Identification of Defective Manufacturing Products with a Convolutional Autoencoder

Benjamin János Garzó [LP69CO], Gergő Marcell Miklós [WE507Q], and Péter Herbai [RUS9IV]

Abstract— Defective items are inevitable in a mass production process and companies are investing a lot into the detection and handling of such products. With the help of Artificial Intelligence, these processes can be made cheaper and much more efficient. The aim of our project was to identify faulty products from the MVTecAD Anomaly Detection Database with Unsupervised Learning techniques. For this task we have trained a Convolutional Autoencoder with images of non-defective products and tested its performance on a dataset containing both non-defective and faulty products. Since the autoencoder only encountered correct products during training, we expected it to reconstruct images of normal products correctly, but not the images of defective ones. This would mean a higher reconstruction error for faulty products, thus making the identification of these products possible. We implemented our model in PyTorch and used MSE for our loss function. For comparison we also trained a deep convolutional autoencoder with SSIM loss function as well.

Index Terms—Anomaly detection, Convolutional Autoencoder, Defect Segmentation, Unsupervised Learning

1 Introduction

IN contrary to humans, machines have a hard time recognizing images which represent something anomalous. Supervised learning techniques are the most common approaches to such problems – where a neural network is trained on labeled data and then makes predictions based on the learned information. However, in the manufacturing industry there's often a deficit in labeled data that depicts faulty products – especially in such a large volume, which would be sufficient to properly train a supervised model. On the other hand, there are certain situations, where the huge number of items rolling down the coveyor belt makes the inspection by a human supervisor impossible. This directs a lot of attention to unsupervised learning methodologies, that could possibly overcome both of these problems.

In recent years deep convolutional autoencoders have been widely used for anomaly detection tasks on image data [1],[2],[3],[4]. After being trained on completely anomaly-free images, these models were able to localize anomalies on the test images. This was due to the fact that the autoencoder models were optimized to reconstruct anomaly-free data, so that when images depicting faulty objects were being passed to the network, the reconstruction error for these pictures were higher – especially at the pixel-wise locations of the anomalies. In our work we also implemented a simple convolutional autoencoder model wherewith we wanted to investigate the same function presented above.

As for data we used images from the MVTec dataset [5]. This dataset contained both normal and anomalous images in 16 categories. The images came in varying sizes and had different kinds of defects. For training and testing our model, we chose 2 of these categories – namely ‘Hazelnut’ and ‘Carpet’. Some examples for normal and faulty images with their corresponding segmentation masks are displayed in Figs. 1 and 2.

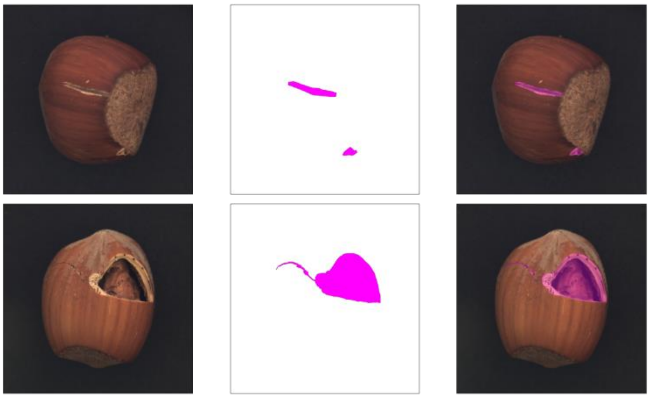


Fig. 1 Two examples for faulty images and their segmentation masks for the ‘Hazelnut’ imageset. The masks were applied by our team as part of the data visualization process.

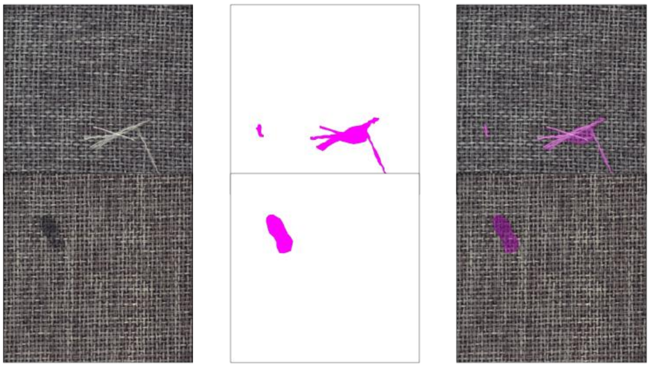


Fig 2. Two examples for faulty images and their segmentation masks for the ‘Carpet imageset. The masks were applied by our team as part of the data visualization process.

2 Literature Review

An interesting approach to anomaly detection and segmentation on the MVTec database using deep convolutional networks was made by [6]. They presented a self-supervised approach, which utilised discriminative information of anomalous patterns in the testing phase. This was based on an observation of how well abnormalities on an image could be reconstructed just from the context of their surrounding pixels, thus making the characterization of abnormalities of a region easier and more accurate. Even though the results obtained from this approach were fascinating, the implementation of such an algorithm would have taken a lot of effort. In other two approaches a different from regular loss functions – CW-SSIM and SSIM - were used to overcome the problem of anomaly detection with convolutional autoencoders [7], [8]. This simple change resulted in a significantly improved detection performance over the most used Mean Squared Error (MSE) loss function. By using (CW-)SSIM loss even a shallower and computationally less demanding network could compete with much deeper and larger networks utilizing MSE loss. These solutions were fundamentally much closer to what we were trying to – due to computational limitations we preferred smaller networks over more complex ones, despite their better performance.

3 Methodology

3.1 Data acquisition, loading and pre-processing

The two datasets - Hazelnut and Carpet - were directly downloaded from the MVTec website using the !wget command.

After obtaining the datasets we began exploring their contents. The train dataset only contained images of normal products belonging to 1 class, whereas the test dataset contained images of both normal and faulty products belonging to 5 different classes, based on the nature of the defects – as seen on Fig. 3.

As for loading and preprocessing our images we created a data loader function in PyTorch using the Generator and DataLoader classes. With the help of these, we loaded the images from the hard drive and after shuffling and resizing we organized them into training – validation – test sets. On our TensorFlow implementation we also applied some minor data augmentation as well, including vertical and horizontal flips and zooming. In the end we didn't use the carpet dataset for training, only the hazelnut.

3.2 Models

In our project, we have developed multiple models to best possibly grasp our task. In this section we'll be talking about these models. Each model will be detailed and later we'll compare them. All our models are basically convolutional autoencoders, their difference lies in the number of convolutional layers, dense layers, use of loss mechanisms, use of normalization, pooling etc.

3.2.1 Convolutional Autoencoder and Multi-loss

This model uses 3 convolutional layers and two dense layers and a few normalization layers also in the encoder and decoder. Handling of the latent dimension is fairly regular to all autoencoders, so we'll not go into detail regarding that.

We first tried this plain convolutional autoencoder model, then we tried to extend it by using multiple loss functions instead of the regular one.

These functions are usually some kind of performance measurements or error measurements, but we can use them while training our models, to help achieve better results. On one hand we use Mean Squared Error, which is a fairly regular loss function. At the beginning of our project, we just used MSE, but later we added a new loss function which is the SSIM (Structural Similarity). This function is commonly used in regards of autoencoders, GAN's and anything that works on generating images.

The model was a good start to our project, where we experienced the use of multiple loss functions and the inner workings of the Autoencoder architecture. However, we couldn't get satisfying results with this approach, so we decided to develop the one-loss solution further.

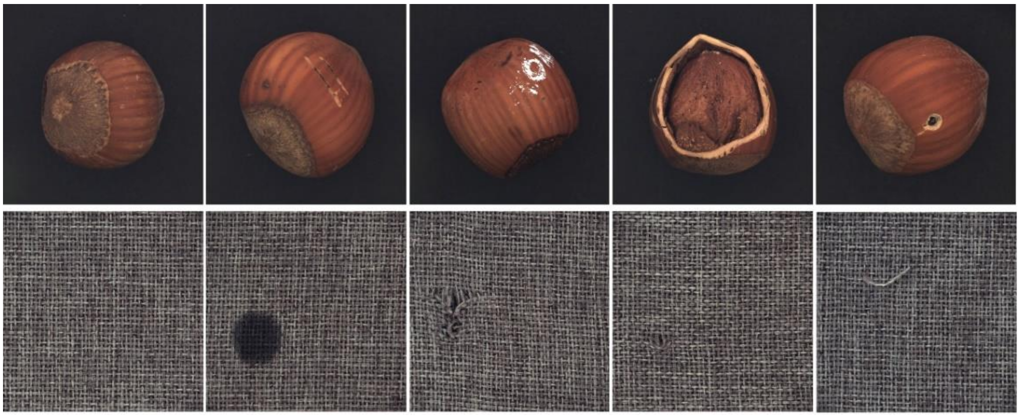


Fig 3. Images from the 'Hazelnut' and 'Carpet' categories belonging to 5 different classes based on the type of defects. In the top row the categories are as following: 'good', 'cut', 'print', 'crack' and 'hole'. As for the bottom row: 'good', 'color', 'cut', 'hole' and 'thread' respectively.

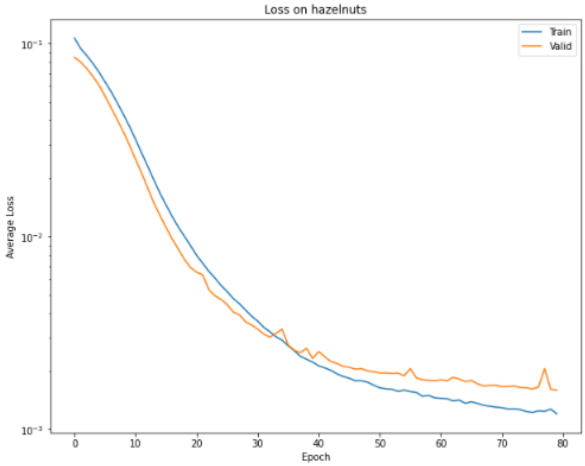


Fig 4. Loss plot for our Multiloss Autoencoder

3.2.2 Transfer learning with SSIM loss

Transfer learning is a powerful technique that allows us to use a pre-trained model and fine-tune it for our specific use case. In the case of a convolutional autoencoder, we can use a pre-trained model such as VGG-19 as the encoder part of the autoencoder [9]. The decoder part of the autoencoder can then be trained from scratch to learn to reconstruct the input images.

The VGG-19 model is a convolutional neural network (CNN) trained on the ImageNet dataset. It was developed by the Visual Geometry Group (VGG) at Oxford University and introduced in a 2014 paper titled "Very Deep Convolutional Networks for Large-Scale Image Recognition".

The VGG-19 model is known for its simplicity, using only 3x3 convolutional filters and a symmetrical architecture that consists of repeating blocks of convolutional and pooling layers. This simplicity makes the model relatively easy to train and understand, while still achieving very good performance on the ImageNet dataset.

One of the key features of the VGG-19 model is its depth. It has 19 layers, including 16 convolutional layers and 3 fully-connected layers. This depth allows the model to learn a hierarchical representation of the data, with the lower layers learning simple features such as edges and corners, and the higher layers learning more complex features such as object parts and objects as a whole.

Overall, the VGG-19 model is a powerful tool for image classification and recognition tasks, and has been widely used in a variety of applications.

In practice transfer learning means, that we take an already trained model and freeze its layers and use it up for our use case. Similar to transfer learning is fine-tuning, which differs in the sense that we further train the pre-trained model too.

Now we will go into detail about our implementation of the transfer learning autoencoder. To keep it simple we used MSE loss. In our research we tried out four different methods: VGG-19/VGG-16/ResNet as encoder and frozen layers, VGG-19 finetuned on our data. The first three implementations only differed from the number of layers of convolution we used with our decoder. The decoder was a simple model consisting of enough strided deconvolutional layers to recreate 32x32 dimensional images.

The first method (using pre-trained model as encoder) yielded the best results, but they we're still really far away from being great. Later on, we'll share some data about the best accuracy we could come up with using transfer learning. The loss plots for this implementation can be seen below on Fig. 5.

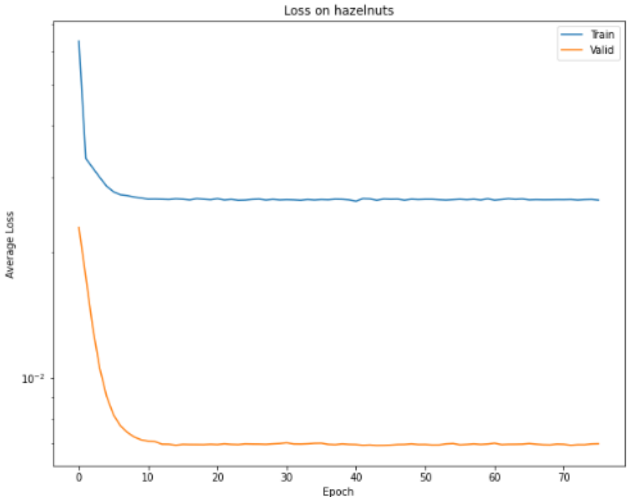


Fig 5. Loss plot for our Autoencoder using transfer learning

The second method (finetuning) couldn't generate any comprehensible images. Trained only on the hazelnut database, it couldn't produce images that could resemble the training data.

A general concern we ran into while implementing the upper mentioned solutions was that our model stopped training relatively early on, we have tried numerous parameters for learning rate, batch size, optimizers, schedulers, other architectures, but we couldn't solve the issue. As we can see on the image above our loss values were fairly unconventional. In the end we couldn't produce better results than our simple convolutional autoencoder model. The model's accuracy was a disappointing 50%, with reconstructed images as such in Fig. 6.

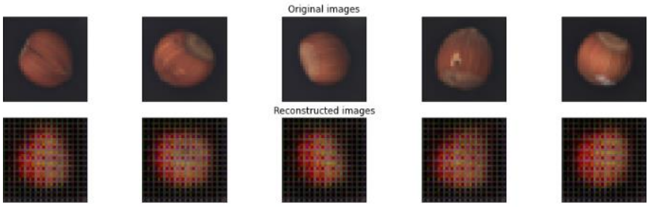


Fig 6. Predicted images for our model using transfer learning

3.2.3 Convolutional Autoencoder with SSIM loss implemented in TensorFlow

One of our team members only had experience with TensorFlow and getting to know PyTorch among other things seemed to be less of a great idea, so we decided to implement a different model using TensorFlow. This model had some minor to major changes compared to the PyTorch versions as discussed below.

This network consisted of only Convolutional, Max-

Pooling and UpSampling layers, and lacked the Dense and Conv2DTranspose layers as these were present in the PyTorch version [10],[11],[12]. This model also had a deeper architecture on both the encoder and decoder parts with more layers as well as larger filter sizes resulting in “more dimensions”.

We also defined our own loss function – a multiscale SSIM variant, which could operate with RGB images, hence the original SSIM loss was meant to be used on Grayscale images only.

Dataset generation and data augmentation was done in one step, using TensorFlow’s ImageDataGenerator function [13]. We applied vertical and horizontal flipping as well as zooming on the train and validation images after normalizing them to the [0,1] range. Before testing we normalized our test dataset as well using the previously mentioned data-generator function.

3.3 Training

Training our models was fairly straightforward using PyTorch. First, we defined an optimizer - our choice landed on the Adam optimizer [14]. The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. We also defined a scheduler to train our models more efficiently, we used a Pytorch defined scheduler the StepLR. It basically decays the learning rate of each parameter group by gamma every number of epochs which we give as argument.

As mentioned earlier in the descriptions of our models, we used two methods of measurement to help our models train better, but we use these same functions to judge their performance.

Our first method was the MSE, which translates to Mean Squared Error. We basically take the “distance” of the original image and our created image (dividing them) and square it. This method is widely used for logistic regression, statistics, etc. [15].

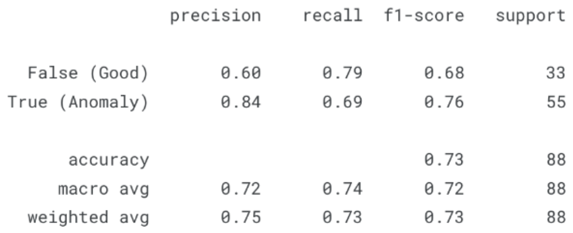
Our second method of choice was the SSIM, which translates to Structural Similarity. It basically aims to address the shortcomings of MSE by taking the texture of the image into account. The SSIM formula The SSIM formula is based on three comparison measurements between the samples of x and y: luminance, contrast and structure [16].

Further describing our training methods, we defined the main training loop in a way to show us the most important information about the model. This includes all the used and calculated training and validation losses, while also at the end of each epoch we illustrate the output of our model.

3.4 Evaluation and Testing

One of our best performing model was developed from the multi-loss convolutional autoencoder architecture (that turned into single loss with MSE), we have wrote about previously. This model was trained to reconstruct error-free images with its encoder-decoder CNN architecture, and we expected that it would also generate error-free images from inputs that contain errors during evaluation. If the input and generated image differ in some parts, then we can say that the original image contains some anomaly. We measured this (squared) error pixel by pixel, and we introduced a minimum threshold to decide whether errors are real anomalies or not. The value of this threshold was calculated based on q-th percentile of the pixels’ squared errors between the training dataset’s inputs and outputs. This method also made it possible to visualize the output of our model in a meaningful way.

The test dataset contained 33 good and 55 faulty images, and we used the accuracy, precision, recall, and f1-score classification metrics to measure the performance of our model. Based on these, we executed several hyperparameter tuning manually to find the optimal number and size layers, learning rate, number of epochs, and threshold percentile. We managed to reach a 73% accuracy with our most simple Convolutional Autoencoder with one loss (See Fig. 7 and 8.).



```
                  precision    recall  f1-score   support

    False (Good)       0.60      0.79      0.68        33
  True (Anomaly)       0.84      0.69      0.76        55

        accuracy                           0.73        88
       macro avg       0.72      0.74      0.72        88
    weighted avg       0.75      0.73      0.73        88
```

Fig. 7. Performance of our simple autoencoder model

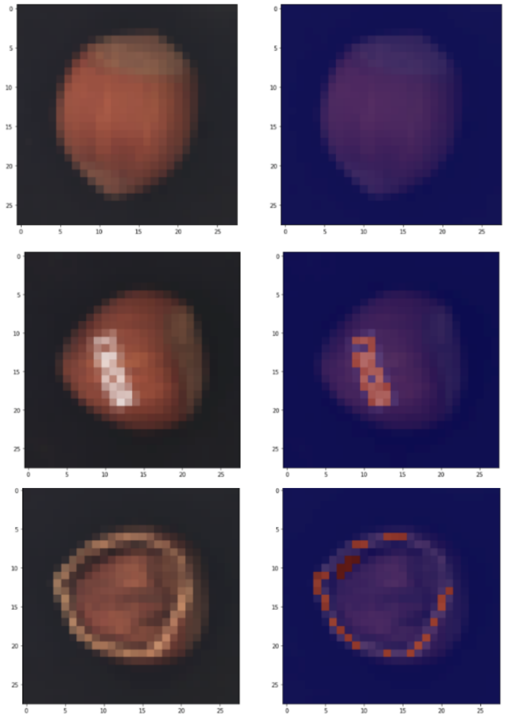


Fig. 8. Our models performance on the test set

3.6 Performance of our TensorFlow SSIM Autoencoder

The TensorFlow implementation served as a reference for our other models with less layers.

We started multiple trainings and our best model terminated after 509 epochs (with the use of Early Stopping) [17]. The results for this model can be seen in Table 1. and on Fig. 9 and 10.)

loss	0.0429
mse	0.000789
val_loss	0.0402
val_mse	0.000744

Table 1. Metrics for our TensorFlow implementation using SSIM loss and MSE error functions.

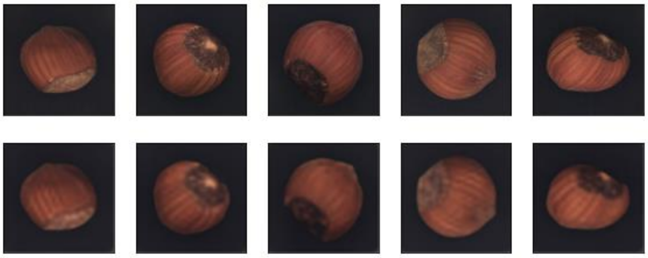


Fig. 9. Our model's predictions on the validation dataset – after training 509 epochs. (Upper row – original images of the validation dataset, Bottom row – reconstructed images). The model could reconstruct the images with great visible accuracy, even with the horizontal and vertical flipping applied to the images beforehand.

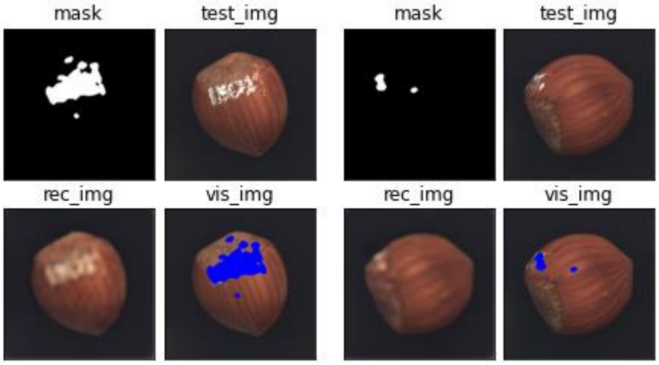


Fig. 10. – Our model's performance on the test set. (mask – segmentation mask, test_img – original test image fed into the network, rec_img – reconstructed image given as an output of the network, vis_img – mask and test image blended into one picture). These are one of our best results – in other cases the model identified brighter regions as anomalies (this could also be seen in some parts of this Figure as well.)

The model produced promising results, however, further modifications and fine-tuning are required to improve it's performance.

4 Conclusion

In conclusion, our study showed that the simplest CNN autoencoder model was able to effectively detect anomalies in product images with acceptable accuracy, while the multi-loss variant and the transfer-learning-based model performed poorly. This suggests that more complex models may not be necessary for this task and could potentially even lead to overfitting. Overall, our findings indicate that the use of simple CNN autoencoder models in image-based anomaly detection has great potential for ensuring the quality and integrity of products in various industries.

References

[1] M. Haselmann, D. P. Gruber, and P. Tabatabai, "Anomaly detection using deep learning based image completion," in 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, M. A. Wani, M. M. Kantardzic, M. S. Mouchaweh, J. Gama, and E. Lughofer, Eds. IEEE, 2018, pp. 1237–1242

[2] C. Li, K. Sohn, J. Yoon, and T. Pfister, "Cutpaste: Self-supervised learning for anomaly detection and localization," in IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021. Computer Vision Foundation / IEEE, 2021, pp. 9664–9674.

[3] J. Pirnay and K. Chai, "Inpainting transformer for anomaly detection," in Image Analysis and Processing - ICIAP 2022 - 21st International Conference, Lecce, Italy, May 23-27, 2022, Proceedings, Part II, ser. Lecture Notes in Computer Science, S. Sclaroff, C. Distante, M. Leo, G. M. Farinella, and F. Tombari, Eds., vol. 13232. Springer, 2022, pp. 394–406.

[4] Zhou, C., Paffenroth, R.C.: Anomaly detection with robust deep autoencoders. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 665–674 (2017)

[5] P. Bergmann, K. Batzner, M. Fauser, D. Sattlegger, and C. Steger, "The MVTEC Anomaly Detection Dataset: A comprehensive real-world dataset for unsupervised anomaly detection," International Journal of Computer Vision, vol. 129, no. 4, pp. 1038–1059, 2021.

[6] A. Bauer, "Self-Supervised Training with Autoencoders for Visual Anomaly Detection," 23-Jun-2022. [Online]. Available: https://arxiv.org/abs/2206.11723. [Accessed: 08-Dec-2022].)

[7] Bionda, A., Frittoli, L., Boracchi, G. (2022). Deep Autoencoders for Anomaly Detection in Textured Images Using CW-SSIM. In: Sclaroff, S., Distante, C., Leo, M., Farinella, G.M., Tombari, F. (eds) Image Analysis and Processing – ICIAP 2022. ICIAP 2022. Lecture Notes in Computer Science, vol 13232. Springer, Cham. https://doi.org/10.1007/978-3-031-06430-2_56

[8] P. Bergmann, S. Löwe, M. Fauser, D. Sattlegger, and C. Steger, "Improving unsupervised defect segmentation by applying structural similarity to autoencoders," Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, 2019.

[9] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2015, https://arxiv.org/abs/1409.1556

[10] "Conv2D," TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D. [Accessed: 11-Dec-2022].

[11] "MaxPool2D," TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D. [Accessed: 11-Dec-2022].

[12] "UpSampling2D," TensorFlow. [Online]. Available:

https://www.tensorflow.org/api_docs/python/tf/keras/layers/UpSampling2D. [Accessed: 11-Dec-2022].

[13] “ImageDataGenerator,” TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator. [Accessed: 11-Dec-2022].

[14] D. P. Kingma, J. Ba, “Adam: A Method for Stochastic Optimization,”3rd International Conference for Learning Representations, San Diego, 2015. https://arxiv.org/abs/1412.6980

[15] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures," in IEEE Signal Processing Magazine, vol. 26, no. 1, pp. 98-117, Jan. 2009, doi: 10.1109/MSP.2008.930649.

[16] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

[17] “EarlyStopping,” TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping. [Accessed: 11-Dec-2022].