Neural Networks & Fuzzy Logic: Research Paper Assessment

Saumil Agarwal 2018A7PS0268P
Atharva Anand Joshi 2018A3PS0515P
Aayush Singhal 2018A1PS0047P

U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for
Biological Signalling Studies, University of Freiburg, Germany

AIM

There is a large consent that successful training of deep networks requires many thousand annotated training samples.

The aim of this paper is to present a network and training strategy that relies on the **strong use of data augmentation** to use the available annotated samples more efficiently and segment biomedical images accurately.

Methodology

The U-Net architecture is built upon the Fully Convolutional Network and modified in a way that it yields better segmentation in medical imaging. The paper uses **excessive data augmentation** by applying **elastic deformations** to the available training images. This allows the network to learn invariance to such deformations, without the need to see these transformations in the annotated image corpus. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization.

Final Outcome

The U-Net architecture achieves very good performance on very different biomedical segmentation applications. This can be accounted for by data augmentation with elastic deformations, due to which it only needs very few annotated images and has a very reasonable training time. This U-Net architecture can be applied easily to many more tasks.

Background

- In the last few years, deep convolutional networks have performed quite well in many visual recognition tasks. However, their success was limited due to the size of the available training sets and the size of the considered networks.
- The typical use of convolutional is on classification tasks, where the output to an image is a single class label. However, in many visual tasks, especially in biomedical image processing, the desired output should include **localization** i.e. a class label is supposed to be assigned to each pixel. Moreover, thousands of training images are usually beyond reach in biomedical tasks.

- Hence, Ciresan et al. trained a network in a sliding-window setup to predict the class label of each pixel by providing a local region around that pixel as input. The reasons for this are:
  - This network can localize.
  - The training data in terms of patches is much larger than the number of training images.

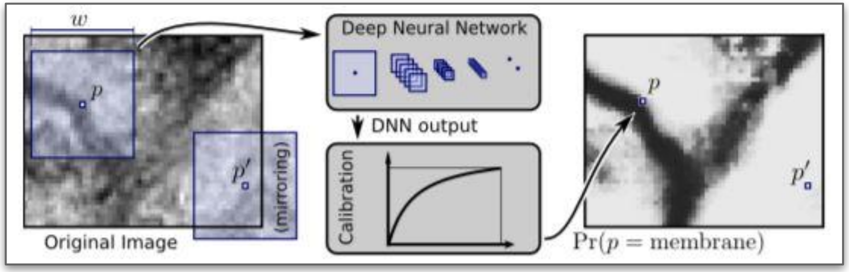


Fig. An overview of their approach

- However, this strategy has two drawbacks:
  - It is quite slow because the network must be run separately for each patch, and there is a lot of redundancy due to overlapping patches.
  - There is a trade-off between localization accuracy and the use of context. Larger patches require more max-pooling layers that reduce the localization accuracy, while smaller patches allow the network to see only little context.

- Hence, more recent approaches proposed a classifier output that takes into account the features from multiple layers. Good localization and the use of context are possible at the same time. The main idea is to supplement a usual contracting network by successive layers, thus increasing the resolution of the output.
- This paper builds upon a more elegant architecture i.e. the **fully convolutional network**.This architecture is modified and extended such that it works with very few training images and yields more precise segmentation.

Dataset

ISBI Challenge: Segmentation of neuronal structures in EMstacks

- Our training dataset is provided by the EM segmentation challenge that was started at ISBI 2012
- It is a set of 30 sections from a serial section Transmission Electron Microscopy (ssTEM) data set of the Drosophila first instar larva ventral nerve cord (VNC).
- The corresponding binary labels are provided in an in-out fashion, i.e. white for the pixels of segmented objects and black for the membranes.

- The test set is publicly available, but its segmentation maps are kept secret. An evaluation can be obtained by sending the predicted membrane probability map to the organizers.

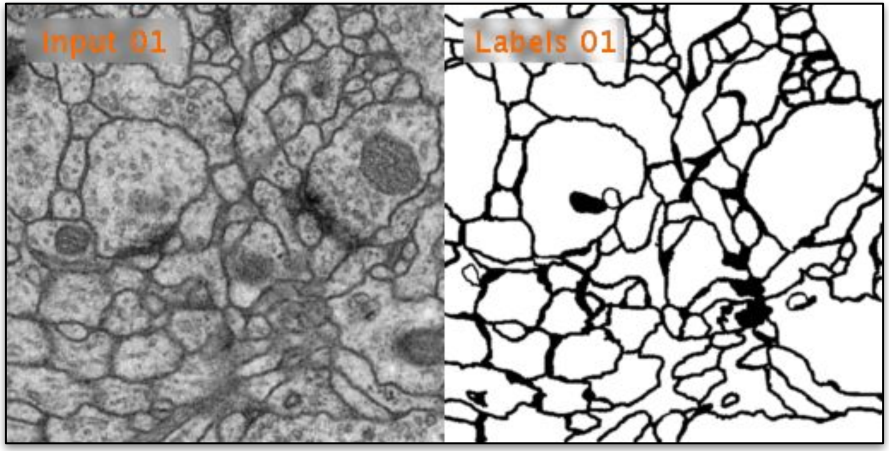


Fig. Dataset images and its corresponding labels

Allotted Tasks

1. **Data augmentation**: Since only 30 training samples are available, it is essential to apply data augmentation. This is the most important step of the problem statement.

2. **Model**: Develop a Keras implementation of the Unet architecture.

3. **Training**: Training the model on the augmented dataset.

4. **Visualization:** Obtain graphs of the training and validation results, that is loss and accuracy. This would help us understand how well the training procedure has taken place.

Progress Summary

	Task	Completion	Progress
1	Data augmentation	✓	Implemented Zoom, Shear, Horizontal Flip, Overlap Tile Strategy, Elastic transform
2	Developing the model	✓	Developed model from scratch in Keras
3	Training the model	✓	Successfully trained the model on colab
4	Obtaining Visualizations	✓	Used matplotlib graphs to plot the training and validation performance.

IMPLEMENTATION

U-Net Architecture

- The u-net comprises of two parts: an encoder/contraction path and a decoder/expansion path.
- **Contraction path** consists of a repeated application of 3x3 convolutions (unpadded) each followed by a ReLU and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled.
- The **Expansion path** consists of upsampling of the feature map followed by a 2x2 convolution that halves the number of feature channels, a concatenation with the cropped feature map from the contracting path, and 3x3 convolutions, followed by a ReLU.The upsampling is done to meet the same size as the block to be concatenated on the left.

- The model is implemented fully using Keras functional API.
- Learning rate: 10^-4
- All the convolutions are valid, that is, without padding.
- We use relu activation function at each layer other than the output. At the output, we use sigmoid activation function, in order to get values between 0 and 1 for predicted segmentation masks.
- The model does not have any fully-connected layers.

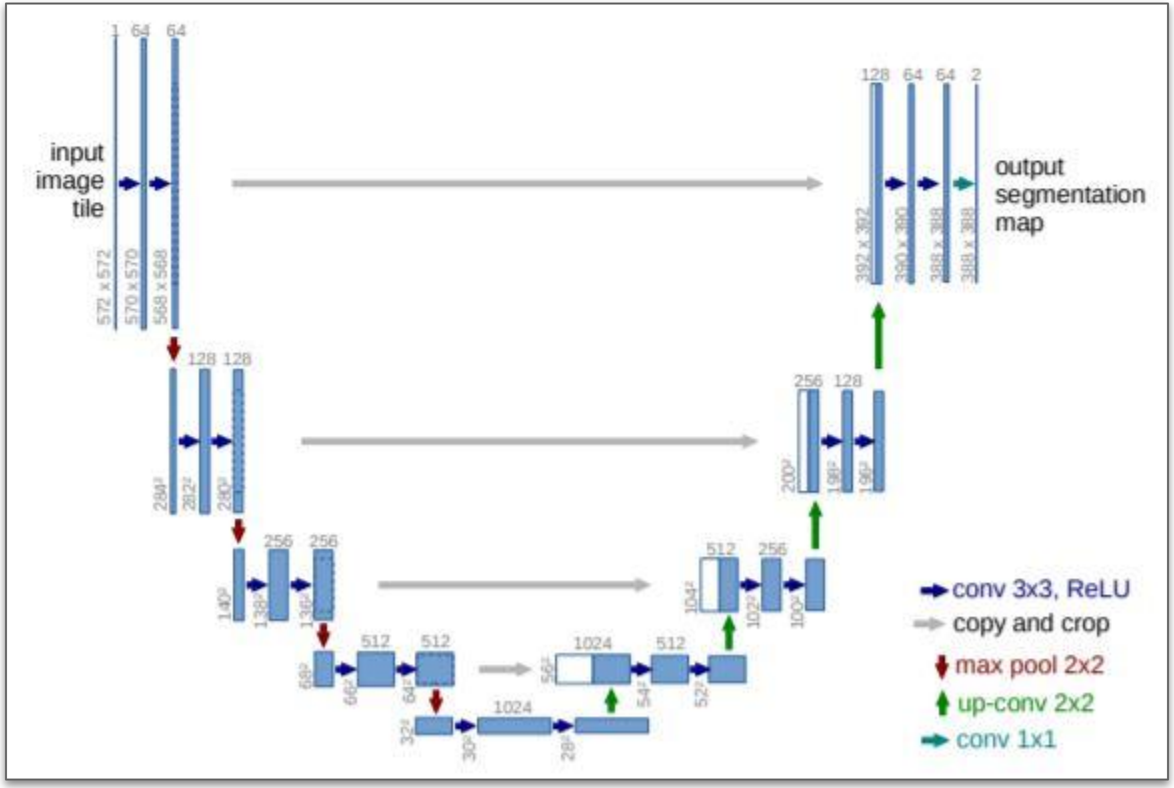


Fig. U-net architecture. Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations

```python
def unet(pretrained_weights = None,input_size = (572,572,1), lr=1e-4):
    inputs = Input(input_size)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2),strides=2)(conv1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(pool1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2),strides=2)(conv2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2),strides=2)(conv3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2),strides=2)(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)
```

```
up6 = Conv2DTranspose(512, (2, 2), strides=(2, 2), activation = 'relu', padding="valid",kernel_initializer = 'he_normal')(drop5)
    merge6 = concatenate([drop4[:, 4:60,4:60,:],up6], axis = 3)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(merge6)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv6)

    up7 = Conv2DTranspose(256, (2, 2), strides=(2, 2), activation = 'relu', padding="valid",kernel_initializer = 'he_normal')(conv6)
    merge7 = concatenate([conv3[:, 16:120,16:120,:],up7], axis = 3)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(merge7)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv7)

    up8 = Conv2DTranspose(128, (2, 2), strides=(2, 2), activation = 'relu', padding="valid",kernel_initializer = 'he_normal')(conv7)
    merge8 = concatenate([conv2[:, 40:240,40:240,:],up8], axis = 3)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(merge8)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv8)

    up9 = Conv2DTranspose(64, (2, 2), strides=(2, 2), activation = 'relu', padding="valid",kernel_initializer = 'he_normal')(conv8)
    merge9 = concatenate([conv1[:, 88:480,88:480,:],up9], axis = 3)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(merge9)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'valid', kernel_initializer = 'he_normal')(conv9)
    conv9 = Conv2D(1, 1, padding = 'valid', activation = 'sigmoid', kernel_initializer = 'he_normal')(conv9)
    # conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

    model = Model(inputs = inputs, outputs = conv9)

    model.compile(optimizer = Adam(lr = lr), loss = 'binary_crossentropy', metrics = ['accuracy'])

    if(pretrained_weights):
      model.load_weights(pretrained_weights)

    return model
```

Data Augmentation and Pre-Processing

Our pipeline broadly consists of the following steps:

1. **Normal Data Augmentation:** This includes Zoom, Shear, Horizontal Flip and Rotation. For this purpose we have made use of the Augmentor library.
2. **Mirror Padding and Overlap-Tile strategy:** The size of the original image is 512*512. To avoid losing context at the edges, we have added a mirror pad of 92 pixels, making the size 696*696. From each image, we have extracted 4 overlapping tiles of size 572*572. We have also performed this step for the validation dataset.
3. **Elastic transform**: We generate smooth deformations with random displacements sampled from a Gaussian distribution. 3 images are generated per tile using different parameters for the distribution.

Therefore, the training set has 25*(4+1)*4*(3+1) = 2000 images and the validation set has 5*4 = 20 images.

Data Augmentation Pseudo-Code

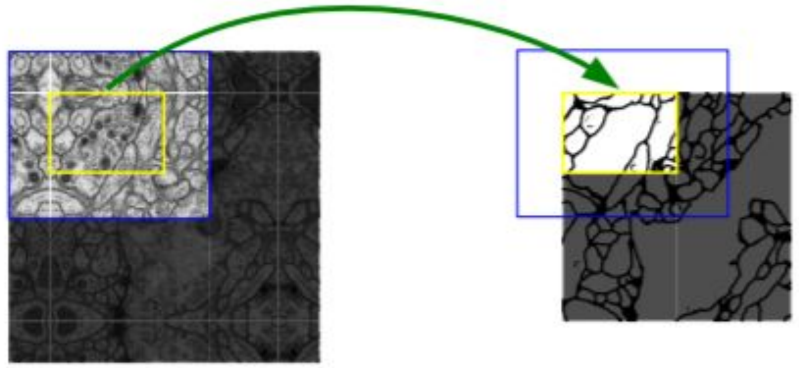


Fig. Overlap-tile strategy for segmentation

- Separate training and validation dataset
- For training images and label apply rotation, horizontal flip, shear and zoom
- Apply overlap Tile Strategy by adding a mirror pad of 92 pixels around the edges of training and validation data and labels and creating an image of 696*696 as shown in figure.
- Crop 4 windows of size 572*572 from the generated image.
- Apply elastic transform on training images and labels as explained before.
- Center crop the training and validation labels to generate an output of size 388*388.

Training

- The augmented images and their corresponding segmentation maps are used to train the network. Due to the unpadded convolutions, the output image (388*388) is smaller than the input (572*572).
- To minimize the overhead and make maximum use of the GPU memory, we favor large input tiles over a large batch size and hence reduce the batch to a single image.
- We use the Adam optimizer with beta_1 = 0.9, beta_2 = 0.999 and epsilon = 10^-7.

- For the weights, we use He-normal initialization: Gaussian distribution centred at 0 with a standard deviation of sqrt(2/N).
- We use two callbacks: ModelCheckpoint to save the best model and TensorBoard to store the training logs.
- The model is trained for 35 epochs with batch size equal to 4.
- Training code:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,update_freq='epoch')
model = unet()
model_checkpoint = ModelCheckpoint(model_name, monitor='loss',verbose=1, save_best_only=True)
history =
model.fit(x=np_img_train,y=np_lbl_train,batch_size=4,epochs=35,validation_data=(np_img_val,np_lbl
_val),callbacks=[model_checkpoint,tensorboard_callback])
```

Visualization pseudo code

```
#  "Accuracy"
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.savefig('unet_final_4_acc')
plt.show()
# "Loss"
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.savefig('unet_final_4_loss')
plt.show()
```

Results & Discussion

Results

Training dataset: 2000 augmented images and their corresponding segmentation maps.

Validation dataset: 20 images and their corresponding segmentation maps.

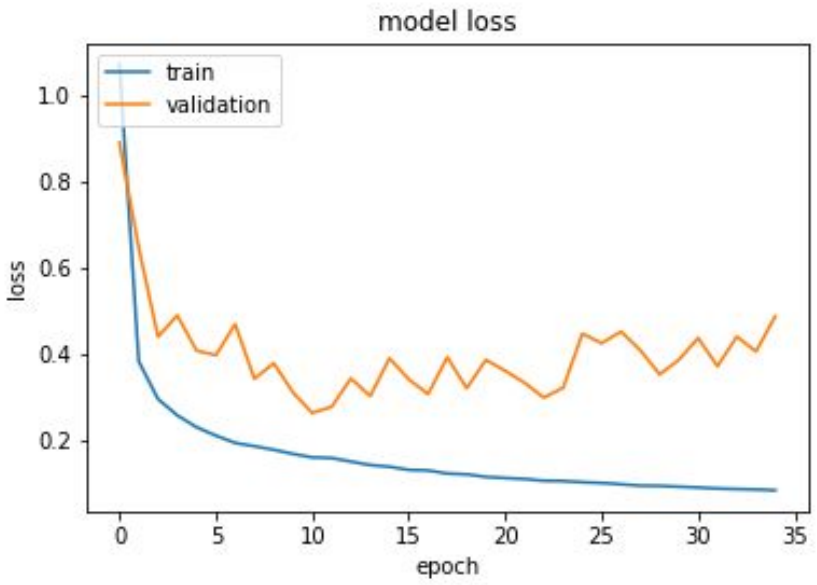
Training Accuracy: 96.39 %

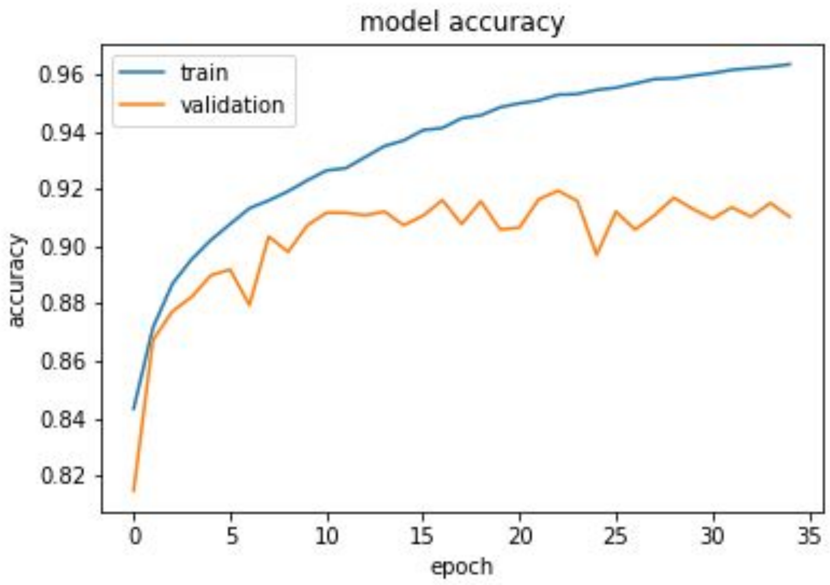
Validation Accuracy: 91.03 %

Discussion

- The model had a reasonable training time of 8 minutes per epoch on the gpu provided by Google Colab (NVIDIA Tesla K80 GPU 12GB). The total time for training was hence roughly 5 hours.
- Drop-out layers at the end of the contracting path perform further implicit data augmentation. This helps prevent overfitting.
- Mirroring, along with Overlap-tile strategy ensures that the information at the edges is not lost.

Visualizations





Comparison of results

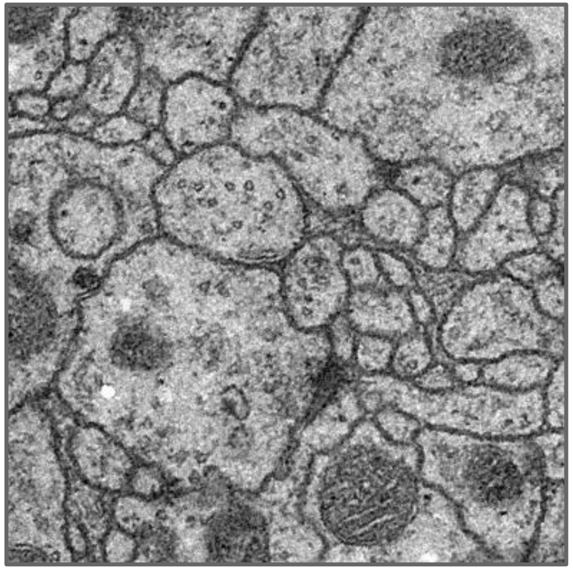
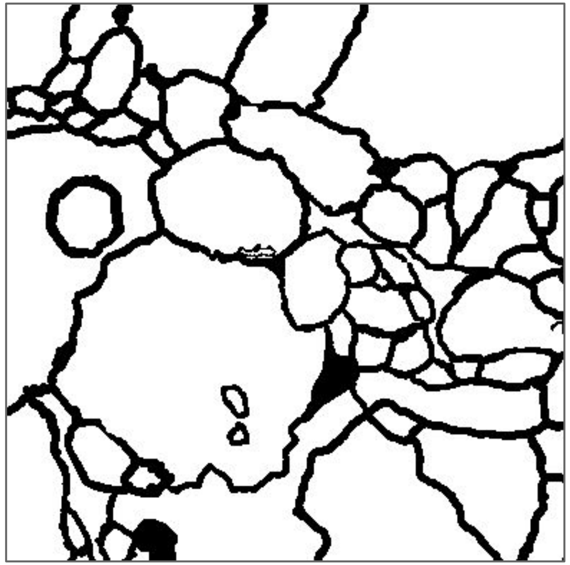
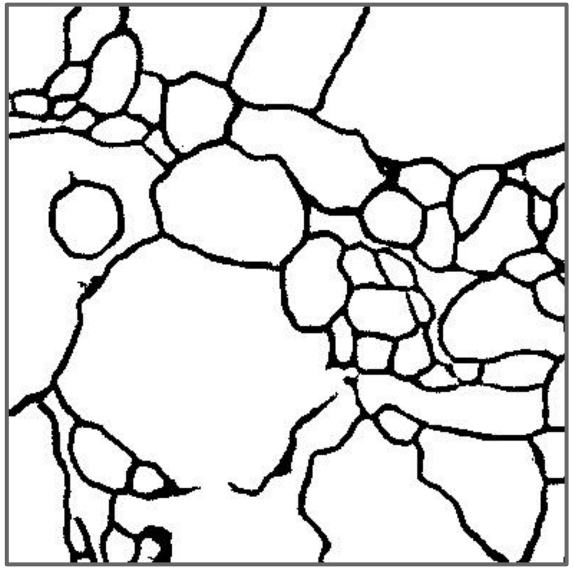


Image data


True label


Predicted label

Challenges Faced in Implementation

- We implemented the Unet architecture from scratch, because online implementations used same convolution and had several deviations from the original paper.
- This caused some initial size mismatch issues, which were later rectified.
- Implementation details for generation of weight maps were not clearly stated in the paper. Also training along with the weight maps was not feasible.

- The computational resources provided by Google colab are limited, this often led to crashes while training.

- While applying elastic deformations to images, we faced difficulty due to different input and output sizes. The parameters are dependent on the image size. We solved it by first generating maps of same size during augmentation and then cropping them as required.

Future Scope

- Thousands of training images are beyond reach in biomedical tasks and require experts and take lots of time to annotate. This could automate the process, this lowering the cost and time it takes to annotate.
- This can also be applied in other areas such as quality control and inspection and manufacturing.

Learning Outcomes

Some things we learnt from this assignment:

1. Biomedical applications of computer vision.
2. Implementation of a research paper from scratch.
3. New useful techniques to augment datasets, especially when very few training samples are available. Random elastic deformations make the most natural augmented data.
4. Efficient use of colab to train medium-sized models.