MARMARA
UNIVERSITY

CSE 4084 Multimedia Systems

Prof. Dr. Çiğdem Eroğlu Erdem

Department of Computer Engineering

Marmara University

Fall 2021

Final Project Report

Comparing Two Deep Neural Networks For Image Segmentation

Team Members

150119900 Emre Okul

150119679 Muhammedcan Pirinççi

Abstract

We want to compare two deep learning architectures for image segmentation. The first architecture we will use will use the U-NET architecture to analyze and generate segmentations for the image. The second algorithm we will use will use the "attention" mechanism to do the image segmentation task. Both architectures give accurate results and are popular and easy to use for developers. That's why we wanted to compare these two architectures. First, we will implement our initial model from the article “Convolutional Networks for Biomedical Image Segmentation[1]” and test the results. We will not change anything in architecture. We will implement the model as on the paper, then we will explain each part, each layer of the architecture. For the second architecture (architecture using the "attention" mechanism), "O. Oktay et al. “Attention UNet: learning where to look for the pancreas” [2] and repeat the other steps we did for the initial architecture.

Overview

In this project, we have accomplished the things below so far.

- We learned how to use particular deep learning architectures.
- We studied Keras-Tensorflow for understanding implementation of architectures better. [3] [4]
    - We understood program flow in tensorflow and keras. How to create models, how creating model flow works, how training flow works.
- We studied how to fine tune the deep learning architectures.
    - Fine tuning is changing architecture layers(input-output) to adapt our problems. Also we will use fine tuning to use freeze layers and train architecture for our data.
- We studied how to do increase performance of general deep learning models [5]:
    - Increasing performance with data:
        - Will get more data.
        - Augment our data.
        - Transform our data.
        - Rescale our data.
    - Increasing performance with hyperparameter tuning:

        - Changing batch size.
        - Changing learning rate.
        - Changing activation functions.
        - Changing number of hidden layers (in Fully connected neural networks).
- We studied how to increase performance of U-NET architecture and Attention mechanism.
    - For U-NET: Because we don't want to change architecture itself, we will just play with our data.
    - For Attention mechanism: Like U-NET, for not changing architecture we will use “Increasing performance with data” methods to increase performance of our model.

Our total tasks from beginning to end are as in the below.

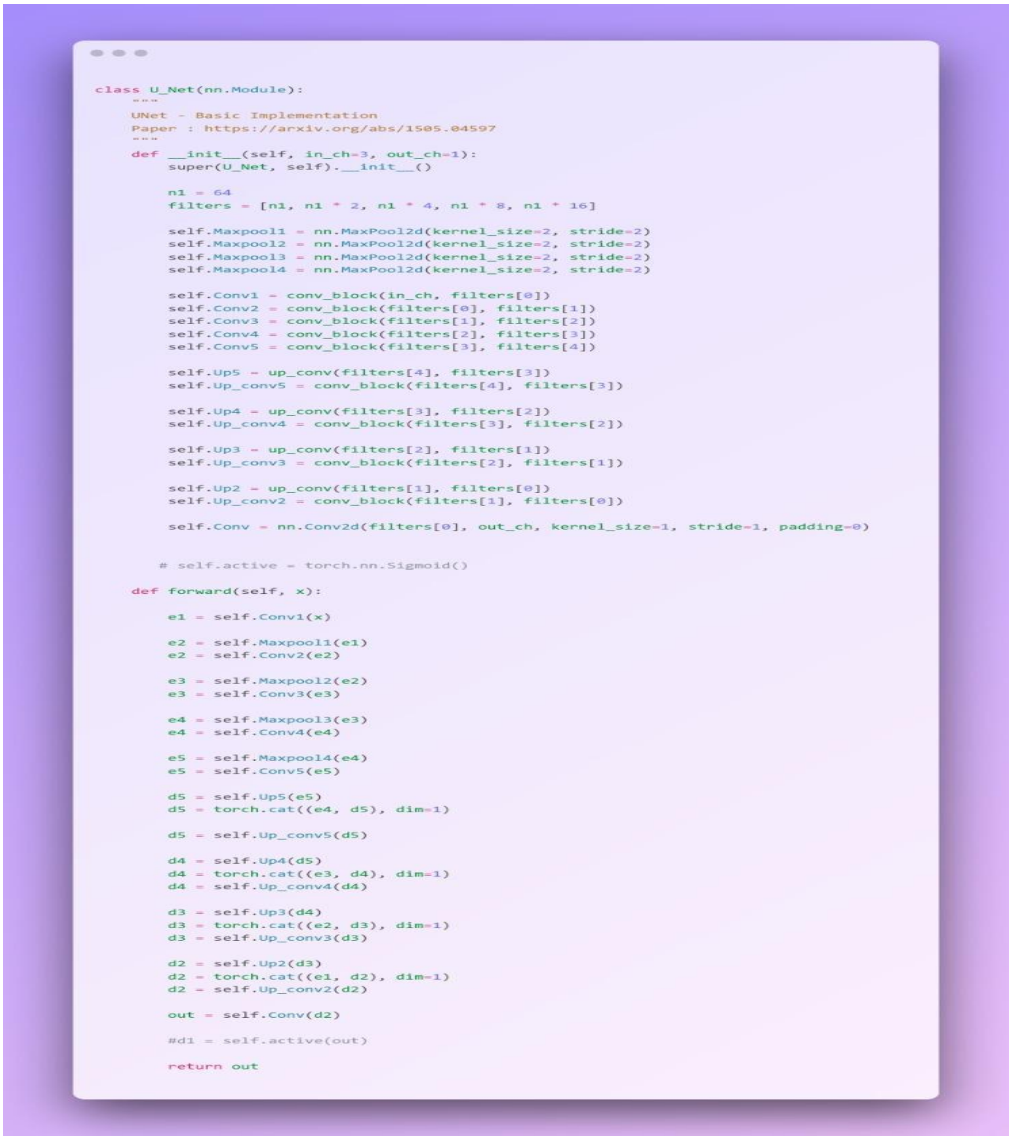
1. Data collection for our first model to train
2. Training → Fine Tuning → Increasing performance for First Architecture
    - 2.1. Training
    - 2.2. Fine Tuning
    - 2.3. Increasing performance
3. Training → Fine Tuning → Increasing performance for Second Architecture
    - 3.1. Training
    - 3.2. Fine Tuning
    - 3.3. Increasing performance
4. Comparing First and Second Architectures with Different Architectures

Project Accomplishment

1. Data collection for our first model to train

We collected our data from Kaggle [6]. We used 843 Mb of data. We used 10% of it in the validation process, 10% of it in the testing process and 80% of it in the training process. Also, we used 30 Mb of data for “train masks”. We transformed our pictures from our data to 224x224 resolution to fit our model. For input, we used 3 channels(RGB). For the input-output mask, we used 1 channel because we are testing our model with one class.

2. Training → Fine Tuning → Increasing performance for First Architecture



```python
class U_Net(nn.Module):
    """
    UNet - Basic Implementation
    Paper : https://arxiv.org/abs/1505.04597
    """
    def __init__(self, in_ch=3, out_ch=1):
        super(U_Net, self).__init__()

        n1 = 64
        filters = [n1, n1 * 2, n1 * 4, n1 * 8, n1 * 16]

        self.Maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.Conv1 = conv_block(in_ch, filters[0])
        self.Conv2 = conv_block(filters[0], filters[1])
        self.Conv3 = conv_block(filters[1], filters[2])
        self.Conv4 = conv_block(filters[2], filters[3])
        self.Conv5 = conv_block(filters[3], filters[4])

        self.Up5 = up_conv(filters[4], filters[3])
        self.Up_conv5 = conv_block(filters[4], filters[3])

        self.Up4 = up_conv(filters[3], filters[2])
        self.Up_conv4 = conv_block(filters[3], filters[2])

        self.Up3 = up_conv(filters[2], filters[1])
        self.Up_conv3 = conv_block(filters[2], filters[1])

        self.Up2 = up_conv(filters[1], filters[0])
        self.Up_conv2 = conv_block(filters[1], filters[0])

        self.Conv = nn.Conv2d(filters[0], out_ch, kernel_size=1, stride=1, padding=0)


       # self.active = torch.nn.Sigmoid()

    def forward(self, x):

        e1 = self.Conv1(x)

        e2 = self.Maxpool1(e1)
        e2 = self.Conv2(e2)

        e3 = self.Maxpool2(e2)
        e3 = self.Conv3(e3)

        e4 = self.Maxpool3(e3)
        e4 = self.Conv4(e4)

        e5 = self.Maxpool4(e4)
        e5 = self.Conv5(e5)

        d5 = self.Up5(e5)
        d5 = torch.cat((e4, d5), dim=1)

        d5 = self.Up_conv5(d5)

        d4 = self.Up4(d5)
        d4 = torch.cat((e3, d4), dim=1)
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        d3 = torch.cat((e2, d3), dim=1)
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        d2 = torch.cat((e1, d2), dim=1)
        d2 = self.Up_conv2(d2)

        out = self.Conv(d2)

        #d1 = self.active(out)

        return out
```

Figure 1. U-NET Architecture in code

U-NET, the first architecture of our project, was developed from a neural network. This architecture was originally designed and implemented for the analysis of anatomical images in the field of biomedicine. In this analysis, the focus is on image classification by taking an anatomical image as input and creating an image as a label as output. In this way, in the biomedical field, it has provided to understand not only whether a patient has a disease, but also the region of the disease. We can explain the application flow of this architecture in our project in 2 parts, decoder and encoder. In the decoder part, the contracting path follows the formula in the Figure 2 below.

```
conv_layer1 -> conv_layer2 -> max_pooling -> dropout(optional)
```

Figure 2

As can be seen in the Figure 3 below, there are 2 convolutional layers formed by 2 convolution processes, which represent 2 blue arrows. The channel numbers of the layer formed as a result of the first convolution process increased from 1 to 64, and this increased the depth of the image. The process that comes after the 2 convolution processes is max pooling, as seen in the Figure 3 below with the red arrow, and the size of the image has decreased to half in this process. After repeating these 3 processes that we have mentioned so far, only 2 convolution processes will repeat 1 more time without the max pooling process.

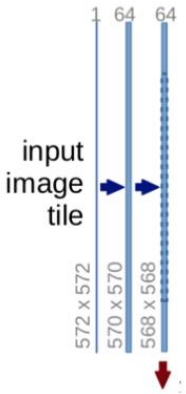


Figure 3

In decoder part after encoder part, the expansive path follows the formula in the Figure 4 below to upsize the image from the encoder part to its original size. The transposed convolution as conv_2d_transpose in the Figure 4 below in this formula is an upsampling technique that expands the size of the input image and does some padding on the original image before the convolution operation as in the Figure 5 below.

```
conv_2d_transpose -> concatenate -> conv_layer1 -> conv_layer2
```

Figure 4

Figure 5

However, this architecture includes some skip connections, and this is because it combines information from previous layers to make a more precise

estimate. Finally, a 1x1 convolution operation is performed while obtaining the final layer as in the Figure 6 below.

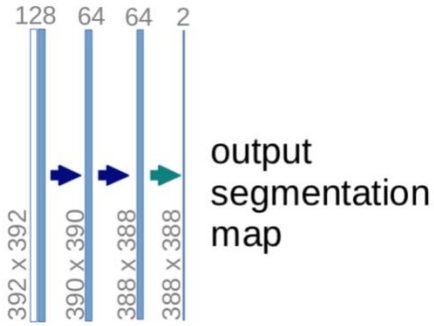


Figure 6

2.1. Training

For U-NET Architecture as our first architecture, we used Dice Coefficient for our loss function and we trained our kernels based on this loss function. And based on validation, if our model's validation score is decreasing per epoch, we will keep the model that has a better validation score.

```
1m 8s
QPQPQPQPQPQPQPQPPQPQ
0
QPQPQPQPQPQPQPQPPQPQ
Epoch: 2/5      Training Loss: 0.040150         Validation Loss: 0.018027
Validation loss decreased (0.066770 --> 0.018027).  Saving model
4m 41s
QPQPQPQPQPQPQPQPPQPQ
0
QPQPQPQPQPQPQPQPPQPQ
Epoch: 3/5      Training Loss: 0.016580         Validation Loss: 0.016018
Validation loss decreased (0.018027 --> 0.016018).  Saving model
4m 41s
QPQPQPQPQPQPQPQPPQPQ
1
QPQPQPQPQPQPQPQPPQPQ
Epoch: 4/5      Training Loss: 0.100352         Validation Loss: 0.053945
1m 7s
QPQPQPQPQPQPQPQPPQPQ
1
QPQPQPQPQPQPQPQPPQPQ
Epoch: 5/5      Training Loss: 0.050793         Validation Loss: 0.041258
1m 7s
```

Figure 7. U-NET Training

2.2. Fine Tuning

Before we fine tune, there were 3 channels in the output. We decreased it to 2.

2.3. Increasing performance

We increased the performance with changing learning rate and batch size as hyperparameters. Our dice scores are as in the Figure 7 and 8 below.

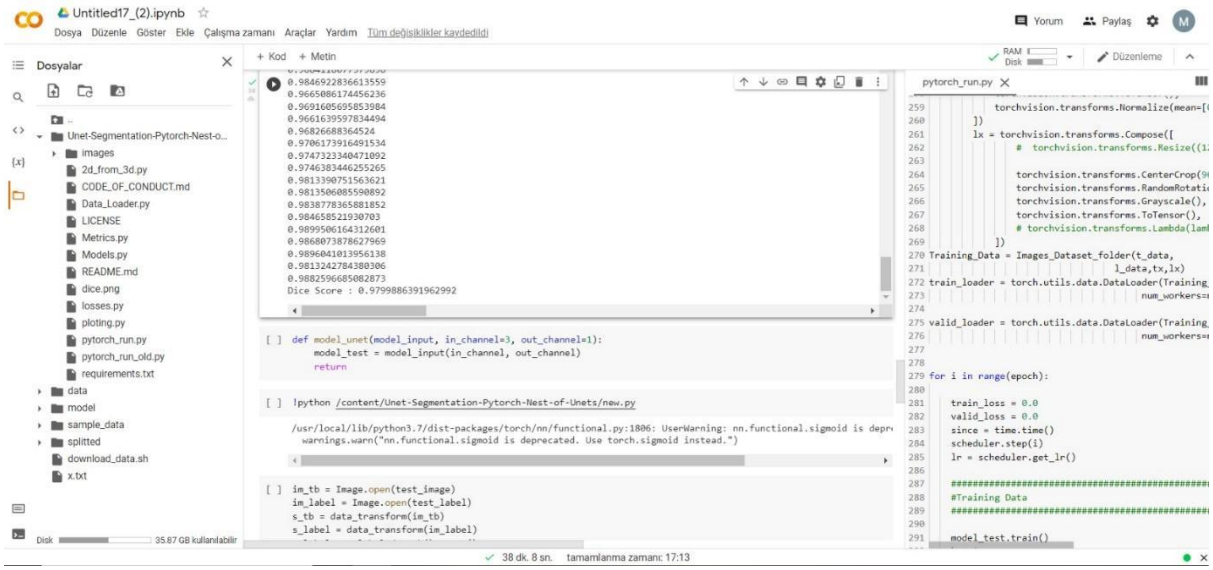


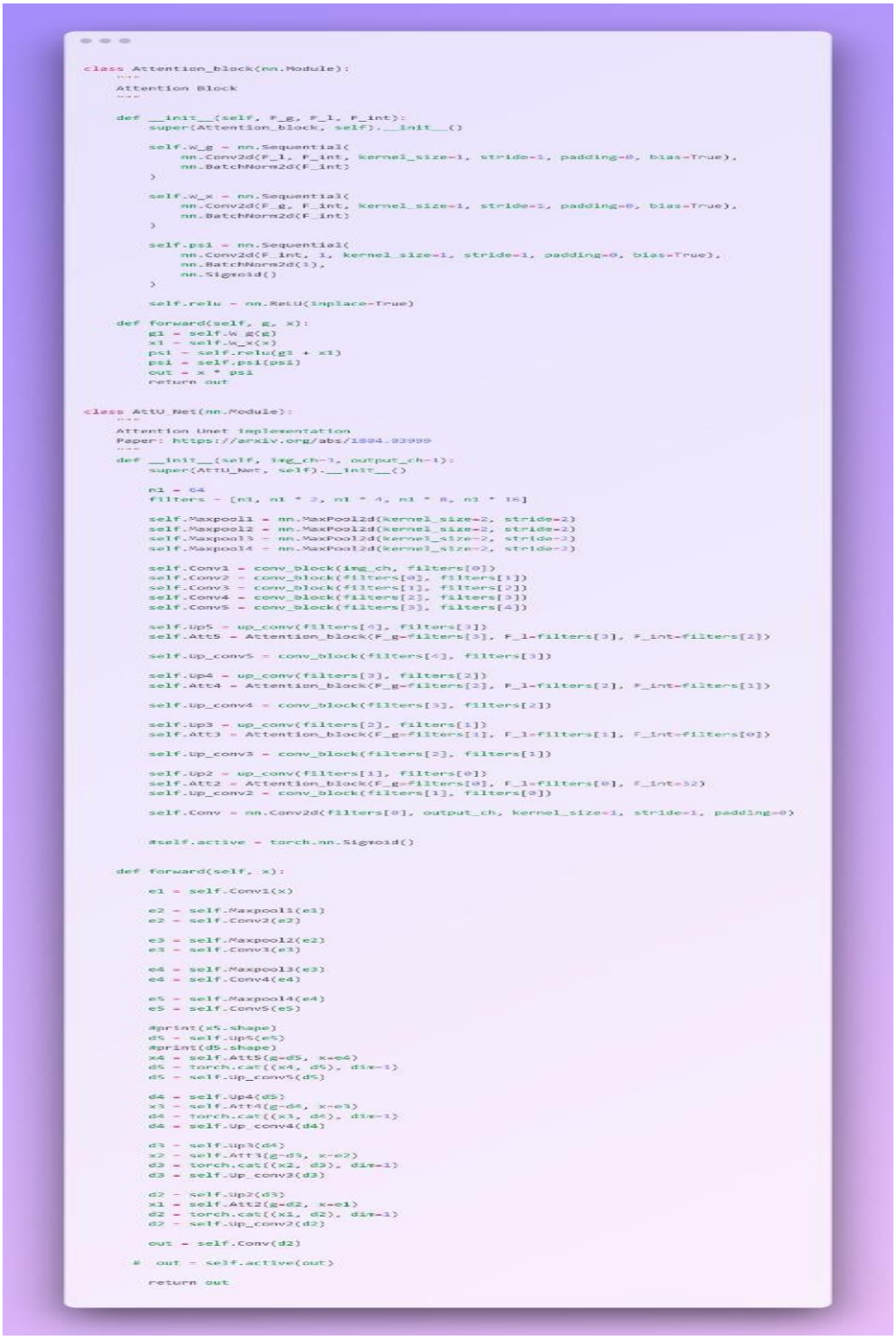
Figure 8. Old Dice Score of U-NET

```
0.9859313449634215
0.9857315034407682
0.9755154031230835
0.9818254925613189
0.9901567910360658
0.9908791293017551
0.9885192551075754
0.9917054420392474
0.9882967404971468
0.9913263462233466
0.9899110263743247
0.9906387665198237
0.99174751419025
0.9865436099760462
0.9828083535248644
0.988146797355824
Dice Score : 0.9872789425704963
```

```
[ ]  def model_unet(model_input, in_channel=3, out_channel=1):
         model_test = model_input(in_channel, out_channel)
         return
```

Figure 9. New Dice Score of U-NET

3. Training → Fine Tuning → Increasing performance for Second Architecture



```
class Attention_block(nn.Module):
    """
    Attention Block
    """

    def __init__(self, F_g, F_l, F_int):
        super(Attention_block, self).__init__()

        self.W_g = nn.Sequential(
            nn.Conv2d(F_l, F_int, kernel_size=1, stride=1, padding=0, bias=True),
            nn.BatchNorm2d(F_int)
        )

        self.W_x = nn.Sequential(
            nn.Conv2d(F_g, F_int, kernel_size=1, stride=1, padding=0, bias=True),
            nn.BatchNorm2d(F_int)
        )

        self.psi = nn.Sequential(
            nn.Conv2d(F_int, 1, kernel_size=1, stride=1, padding=0, bias=True),
            nn.BatchNorm2d(1),
            nn.Sigmoid()
        )

        self.relu = nn.ReLU(inplace=True)

    def forward(self, g, x):
        g1 = self.W_g(g)
        x1 = self.W_x(x)
        psi = self.relu(g1 + x1)
        psi = self.psi(psi)
        out = x * psi
        return out

class AttU_Net(nn.Module):
    """
    Attention Unet Implementation
    Paper: https://arxiv.org/abs/1804.03999
    """

    def __init__(self, img_ch=3, output_ch=1):
        super(AttU_Net, self).__init__()

        n1 = 64
        filters = [n1, n1 * 2, n1 * 4, n1 * 8, n1 * 16]

        self.Maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.Maxpool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.Conv1 = conv_block(img_ch, filters[0])
        self.Conv2 = conv_block(filters[0], filters[1])
        self.Conv3 = conv_block(filters[1], filters[2])
        self.Conv4 = conv_block(filters[2], filters[3])
        self.Conv5 = conv_block(filters[3], filters[4])

        self.Up5 = up_conv(filters[4], filters[3])
        self.Att5 = Attention_block(F_g=filters[3], F_l=filters[3], F_int=filters[2])

        self.Up_conv5 = conv_block(filters[4], filters[3])

        self.Up4 = up_conv(filters[3], filters[2])
        self.Att4 = Attention_block(F_g=filters[2], F_l=filters[2], F_int=filters[1])

        self.Up_conv4 = conv_block(filters[3], filters[2])

        self.Up3 = up_conv(filters[2], filters[1])
        self.Att3 = Attention_block(F_g=filters[1], F_l=filters[1], F_int=filters[0])

        self.Up_conv3 = conv_block(filters[2], filters[1])

        self.Up2 = up_conv(filters[1], filters[0])
        self.Att2 = Attention_block(F_g=filters[0], F_l=filters[0], F_int=32)
        self.Up_conv2 = conv_block(filters[1], filters[0])

        self.Conv = nn.Conv2d(filters[0], output_ch, kernel_size=1, stride=1, padding=0)


        #self.active = torch.nn.Sigmoid()

    def forward(self, x):

        e1 = self.Conv1(x)

        e2 = self.Maxpool1(e1)
        e2 = self.Conv2(e2)

        e3 = self.Maxpool2(e2)
        e3 = self.Conv3(e3)

        e4 = self.Maxpool3(e3)
        e4 = self.Conv4(e4)

        e5 = self.Maxpool4(e4)
        e5 = self.Conv5(e5)

        #print(x5.shape)
        d5 = self.Up5(e5)
        #print(d5.shape)
        x4 = self.Att5(g=d5, x=e4)
        d5 = torch.cat((x4, d5), dim=1)
        d5 = self.Up_conv5(d5)

        d4 = self.Up4(d5)
        x3 = self.Att4(g=d4, x=e3)
        d4 = torch.cat((x3, d4), dim=1)
        d4 = self.Up_conv4(d4)

        d3 = self.Up3(d4)
        x2 = self.Att3(g=d3, x=e2)
        d3 = torch.cat((x2, d3), dim=1)
        d3 = self.Up_conv3(d3)

        d2 = self.Up2(d3)
        x1 = self.Att2(g=d2, x=e1)
        d2 = torch.cat((x1, d2), dim=1)
        d2 = self.Up_conv2(d2)

        out = self.Conv(d2)

      #  out = self.active(out)

        return out
```

Figure 10. Attention U-NET Architecture Code

During upsampling in the expanding path, the reconstructed spatial information is imprecise. To resolve this issue, U-Net uses hopping links that combine spatial information from the downsampling path with the upsampling path. However, this results in a lot of unnecessary low-level feature extraction, as the feature representation is poor in the early layers. Soft attention applied in jump

connections actively suppresses activations in irrelevant regions, reducing the number of redundant features that arise.

3.1. Training

Our second architecture is Attention U-NET. This architecture takes account of attention so that it reduces the computational resources wasted on irrelevant activations by providing the network with better generalization power.

3.2. Fine Tuning

Before we fine tune, there were 3 channels in the output. We decreased it to 2.

3.3. Increasing performance

We increased the performance with changing learning rate and batch size as hyperparameters as in the Figure 11 and 12 below.

```
0.992766414010314
0.9926771048266375
0.9923115255926532
0.9841063515509602
0.9904045444077685
Dice Score : 0.9893833034295003
```

Figure 11. Old Dice Score of Attention U-NET

```
0.9931614406565017
0.9895219065256592
0.9915591462679368
0.9905844026856303
0.9919856911911396
0.9931045606677689
0.993297159272263
0.9878812715883145
0.9914882247414023
Dice Score : 0.9902483569681797
```

Figure 12. Attention U-NET Accuaracy After Performance Improve

U-NET and Attention U-NET Metrics

As evaluation metrics, it is necessary to talk about 2 subjects, namely Pixel Accuracy and Dice Coefficient [7].

1. Pixel Accuracy

To explain Pixel Accuracy, for example, the pixel accuracy of the image in the Figure 13 below is very high. However, the segmentation of the image looks like the Figure 14 below.

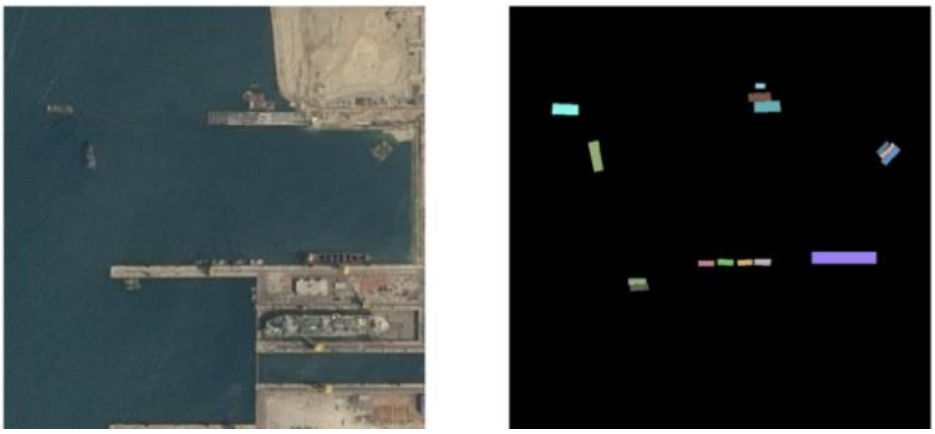


Figure 13

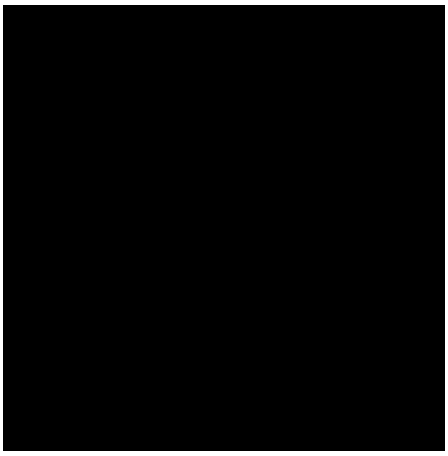


Figure 14

This is due to class imbalance. That is, if classes are extremely unstable, it means that a class or some classes dominate the visual. However, the remaining classes are not significant as they only make up a small part of the image.

2. Dice Coefficient

Simply put, Dice Coefficient is the Area of Overlap divided by the total number of pixels in both images, after multiplying by 2 as in the Figure 15 below.

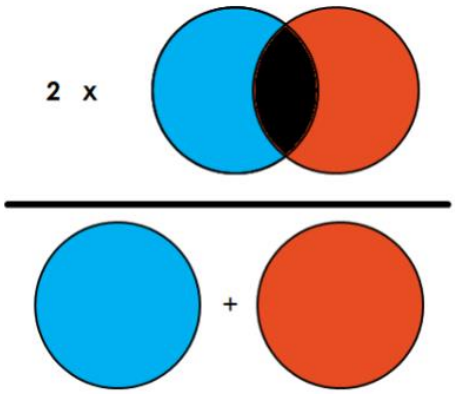


Figure 15

4. Comparing First and Second Architectures with Different Architectures

We will explain this part in the next section.

Summary

When we compare the two architectures we have used in the project, Attention U-NET performs better. Because this architecture focuses on more important details and gives more accurate results. On the other hand, although U-NET shows almost a similar performance, it will be inevitable for us to use Attention U-NET as the dataset we use is more detailed. As we explained above, both architectures have their own performances. However, when we need to choose between two architectures according to the dataset we use, this choice will be easier for the user, considering the research we have done.

References

[1] U-Net: Convolutional Networks for Biomedical Image Segmentation, (2021, November, 11), https://arxiv.org/pdf/1505.04597.pdf

[2] O. Oktay, (2018), et al. “Attention UNet : learning where to look for the pancreas”

[3] Keras, (2021, December, 5), https://keras.io/

[4] TensorFlow, (2021, December, 5), https://www.tensorflow.org/

[5] How To Improve Deep Learning Performance, (2021, December, 5), https://machinelearningmastery.com/improve-deep-learning-performance/

[6] Kaggle, (2021, December, 3), https://www.kaggle.com/

[7] TowardsDataScience, (2022, January 18), https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation-model-6bcb99639aa2