which might not capture the full range of variations in breast tissue, especially across different demographics and imaging conditions. A more comprehensive dataset could help the model generalize better and improve its accuracy in real-world applications.

Another limitation is the computational power used. The model training was constrained by the available GPU, which may have limited both the training speed and the depth of tuning we could apply. With a more powerful GPU, we could train the model faster and experiment with more complex architec-

tures or deeper layers, which might yield better segmentation results.

Lastly, like many models, this U-Net could potentially struggle with edge cases, such as very small or unusually shaped anomalies. These kinds of cases would likely require further data augmentation techniques or even more advanced network architectures to capture accurately.

6 Conclusion

In summary, the U-Net model achieved strong performance in segmenting breast ultrasound images, with an accuracy of 94.54% and a low loss of 0.1501. These results indicate the model's potential to assist in the early detection of breast cancer by accurately identifying regions of interest in ultrasound images. The close alignment between the predicted and true masks shows promise for the model's practical application in medical diagnostics, potentially serving as a valuable tool for radiologists.

Looking ahead, there's room for further refinement. Expanding the dataset to include a wider range of cases would likely help the model generalize more effectively. Additionally, training on a more powerful GPU could enable deeper tuning and faster experimentation, possibly unlocking even better performance. Exploring advanced architectures or hybrid models might also boost the model's ability to handle more complex or subtle patterns in breast tissue. With these improvements, this model could become an even more reliable asset in aiding early breast cancer detection and supporting clinical decision-making.

7 References

References

[1] World Health Organization. *Breast Cancer Factsheet*. Available at: https://www.who.int/news-room/fact-sheets/detail/breast-cancer. Accessed [Date Accessed].

[2] Ronneberger O, Fischer P, Brox T. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. arXiv preprint arXiv:1505.04597, 2015. Available at: https://arxiv.org/abs/1505.04597.

[3] Al-Dhabyani W, Gomaa M, Khaled H, Fahmy A. *Dataset of Breast Ultrasound Images*. Data in Brief. 2020 Feb;28:104863. DOI: 10.1016/j.dib.2019.104863. Available at: https://www.kaggle.com/datasets/aryashah2k/breast-ultrasound-images-dataset.