Todo list

Universität Stuttgart
Institut für Statik und Dynamik
der Luft- und Raumfahrkonstruktionen
Prof. Dr.-Ing. Tim Ricken

Near Realtime Damage Detection of Wooden Planks

Lena Scholz, Samuel Lehmköster
& Julian Franquinet

Supervisors:
André Mielke M. Sc.,
Luis Mandl M. Sc.

Pfaffenwaldring 27, 70569 Stuttgart

Report

September 24, 2021

Contents

List of Figures

Greek letters

Symbol	Unit	Description
α	–	learning rate
λ	–	regularisation

Abbreviation

Abbreviation	Description
CNN	Convolutional Neural Network
Param.	Parameter
nS	Trainingsset without shift
wS	Trainingsset with shift
ES	Early stopping method applied
MC	Multi-Classification
SC	Single-Classification
BN	Batch normalization function applied
ReLU	Rectified linear unit
SELU	Scaled exponential linear unit

Listings

1 Introduction

1.1 Description of the Task

In this report the segmentation of wooden panels in a 256x3072 images is investigated for a real-time analysis task. The motivation for this report is the creation of an complete hardware and software system for a real-time damage detection on coated wooden panels in a fully automatized factory for built-in kitchen furniture.
The task is to detect damages due to a coating process on wooden panels. In order to be able to adjust the sensitivity of the detection, it is divided into two main tasks. The first task is the segmentation of the image in order to find the wooden panel inside the image. The second task is the actual detecting of the damage. This should be easily accomplished by a simple threshold. Therefore this work will focus on the more complicated part, creating and training a Convolutional Neural Network (CNN) for the purpose of the segmentation. For the training, real example pictures (1.1) of automatically coated wooden panels were used. The Segmentation Network should find these panels in a picture section and distinguish them from the conveyor belt system in the background.

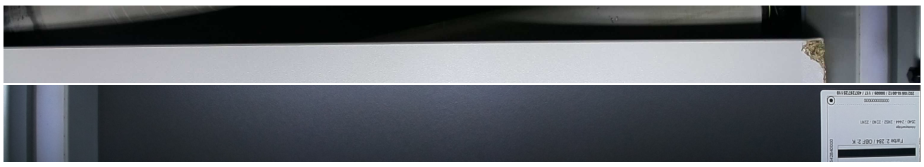

Figure 1: Example of the coated wooden panels with an error on the top right corner in the white one and a lobel on the black one

vlt noch ein bild mit sichtbarem label

1.2 Approach to the Solution

For the approach to the solution for this problem at first a pre-trained encoder with a small decoder as segmentaion network was applied. Therefore several pre-trained encoders of the `Tensorflow` library were investigated and later on different decoder structures tested. Additional to this simple Fast-forward CNNs [4] smaller, self-edited codes for more complex Segmentation Network Architectures were researched later on, like the U-Net[5] and the SegNet[1].
The original example set contains 126 pictures of the size 256x3072 further referred as 'big images'. In a data generation process, the images were split, flipped and shifted, in order to generate more data for the training. Because of the usage of pre-trained encoders, which have a build-in input (no further layers were included on the input-side of the encoders), the original image (1.1) with a size of 256x3072 is split into 12 images of 256x256, further referred to as 'sliced images'.
Since some of the planks have bar code labels, two setups are considered. At

first we only aim to distinguish planks from the background (single-class setup). In order to get rid of the confusion around labels we separate planks, labels and background (multi-class setup).

2 Pre-trained Encoders with a Simple Decoder

In this first part of the project the pre-trained encoders of the `VGG16`, the `MobileNetV2` and the `ResNet50V2` were chosen out of the available models in the keras software (`https://keras.io/api/applications`). These were selected due to there broad use in other open projects and their further development. Furthermore these CNN differ a lot in the amount of weights and thus it could be evaluated if deeper CNN perform better fin this task. The simple Decoder, which was used, is shown here:

```
def generate_model(img_size, multiclass = True):

    encoder = MobileNetV2(include_top=False, weights='imagenet',
                                        input_shape=(img_size),
                                         classifier_activation =None)
    encoder.trainable = False

    d1 = UpSampling2D(size=(2, 2))(encoder.layers[-1].output)
    c1 = Conv2D(8, kernel_size=(3, 3), activation='selu',padding='same')(d1)
    d2 = UpSampling2D(size=(2, 2))(c1)
    c2 = Conv2D(16, kernel_size=(3, 3), activation='selu',padding='same')(d2)
    d3 = UpSampling2D(size=(2, 2))(c2)
    c3 = Conv2D(16, kernel_size=(3, 3), activation='selu',padding='same')(d3)
    d4 = UpSampling2D(size=(2, 2))(c3)
    c4 = Conv2D(16, kernel_size=(3, 3), activation='selu',padding='same')(d4)
    d5 = UpSampling2D(size=(2, 2))(c4)

    if mutliclass:
        c5 = Conv2D(3, kernel_size=(1, 1), activation='softmax',padding='same')(d5)
    else:
        c5 = Conv2D(1, kernel_size=(1, 1), activation='sigmoid',padding='same')(d5)

    output = c5

    model = Model(inputs=encoder.inputs, outputs=output)
    return model
```

Listing 1: Code for the generation of the Machine Learning Model

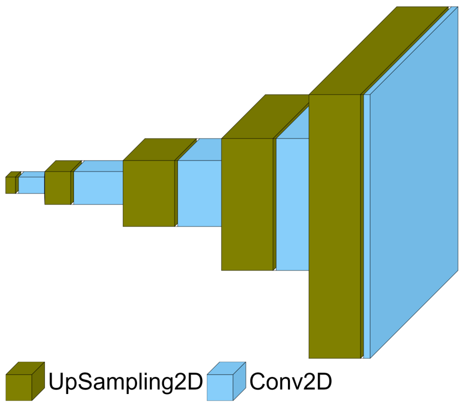

Figure 2: Architecture of simple decoder

2.1 VGG16

This CNN was build for the ImageNet Challenge 2014 for image classification and localisation[8]. It's therefore the oldest architecture. The objective was to develop deeper CNN architectures, here with with 23 weigh layer and over 138.00.000 parameters. The `VGG16` is the biggest segmentation net used in this project regarding the amount of parameters.

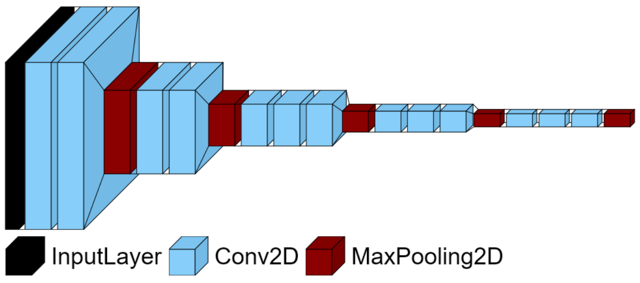

Figure 3: Architecture of VGG16

2.2 MobileNetV2

The `MoblieNetV2`, introduced in [6], was build and improved in order to obtain a small mobile model for semantic segmentation. It is the smallest pre-trained model and also the latest model used in this project. It contains around 3.500.000 parameters in 88 layers.

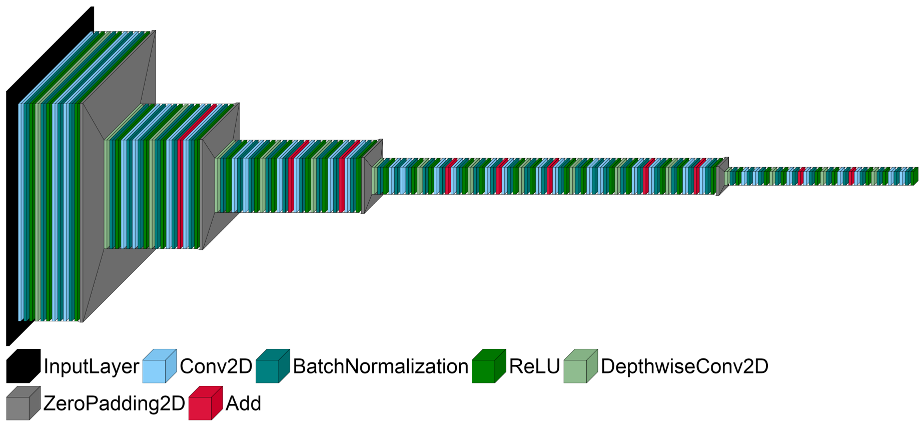

Figure 4: Architecture of MobileNetV2

2.3 ResNet50V2

The `ResNet50V2` architecture was build in order to ease the training of deep CNN (see [2]) by using layers as residual functions regarding the input. Its size lays between the `VVG16` and the `MobileNetV2` while containing around 25.600.000 parameters.

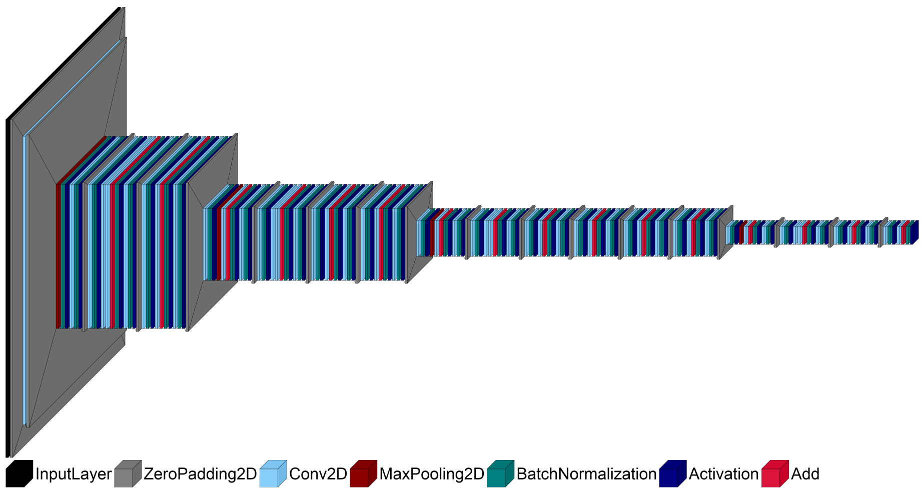

Figure 5: Architecture of ResNet50V2

3 Advanced Net Architectures and Decoder

In this second part of the project, different architectures of segmentation nets are tested. Because of the different difficulties in every architecture, not all CNN performed the same test cases. So several different settings in each investigation are applied.

3.1 U-Net Architecture

The U-Net architecture was motivated by biological image segmentation and the winning segmentation net in the ISBI challenge 2015. Similar to the SegNet, it applies encoder feature maps to the decoder in order to use trained feature maps more efficiently. With this approach, the U-Net presented in [5] could achieve very good results by using only little data for the training.

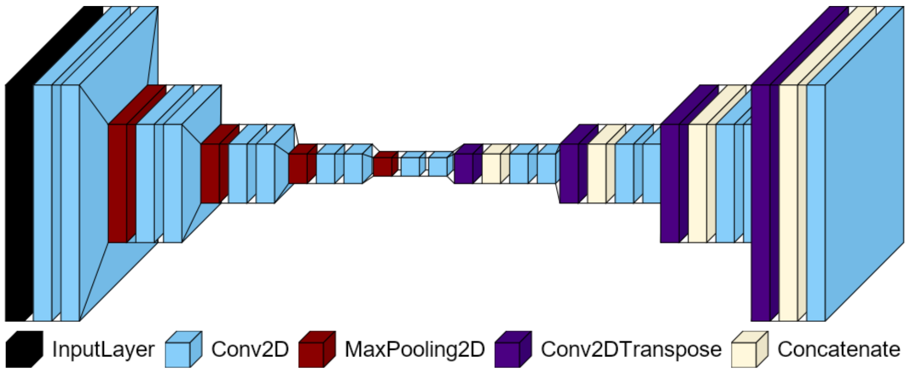

Figure 6: Architecture of U-Net with complexity of 4

3.2 SegNet Architecture

The SegNet Architecture in [1] was created for a good segmentation performance while using fewer parameters. This is achieved by using max-pooling indices from the encoder for the up-sampling in the decoder. It was motivated for scene understanding applications.

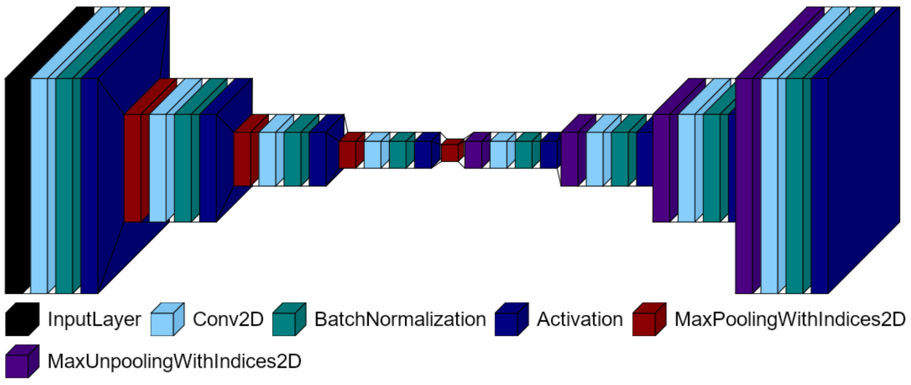

Figure 7: Architecture of simplified SegNet

Alternative Formulierung: In order to accelerate the training of the SegNet, a reduced architecture with fewer parameters (see fig. 7) is considered as it is in ... presented.

In order to limit the number of parameters in the net that need to be trained, we consider a reduced setup (see fig. 7) of the architecture presented in [1]. The SegNet architecture can be extended by adding more blocks of `Conv2D`, `BatchNormalization` and `Activation` before the `MaxPoolingWithIndices` is performed. The presented implementation below is based on existing setups using keras (see `https://github.com/danielenricocahall/Keras-SegNet` and `https://github.com/Runist/SegNet-keras`).
In contrast to the other models that are presented within the scope of this project the SegNet requires the use of layers that are not predefined in `Keras` for the use of the mentioned max-pooling indices. This concerns `MaxPoolingWithIndices2D` and `MaxUnppolingWithIndices2D`.

3.3 Advanced Decoders

Starting with the simple decoder in the first part of the project (listing 1), more complex decoders were created and tested. In order to have more parameters for the learning, additional layers like `Upsampling`, `Conv2D`, `Conv2DTranspose`, `BatchNormalization`, `Activation`, `Add` and `Dropout` are applied in the decoder architecture. Different combinations are tested as well as an architecture inspired by the examples for Computer Vision (`https://keras.io/examples/vision/oxford_pets_image_segmentation/`). The encoder used for these investigations is the `MobileNetV2`, because of its fewer parameters and therefore faster training and prediction. The most important test are shown in this report:

- Multi- & Single-Class Segm. with `MobileNetV2` using `Conv2DTranspose` with and without `BatchNormalization`

- Multi- & Single-Class with `MobileNetV2` using two `Conv2D` while `Add` an `Upsampling`
- Multi- & Single-Class with `MobileNetV2` using `Conv2DTranspose` while `Add` an `Upsampling`
- Multi- & Single-Class with `MobileNetV2` using `Dropout` while `Add` an `Upsampling`
- Multi- & Single-Class with `MobileNetV2` using two paths with different layer structures, seen in listing 4.

In the following code the decoder structure for the most advanced case is shown. In all of these test, five layer were applied in the decoder. These layers were built always exactly the same and similar but simpler to the here shown structure. They all consisted of an addition of an `Upsampling` and another path including more parameters. Before the paths were added a `BatchNormalization` and an `Activation` were computed.

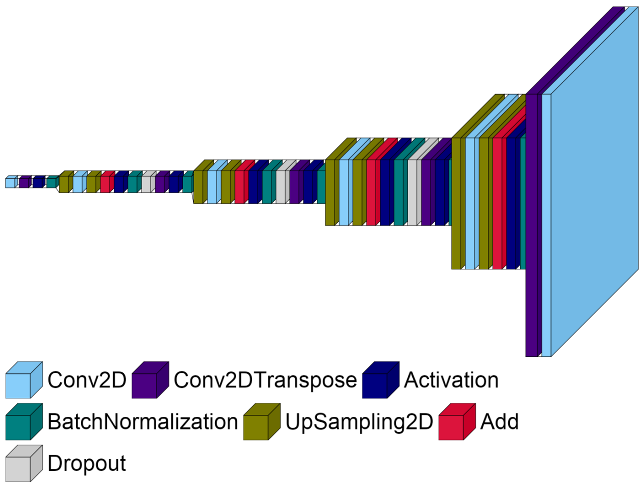

Figure 8: Architecture of the Advanced Decoder

4 Final Segmentation Architecture with a Pre-trained Encoder

For the final segmentation net, the U-Net architecture was combined with a `MobileNetV2` pre-trained encoder. The skipped connections from the encoder were exported and added in the different decoder layers as it can be seen in listing 8. This combination was trained in over 100 epochs for a single-classification and a multi-classification.

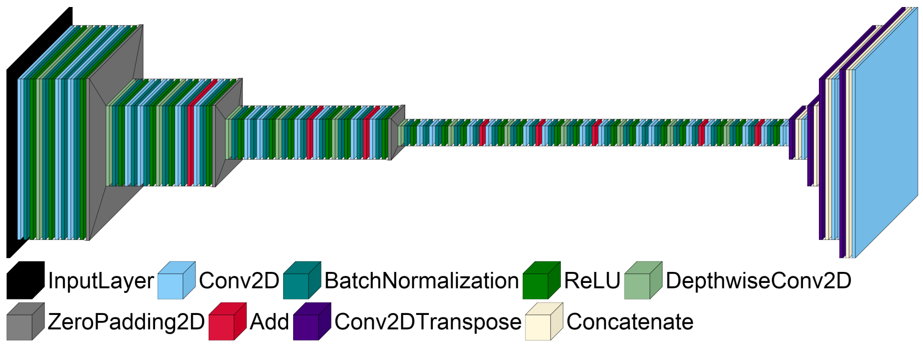

Figure 9: Architecture of Final Model

5 Evaluation of the results

5.1 Pre-trained Encoders with a Simple Decoder

In the first investigations, three pre-trained encoder were applied together with a simple decoder architecture. Only the decoder were trained. The `VGG16`, the `MobileNetV2` and the `ResNetV2` were to be tested.
The results for single- & and multi-classification can be seen in the results plots in fig. 10 and fig. 11. The results were achieved by using an early stopping as `callback` in training and with a shifting of the images in the data generation of the training data.

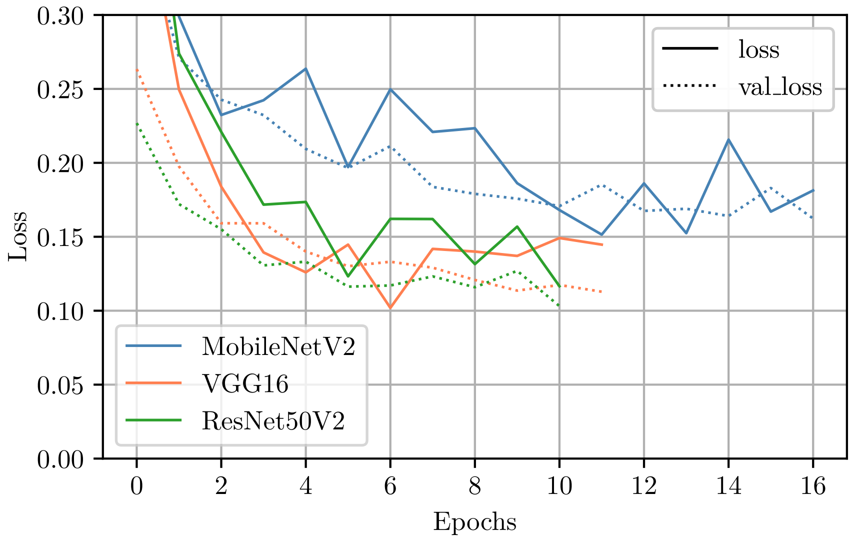

Figure 10: Losses of different pre-trained encoder for single-classification with training settings: Shift = True; Multi-Class = False; EarlyStopping = True; Epochs = 50; Image size = 256x256

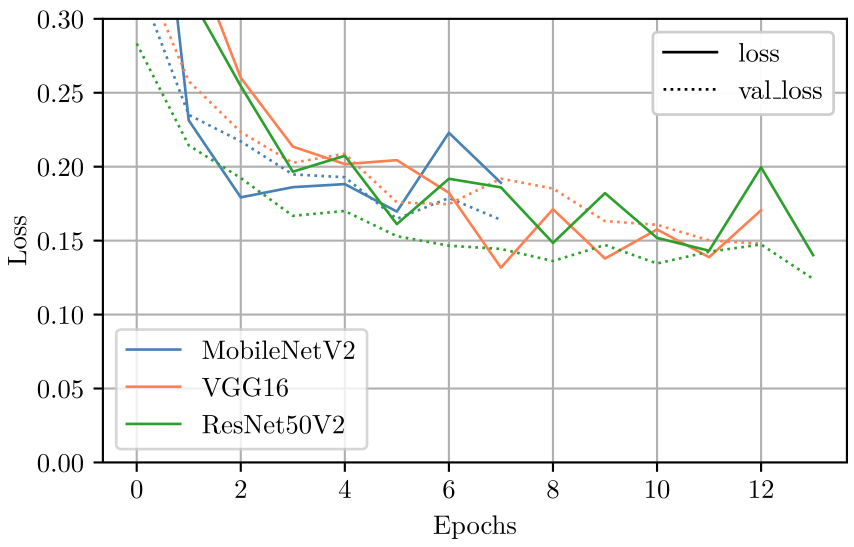

Figure 11: Losses of different pre-trained encoder for multi-classification with training settings: Shift = True; Multi-Class = True; EarlyStopping = True; Epochs = 50; Image size = 256x256

It was found that the shifting of the images for data generation could yield more training data, but hindered the CNN to achieve finer results. Also the early stopping ended the training to early, without beeing sure that convergence of the loss-curve was already reached.
Thus another training of the pre-trained encoders was performed with these improvements. The loss-curves for single- (fig. 12) & multi-classification (fig. 14) show this improvement of the learning regarding the previous results (fig. 10 & fig. 11).

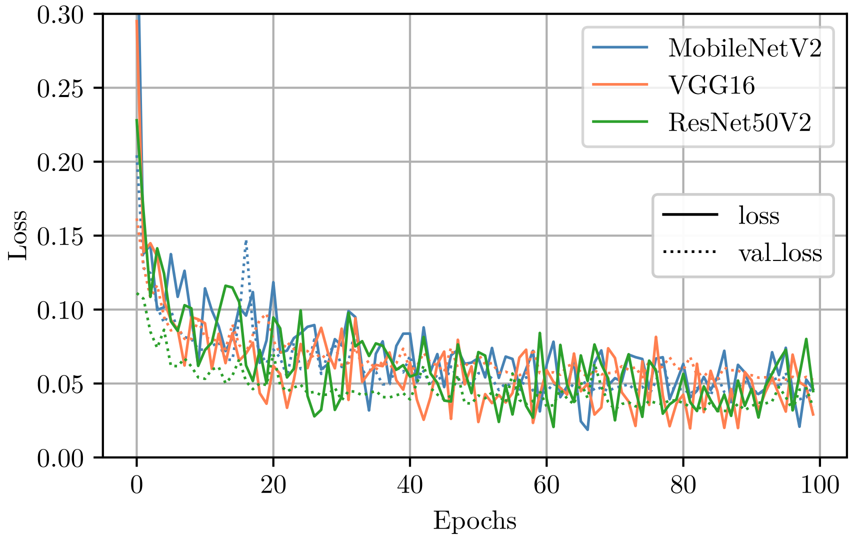

Figure 12: Losses of different pre-trained encoder for single-classification with training settings: Shift = False; Multi-Class = False; EarlyStopping = False; Epochs = 100;Image size = 256x256

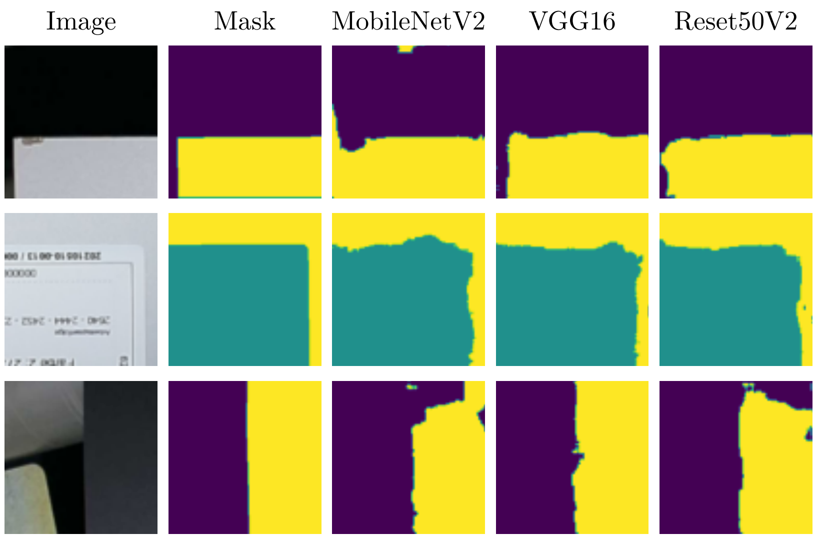

Figure 13

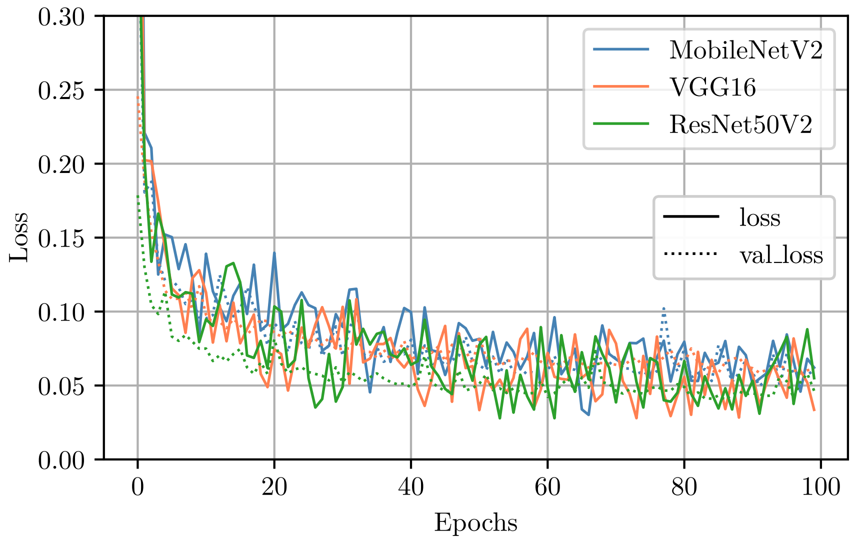

Figure 14: Losses of different pre-trained encoder for multi-classification with training settings: Shift = False; Multi-Class = True; EarlyStopping = False; Epochs = 100; Image size = 256x256

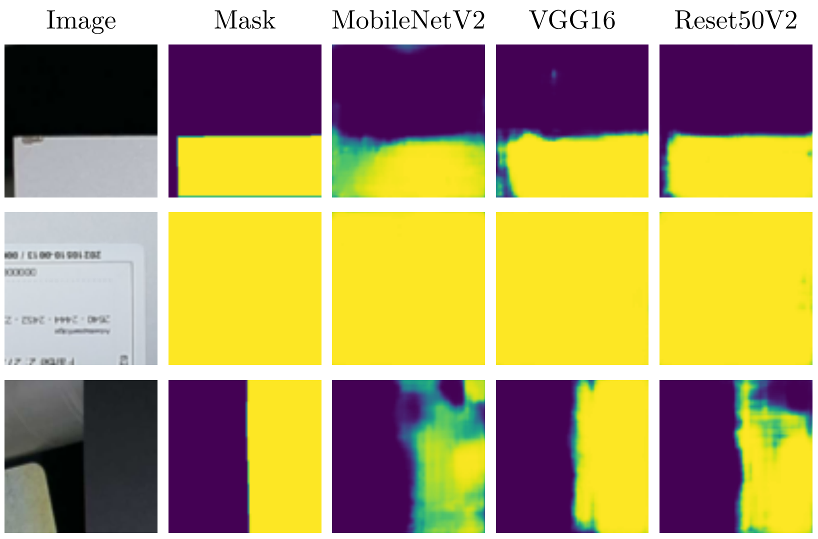

Figure 15

5.2 SegNet

As one can see from fig. 16 even the simplified version of the SegNet convinces with its ability to accurately detect boundaries. However, relevant boundaries in the background are detected as well, especially in the case of dark-colored planks. This causes the prediction within the homogeneous surfaces to be significantly blurred compared to other architectures considered within this project.

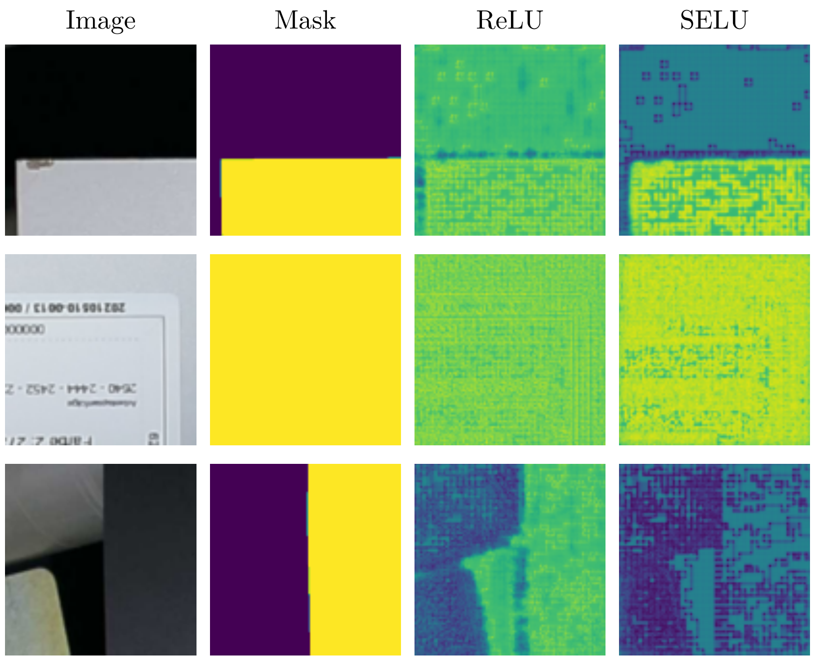

Figure 16: Predictions of simplified SegNet for ReLU- and SELU-activation with training settings: Shift = False, Multi-Class = False; EarlyStopping = False; Epochs = 50; Image size = 256x256

Despite the use of `ReLU-Activation` in [1] we considered a replacement by `SELU-Activation`. The respective losses of the training in fig. 17 with both activation functions do not indicate significant differences. In the presented images the detection of light-colored planks still seems to be clearer when `SELU` is used.
For the multi-class segmentation task the simplified SegNet does not perform as well as before. However, the loss plot in fig. 18 suggests that further training might lead to better results in this case.

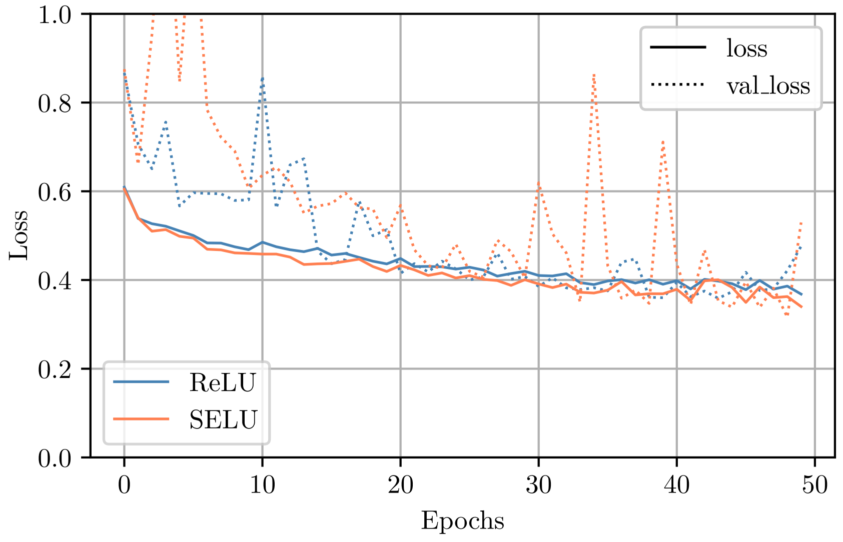

Figure 17: Loss of simplified SegNet for ReLU and SELU activation with training settings: Shift = False; Multi-Class = False, EarlyStopping = False; Epochs = 50; Image size = 256x256

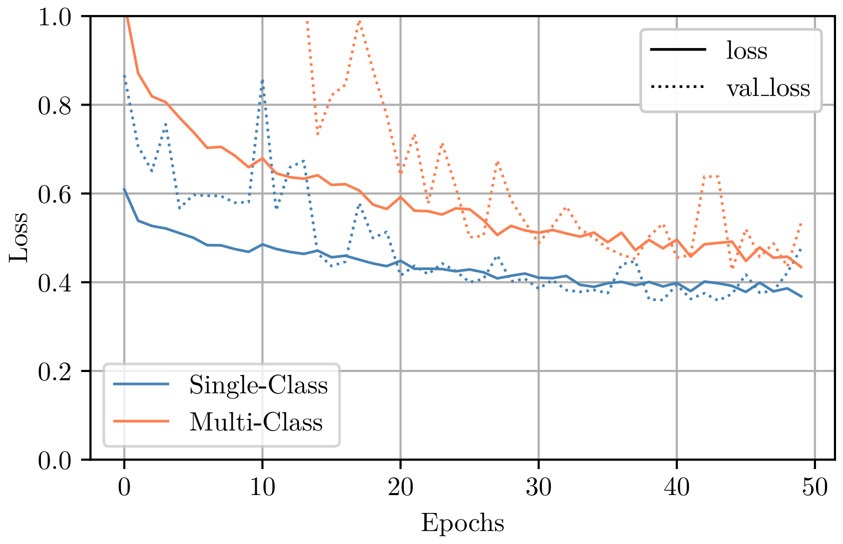

Figure 18: Loss of simplified SegNet for single- and multi-classification with training settings: Shift = False; Multi-Class = **X**, EarlyStopping = False; Epochs = 50; Image size = 256x256

5.3 U-Net

The results of the U-Net architecture are differ with a change in complexity regarding fig. 19 and fig. 22 for single-classification. As in listing 2 can be seen that the complexity is used as a measure for the applied layer packages in the U-Net.
Small complexities like a complexity of 2 doesn't seem to manage the training as well as higher complexities like 4 or 5. However higher complexities (complexity of 20 in fig. 19) also do not improve the training. So a suitable complexity had to be found. The best results were achieved while using a complexity of 5 as it can be seen in fig. 22 and in fig. 23. Therefore all the predictions (fig. 20 and fig. 21) are computed using U-Nets of a complexity of 5.

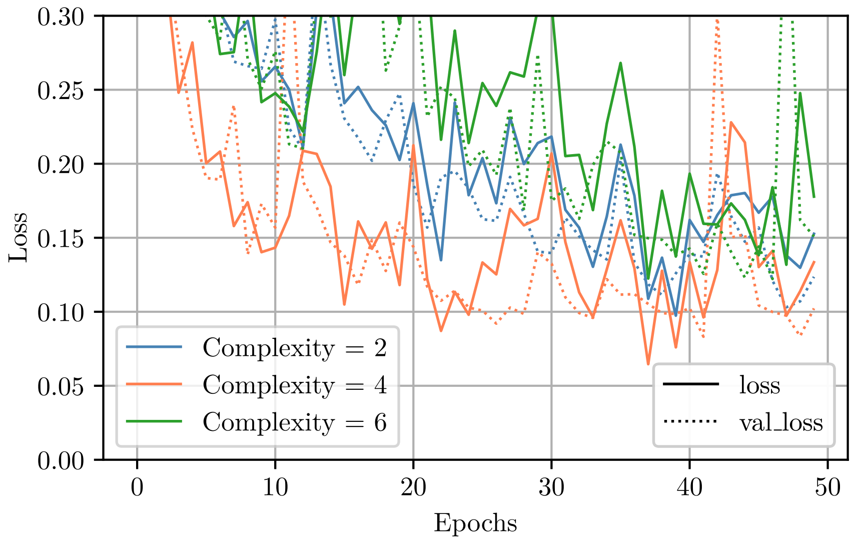

Figure 19: Losses of U-Net with different complexities using sliced images for single-classification with training settings: Shift = True; Multi-Class = False; EarlyStopping = False; Epochs = 50; Complexity = **X**; Image size = 256x256

Regarding the predictions in fig. 21 for multi-classification and in fig. 20 for single-classification, one can see a clear reproduction of the panel, even more precise than the actual mask for the single-classification.
Differences can be found when increasing the complexity. It leads to a visibly stronger consideration of other features in the image, so that already at a complexity of 6, the U-Net founds parts of the conveyor belt system in the background as panel. This may be caused by a strong dependence of the learning on white plates due to a ratio of 3:1 of white to black panels in the data. This may further influenced by the darker background, reinforcing problems with segmentation of the black panels.
In the multi-classification U-Net the predictions are more accurate when the

U-Net was trained on big-sized images compared to a U-Net trained on sliced images. The predictions of this second U-Net show a great discrepancy compared to the mask regarding the second and third example in fig. 21.

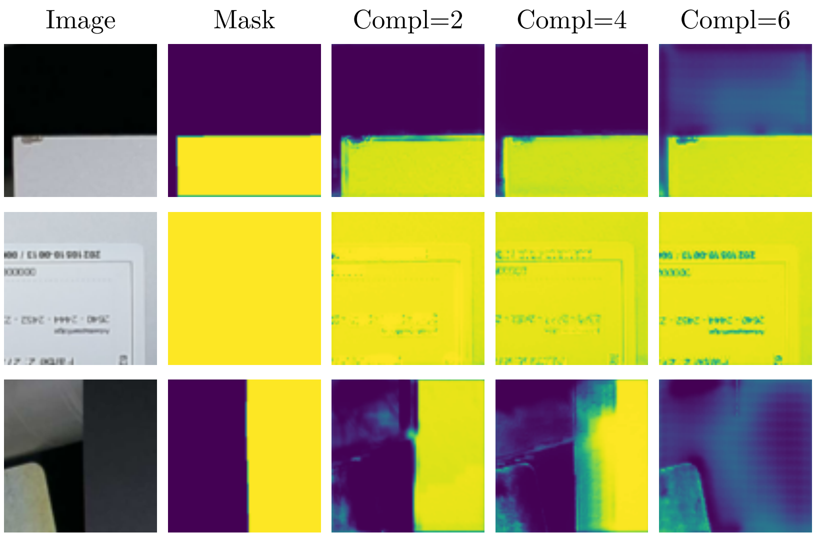

Figure 20: Predictions of U-Net with different complexities with training settings: Shift = True, Multi-Class = False; EarlyStopping = False; Epochs = 50; Complexity = **X**; Image size = 256x256

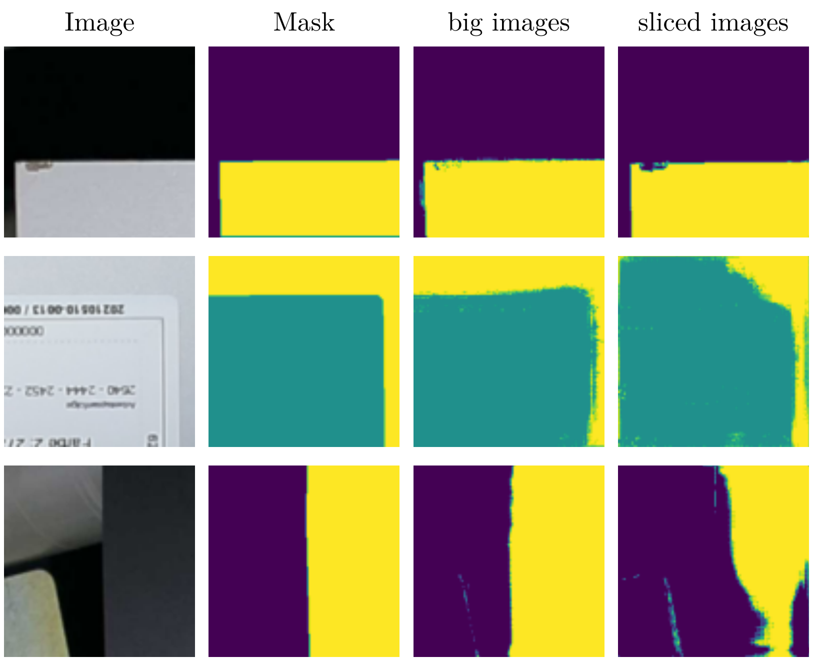

Figure 21: Predictions of U-Net with different sizes of images with training settings: Shift = False, Multi-Class = True; EarlyStopping = False; Epochs = 50; Complexity = 5; Image size = **X**

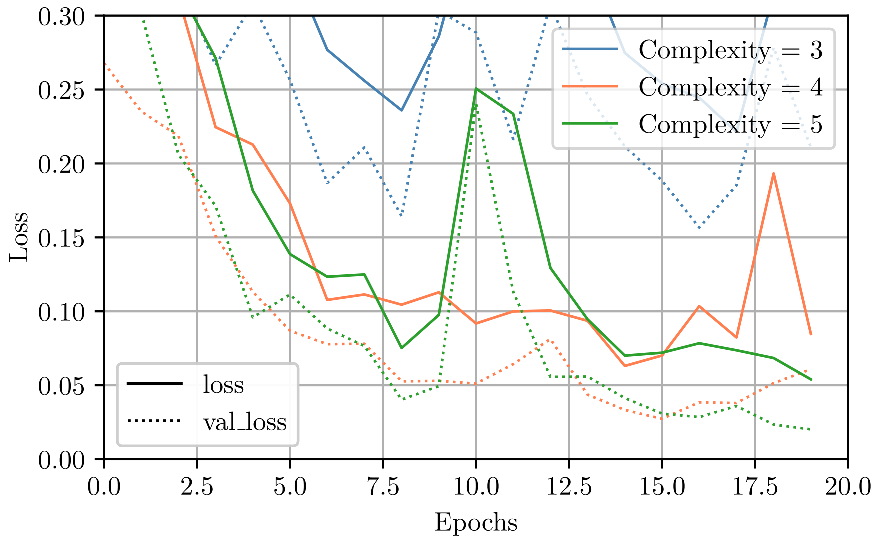

Figure 22: Losses of U-Net with different complexities using big images for single-classification with training settings: Shift = False; Multi-Class = False; EarlyStopping = False; Epochs = 50; Complexity = **X**; Image size = 256x3072

The results in the loss-curves for the single-classification U-Net in fig. 19 cover the conclusions drawn before from the predictions in fig. 15. A training of the U-Net on the big-sized images yields better results compared to a training on the sliced images.

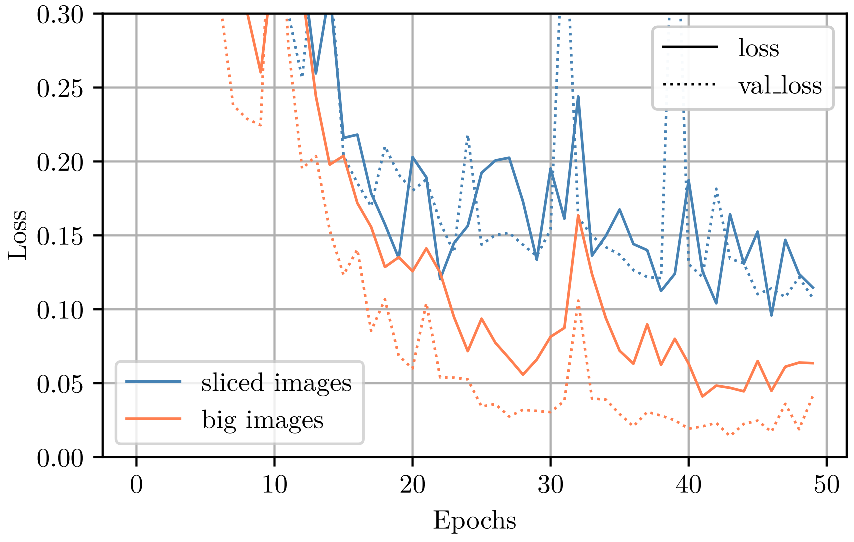

Figure 23: Loss of U-Net with complexity of 5 for multi-classification with training settings Shift = False; Multi-Class = True; EarlyStopping = False; Epochs = 50; Complexity = 5; Image size = **X**

5.4 Advanced Decoder

From the results for the advanced decoder architecture one can see in fig. 24 that the `BatchNormalization` doesn't lead to a big improvement for the learning for single- & multi-classification. In the case of multi-classification the loss-curves with the `BatchNormalization` lies even over the loss-curve without normalization.
However it must be considered that only one `BatchNormalization` was used (listing 7) and not in an reasonable position. This could disturb the result and so the conclusions. It was repaired in the advanced decoder architecture in listing 4 later on.

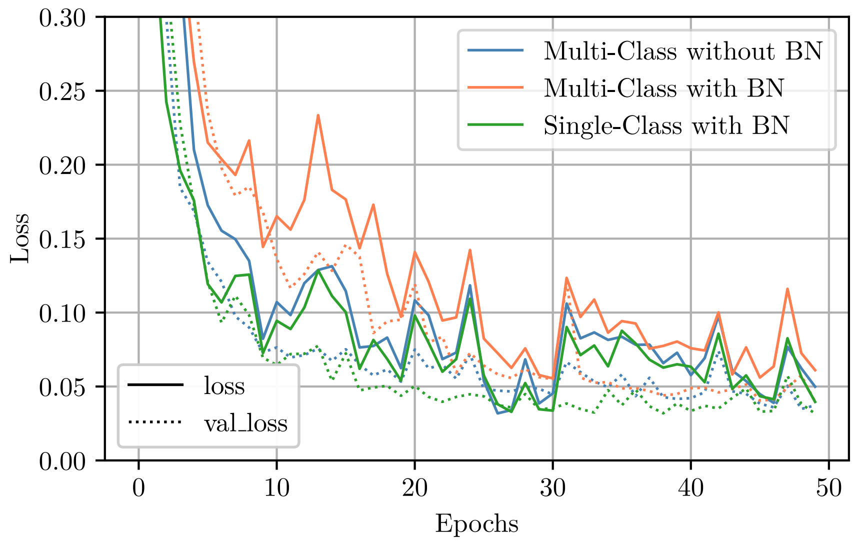

Figure 24: Loss for a decoder with Conv2DTranspose layers for multi- & single-classification with training settings: Shift = False; Multi-Class = **X**; EarlyStopping = False; Epochs = 50; Image size =256x256

For the four advanced decoder structures (listing 4) tested here, the losses are plotted in fig. 25 together with the predictions (fig. 26) for the multi-classification. All different decoder models perform in a similar.
In the loss-plot fig. 25 the decoder models reach a minimal loss of around 0.05 whereby the decoder with the transposed convolutional layer can be found as slightly better compare to the three other models. The predictions are nevertheless only roughly similar to the mask, the results are not usable for the segmentation purpose.
A similar conclusion can be found for the single-classification task while using these four decoder architectures. There, the losses in fig. 27 end up at around 0.1. As well as with the multi-classification the predictions for the single-classification in fig. 28 are not usable for the segmentation task. At the edges the predictions are still blurry and coarse and not sharp lines as it would needed to be. Regarding this point, the multi-classification yields clearer edges although these edges are not straight as required.

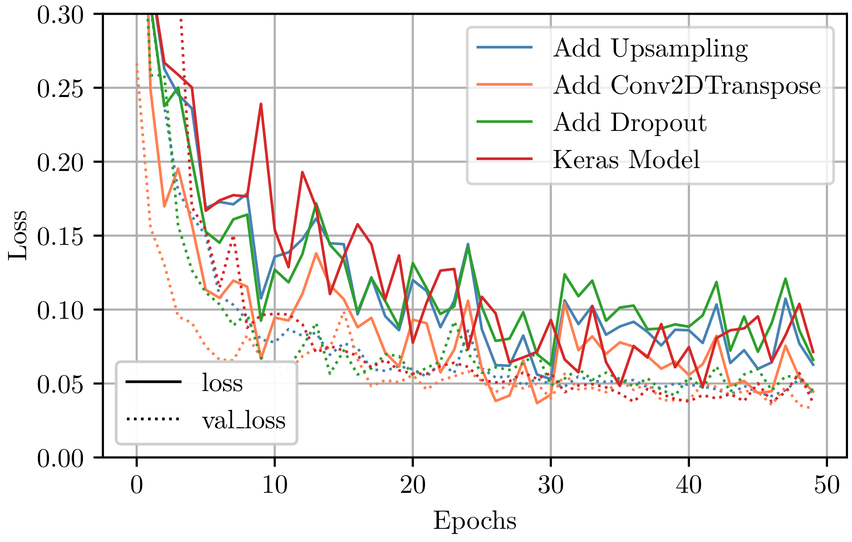

Figure 25: Loss for different decoder with added layers for multi-classification with training settings: Shift = False; Multi-Class = True; EarlyStopping = False; Epochs = 50; Image size =256x256

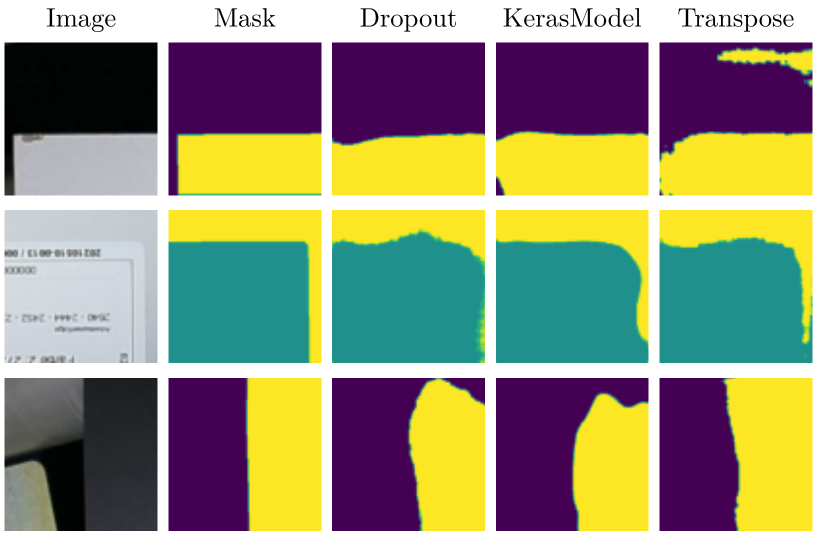

Figure 26: Predictions for different decoder with added layers for multi-classification with training settings: Shift = False; Multi-Class = True; EarlyStopping = False; Epochs = 50; Image size =256x256

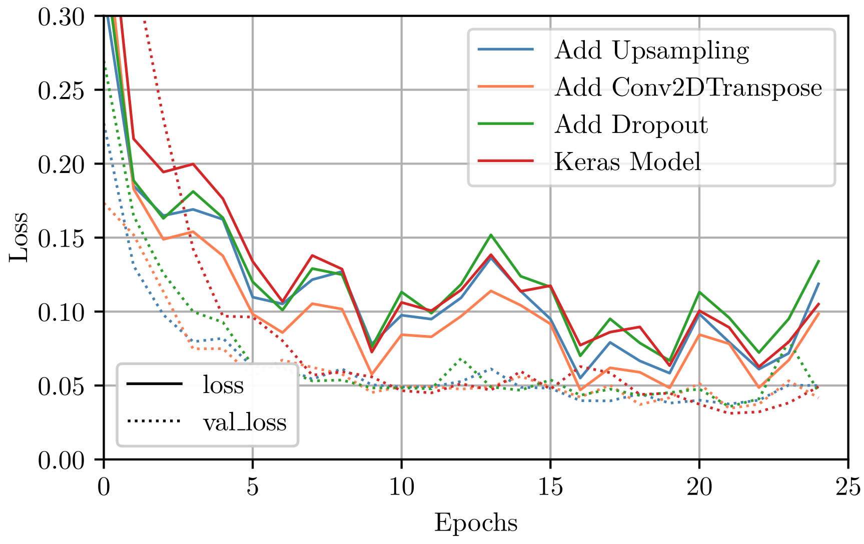

Figure 27: Loss for different decoder with added layers for single-classification with training settings: Shift = False; Multi-Class = False; EarlyStopping = False; Epochs = 25; Image size =256x256

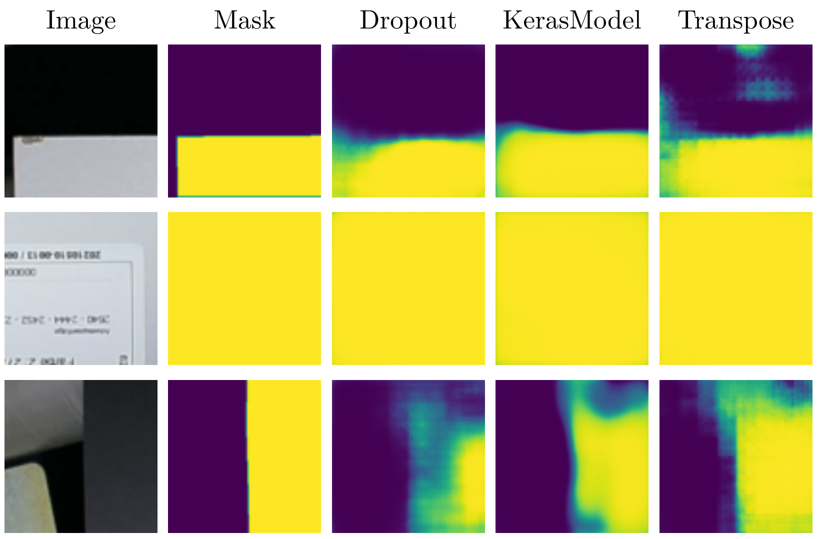

Figure 28: Predictions for different decoder with added layers for single-classification with training settings: Shift = False; Multi-Class = False; EarlyStopping = False; Epochs = 25; Image size =256x256

5.5 Benchmark studies on a Coral Chip

As final step, the segmentation nets are implemented on a coral chip which can later on be used with a raspberry pi 4 for real-time segmentation of wooden panels. In this report only the benchmark study are discussed.
Therefore the models containing `Keras` software based on the `tensorflow` package are compressed into `tensorflowlite`-models. For the thes 'lite' models only few layer-types are implemented and a nightly version of `tensorflowlite` had to be used.
The results are shown in fig. 29 for all pre-trained models, the U-Net and the SegNet. The results for different types of the U-Net are shown in fig. 30 .

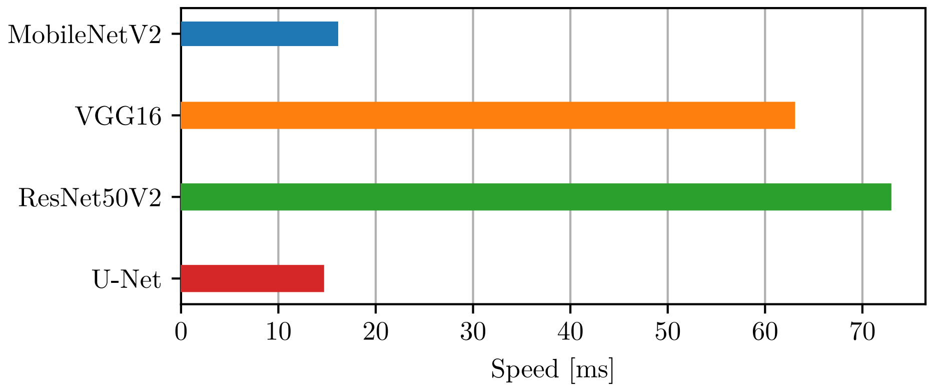

Figure 29: Benchmark study for the Pre-trained models, the U-Net and the SegNet

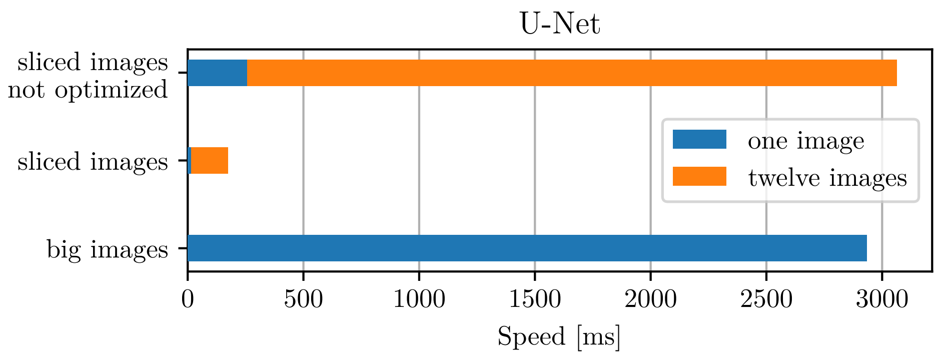

Figure 30: Benchmark study for different inputs while using a U-Net: Comparing a U-Net trained with not optimized, sliced images, a U-Net trained on optimized, sliced images and a U-Net trained on full-scale images (256x3072).

6 Outlook

For the further development, a lot of improvements can be realized. In the post-processing, now a threshold function could be applied to the results in order to detect the damages on the panel while controlling the damages size over a threshold value. Alternatively, another, small CNN could be used for the damage detection.
A second problem that appeared in the SegNet and the U-Net predictions, could be due to the different colors of the panels. This could be improved with more data of black panels and maybe with two CNN, each trained on only white or only black panels. Also a deeper SegNet architecture as used in the first proposal of the SegNet [1] could lead to better results. The noise in predictions of the SegNet could be suppressed using a post-processing functions.
For the advanced decoder structures also deeper structures and a longer training could yield better results. Nevertheless these decoder will not perform as good as advanced architectures like the SegNet and the U-Net can with the same training and data. The new findings of implementing these decoders can be used for improving the SegNet and the U-net architecture further as already done in the final model in this project.

References

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks, 2016.

[4] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *Artifical Intelligence Review*, 34:1–54, 2015.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation.

[6] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[7] Adnan Saood and Iyad Hatem. Covid-19 lung ct image segmentation using deep learning methods: U-net versus segnet. *BMC Medical Imaging*, 21(1), 2021.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[9] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net.

[10] Zhi Tian, Tong He, Chunhua Shen, and Youliang Yan. Decoders matter for semantic segmentation: Data-dependent decoding enables flexible feature aggregation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Appendix for Codes

```
def generate_model(img_size, complexity, multiclass = True):

    x = Input((img_size))
    inputs = x

    # Encoder
    f = 8
    layers = []

    for i in range(0, complexity):
        x = Conv2D(f, 3, activation='relu', padding='same')(x)
        x = Conv2D(f, 3, activation='relu', padding='same')(x)
        layers.append(x)
        x = MaxPooling2D()(x)
        f = f*2
    ff2 = 64

    # Bottleneck
    j = len(layers) - 1
    x = Conv2D(f, 3, activation='relu', padding='same')(x)
    x = Conv2D(f, 3, activation='relu', padding='same')(x)
    x = Conv2DTranspose(ff2, 2, strides=(2, 2),
                        padding='same')(x)
    x = Concatenate(axis=3)([x, layers[j]])
    j = j -1

    # Decoder
    for i in range(0, complexity-1):
        ff2 = ff2//2
        f = f // 2
        x = Conv2D(f, 3, activation='relu', padding='same')(x)
        x = Conv2D(f, 3, activation='relu', padding='same')(x)
        x = Conv2DTranspose(ff2, 2, strides=(2, 2),
                            padding='same')(x)
        x = Concatenate(axis=3)([x, layers[j]])
        j = j -1

    # Classification
    if multiclass:
        outputs = Conv2D(3, 1, activation='softmax')(x)
    else:
        outputs = Conv2D(1, 1, activation='sigmoid')(x)

    # Model creation
    model = Model(inputs=[inputs], outputs=[outputs])

    return model
```

Listing 2: Code for the generation of the U-Net

```
def create_segnet(img_size, num_filters, multiclass = True):

    inputs = Input(shape=img_size)

    # Encoder
    conv_1 = Convolution2D(num_filters, (3, 3), padding="same",
                                            kernel_initializer ='he_normal')(inputs)
    conv_1 = BatchNormalization()(conv_1)
    conv_1 = Activation("relu")(conv_1)
    pool_1, mask_1 = MaxPoolingWithIndices2D(pool_size=(2, 2))(conv_1)
    ### Add two equivalent blocks ###
    conv_4 = Convolution2D(4 * num_filters, (3, 3), padding="same",
        kernel_initializer='he_normal')(pool_3)
    conv_4 = BatchNormalization()(conv_4)
    conv_4 = Activation("relu")(conv_4)
    pool_4, mask_4 = MaxPoolingWithIndices2D(pool_size=(2, 2))(conv_4)

    # Decoder
    unpool_1 = MaxUnpoolingWithIndices2D(size=(2, 2))([pool_4, mask_4])
    conv_5 = Convolution2D(2 * num_filters, (3, 3), padding="same",
        kernel_initializer='he_normal')(unpool_1)
    conv_5 = BatchNormalization()(conv_5)
    conv_5 = Activation("relu")(conv_5)
    unpool_2 = MaxUnpoolingWithIndices2D(size=(2, 2))([conv_5, mask_3])
    ### Add two equivalent blocks ###
    conv_8 = Convolution2D(n_labels, (1, 1), padding="same",
        kernel_initializer='he_normal')(unpool_4)
    conv_8 = BatchNormalization()(conv_8)
    outputs = Activation(output_mode)(conv_8)

    # Classification
    if multiclass:
        outputs = Activation("softmax")(conv_8)
    else:
        outputs = Activation("sigmoid")(conv_8)

    # Model creation
    segnet = Model(inputs=inputs, outputs=outputs)

    return segnet
```

Listing 3: Code for the generation of the simplified SegNet

```
def generate_model(img_size, multiclass = True):
    # Encoder
    encoder = MobileNetV2(include_top=False, weights='imagenet',
                                        input_shape=(img_size),
                                         classifier_activation =None)
    encoder.trainable = False
    x = encoder.layers[-1].output

    # Decoder
    f = [16,32,64,128,256]
    d = [16,16,32,48,64]
    for i in range(len(f)):
        dUp = UpSampling2D(size=(2, 2))(x)
        dUp = Conv2D(d[-i], kernel_size=(3, 3),padding='same')(dUp)
        dUp = BatchNormalization()(dUp)
        dUp = Activation("selu")(dUp)
        x = Conv2DTranspose(f[i], (3, 3), strides=2, activation="selu",
             padding="same")(x)
        x = Conv2D(d[-i], kernel_size=(3, 3),padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation("selu")(x)
        x = Conv2D(d[-i], kernel_size=(3, 3),padding='same')(x)
        x = BatchNormalization()(x)
        x = Activation("selu")(x)
        x = add([x,dUp])
        x = Activation("selu")(x)

    # Classification
    if multiclass == True:
        c5 = Conv2D(3, kernel_size=(1, 1), activation='softmax',padding='same')(x)
    else:
        c5 = Conv2D(1, kernel_size=(1, 1), activation='sigmoid',padding='same')(x)
    output =c5

    # Model creation
    model = Model(inputs=encoder.inputs, outputs=output)

    return model
```

Listing 4: Code of the Advanced Decoder

```
def generate_model(img_size, multiclass = True):
    # Encoder
    encoder = tf.keras.applications.MobileNetV2(include_top=False,
                                    weights='imagenet', input_shape=(img_size),
                                     classifier_activation =None)
    encoder.trainable = False
    x = Conv2D(16, kernel_size=(3, 3), activation='selu',
                                    padding='same')(encoder.layers[-1].output)

    # Decoder
    f = [16,32,64,128,256]
    for i in range(len(f)):
        dUp = UpSampling2D(size=(2, 2))(x)
        x = Conv2DTranspose(16, (3, 3), strides=2, activation="selu",
             padding="same")(x)
        x = Conv2D(16, kernel_size=(3, 3), activation='selu', padding='same')(x)
        x = add([x,dUp])
        x = Activation("selu")(x)
        x = BatchNormalization()(x)
        x = Dropout(0.3)(x)
    d5 = Conv2DTranspose(256, (3, 3), strides=2, activation="selu", padding="same")(x)

    # Classification
    if multiclass == True:
        c5 = Conv2D(3, kernel_size=(1, 1), activation='softmax', padding='same')(d5)
    else:
        c5 = Conv2D(1, kernel_size=(1, 1), activation='sigmoid', padding='same')(d5)
    output =c5

    # Model creation
    model = Model(inputs=encoder.inputs, outputs=output)

    return model
```

Listing 5: Code of the Advanced Decoder with Dropout

```
def generate_model(img_size, multiclass = True):
    # Encoder
    encoder = tf.keras.applications.MobileNetV2(include_top=False,
                            weights='imagenet', input_shape=(img_size),
                             classifier_activation =None)
    encoder.trainable = False
    x = Conv2D(16, kernel_size=(3, 3), activation='selu',
                            padding='same')(encoder.layers[-1].output)

    # Decoder
    f = [16,32,64,128,256]
    for i in range(len(f)):
        dUp = UpSampling2D(size=(2, 2))(x)
        x = UpSampling2D(size=(2, 2))(x)
        x = Conv2D(16, kernel_size=(3, 3), activation='selu', padding='same')(x)
        x = add([x,dUp])
        x = Activation("selu")(x)
        x = BatchNormalization()(x)
        x = Dropout(0.3)(x)
    d5 = Conv2DTranspose(256, (3, 3), strides=2, activation="selu", padding="same")(x)

    # Classification
    if multiclass == True:
        c5 = Conv2D(3, kernel_size=(1, 1), activation='softmax', padding='same')(d5)
    else:
        c5 = Conv2D(1, kernel_size=(1, 1), activation='sigmoid', padding='same')(d5)
    output =c5

    # Model creation
    model = Model(inputs=encoder.inputs, outputs=output)

    return model
```

Listing 6: Code of the Advanced Decoder with Upsampling

```
def generate_model(img_size, multiclass = True):
    # Encoder
    encoder = tf.keras.applications.MobileNetV2(include_top=False,
                            weights='imagenet', input_shape=(img_size),
                             classifier_activation =None)
    encoder.trainable = False
    x = Conv2D(16, kernel_size=(3, 3), activation='selu',
                            padding='same')(encoder.layers[-1].output)

    # Decoder
    f = [16,32,64,128,256]
    for i in range(len(f)):
        dUp = UpSampling2D(size=(2, 2))(x)
        x = Conv2DTranspose(16, (3, 3), strides=2, activation="selu",
            padding="same")(x)
        x = Conv2D(16, kernel_size=(3, 3), activation='selu', padding='same')(x)
        x = add([x,dUp])
        x = Activation("selu")(x)
        x = BatchNormalization()(x)
    d5 = Conv2DTranspose(256, (3, 3), strides=2, activation="selu", padding="same")(x)

    # Classification
    if multiclass == True:
        c5 = Conv2D(3, kernel_size=(1, 1), activation='softmax', padding='same')(d5)
    else:
        c5 = Conv2D(1, kernel_size=(1, 1), activation='sigmoid', padding='same')(d5)
    output =c5

    # Model creation
    model = Model(inputs=encoder.inputs, outputs=output)

    return model
```

Listing 7: Code of the Advanced Decoder with Transposed layer

```
def generate_model(img_size, multiclass = True):
    inputs = Input(shape=(img_size), name="input_image")

    # Encoder
    encoder = MobileNetV2(include_top=False,
                          weights='imagenet',
                          input_tensor=inputs,
                           classifier_activation =None)
    encoder.trainable = False
    skip_connection_names = ["input_image",
                             "block_1_expand_relu",
                             "block_3_expand_relu",
                             "block_6_expand_relu"]
    encoder_output = encoder.get_layer("block_13_expand_relu").output
    x = encoder_output

    # Bottleneck
    f = 64
    ff2 = 8
    x = Conv2D(f, 3, activation='relu', padding='same') (x)
    x = Conv2D(f, 3, activation='relu', padding='same') (x)
    x = Conv2DTranspose(ff2, 2, strides=(2, 2), padding='same') (x)
    x_skip = encoder.get_layer(skip_connection_names[-1]).output
    x = Concatenate(axis=3)([x, x_skip])

    # Decoder
    for i in range(2, 5):
        ff2 = ff2 * 2
        f = f // 2
        x = Conv2D(f, 3, activation='relu', padding='same') (x)
        x = Conv2D(f, 3, activation='relu', padding='same') (x)
        x = Conv2DTranspose(ff2, 2, strides=(2, 2), padding='same') (x)
        x_skip = encoder.get_layer(skip_connection_names[-i]).output
        x = Concatenate(axis=3)([x, x_skip])

    # Classification
    if multiclass:
        outputs = Conv2D(3, 1, activation='softmax') (x)
    else:
        outputs = Conv2D(1, 1, activation='sigmoid') (x)

    # Model creation
    model = Model(inputs=[inputs], outputs=[outputs])

    return model
```

Listing 8: Code of the final combined version of U-Net with the MobileNetV2