Using Deep Convolutional Networks to Regress a C-arm's Position from a single X-ray Image

Bachelor Thesis

Maternus Herold

Department of Statistics
Ludwig-Maximilians-Universität München
16.01.2020

Declaration

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Munich, 16.01.2020: .....
Maternus Herold

Acknowledgements

I want to thank my supervisors Dr. Fabian Scheipl and Prof. Antony Hodgson for their support and advice during the process of establishing this thesis. I am lucky to have had supervisors responding so quickly to questions and providing feedback on the different subjects I was working on. This clearly helped me writing this thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Surgeons use preoperative planning to investigate the interior of the patient with a focus on the surgical procedure itself, e.g. the details of a broken vertebrae and how to fixate it. For those plannings a preoperative obtained CT-scan is used to observe the patient in 3D space [7][26][11][19].

Also, in the advent of fluoroscopy guided procedures images obtained during surgery support the surgeon's navigation while there is no clear view on the scene. As an example one could imagine spine surgery where a vertebrae is fixated using screws. In that case the screw has very limited range of motion to not hurt adjacent organs, the spinal cord or other nerves. Further, the surgeon's view is occluded by tissue as the spine cannot be uncovered for a 360° view. Therefore, surgeons use fluoroscopy images to obtain real time feedback where the screw is going while drilling and being able to adjust [14].

The C-arm, a mobile fluoroscopy machine, is popular for such fluoroscopy guided procedures as it provides a wide range of motion and takes up little space in the operation room (OR). It is placed around the operation table in the form of a C, where it gets it's name from. See Figure 1.1 for illustration.

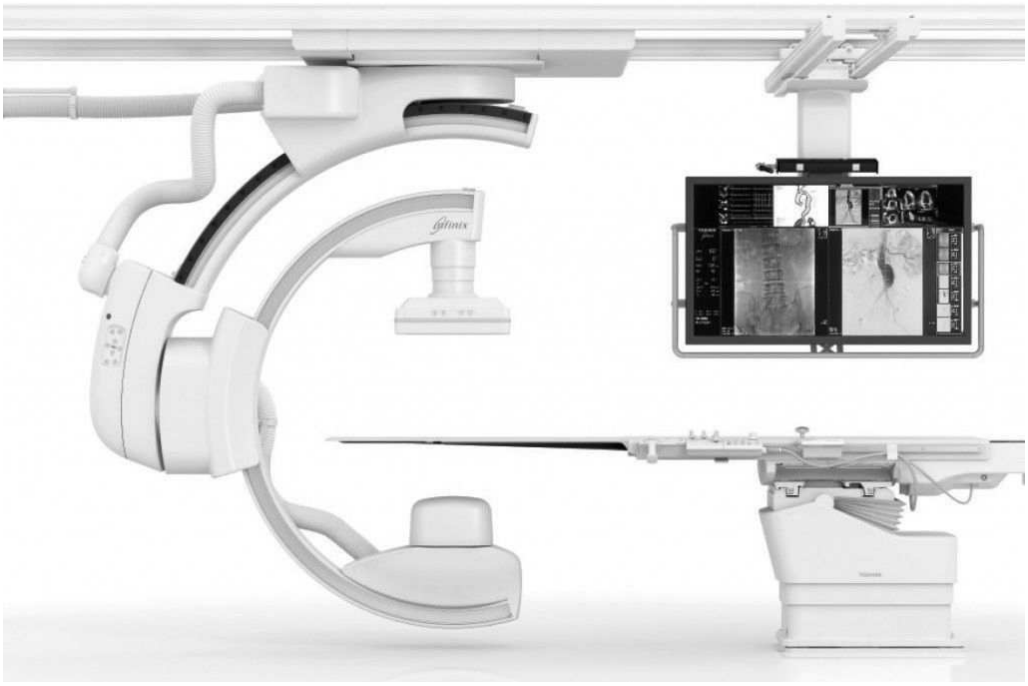


Figure 1.1: Ceiling suspended C-arm ranging around the operation table with many degrees of freedom; Model: Toshiba Infinix-I Sky + [18].

Those machines are set up in an iterative manner where a scout shot is taken from the patient on the operation table and the machine is then further adjusted. To reduce setup time in the OR I propose automating this in the following manner: first, the optimal view during the surgery is selected by the surgeon doing the preoperative planning on the CT-volume, specified by a six dimensional vector for rotation and translation. Also, based on the already existing CT-volume, a data set of digitally reconstructed radiographs (DDR) is computed in many different settings. Those images facilitate training data for a Convolutional Neural Network (CNN) which is trained to regress the six dimensional position vector a shot was taken from. Second, while the patient was placed on the operation table a C-arm is brought into position for taking a first orientation shot. Third, based on this shot the machine then locates itself w.r.t. the patient and calculates the needed movement to obtain the desired position. This is based on the first step which can be completely done in prior to the surgery. The proposed pipeline is illustrated in Figure 1.2.

This would reduce setup time in the OR by moving the computational steps to a non-time critical phase before the operation. A self calibrating C-arm would save work for the nurses and the surgeon which can then focus on more important tasks or be less time pressured. In addition the radiation exposure of both staff and patients would be reduced as well due to less shots taken to find the desired view.

This work covers step 1 including automatically generating a data set size of DRRs from a CT-volume consisting of high-resolution images and highly-accurate labels, building a model to regress the six dimensional position vector as well as analyzing the model's performance on a separated test set of DRRs, obtained from the same source CT-volume. The whole pipeline can be run automatically in sequence by providing a CT-volume and desired sampling ranges for the observation positions.

It was not possible to test this setup in a clinical case to inspect the model's ability to generalize on real fluoroscopy images. This is due the fact that I did not have access to an operation nor clinical data of needed extent.

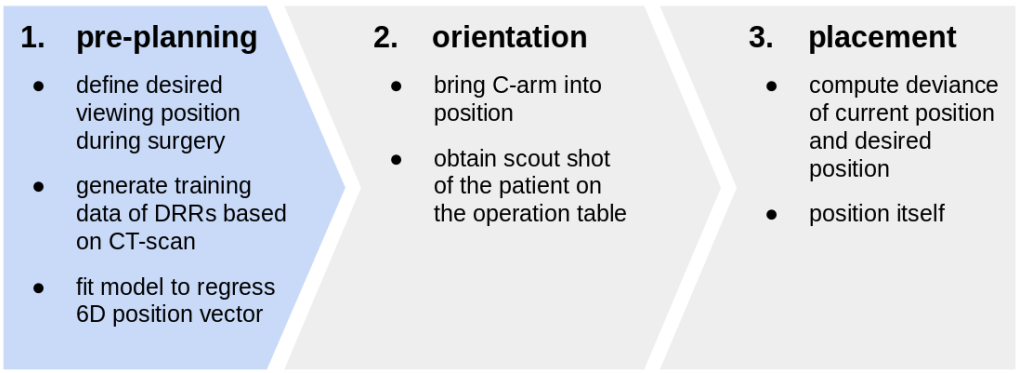


Figure 1.2: Overview of the proposed pipeline. In this work the first step is covered.

Chapter 2

Digitally Reconstructed Radiographs

Digitally reconstructed radiographs (DRR) are simulated radiographs computed as perspective projections of a three dimensional volume onto a two dimensional plane [7]. Usually the 3D volume is provided as a CT-scan. The synthetic generation of realistic X-ray images is important for medical 3D to 2D registration, which is used e.g. for surgery planning, inter-operative therapy guidance, patient placement for radiography planning and treatment verification and many other treatments requiring visual observation [7].

The use of DRRs can be motivated with a use case from radiotherapy. In cancer therapy the patient is exposed to a very high dose of radiation which has to hit the target to be effective but also must not hit healthy tissue to prevent damage. To locate the target, a pre-treatment CT-scan is used for Radiography Treatment Planning (RTP). This includes decisions regarding patient placement on the treatment table, radiation intensity as well as other technical details. As the setup depends on the patient's position relative to the emitter and is defined using a pre-obtained CT-scan, efficient positioning is of great interest. For prostate cancer treatment, for example, 35 daily fractions are necessary [26], making the patient positioning a task of high frequency. A popular solution to reduce setup time uses Portal Imaging which is often used in combination with immobilization techniques like foam beds. Films on the beam-exit side of the patient enable verification of positioning by comparing the obtained portal image with a reference, see Figure 2.1. Thus, positioning errors can be detected and quantified. The detection however, is done manually by visual inspection making it time consuming and inaccurate. Electric Portal Imaging Devices allow immediate generation of such films and produce digital images. Therefore, an automated comparison would be obvious [26]. Using DRR techniques, the expected Portal Image can be generated during the RTP and be easily compared to the obtained image, as both are digital and of same size.

Adding to those treatments, computer vision applications are becoming popular in the medical field due to their performance improvement in the last decade [30][31]. The mainly supervised algorithms require vast amounts of sample data consisting of observations paired with highly accurate labels. In the field of histopathology or mammography, X-ray images showing the tumor are annotated with a region of interest (ROI) indicating the tumorous tissue [28]. An algorithm, e.g. similar to the Convolutional Neural Network (CNN) presented in section 3.2, can then learn the characteristics of such tissue.

This contains two difficulties as an X-ray is taken by visual adjustment of the object or the emitting/receiving sources and the needed accuracy in labels is therefore not obtained. In the case of DRRs the annotation and further processing can be either performed in the CT-volume or during the configuration of the DRR algorithm [31]. Increasing the label's accuracy and reducing the burden of labeling each image individually or tracking the

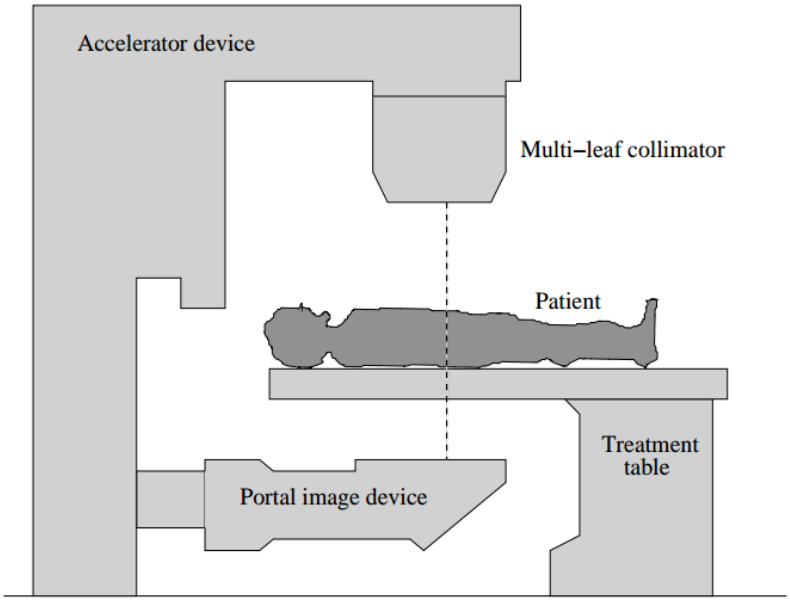


Figure 2.1: Setup of a Electrical Portal Imaging Device beneath the treatment couch capturing the patient's position for visual displacement analysis [26]. While patients can be immobilized using e.g. foam beds a displacement analysis and correction prior to treatment is necessary to assure effectiveness of the treatment.

machine's configurations. Second, exposing a human to radiation for a frequency needed for a decent machine learning test set is negligent. While a single X-ray image poses acceptable radiation exposure compared to its diagnostical value, constructing even a small data-set-size of scans would put the patient under great exposure. Assuming one could label the obtained images accurately and a data set of 1000 images is desired, which is at the far lower end of data set sizes in that category, the exposure on a human would be equivalent to the amount a human is exposed to in roughly 27 years of background radiation, according to [17] listed in Table 2.1.

$$1000 * \frac{10\text{d}}{365\text{d p. year}} = 27.4 \text{ years}$$

Clearly, establishing such a data set in vivo is not possible and also not feasible to use in clinical environments like RTP or others listed above due to its construction complexity.

Capturing a data set posthumously would drop the risk for a living object but would still pose risk to impure labels as well as limit the data set to a dead body ruling out the usage of such data in clinical treatments as the one discussed above or the one proposed in this work. This leads to the use of DRRs.

There are many different methods to generate DRRs but in all cases the generation is still the most time and computational intensive job for an application including DRRs. The basic principle relies on ray-casting techniques were the attenuation of a beam at the detector pixel is calculated by sampling and averaging the intensity values along the ray passing from source to pixel, see Figure 2.2. The intensity per sample is computed by trilinear interpolation of the eight closest voxels to the sampled point. This algorithm runs in $O(n^3)$ while n is the largest number of either DRR width, height or number of samples per pixel. Several optimizations come with the cost of lower quality of the resulting DRR or are based on impractical hardware specifications [7].

Examination	Radiation dose (mSv)	Time to accumulate comparable radiation with background dose
CT: Head	2.0	8 mo
CT: Chest	7.0	2 y
CT: Abdomen & Pelvis	10.0	3 y
X-ray: Chest	0.1	10 d
X-ray: Lumbar spine	0.7	3 mo
X-ray: Abdomen	1.2	5 mo
Mammography	0.7	3 mo

Table 2.1: Comparing radiation doses from medical imaging to background radiation a human is exposed to on average as an subset from what's presented in [17].

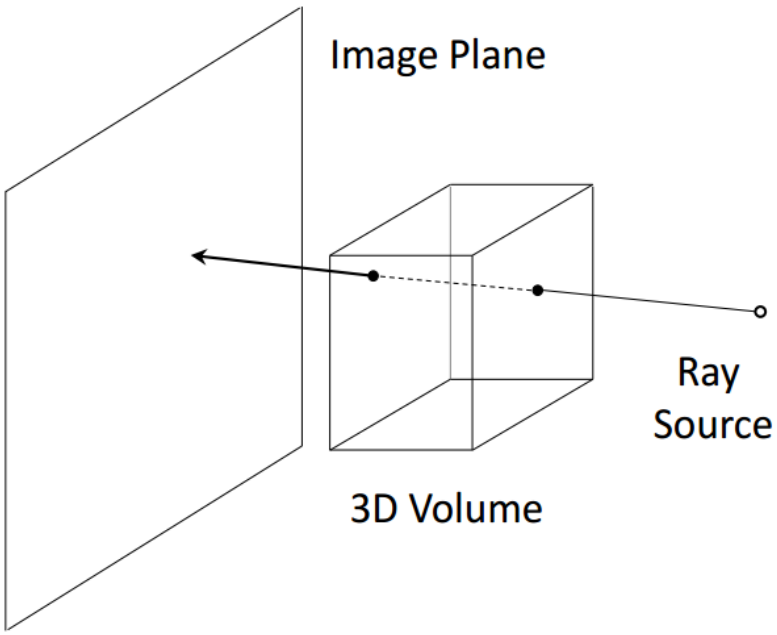


Figure 2.2: Ray-casting, a traditional way to generate digitally reconstructed radiographs, simulates the radiographic imaging system by setting the pixel value in the 2D output to the attenuation value of a ray passing from source through object to detector. While in real applications the attenuation is determined by tissue, bone and air in the scanned object it is approximated by sampling voxel intensities along the ray's path though the object [7]. In the above figure the 3D volume represents the CT-volume. Many applications use fixed values for the attenuation called Hounsfield Units (HU).

2.1 Deep Digitally Reconstructed Radiographs

The now presented framework is intended to overcome the mentioned obstacles in the usage of DRRs as well as to include machine learning compatibility. Mathias Unberath et al. focussed on efficiency as well as scalability of their framework [31]. Further, machine learning was used to overcome expensive computations to estimate stochastic artifacts as noise and scatter.

Secondly, machine learning algorithms fit on regular DRRs do not generalize well to clinical data as they do not capture a natural X-ray's image formation [31]. In their work, a generalization of models trained on DeepDRRs is shown. This lead to further consideration of the framework for this work.

According to Mathias Unberath et al. DRR algorithms can be grouped into analytical and stochastic algorithms. While the typical ray-tracing algorithm is an analytical one, algorithms using stochastic methods as e.g. Monte Carlo (MC) simulations are classified as stochastic algorithms. Ray-tracing algorithms are considerably efficient, compared to stochastic ones, but they often only consider one material's attenuation for the whole body and therefore fail to model specific artifacts as beam-hardening [31], where the edges of an object appear brighter than the center. Also, due to their nature, they're not capable of modelling statistical artifacts as noise or scatter which result from deflected photons hitting the receptor at random positions. In contrast, MC approaches model photon-matter interaction individually per photon using material decomposition as the interaction probability is material dependent [31]. The material decomposition is done with thresholding by the above mentioned Hounsfield Units (HU). While those DRRs are a lot more realistic, they also require far longer to generate. When using 10^{10} photons, a simulated image can take 4 hours to generate in an accelerated MC simulation on a graphical processing unit [2]. This is not plausible for a machine learning data set.

The Deep Digitally Reconstructed Radiograph framework consists of a four-step pipeline segmenting the volume into three materials $M = air, soft\ tissue, bone$, calculating the attenuation values at the receptor as well as estimating scatter and noise in the image. A schematic overview is shown in Figure 2.3.

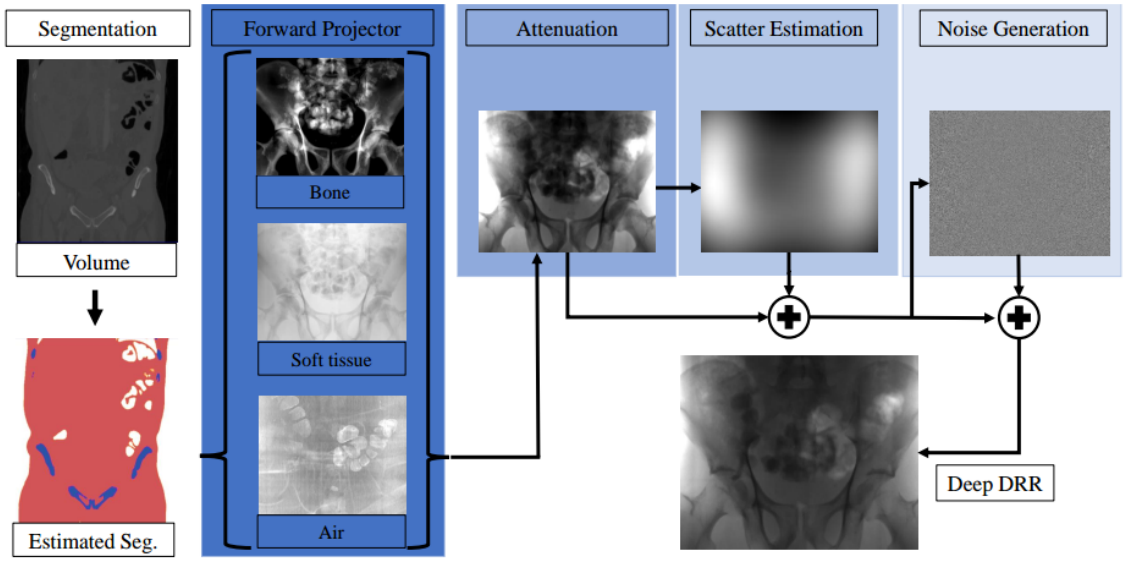


Figure 2.3: Overview of the DeepDRR pipeline including intermediate outputs. Especially on the steps of segmentation, scatter estimation and noise generation the authors hope to achieve more realistic while efficient outputs using machine learning [31].

For the first step the authors have adapted and trained a Convolutional Neural Network

on whole-body CT-scans for material decomposition. The output yields a material map $M(x)$ assigning a material to a point x in 3D space. The same effect can be achieved using thresholding on the HU while this is not guaranteed to work well on volumes with little discrepancy. While air ($[-1000]$HU) and soft tissue ($[200, 300]$ HU) are good to differentiate, the same does not hold for soft tissue and bone ($[-150, 300]$HU) [31] as they have a non-empty intersection of ranges. Also, both materials have different attenuation characteristics. A work around to still use thresholding would be to search for good thresholding values prior to segmenting with a visualization tool such as 3DSlicer [6].

Based on the segmentation, the contribution of each material to the total attenuation at position u of the detector is computed by defining the position of u with the projection matrix $P \in \mathbb{R}^{3\times4}$.

$$p(u) = \int p(E,u)dE \tag{2.1}$$

$$= \int p_0(E) exp\left(\sum_{m\in M} \delta(m, M(x)))(\frac{\mu}{\rho})_m(E) \int \rho(x) dl_u\right) dE \tag{2.2}$$

In the equation describing the attenuation density (2.2) $p_0(E)$ is the X-ray spectral density depending on the energy, $\delta(\cdot)$ the Kronecker delta comparing the current material being summed over to the one observed at the current position, $\frac{\mu}{\rho}_m(E)$ the material and energy dependent linear attenuation coefficient, $\rho(x)$ the material density and l_u the 3D ray connecting the source and the detector pixel u and is determined by the projection matrix P. The result $p(u)$ is used as input to the scatter estimation done with another CNN [31].

The third step in the pipeline is scatter estimation. Described above, this is a feature of stochastic algorithms and usually done using MC simulations. Maier et al. however, have shown that a CNNs outperform conventional methods for scatter estimation while retaining a low computational expense [4]. Therefore, the scatter estimation is done using a CNN of ten layers while the first layers generate scatter estimates and the following ensure smoothness [31]. The training data was provided by MC simulations with some augmentations. Again, the output of this step is shown in Figure 2.3.

To finalize the DRR, a combination of uncorrelated quantum noise due to photon statistics and correlated electronic readout noise is added. For the first one a Poisson-noise model is used while the latter is modelled with additive Gaussian noise [31].

With minor changes to it, the discussed framework was used to generate the data for this work. A first sample of the generated images is shown in Figure 2.4 depicting the pelvic area on an anterior-posterior (AP) shot.

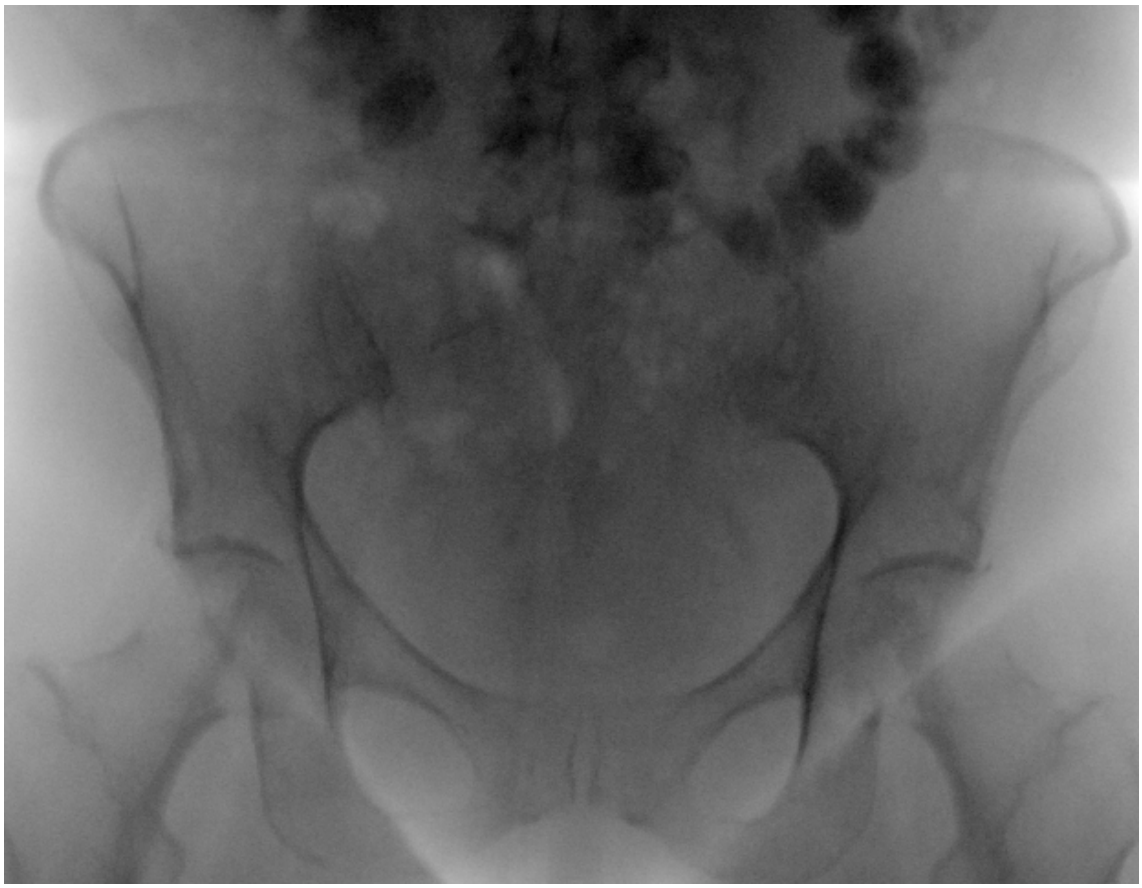


Figure 2.4: Example of a generated DRR using DeepDRR during data preparation for this work. The resulting image was generated with minor changes to the architecture and functionality of the described DeepDRR framework. All changes can be observed in the GitLab repository accompanying this work.

Chapter 3

Convolutional Neural Networks

Predicting the orientation the X-ray was taken from was approached with a Convolutional Neural Network (CNN), an architecture specialized on working on grid data. To understand CNNs the more basic model, the Multilayer Perceptron (MLP), has to be discussed. By doing so, the architectural differences and task-specific advances of a CNN are better understood.

A fundamental building block, as the name suggests, is the Perceptron. The Perceptron was initially defined as an algorithm in 1957 by Frank Rosenblatt to be implemented into hardware at the United States Office of Naval Research [23]. The algorithm was intended for image recognition and inspired by the human brain, but is nevertheless far away from the biological principle. The first hardware implemented Perceptron algorithm was the Mark 1, using a grid of 20 by 20 photocells randomly connected to the neurons. The weights were implemented mechanically as well as in form of potentiometers and the weight updates were carried out by electric motors. A topological overview of Mark 1 is given in Figure 3.1.

The Perceptron as discussed here and implemented today in software is a binary classifier learning a threshold function by building a linear function through addition of the dot product of the inputs x and weights w with a bias value. The linear combination is then compared to a threshold value. In case the threshold is exceeded, the unit fires. The weights and biases are usually learned by the model via iteratively comparing the computed output with a ground truth and adjusting the parameters accordingly. The threshold value can either be treated as hyperparameter, by defining it prior to training, or adjusting it by the training algorithm as the other value as well.

$$f_\theta(x) = \begin{cases} 1, & w' * x + b > \theta, \\ 0, & \text{otherwise} \end{cases}$$

The Perceptron algorithm as a combination of many threshold functions was not as powerful as it was hoped by its designer Frank Rosenblatt and others at the Naval Research center. It was only able learn linearly separable patterns. A few years later Marvin Minsky and Seymour Papert proved the algorithm's inability to even compute the XOR function [20]. This was then achieved by adding another layer of units of the threshold function. This will be discussed in the following section 3.1. The addition of more layers also provided the name of the architecture: Multilayer Perceptron.

In todays research, the Perceptron is often referred to as just the single unit computing the threshold function mimicking a biological neuron. To prevent confusion on the nomination, a single unit of computation will be referred to as unit in the following.

Although the way a unit computes its output was initially inspired by neuroscience it is now heavily influenced by mathematics and engineering to achieve statistical generalization.

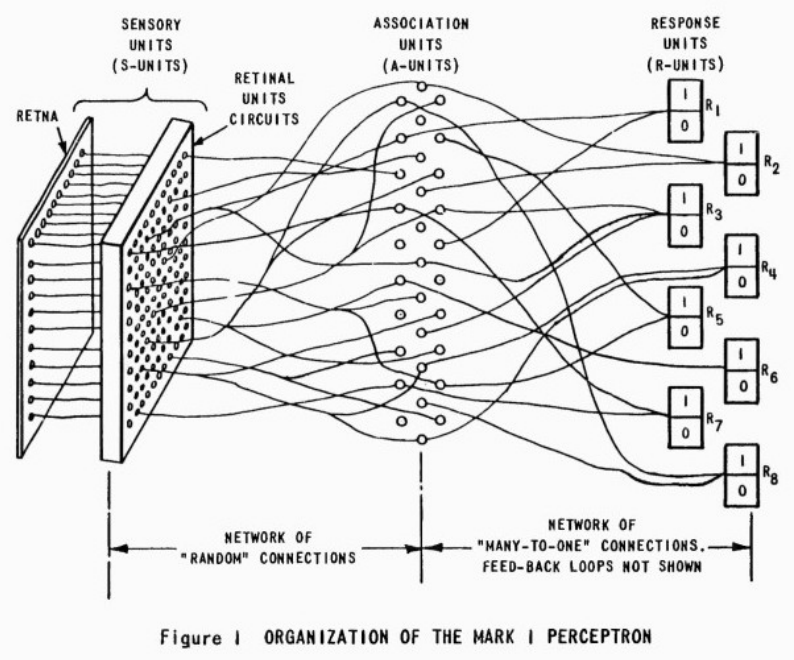


Figure 3.1: Topology of the Perceptron Algorithm implemented in hardware using a 20 by 20 array of photocells which are randomly connected to the artificial neurons [12].

3.1 The Multilayer Perceptron

The Multilayer Perceptron (MLP) derived from the single layer Perceptron algorithm as discussed above. It has to be understood as an algorithm, stacking together nonlinear functions in units. Those units are themselves grouped into layers. One layer receives the input from its predecessor or is itself the input layer, piping the input data into the algorithm. The layers following the input layer are called hidden layers as the user does not interact with those directly as the computations are not returned but piped to the next layer. The last layer is called output layer accordingly. A layer can be seen as a vector-to-vector function and its units as vector-to-scalar functions [9].

Given a network with a single unit per layer with n such layers, the algorithm's computation is described by the functions $f^{(1)}, f^{(2)}, \ldots, f^{(n)}$. As the last function uses the output of its direct predecessor as input, just as every other function's input is the preceding function's output the complete computation can be described by chaining the functions together yielding $f(x) = f^{(n)}(\ldots(f^{(2)}(f^{(1)}(x)))\ldots)$. This notation hints an important property, the density of a MLP. Density describes the property of connecting all units of a preceding layer to all units of the current layer while those units in turn are again connected to all units in the following layer. Such layers are often referred as dense layers or fully connected layers. Using the same notation on an architecture with more than one unit per dense layer, such a chained representation becomes to complex. Instead the topology is depicted as an acyclic graph connecting each unit with its input and output units through edges, Figure 3.2. Depicting the topology in such manner is responsible for the network attribute in the name of the algorithm class.

The acyclic graph characteristic implies the network only has forward connections. This is called a feed-forward architecture. Feedback connections in contrast would allow to include the output of a previous computation for the current or future computations. Those connections are common in architectures like the Recurrent Neural Network but are out of scope for this work.

Just as a linear model, a MLP tries to approximate a function $y = f^*(x)$ by elaborating

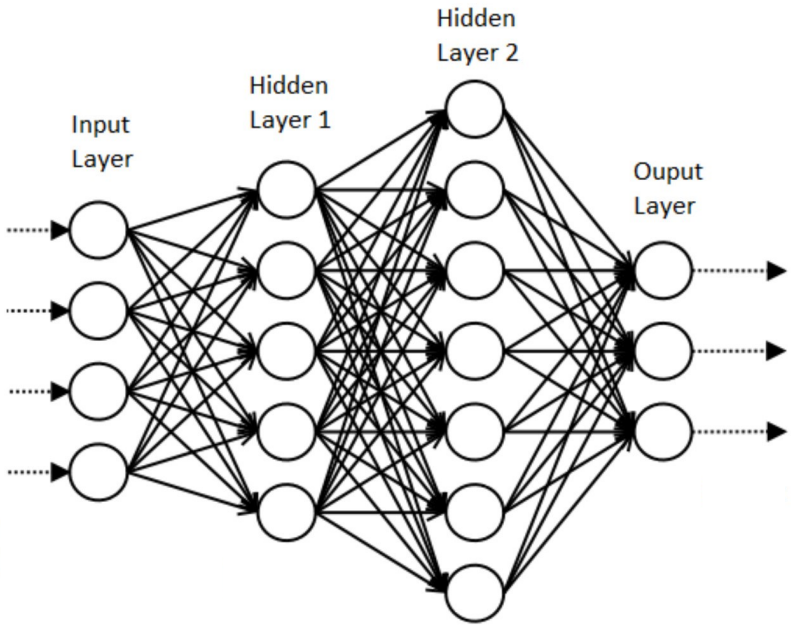


Figure 3.2: Topology of a model with an input layer and three dense layers forming a MLP [1]. The middle layers are called hidden as not directly observed by the user.

the best set of parameters θ which lead to a function $\hat{y} = f(x;\theta)$ best describing the actual data. It is important to note here that it's usually not the aim to interpolate the data points exactly as this would reduce the model's ability to generalize on new data. Just as it is with linear models.

To elaborate more on the advances of the MLP it shall be compared to a linear model. Linear models are in general nice to work with as they can be fit efficiently and are well interpretable by its parameters. However, the pure linear model is, by definition, limited to linear relationships in the data. To overcome this limitation, variables can be transformed by some function ϕ and the model then uses $\phi(x)$ instead of x. In this case ϕ can as well be a nonlinear function as $\phi(x)$ is included linearly into the model. This motivates the question how to find a proper function ϕ. Ian Goodfellow et al. [9] propose three main strategies to do so:

1. One could use a generic infinit dimensional ϕ that provides enough dimensions to map each data point into a se perate dimension, as it e.g. is the approach for clustering with Support Vector Machines. However, such a mapping will work well on training data but the model will most likely fail to generalize on new data as it overfitted the data.

2. Manually engineer ϕ, which is still popular for many algorithms. However, to do so the user must have deep domain knowledge to decide on feasible transformations and advanced mathematical and statistical knowledge.

3. Lastly, defining ϕ could be transferred to the model as it is the case for training Neural Networks. Given a function f with $y = f(x;\theta,w) = \phi(x;\theta)'w$, the model uses the parameters θ to learn the function ϕ while w is then used to map $\phi_\theta(x)$ to y. An optimization algorithm is used to minimize the function's deviation from its target. With this approach it's possible to either search a very broad function space (as in item 1) or to incorporate human experience to narrow down the function space by defining a general function family (as in item 2). Ultimately, the practitioner does not have to define the exact function but a broader function family.

In the same way as the linear model, the MLP achieves its nonlinearity by applying a nonlinear functions to the input. Similar as a model of the generalized linear model family applies a response function to the linear predictor, a unit in the MLP applies a nonlinear function to the linear combination of weights, inputs and biases. This nonlinear function is called the activation function and can differ per layer (and even per unit) and transformation to achieve. A common activation function is the rectified linear unit (ReLU) [9].

The output of one unit is then computed by the chain of functions $h = g(w'x + b)$ while g is a nonlinear activation function as f_{ReLU}.

To adjust the model's parameters, the prediction is compared to ground truth data in the supervised case. The deviation between both is valued with an error or loss function which shall be minimized to find the optimal set of parameters θ. A basic loss function can most often be derived from Maximum Likelihood Estimation as the model can be seen to approximate a distribution $p(y|x, \theta)$ based on the input data, the primary variable derived from. This approach is clarified in Theorem 3.1.1, as a multi-class derivation of the Binary Cross-Entropy (BCE) loss is used later in section 3.4.

Example 3.1.1. *Suppose a model is fitted on a classification task with binary output* $z \in \{0, 1\}$. *The output hereby follows a Bernoulli distribution* $p(y|\pi)$, *such that the distribution is defined by:*

$$p(y|\pi) = \prod_{i}^{n} \pi_i^{y_i} * (1 - \pi_i)^{1-y_i}$$

To estimate the parameter π *using Maximum Likelihood Estimation the Likelihood function* $L(\pi|y, x)$ *is maximized w.r.t. the parameter* π. *To do so the Likelihood of the distribution is converted to a* log*-Likelihood* $l(\pi|y, x)$ *due to monotonial and calculational properties of the* log.

$$\begin{aligned} L(\pi|y, x) &= \prod_{i}^{n} p_\pi(y|x_i)^{y_i}(1 - p_\pi(y|x_i))^{1-y_i} \\ l(\pi|y, x) &= \log(L(\pi|y, x)) \\ &= \sum_{i}^{n} y_i \log(p_\pi(y|x_i)) + (1 - y_i) \log(1 - p_\pi(y|x_i)) \end{aligned}$$

As loss functions are usually minimized instead of maximized, simply $l(\pi|y, x)$ *is converted to* $-l(\pi|y, x)$ *which is exactly the definition of the Binary Cross-Entropy loss function*

$$BCE(y|x, \pi) = -\sum_{i}^{n} y_i \log(p_\pi(y|x_i)) + (1 - y_i) \log(1 - p_\pi(y|x_i))$$

3.1.1 Properties of the ReLU activation function

As mentioned above, the ReLU function is the most popular activation function, as of 2017 [22], used in several network architectures such as the MLP or CNN. The latter is the reason it is discussed in more detail here. While the name is used interchangeably, the function is called the rectifier while the unit utelizing this function as its activation is the rectified linear unit.

The rectified linear unit's domain is the complete set of real numbers while the range is limited to the positive real numbers, including 0, depicted in Figure 3.3.

$$f_{ReLU}(x) = x^+ = max(0, x), x \in \mathbb{R}$$

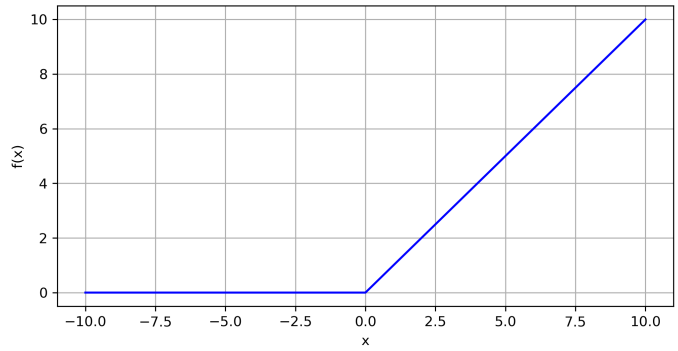


Figure 3.3: Illustration of the ReLU function with $x \in [-10, 10]$

The rectifier function is a biological feasible activation function in the field of computational neuroscience and performs well in machine learning environments. This is due to some of its properties such as sparsity, leading to linear separability, better gradient propagation and efficient computation [8].

Glorot et al. [8] show in their paper on deep sparse rectifier networks that the usage of the rectifier function leads to real zero activation probability of units instead of a very close to zero probability. A sparse representation has the advantage of describing information with less information as initially provided. Meaning redundancy has been removed by the model or better features have been created. This accompanies one of the claimed objectives of deep learning algorithms by Yoshua Bengio to simplify and reveal the factors for variations in data [3]. Also, as mentioned earlier in elaborating the function ϕ to transform a linear model's input, the information is represented in a higher space leading to better separability with less nonlinear methods [8]. However, sparsity has to comply with model performance as too much sparsity limits the model's capabilities. In addition, starting with a uniform initialization of weights on a symmetric interval 50% of the unit's outputs are 0 already. A further increase in sparsity can be achieved by the use of sparsity-induced regularization [8], which will not be covered.

A model's weights are usually updated using backpropagation which calculates the gradient of the error function w.r.t. its weights [9]. The name indicates the direction the algorithm takes to compute the gradients by starting with the weights of the output layer and ending at the input layer. By doing so the problem of vanishing gradients can arise, meaning a gradient becomes too small to change the according weight. This results in a saturated state of the model where it does not learn anymore. Especially in deep networks with many layers this poses a challenge [13]. The first derivative for the ReLU function is always either 0 or 1, depending on the input. Therefore, it does not saturate in contrast to e.g. the Sigmoid activation function, another common activation function, and the gradient does not vanish. Figure 3.4 gives a visual comparison of both functions and their derivative. Especially in the zoomed-in version in Figure b it can be observed how the derivatives of the Sigmoid function converge to 0 fast at both ends.

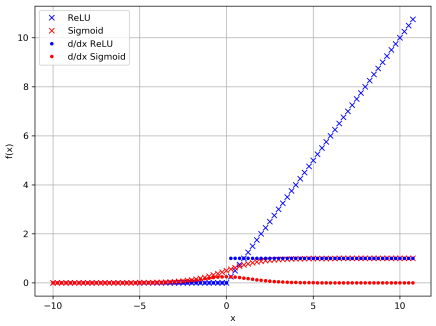
Remark 3.1.2. *The Sigmoid activation function is defined as*

$$\sigma(x) = \frac{1}{1 + \exp -x}, \quad x \in \mathbb{R}$$

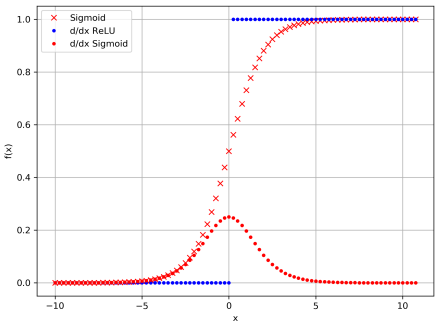
with a first order derivative of

$$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$$

Clearly, the $max(\cdot)$) function is computationally more efficient as most other activation functions and especially the Sigmoid function as the exponential function does not have to be computed.



(a) Depicting the function's ranges and first order derivatives



(b) Depicting the Sigmoid function and with derivative vs. ReLU derivative

Figure 3.4: Comparing domain and range of the Sigmoid and ReLU function with the first derivative

Even though the fitting process is different, training a MLP includes making design decisions similar to those for a linear model where one has to chose an optimizer (e.g. ordinary least squares vs. gradient descent), a cost function (e.g. the mean squared error) and the form of the output units which influences the model definition (binary, discrete or continuous - classification vs. regression).

Fitting a MLP will be demonstrated in a later part of this work.

3.2 Convolutional Neural Networks

Similar to the MLP, the Convolutional Neural Network is a feed-forward network building up on layers with connected units to make a prediction. Especially in the lower layers both architectures are quite similar. In contrast to the MLP the CNN utilizes the mathematical operation of convolutions to carve out features presented in the data. This is done instead of regular matrix multiplications. Those convolutions are usually used in blocks combined with an activation function, called detector stage, introducing nonlinearity, followed by the pooling operation. As an example the first two blocks of the VGG16 and the dense layers following the convolutional blocks are depicted in Listing 3.1. The single components will be explained in the following.

```
1  Model: "vgg16"
2  _________________________________________________________________
3  Layer (type)                 Output Shape              Param #
4  =================================================================
5  input_1 (InputLayer)         [(None, 224, 224, 3)]     0
6  _________________________________________________________________
7  block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
8  _________________________________________________________________
9  block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
10 _________________________________________________________________
11 block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
12 _________________________________________________________________
13 block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
14 _________________________________________________________________
15 block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
16 _________________________________________________________________
```

```
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0

...
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

Listing 3.1: Excerpt of VGG16 model summary [29]

Convolutional networks perform very well on grid-like structures and are used for time-series analysis and, most prominently, for image recognition tasks [9]. The latter is intuitive as images do have a grid structure described by their width, height and depth. Also, that is the reason this architecture was used to analyze the generated radiographs described in chapter 2. A performance based comparison is presented at the end of this chapter.

3.2.1 Convolution Operation

The convolution in mathematical terms is an operation on two functions, describing the change of the first one over time or in space when it is modified by the second one. By its definition it can also be explained as a weighted average where the second function weights the value of the first function while increasing towards the time of measurement t.

Definition 3.2.1. *Given two functions f, g on the same domain. The convolution of both functions until a time t is given by*

$$(f * g)(t) = \int f(\tau)g(t-\tau)d\tau$$

In the case of machine learning and especially CNNs the convolution is a discrete operation and can be expressed as a summation over time t. Also, the first argument, the first function, is referred to as the input to the convolution and the second argument as the kernel. The resulting function is called the feature map [9]. In the case of image recognition, the convolution is a operation over two or more axis - at least the width and height of the image. Defining an image as proper continuous function is out of scope for this work.

In image recognition applications of 2D convolutions the kernel or window slides over the input - the input image on the first layer or outputs of deeper layers - and computes the feature map by summing over the Hadamard product of the window and a same size submatrix of the image as depicted in Figure 3.5. Adjusting the kernel values is part of the model fitting process while there are as well predefined kernels for different kind of features. In general, kernels in the higher layers at the beginning of the network are used to detect low-level features such as edges of different styles. The deeper the convolutional layer, the higher-level or more abstract feature it detects [27]. Visualization of such layers will be carried out in the latter part of the chapter.

The convolution operation incorporates 2 important advantages over a simple MLP for image analysis. While an MLP utilizes dense layers as described in section 3.1 a CNN

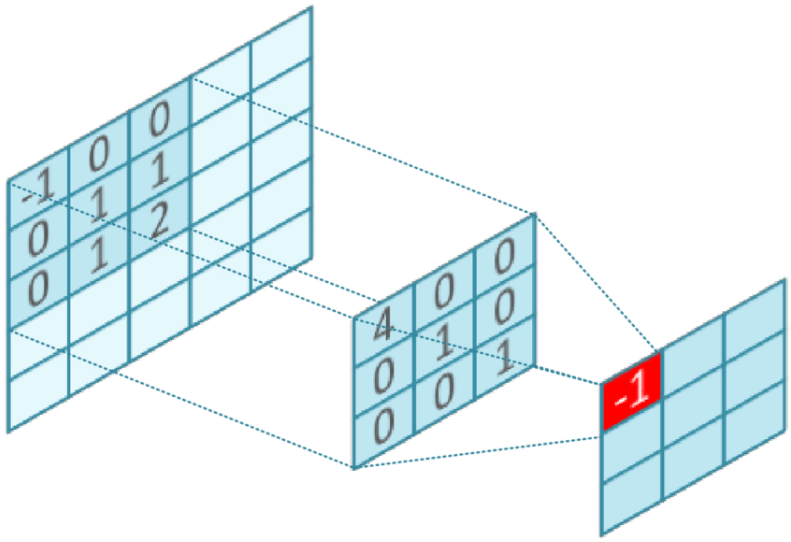
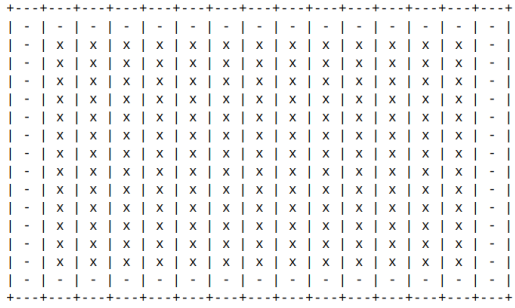


Figure 3.5: Convolution operation visually explained by sliding a 3 kernel over a 5×5 image with *valid-padding* producing a 3×3 feature map [33]. The resulting value is calculated by summing of the Hadamard product of the highlighted matrices: $-1 * 4 + 1 * 1 + 2 * 1 = -1$.

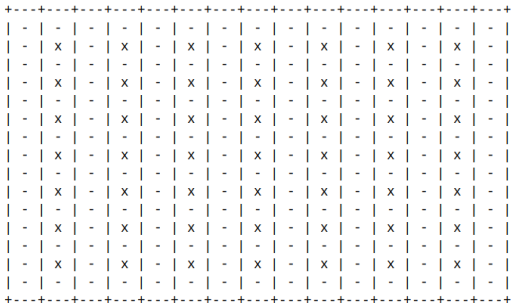
uses sparse connectivity meaning the output units are not fully connected with the input units [9]. By making the kernel a matrix of size smaller than the input a reduction in size is achieved. Besides setting the kernel size the stride size, the amount of pixels to move on per step, can be adjusted to decrease or increase the output size as well. A stride size of 1 means the kernel is moved column by column in the lateral direction and row by row in the vertical direction once the width of the input is reached. This produces a feature map of almost the same size as the input except for cutting of the outermost rows and columns. In comparison a stride size of 2 results in a feature map of half the width and half the height. This is due to computing a new value every second input pixel as depicted in Figure 3.6. Another factor describing the output size is the padding. In Figure 3.5 a valid-padding was used leading to a reduction of the output size. This is due to the fact that the 3×3 computes one output for the first 9 inputs which is in the middle of the kernel grid. If a padding, like a frame of zeros, would be added around the input, the kernel would compute the first output at the position of the top-left most input of the original input resulting in an output of same size. Therefore, this padding is called *same-padding*. An example is illustrated in Figure 3.7.

The sparse connectivity leads to fewer parameters to store and less computations to execute. This increases both the statistical and computational efficiency. While an image usually has thousands of pixels and the input and output have m and n parameters, respectively. This leads to a runtime of $O(m \times n)$. If the number of connections an output may have is limited to a number $k << m$ the sparsely connected approach has a runtime of $O(k \times n)$. In many applications it is possible to obtain good performance while keeping k several orders of magnitude smaller than m [9].

With a MLP, each element of a layer's weight matrix W is used exactly once to compute the layer's output. In a CNN the kernel is sled over the complete input. Therefore, the weights of that operation are shared across many input values. In the literature this property is described as tied weights, as the weight value applied on one input is tied to the weight value applied on another input value [9]. This reduces the computational cost in a CNN further as the model does not have to learn a different set of parameters per input. This characteristic is motivated by the fact that similar edges need to be detected across the whole input. Tied weights means sharing a filter to detect similar artifacts.



(a) stride size 1 resulting in 13×13 output units



(b) stride size 2 resulting in 7×7 output units

Figure 3.6: Comparing the resulting output units of a 3×3 kernel run over a 15×15 input with stride size 1 vs. stride size 2. In the figures above the grid represents the 2D input grid. The `x` shall represent an output unit resulting from the convolution operation of the neighbor units and itself. The `-` represents a unit that is not relayed to the feature map. The latter having less than half of the output units of the first convolution.

3.2.2 Pooling Operation

The third component to a convolutional layer, as described above, is the pooling function. The input to the pooling function are the outputs of the activation function run on the output of the convolution operation. In Figure 3.9 the convolutional layer is illustrated as a block of the three operations, convolutional, activation and pooling.

By taking in a matrix of activation values the pooling function computes a summary statistic of that matrix. By doing so the dimensionality given is reduced to one value per inputs of the pooling kernel size. The pooling kernel is, in contrast to the convolutional kernel a window that is sled over the input to define the input region for the pooling function.

Common pooling functions are maximum pooling, average pooling and deviations of those. Maximum pooling is just the $max(\cdot)$ function applied on the input matrix. Average pooling is, as the name suggests, the average function applied on the matrix. Especially average pooling is often altered to use different averaging function for a weighted average, as indicated in Definition 3.2.3 by substituting $g(\cdot)$ with some function of the input. The input's weight can, for instance, be based on the input's distance to the kernel's center.

Definition 3.2.2. *Maximum pooling*

$$f(X) = max(X), \quad X \in \mathbb{R}^{n \times m}$$

Definition 3.2.3. *Average pooling*

$$f(X) = \frac{1}{k} \sum^{k} x_i * g(x_i), \quad X \in \mathbb{R}^{n \times m},\ g(\cdot) \in \mathbb{R} \text{ and } k = m * n$$

Finally, pooling also leads to translational invariance w.r.t. the input. Meaning, when the input is slightly translated, e.g. rotated, the pooling function will still be able to detect the object, the filter would have detected without translation. One can think of this as a convolutional kernel detecting an edge in the input. This leads to high activation values which are then summarized by the pooling operation. In case the same input is rotated slightly, the activation function will return high values at slightly different outputs, which will then again be summarized by the pooling function. This feature is visualized in Figure 3.10.

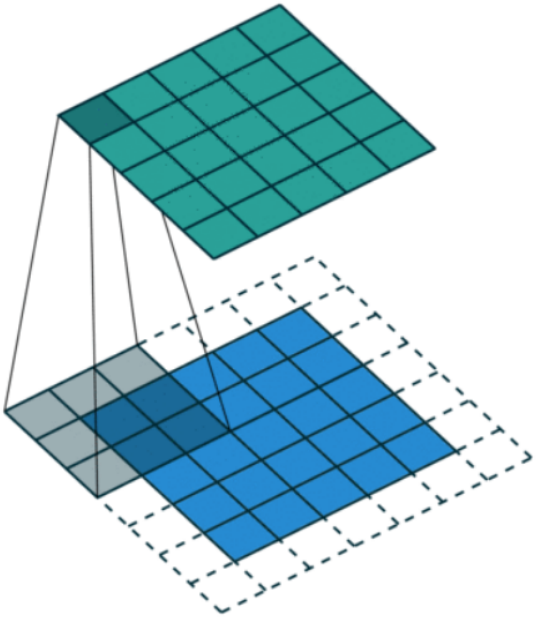


Figure 3.7: Example of a 3×3 kernel sliding over a 5×5 input with stride 1 and *same-padding* generating a same size output of 5×5 [25].

3.3 Training a Neural Network

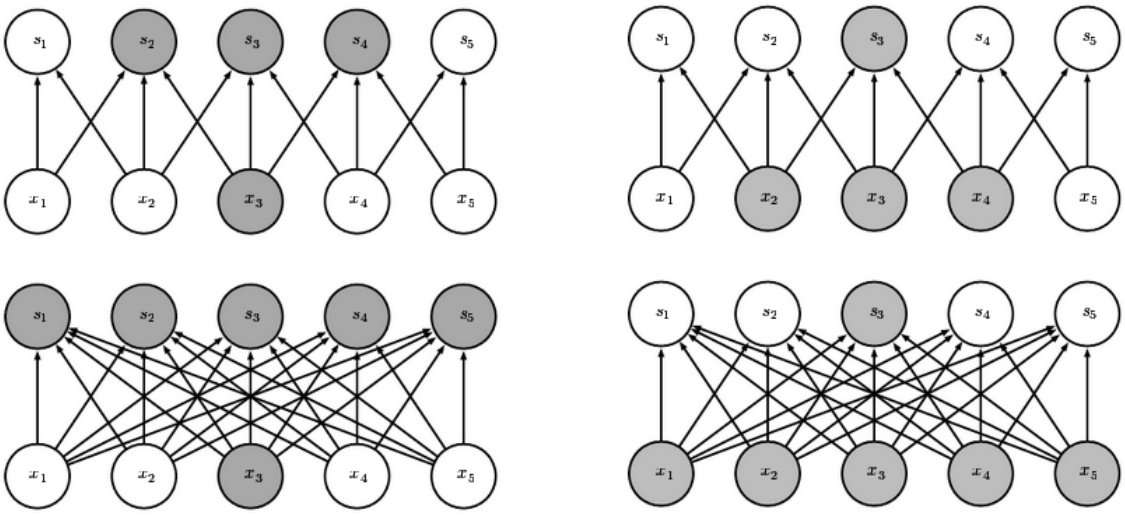
Crucial to the model's performance is its ability to minimize the deviation between predictions and the ground truth. Therefore, the model has to adjust its parameters to some set θ which brings $\hat{y} = f(x; \theta)$ possibly close to y. As mentioned in section 3.1 the basic building block to update the model's parameters is the backpropagation algorithm [16]. It "repeatedly adjusts the weight on connections in the network to minimize the measure of the difference between the actual output vector of the network and the desired output vector given the current input" [21] through updating the parameter's value by the error function's derivative w.r.t. the parameter, weighted by a learning rate. By definition, a function's gradient points in the direction of the steepest increase in value. In the case of an error function the goal is to change the parameters to decrease the error. Therefore, steps in the opposite direction are taken. The repeated calculation of the gradient uses the chain rule.

Definition 3.3.1 (Backpropagation algorithm)**.** *Given a cost function $J(\theta)$, repeatedly update the model's parameter by the cost function's derivative w.r.t. that parameter. The amount the parameter's value is updated by the gradient is controlled by the learning rate ϵ. The gradient w.r.t. the parameter is calculated using the chain rule.*

$$w^i_{t+1} \leftarrow w^i_t - \epsilon_t * \frac{\partial J}{\partial w^i}$$
$$\frac{\partial J}{\partial w^i} \leftarrow \frac{\partial J}{a^l} * \frac{\partial a^l}{z^l} * \cdots * \frac{\partial z^i}{w^i}$$

Where w^i is the weight in the ith layer to be updated and a^l the last output unit resulting in the output vector.

The learning rate is a crucial hyperparameter to tweak as it controls the effect a gradient has on the update. In extrem cases the update can learn nothing as the update is either too small or too high and the update therefore misses the optimal value by



(a) outbound connections (b) inbound connections

Figure 3.8: Illustrating the concept of sparse connectivity by highlighting the outbound connections of a unit a and highlighting the inbound connections of a unit b. Each in contrast to the MLP case [9].

jumping over it. However, a larger learning rate is sometimes desired to not get stuck in local minima in the beginning and to explore the value space more rapidly. As soon as the algorithm is close to a local minimum the learning rate shall decrease to explore the values in more detail. This is achieved by decaying the learning rate by either a function or a constant factor over the training period [10][16]. Time based decay, for example, alters the learning rate based on the iteration step and on the current learning rate. In the following ϵ_i is the learning rate for step t and η the learning rate decay parameter.

$$\epsilon_{t+1} = \frac{\epsilon_t}{1 + \eta t}$$

Instead of following a gradient for a complete training data set the update can be sped up by computing the gradient for minibatches, i.e. smaller samples of the training data. This approach is called Stochastic Gradient Descent (SGD) and is the basic optimization algorithm for calculating the update [9]. This approach is based on the unbiased gradient estimate obtained as average gradient from an i.i.d. minibatch of size m. The update is computed by the gradient estimate $\hat{g}$, given a sample of m training examples $(x^{(i)}, y^{(i)})$:

$$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$
$$\theta_{t+1} \leftarrow \theta_t - \epsilon_t * \hat{g}$$

In the above case the likelihood function is used as cost function which shall be optimized as a cost function can easily be derived from the likelihood as in Example 3.1.1.

The work presented in chapter 4 uses the Adam optimizer instead of SGD which is a special from of the SGD algorithm. It uses the computed gradients during backpropagation, historical values as well as the first and second moment estimates of the gradient [15]. The update is computed by

$$\theta_{t+1} \leftarrow \theta_t - \epsilon_t * \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \alpha}$$

where $\hat{m}_i$, $\hat{v}_i$ are the first and second moment estimates of the gradient and α a standardization constant.

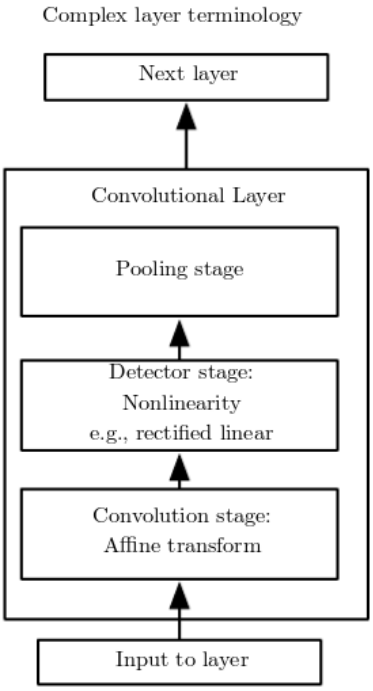


Figure 3.9: Coupling the operations of convolution, activation function and pooling together builds the convolutional layer or block. With such a representation [9] a CNN is constructed of several complex blocks. Such a block is not restricted to one convolution operation or pooling. Indeed, it is common to use several convolution operations in one block as shown in the VGG example Listing 3.1.

3.4 Clothing Classification Example

In the following example the before described CNN architecture will be used to classify clothings in 28×28 single channel images from 10 categories. The Fashion-MNIST data set is composed of 70.000 clothing samples (60k train, 10k test) from Zalando and is inspired by the MNIST data set which facilitates same size images of handwritten digits for fast model exploration [34]. A visualization of the convolutional layers and a direct comparison between a MLP and a CNN on the same task are presented.

The convolutional network uses a similar architectural approach as the before mentioned VGG16 and is built up in blocks of convolutional operations paired with the ReLU activation function and followed by a maximum pooling operation. In the second column of Listing 3.2 the output dimensions are noted. For the first two layers the same-padding was used which results in the preserved sizes. The third number in the tuple indicates the number of kernels (filters) in those layers, i.e. the first layer runs 32 filters on an input of 28×28 and outputs the same dimension. For the maximum pooling a kernel of size 2×2 was chosen, resulting in halfing the width and height or quartering of the output. At the end of the model one *flatten* and two dense layers were added. The earlier is necessary to reduce the dimension from 2D to 1D by stacking the values to one vector. The latter two reduce the size further until only 10 units are left, corresponding to the 10 categories to predict. Besides for the last layer only ReLU activation functions were used. The last layer uses the *softmax* function which outputs the class probabilities over the possible output categories, i.e. returning a vector z with $z \in [0,1]^{10},\ \sum_i z_i = 1$.

Definition 3.4.1. *Softmax activation function* $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ *where* $K \in \mathbb{N}$

$$\sigma(z)_i = \frac{\exp z_i}{\sum^K \exp z_j}, \forall i \in 1, \ldots, K,\ (z_1, \ldots, z_k) \in \mathbb{R}^K$$

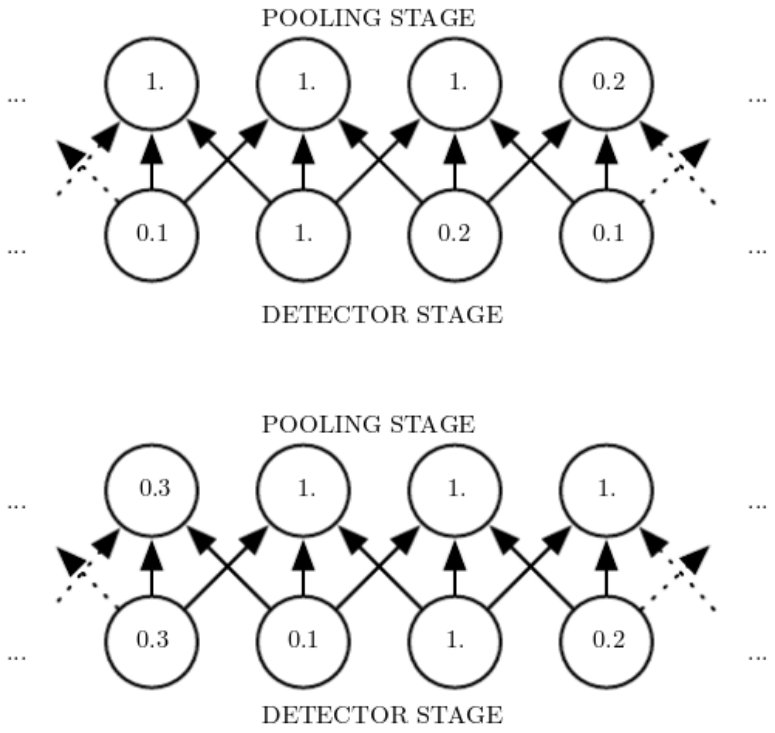


Figure 3.10: The above figure shows the initial output from the detector stage entering the pooling operation. Below, the detector's output is shifted a little to the right due to translation of the image but still, the pooling result is very similar [9].

Of the 60k training images the model was trained on 50k images and 10k images were used as validation set during training. A validation set is useful to monitor the training performance and stop in case of overfitting or convergence. This is known as *early stopping*. A total of 10 epochs were run as depicted in Figure 3.12a.

When run on the test set the model achieves a loss of 0.27 and an accuracy of 0.92 which is a little below the training results. As the model has not seen the test images before, this is normal and still a good result. A visual evaluation of the model is presented in Figure 3.13.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_25 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 28, 28, 32)        9248
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 12, 12, 64)        18496
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 12, 12, 64)        36928
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 4, 4, 64)          36928
_________________________________________________________________
flatten_4 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_8 (Dense)              (None, 64)                65600
_________________________________________________________________
```

Figure 3.11: 48 random samples from the Fashion-MNIST data set. The images belong to one of 10 clothing categories and are each of size 28×28 with one channel.

```
dense_9 (Dense)              (None, 10)                650
=================================================================
Total params: 168,170
Trainable params: 168,170
Non-trainable params: 0
_________________________________________________________________
```

Listing 3.2: Example CNN architecture

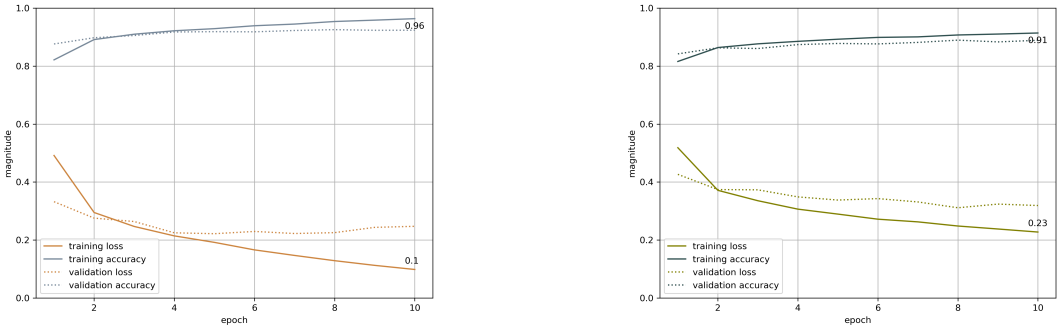
The third column in Listing 3.2 states the parameter count in that layer, which has been discussed several times above when explaining the efficiency of CNNs on grid shaped data. The parameter count in a CNN results from the kernel k dimensions, channels c of the input and the number of kernels $n_{kernels}$ to use in a layer

$$\#parameters = (height_k * width_k * c + 1) * n_{kernels}$$

while the extra 1 is added for the bias. In the next layer the number of output channels of the preceding layer is used as input channels. For the second layer this would be 32 and results in $(3 * 3 * 32 + 1) * 32 = 9248$.

In total the CNN uses 168.170 parameters. Parameters can be frozen to not change over the training process. Those would appear in the last line as non-trainable parameters. Note how fast the parameter count increased in comparison to most other statistical methods.

Also, w.r.t. the MLP shown later, it is possible to build up many abstraction layers with that number of parameters. In the case of the MLP this depth will not be reached. It has to be mentioned that the maximum depth of a model was not in scope of this example but an artifact.



(a) CNN training evaluation

(b) MLP training evaluation

Figure 3.12: Training processes of the CNN and MLP trained for 10 epochs on batches of size 64. A clear trend of increasing accuracy and decreasing loss is seen. The final values of training accuracy and training loss are 0.96 and 0.09 for the CNN and 0.91, 0.23 for the MLP, respectively. By observing the close connection between training and validation accuracy no overfitting took place.

3.4.1 Visualization of Convolutional Layers

As convolutional networks are often referred as opaque due to their shear amount of parameters and therefore, little transparency. To still get a feeling for what happens inside a CNN the filters per layer as well as their feature maps can be visualized by plotting those values. Still, filters are just weights arranged in a grid.

The first layer has a number of 32 filters, each of dimension 3×3. Those filters are visualized in Figure 3.14. By observing the gradients per filter, one can examine the feature this filter extracts. Taking the filter at position *(row: 2, column: 8)*, the values in the first column are large while those in columns 2 and 3 are smaller (*darker*). This filter extracts a vertical edge as the values along that vertical axis in the input are high on the foreground and low on the background as all images have a black background.

In the first layer, visualizing all filters is possible as there are only a few. This is not possible when reaching further into the model. In the current example the next layer has $32 * 32 = 1024$ filters as there are 32 output channels by the first layer and the second layer implements 32 filters itself. As mentioned above, the feature map of each layer incorporates the output of the convolutional operation. Therefore, applying the filters of a layer on an image will return the model's perception (detected features) of that input as the filters were applied. This way the model's perception can be analyzed further into the model and is more meaningful than the linear filters alone.

To do so, the output of the specific layer has to be visualized. Figure 3.15 is from the Fashion MNIST data set and included in the test set as well. This image was run through the network and the first layer's output is shown in Figure 3.16. As this is the first layer, more detailed features shall be detected. This is the case as the most detected features are edges. Also, the feature maps are arranged according to the filters displayed in Figure 3.14. Filter $(3, 8)$ was identified to detect right edges in the input. This is indeed the case when observing feature map $(3, 8)$ where the right edge is highlighted. Again, highlighted regions are the representation of higher values indicating activation.

3.4.2 Comparison of MLP and CNN on image analysis

Finally, the initially introduced MLP shall be compared to the CNN in this example. Recall, that the MLP heavily utilizes densely connected layers. Even though the images

Figure 3.13: 64 predictions of the trained CNN on the test set. The color coding of the x-labels shows whether the prediction was correct. Note that predictions are coupled with a high confidence of the model for both correct and incorrect predictions.

of 28×28 pixels are very small compared to those taken by todays cameras, this causes the model to start with a large number of parameters. As the MLP expects a 1D input the image has to be flattened resulting in $28 * 28 = 784$ pixels which results in the same size first layer as there needs to be one unit per input pixel. Imagine applying the MLP to an image of the input size of the VGG16 with $224 * 244 = 50.176$ [29], which won the ImageNet competition [24] in 2014. For the default MLP the parameter count per layer is simply calculated as a product of inputs and units per layer:

$$\#parameters = n_{input} * n_{units}$$

The MLP was defined accordingly with an input layer suiting for all 784 pixels, followed by 4 dense layers extracting necessary information from the pixels, see Listing 3.3. For the MLP the same training, validation and testing data was used as for the CNN above. Within 10 epochs of training, the MLP reached a training accuracy of 0.91 and a training loss of 0.23, see Figure 3.12b. While the accuracy is fairly close, the loss is clearly higher compared to the CNN.

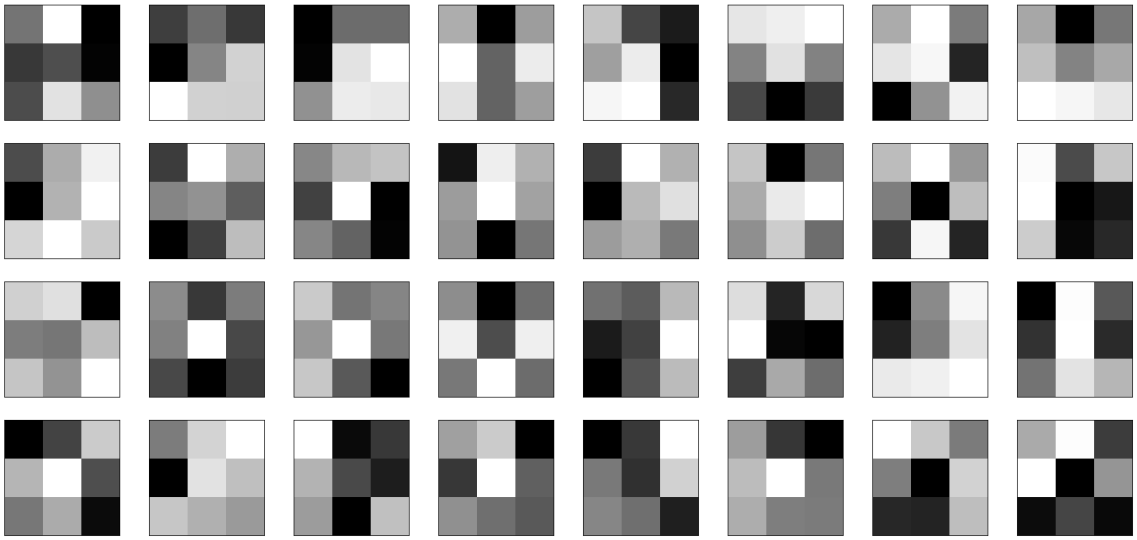


Figure 3.14: Visualization of the 32 filters from the first layer in Listing 3.2. The coloring indicates low values for darker squares and higher values for lighter squares.

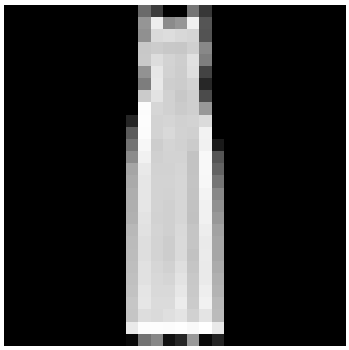


Figure 3.15: Sample image from the Fashion MNIST data set [34] displaying a dress. Actually, this dress was included in the test set (Figure 3.13, first row third image) and predicted correct with a confidence of 100%.

Also, the CNN is better capable of generalizing to new data. On the test data the MLP only achieves a test accuracy of 0.88 and yields a loss of 0.34 in contrast to 0.92 and 0.27 for the CNN, respectively. The MLP was used to predict on the same data as the CNN before in Figure 3.13. Without a quantitative analysis it is observed that the MLP is not as confident on predictions. On the correct predictions, the MLP is less confident but also seems to be more wrong on the incorrectly classified images.

Comparing the total parameter count of the MLP with 242.762 to that of the CNN with 168.170, the MLP is roughly 145% the CNN's size. Note, the MLP is also shallow compared to the CNN with 5 vs. 10 layers. While the performance results are not far apart, the MLP is larger than the CNN and this difference in size will increase with the input size ranking MLP impractical for real-life images. Making the case to prefer a CNN architecture when working with grid-structured data as done in chapter 4.

```
1
2   _________________________________________________________________
3   Layer (type)                 Output Shape              Param #
4   =================================================================
5   flatten_8 (Flatten)          (None, 784)               0
6   _________________________________________________________________
7   dense_20 (Dense)             (None, 256)               200960
8   _________________________________________________________________
    dense_21 (Dense)             (None, 128)               32896
```

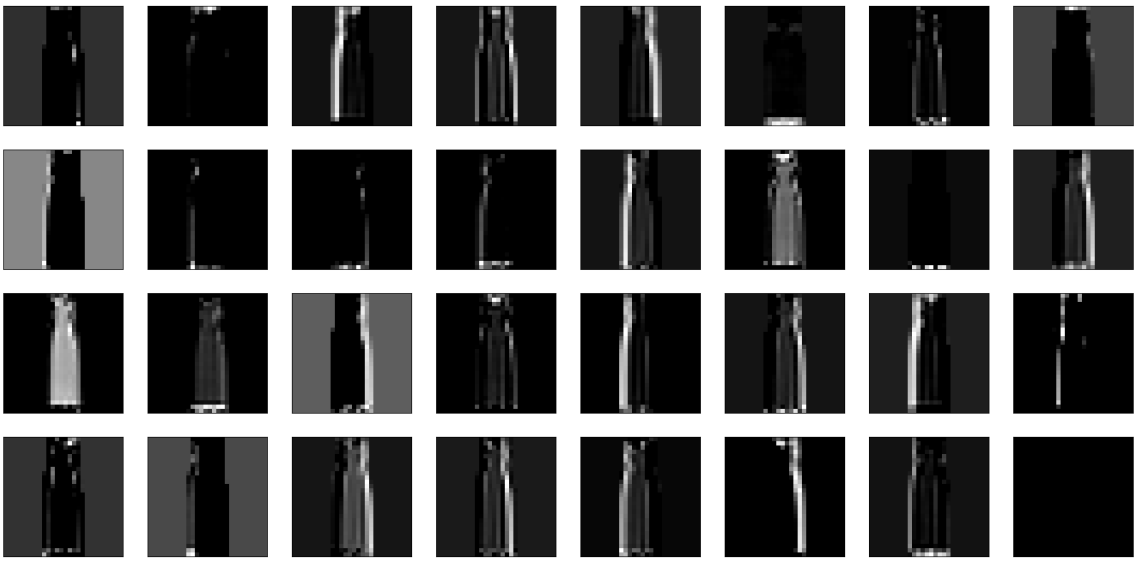


Figure 3.16: Visualization of the 32 resulting feature maps when run on the sample image in Figure 3.14. The first layer observes details as edges. This is obvious on the white parts of the image which indicate an activation as those represent higher values compared to lower dark ones.

```
9     _________________________________________________________________
10    dense_22 (Dense)             (None, 64)                8256
11    _________________________________________________________________
12    dense_23 (Dense)             (None, 10)                650
13    =================================================================
14    Total params: 242,762
15    Trainable params: 242,762
16    Non-trainable params: 0
17    _________________________________________________________________
```

Listing 3.3: Example MLP architecture

Finally, Jiuxiang Gu et al. have written a comprehensive summary of good practices and improvements on Convolutional Networks including a discussion on different kinds of ReLU activation functions, convolutions, pooling functions regularization and optimization in their work *Recent Advances in Convolutional Neural Networks* [10]. The interested reader shall also be referred to their work.

Chapter 4

Pipeline Implementation

The proposed workflow consists of three steps where the surgeon *(1)* first defines a desired view on the patient using the preoperative CT-scan, then, *(2)* during surgery preparation, the fluoroscopy machine locates itself by analyzing an X-ray image from the current perspective on the patient and then *(3)* moves into the desired postion. To implement this workflow a model capable of localizing the fluoroscopy machine from a X-ray image is key. Besides, the model has to be trained with available, high-accuracy data. Step two further involves human workforce and feeding the model from step one with an X-ray image. Step three concludes the workflow by an algorithm computing the movement to reach from the current position to the desired position. A schematic overview of the proposed pipeline is given in Figure 1.2. This work covers discussion and implementation of the first step.

For position encoding a six dimensional vector was chosen describing rotation in degrees and translation in millimeters. Problems of uniqueness introduced by this notation, like gimbal lock, where taken into account during implementation. Gimbal lock describes the loss of one degree of freedom in a three gimbal rotation system when two gimbal-axes are in a parallel state, see Figure 4.1. Also, the parameter settings do not include a close to 360° rotation which mitigates the problem in accounting for degrees with a close projection but great numerical distance, e.g. 1° vs. 359° which are both close to a true 360° rotation but have errors of 359° vs. 1°.

The problem to solve in step one therefore, is defining a function to map an image of certain size to a six dimensional vector describing the orientation the image was taken from. Figure 4.2 gives an idea of the setup where the fluoroscopy machine is depicted as the gray angel bar, which is static, and the patient is moved w.r.t. to the machine.

Problem Definition. *Given an X-ray image I, a model f shall be able to detect the patient's orientation of which the image was taken, encoded in six degrees of freedom in rotation and translation. As the fluoroscopy machine is static the patient is moved.*

$$f : I \mapsto (\theta, \phi, \rho, t_x, t_y, t_z), \quad I \in \mathbb{R}^{480 \times 620 \times 1}$$

To approach the problem a CNN, described in section 3.2, will be fit on synthetic X-ray images generated using a modified version of the DeepDRR framework, discussed in section 2.1. In the following the image generation and model definition will be discussed thoroughly followed by the model's performance analysis.

4.1 Generating Digitally Reconstructed Radiographs

A large data set of highly accurate synthetic radiographs is needed for developing the model. To create the data set, digitally reconstructed radiographs are generated from

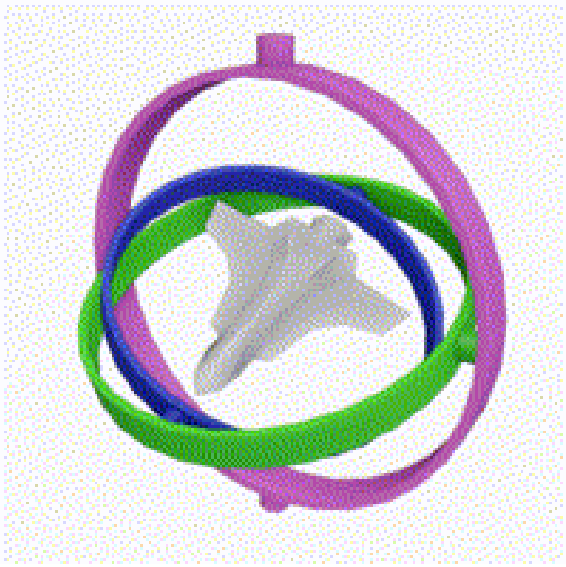


Figure 4.1: Illustration of the gimbal lock problem on an airplane. The three rotations available are pitch (*gree*), roll (*blue*) and yaw (*magenta*). In case of aligning the axis for pitch and yaw the plane looses one degree of freedom in movement as rolling and yawning apply the same change to the airplane [5].

different perspectives as this ensures the accuracy needed in the labels. The framework to generate the images is described in section 2.1. Using the framework, the object of which the synthetic radiograph shall be computed for can be transformed by specifying a rotation around each axis as well as a translation in each of the three directions.

To specify each variable a notion of the anatomical coordinate system is needed. In which the value ranges are specified by the directions on a human with the human's center usually encoded as the value 0. A common system, and the one used here is abbreviated as *LPS* for Left, Posterior, Superior. In such a system the values increase from right to left, anterior to posterior and from inferior to superior. Also, those directions lead to abbreviations of special X-ray shots. An X-ray image taken e.g. along the anterior-posterior axis is called AP-shot.

As input the framework requires a CT-volume which was taken from the Visible Human Project's [32] website, offering such data. It resulted from a selection process regarding different available scans which where evaluated w.r.t. their label's completeness. A CT-volume is usually built up of several *DICOM* images depicting the slices along the inferior-superior axis, see Figure 4.3. Besides relevant meta information like the used coordinate system - usually LPS - the DICOM images include metrics on how many slices are contained in the current volume, at which position the current slice is located w.r.t. the other slices and the thickness of one slice. Those attributes are needed to reconstruct the CT-scan as a volume as shown in Figure 4.4. Especially, the attribute *SliceThickness* is important for the DRR framework as it specifies the spacing between the slices. In the case of the above mentioned CT-volume neither *SliceLocation* nor *SliceThickness* were present and had to be interpolated. As the file names hinted an ordering, this was used to reconstruct the location of each slice. The thickness was interpolated by using an estimation of the scanned body size divided by the available slices. The reconstructions in Figure 4.4 hint a nearly correct interpolation.

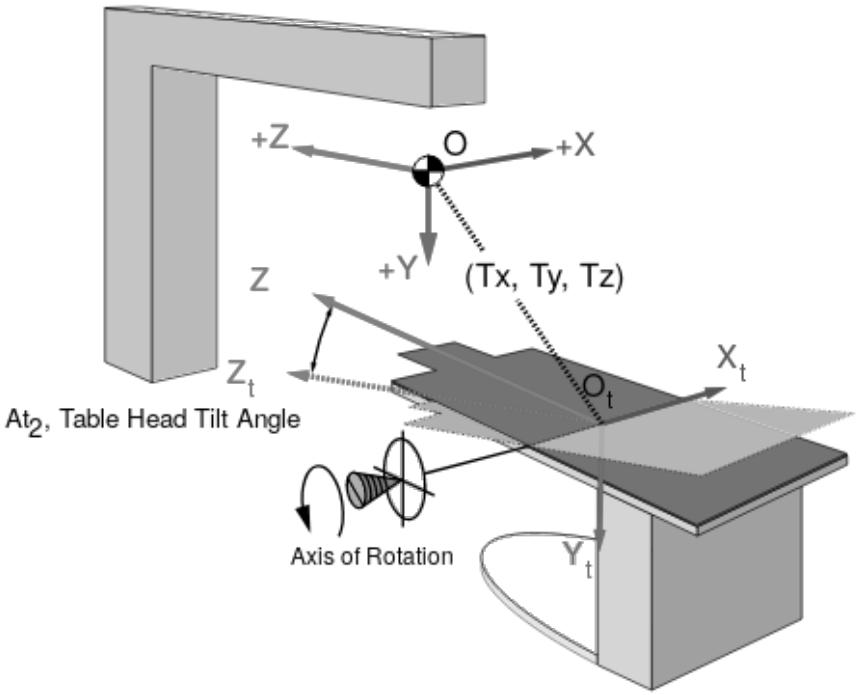


Figure 4.2: Illustrating the setup in which a function f shall predict the rotation and translation applied to a patient on the here illustrated treatment couch. The figure illustrates the three translational axis and one rotational axis. The rotation here indicated spins around the RL-axis of the patient in a *LPS* anatomical coordinate system. Note, the used notation in this figure does not comply with the notation used in this work.

4.1.1 Data Generator

Besides a few smaller adaptions to the framework, an important improvement was added by implementing a *Data Generator* class which facilitates the automatic generation of the data set while offering a possibility to parallelize the same to decrease the computation duration. A second useful addition was the *Parameter Visualizer* class offering an interface to create a DRR based on the model's predictions. This is useful as the input image can be compared to the image the model says it has seen, i.e. the image according to the parameters predicted by the model.

Concerning the Data Generator, the range of attributes the DRR shall have need to be specified. Further, either the ranges are specified directly or are computed based on a lower and upper bound per parameter from which an uniform sample is drawn with a defined step size. Listing 4.1 shows the class' constructor signature. In prior to generation the file configurations as the cartesian product of all possible settings are saved to then be pulled by the DRR generator. Also, the CT-volume is only loaded once and the according projections are calculated per configuration. In total, this enables a parallelization of the generation process as the configurations can be split onto different workers. Also, the methods in Data Generator are structured in a way to minimized reading and writing to disk.

```
def __init__(self, complete_segmentation: bool = True,
                 angle_step: int = 5, translation_step: int = 5,
                 phi_range: Tuple[int, int] = (0, 180),
                 theta_range: Tuple[int, int] = (0, 180),
                 rho_range: Tuple[int, int] = (0, 0),
                 origin_x_range: Tuple[int, int] = (0, 0),
                 origin_y_range: Tuple[int, int] = (0, 0),
```

Figure 4.3: Sample slice contained in the used CT-volume.

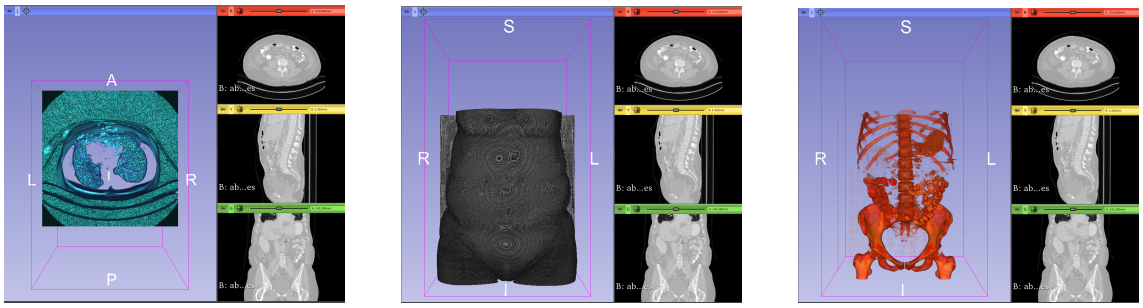
```
8                   origin_z_range: Tuple[int, int] = (0, 0),
9                   cartesian_prod: bool = True, camera: Camera = None,
10                  ct_volume_path: str = None,
11                  save_to_dir: str = './generated_data',
12                  random_pick: bool = True, sample_size: int = None,
13                  spectrum=SPECTRUM120KV_AL43, scatter: bool = False,
14                  photon_count: int = 100000):
```

At this point one is referred to the accommodating GitLab repository including more documentation on the modifications and an usage example.

4.1.2 Projection Parameters

The six dimensional orientation vector is encoded as in the problem definition above. The parameter's effects are listed in Table 4.1. The value ranges as well as the step size for sampling are also included in that table. To limit the computation time and still include a wide variety of views, a trade-off was made to restrict the configuration to a frontal view covering the sides to an angle of 45° each as well as covering the top and bottom to an angle of 60° each. The view from both minimal angle values is depicted in Figure 4.5.

In total a number of 69300 images were generated as the result of the setting's cartesian product. A Google Cloud VM with a Nvidia P4 GPU was used and ran for roughly 5 days. Note, that this setup was deployed as a minimum viable product and therefore did not focus on speed. Based on the metric of DRR generation used in Data Generator, processes could easily be parallelized. As the GPU's RAM was only utilized to *25%* and the GPU ran idle for short periods of time parallelization should be considered. Another possibility would be to deploy the setup in a cluster using e.g. Docker images and initialize the Data Generator with different ranges in the first place. As the machines do not have



(a) segmented air (b) segmented soft tissue (c) segmented bone

Figure 4.4: Figures showing the used CT-volume segmented using thresholding showing the three materials, air, soft tissue and bone. Note the anatomical axis are depicted in the images as well indicating the view point. In 4.4a an superior-inferior view is shown, i.e. observing the lungs as well as the air in them.

Parameter	**Range**	**Step size**	**Effect**
ϕ	$[45, 135]$	$10°$	rotation around IS-axis in degrees (roll)
θ	$[60, 120]$	$10°$	rotation around RL-axis in degrees (pitch)
ρ	$[0]$	-	rotation around AP-axis in degrees (yaw)
t_x	$[-80, 80]$	$20mm$	translation on the RL-axis in millimeters
t_y	$[-100, 100]$	$20mm$	translation on the AP-axis in millimeters
t_z	$[-60, 120]$	$20mm$	translation on the IS-axis in millimeters

Table 4.1: Listing the parameters with their effect and the values used to generate the DRRs.

to communicate, latency is not a problem and the efficiency will scale linearly.

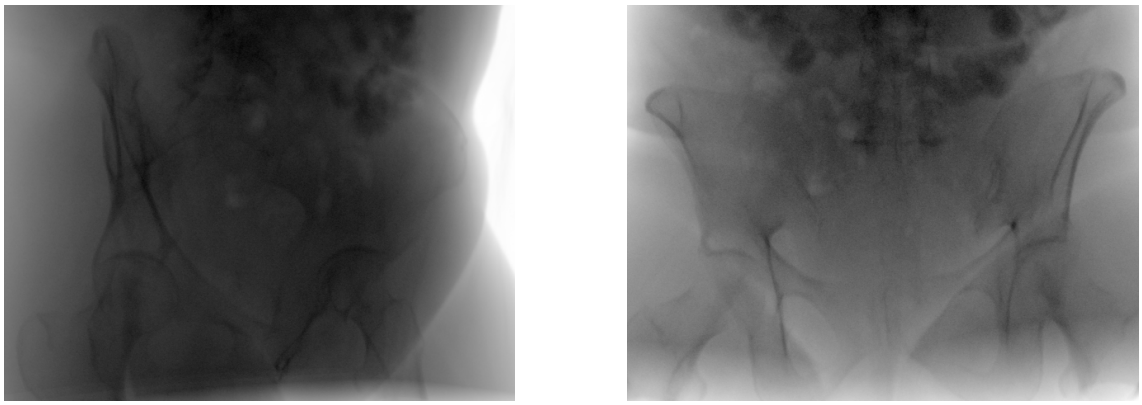
$$\begin{aligned} \#Images &= \#range_{\phi} + \#range_{theta} + \#range_{\rho} + \#range_{t_x} + \#range_{t_y} + \#range_{t_z} \\ &= \frac{90+10}{10} * \frac{60+10}{10} * \frac{160+20}{20} * \frac{200+20}{20} * \frac{180+20}{20} \\ &= 69300 \end{aligned}$$

The resulting DRRs use *16bit* encoding for their colors and are saved in *.tiff* format. The very detailed color encoding is responsible for a large file size of roughly *5MB* per image. Also, the pixel values are in the thousands and therefore show very little deviation making them hard to observe for humans. To tackle both problems the images were down sampled to a *8bit* encoding using a custom script for batch processing the generated files as well as log-scaling and normalizing the images to the common $[0, 255]$ range per pixel to increase human visibility.

Note, while segmentation was tried with CNNs and implemented in the DeepDRR segmentation, using thresholding worked a lot better. This is most probably due to the litte training data used for fitting the segmentation network. Therefore, the network was not capable to generalize on other CT-volumes. The segmentation results are depicted in Figure 4.6.

4.2 Model definition

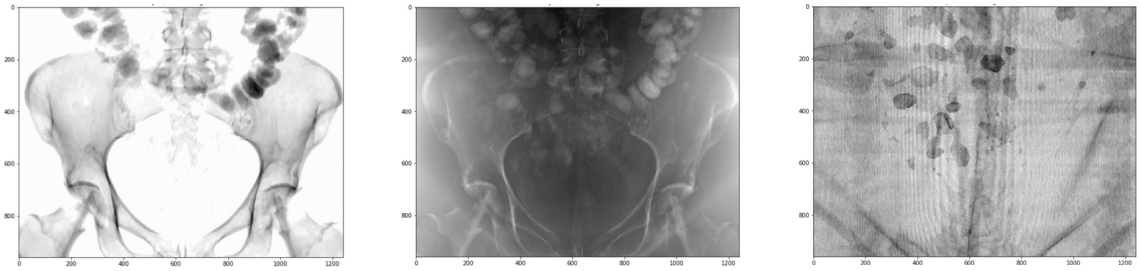
Finally, the model was constructed similar to those shown in section 3.4 implementing a block structure of convolutional operations, ReLU activations and maximum pooling operations. Following the convolutional block one flatten and three dense layers reduced



(a) $\phi = 45°,\ \theta = 90°,\ \rho = 0°$
$t_x = 0mm, t_y = 0mm, t_z = 120mm$

(b) $\phi = 95°,\ \theta = 60°,\ \rho = 0°$
$t_x = 0mm,\ t_y = 0mm,\ t_z = 120mm$

Figure 4.5: Figures depicting the two most extreme rotation configurations of ϕ and θ with a view of the pelvic area. While the one angle is set to the range's minimum the other angle is set to either 90° or 95° which is to mitigate problems as gimbal lock.



(a) bone (b) soft tissue (c) air

Figure 4.6: Resulting DRRs using manual segmentation as the segmentation using the CNN did not work well. Note, bone and soft tissue do overlap as described in section 2.1 as the pelvic bone structure is visible in the soft tissue segmentation.

the dimensionality to a 6D output vector matching the desired prediction dimension. The last layer used a linear activation function. The full model is listed in Listing 4.2 as those in section 3.4.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 64)        36928
_________________________________________________________________
flatten (Flatten)            (None, 36864)             0
_________________________________________________________________
dense (Dense)                (None, 256)               9437440
_________________________________________________________________
```

```
dense_1 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 390
=================================================================
Total params: 9,547,526
Trainable params: 9,547,526
Non-trainable params: 0
_________________________________________________________________
```

Listing 4.2: Architecture of the basic prediction model

To train the model, a dedicated training-pipeline was implemented building a generator to prepare the training, validation and testing data, including preprocessing, as batches. This results in one stream of data and single loading. Preprocessing included shuffling, standardizing the labels as well as the pixel values and resizing the image as a whole to 224×224. The parameters for standardization were saved to scale the predictions back to their original scale. Besides positive learning effects the parameter transformation also enabled the use of a standard deviation measure for both rotation and translation. Batch preparation is limited to solely run on the CPU, reserving the GPU for training.

The batches are loaded into memory just before they're needed to spare GPU-memory space. Also, pre-fetching was used to pre-load four batches to minimize GPU idling between the last backward and the next forward propagation. In total the model was trained, validated and tested on a $0.7,\ 0.15,\ 0.15$ split with a batch size of 256 images. Therefore, an epoch needed 190 steps for the training and 41 steps for the validation set.

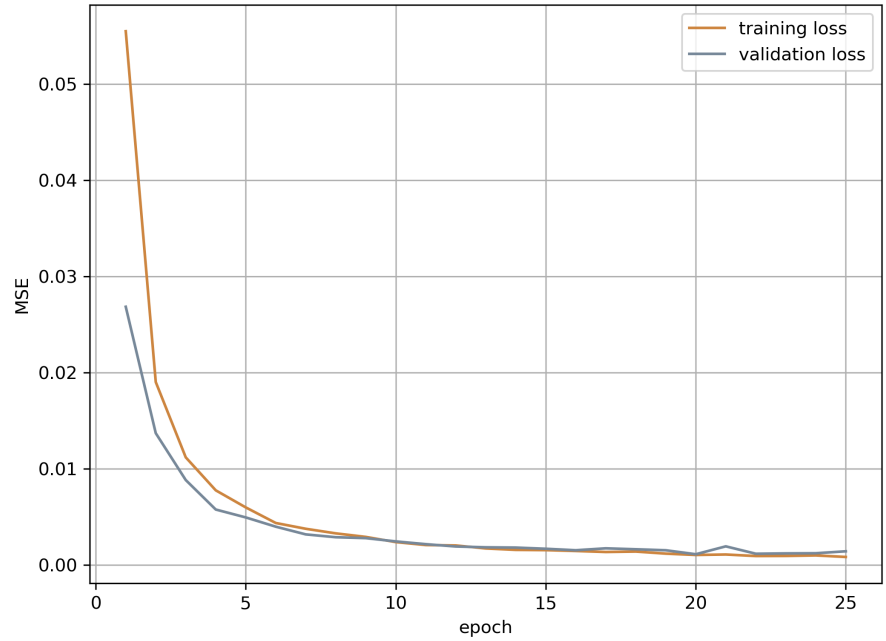
The mean squared error (MSE) was used as a loss metric to optimize. As the labels were standardized one metric could be used for both. The MSE was chosen as a fast to compute and robust loss function as it uses the squared deviation. Also, the absolute error could have been used.

$$MSE(y, \hat{y}) = \frac{1}{n} * (y - \hat{y})'(y - \hat{y})$$

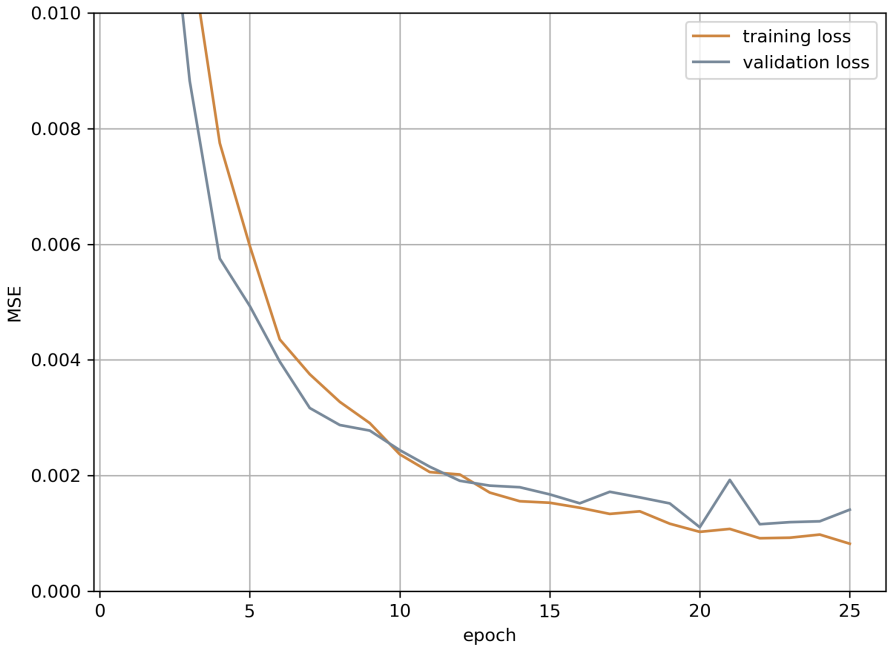
To optimize the loss function the Adam optimizer with a learning rate of 0.001 which was lowered by $5 * 10^{-6}$ per step was used. Starting with a greater learning rate makes sense as a faster exploration resulted in a fast decay of the loss function, see Figure 4.7a.

Figure 4.7a hints a fast adoption of the model to the data while it seams as the performance converges after the 15th epoch. Evaluating the training losses issue a minimum training loss of 0.00082 which is reached in epoch 24. The minimum loss on the validation data is reached in epoch 19 with an MSE of 0.0011. The minimum training and validation loss therefore differ by 0.00028. However, the validation loss in epoch 24, where the training loss in minimal is at 0.0014 and therefore marginally larger than at its minimum.

Figure 4.8 shows four sample predictions mainly from the pelvic area as it possesses the most interesting structure. The predictions are each rounded to the nearest integer as micrometer or decimal degrees do not add any information but the model's trend is observable. While most deviations are in the range of a few units predictions of t_y (translation on the AP-axis) are greater than those on $t_x,\ t_z$. The absolute average deviations on the test set per image are $3.5°$ for rotations and $22.24mm$ for translations, and depicted in Figure 4.9. Although the absolute deviation is of great magnitude the test set has 10395 samples and the values were transformed back to their initial scale. Finally, the model reaches a loss of 0.00158 on the test set which is only 0.00018 greater than on the validation set and 0.00046 greater than the loss on the training set.



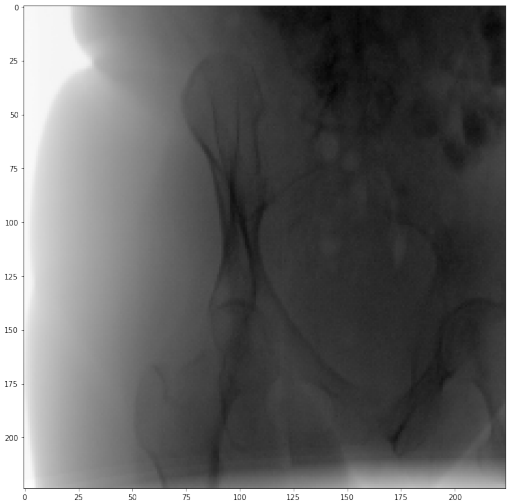
(a) Training evaluation of the proposed CNN on 70% of the data and validating it on 15%. The similar training and validation loss curves indicate a good training without overfitting the data. Minimum training loss at 0.00082; minimum validation loss at 0.0011.



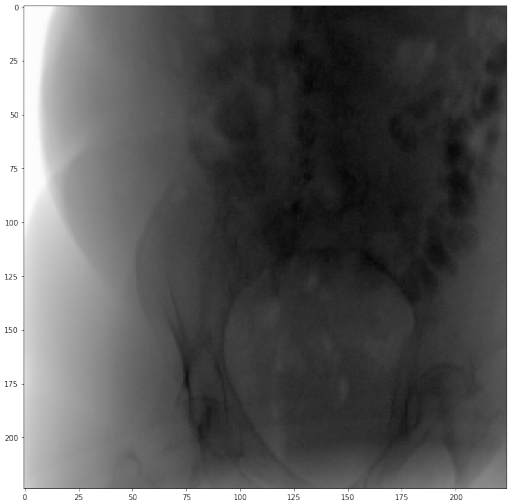
(b) As the MSE declines rapidly the scale needs to be reduced for a more detailed view on the model's performance. The first losses below 0.1 are in epoch 4. Therefore a restriction to the MSE of $[.00, .01]$ is reasonable. The training curves are clearly not as smooth as in Figure 4.7a. Therefore, the training could run for more than 25 epochs while further decreasing the learning rate.

Figure 4.7: Training evaluation of the proposed CNN on the generated DRRs.

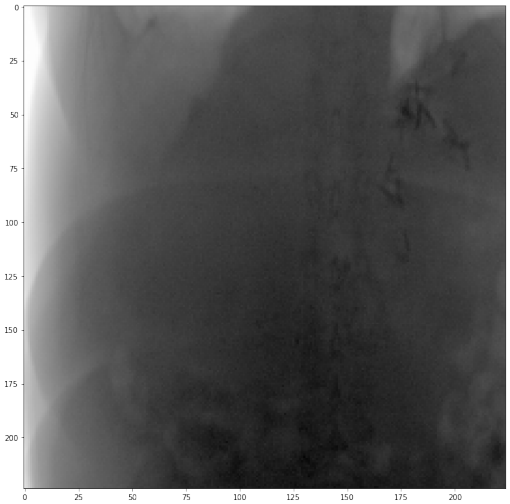
(a) $\hat{y}$: [96, 91, 0, −24, 76, 75]
y : [100, 95, 0, −20, 80, 80]



(b) $\hat{y}$: [89, 65, 0, 78, 22, 108]
y : [90, 65, 0, 80, 20, 120]



(c) $\hat{y}$: [109, 74, 0, 59, 68, 33]
y : [110, 75, 0, 60, 60, 40]



(d) $\hat{y}$: [119, 94, 0, 33, −58, −63]
y : [120, 95, 0, 40, −60, −60]

Figure 4.8: The predictions where computed on samples in the test set. As mentioned earlier the deviations in translations are larger which is quantified in Figure 4.9. Note, the predictions were rounded to the nearest integer as this still indicates the model's tendency but spares unneeded accuracy as a result on the micrometer scale is not interpretable. Loss on complete test data: 0.00158.

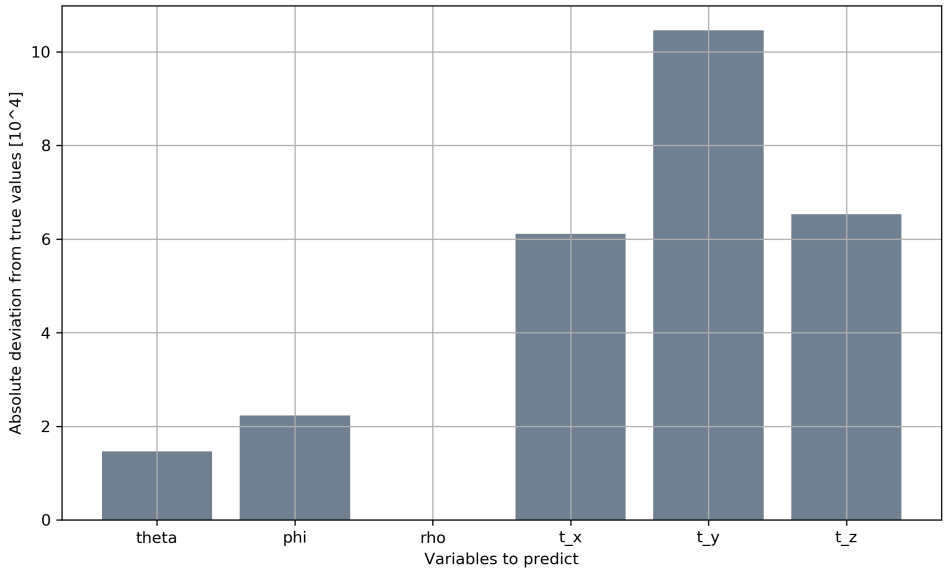


Figure 4.9: As suggested above, the absolute deviation in translational variables is greater as in rotational variables. Deviations are computed on all batches in the test set with 256 images each. Note, the largest deviation is in direction t_y, which is translation along the AP-axis and therefore in viewing direction. Also note, even though the values are high, for t_x and t_z the average absolute deviation is $\sim 6.0mm$. Lastly, as the translations have one degree of freedom more than the rotations, this is an expected behavior.

Chapter 5

Conclusion

In this work I proposed a three step pipeline to decrease preparation time for fluoroscopy guided surgery by automating the fluoroscopy machine's setup. The pipeline starts with obtaining training data in form of digitally reconstructed radiographs from an already existing CT-scan, used for preoperative planning. Then, fitting a CNN on this data to regress the six dimensional position vector a X-ray image was taken from, which can be done completely in prior to the surgery. This is followed by positioning the fluoroscopy machine above the patient in the operation room and taking a scout shot. From this image the model regresses its current position. Finally, the machine then calculates the path to travel to reach the desired position w.r.t. the patient.

The results in this work shown that a training data set of DRRs can be generated in roughly five days on a Tesla P4 GPU (mono-process). The developed framework supports parallelizing the DRR's computation as this would speed up the generation time drastically but was not used for time benchmarking. In the second part it was shown that a model indeed can be fitted on the training images to regress the position vector with a MSE loss of 0.00082 on training data and 0.00158 on the final test set. As the scale, prior to standardization, for translation is in millimeters and that for rotation in degrees, the reached MSE shows a very good ability of learning and generalization of the model. To further decrease training time and probably increase generalization, transfer learning could be utilized when retraining a model on a new patient.

The next steps to evaluate and improve this pipeline are to test the model's accuracy on clinical data. Meaning, there's a CT-scan provided for fitting a model coupled with several real X-ray images of the same object with highly accurate orientation information and to implement calculation of movement from the current (regressed) postion, to the desired position. Further, a model's generalization from digitally reconstructed radiographs to real fluoroscopy images poses difficulty, as [31] mentioned. Therefore, achieving good performance on clinical data is an iterative process including adapting the generation of the training data. Finally, for better comparability of models, an error statistic as the pixel wise difference should be established instead of using the mean squared error to capture the actual deviation.

Bibliography

[1] Jayesh Bapu Ahire. *The Artificial Neural Networks handbook: Part 1*. 2018. URL: https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4.

[2] A. Badal and A. Badano. "Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit". In: *Medical Physics* (2009).

[3] Yoshua Bengio. "Learning Deep Architectures for AI". In: *Foundations and Trends® in Machine Learning* 2 (2009).

[4] "Deep scatter estimation (DSE): Feasibility of using a deep convolutional neural network for real-time x-ray scatter prediction in cone-beam CT". In: *SPIE Medical Imaging* (2018).

[5] Drummyfish. *Animation of a plane in Gimbal lock*. 2019. URL: https://commons.wikimedia.org/wiki/File:Gimbal_Lock_Plane.gif.

[6] An Fedorov et al. "3D Slicer as an Image Computing Platform for the Quantitative Imaging Network". In: *Magnetic Resonance Imaging* (2012). URL: https://www.slicer.org/.

[7] Soheil Ghafuriana et al. "Fast Generation of Digitally Reconstructed Radiograph through an Efficient Preprocessing of Ray Attenuation Values". In: *Medical Imaging* (2016).

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Precedings of the 14th International Conference on Artificial Intelligence and Statistics* (2011).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Jiuxiang Gu et al. "Recent Advances in Convolutional Neural Networks". In: *arXiv* (2017).

[11] Maria Eugenia Guerrero et al. "State-of-the-art on cone beam CT imaging for preoperative planning of implant placement". In: *Clinical Oral Investigations* (2006).

[12] J.C. Hay et al. "Mark I Perceptron Operators' Manual". In: *Cornell Aeronautical Laboratory Report No. VG-1196-G-5* (1960).

[13] Josef Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.* 1991.

[14] Langston T. Holly and Kevin T. Foley. "Three-dimensional fluoroscopy-guided percutaneous thoracolumbar pedicle screw placement". In: *ournal of Neurosurgery: Spine* (2003).

[15] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR* (2015).

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* (2015).

[17] Eugene C. Lin. "Radiation Risk From Medical Imaging". In: *Mayo Clinic Proceedings* (2010).

[18] MedicalEXPO. *Digital fluoroscopy system for cardiovascular fluoroscopy with ceiling-suspended C-arm*. 2019. URL: https://www.medicalexpo.com/prod/canon-medical-system-europe/product-70354-758057.html.

[19] Kamimura Mikio et al. "Preoperative CT Examination for Accurate and Safe Anterior Spinal Instrumentation Surgery with Endoscopic Approach". In: *Journal of Spinal Disorders and Techniques* (2002).

[20] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. MIT Press, 1969.

[21] Davic C. Plaut and Geoffrey E. Hinton. "Learning sets of filters using back-propagation". In: *Computer Speech and Language* (1987).

[22] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *Google Brain* (2017).

[23] Frank Rosenblatt. "The perceptron, a perceiving and recognizing automaton". In: *Cornell Aeronautical Laboratory report* (1957).

[24] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[25] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[26] D. Sarrut and S. Clippe. "Fast DRR generation for intensity-based 2D 3D image registration in radiotherapy". In: *Elsevier Science* (2003).

[27] Rajalingappaa Shanmugamani. *Deep Learning for Computer Vision*. Packt Publishing, 2018.

[28] L. Shen, L.R. Margolies, and J.H. Rothstein et al. "Deep Learning to Improve Breast Cancer Detection on Screening Mammography". In: *nature* (2019).

[29] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large Scale Image Recognition". In: *arXiv* (2015).

[30] Christopher Syben et al. *Self-Calibration and Simultaneous Motion Estimation for C-Arm CT Using Fiducial*. Springer Berlin Heidelberg, 2017.

[31] Mathias Unberath et al. "DeepDRR - A Catalyst for Machine Learning in Fluoroscopy-guided Procedures". In: *arXiv* (2018).

[32] *Visible Human Project*. URL: https://mri.radiology.uiowa.edu/visible_human_datasets.html.

[33] Baptiste Wicht. "Deep Learning feature Extraction for Image Processing". PhD thesis. 2018.

[34] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *CoRR* (2017).