Master Thesis

Residual Neural Networks for Medical Image Processing: A Case Study in Liver Cancer Diagnostics

Submitted to the Faculty of Mathematics
at the University of Duisburg-Essen

by

Oleh Bakumenko
Matriculation number: 3058575

to fulfil the requirements of the

Master of Science (M.Sc.)

Supervisors:	Prof. Dr. Johannes Kraus
	Prof. Dr. Jens Kleesiek
Degree:	Mathematics
Date:	06.06.2023

Contents

1 Introduction and abstract

In terms of the current state of art, machine learning models have achieved prominent results in the optimization of life processes and helping people in a wide variety of industries and applications. Some of the examples are speech recognition in computer vision, natural language processing, forecasting in finance and economics, machine vision for self-driving cars, and other forms of autonomous transportation, and last but not least image recognition and context analysis.

Medical machine learning is the application of machine learning techniques and algorithms to healthcare data, such as electronic health records, medical images, and genomic data. The goal is to use these data to improve medical decision making, automate repetitive tasks, and develop new diagnostic and treatment methods. Examples of medical machine learning applications include predictive modeling of patient outcomes, image analysis for diagnostic imaging, and natural language processing to extract information from electronic health records.

This master's thesis was created as an extension of a seminar «Machine Learning in Medicine» by Prof. Dr. Johannes Kraus at the Faculty of Mathematics of the University of Duisburg-Essen in cooperation with Prof. Dr. Jens Kleesiek from the Institute for Artificial Intelligence in Medicine (IAIM) [6]. During the seminar, important deep learning models from the past years were implemented, leading to the main goal of this thesis: to dive into medical machine learning algorithms in liver cancer diagnostics, compare performance, and create space for other experiments.

In the beginning, the medical aspects will be discussed, including what a liver tumor is, its causes, and its diagnostics. Then, the formation and structure of the dataset will be examined. The architectures of neural networks and their implementation will be explained in the fourth section. In this section, convergence analysis and mathematical optimization will also be mentioned. Lastly, the network performance on the dataset will be demonstrated in the remaining three sections. The results of the classification and segmentation models will be illustrated, and a summary of the thesis will be provided in the conclusion.

2 Medical information and problem

In the research context, it is important to have a thorough understanding of the data being worked with, including the nature of the malignancy and the process involved in creating the image. Medical information in this section was obtained from the cited Internet resources [8] and [9], as it is not the main focus of the thesis.

2.1 What is liver and liver tumor

Human liver is an organ and a gland in the human body. It is lobed in texture, shaped like a wedge, and typically appears reddish-brown in color. It performs numerous vital functions for survival and also acts as a gland by producing proteins and hormones needed by other parts of the body. On average, it weighs about 1.5 kilograms in adults and is the largest internal organ in the body. It is usually located on the right side of the body beneath the ribs.

The liver performs a wide range of vital functions, including the following.

- Detoxifying the blood by removing toxins and harmful substances
- Discarding old red blood cells
- Secretion of bile, a fluid that aids in digestion
- Metabolizing proteins, carbohydrates, and fats to make them usable by the body

Liver cancer, also known as hepatic cancer, is a relatively rare form of cancer, but is a highly aggressive and often fatal disease. There are two main types of liver cancer: primary liver cancer, which starts in the liver, and secondary liver cancer, which starts in another part of the body and spreads to the liver. The most common type of primary liver cancer is hepatocellular carcinoma (HCC).

Risk factors for liver cancer include chronic infection with hepatitis B or C, cirrhosis (scarring of the liver), and certain inherited conditions such as hemochromatosis (an excess of iron in the body) and Wilson's disease (a disorder that affects the way the body processes copper). Other risk factors include high alcohol consumption, exposure to certain chemicals and toxins, and diabetes.

Symptoms of liver cancer can include abdominal pain, weight loss, and jaundice (yellowing of the skin and eyes). Treatment options for liver cancer include surgery, radiation therapy, and chemotherapy. The best treatment option depends on the stage and location of the cancer, as well as the overall health of the patient. In some cases, a liver transplant may be an option.

Early detection and prevention are the key to improving the survival rate of liver cancer. Regular check-ups, screenings, and vaccinations against Hepatitis B can help in the early detection and prevention of liver cancer.

2.2 How liver tumor is being studied and diagnosed

Liver cancer is generally diagnosed by a combination of imaging tests and biopsy. Some of the common diagnostic methods for liver cancer include:

- Imaging tests: Ultrasound, computed tomography (CT) scan, and magnetic resonance imaging (MRI) can be used to create images of the liver and detect abnormal growths or tumors. CT scans often involve the use of a radiocontrast agent to enhance visibility. Also known as contrast medium, it is an iodine-based substance that is administered to the patient prior to scan, absorbs the X-rays, and appears white on the images.

- Blood tests: Elevated levels of certain liver enzymes or proteins in the blood can indicate the presence of liver cancer. For example, [2] provides an example of such tests.

- Tumor markers: Tumor markers are substances that are produced by cancer cells and can be found in blood. These markers can be used to help diagnose and monitor liver cancer.

- Biopsy: Typically, imaging procedures are performed initially and a biopsy is performed only if there are therapeutic implications and if the radiological criteria are incorrect or imprecise. A biopsy involves the extraction of a small tissue sample from the liver that is later examined under a microscope to detect the presence of cancer cells.

Once the diagnosis is confirmed, the doctors determine the stage of the cancer, which severely influences treatment decisions. American Cancer Society, [10], defines stages as numbers 0 to IV, with higher numbers indicating more advanced and severe cancer. It is important to note that these tests are used not only for diagnosis, but also to monitor the progression of the disease, response to therapy, and to detect recurrences.

2.3 Medical imaging and preprocessing

For the purpose of our discussion, the focus will be on computed tomography (CT) scans. This is a medical imaging technique that uses X-rays to create detailed cross-sectional images (also called *slices*) of the body. It is often used to diagnose conditions such as tumors, bone fractures, and internal injuries. CT scans are quick, painless, and non-invasive. The patient is positioned on a table that slides into a large machine called a gantry, which contains the X-ray source and detector. During the scan, the patient is asked to lie still while the X-ray source and detector rotate around the body.

CT scans provide an enormous amount of information as they allow for the analysis of three anatomical planes. However, for the purpose of this thesis, the focus will be on transverse scans that are parallel to the ground plane. An output of this procedure is a 3D array, with dimensions (Number of cross-sections $\times$ Encoding $\times$ Image size). The number of slices depends widely on which part of the body one wants to study.

Images are often coded in the RGB palette, which means that each pixel is represented by three values, one for each color channel (red, green, and blue). Alternatively, pixels can be coded using different shades of black, with 0 representing white and 1 representing black.

It is common for a CT scan to have a "bandwidth" of 1 millimeter, such that a scan of the body could conclude with more than 5000 slices for each patient. Assuming that each cross-sectional photo has a 256×256 pixel size, the dimensions of the output array would be $(5000 \times 3 \times 256 \times 256)$.

However, analyzing such a large array of data can be a challenge for humans. To identify subtle variations in density, such as areas that may indicate potential malignancy, each individual slice must be examined.

In our consideration, the array mentioned above will be saved as a number of cross-sections times separate images with dimensions (Encoding $\times$ Height $\times$ Width). In this context, the data from one patient, which is dependent, will be separated and combined with other images.

With the goal of optimizing the examination, two types of deep neural networks are introduced: classification and segmentation. Both networks take an image as input, with the classification network focusing on identifying specific characteristics that are inherent within the image, such as, for example, the presence of a liver. These characteristics are known as targets or classes. The segmentation network, on the other hand, outputs an image with an approximation of an object's location within the original image, with an emphasis on where the object's boundaries are located. In summary, the classification model is used to identify whether the object is present in the image, while the segmentation model deals with the location of the object.

Certainly, no model can ever replace a human doctor when it comes to making decisions for a patient who is ill or who has cancer. However, a trained model can help optimize the routine task of examining data and alerting a human to double check if something appears unusual. Ultimately, the model approximates the probabilities rather than providing a definitive diagnosis.

3 Dataset and augmentations

The first step in processing and analyzing medical images is, of course, the source of the images. In this thesis, the LiTS (Liver Tumor Segmentation) [1] data set was used. The reorganized and refined version of the dataset, so called Clean_LiTS, which includes the class and constructor, was provided by IAIM [6] for future applications in Section 5.

The dataset is separated into 3 sections: train, validate, and test sets. Each category has a folder with images and the corresponding targets for both types of models. For a classification model, the dataset class returns the image and its target when called. Possible classification targets are:

0 : Image does not include liver
1 : Liver is visible
2 : Liver is visible and lesion is visible

The total sum of images in the dataset is 41561, more precise numbers for each subset and each target class can be found in Table 1. As the table illustrates, some of the classes have an imbalanced number of samples, with the majority of images belonging to the first class. Despite this, the samples are evenly distributed within the target classes of train, validate, and test sets. To address the issue of class imbalance during training, weighted error functions will be used.

In terms of segmentation, all images also have a human-made mask with two layers: the liver and tumor boundaries. The model will be trained to predict 3 segments of the input image: background, liver, and tumor. Furthermore, it is vital to understand that the prediction of the tumor will overlap the prediction of the liver and be within its boundary.

The following paragraph analyzes several instances from the dataset. Figure 1 illustrates images from each of the 3 target classes for the classification problem. In all three images, the spinal bones and ribs can be observed surrounding the outer circle and in the lower middle section.

	No. images	Percentage	No. images targets		
			Target	No. images	Percentage
Train	35484	85.38	0	24274	68.41
			1	7085	19.97
			2	4125	11.62
Validate	3039	7.31	0	2087	68.70
			1	490	16.13
			2	461	15.17
Test	3038	7.30	0	1838	60.50
			1	837	27.55
			2	363	11.95

Table 1: Dataset.

The image on the left shows the lungs and heart, indicating that the slice is too high up in the body to capture the liver.

In the middle image, only a portion of the liver is visible, specifically the larger lobe, while the middle section displays segments of the gastrointestinal tract, spleen, and kidneys.

The final image shows a significant discolored spot at the top of the liver, which can serve as a sign of changes in the structure and density of the organ.

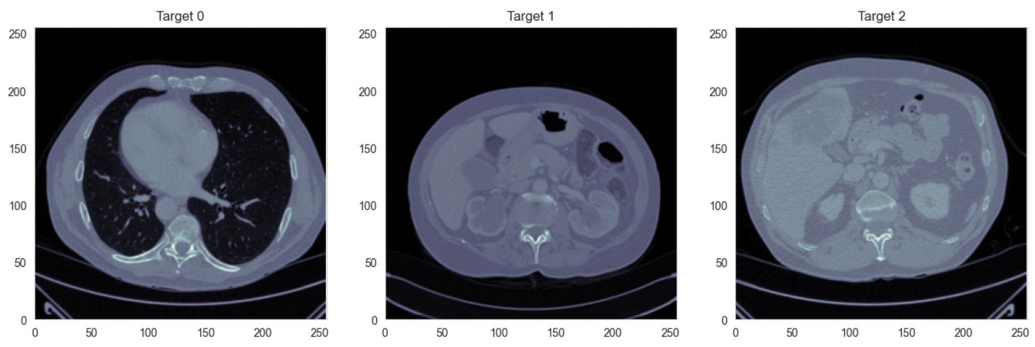


Figure 1: Example classification.

Figure 2 illustrates the original image and its two targets for the segmentation problem.

3.1 Augmentations

During the experiments conducted in Section 5, the images may be displayed in a mirrored manner to simulate real-life medical scenarios, where the liver

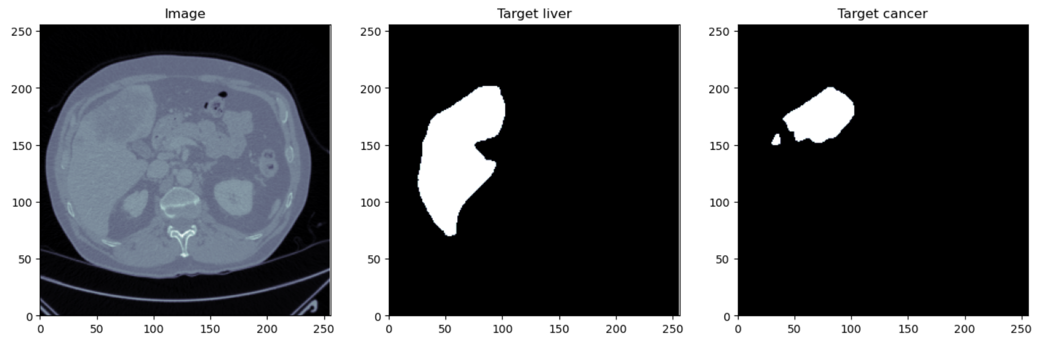


Figure 2: Example segmentation.

appears on the right side. Moreover, there exists a rare condition called "Situs inversus," in which a patient is born with their internal organs mirrored from their normal positions.

This fact does not play an important role in classification and segmentation problems because the focus is on the location of the liver and not its position. In fact, the model will learn to recognize an invariant alternative and the shape of the liver, regardless of its location in the image.

The process of artificially changing the input image to increase the size of the dataset or to make the model invariant to certain changes in the input, more robust, and less prone to overfitting is called augmentation. This technique is well known and is used in most application tasks. Overfitting is a phenomenon that occurs when a machine learning model becomes too complex and is able to fit the training data too well but performs poorly on new and unseen data. In other words, it learns the local features of the sample data and not the general pattern of the process, which is at the end the point of interest.

Commonly used data augmentation techniques include image flipping, rotation, zooming, changing the brightness, contrast, and more. The augmentation chosen for each model will be listed in the following sections.

4 Neural networks and architectures

Neural networks are machine learning algorithms that mimic the operations of the human brain. They are used to identify patterns in the data, make predictions, and perform tasks that include classification and regression. The networks are made up of interconnected artificial neurons that receive and process information before passing it on. The connections between these neurons have assigned weights that control the strength of the transmitted signal. During training, the weights are modified to allow the network to perform the task accurately. This process of weight adjustment is known as backpropagation and involves minimizing a loss function that calculates the discrepancy between the network's predictions and actual targets. Throughout this section, the main source of information is [Section 6 in 4].

4.1 Preliminaries

A feed-forward neural network is a type of artificial neural network in which the information flows only in one direction, from input to output. The main goal is to approximate some function $y = f(x)$ for the input vector x and the class y for the classification problem. In case of a segmentation problem, classification takes place for each pixel, such that input x and output y images have the same dimensions and each pixel of y will be classified.

Neuron. The basic building block is called a neuron. It consists of a linear transformation of the input x and a non-linear *activation function* $\sigma()$.

$$f = \sigma(W * x + b) \tag{4.1}$$

Network. When multiple neurons are arranged in a layer, input, output, and bias can be combined to a vector, and the weights are transformed into a matrix. The optimization process involves determining the weight and bias values, known as parameters, that lead to the best approximation of the function f. Composing a network from multiple layers is achieved by using an output of layer $i - 1$ as input for the ith layer.

Artificial Neural Networks have three distinct layer types: input layer, hidden layers, and output layer. The input layer f_1 receives the input x and forwards it to the hidden layer for processing. The hidden layers $f_2, ..., f_{n-1}$ undergo multiple transformations to arrive at the final output, which can be considered as the learning process.

A network with $n-1$ hidden layers could be represented in this manner:

$$\begin{aligned} f_1 =& \sigma(W_0 * x + b_0) \\ f_2 =& \sigma(W_1 * f_1 + b_1) \\ &\dots \\ f_n =& \sigma(W_{n-1} * f_{n-1} + b_{n-1}) \\ \hat{y} =& W_n * f_n \end{aligned}$$

The output layer calculates a value $\hat{y}$, with the aim to approximate the true value y. Its definition may vary depending on the problem at hand. In some cases, an affine shift is used to adjust the output, while in other classification problems, the softmax function

$$\hat{y} = \text{softmax}(x), \text{ where } \hat{y_i} = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{4.2}$$

acts as the output layer. The highest value among the coordinates of x indicates the predicted class with the highest probability according to $\hat{y}$.

The choice of activation function σ depends on the problem being solved and the type of neural network being used. It determines whether the neuron should produce an output or not on the basis of a certain threshold. The activation function introduces non-linearity into the output of the neuron, allowing the neural network to learn complex relationships between inputs and outputs. Common activation functions include

$$\begin{aligned} &\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}, \\ &\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}, \\ &\text{Rectified linear unit ReLU}(x) = \max(x, 0), \\ &\text{Parameterized ReLU pReLU}(x) = \max(0, x) + \alpha \min(x, 0) \end{aligned} \tag{4.3}$$

and their variants.

As the model is first set up, the weights and biases for each layer are randomly chosen. The input x is processed by the model to produce an estimate $\hat{y}$ of the target value y. The next important step is to determine the manner in which the model will be penalized or rewarded based on changes in the parameters. The aim is to minimize the loss function and increase the accuracy of the neural network, which requires adjustments to its parameters θ.

Loss function. In machine learning empirical risk minimization (ERM) is employed to minimize the prediction errors on the training data

$$\min_{\theta} R_{\text{emp}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, f(x_i|\theta)\right) \tag{4.4}$$

where n is the number of samples, y_i the target label of the x_ith sample and $\hat{y}_i = f(x_i)$ the prediction. Commonly used *loss functions* L are:

- L^1 loss or Mean Absolute Error

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| = \frac{1}{n} \sum_{i=1}^{n} |y_i - f(x_i)|$$

- L^2 loss or Mean Squared Error

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i)))^2$$

- Cross-entropy loss, also known as negative log-likelihood loss. Measures the difference between the predicted probability distribution and the actual target distribution.

$$L(y, \hat{y}) = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) = -\sum_{i=1}^{n} y_i \cdot \log(f(x_i)) \tag{4.5}$$

  If $\hat{y}$ is a probability vector, which means that all its entries $\hat{y}_i$ are in the interval $[0, 1]$, the loss L has a minimum value of 0. Due to this fact, the softmax function (4.2) is often used as the output layer of a network to obtain a representation as a probability vector. For further information the reader is referred to [Section 4.1 in 19, pp. 132–133]

4.2 Optimization

Finding the optimal values for parameters θ that result in the minimum of the loss function L is crucial in deep learning. This involves solving a non-linear optimization problem. With a huge number of parameters that need to be adjusted for optimal results, various optimization algorithms are employed in deep learning, such as Stochastic Gradient Descent (SGD), Adam, and Adagrad. These algorithms iteratively adjust the model parameters in the direction that reduces the empirical risk function (4.4) until either a minimum

is reached or a stopping criterion is satisfied.
Classic gradient descent: For a risk function $R_{emp}(\theta)$ with the parameter vector $\theta = [\theta_1, \ldots, \theta_d]$, *learning rate* α and the gradient

$$\nabla R_{emp} = \left[\frac{\partial R_{emp}}{\partial \theta_1}, \ldots, \frac{\partial R_{emp}}{\partial \theta_d}, \right]$$

the iterative update in step k is set as

$$\theta^{k+1} = \theta^k - \alpha \nabla R_{emp}(\theta^k) = \theta^k - \alpha \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L \left(y_i, f(x_i | \theta^k) \right)$$

Stochastic gradient descent: (SGD) Applying the whole gradient of the loss function greatly increases the computational costs and slows down the training speed. Stochastic gradient descent operates on one randomly selected training example to compute the gradient, rather than using the entire training set. This makes SGD computationally efficient and well suited for large-scale problems, as it can handle huge amounts of data by updating parameters after each example is processed. The convergence profile of SGD looks noisy, and the rate is slower than for the classic gradient descent.

Batch Stochastic Gradient Descent: (BSGD) refers to the breaking down of the training set into smaller fragments, known as mini-batches. The mini-batch size is generally set to a low number, such as 32 or 64, which is much smaller than the total number of training examples. This allows each iteration of the algorithm to update the parameters based on a small subset of randomly selected training data. As mentioned in [Section 12.5 in 19], BSGD balances the trade-off between the noise introduced by SGD and the computational cost of classical gradient descent. A possible implementation is given by [Algorithm 8.1 in 4, p. 286]. The full run-through of the algorithm on the train set is called an *epoch*.

Algorithm 1: BSGD

```
For each epoch divide the train sett into p mini-batches $\mathcal{B}_1, \ldots, \mathcal{B}_p$
input  : Parameter vector $\theta^{(k)}$, learning rate $\alpha$
for $j$ = 1,...,$p$ do
    Sample a mini-batch of $m$ examples $x^{(1)}, \ldots x^{(m)} \in \mathcal{B}_j$ with
      corresponding targets $y^{(1)}, \ldots y^{(m)}$
    Compute the gradient estimate: $\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L \left( y^{(i)}, f(x^{(i)} | \theta^{(k)}) \right)$
    Apply the update: $\theta^{(k+1)} = \theta^{(k)} - \alpha \hat{g}$
end
```

Backpropagation. [Section 6.5 in 4] and [Section 5.3 in 19] were composed to form this short description of backpropagation. In a feed-forward network, the input x serves as the initial source of information that is then transmitted through the hidden layers to produce the final output $\hat{y}$. This is known as forward propagation or forward pass. On the other hand, backpropagation is the process of updating the weights of the network's parameters to minimize the error between the predicted output and the actual output. This is achieved by computing the gradient of the loss function with respect to the network's parameters. The error is initially calculated at the output layer, and then, using the chain rule of calculus, it is backpropagated through the network's layers, updating the weights at each layer along the way. The recursive algorithm of backpropagation, beginning with the output and iterating back through layers, is presented in [Algorithm 6.4 in 4, p. 206],

4.3 Challenges in optimization

The training process can be hindered by several issues such:
overfitting - When a model performs well on the training set but fails to generalize on unseen data;
saddle points, local minima - Gradient descent converges to a stationary point, which is a global minimum in the case of a convex function, but it can also get stuck at a local minimum or saddle point in other cases. The choice of learning rate helps to address this problem;
vanishing/exploding gradients - During backpropagation, the partial derivatives with respect to different parameters often have values that are less than 1. These derivatives, obtained through the application of the chain rule in calculus, are multiplied together multiple times, which ultimately leads to a significant decrease in their magnitude. As a result, the gradients become extremely small, making it difficult for the network to update all the parameters effectively. On the other hand, high gradient values result in a dramatic change in the parameters of the network. This scenario may result in numerical instability, impeding the network's ability to converge. The exponential increase in the magnitude of the gradient can cause the network parameters to oscillate leading to an unstable optimization;
slow convergence - Poor choice of hyperparameters, for example, small learning rate, will hinder the optimization;
sensitivity to weight initialization - Proper initialization is imperative for the successful convergence of extremely deep networks such as ResNet152 (Section 4.6, Figure 5). Random initialization may result in a failure to converge;
imbalanced dataset - The model may become biased towards the majority

class. As a result, the model may classify most instances as the majority class and fail to accurately classify instances from the minority classes.

4.3.1 Techniques for acceleration

To address the issues mentioned above, the some employed convergence acceleration strategies will be discussed next:

Weight decay. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as *regularization* [Section 7 in 4]. Adding an additional penalty term $J(\theta)$ and weight $\lambda \in [0, \infty)$ in equation 4.4 yields:

$$\min_{\theta} R_{\text{reg}}(\theta) = \min_{\theta} R_{\text{emp}}(\theta) + \lambda J(\theta)$$

Choosing J as the halved L^2 norm, one obtains:

$$\min_{\theta} R_{\text{reg}}(\theta) = \min_{\theta} R_{\text{emp}}(\theta) + \frac{\lambda}{2} \|\theta\|_2^2$$

The gradient descent step for the iteration k will then result in

$$\theta^{k+1} = \theta^k - \alpha \left(\nabla R_{emp}(\theta^k) + \lambda \theta^k \right) = \theta^k (1 - \alpha\lambda) - \alpha \nabla R_{emp}(\theta^k)$$

This results in a decline of the parameters proportional to their size and decay factor λ. When $\lambda = 0$, one would compute the classic gradient descent step. In other words, regularization is designed to tackle problems related to overfitting, model stability, and generalization.

Dropout. Dropout works by randomly setting the weights of a fraction of neurons to zero in a layer during training. This means that during each iteration, different neurons are dropped out, which introduces noise into the network and prevents it from relying too heavily on any particular feature. Dropout encourages the network to learn more robust features that are useful for making predictions even when some of the input features are missing. This helps to prevent overfitting and improve the generalization performance of the network on unseen data.

Batch Normalization. Batch normalization is a widely adopted and efficient technique that helps to boost the convergence rate of deep neural networks. Typically, it is treated as a layer within the neural network. An input sample $x \in \mathcal{B}$ in a mini-batch $\mathcal{B}$ is transformed into

$$\mathrm{BN}(x) = \gamma \frac{x - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \varepsilon} + \beta$$

where $\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum\limits_{x \in \mathcal{B}} x$ denotes the sample mean, $\sigma_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum\limits_{x \in \mathcal{B}} (x - \mu_{\mathcal{B}})^2$ the standard deviation of the batch $\mathcal{B}$ and ε is a stability constant. After applying standardization, the resulting mini-batch has zero mean and unit variance with two additional trainable parameters γ and β.

As shown in [Section 8 in 19], batch normalization offers benefits such as stabilizing and accelerating the training process, reducing overfitting for improved generalization, and exhibiting robustness to different architectures and hyperparameters. However, it comes with drawbacks, such as increased computational requirements and the need for sufficient sample sizes within a batch to compute normalization statistics.

Adaptive learning rate. Using fixed learning rate α during training can also cause convergence problems. Selecting an appropriate rate that is neither too small nor too large is a significant challenge for classical gradient descent techniques. As a result, an adaptive choice of the learning rate can considerably accelerate the convergence of many gradient-based optimization algorithms. For instance, in scenarios where local minima are present, increasing the learning rate can aid in "overshooting" the problematic area. Conversely, when encountering a very steep hill, reducing the step size becomes necessary. Adagrad (Adaptive gradient) and RMSProp (Root Mean Square Propagation) are two algorithms commonly used to estimate the appropriate learning rate for each gradient step.

Adagrad rescales the learning rates based on the inverse square root of the sum of past squared derivatives. This causes the parameters with high partial derivatives of the loss to experience a rapid decrease in their learning rate, while those with small partial derivatives experience a relatively smaller decrease in their learning rate. According to [Section 8.5 in 4], the accumulation of squared gradients can lead to a premature and excessive decrease in the effective learning rate in practice. While Adagrad performs well for some deep learning models, it may not be suitable for all of them.

RMSProp is a modification of Adagrad which deals with the issue of radically diminishing learning rate by using an exponentially decaying average to discard history from the extreme past, so that in can gain performance in non-convex landscape. The pseudocode of Adagrad and RMSProp can be found in [Section 8.5.1& 8.5.2 in 4] and [Section 12.7.3. & 12.8.1. in 19]

Momentum. Momentum is a modification of the classic gradient descent algorithm designed to speed up the optimization process. This is achieved by incorporating a momentum vector v and a hyperparameter $\varphi \in [0,1)$ that can help the algorithm deal with situations where the gradient signals are small, noisy, or consistently pointing in the same direction due to high curvature. The momentum term maintains an exponentially weighted moving average of past gradients; for a given learning rate α the next parameter update θ^{k+1} is computed as:

$$v = \varphi v - \alpha \nabla_\theta R_{emp}$$
$$\theta^{k+1} = \theta^k + v$$

Momentum replaces gradients with a leaky average over past gradients. This may significantly accelerate convergence.

Algorithm 2: Adam

```
For each epoch divide the train sett into $p$ mini-batches $\mathcal{B}_1, \ldots, \mathcal{B}_p$
For optimizer step $k$
set   : $s^{(k)} = 0, r^{(k)} = 0$
input : Parameter vector $\theta^{(k)}$, learning rate $\alpha$, exponential decay
        rates $\rho_1, \rho_2 \in [0,1)$, small numerical stability constant $\varepsilon$
for $j$ = 1,...,$p$ do
    Sample a mini-batch of $m$ examples $x^{(1)}, \ldots x^{(m)} \in \mathcal{B}_j$ with
     corresponding targets $y^{(1)}, \ldots y^{(m)}$
    Compute the gradient estimate: $\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(y^{(i)}, f(x^{(i)}|\theta^{(k)})\right)$
    Update biased first moment estimate: $s^{(k+1)} = \rho_1 s^{(k)} + (1-\rho_1)\hat{g}$
    Update biased second moment estimate:
     $r^{(k+1)} = \rho_2 r^{(k)} + (1-\rho_2)\hat{g}^2$ (applied element-wise)
    Correct bias in first moment: $s^{(k+1)} = \frac{s^{(k+1)}}{1-\rho_1^t}$
    Correct bias in first moment: $r^{(k+1)} = \frac{r^{(k+1)}}{1-\rho_2^t}$
    Compute the update: $\Delta\theta = -\alpha s^{(k+1)} \frac{1}{\varepsilon + \sqrt{r^{(k+1)}}}$
    Apply the update: $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta$
end
```

The optimization algorithm known as Adam (Adaptive Momentum) was utilized in all the neural networks discussed in this thesis. Adam (Algorithm 2) combines the advantages of the momentum and RMSProp optimization methods and uses both the first and second moments of the gradients. Specifically, Adam maintains an exponentially decaying moving average of past gradients (first moment) and an exponentially decaying moving average of past squared gradients (second moment), similar to RMSProp. Overall, Adam is a powerful optimization algorithm and is popular for its robustness, convergence speed, efficient memory utilization, and ability to train large neural networks.

4.4 Evaluation metrics

When evaluating a neural network, the specific metrics are used depending on the task the model is designed to perform. For a given model f, number of samples N, sample x_i, target y_i, prediction $\hat{y}_i = f(x_i)$, and loss function L, some basic classification metrics include the *accuracy* $= \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{y_i=\hat{y}_i}$ and *average loss* $= \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y}_i)$. During training, it is common to track the increase in validation accuracy and the decrease in average validation loss over epochs.

		Prediction	
	N	Positive	Negative
Target	Positive	True positive (TP)	False negative (FN)
	Negative	False positive (FP)	True negative (TN)

Table 2: Binary classification confusion matrix.

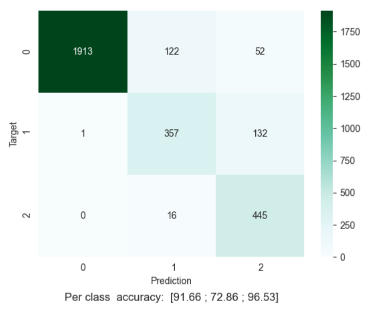


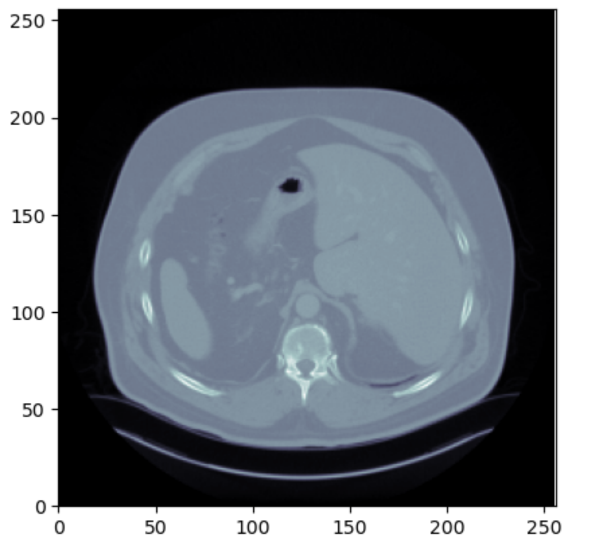
Figure 3: Validation confusion matrix of ResNet34 with per class accuracy

Another technique commonly used to evaluate the performance of a model is the confusion matrix. The confusion matrix organizes the actual and predicted values into rows and columns, respectively. Each row represents the samples in a target class, while each column represents the samples in a predicted class. Table 2 provides an illustration of a confusion matrix for a binary classification problem.

Using the confusion matrix, one can calculate the per-class accuracy by dividing the diagonal element of each class by the sum of all samples in that class. For binary classification, this corresponds to calculating $\frac{TP}{TP+FN}$ and $\frac{TN}{FP+TN}$. For three classes, an example of the confusion matrix is shown in Figure 3. The evaluation metrics for segmentation models will be discussed in Section 4.7.

4.5 Neural Networks for imaging

The next step involves applying the model definition to understand how the input image of a CT scan is transformed through the model into a probability vector of three targets. It is important to recall that the image format is a $3 \times 256 \times 256$ tensor, with each entry defining the color of a pixel in the red, green and blue channels, within a range of 0 to 255. Before feeding the image to the network, some transformations are performed. The model generates three output values, with the highest value indicating the predicted class. Applying the softmax function (Eq. (4.2)), the output can be converted into a probability vector. Figure 4 visually presents an image, the output of a trained classification model, and the corresponding probability vector.



Target: 1; Model output: [0.968 ; 2.768 ; -7.546];
Probability vector: [14.2 ; 85.8 ; 0.]

Figure 4: Example of an image, target value, ResNet18 output with the probability for each target.

The key part of image analysis is a convolution function.

4.5.1 Convolution

Convolution is a mathematical operation that combines two Lebesgue measurable functions $f, g \in \mathcal{L}_1(\mathbb{R})$. The presented definition can be found in [Section 4.2.2, Definition 4.7 in 3, p. 134]. To perform the convolution, one of the functions (usually the filter) is flipped in time and shifted over the other function (usually the input signal). At each point of overlap, the overlap values are multiplied, and the products are summed. The resulting function h represents the overlap between the two original functions

$$h = (f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\, d\tau$$

For the discrete sequence of integers, the convolution of an image f and a filter g is defined by the following.

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

In this case, discrete convolution can be visualized as sliding a kernel or filter over an image and calculating the sum of the element-wise product between the filter and the image pixels within the overlapping region. The resulting sum is then placed in the corresponding pixel of the output image. This operation is performed for every pixel in the input image, using specific parameter settings, typically resulting in an output image with the same dimensions as the input image. If indicated, the bias value is added to the result of each pixel after the kernel operation. In this scenario, the output pixels are a combination of weighted input pixels and bias. This is why we can view convolution as a weighted operation, represented by Equation (4.1).

A simple example (Figure 5) with a kernel of size 2×2 and bias 0 could be found in [Figure 7.2.1, 19].

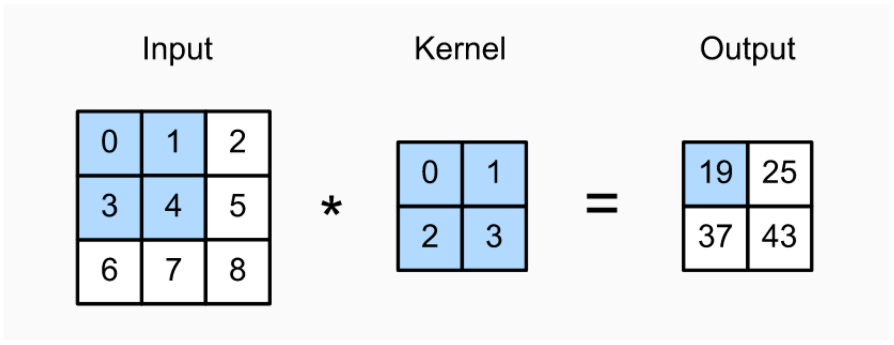


Figure 5: Convolution example.

The generation of multiple kernels to detect specific features or patterns in the input image, such as edges, corners, or textures, is a sensible approach. The number of features that a layer is designed to recognize is called the *number of output channels*.

There are some important parameters regarding convolution. Though the kernel size could vary, the common practice is to use quadratic kernels smaller size, for example 3×3 or 5×5. The earlier results, [Section 2.3 in 17], show that a sequence of multiple layers with small kernels has the same receptive field as a single layer with a larger kernel. For example, three sequential 3×3 layers have the same receptive field as a 7×7, but with reduced number of parameters from 49 to 27.

The convolutional layer use a *stride* parameter to determine how far the kernel moves across the input image. It determines by how many pixels the filter is shifted horizontally and vertically at each step during the operation. In the example above, the stride is 1. When the stride is set to a value greater than 1, the filter skips pixels during each shift. Hence, the stride greater than 1 reduces the spatial dimensions from input to output feature maps.

Padding can be utilized when both larger stride value and maintaining the dimensions are required simultaneously. It is defined as the addition of extra values (zeros or from the opposite edge) around the edges of the input image before applying the convolution operation. This ensures that the kernel processes all pixels in the input image and prevents the loss of information at the edges of the image.

4.5.2 CNNs

Convolutional Neural Networks (CNNs) are deep learning models specifically designed to analyze visual data. CNNs are constructed using multiple layers, each of which performs a specific function in analyzing the input image. The first layer is typically a convolutional layer that applies a set of filters to the image to extract features such as edges, corners, and textures. To introduce non-linearity into the model, the output of the convolutional layer is then passed through an activation function, such as ReLU (Equation (4.3)). Subsequent layers in the network typically perform downsampling operations, which reduce the size of the feature maps and help to make the model more robust to small variations in the input.

As a couple of examples, one can mention AlexNet, [7], which consists of 8 layers and was a breakthrough in computer vision in 2012. Table 3 contains the structure of the network. The input image would be processed from left to right through the network of convolutions denoted as "conv$\langle$kernel size$\rangle$-$\langle$number of output channels$\rangle$" and pooling operations denoted as "pool". After the convolution operation, the activation function tanh() is applied.

The reference [Section 7.5 in 19] provides comprehensive details on the pooling operation, including its properties and parameters. The final layers of the model consist of linear transformations represented by the equation (4.1), commonly called *fully connected* layers (FC).

input	conv11-96	pool	conv5-256	pool	$\underbrace{\text{conv3} - 384}_{\times 3}$	pool	$\underbrace{\text{FC}}_{\times 3}$

Table 3: AlexNet architecture.

VGG, or the Visual Geometry Group, is a deep CNN family that was introduced in 2014 in [17]. The main idea behind VGG was to investigate the effect of depth increase on the accuracy of a CNN. The network consists of 16 or 19 layers, and all layers in the network use a small kernel size of 3x3. The architecture of VGG (Table 4) is very simple and consists of a series of convolutional layers, each followed by a max-pooling layer. The last 3 layers similar to the AlexNet are fully connected layers that produce class probabilities.

input	$\underbrace{\text{conv3} - 64}_{\times 2}$	pool	$\underbrace{\text{conv3} - 128}_{\times 2}$	pool	$\underbrace{\text{conv3} - 256}_{\times 4}$	pool
$\underbrace{\text{conv3} - 512}_{\times 4}$		pool	$\underbrace{\text{conv3} - 512}_{\times 4}$	pool	$\underbrace{\text{FC}}_{\times 3}$	

Table 4: VGG19 architecture.

Although AlexNet pioneered many of the features that make deep learning successful on a large scale, VGG is often credited for introducing critical characteristics such as using ReLU (Equation (4.3)) as an activation function to address the problem of vanishing gradients, utilizing blocks of multiple convolutions and favoring deep and slender networks. Furthermore, the VGG architecture has become a fundamental building block in many state-of-the-art deep learning models because of the number of learnable parameter reduction. It introduced the advantages of using 1×1 convolutional layers and a global average pooling layer as a convenient solution to tackle overfitting behavior in the final layers.

Theoretically, deeper neural networks have the potential to capture more features of data, raising the question of whether adding more layers to a network improves its performance. The answer is yes, but under certain circumstances. As one shall see in the following section, classical networks suffer from deteriorating performance when the number of layers increases.

4.6 ResNet

In this section, a family of neural networks known as residual neural networks (ResNets) will be introduced. They were initially introduced and widely tested in the article [5] in 2015. Practical experiments will be compared with own implementation of residual networks (as well as some others), highlighting the main differences and advantages.

When training deeper networks, a common obstacle is the vanishing/exploding gradient problem. Fortunately, this problem has been successfully resolved through normalized initialization and batch normalization layers, allowing networks with multiple layers to start converging efficiently.

Upon successful convergence of deeper networks, a degradation problem arises, when the accuracy saturates and then declines as the network depth increases. Surprisingly, this degradation is not a result of overfitting, and adding more layers to a sufficiently deep model results in higher errors, as verified by experiments.

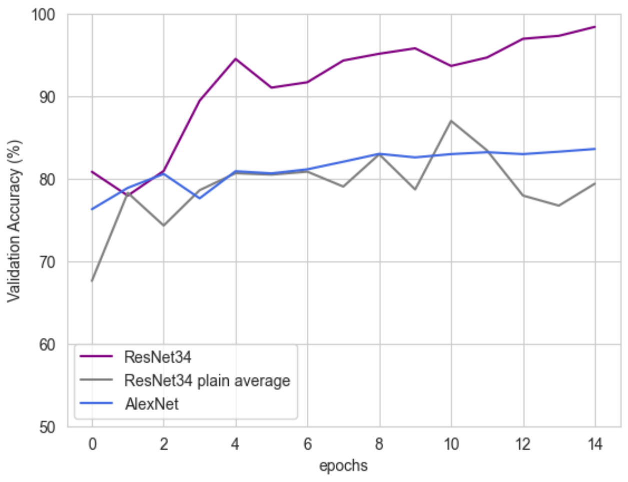


Figure 6: Accuracy comparison ResNet34 with the plain variant.

Figures 6 and 7 provide a typical example of this phenomenon. Three models were trained for 15 epochs on the the same dataset, cross entropy loss function (Equation (4.5)), hyperparameters (5.2) and optimizer Adam (Algorithm 2). AlexNet is a relatively small network of 8 layers discussed in Table 3. ResNet34 plain is based on ResNet with 34 layers, but without an important concept of "shortcut connection", which will be discussed in detail later (Figure 8). Both AlexNet and ResNet34 exhibit stable training and produce similar results when the training process is reproduced. However, the plain variant of ResNet with 34 layers is prone to instability, with the

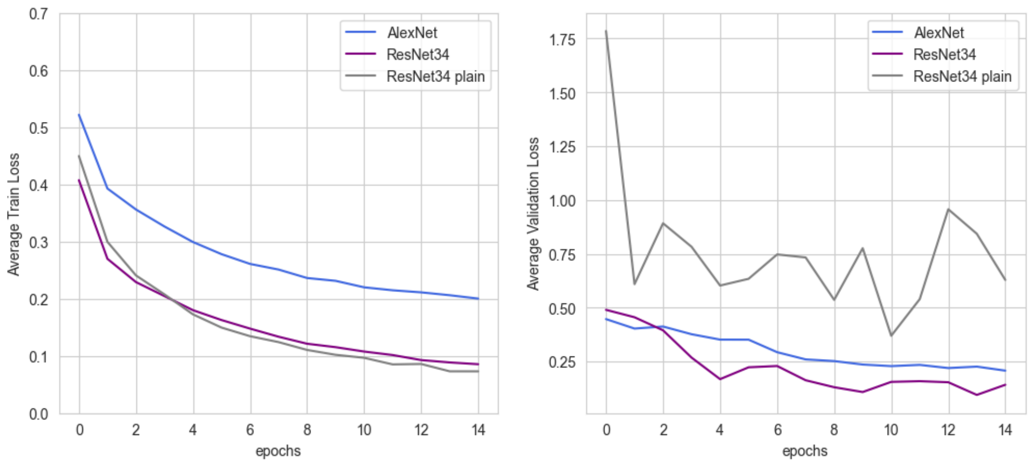


Figure 7: Losses comparison ResNet34 with the plain variant.

optimizer failing to update all parameters even when using an adaptive algorithm. Consequently, the data for plain ResNet in Figure 6 were obtained by sampling from triple training and averaging the results. The validation accuracy is calculated at the end of each training phase using the validation subset of the dataset (Table 1).

If both the training error and the validation error decrease, it suggests that the model is not overfitting (Figure 7).

Despite its smaller size, AlexNet outperforms the plain deeper model but loses to the original ResNet34 due to its depth. The capacity of the small model is depleted, and the maximum validation accuracy that could be achieved from it is only around 84%. The larger model captures the data distribution faster and achieves a higher validation accuracy rate of over 98%. On the contrary, the plain variant struggles to converge.

4.6.1 Residual Blocks

A residual block in Figure 8 is the basic building block of a ResNet. It consists of two or more convolutional layers with regularization and ReLU (Equation (4.3)) activation functions, along with skip connections that allow the input to be directly added to the output of the block. The convolutions contained within the block typically employ small kernel sizes, such as 3×3 or 1×1.

Let x be the input. The objective is to learn the underlying mapping $f(x)$ that will be used as input for the activation function at the top. The dotted line box on the left side must learn the mapping $f(x)$ directly, while

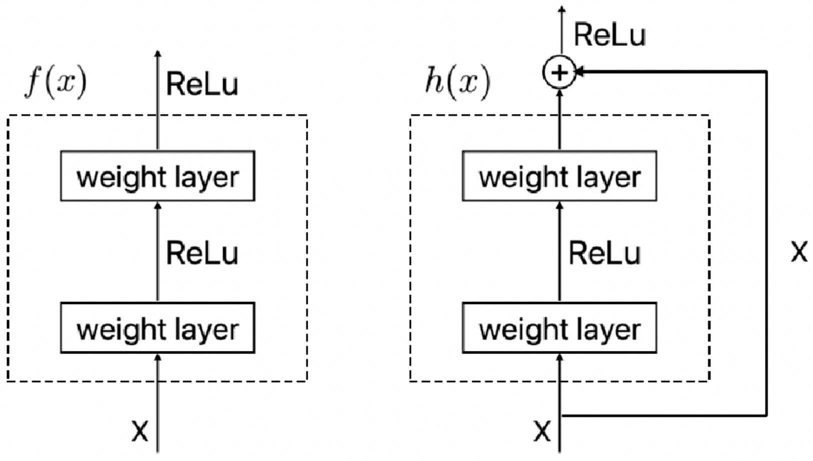


Figure 8: Plain block (left) vs residual block (right).

the one on the right needs to learn the residual mapping $h(x) = f(x) - x$. If the desired underlying mapping is the identity mapping $f(x) = x$, then the residual mapping can be simplified to $h(x) = 0$, making it easier to learn. This is accomplished by adjusting the weights and biases within the dotted line box to zero.

The residual block on the right includes a solid line that represents a shortcut connection created by an identity mapping. This connection, known as a residual connection, accelerates the propagation of inputs through layers, allowing very deep neural networks to be trained more easily. Additionally, one of the benefits of the block is its ability to detect redundant information, which helps to avoid the vanishing gradient problem that may arise in deep networks. As a result, adding more blocks to the model will not have a negative impact, since the model can learn to skip unnecessary information. Another significant advantage is that shortcut connections do not increase the number of parameters or computational complexity of the model and are easy to implement.

4.6.2 Residual Network

Rephrasing the idea from the VGG network, the residual blocks can be stacked together to form very deep neural networks with hundreds of layers. This allows ResNets to achieve state-of-the-art performance on a wide range of computer vision tasks. The primary approach is to increase the number of channels while simultaneously decreasing the spatial dimension through the use of a convolution with a stride of 2. The network is categorized into five prominent layers, each having a specific number of stacked blocks. At the beginning of each layer, a downsampling operation is performed to reduce the spatial dimension of the input by half. This implies that the shortcut connection cannot be added as an identity function anymore, because the input and output of dimension reduction block have different dimensions. In this case, a linear projection W_p in the form of a 1×1 convolution is used to shape the input to the right dimensions.

standard block	dimension reduction block
$f(x) = h(x) + x$	$f(x) = h(x) + W_p x$

To build deeper networks, it becomes necessary to optimize the block structure for improved computational efficiency and reduced training time. In the bottleneck ResNet block, each residual block consists of three convolutional layers, where the first and the last convolutions are 1x1, while the middle one is 3x3. This design helps to reduce computational cost by reducing the number of channels passed between layers.

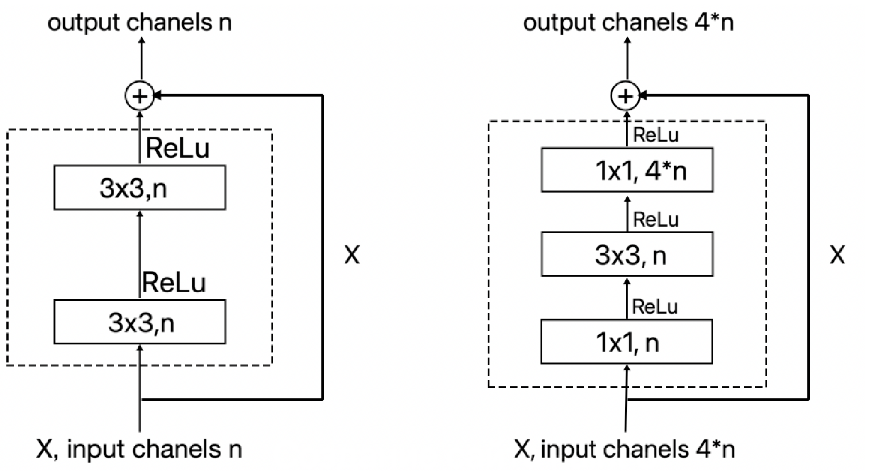


Figure 9: The standard ResNet block (left) vs. Bottleneck block (right).

Additionally, bottleneck ResNets are designed to use fewer parameters than traditional ResNets by replacing some of the 3x3 convolutions with 1x1 convolutions, which can be thought of as feature pooling layers that reduce the number of input channels. This reduces the overall number of trainable parameters but preserves the network's ability to learn complex features. The difference between a classical block and a bottleneck block is illustrated in Figure 9.

layer name	out size	18-layer	34-layer	50-layer	152-layer
conv1	110×110	$7 \times 7, 64$ stride 2			
conv2_x	54×54	3×3 max pool, stride 2			
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	27×27	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, FC - dims 3, softmax			
num. params $\cdot 10^6$		11.1766	21.2799	23.5080	58.1437

Table 5: Block structure of residual networks.

Table 5 highlights different ResNets, depending on the total number of layers. The square brackets define the parameters for the residual block as follows $\left[\text{Kernel size}, \text{Output channels}\right] \times$ Number blocks. The subscript _x indicates the number of repeating blocks considered in the global horizontal layer. Specifically, conv3_2 denotes the second block in the third global layer.

To enable deeper networks with 50 and 152 layers, the bottleneck block is used. From the second global layer onward, the dimension reduction block is used as the starting block for conv3_1, conv4_1, conv5_1. This block halves the output spatial dimension by using convolution with a stride of 2. It is worth mentioning that earlier networks such as AlexNet and VGG used the max-pooling operation for this purpose.

All networks use average pooling in the last three layers, followed by linear layers that transform the output into a dimension of 3 targets, and the softmax function is used to obtain a probability vector. Moreover, in the PyTorch framework (Section 5.1.1), the last softmax function is not necessary because it is already included in the definition of the cross-entropy loss function.

4.7 U-Net

Switching from classification problem to segmentation, one architecture that stands out is U-Net, derived from its "U-shaped network" structure. It adopts a block structure similar to that of previous networks. U-Net is a widely used CNN architecture specifically designed for semantic segmentation tasks. Originally proposed by researchers at the University of Freiburg in 2015, [16], it has gained popularity in various fields, particularly in medical image analysis, where precise segmentation of organs or abnormalities is crucial. Figure 11 depicts the U-Net architecture, which aims to capture both local and global characteristics.

The main goal of U-Net is to perform classification on individual pixels or regions within an input image. This is particularly useful in tasks such as medical image analysis, where precise segmentation of organs or abnormalities is essential. The resulting output consists of three segmentation layers, stacked together, with the same dimensions as the input. These layers indicate the predicted boundaries for the background (0), liver (1), and tumor (2). Figure 10 provides an example of the output.

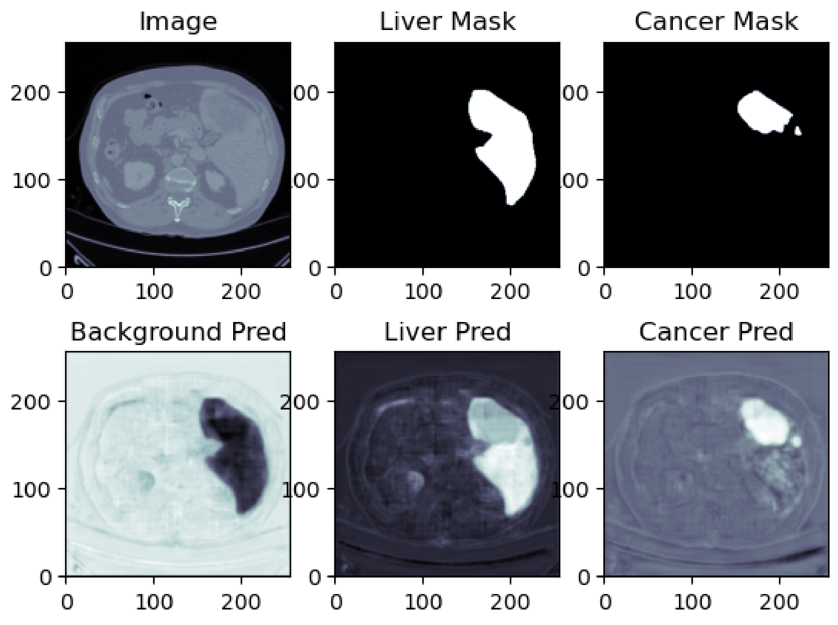


Figure 10: U-Net model output.

The encoding path, situated on the left side, gradually reduces the spatial dimensions of the input image. It accomplishes this by using a sequence of 3×3 convolutions, activation functions, and pooling layers, thus doubling the

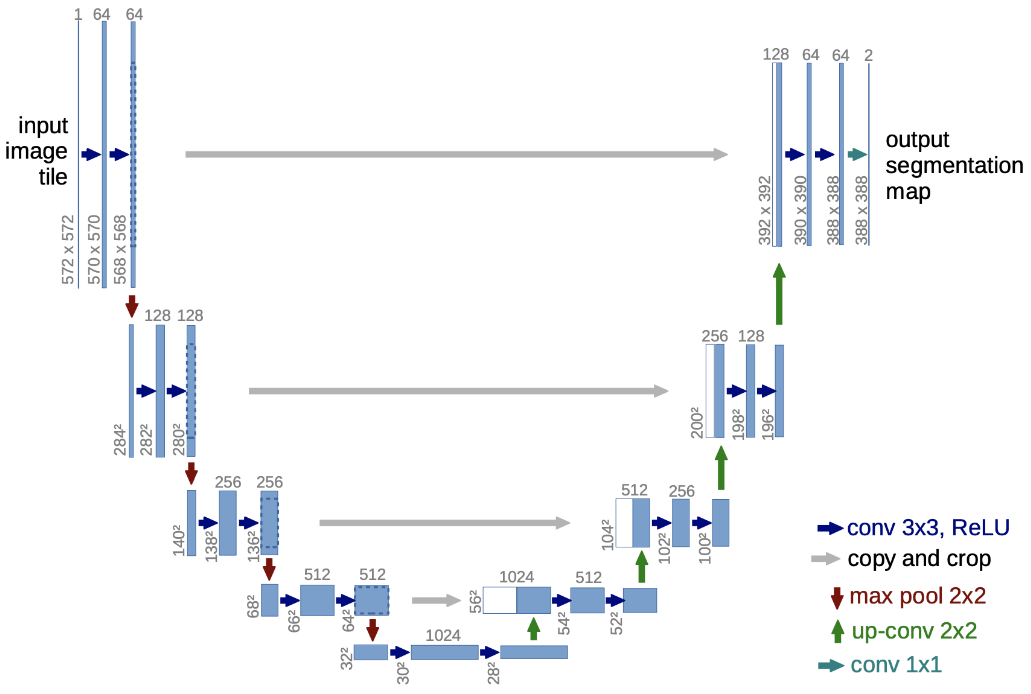


Figure 11: U-Net architecture, source [16, Fig.1].

number of feature channels and extracting high-level features. This process is similar to that of a traditional CNN. The horizontal gray arrows represent the shortcut connections on a per-layer level. Before each downward-pooling operation, intermediate results are saved and later concatenated with the corresponding results from the decoding path. In our implementation, unlike the original network, the horizontal 3×3 convolutions include padding, eliminating the need for cropping during skip connections. Additionally, since our dataset contains smaller images with dimensions of $1 \times 256 \times 256$ as opposed to $1 \times 572 \times 572$, we opted for 3 skip connections instead of 4. Therefore, the dimensions in the bottom layer before the decoder are $512 \times 32 \times 32$.

The decoding path of UNet focuses on upsampling the spatial dimensions and recovering lost details. It achieves this through the utilization of transposed convolutions, also known as inverse convolutions [15]. Each decoding step involves upsampling of the feature map with a transposed convolution 2×2 that half the number of feature channels. Subsequently, the feature map is concatenated with the corresponding saved feature map from the contracting path. Finally, two 3×3 convolutions are applied, each followed by an activation function, to refine the segmentation.

Specific loss function. Since the classification now operates per pixel, a new definition of the loss function becomes necessary. After consideration and discussion during the seminar, the pixel-wise Cross-Entropy Loss function was selected due to its better performance. This choice was made among several other loss functions discussed. Let y_{bg}, y_{liv}, and y_{tum} represent the target binary masks for the input image x. The model output for x consists of three prediction layers, namely $\hat{y}_{\text{bg}}$, $\hat{y}_{\text{liv}}$, and $\hat{y}_{\text{tum}}$. To compute the loss, element-wise products and logarithm functions are utilized on all pixels, resulting in the following expression (additional weight coefficients can be included for each sum if necessary):

$$\sum_{\text{all pixels}} (-1) y_{\text{bg}} \log(\hat{y}_{\text{bg}}) + \sum_{\text{all pixels}} (-1) y_{\text{liv}} \log(\hat{y}_{\text{liv}}) + \sum_{\text{all pixels}} (-1) y_{\text{tum}} \log(\hat{y}_{\text{tum}})$$

Evaluation metrics for segmentation. By incorporating per-pixel cross-entropy loss, the accuracy of individual pixels can be improved. However, this approach may not be suitable in certain situations. In some cases, it is important to accurately identify entire structures or consider the relationships between different structures. Therefore, when evaluating segmentation metrics, it is crucial to consider the overlapping areas between the predicted masks and the ground-truth target masks. In addition to evaluation functions for classification models, IAIM [6] has also introduced evaluation functions for segmentation models. Several notable segmentation metrics were discussed during the seminar, including, among others:

$$\text{Pixel Error PE}(tar, pred) = \sum_i |tar_i - pred_i|$$

$$\text{Dice Coefficient} = 1 - \text{dice}(tar, pred) = 1 - \frac{2|tar \cap pred|}{|tar| + |pred|}$$

$$\text{Intersection over Union} = \text{IoU}(tar, pred) = \frac{|tar \cap pred|}{|tar \cup pred|}$$

where *tar* denotes the target mask and *pred* the model output with predicted segmentation.

Softmax. The issue of probability output, similar to residual networks, also persists with U-Net. It is desirable to have an output in the range of [0,1]. To achieve this, the per-pixel softmax function is utilized. However, in the PyTorch framework (Section 5.1.1), the softmax function is already incorporated into the definition of the cross-entropy loss function. For the purpose of creating plots, the softmax function is applied manually, as demonstrated in Figure 25.

5 Experiments on Clean_LiTS

In this section, the theoretical findings will be integrated with practical implementation by performing experiments and applying them to the dataset outlined in Section 3.

Our main objective is to assess the performance of various deep residual networks on the provided dataset using identical hyperparameters. It is important to note that, due to the stochastic nature of training processes, the results may differ between runs. As highlighted in [20], it is crucial to understand the sources of randomness and exercise control at various levels. The randomness observed in our tests can be classified as algorithmic randomness, which includes data augmentation (random transformations), shuffling of data inputs (batch order randomness), adaptive optimizer, model initialization, and stochastic layers within the model (such as dropout or batch normalization).

Taking into account the stochastic nature, a comparison will be made between residual models (ResNet18 - 152, Table 5) with standard convolutional networks without shortcut connections (Table 3 and 4). In the subsequent section, the impact of hyperparameters and model initialization on convergence will be explored, comparing random weight initialization with normalized initialization. Later, the practical implications will be examined by investigating if models with different capacities and operation counts require varying amounts of time per epoch. If a time threshold is set and the models are trained for a fixed amount of time, what happens? Can a model that is trained faster with more epochs achieve better results than a deeper model with fewer epochs? The final part of the section presents the results for the segmentation model.

The experiments were carried out using an external Amazon Web Services (AWS) workspace provided by IAIM [6], which used a cloud Graphics Processing Unit (GPU) for execution operations. Utilizing GPUs to train neural networks can greatly reduce the time and computational resources necessary for model training, making it a prevalent choice in deep learning. The primary advantage of GPUs lies in their ability to perform extensive parallel processing, allowing for faster computations on large datasets compared to Central Processing Units (CPUs).

During the seminar, there were technical issues and high server demand, which caused changes in the utilization of GPU cores. However, in most cases, the NVIDIA Tesla V100 GPU was used for training. This GPU has 5,120 CUDA-Cores, 640 Tensor-Cores, and 16GB of VRAM.

5.1 Implementation

5.1.1 Code framework & Git

To conduct practical code experiments and case study, it was chosen to use PyTorch [12], an open source machine learning framework that was developed by Facebook's AI Research team and was released in 2016. PyTorch provides a range of Python modules that simplify the process of creating, training, and deploying deep learning models. These modules include pre-written layers and functions, such as dataset definition, data augmentation, activation functions, optimizer, loss function, and backpropagation. To use them efficiently, one must maintain the syntax, input, and output dimensions of the functions.

The Git repository [18] contains a set of codes and functions that were presented in the thesis as Python Jupiter notebooks. In addition, there is a requirements file that lists all the necessary packages required to run the code. Furthermore, IAIM has provided supporting files (dataset creation, evaluation, logging routine, etc.) that will also be available on Git. However, these files will include a credit comment at the beginning to acknowledge their contribution.

5.1.2 Models

ResNet18 and 34. As an example, a definition of a generalized ResNet Block for the ResNet18 and 34 networks will be presented.

The PyTorch *torch.nn* Neural Network module provides fundamental tools for building neural networks. One common strategy for building a neural network is to use the *nn.Sequential()* class, which allows the straightforward definition of a layer sequence where the output of the preceding layer serves as input to the subsequent layer. Furthermore, the *ResBlock()* class, which inherits from the *nn.Module*, can be applied by implementing and overwriting two functions: *__init()__* for initialization and *forward()* for transmitting data through the block.

```
import torch, torch.nn as nn
# ResBlock Class
#       - constructs a block [conv -> batch_norm -> activation]*2, which we
    will stack in the network
# Input:     int: num_chans - number of channels
# Output:    nn.Sequential() block

class ResBlock(nn.Module):
    def __init__(self, num_chans):
        super().__init__() #accessing the superclass nn.Module
        self.conv1 = nn.Conv2d(num_chans, num_chans, kernel_size=3, stride
            =1, padding=1, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(num_features=num_chans)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(num_chans, num_chans, kernel_size=3, stride
            =1,  padding=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(num_features=num_chans)
        ... # weights initalization part
    def forward(self, x):
        out_1 = self.relu(self.batch_norm1(self.conv1(x)))
        out_2 = self.relu(self.batch_norm2(self.conv2(out_1)))
    return out_2 + x # this sum realises the skip connection
```

The dimension reduction block is defined in a way that is very similar to the *ResBlock()*, but with the addition of a downsampling layer to project the input into dimensions of the output.

```
class ResBlockDimsReduction(nn.Module):
    def __init__(self, num_chans_in, num_chans_out):
        ...
        self.downsample = nn.Sequential(
            nn.Conv2d(num_chans_in, num_chans_out, kernel_size=1, stride=2,
                bias= False),
            nn.BatchNorm2d(num_features=num_chans_out),
            nn.ReLU()
        )
    def forward(self, x):
        ...
    return out_2 + self.downsample(x)
```

To form the network in the next step, one only needs to stack blocks together as described in Table 5:

```
class ResNetMLMed34(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 64,
            kernel_size =7, stride =2, padding=1, bias= False)
        self.batch_norm1 = nn.BatchNorm2d(num_features=64)
        self.pool2 = torch.nn.MaxPool2d(kernel_size = 3, stride = 2)
        self.relu = torch.nn.ReLU()
        self.resblocks2 =nn.Sequential(
            *(3 * [ResBlock(num_chans=64)]))
        self.resblocks3 = nn.Sequential(
            ResBlockDimsReduction(num_chans_in=64,num_chans_out=128),
            *(3 * [ResBlock(num_chans=128)]))
        self.resblocks4 = nn.Sequential(
            ResBlockDimsReduction(num_chans_in=128,num_chans_out=256),
            *(5 * [ResBlock(num_chans=256)]))
        self.resblocks5 = nn.Sequential(
            ResBlockDimsReduction(num_chans_in=256,num_chans_out=512),
            *(2 * [ResBlock(num_chans=512)]))
        self.avgpool6 = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(in_features=512, out_features=3, bias=True)
    def forward(self, x):
        out_1 = self.conv1(x)
        out_1 = self.batch_norm1(out_1)
        out_1 = self.relu(out_1)
        out_1 = self.pool2(out_1)
        out_2 = self.resblocks2(out_1)
        out_3 = self.resblocks3(out_2)
        out_4 = self.resblocks4(out_3)
        out_5 = self.resblocks5(out_4)
        out_6 = self.avgpool6(out_5)
        out_6= self.fc(torch.flatten(out_6, start_dim=1))
    return out_6
```

ResNet50 and 152. To define the bottleneck block, a more generalized approach was adopted. The dimension reduction block was replaced with an additional parameter during the initialization of the block. Now, for each block, the *stride* for the second 3×3 convolution is defined (by default set to 1) and a Boolean variable *downsample* is added. This variable must be set to *True* to perform downsampling.

```
class ResBlockBottleneck(nn.Module):
    def __init__(self, num_chans_in, num_chans_between, num_chans_out,
        downsample = False, stride = 1):
        super().__init__()
        self.conv1 = nn.Conv2d(num_chans_in, num_chans_between, kernel_size
            =1, padding=0, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(num_features=num_chans_between)
        self.relu = torch.nn.ReLU()
        self.conv2 = nn.Conv2d(n_chans_between, num_chans_between,
            kernel_size=3, stride= stride, padding=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(num_features=n_chans_between)
        self.conv3 = nn.Conv2d(num_chans_between, num_chans_out, kernel_size
            =1, padding=0, bias=False)
        self.batch_norm3 = nn.BatchNorm2d(num_features=num_chans_out)
        ... # weights initalization part
        if downsample:
            self.downsample = nn.Sequential(
                nn.Conv2d(num_chans_in, num_chans_out, kernel_size=1,padding
                    =0,stride=stride, bias=False),
                nn.BatchNorm2d(num_features=num_chans_out),
                nn.ReLU()
            )
        else: self.downsample = nn.Identity()
    def forward(self, x):
        out = self.relu(self.batch_norm1(self.conv1(x)))
        out = self.relu(self.batch_norm2(self.conv2(out)))
        out = self.relu(self.batch_norm3(self.conv3(out)))
    return out + self.downsample(x)
```

U-Net. Implementing U-Net is a straightforward process, especially if one is already familiar with the block structure of residual models. The encoder and decoder components of U-Net also employ block structures, which provide significant advantages in terms of generalization and flexibility. The same generalization approach applied to the *ResBlock()* can be employed for the block comprising double convolutions in the horizontal direction of the model. The psydocode Block implementation therefore can be summarized to:

$$\underbrace{\text{DoubleConvBlock()} \rightarrow \text{MaxPool()}}_{\times 3} \rightarrow \underbrace{\text{InverseConv()} \rightarrow \text{DoubleConvBlock()}}_{\times 3}$$

finishing with the output convolution projecting features into right dimensions.

5.2 Hyper-parameters

Hyperparameters are parameters set before training that determine how the network is trained. They cannot be learned by the network during training. If not specified, the default hyperparameters are

- Augmentations: random horizontal and vertical flip with probability 0.5; random color jitters; random crop to the image of size (224,224)
- batch_size = 32
- starting learning_rate = 10^{-4}
- weight_decay = 10^{-5} - L2 regularization.
- number epochs = 15

If not specified, the optimizer algorithm used is Adam (Algorithm 2) and the loss function is the cross-entropy loss (4.5).

5.3 CNN's Comparison

Overall classification results. The classification models trained on the Clean_LiTS dataset and the hyperparameters from Section 5.2 include AlexNet, ResNet18, ResNet34, ResNet50, VGG19 and ResNet152. Table 6 presents the number of parameters (in millions), average runtime (in seconds), and test accuracy (in %) of these models. With the exception of ResNet152 and AlexNet, all models achieved high accuracy rates of more than 95%. However, because the dataset contains only 40,000 images, ResNet152 is too large to be practical. Furthermore, it takes an average of 260 seconds per epoch to train ResNet152, making it the slowest among the listed models.

Due to the imbalance of the dataset, the first target (number 0) is classified faster than the other targets during training. To address this issue, increasing the data augmentation for the minority class to generate more images and using class weighting are two possible methods. Class weighting assigns higher weights to the loss function during training to give more importance to the underrepresented class, which encourages the loss function to favor the minority target over the others. Other methods include undersampling, data generation, and transfer learning. Residual models and VGG19

	AlexNet	ResNet18	ResNet34	ResNet50	ResNet152	VGG19
num. params $\cdot 10^6$	46.736	11.1176	21.2799	23.5080	58.1437	139.5814
runtime (sec.)	36.46	50.67	69.98	119.28	267.67	200.83
test acc. (%)	83.54	98.35	94.8	98.39	82.52	98.35

Table 6: Comparison of number parameters, runtime and test accuracy of all classification networks.

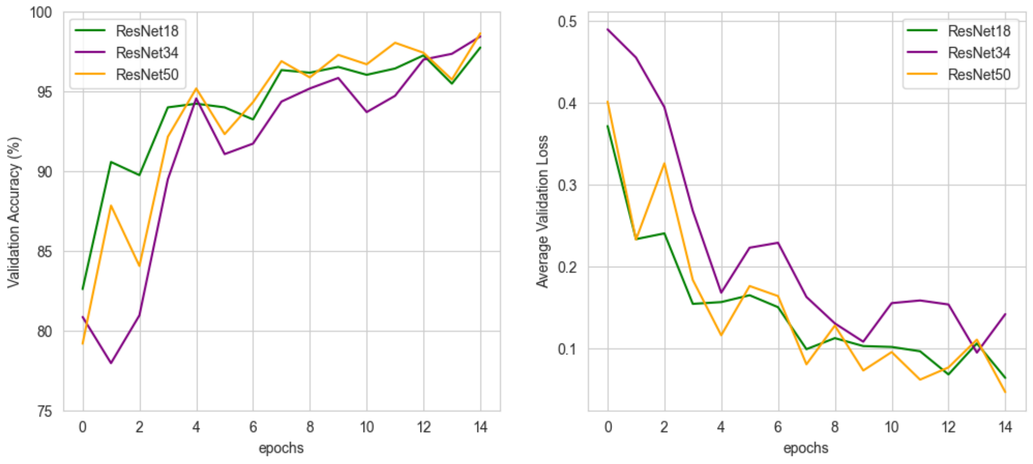


Figure 12: Comparison ResNet18, ResNet34 and ResNet50.

are less affected by the imbalance, but AlexNet may encounter serious convergence problems, as will be illustrated later.

In Figure 12, the accuracy of the validation and the loss of the primary residual networks tested are displayed. A notable finding, as mentioned in Section 4, is that adding more layers to ResNet34 and obtaining ResNet50 led to faster and better learning, as evidenced by the higher accuracy rate and lower loss. Moreover, the decreasing validation loss indicates that the models are not overfitting. Notably, the relatively compact ResNet18 performed exceptionally well, surpassing the deeper network by achieving 90% accuracy from the third epoch, which is the fastest rate among all the tested networks.

Best residual network vs best CNN. The validation accuracy and loss of ResNet 18, VGG19 and AlexNet are compared in Figure 13. ResNet 18 and VGG19 are high-performing residual and classic convolutional networks, respectively. Both networks exhibit very similar accuracy curves and achieve high accuracy rates in data classification. However, ResNet18 has significantly fewer trainable parameters than VGG19, resulting in 4 times faster training times with comparable accuracy rates.

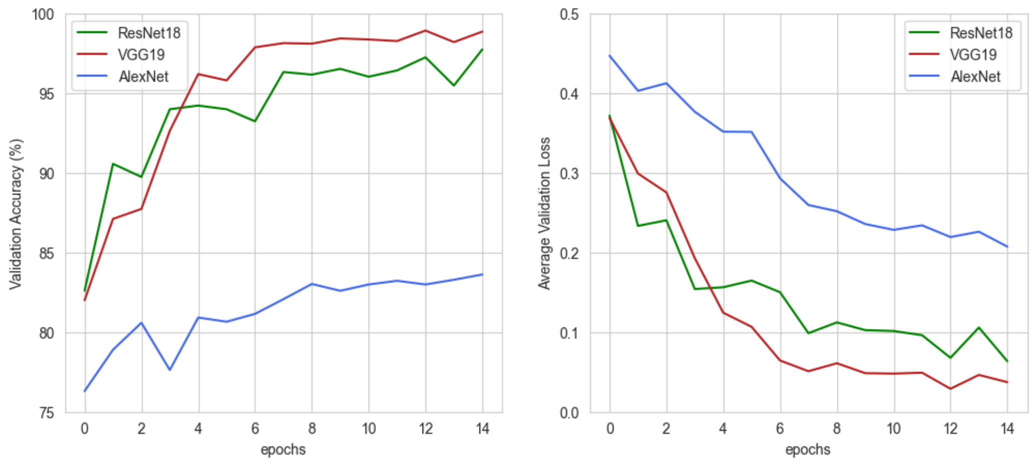


Figure 13: Comparison ResNet18, VGG19 and AlexNet.

Figures 12 and 13 provide additional evidence that deeper models do not reach a performance plateau once they reach 90% accuracy. After learning the largest class, the models gradually improve the classification of the remaining targets, as shown in Figure 14, which shows the accuracy rates per class of ResNet18 and VGG19 calculated from the confusion matrix after the validation phase.

It is important to note that the initial dataset imbalance, as mentioned earlier in this section, only significantly impacts the models' performance in the early epochs. However, by the fifth epoch, both ResNet18 and VGG19 surpass the accuracy of 85% for each class. This can be attributed to the fact that when all images are initially classified as the first target (number 0), the losses for the remaining targets increase. Higher loss values result in larger updates during gradient descent, leading to significant improvements in the classification of the second and third classes. In particular, Figure 14 shows substantial jumps in accuracy for targets 1 and 2 after the third and fourth epochs, for both ResNet18 and VGG19 models.

With an accuracy of over 98% in the test set, ResNet18 and VGG19 became the most effective classifiers for the given problem (Figure 15). When the models achieve 95% accuracy, the differences in their performance become negligible, indicating satisfactory training.

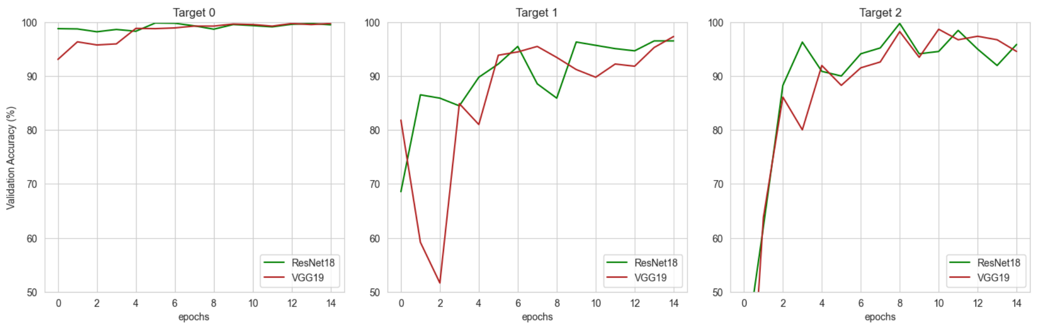


Figure 14: Per class validation accuracy of ResNet18 and VGG19.

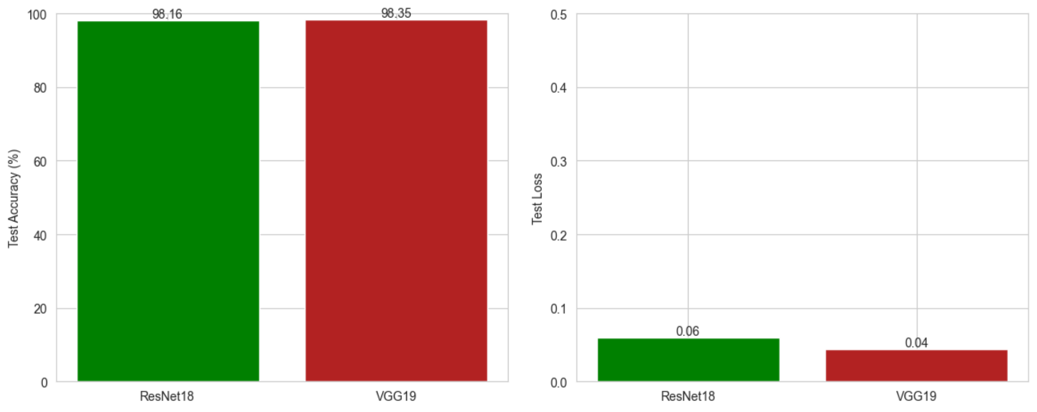


Figure 15: Test accuracy and test loss of ResNet18 and VGG19.

Robustness with respect to the learning rate. Selecting suitable hyperparameters is a crucial component of constructing a reliable model that can generalize well to new data and ensure the optimizer algorithm's convergence. This typically involves experimenting with different hyperparameter values and evaluating the model's performance on a validation set. The optimization learning rate is one of the most critical hyperparameters, as it controls the step size taken during gradient descent and can impact the speed and accuracy of convergence. A high learning rate may cause the model to overshoot the optimal solution, whereas a low learning rate may result in slow convergence.

Figure 16 displays three graphs that depict the validation accuracy of four models tested over 15 epochs. The VGG19 and ResNet152 models were not included in the tests due to limited resources on the web kernels and the long training time required. Additionally, the results of ResNet152 were too unstable to justify repeating the training process. The influence of weight

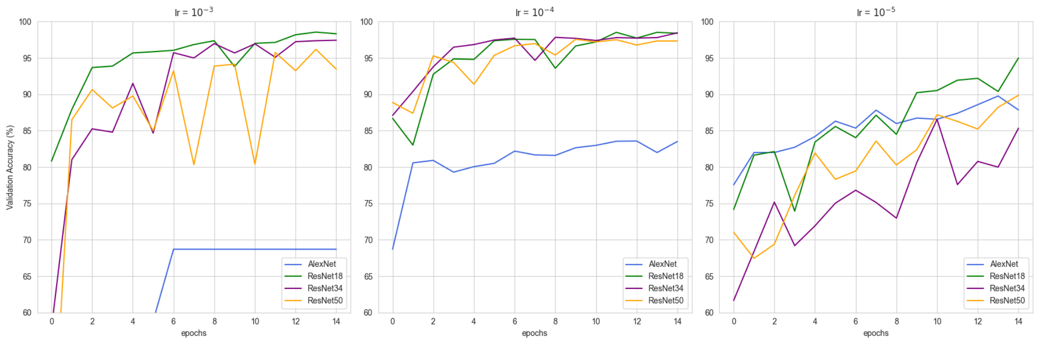


Figure 16: Validation accuracy of AlexNet, ResNet18, 34, 50 for different learning rates.

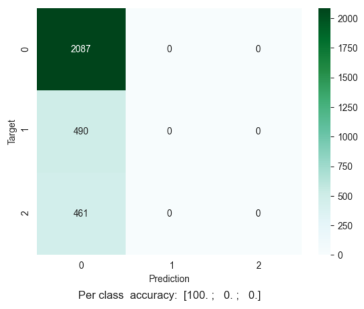


Figure 17: AlexNet validation saddle point confusion matrix.

decay of the L^2 regularization parameter was excluded to focus on the impact of the selection of the learning rate.

A learning rate of 10^{-4} was selected for most of the tests in Section 5.2 as it demonstrated the best performance and reproducibility. The middle graph in Figure 16 provides insight into why this learning rate was chosen. All networks perform well with this learning rate, achieving high accuracy after a few epochs and gradually increasing accuracy over time.

When using a large learning rate of 10^{-3}, ResNet34 and ResNet50 show more unstable behavior, with spikes up and down, but they still manage to converge in the end. On the contrary, ResNet18 remains stable under these conditions.

The third graph illustrates that a small rate of 10^{-5} does not provide sufficient updates for the residual models to converge quickly. The accuracy growth rate decreases with model depth, with ResNet50 exhibiting the slowest growth rate. The slopes of the curves suggest that all residual models would achieve higher accuracy with longer training times.

AlexNet does not perform well for any of the three learning rate choices. In the first graph, the network reaches an accuracy of 68.7% before encountering a saddle point issue where the optimizer becomes trapped without connections to the outputs of two other targets. As a result, all outputs are classified as target 0. The confusion matrix in Figure 17 depicts this problem.

When the learning rate is set to 10^{-4}, only two targets receive updates, as observed in Figure 22 with the accuracy analysis per class. With a high learning rate, the model quickly exceeds its capacity to capture and learn new features. Since the model is too small to capture all the details, the learning rate pushes the optimizer in the direction of the major class. Even if the first and second targets are perfect classified, the accuracy for this learning rate is always less than 85%.

For the last learning rate of 10^{-5}, the model finally starts learning the third target. However, the learning rate is too small for an effective update. In Figure 18 AlexNet was trained for the double number of epochs for the learning rate 10^{-5} and both the second and third targets barely reaching 80% validation accuracy each.

While the dynamics of residual models allows them to gradually improve their accuracy on minor targets, AlexNet reaches its limits in capturing details, similar to a strainer filled with larger particles from the first class. Figure 18 and Table 7 highlight an interesting phenomenon discovered during testing: An increase in the accuracy rate of target 1 corresponds to a decrease in the rate of target 2, and vice versa.

Target 1	43.88	47.14	51.22	71.43	54.9	67.55	48.78	85.51
Target 2	62.26	70.07	65.73	42.95	60.95	45.77	72.89	36.44

Table 7: Selected target 1 and 2 class accuracy rates of AlexNet from Figure 18 (epoch 7-15).

This fact is not only visually observed but also mathematically demonstrated when calculating the correlation coefficient, [11], of two accuracy vectors, resulting in a value of -0.17.

```
import pandas as pd
x = pd.Series(per_class_accuracy_AlexNet_30epochs[:,1].detach().numpy())
y = pd.Series(per_class_accuracy_AlexNet_30epochs[:,2].detach().numpy())
x.corr(y,method = 'pearson')

-> -0.1674841995537894
```

One potential explanation for this could be that the model's capacity is exhausted by target 0, creating a barrier that cannot be overcome in terms of accuracy. When calculating losses for the current epoch and adjusting weights towards the minor class, the optimizer pushes the model towards one class, resulting in a loss of accuracy for the other class due to changes in convolution kernels.

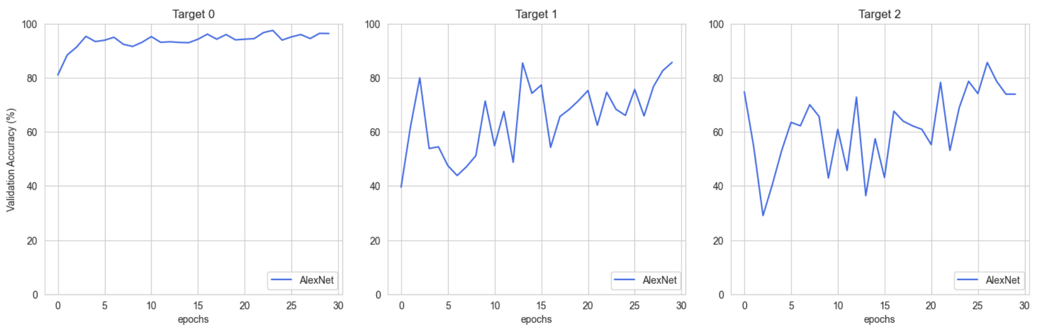


Figure 18: Per class accuracy of AlexNet when trained for 30 epochs.

In conclusion, residual networks exhibit greater flexibility when dealing with changes in learning rates and dataset imbalance, with ResNet18 showing the best adaptability to unfavorable learning rates. The number of parameters plays an important role in this scenario. Smaller learning rates allow the optimizer to fine-tune fewer parameters effectively, while larger learning rates can still quickly recalculate gradients over the next batch in case of overshooting.

Robustness with respect to initialization. In PyTorch, the framework takes care of initializing the values inside the kernel of the convolution block of residual networks. If the weights are not specified, PyTorch samples them from a uniform distribution $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where the value of k is proportional to the inverse of the input channels times kernel size, as stated on the PyTorch website [13, section Variables].

During the seminar, it was suggested to improve the performance of ResNet152, which had previously failed to train, by using He-initialization or kaiming-normal weights initialization [14]. This technique involves sampling weights from a normal distribution $\mathcal{N}(0, \text{std}^2)$ with a mean of 0 and a standard deviation that is by default proportional to the inverse square root of the input channels. During this section the "_init" subscript will indicate the normal-initialized version of the model; a model without the subscript indicates the default PyTorch uniform initialization.

The experiments on ResNet18, ResNet34 and ResNet50 showed very similar paths in validation accuracy rates during training, making it unnecessary to highlight the differences. However, ResNet152 and AlexNet showed significantly different results.

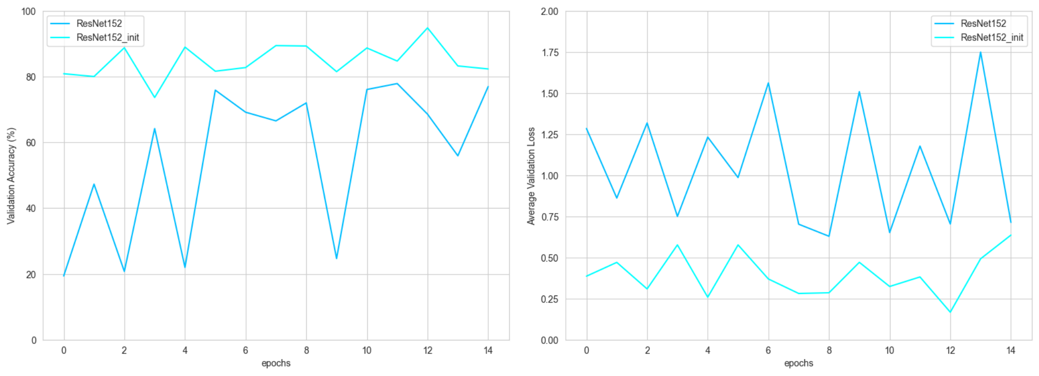


Figure 19: Validation accuracy and loss of ResNet152 and ResNet152_init.

Figure 19 compares the validation loss, accuracy and mean train loss of the ResNet152 and ResNet152_init. Training of this model was found to be unstable, resulting in a significant loss of performance and a lack of convergence in the validation error profile. Although one can observe an overall trend of increasing validation accuracy and decreasing error during training, the uninitialized version does not offer assurance of successful training in the end. Normal initialization smooths the profile, but it does not work miracles and the model still has convergence issues, eventually achieving a test accuracy of 82.52%, as shown in Figure 20. The train loss profile has also been improved.

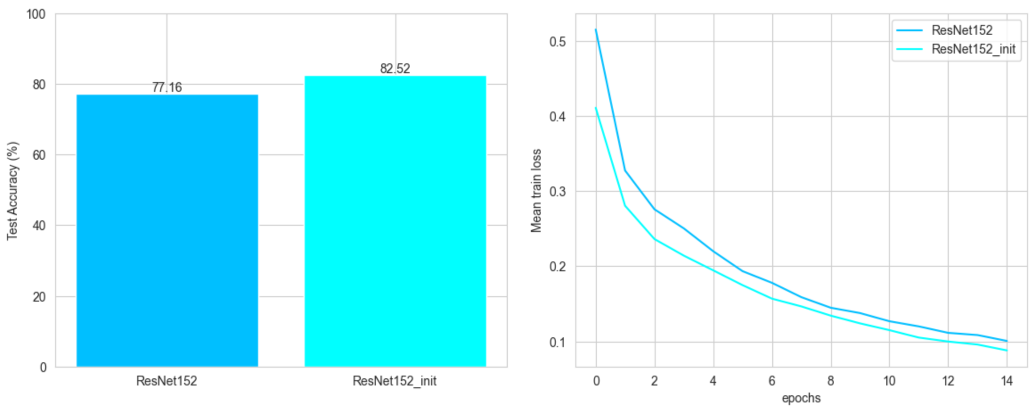


Figure 20: Test accuracy and mean train loss of ResNet152 and ResNet152_init.

However, the uniform initialized AlexNet suffers from convergence problems, as the model fails to connect the output of the third target, as illustrated on the right side of Figure 22., resulting in a test accuracy of only

83.54%. In contrast, Figure 21 shows that the normally initialized AlexNet achieves a validation accuracy of 90% after the seventh epoch and a test accuracy of 95.85% , representing a substantial improvement over the uniform initialized case.

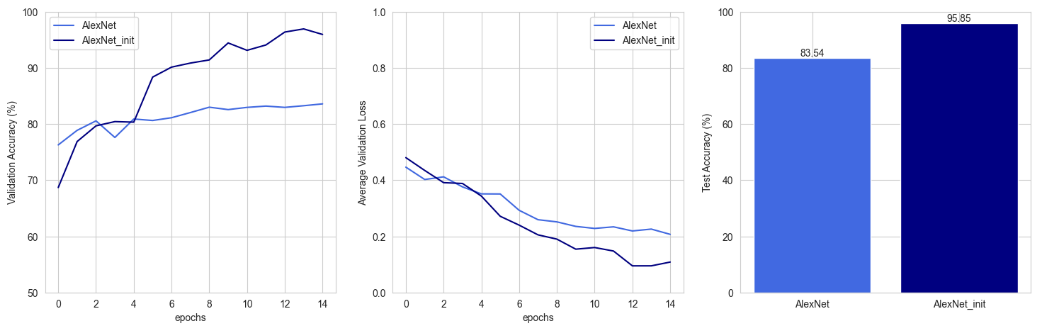


Figure 21: Validation accuracy and loss of AlexNet and AlexNet_init.

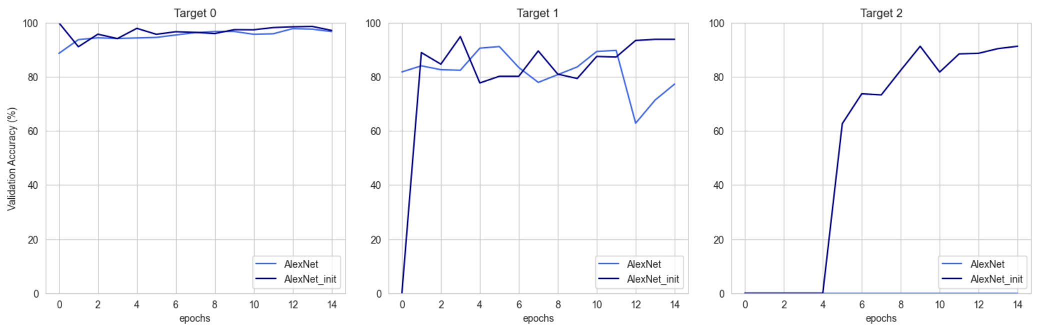


Figure 22: Per class accuracy of AlexNet and AlexNet_init.

Through the experiment, it was demonstrated that using the normal initialization method can provide numerous benefits for the model's performance, such as improved efficiency of the learning process, normalized, lower and smoother train error profile, and a solid starting point for optimization algorithms to identify a suitable set of weights. More examples, including test accuracy and mean test loss for each model, can be found in the Git repository [18].

Fixed time. Setting a time limit for evaluating the performance of a deep learning network at different intervals can provide valuable insight into its efficiency and effectiveness. These time limits are particularly relevant when considering the trade-off between model size and training duration.

Clearly, the runtime of the tested models varies depending on the number of parameters that need to be updated by the optimizer. Following the tests, a question arises: could it be more advantageous to train a smaller model for more epochs, particularly in situations where GPU resources are limited and training time must be fixed? In such cases, which network would yield greater benefits?

By imposing time limits of 10, 15, and 20 minutes, we can observe different behaviors in the models. The smaller and consequently faster model would be capable of completing more epochs in the given time, potentially leading to a better understanding of the dataset. However, the deeper model has the theoretical advantage of capturing more details.

	10 min	15min	20min
		lr $= 10^{-3}$	
AlexNet	68.70	68.70	68.70
ResNet18	98.06	98.12	97.73
ResNet34	89.30	93.15	95.52
ResNet50	90.95	94.21	93.98
		lr $= 10^{-4}$	
AlexNet	80.09	80.58	81.27
ResNet18	97.76	97.83	98.55
ResNet34	97.20	96.68	97.24
ResNet50	95.00	96.94	97.10
		lr $= 10^{-5}$	
AlexNet	78.97	80.88	80.68
ResNet18	86.18	91.94	94.01
ResNet34	88.31	82.26	92.33
ResNet50	69.03	76.00	78.24

Table 8: Test accuracy fixed time.

To address this problem, modifications were made to the train scripts. The elapsed time is compared with the time limits (10, 15, and 20 minutes) at the beginning of each epoch and the evaluation on the test data set is executed accordingly. The test accuracy of AlexNet, ResNet18, ResNet34, and ResNet50 after a fixed period of time is shown in Table 8.

As in previous tests, AlexNet experienced convergence problems, particularly with a high learning rate of 10^{-3}, where it fell into the saddle point problem. With lower learning rates of 10^{-4} and 10^{-5}, the model reached a performance plateau of around 81%, and longer training time did not significantly improve its performance.

Residual models, except for ResNet50, demonstrated robustness and flexibility in achieving high accuracy rates from the early epochs. However, due to its longer training time, ResNet50 completed only half of an epoch compared to ResNet34. Additionally, with a very low learning rate of 10^{-5}, ResNet50 achieved a test accuracy of only 78%, which is the lowest among residual models (excluding non-convergent ResNet152).

In summary, when computational resources are limited, ResNet18 is the optimal choice, as it offers a favorable balance between high accuracy rates and training speed. For larger datasets, opting for ResNet34 would be more sensible, given its nearly twice the depth, which allows it to capture more information.

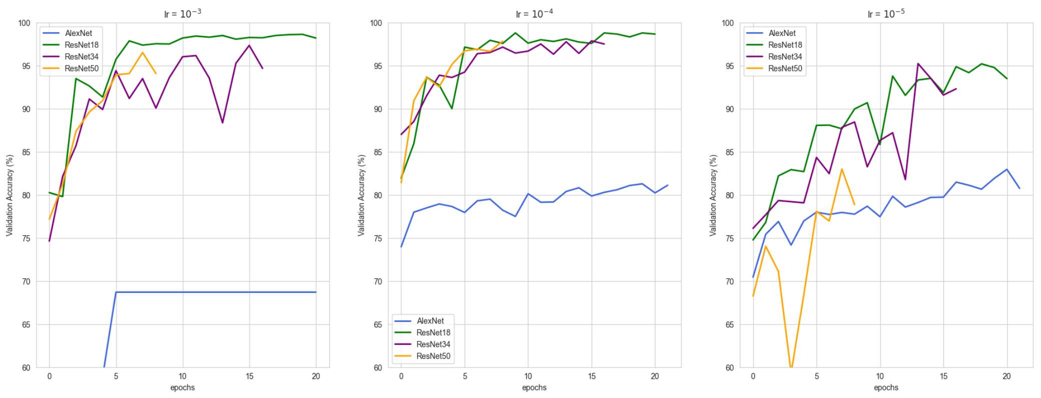


Figure 23: Validation accuracy (in %) after 10, 15 and 20 min. elapsed.

	Target 0		**Target 1**		**Target 2**	
	Dice 0	**IoU 0**	**Dice 1**	**IoU 1**	**Dice 2**	**IoU 2**
0	99.44	98.88	72.63	57.82	29.23	18.18
1	99.61	99.23	80.47	68.69	46.55	32.54
2	99.72	99.45	85.21	75.57	65.32	53.03
3	99.88	99.77	91.14	84.63	70.34	57.7
4	99.9	99.8	93.24	87.58	76.48	64.39
5	99.91	99.82	93.38	88.17	71.96	59.18

Table 9: U-Net validation evaluation.

Overall segmentation results. The U-Net was trained for 6 epochs using pixel-wise cross-entropy loss, the Adam optimizer algorithm, and a default learning rate of 10^{-4}. Each epoch took approximately 460 seconds to complete due to the strain on the processor cores, imposing a limitation on the training duration. The results of the evaluation, including the Dice coefficient and IoU scores (Eq. (4.7)) for three classes, are presented as percentages in Table 9 and Figure 24. It is evident that the dataset imbalance significantly affects the segmentation problem, as the evaluation rates for targets 1 and 2 are relatively low. On the contrary, the binary mask contains significantly fewer pixels identified as cancer compared to the liver or background pixels.

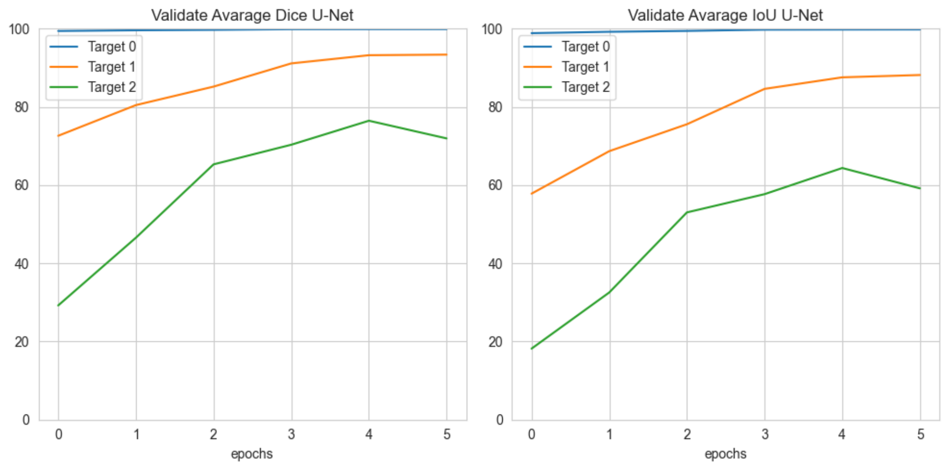


Figure 24: Validation accuracy U-Net.

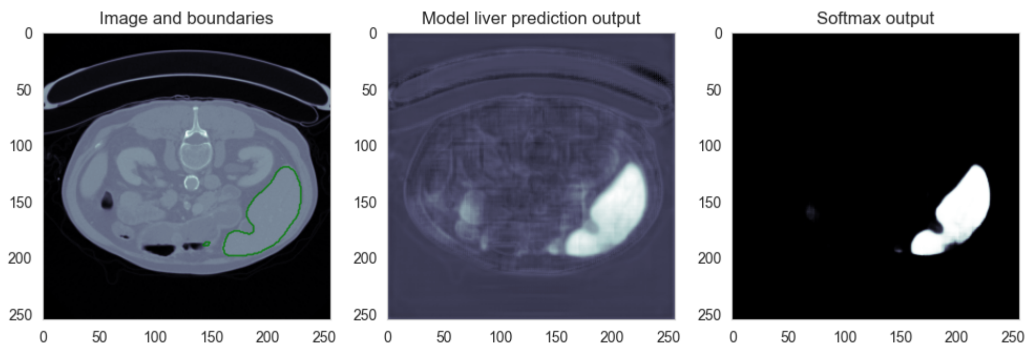


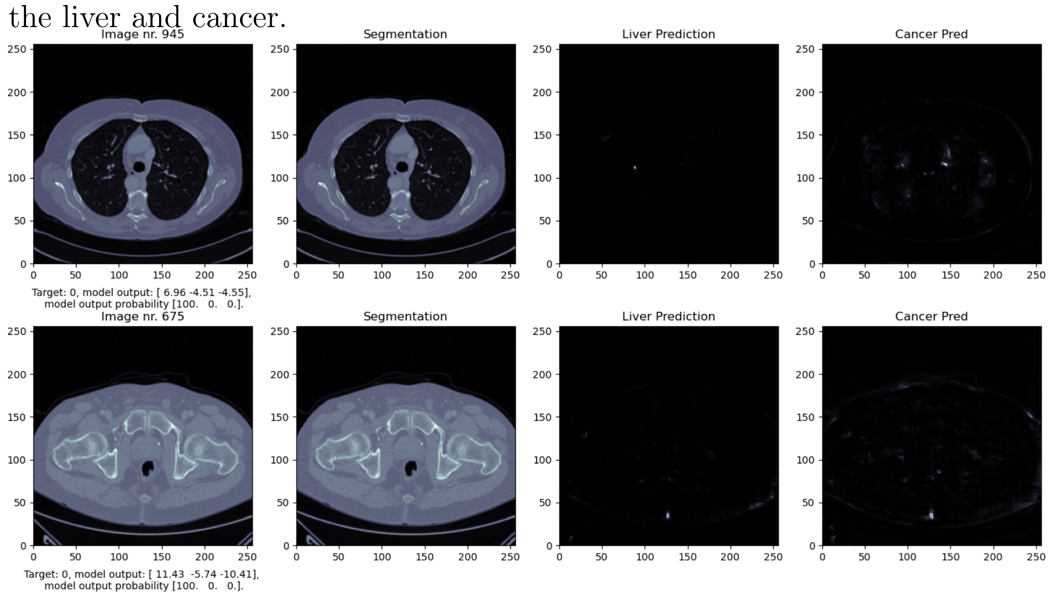
Figure 25: U-Net output with a softmax.

In Figure 25, the left image presents a sample of the test dataset, with the liver's target boundary highlighted in green. The middle and right images show the direct output of the U-Net (liver prediction) and the same output with the applied softmax function, respectively. The utilization of the softmax layer in the visualization enhances the comprehension of the segmentation, which is why plots with the softmax layer are consistently presented.

6 Gallery

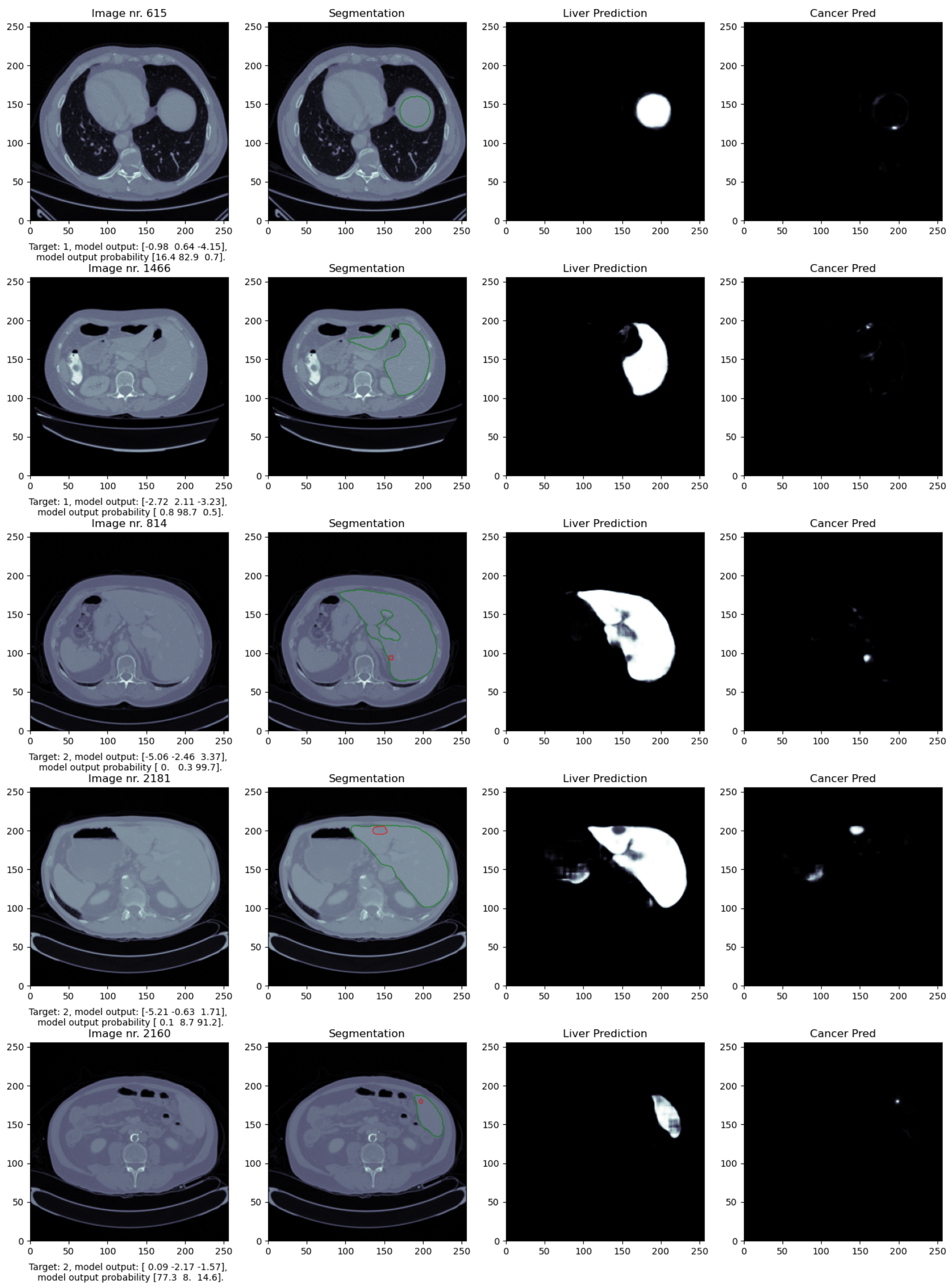
This section presents different classified and segmented images taken from experiments performed in Section 5 and focuses on the illustration of various medical images.

The image on the left side displays the target, classification output and softmax probability vectors (in %). The Section 3 contains a list of target values for reference. The index corresponding to the highest value in the model's output vector indicates the predicted class for the input. In the first image number 945, for example, the value of 6.69 corresponds to index 0, which is the same as the target. By applying the softmax function to the output vector, a probability vector is obtained. In this example, the model is 100% confident that the input belongs to the first class, which represents the target number 0 (representing a non-visible liver). These results were obtained using a ResNet34 model trained for 15 epochs with default hyper-parameters. The second image on the left represents the input given to the U-Net model described in Section 5.3, with the target segmentation highlighted by a green border for the liver and red for cancer. The remaining two images show the output of the U-Net, revealing the predicted segments for the liver and cancer.

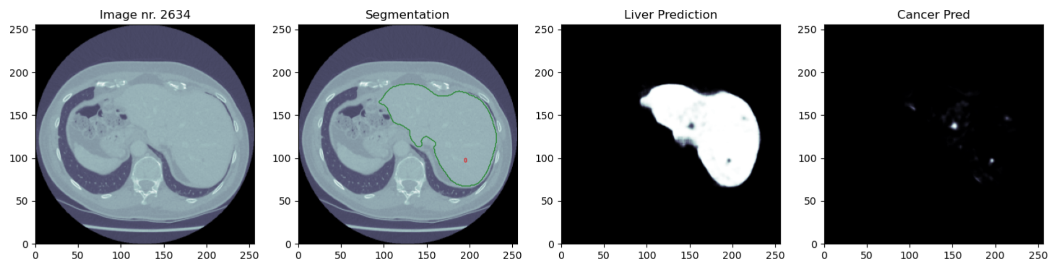


```
Target: 0, model output: [ 6.96 -4.51 -4.55],
  model output probability [100.   0.   0.].
```

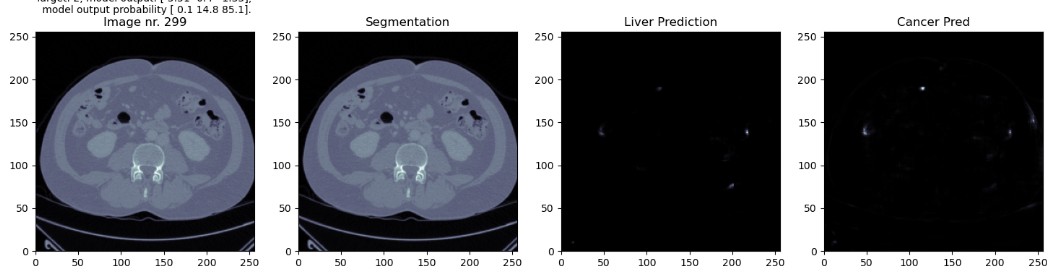
```
Target: 0, model output: [ 11.43  -5.74 -10.41],
  model output probability [100.   0.   0.].
```



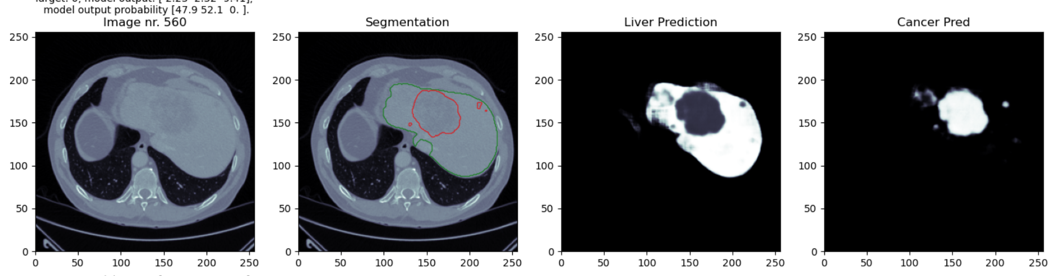
```
Target: 1, model output: [-0.98  0.64 -4.15],
  model output probability [16.4 82.9  0.7].
Target: 1, model output: [-2.72  2.11 -3.23],
  model output probability [ 0.8 98.7  0.5].
Target: 2, model output: [-5.06 -2.46  3.37],
  model output probability [ 0.   0.3 99.7].
Target: 2, model output: [-5.21 -0.63  1.71],
  model output probability [ 0.1  8.7 91.2].
Target: 2, model output: [ 0.09 -2.17 -1.57],
  model output probability [77.3  8.  14.6].
```



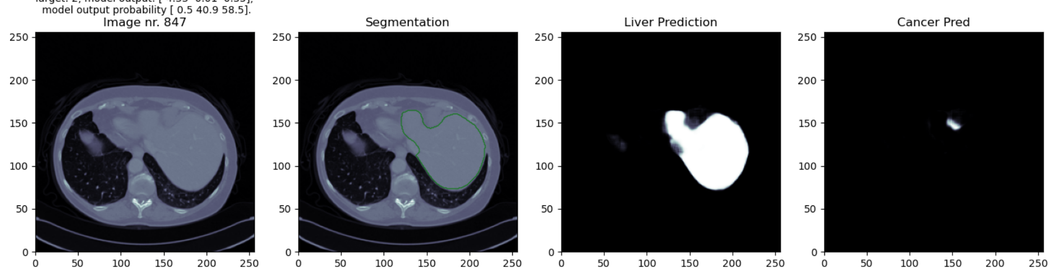
Target: 2, model output: [-5.51 -0.4   1.35],
model output probability [0.1 14.8 85.1].



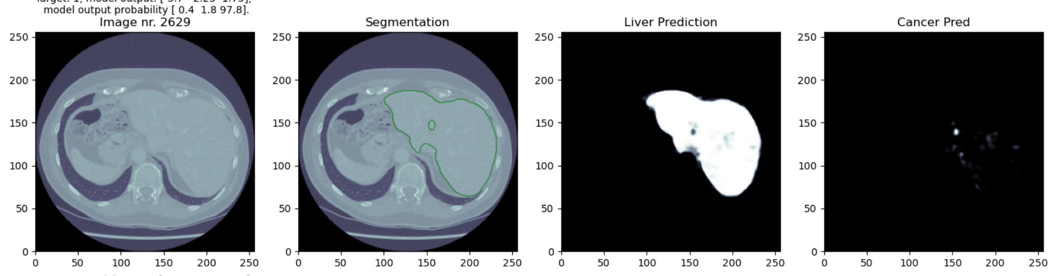
Target: 0, model output: [2.23 2.32 -9.41],
model output probability [47.9 52.1 0.].



Target: 2, model output: [-4.33 -0.01 0.35],
model output probability [0.5 40.9 58.5].



Target: 1, model output: [-3.7 -2.23 1.75],
model output probability [0.4 1.8 97.8].



Target: 1, model output: [-5.47 -1.37 2.37],
model output probability [0.   2.3 97.7].

7 Summary and conclusion

In conclusion, this thesis has explored various aspects of the application of deep neural networks in computer vision tasks. Different architectures, including convolutional, residual neural networks, and U-Net, were explored to understand their efficacy in handling complex visual data. Through experimental evaluations and practical implementations, we gained insight into the strengths and limitations of these models.

In general, the residual models exhibited higher flexibility and robustness compared to classical convolutional networks. Among them, ResNet18 performed exceptionally well, demonstrating significant convergence rates and fast training. In a comparison between high performing VGG19 and ResNet18, which have similar number of layers and convergence rates, ResNet18 outperformed VGG19 in terms of training speed. This outstanding performance of ResNet18 can be attributed to the size of the dataset and the number of criteria required for effective classification, especially when dealing with only three target classes. Therefore, for future real-life applications that involve a few observed classes or simple sample characteristics, such as document detection automation or disease diagnosis, ResNet18 appears to be a prominent choice. It is easy to train, has a smaller number of parameters, and can be readily deployed on low-computation-power portable devices.

However, it is important not to overlook the deeper update provided by ResNet34, which exhibits a comparable convergence speed but possesses almost twice as many parameters and capturing abilities. ResNet34 strikes the best balance between speed and the ability to capture intricate details, with ResNet18 being the primary candidate mainly due to its speed.

AlexNet showed an example of an outdated network, which in our study indicates that its model capacity may be insufficient for addressing the problem at hand. The correlation observed between the target validation accuracy rates for AlexNet with 30 epochs suggests an interesting phenomenon that may also be present in ResNet18 when dealing with much larger datasets, such as ImageNet with 100 classes, and needs further examination.

References

[1] Patrick Bilic et al. *The Liver Tumor Segmentation Benchmark (LiTS)*. 2019. arXiv: 1901.04056 [cs.CV]. URL: https://www.kaggle.com/datasets/andrewmvd/lits-png.

[2] Xingdong Chen et al. "Non-invasive early detection of cancer four years before conventional diagnosis using a blood test". In: *Nature Communications* 11 (June 2020). ISSN: 2041-1723. DOI: 10.1038/s41467-020-17316-z. URL: https://doi.org/10.1038/s41467-020-17316-z.

[3] Steven B. Damelin and Willard Miller Jr. *The Mathematics of Signal Processing*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2011. ISBN: 9781107013223. URL: https://books.google.de/books?id=MtPLYXQ9d9MC.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. eng. Adaptive computation and machine learning. http://www.deeplearningbook.org. Cambridge, Massachusetts ; London, England: The MIT Press, 2016. ISBN: 9780262035613.

[5] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[6] *Institute for Artificial Intelligence in Medicine*. URL: https://mml.ikim.nrw/ (visited on 01/22/2022).

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: (2012). URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[8] *Liver*. URL: https://www.encyclopedia.com/medicine/anatomy-and-physiology/anatomy-and-physiology/liver (visited on 01/22/2022).

[9] *Liver cancer*. URL: https://www.encyclopedia.com/medicine/diseases-and-conditions/pathology/liver-cancer (visited on 01/22/2022).

[10] *Liver Cancer Stages ACS*. URL: https://www.cancer.org/cancer/liver-cancer/detection-diagnosis-staging/staging.html (visited on 01/22/2022).

[11] *pandas.DataFrame.corr, Pairwise correlation of columns*. URL: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html (visited on 01/22/2022).

[12] *PyTorch*. URL: https://pytorch.org/ (visited on 01/22/2022).

[13] *PyTorch, Conv2D*. URL: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d (visited on 01/22/2022).

[14] *PyTorch, He-initialization*. URL: https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_normal_ (visited on 01/22/2022).

[15] *PyTorch, Transposed Convolution*. URL: https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html (visited on 01/22/2022).

[16] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[17] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (2014). DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556.

[18] *Thesis Git repository*. URL: https://git.uni-due.de/skolbaku/residual-neural-networks-for-medical-image-processing-a-case-study-in-liver-cancer-diagnostics (visited on 03/10/2023).

[19] Aston Zhang et al. *Dive into Deep Learning*. arXiv, 2021. DOI: 10.48550/ARXIV.2106.11342. URL: https://arxiv.org/abs/2106.11342.

[20] Donglin Zhuang et al. "Randomness In Neural Network Training: Characterizing The Impact of Tooling". In: *CoRR* abs/2106.11872 (2021). arXiv: 2106.11872. URL: https://arxiv.org/abs/2106.11872.

Declaration by the candidate

I solemnly declare under oath, through my signature, that I have independently and without external assistance prepared the above-mentioned work and have clearly indicated all passages that I have taken literally or nearly literally from publications, and that I have not used any other literature or resources than those mentioned.

The work has not been submitted to any other examination authority in this or a similar form.

Essen, 06.06.2023 Oleh Bakumenko