Linköping University | Department of Science and Technology
Master's thesis, 30 ECTS | Media technology
2019 | LIU-ITN/LITH-EX-A--19/001--SE

Deep Learning-based Lung Triage for Streamlining the Workflow of Radiologists

Djupinlärningsbaserat Lungtriage för Effektivisering av Radiologers Arbetsflöde

Michaela Rabenius

Supervisor : Daniel Jönsson
Examiner : Daniel Nyström

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

The usage of deep learning algorithms such as Convolutional Neural Networks within the field of medical imaging has grown in popularity over the past few years. In particular, these types of algorithms have been used to detect abnormalities in chest x-rays, one of the most commonly performed type of radiographic examination.

To try and improve the workflow of radiologists, this thesis investigated the possibility of using convolutional neural networks to create a lung triage to sort a bulk of chest x-ray images based on a degree of disease, where sick lungs should be prioritized before healthy lungs.

The results from using a binary relevance approach to train multiple classifiers for different observations commonly found in chest x-rays shows that several models fail to learn how to classify x-ray images, most likely due to insufficient and/or imbalanced data. Using a binary relevance approach to create a triage is feasible but inflexible due to having to handle multiple models simultaneously. In future work it would therefore be interesting to further investigate other approaches, such as a single binary classification model or a multi-label classification model.

Acknowledgments

I would like to thank Sectra Imaging IT Solutions AB for welcoming me and supporting me during the thesis work. Special thanks goes to my supervisor **Grayson Webb** who has been a great support and source of encouragement throughout. Another special thanks goes to my fellow thesis workers at Sectra, who have given me a lot of support and advice when I have felt lost and confused. I also want to thank my supervisor and examiner at Linköping University, **Daniel Jönsson** and **Daniel Nyström**, who have patiently given me helpful advice that has guided me through even the most difficult patches of my thesis. Lastly I would like to thank my family and friends who listened and gave me the energy to continue doing my best whenever I needed it. I could not have done it without your support!

Contents

List of Figures

List of Tables

1 Introduction

Every year many people are affected by lung diseases, ranging from clinical pathologies, such as pneumonia, to life-threatening tumours. To be able to diagnose patients suffering from lung disease, a chest x-ray examination is often performed. Because of the high prevalence of lung diseases, this type of examination have become one of the most common tasks for practicing radiologists.

Despite being so common, diagnosing lung diseases is not an easy task. Some diseases can be near impossible to discover from radiological scans alone and sometimes can only be inferred from other clinical information (such as the previous medical history of the patient). Radiologists spend a considerable amount of time analysing the bulk of radiological images resulting from a chest x-ray examination, time that perhaps could be spent more effectively on other tasks. The pressure on hospitals and general healthcare today is large. Improving and streamlining the workflow of radiologists could therefore aid in both being able to treat more patients by freeing up resources and finding a diagnosis more quickly, which in turn could aid in saving more lives.

One way this could be achieved is by letting a computer analyze the x-rays before passing them on to a radiologist for further assessment. If a computer can analyze an x-ray image and detect abnormalities in a first scan of the problem, it could aid the radiologist by giving some indication where to look when examining and diagnosing the patient. This problem falls under the field of computer vision; giving a computer a high-level understanding of the content in images.

Machine learning has over the last few years been gaining popularity in the medical field. Machine learning is a field that grew out of Artificial Intelligence with the aim of giving a computer the ability to "learn" certain behaviours on its own, as opposed to following hard-coded instructions[1][2]. Machine learning has proven itself to be useful for several different tasks, such as automatization of previously manual tasks. Machine learning techniques can be used in computer vision problems, which has made them interesting for use in medical imaging. Primarily, it is useful for tasks such as object detection and classification.

The usability and accuracy of machine learning algorithms depend heavily of how the underlying data is *represented* and what *features* that representation contains. Choosing a suitable set of features to describe a particular problem can be difficult, sometimes near impossible for a human. A solution to this issue is use *representation learning* techniques, which allows the

algorithm to learn which features are important to the representation in addition to how to best solve the problem.

One kind of representation learning that has been been useful in computer vision is **deep learning**. Using computer vision as an example, a deep learning model can learn to recognize a human face by combining simpler concepts, such as lines and edges. Deep learning models like feedforward neural networks have layers that processes the input data and extracts features that can be combined to form increasingly abstract representations.

One deep learning method that is effective in image analysis is **Convolutional Neural Networks**. A convolutional neural network is a specialized feedforward neural network that works well on images due to its ability to discover patterns in grid-structured data. Convolutional neural networks have been applied successfully to different kinds of medical images to solve various computer vision tasks, such as organ detection and pathology classification.

1.1 Motivation

This thesis explores the possibility of using convolutional neural networks to detect signs of disease and other abnormalities in x-rays of lungs and from it create a triage; a way of prioritizing tasks based on the seriousness of the patient's condition. The idea is to prioritize the order in which a bulk of chest x-ray images should be examined. Essentially, this means that a model performs a first scan of the x-rays and should sort them based on the likelihood of a disease being present before they are viewed by an expert. By prioritizing images with high risk of disease over images with lower risk, it could help the radiologist find signs of the disease more quickly. Applying the lung triage could thus help streamline the workflow of radiologists.

This project was carried out at Sectra AB and the department of Medical Imaging IT Solutions. Sectra is a company that offers products and services within the fields of medical imaging and cybersecurity. The company develops several products with the goal of improving the workflow of healthcare professionals, often related to work with radiographic images. Sectra's main office is located in Linköping.

1.2 Aim

The aim of this thesis is to investigate and evaluate the possibility of creating a lung triage by using convolutional neural networks to classify chest x-rays based on potential lung diseases. The result of the thesis is a lung triage that can sort a bulk of chest x-ray images based the probability of disease in the image, prioritizing images of sicker lungs before healthier ones.

1.3 Research questions

The following research questions will be answered in this thesis:

- How well does using convolutional neural networks work for classifying chest x-rays and how effectively can the resulting classification models be used to create a lung triage?
- How well does training multiple classification models using a binary relevance-approach work for creating a triage compared to other classification approaches, such as multi-label classification (i.e. is it faster, more accurate, etc.)?
- How well can the classification models implemented in this thesis perform on data from different distributions, i.e. can the models give the same level of performance for x-ray images from different hospitals? This is interesting since ideally the models would perform the same on images taken at different hospitals.

1.4 Delimitations

The classification models and the lung triage created in this project are limited in the matter diagnostic use. The models can only give an indication of some disease but is limited from diagnosing the patient with said disease. The actual diagnosis should only be made by a radiologist or doctor. Thus the triage is limited to being only a tool for aiding the radiologist and not a tool capable of diagnosing patients on its own.

The types of diseases that can be discovered is limited to the diseases/observations present in the data set used to train the models.

2 Theory

This chapter will present theory and background information relevant to the thesis. It contains theory behind deep learning and convolutional neural networks, as well as information about training neural networks.

2.1 Deep learning

As mentioned in the introduction, machine learning algorithms have the ability learn how to perform different tasks on their own. The algorithms do this by learning from previously observed data[1]. However, many machine learning techniques are limited in their ability to process data in its raw form[3]. Usually, the raw data must be transformed in some way that makes it understandable to the model. This can be achieved by extracting meaningful features in the data with the help of a feature extractor. However, designing a feature extractor that chooses the best features can be a significant challenge. Because of this, **representation learning techniques** have shown themselves to be very useful since they are able to automatically discover which features make a good representation from the raw data[2]. However, it is still difficult to learn abstract and high-level features from raw data with a lot of variance. This problem is solved by using a special kind of representation learning called **deep learning**.

Deep learning has a long history, having been known under many different names since its conception, and has seen multiple peaks and lows in popularity[2]. Today deep learning refers to a broad collection of models and algorithms that uses multiple layers of processing units to perform representation learning. Compared to other types of representation learning, the layers allow the deep learning models to form increasingly abstract representations of the raw data by combining smaller and simpler representations. The features that are important to the representation are learned from the data itself through some general learning process.

While deep learning as a concept has existed for a long time, it has not been extensively used in practice until recent years. This is mostly because deep learning requires large amounts of data and computational power that has previously not been available. However, with new larger data sets and improved hardware, such as the rapidly improving GPUs (Graphic Processing Units) for parallel computing, deep learning on a large scale has been made possible[2].

2.2 Deep learning in medical image analysis

A field in which the usage of deep learning algorithms has grown in popularity is medical image analysis. In particular, deep learning models like **Convolutional Neural Networks** have gained a lot of attention for being able to perform tasks such as image classification and object detection with good results[4]. Notably, convolutional neural networks have been used to detect different types of abnormalities in chest x-rays. For example, Bar et al.[5] explored different approaches of using pre-trained convolutional neural networks trained on the non-medical data set **ImageNet**[6] to detect lung pathologies in chest x-rays. They found that with their method it is possible to detect pathologies and discusses that their results can be improved by fine-tuning their network with actual x-ray data.

Shin et al. combined convolutional neural networks with reccurent neural networks to both detect a disease from a chest x-ray and describe its contextual information, for example the location or severity[7].

Rubin et al. trained deep convolutional neural networks to automatically classify 13 different diseases in frontal and lateral chest x-rays[8].

Another notable example is CheXNet created by Rajpurkar et al[9]. CheXNet is a 121-layered convolutional neural network trained on the **ChestX-ray14** data set, a data set containing over 100,000 chest x-rays. It takes chest x-rays as input and outputs the probability for the patient having pneumonia with an accuracy rivaling that of a human expert.

2.3 Convolutional Neural Networks

The previous works described in Section 2.2 motivates using convolutional neural networks as a method for creating the lung triage aimed for in this project. To explain the structure of this type of deep learning model, this section will firstly describe the structure of a regular feedforward neural network, and then move on to describe the specialization of it that makes it a convolutional neural network.

2.3.1 Feedforward neural networks

The typical deep learning model is the **Feedforward Neural Network**, also known as *multilayer perceptron (MLPs)* or *deep feedforward network*. These models are a type of *Artificial Neural Network (ANN)*: models loosely inspired by the biological brain. Originally, ANNs were intended to be a computational model for biological learning, but has since found other application areas.

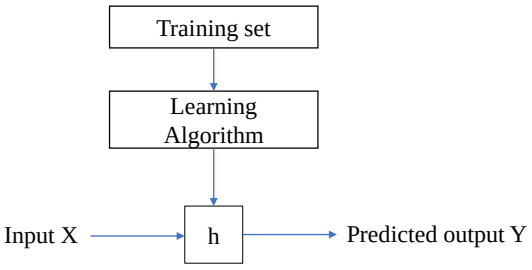


Figure 2.1: The process of supervised learning

Feedforward neural networks can be used to solve various different tasks depending on what type of learning strategy is used. Most commonly, the network is driven by a *supervised learning*-paradigm[3]. Formally, the goal of supervised learning is to learn a function $h(\mathbf{x})$ so that $\mathbf{y} = h(\mathbf{x})$, given a set of input data $\mathbf{x}$ and corresponding output data $\mathbf{y}$[10]. The function $h(\mathbf{x})$, also known as the *hypothesis*, should be able to predict the corresponding output $\mathbf{y}$, also

referred to as the *ground truth*. The network improves its hypothesis by training on a set of input data wit corresponding output, also known as a training set. A scheme of the process can be seen in Figure 2.1.

Supervised learning is often used for **classification** problems, where some input data sample should be classified into one or more discrete categories[10]. By learning the hypothesis that can predict the correct category for multiple input samples, the model can use the same hypothesis to predict the categories for samples it has never seen before. The model's ability accurately predict the output for new data, its ability *generalize*[11], is a large factor in determining the usefulness of the model. Another type of problem that can be solved is regression problems, in which the network should predict a continuous quantity instead of discrete values[10]. Since the aim of this project is to classify whether a chest x-ray contains signs of disease or not, the main focus of this thesis will be on classification problems. Further information about classification can be read in Section 2.4.1.

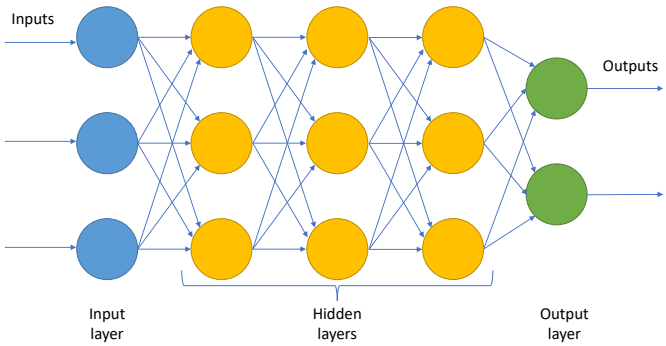


Figure 2.2: A small feedforward neural network with several hidden layers. The input layer(blue), the hidden layers(yellow) as well as the output layer(green) contains artificial neurons which are computational units that processes the input data.

The structure of a simple feedforward neural network can be seen in Figure 2.2. It consists of a number of artificial neurons arranged in layers. Usually, there are at least three different types of layers: the *input layer* which is the first layer that receives the input data, one or more *hidden layers* responsible for processing the input data, and an *output layer* which outputs the final results of the data processing. The more layers in network, the greater the network's *depth*[2].

The artificial neurons in the network are small computational units that together computes the output of the network. Each neuron in each layer has connections to the neurons in the neighbouring layers, as can be seen in Figure 2.2. Layers with these kinds of connections are often referred to as **fully connected layers** or **dense layers**. The purpose of the neurons is to process the input signals from the previous layer and compute a single output that can be forwarded to the next layer. This creates a chain of computations that together form a function for the whole network. Depending on the input to the network, different neurons will be activated and can pass the data forward. This is similar to a biological neuron, which receives several input signals from other neurons and, if activated, can compute and pass on one signal to the rest of the network.

The structure of an artificial neuron can be seen in Figure 2.3. The neuron receives a number of n input signals from the neurons in the previous layer, each with an associated weight θ_i ($1 \leqslant i \leqslant n$) and a bias b. The bias term resolves cases where all input signals are equal to 0, providing the same functionality as the intercept term in a linear equation, meaning it can be used to shift the hyperplane of the hypothesis in the multi-dimensional

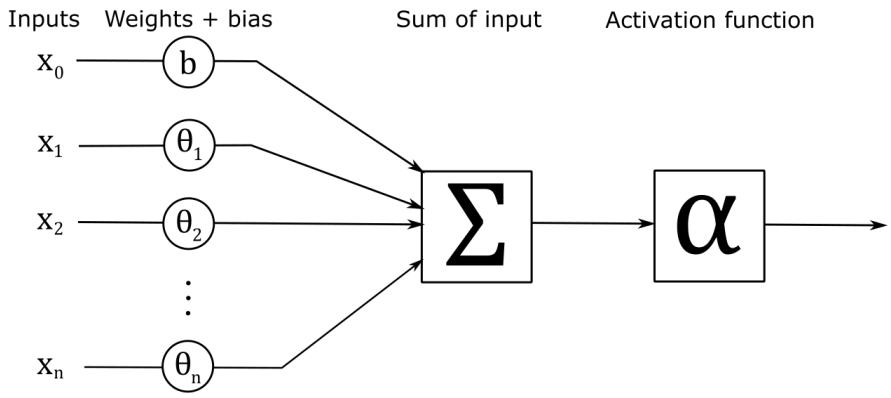


Figure 2.3: Structure of an artificial neuron. The neuron receives a number of input signals x and a bias (here denoted x_0). Each signal has an associated weight θ (the weight for the bias signal is here denoted b). The weighted sum of the input signal is computed and passed forward to the activation functions which will compute an activation value that can be passed along to the next layer in the network if the weighted sum is large enough.

solution space. To pass the signals forward, the neuron must activate and transform them into a single value that can be passed on to the rest of the network. Whether a neuron activates or not is determined by an **activation function**.

Usually, the activation function takes a weighted sum of all the input signals (bias included) as input. If the sum is large enough, the activation function can transform it into a single output that can be received by the neurons in the next layer. The output activation z_j for the j:th layer with n neurons in the previous layer is thus given by applying the activation function α to the weighted sum of input signals x:

$$z_j = \alpha \left(bx_0 + \sum_{i=1}^{n} \theta_{ij} x_i \right) \tag{2.1}$$

There are different kinds of activation functions and the choice of which to use greatly affects the behaviour of the network. The activation function determines the shape of the layer output: most commonly, the activation function maps the resulting weighted sum to a range of values, for example [0, 1] or [-1 1]. By using activation functions it is possible to introduce *non-linearity* to the network which allows the network to model complex, non-linear relationships. Without activation functions, the network can only work well for data that is linearly separable, which severely limits the type of relationships the network can represent. Activation functions are further discussed in Section 2.3.2.

When data is fed to the network, the information is passed forward through the network from the input layer to the output layer which produces the final output. This process is called **forward-propagation**. The value of the final output depends on the set of weights θ in the network. The weights θ, also known as the parameters of the network, can be adjusted to change the final output and receive a better result. The performance of the feedforward neural network can be improved by letting the model learn which weights give the best output for the given input.

When training the network, the output from a forward-propagation pass is passed to a **cost function** $J(\theta)$, sometimes also referred to as a *loss function* or an *objective function*. The cost function is used to calculate how much the output of the network differs from the ground truth, i.e. the model error. The smaller the error, the more accurate the model. To improve its predictions, the network must updates it weights in such a way that it minimizes the cost function. The cost function can be minimized by computing its gradients with respect to the

weights θ and using them in an *optimization algorithm* that updates the weights[2]. Further information about using **optimizers** to update the weights in the network can be read in Section 2.4.5.

A feedforward neural network computes the gradients through a process called **back-propagation**. With back-propagation, the error produced by the cost function flows backwards through the network and is used to compute the gradients one layer at a time[3][2]. The gradient of the last layer of weights is computed first, followed by the second-to-last and so on until the first layer is reached. For each layer, the partial derivatives of the cost function with respect to the weights and biases in the layer are computed. These computations are then reused when computing the gradient for the next layer. Back-propagation can be seen as analogous to the chain rule for calculating derivatives in regular calculus, viewing the network as a compound function of many smaller functions(the neurons) and is a comparatively inexpensive approach for calculating the gradients[2].

2.3.2 Convolutional Neural Networks (CNNs)

A specialization of feedforward neural networks is **Convolutional Neural Networks (CNNs)**, also known as *ConvNets*. CNNs are capable of processing data with grid-like structure[2], for example images which can be represented as a grid of pixel values. The power of CNNs lies in their ability to find and recognize complex patterns in data. For example, a CNN can with good accuracy learn to recognize and locate objects in images, such as vehicles or animals. This is something that while easy for a human, is very difficult for a machine.

What sets a CNN apart from a regular feedforward neural network is that it contains special types of hidden layers that applies **convolution** to the input data. The convolution operation is what makes it possible for the CNN to recognize patterns in the data. In addition to the convolutional layers, a CNN also commonly contains layers that perform **pooling** which is a kind of downsampling. Just like in regular feedforward neural networks, the final layers that determines the final output of the network in a CNN are **fully connected layers**. An overview of the different layers in a CNN can be seen in 2.4.

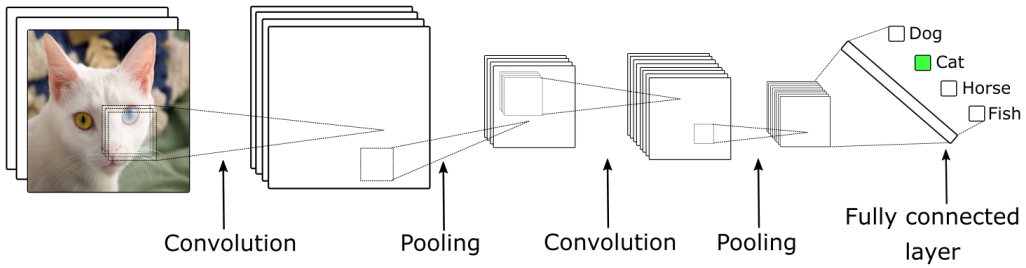


Figure 2.4: An illustration of the different layers in a convolutional neural network. The convolutional layers detect features in the original image and creates feature maps based on those features. The pooling layers downsample the feature maps, increasing the computational effectiveness of the network. Finally, the fully connected layers takes the information produced by the convolutional layers and the pooling layers and produces the final output. In this example, the fully connected layers determines if the image contains a dog, cat, horse or a fish.

Convolution and convolutional layers

Generally speaking, convolution is a mathematical linear operation applied to two functions f and g that produces a third function s that describes how one function is modified by the other, denoted as $s = f * g$. The convolution operation is an integral of the product of the two functions f and g, where one of them is reversed, as can be seen in Equation 2.2. It can

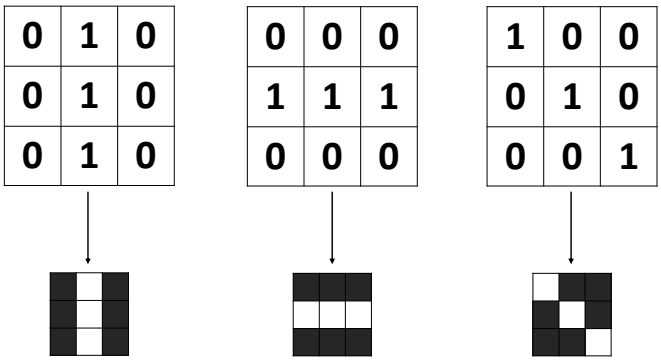


Figure 2.5: Three examples of simple two-dimensional 3X3 edge detecting filters. (Left) Filter able to detect vertical edges or lines. (Middle) Filter able to detect horizontal edges or lines. (Right) Filter able to detect diagonal edges or lines.

be described as letting the function g slide over function f and compute the integral of their product wherever the two functions overlap with respect to some variable t.

$$s(t) = f(t) * g(t) = \int f(\tau)g(t-\tau)d\tau \tag{2.2}$$

In the context of convolutional neural networks, the first argument f is called the **input**, while the second argument g is called a **filter** or **kernel**. The output of the convolution operation is referred to as the **feature map**. In this context, instead of being just functions the input and the filter are usually multidimensional tensors. For example, if the input is an image with three channels, it can be viewed as a three-dimensional matrix of pixel values while the filter is a three-dimensional matrix of learnable parameters.

A feedforward neural network must contain one or more **convolutional layers** in order to be classified as a CNN. As previously mentioned, the convolutional layers are what applies the convolution operation to the input data. The convolutional layers contain a number of filters (i.e. the *weights* of the convolutional layers) that each detect a particular image feature. By training on a set of images, the model can learn which filters find the most relevant features. For example, one filter could be able to find horizontal edges, while another filter is able to find circles, etc. A CNN with great depth and several convolutional layers will be able to learn more advanced filters by combining simpler features detected by other filters in the network, as described in 2.1. A few examples of so called "edge detector" filters that could be used in a CNN can be seen in Figure 2.5.

To explain the process of convolution in CNNs, the input image I can be seen as a simple two-dimensional matrix of different pixel values while the filter K is a much smaller matrix, for example of size 3x3, see Figure 2.6. The convolution operation is performed by sliding the smaller filter matrix a small distance at a time (known as *stride*) over the entirety of the input image. The portion of the input image covered by the filter has its pixel values multiplied with the corresponding values in the filter matrix, resulting in a dot product of the two. This dot product will produce a single value corresponding to the covered area. If the portion of the input image matches the filter exactly, the feature associated with the filter has been found and the result of the convolution operation will be a higher value. Conversely, if the filter does not match the portion of the image, the feature has not been found and the resulting value from the convolution will be small. In other words, the filters will react more to areas where their associated feature can be found.

The resulting values from performing convolution on all pixels in the input image will then represent a pixel value in the outputted feature map S. The feature map in actuality is

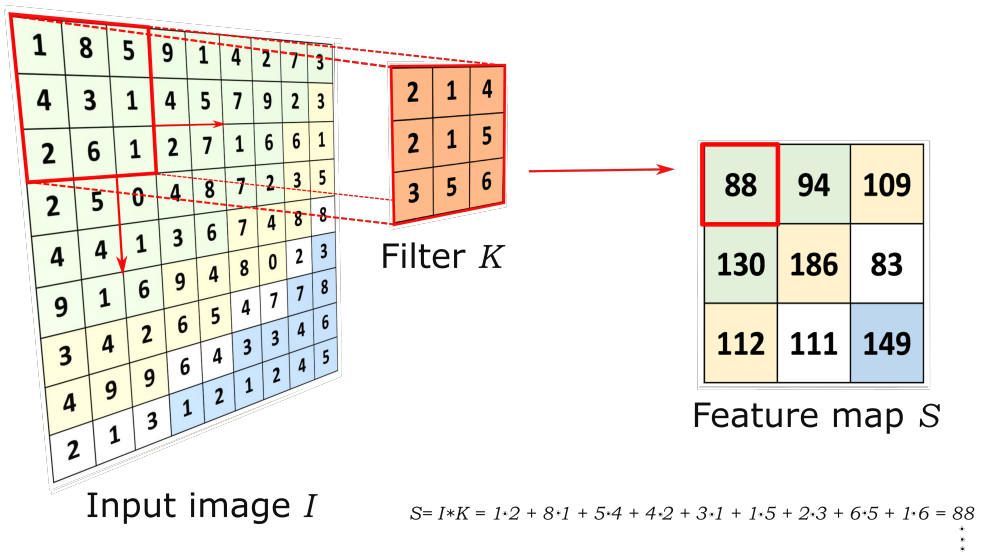


Figure 2.6: Convolution of an image and a small filter. The filter slides over all image pixels and computes the dot product of the image and filter. The result is stored as a pixel value in the corresponding feature map.

a map of *linear activations* that shows where in the image the feature has occurred. This process is repeated for all the filters in the convolutional layer, producing a number of different feature maps, which will be the input to the next layer. Mathematically, the convolution of I and K to produce one feature map S can be described with a double sum, see Eq. 2.3[2].

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.3}$$

To make the linear activations in the feature map non-linear they must be transformed using an activation function. As explained in Section 2.3.1, the activation function determines the output of a layer in the network. Most commonly paired with convolutional layers is the *ReLU*-activation function. ReLU is a simple ramp function that forces negative values to be 0, while positive values are outputted directly, see Eq. 2.4. Using ReLU as activation function is common since it has been shown to enable better training of deeper networks[12] and therefore is a good default choice.

$$ReLU(x) = max(0, x) \tag{2.4}$$

As is most often the case, the input images are not two-dimensional but rather three-dimensional, as images typically contains three different channels (red, green and blue). In this case, the input and filter are volumes instead.

Using convolution in a neural network has some benefits. One of the benefits of CNNs is that they typically have *sparse interactions* (also known as sparse connections), caused by making the filter smaller than the input. Sparse interactions means that that a neuron in the hidden layer is connected to only a smaller fraction of the neurons in the another layer, in contrast to a fully connected layer where one neuron is connected to all other neurons in the other layer. This is because the neuron is connected to a small region of the input image (i.e. the region covered by the filter) and not each pixel in the whole image. This mean fewer parameters need to be stored, which reduces the amount of memory required by the model and improves statistical efficiency[2]. It also means that fewer operations are required to

compute the output of the layer. For example, if there are m inputs and n outputs to one layer, the number of parameters processed in the layer is $m \times n$. If the number of connections is limited to k, this number can be lowered to $m \times k$, see Figure 2.7.

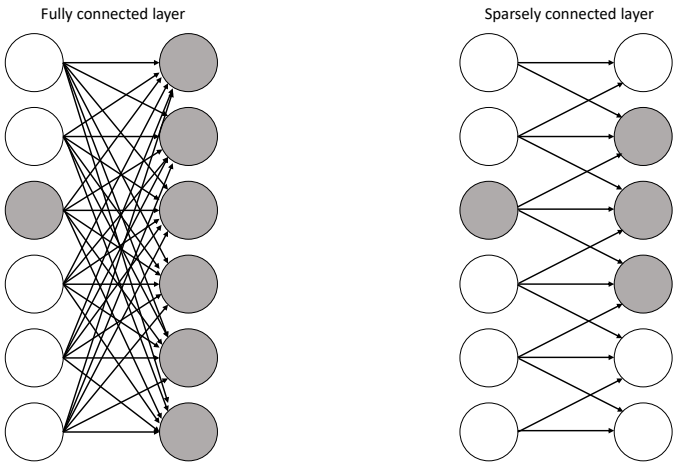


Figure 2.7: Connections between neurons in a fully connected layer (left) and a sparsely connected layer (right). In a fully connected layer, 1 input affects all outputs, visualized as grey nodes. In a sparsely connected layer created by using convolution 1 input only affects a smaller number of the outputs.

Another benefit of using convolution in a neural network is *parameter sharing*. Parameter sharing is when several neurons share the same filter parameters with each other. If a feature found in one location of the input image can also be found at another location, parameter sharing makes it possible to use the same filter to detect both, instead using two filters for the two different locations. This is possible since the filter moves of the entirety of the input image and it further reduces the memory requirements of the model, since there is no need to store filters for every single location in the image.

Pooling and pooling layers

Convolutional layers are often followed by a **pooling layer**. Pooling is a procedure that further modifies the outputs of a convolutional layer by summarizing the response over a neighbourhood in the feature map. Pooling helps reduce the spatial size of the feature map whilst keeping the information about the feature intact, which is beneficial since it decreases the amount of computational power needed to process the data[2].

Similar to the convolution operation, pooling of the feature map is computed by letting a small window slide over the feature map, computing a value from the covered neighbourhood. The most common type of pooling function is **max pooling**[13][14][15] which takes the maximum value within the neighbourhood to represent the neighbourhood as a whole. The window is then moved to compute the values in the rest of the image. The results create a downsampled version of the feature map, see Figure 2.8.

Besides computational benefits, pooling makes the representation of the feature approximately invariant to small translations of the input, meaning that if the input image is moved a small amount, most outputs of the pooling layer will remain the same. This property is useful when the knowledge that a feature simply exists somewhere in the image is more important than its exact location. Since pooling summarizes the information in a neighbourhood, it also makes it possible to use a smaller number of neurons to process the pooled output, compared

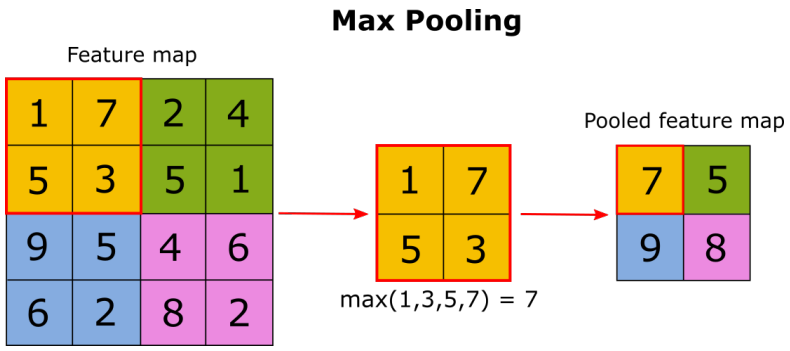


Figure 2.8: Example of max pooling. The feature map is downsampled by summarizing the information in a specific region by taking the maximum value in that region.

to the amount needed to detect the features. This leads to further increased computational efficiency as the next layer in the CNN can process a smaller number of inputs.

The final fully connected layers

A CNN may contain many alternating convolutional and pooling layers, but the final layer or layers are usually fully connected layers like in a regular feedforward network. The last fully connected layer in the network (the output layer) is responsible for computing the final output based on the input it receives from the preceding layers, i.e. which features were detected by the convolutional and pooling layers[16].

Depending on the nature of the problem the CNN is trying to solve, the output of the network is different. For example, the output be a single number or a vector of numbers. The form of the final output is determined by the number of neurons and the choice of activation function in the output layer. Further information about activation functions can be read in Section 2.4.3.

2.4 Training a Convolutional Neural Network

This section describes the process of training a CNN to perform classification. Different types of classification problems will be described and how to design the network according to the situation.

2.4.1 Different kinds of classification problems

The design of a CNN, i.e. the number of layers used in the network, what kind of cost function is used etc., depends on the type of problem that the CNN is trying to solve. As mentioned in Section 2.3.1, CNNs are commonly used in classification problems, where the CNN is a classifier that should classify images into different discrete categories. There are mainly three different types of classification problems:

- Binary classification
- Multi-class classification
- Multi-label classification

In binary classification, the input image belongs to one of two classes, usually a positive class and a negative class, see Figure 2.9. The output of a binary classifier can be a single value in the range $[0, 1]$, where a value closer to 1 indicates belonging to class A while a

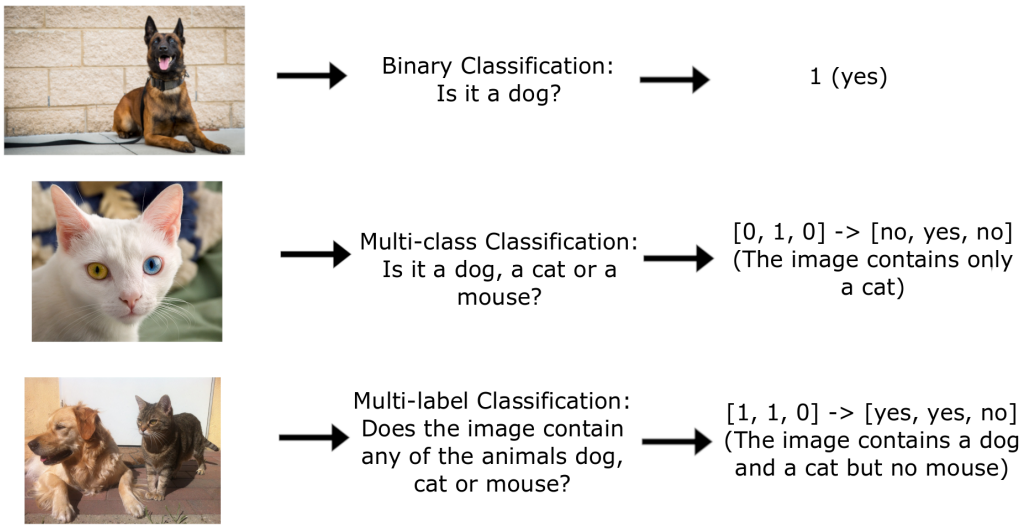


Figure 2.9: Different kinds of classification problems. The output of the network is either 1 (positive) or 0 (negative) for each class.

value closer to 0 indicates belonging to class B. Binary classification is often used to answer "yes/no" questions, e.g. does this object exist in the image, yes or no?

In multi-class classification the input image can belong to one of multiple classes. For example, a multi-class classifier can look at an image of an animal and determine whether that animal is a dog, a cat, a mouse, etc., see Figure 2.9. The output of a multi-class classifier is usually a probability distribution of the different classes, where the image is classified as the class with the highest probability.

Multi-class classification works fine for images containing only one type of animal. However, more often than not the image will contain several different types. To be able to classify an image containing both a dog and a cat, the image must be able to belong to both categories. This is solved by multi-label classification which allows the image to independently belong to multiple classes simultaneously[17][15], see Figure 2.9.

Compared to both binary and multi-class classification where data is associated to only a single class label, in multi-label classification the data is associated to a set of class labels. This makes multi-label classification a more complex problem. To train a CNN to perform multi-label classification may therefore require some special strategy[17][18]. One such strategy is the *Binary Relevance method*[18], which simplifies a multi-label problem of n classes by dividing it into n different binary classification problems, one for each class. This is a simple method that assumes that there is no dependence between the different class labels.

2.4.2 Data sets used for training

Training a CNN can be divided into different phases: usually there is a training phase and a test phase. During the training phase, the model has to learn some type of hypothesis from some input data. During the test phase, the models performance is tested to see if the model has learned anything valuable from training. During both phases, the CNN is fed a set of images with corresponding output class labels. The set used during the training phase is referred to as the *training set* and will influence how the weights in the network changes[10]. The set of images used for testing, the *test set*, tests how well the CNN can generalize to new data it has not seen before. For this reason, the test set should be completely separated from

the training set with no overlap. Sometimes a *validation set* is used as well. The validation set can be used to confirm that the model generalizes during training and give an indication of what external parameters can be changed to improve the model.

Typically when training a network, all the data is in only one big data set. This data should be split into the three different sets needed. There are different ways of splitting the data, and which strategy is best depends on several different factors, such as the size of the data set or its general complexity. As described by Goodfellow et al.[2], one common practice is to first make a 80-20% split and let the 20% be the test set. The remaining 80% is then split again at 80-20% and the second 20% is used as the validation set. The remaining data form the training set.

2.4.3 Activation function for the output layer

Activation functions, introduced in Section 2.3.1, are an important part of the CNN. One of the most important choices when designing the CNN is what activation function to use in the output layer as it determines the output of the entire network, as mentioned in Section 2.3.2.

Different activation functions are suitable for different things. For example, the **sigmoid** function(Eq. 2.5) is useful in binary and multi-label classification as it outputs a value in the range of [0, 1], see Figure 2.10 for an example.

$$\alpha_{sigmoid}(x_j) = \frac{1}{1 + e^{-x_j}} \tag{2.5}$$

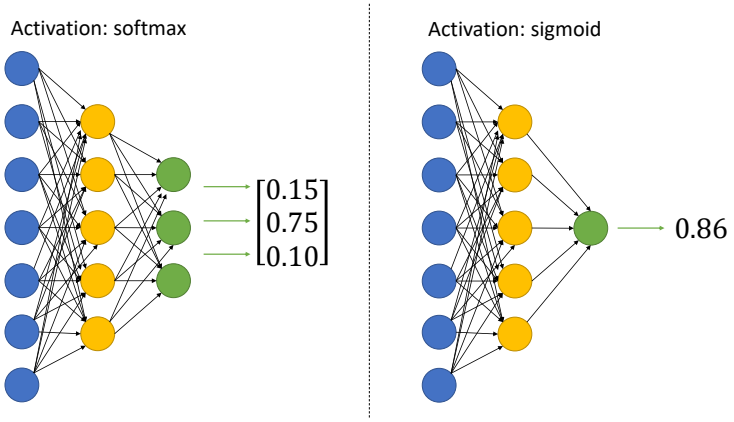


Figure 2.10: Two networks with different output activation functions. (Left) a multi-class classifier that classifies three different classes with softmax as its activation function. The output is a probability distribution between the different classes. (Right) a binary classifier that classifies two classes. The output is a single value between 0 and 1.

A common activation function used in multi-class classification is the **softmax** function, see Eq. 2.6. The softmax function takes a vector of n values and normalizes it into a probability distribution of n probabilities, meaning that the softmax function can be used to compute the probability of a object belonging to one of several different classes. Because it is a probability distribution, the sum of all values produced by the softmax function is 1. The higher the value outputed by the softmax function, the higher the probability it belongs to that particular class, see Figure 2.10.

$$\alpha_{softmax}(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}, \tag{2.6}$$

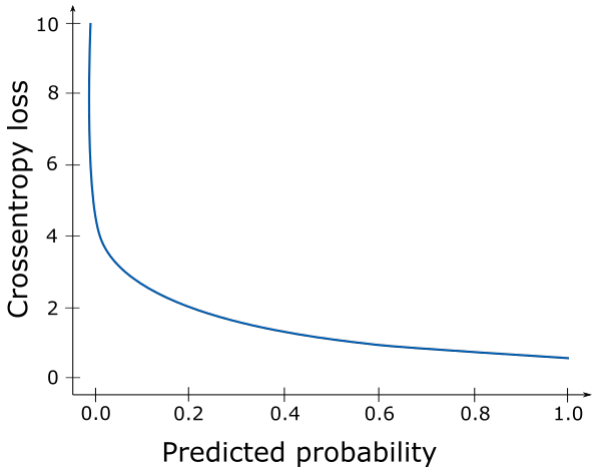


Figure 2.11: Example plot of the predicted probability against the cross entropy loss.

2.4.4 Choosing the cost function

The choice of cost function is closely linked with the choice of output activation function, as it takes the output of the activation function as an input. As explained in section 2.3.1, the cost function is used to calculate the model error which is then used to train the model. How that error is calculated is therefore very important.

There are several different kinds of cost functions. For classification problems it is common to use *cross entropy loss* as the cost function[2]. The cross entropy can be used to measure the error between two probability distributions y and $\hat{y}$[19] and is given by:

$$J(y, \hat{y}) = -\sum_{i=1}^{N} y_i log(\hat{y}_i) \tag{2.7}$$

In a deep learning context, the distribution y over N classes can be the ground truth label probabilities while $\hat{y}$ is the predicted label probabilities outputed by the model. The cross entropy loss measures how close the predicted distribution is to the true distribution, i.e. how "far away" the prediction is from the ground truth.

The value computed by the cross entropy loss increases when the label probability predicted by the model diverges from the the ground truth labels. This means that if the model predicts 0.1 when the actual answer is 1, the error value given by the cross entropy will be high, see Figure 2.11.

Depending on the type of classification problem, there are two variants of cross entropy loss that can be used: *binary cross entropy* and *categorical cross entropy*. Binary cross entropy is a special case of categorical cross entropy, suitable for binary classification (i.e. when the number of classes $N = 2$). Categorical cross entropy in turn can be used to compute the cost for multi-class classification problems with more classes.

As Chollet explains: different combinations of output activations and cost functions work well for certain problems[15]. Table 2.1 summarizes which type of cost function and output activation function is suitable for the three different kinds of classification problems.

Table 2.1: Suitable output activations and cost functions for different types of classification problems.

Problem type	Output activation	Loss function
Binary classification	sigmoid	binary crossentropy
Multi-class classification	softmax	categorical crossentropy
Multi-label classification	sigmoid	binary crossentropy

2.4.5 Optimizers

As described in section 2.3.1, training a regular feedforward neural network is done by updating the weights in the network to minimize the cost function. This process also applies for CNNs and is done with the help of an optimization algorithm, a.k.a. an **optimizer**.

The most basic optimizer is *gradient descent*. In gradient descent the weights are updated by following the negative gradient direction of the cost function[2]. Since the gradient gives the direction in which the function is growing the fastest, following along the opposite direction will lead to a smaller value and eventually to a minimum. Thus, the gradient gives an indication in which direction the weights should change (i.e. whether they should be increased or decreased). A simple example of this can be seen in Figure 2.12. As explained in Section 2.3.1, the gradients of the cost function with respect to the weights are computed using back-propagation.

The optimizer takes a small step in the the negative gradient direction towards the minimum of the cost function and updates the weights accordingly. By repeatedly taking small steps in this direction, the minimum of the cost function can be reached. The size of the step taken to reach the minimum is known as the **learning rate**.

The learning rate infers how much the weights should change during each update and thus has a significant effect on how fast or well the optimizer will work. By using a large learning rate it is possible to find minimum point quickly, but it is also possible to overshoot and miss the minimum completely, which in turn can make the model worse. Decreasing the learning rate may help in avoiding overshooting the minimum, but it will also require many more steps and longer time to reach it, see Figure 2.12. It is also possible for the optimizer to find a local minimum and get stuck, before it can find the better global minimum. In this situation it could potentially be good to have a larger learning rate and overshoot the local minimum in order to find the global minimum.

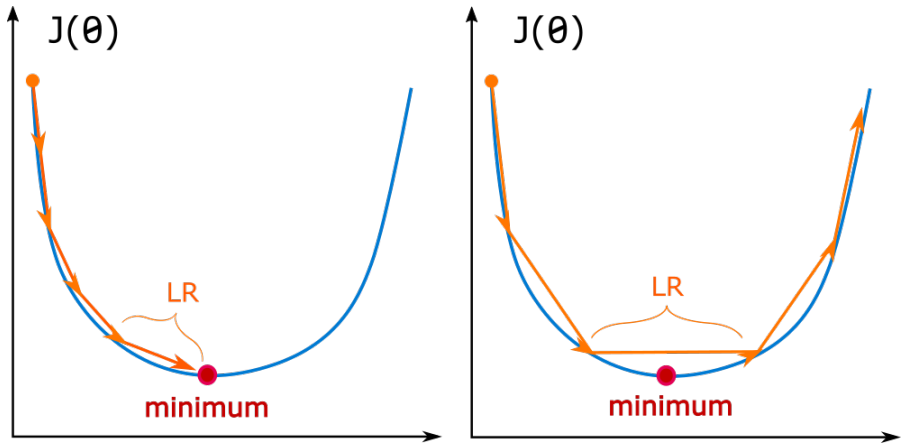


Figure 2.12: The effect of varying learning rates on a cost function $J(\theta)$. By using a too large learning rate, it is possible to overshoot and miss the minimum of $J(\theta)$ completely (Right). To ensure this does not happen, the learning rate can be lowered but this also means more steps are required to reach the minimum (Left).

Over the last couple years, many other optimization methods have evolved from gradient descent. One popular type is optimization algorithms with adaptive learning rates, which often uses the concept of *momentum*[20]. Momentum can accelerate training by moving in the direction of an exponentially decaying moving average, accumulated from past gradients[2]. This means that by using momentum, the optimizer can adjust the learning rate to prevent oscillations when searching for a minimum and thus reach the goal more quickly.

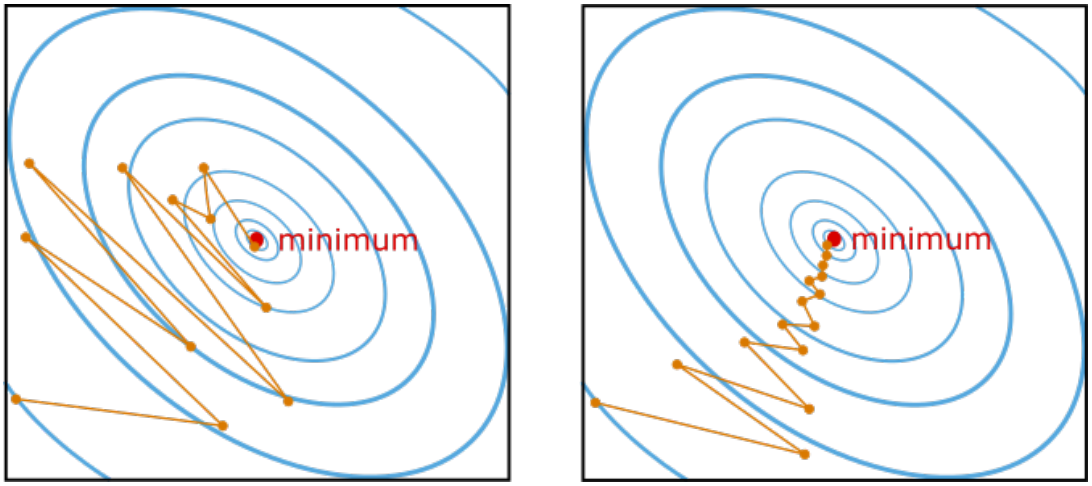


Figure 2.13: Visualization of momentum. Without momentum, the steps taken towards the minimum can oscillate a lot instead of moving along the straight path to the minimum (left). By using momentum the learning rate can be adjusted, leading to a accelerated steps towards the minimum (right).

One optimizer that makes use of adaptive learning rates is **Adam**(*Adaptive Moment Estimation*)[21], which can be seen as a combination of two other famous optimization algorithms: **AdaGrad**[22] and **RMSProp**[23]. In comparison to regular gradient descent, Adam computes individual learning rates for each parameter. Adam stores an exponentially decaying average of past gradients m_t(like momentum) as well as an exponentially decaying average of past *squared* gradients v_t, see Eq. 2.8. m_t and v_t are estimates of the first moment (the mean) and second moment (the uncentered variance) of the gradients respectively, while β_1 and β_2 are decay rates.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{2.8}$$

To counteract biases, m_t and v_t are bias-corrected according to:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.9}$$

Adam then updates the weights in the network according the update rule in Eq. 2.10, where η is the varying step size and ϵ is a small number used to prevent division by 0 (commonly set as 10^{-8}):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{2.10}$$

Kingma et al. shows that Adam can achieve a better result compared to other similar optimization algorithms[21] and using the Adam optimizer is usually a good default choice[24].

However, Adam has also been shown not to converge to an optimal solution and may not generalize as well as other optimizers[25]. It should therefore be noted that each optimizer comes with its own advantages and disadvantages and may work well for different types of problems

Mini-batches

Calculating the weight updates for all training examples in the training set is very expensive and it reduces the training error by a comparatively small amount. In practice it is therefore more efficient to divide the data set into smaller subsets, called **mini-batches**, and do the weight update computations based on the smaller batches instead[2]. Computing the updates for smaller batches of images gives the approximately same result with a considerable efficiency gain.

While training, the model is fed example images to learn from. One pass through all example images in the data set is known as an **epoch**. Without batches, the entire data set is fed to the network at once, meaning it only takes one iteration or step of the learning algorithm (i.e. one forward pass and one backward pass) to complete one epoch. If the data set is divided into smaller batches, more steps are needed to complete one epoch, as only a smaller number of images are contained in one batch. How many training examples that are in one batch passed through the network at one time is determined by the batch size. For example, if the number of training examples is 1000 and the batch size is 100, 10 steps are needed to complete one epoch. During each iteration the weights of the network are updated, which means that by using batches more updates can be done during one epoch.

The choice of batch size depend on several things. Larger batch sizes can give more accurate gradient estimations, but smaller batch sizes require less memory. For this reason, batch size is usually limited by the hardware used. It is common to use a batch size that is a power of 2 (8, 16, 32, 64 etc.), since some hardware (e.g. GPUs) can achieve a better run time with these sizes[2]. Using small batches can also give a regularizing effect, possibly due to added noise in the learning process[26].

2.4.6 Training a CNN to convergence

The goal of training a CNN classifier is to approximate the relationship between input images and the output labels so it can be used to predict the labels for new images. Training a CNN to map images to certain class labels is done by feeding the network a training set of images and letting the network iteratively learn which image features are important to a particular class. The process of training a CNN can be summarized in a couple of steps:

1. **Initialization of all weights and biases** in the network. The weights can be initialized randomly or with predetermined values. It has been shown that initializing the weights according to some distribution is usually makes training converge more quickly with lower error rates [27] [28].

2. **Forward propagation of data through the network**. The output will be a prediction of the output based on the input image.

3. **Computing the cost function**. Using the prediction output gained from performing forward propagation, the cost function can be computed. The cost function gives an error value that indicates how close the predicted output is to the actual output.

4. **Update the weights using an optimizer with back-propagation**. The error value is sent back through the network in order to update the weights using an optimizer and back-propagation.

For the model to learn, step 2 to 4 should be repeated multiple times. Over time as the model trains, the *training error*, i.e. the error rate for the training set, should decrease. The lower the training error is, the better the models predictions are on the training set. When the training error converges it means that a minimum of the cost function has been found. Consequently, one goal of training is to make the training error converge to a small value.

While the model may have a good training error, the same may not be true for images not in the training set: the model may start to memorize the training set, and can thus not generalize well to new images. It is therefore important to test the model's ability to generalize with the help of a test set once the model has been trained to convergence. The error from testing the model on a set different from the training set is referred to as the *generalization error* and should decrease as the model learns. Typically, it is desired to have a small training error and that the gap between training error and generalization error is small as this means that the model is both accurate and is able to generalize, see Figure 2.14.

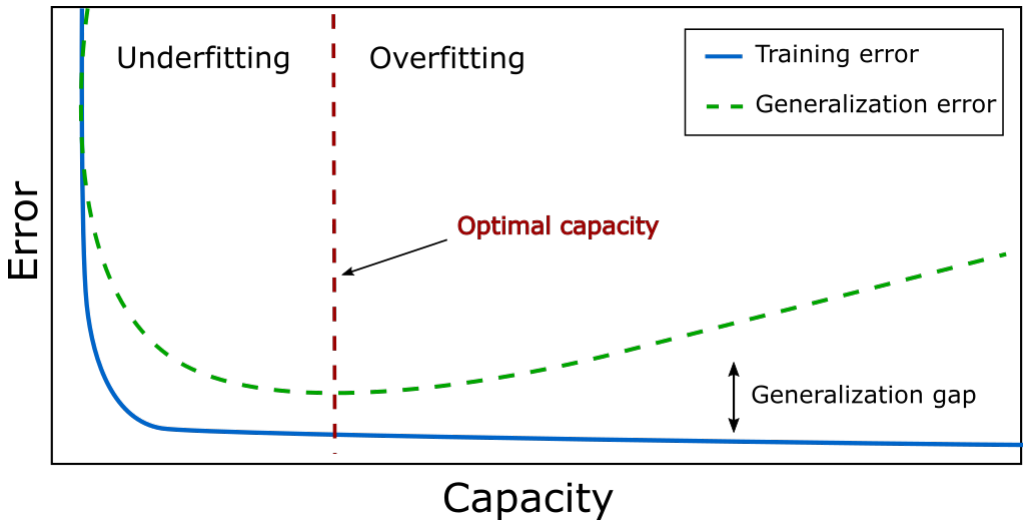


Figure 2.14: Training error and validation error over capacity. As the model learns, its training- and validation error decreases, making it less underfit. As capacity increases, the training error and validation error may converge, making the model overfit. The challenge is to train the model to an optimal capacity(red line) before it overfits.

Two notable challenges when training a neural network are **underfitting** and **overfitting**. Underfitting is the result when the model is not able to fit the data (i.e. map the image to correct label) at all (see Figure 2.15), usually caused by a lack of training data or not training long enough. There is simply not enough data for the model to learn from in order to make a good approximation of the relationship between input and output. An underfit model is not able to accurately predict outputs of either the training set nor the test set, leading to both high training error and generalization error, see Figure 2.14.

On the other end, overfitting is the result when the model tries too hard to fit the training data, i.e. it memorizes the training set, see Figure 2.15. A model suffering from overfitting may give a small training error but have a large generalization error, see Figure 2.14. The model has approximated a function that fits the training data extremely well but is not general enough for data not in the training set, which makes it useless for most use-cases.

The ideal model should both fit the training data and be able to generalize well, a state somewhere between underfitting and overfitting where the model is "just right", see Figure 2.15. As explained by Goodfellow et al., the likelihood of a model underfitting or overfitting can be controlled via its **capacity**[2]. The capacity of the model refers to the range of different types of functions the model can learn in order to map input data to output data. A model with low capacity can only learn a small set of functions, making it unable to model complex

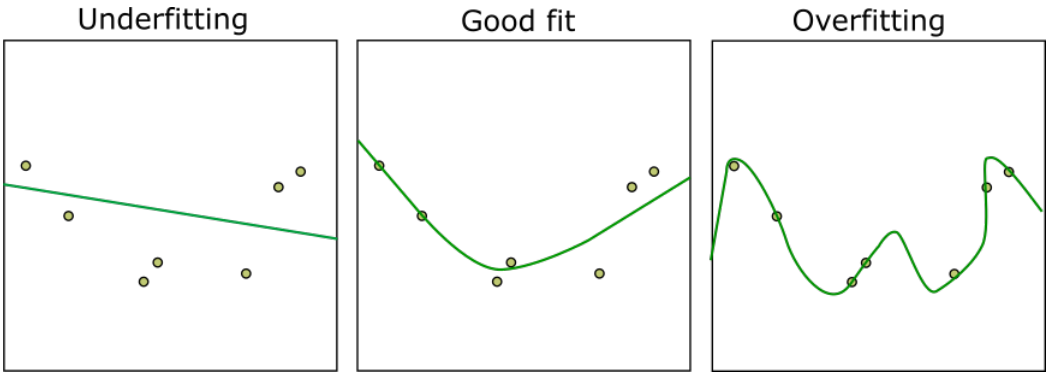


Figure 2.15: Examples of overfitting and underfitting

relationships and is therefore prone to underfitting. A model with high capacity can learn more functions but can also start memorizing the training set, causing the model to overfit. As the model learns during training, its capacity will increase and its error will decrease. However, at a certain point the capacity may become too high, causing the model to overfit, which increases the generalization gap between training error and validation error as can be seen in Figure 2.14. Therefore, it is desired to find the optimal capacity where the training error is low and before the model starts to overfit, see Figure 2.14.

There are measures that can be taken to avoid underfitting and overfitting. A model that is prone to underfitting can be improved increasing the size of the training set, giving it more data to learn from. Underfitting can also be avoided by increasing the number of nodes and layer in the network, effectively increasing the amount of learnable parameters. This allows the model to learn more complex relationships.

If the model contains too many learnable parameters overfitting may occur. Overfitting may be prevented by training with more data, removing excessive input features or by using some kind of *regularization technique*. Regularization techniques are methods that "forces" the model to be simpler. As described by Goodfellow et al., *"Regularization is any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error"*. Note that there is no guarantee that these strategies will always prevent overfitting.

A common type of regularization in deep learning is *early stopping*[2]. With early stopping, the weights are saved every time the model improves. If the model starts to overfit, it will stop improving but the best weight configuration can still be retained. This way it is possible to return a point before the validation error starts diverging from the training error and still get a good model even if it overfitted during training. This also means that the model can stop training early, as a model is unlikely to improve once it has converged or overfit.

Improving a CNN by fine tuning hyperparameters

Most parameters in the CNN are learned automatically through the learning process. There are however some external parameters of the network that can be manually manipulated to improve the network. These external parameters are referred to as *hyperparameters*.

In contrast to the weights in the network, the hyperparameters are not affected by the model itself but choosing a good set of hyperparameters can improve the model performance. Hyperparameters typically controls the model's behaviour in various aspects. The size of the training set, regularization techniques, learning rate, batch size, the number of epochs to train, etc., are all different kinds of hyperparameters that affect the model behaviour in some way.

A big part of improving a deep learning model is fine tuning the available hyperparameters. For example, a model stuck in a local minimum may be unstuck by increasing the learning rate. Fine tuning the model is done with the help of a validation set. The validation

error can give indications on what hyperparameters need to change and can thus be used to indirectly affect the model weights. This is the reason that the validation set is separate from the test set: as previously mentioned, the test set must be completely unknown to the model in order to test its generalization abilities. The model can be fine tuned to give a good validation error, but if it performs poorly on the test set it is still not a good model.

Improving CNN through transfer learning

To perform transfer learning in practice, some layers in the network are frozen, i.e. their weights will not be updated when the model trains. Usually, it is the layers in the first portion of the network that are frozen while the later layers remain trainable. This is because the early layers in the network learns more basic features that are then used to form the more advanced features in the later layers. Very often when training models using pre-exisiting models and transfer learning, all layers but the final output layer are frozen, which can then be trained to give the correct output.

2.5 Evaluation metrics

Evaluation metrics are used to determine the quality of a trained CNN model and different metrics that are well-suited to different scenarios. One of the most common metrics used to measure the quality of a model is to look at its *accuracy*, i.e. the fraction of correctly classified examples as a percentage. Models with high accuracy are able to correctly classify most test examples compared to models with low accuracy. Accuracy as metric can be misleading if there is a large imbalance in the number of examples belonging to each class. For example, if 99% percent of all examples belong to class A, it is possible to get an accuracy of 99% by simply classifying all examples as class A. Therefore it may be wise to use other metrics if this is the case.

Since the error value given by the *cost function* gives the difference between actual and predicted output it can also be used a metric for the model quality. If the model error for a set of example inputs is small, the better the model was able to predict the correct output. Since the cost function sums up all errors made on the data it can be a more reliable metric than the accuracy percentage in cases such as the one explained above.

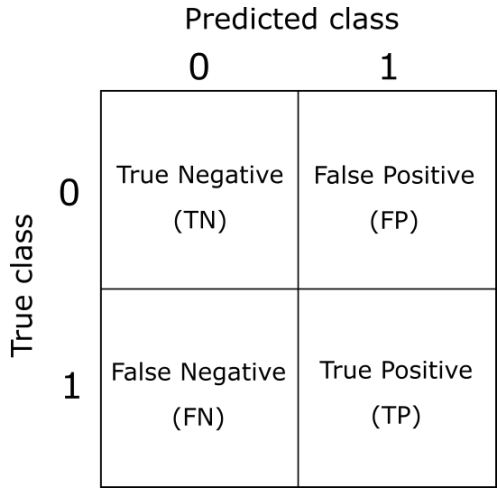


Figure 2.16: A confusion matrix showing the four possible outcomes of classification for some input. Here, 1 represents that the input belongs to a class while 0 represents the opposite. If the model predicts 0 and the true answer is 0 it is a true negative (TN) and if it predicts 1 and the true answer is 1 it is a true positive(TP), meaning that the model correctly classified the input. If the model predicts 0 when the true answer is 1 it is a false negative (TN) and if it predicts 1 when the true answer is 0 it is a false positive (TP), meaning that the model incorrectly classified the input.

For a binary classifier, there are four useful properties that can be used to infer how well the model performs: the number of *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)* and *False Negatives (FN)*. These properties are given by the four different possible outcomes of when the model makes a prediction, see Figure 2.16. TP is the number of positive examples that were correctly classified as positive and TN is the number of negative examples correctly classified as negative. Conversely, FN is the number of positive examples that were incorrectly classified as negative and FP is the number of negative examples incorrectly classified as positive. These properties can be visualized using a *confusion matrix* which shows the number of examples that were correctly classified in its diagonal (see Figure 2.16)[19]. Ideally, both FN and FP should be 0 since it means the model was correct for all examples. Because of this, TP, TN, FP and TN are used to define several other evaluation metrics.

Since the network usually outputs values in the range between 0 and 1, the values must be thresholded to either 0 or 1 for classification. Different classification thresholds can lead to different results, see Figure 2.17. If there is an overlap between positive and negative examples in the data set, varying the threshold may increase the number of TP but also the number of FP and vice versa. The classification threshold thus affect metrics that uses these properties.

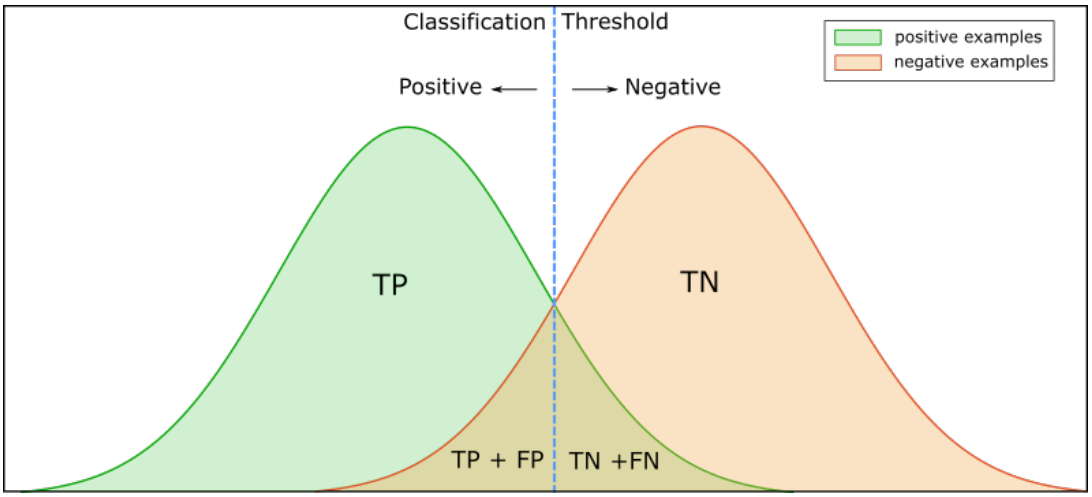


Figure 2.17: Classifying data with overlapping examples using a classification threshold. Everything to the left of the threshold line(blue) is classified as positive while everything to the right is classified as negative. By moving the threshold to the right, the amount of true positives will increase but so will the amount of false positives. By moving it to the left, the amount of true negatives will increase but so will the amount of false negatives.

Two important metrics that take class imbalance into account are **precision** and **recall**[2]. Precision gives the fraction of positive predictions made by the model that were actually correct. For example, if a classifier that classifies whether a patient has cancer or not has a precision of 0.75, it is correct 75% of the time when it predicts that a patient has cancer. Precision can be calculated using Eq. 2.11.

$$Precision = \frac{TP}{TP + FP} \tag{2.11}$$

Recall gives the fraction of actual positives that were predicted correctly by the model. If the same classifier has a recall of 0.15, it means it correctly classifies 15% of all actual cancer patients. Recall can be calculated using Eq. 2.12.

$$Recall = \frac{TP}{TP + FN} \tag{2.12}$$

Precision and recall both output a value in the range [0, 1]. Ideally, both precision and recall should be close to 1. However, improving one them usually has the trade-off of reducing the other. Even if all detected positives are true (i.e. precision is 1), there may be several missclassified positives as well, resulting in a lower recall. A challenge is therefore to find a good balance where both precision and recall are as high as possible. Depending on the problem, high precision may be prioritized over high recall or vice versa.

Precision and recall are used to calculate several other evaluation metrics. One such metric is the **F1 score**, which is a score between 0 and 1 that summarizes precision and recall into a single value: the closer to 1 the better. The F1 score is the harmonic mean of precision and recall, see Eq. 2.13. Note that the F1-score can only be computed for a fixed classification threshold.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.13}$$

Another important metric for binary classification is the **receiver operating characteristic curve**(**ROC curve**)[19]. The ROC curve is a plot of the *true positive rate* (recall) against the *false positive rate* ($FP/(FP + TN)$) at different classification thresholds, see Figure 2.18, and it shows how well the model can distinguish between two classes. The *AUC*, the Area Under

the Curve, summarizes the quality of the ROC curve into a single value[19]. The higher the AUC, the better the model is at classifying positives as positives and negatives as negatives. The optimal ROC curve gives the biggest AUC, meaning that the true positive rate should be maxed out when the false positive rate is low, bringing the curve closer to the upper left corner. In this case, the model is able to separate the two different classes perfectly. The worst case scenario is given when the ROC curve is a straight diagonal line and the AUC equals 0.5 (in Figure 2.18, this worst case base line is plotted as a dashed line). In this case, the model cannot distinguish between the classes at all and can do no better than guess the classes randomly. The ROC metric can be extended to multi-class and multi-label classification by computing the metrics for each class separately and then computing an average.

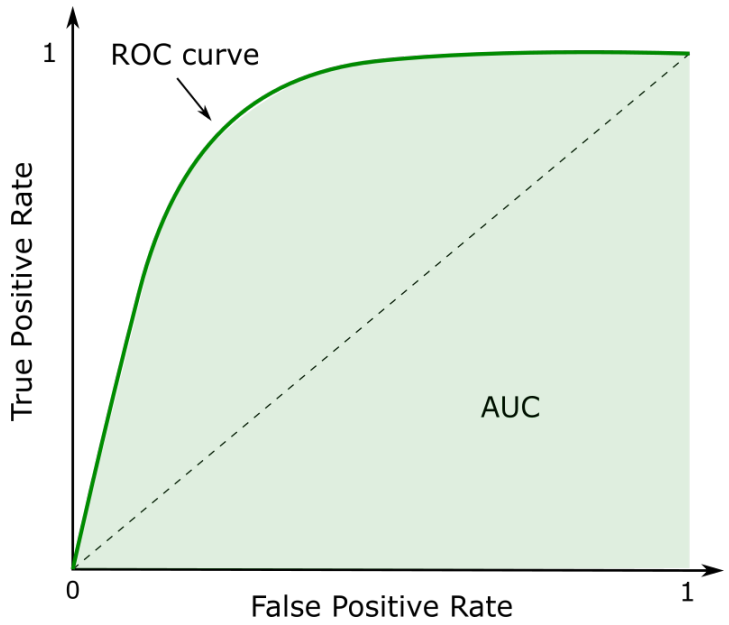


Figure 2.18: Example of a ROC curve and corresponding AUC (Area Under the Curve). The closer the ROC curve is to the upper left corner, the more accurate the model. The dashed line gives the worst case baseline where the model cannot at all distinguish between classes.

While the AUC-ROC metric is recommended for evaluating binary classifiers over the model accuracy metric[29], its results can give an overly optimistic view of the model performance if the there is a large class imbalance in the data set[30]. An alternative to ROC is **Precision-Recall curves (PR curves)**, which are better at taking class imbalance into account. The PR curve plots precision against recall over varying thresholds, as can be seen in Figure 2.19. The optimum PR curve is close to the upper right corner, where both precision and recall are high. Like the ROC metric, the PR curve can be summarized into a single value by computing the AUC. The AUC is approximately the mean precision averaged over recall values. The higher the AUC (with maximum being 1), the better the PR curve is. Davis and Goadrich explains that the PR curve manages to capture the effect of overwhelmingly many negative examples by comparing false positives to true positives instead of to true negatives as is the case with ROC. Davis and Goodrich also explains that an algorithm that optimizes the ROC-AUC is not guaranteed to optimize the PR-AUC.

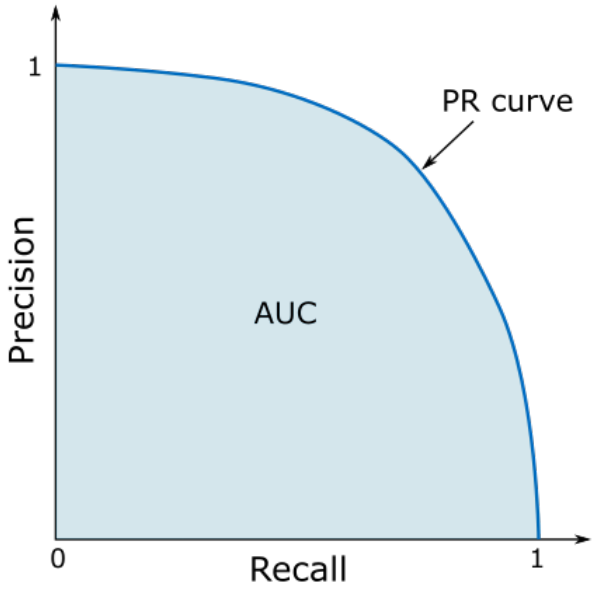


Figure 2.19: Example of a PR curve and corresponding AUC.

3 Method

To solve the task of the project and answer the research questions, a number of CNN classifiers were trained. Three different approaches of training the CNNs were investigated. One of these approaches was then used as basis for a lung triage. This chapter presents the different approaches and the methods used to train them. An overview of the hardware and frameworks used to carry out the training of the classifiers will be presented, as well as information about the data set used for training. Details and motivations for the CNN implementation are then presented and discussed. Lastly, the evaluation metrics used to determine the quality of the models.

3.1 Frameworks and hardware used

The CNN classifiers were implemented using **Keras**[31]. Keras is a high-level deep learning API written in Python that runs on top of other machine learning frameworks like **Tensorflow**[32] or **Theano**[33]. In this project Tensorflow was used as the base framework. Keras supports implementation of CNNs and can run on both CPU and GPU. In this work, a GPU and the parallel computing platform **CUDA**[1] was used, allowing for acceleration of the learning process. To use the GPU efficiently to perform deep learning, the **cuDNN**[2] library (a deep neural network library for NVIDIA CUDA) was used. The graphics card used for training was the NVIDIA GeForce GTX 1060 with 6GB memory.

3.2 Data set

The data set used for training the different classifiers was **CheXpert**, created by Irvin and Rajpurkar et al.[34]. CheXpert is a publicly available data set consisting of 224,316 chest radiographs from 65,240 patients, labelled with 14 observations commonly found in chest x-rays. The CheXpert dataset comes in three different subsets: a larger training set (223,414 images from 187641 different studies), a smaller validation set (234 images from 200 different studies) as well as a test set used for a competition. At the time of writing, the test set is not publicly available so it was not used in this project.

[1] CUDA: `https://developer.nvidia.com/cuda-zone`
[2] cuDNN: `https://developer.nvidia.com/cudnn`

Table 3.1: The CheXpert training set labels consisting of a total of 223,414 images. The table shows the number of images labeled either positive, negative, uncertain or not mentioned in each label category.

Label	Positive	Negative	Uncertain	Not mentioned
No Finding	22381	0	0	201033
Enlarged Cardiomediastinum	10798	21638	12403	178575
Cardiomegaly	27000	11116	8087	177211
Lung Opacity	105581	6599	5598	105636
Lung Lesion	9186	1270	1488	211470
Edema	52246	20726	12984	137458
Consolidation	14783	28097	27742	152792
Pneumonia	6039	2799	18770	195806
Atelectasis	33376	1328	33739	154971
Pneumothorax	19448	56341	3145	144480
Pleural Effusion	86187	35396	11628	90203
Pleural Other	3523	316	2653	216922
Fracture	9040	2512	642	211220
Support Devices	116001	6137	1079	100197

Table 3.2: The CheXpert valid set labels

Label	Positive	Negative	Uncertain	Not mentioned
No Finding	38	196	0	0
Enlarged Cardiomediastinum	109	125	0	0
Cardiomegaly	68	166	0	0
Lung Opacity	126	108	0	0
Lung Lesion	1	233	0	0
Edema	45	189	0	0
Consolidation	33	201	0	0
Pneumonia	8	226	0	0
Atelectasis	80	154	0	0
Pneumothorax	8	226	0	0
Pleural Effusion	67	167	0	0
Pleural Other	1	233	0	0
Fracture	0	234	0	0
Support Devices	107	127	0	0

The different class labels represented in CheXpert can be seen in Table 3.1 and 3.2. The training data has been labeled using an automated rule-based labeler that extracts observations from the radiology reports connected to the images in a study[34]. Each label is either positive (has the value 1), negative (has the value 0) or uncertain (has the values -1) with respect to the presence of a particular observation. If an observation was not mentioned for an image in a report, the observation label is left blank. The validation set was manually annotated by certified radiologists and thus provide no uncertainty labels, as can be seen in Table 3.2.

The images in CheXpert can have have multiple positive labels simultaneously (for example, the patient can have a positive label for both "Cardiomegaly" and "Lung Opacity" at the same time). An exception to this is the "No Finding"-category which is used to indicate the absence of all other pathology labels (note that an image labeled with "No Finding" can also have a positive "Support Devices"-label as that is not a pathology).

Table 3.3: CheXpert subset where all uncertainty labels are ignored. The table shows the number of positively and negatively labelled images for each class category.

Label	Positive	Negative
No Finding	22271	111831
Enlarged Cardiomediastinum	7106	126996
Cardiomegaly	17851	116251
Lung Opacity	53190	80912
Lung Lesion	5621	128481
Edema	35756	98346
Consolidation	8716	125386
Pneumonia	3914	130188
Atelectasis	23749	110353
Pneumothorax	15069	119033
Pleural Effusion	55837	78265
Pleural Other	1982	132120
Fracture	6037	128065
Support Devices	74012	60090

3.2.1 Handling uncertainty labels

To be able to use the data set, some preprocessing had to be done. All missing labels, i.e. labels that were not mentioned during the label extraction, are assumed to be negative (i.e. set to 0). In some cases, such as when the image has some negative labels and the rest are not mentioned, this assumption resulted in an image associated only negative labels. This is not logical due to the "No Finding"-label, which should be positive to reflect that the rest of the pathology labels are negative. While it is possible to make an assumption and set the "No Finding"-label as positive in the case where all others are negative, there is no way to assure that this labelling is correct given the fact that this information is not present in the original data. Hence, all images with all negative labelling were excluded from the data set.

Classification models typically handle only positive/negative values, meaning some strategy must be used to decide how to handle the uncertainty labels in CheXpert. Irvin and Rajpurkar et. al. presents a number of different approaches on how to handle the uncertainty labels, each with varying results[34]. The approach used in this project was to ignore all images with uncertainty labels. This strategy is simple, but it also largely reduces the number of the available images, effectively shrinking it down to a smaller subset. The final subset of CheXpert used contains 134,102 images with only positive and negative labels. The distribution of positive and negative labels for each class category in this subset can be seen in Table 3.3.

3.2.2 Splitting the data into training, validation and test sets

While CheXpert comes with a manually annotated validation set, this set is quite small (only 234 images). Usually it is good to have a moderately sized data set in order to have good testing variance. A larger validation set can give more accurate result of how the model behaves.

The resulting subset from excluding the uncertainty labels was therefore split into three smaller sets; a training set, a validation set and a test set. Splitting the data set naively may result in images taken at the same occasion of the same patient existing in two or all sets at the same time. To ensure that there was no patient overlap between the sets, the data set was split based on the patients. Each patient was given a unique ID and then the list of IDs was randomly split 80%-20% twice, letting the first 20% be the test set and the second 20% be the validation set, leaving the rest to form the training set. All images belonging to a certain

Table 3.4: The distribution of the training-, validation- and test set achieved from splitting the CheXpert subset. Note that the percentage of images in each set have been rounded to the closest 2 decimals.

Set	Number of images in the set	Image percentage(%)
Training	80022	$\approx 59.67\%$
Validation	26805	$\approx 19.99\%$
Test	27275	$\approx 20.34\%$

patient ID was moved to the same subset. The final distribution of images between the set can be seen in Table 3.4.

3.3 Training the CNN classifiers

This section describes the implementation and training of the different classifiers. As previously mentioned, three different approaches to classifying chest x-rays were investigated. These approaches differ mostly in how the base problem (i.e. classifying chest x-rays) is tackled: the first approach was to treat it as a single binary classification problem, the second approach was to treat it as a multi-label classification problem, and the third approach was treat it as multiple binary classification problems. The third approach ended up being the one used to achieve results in the end. Details and discussions of these different approaches will be explained in the following sections. Approach 1 and 2 had some problems which made it difficult to achieve a useful result, and will thus only be described briefly. Because the third approach was the one that in the end was used to produce results, more details and specifics about its implementation will be given in Section 3.3.3.

3.3.1 Approach 1: Healthy VS Sick binary classifier

The first approach was to train a single binary classifier to differentiate between healthy and sick lung x-rays. This approach was a simplified take on the data provided in the data set: all images with the "No Finding"-label were assumed to be healthy since the "No Finding"-label indicates the lack of any disease, while all the pathology labels were combined and treated as single class: "Sick". This resulted in a data set containing only two classes ("Healthy" and "Sick") instead of 14. This simplification was motivated by the fact that simply knowing if signs of sickness exists or not would be enough to make a triage. This approach would also simplify sorting images as the output of the network would be a single comparable value for each image.

A CNN classifier was trained using this approach and transfer learning. As explained in Section 2.4.6, a new model can be trained by using pre-trained models and weights as the basis and only fine tune the later layers that perform the classification. Two different types of pre-trained model architectures were tested as base model with this approach: **DenseNet121**[35] and **VGG16**[36].

DenseNet is a very deep model architecture proposed by Huang et al.[35] that further builds on the idea that accuracy increases with model depth (i.e. the model is able to find more advanced features with more layers). What separates the DenseNet architecture from other deep models is that it contains dense connections between layers, meaning that each layer connects to all other layers in the network in a feedforward way, see Figure 3.1. These dense connections have a number of advantages, such as reducing the number of parameters (compared to regular deeper networks) and encouraging reusing of features[35]. The DenseNet model architecture follows the naming convention that the number after the name indicates the number of layers in the network, meaning that DenseNet121 contains 121 layers. The DenseNet121 architecture was tested in this project because it has been used successfully

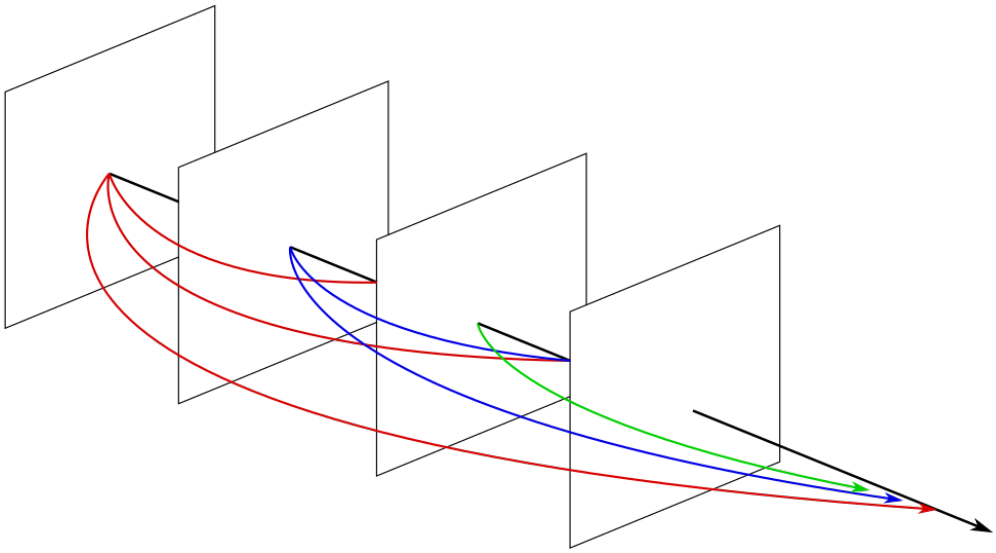


Figure 3.1: The dense connections in DenseNet. Each layer has connection to all subsequent layers.

in similar medical image analysis tasks. For example, Rajpurkar et al. used DenseNet121 to build their CheXnet model for detecting pneumonia.

Training such a deep network was difficult due to its size and therefore the smaller VGG architecture was also tested. The VGG model architecture created by Simonyan and Zisserman[36] is a popular CNN architecture famous for winning the ILSVRC competition in 2014, outperforming other state-of-the-art models at the time. The VGG models follow the same naming convention as DenseNet, i.e. the number following the name indicates the number of layers in the network (commonly 16 or 19). The VGG16 model was tested since has significantly fewer parameters compared to DenseNet121, meaning that training VGG16 takes less time. Pre-trained implementations of both the DenseNet121 and VGG16 architectures exist in Keras which further made them a suitable choice for this project.

Both pre-trained models were trained on the ImageNet data set which contains 1000 different classes[6]. To perform transfer learning, the weights were initialized using the pre-trained weights from ImageNet. In order to train a new binary classifier using the pre-trained models as the basis, the output layer was removed and replaced with a new output layer containing only one neuron and the sigmoid function as its output activation. The model was compiled using the Adam optimizer[21] and binary crossentropy as its cost function(See Section 2.4.4 and 3.3.3). The Adam optimizer was chosen as it is a recommended default choice, see Section 2.4.5. To perform transfer learning, all layers but the last one were frozen.

Several different configurations of batch size, learning rate, etc., were experimented with to see if the approach could achieve some results. However, the resulting classifier could not give a satisfactory to result as it tended to predict the class "Sick" for all images indiscriminately. This was theorized to be caused by the imbalance of the number of samples between the two classes or due to transfer learning from the ImageNet weights: some of the frozen layers could contain filters that are not suitable for the lung x-ray images.

There was also an interest from Sectra to be able to see which pathology had been detected by the CNN. For these reasons, implementation of the CNN shifted to a multi-label perspective.

3.3.2 Approach 2: multi-label classifier for 14 different observations

The second approach investigated was to create a classifier able to detect all the 14 different classes of the CheXpert data set. Since the classes in CheXpert are not mutually exclusive, i.e. one x-ray image can be positive for more than one class label, it made it a multi-label classification problem.

The implementation of this approach and third approach took inspiration from the implementation of CheXnet by Rajpurkar et al.[9]. Because the third approach was the one ultimately used in this projct, specific details about the implementation (such as preprocessing of the data, configuration of hyperparameters, etc.) will be explained in the next section covering the third approach.

Like in the first approach, both the VGG16 and DenseNet121 architectures were tested. Instead of using transfer learning, all weights in the network were trained from scratch. The weights were still initialized with weights from ImageNet as this can have some benefits over initializing them randomly (see Section 2.4 and 2.4.6). The model was compiled using Adam as optimization function. Binary crossentropy was chosen as the cost function, as it is suitable for multi-label classification, see Section 2.4.1. In contrast to the binary approach, the output layer was replaced with a layer of 14 outputs with sigmoid as its activation function in order to capture all class labels in CheXpert.

A few experiments with different configurations were run and a few issues arose. The model was tested using the metrics described in Section 2.5, which can be extended for multi-label classification. However, evaluating the model performance became difficult due to the model output being a set of predicted class labels for each image, meaning that what constitutes true positives, false positives, etc., had to be defined slightly differently. By testing the performance on a number of individual images with known labels it was observed that what according to the evaluation metrics appeared to be a good model did not perform well on the individual images, often predicting the same class for several images (even for images in the training set). The difficulty of measuring the performance was further exacerbated due to the data being *sparse*, i.e. each set of labels belonging to the images contains vastly more negative labels than positive labels. This sparseness can influence multi-label metrics and give a skewed view of the actual model performance.

Another issue that came about using this approach was that all values predicted were extremely small which caused difficulties when thresholding the output. As explained in Section 2.4.4, using sigmoid as output activation results in values between 0 and 1, with values close to 0 indicating the absence and 1 the presence of a particular class label. A typical classification threshold is therefore 0.5, classifying images with value above as 1s and below as 0s. In this case, all labels got output values very close to 0 (often somewhere in the range [0.01 to 0.000001]), causing all labels to be classified as 0. While this could be expected behaviour for some other type of classifier, in this case it does not work since if all pathology are negative, the "No Finding"-label should be positive.

Upon further inspection it was observed that even if the predicted values were small there were still some notable peaks where some labels had larger predictions compared to others for several test images. For some images, these peaks actually corresponded to the positive labels for that particular example, indicating that the model may still be able to somewhat differentiate between the classes. But because the peaks have small values they still fall below the threshold value (initially set to 0.5). One solution to this could have been to lower the threshold value to classify these labels as positive. However, the actual range the values took differed greatly between images and class labels so that accurately choosing a universally suitable threshold for all class labels became impossible. To solve this issue, the threshold would have needed variate between the classes.

Because of these reasons, implementation shifted to creating binary classifiers for each class individually in an attempt of simplifying the problem.

3.3.3 Approach 3: multiple binary classifiers using the binary relevance method

The final approach investigated and eventually used to create a triage was to train multiple classifiers using the binary relevance method. As described in Section 2.4.1, the binary relevance method can be used to simplify a multi-label problem by dividing it into several binary classification problems. This meant that a total of 14 different binary classifiers were trained, one for each class. By switching approach to the binary relevance method, evaluation of the models became simpler compared to a multi-label model.

Most of the implementation of this approach is identical to the multi-label approach: the main difference is that instead of training just one model to classify all classes, this approach instead trains several different models. As mentioned in the previous section, the implementation specifics of this and the multi-label approach was inpired by Rajpurkar et al.'s implementation of CheXnet[9]. This was because the CheXnet model is used to perform a similar task on similar data to that used in this thesis and does so with good results.

Classification using ResNet50

One of the biggest differences in this approach compared to the others was the choice of base model. As previously explained, experiments with both VGG16 and DenseNet121 were performed, but none of them were able to give a satisfactory result. Since VGG16 is a comparatively small network, it may not be able to find the features needed to classify pathologies in chest x-rays. This could be a factor in its poor performance in the first two approaches. On the other hand, training a full DenseNet121 took longer time to train but gave comparative results to that of VGG16. This could be a consequence of using such a deep network architecture: as He et al explains, a model with too many layers can start degrading: accuracy gets saturated when the depth of the model increases. For this reason, a new base model was tested as a middle ground between the two: **ResNet50**.

The ResNet architecture proposed by He et al.[37] is a deep neural network architecture with 50 layers that uses residual learning. He et al. hypothesize that instead of learning the mapping function $H(x)$ directly, it would be easier for the model to learn the residual $F(x) = H(x) - x$ (this mapping is usually recasted as $H(x) = F(x) + x$). ResNet uses residual blocks with connections that connect the input of one layer to the output of another layer, also known as "shortcut connections" (see Figure 3.2). The shortcut connections adds the identity function x (i.e. the function that returns its input as output) to the output of the stacked layers $F(x)$. The advantage of these residual blocks are that layers can be skipped, which alleviates the degradation issue that arises when the model has many layers. The ResNet architecture performed better than other state-of-the-art models and has won several classification competions, such as the ILSVRC 2015 classification competition[37].

The models were trained with a mini-batches with a batch size of 16. This size was chosen as it is a power of 2 (which can let the hardware achieve better runtime, see Section 2.4.5) and larger batch sizes caused hardware failure. The maximum number of epochs was set to 40, but training could finish earlier thanks to early stopping, see Section 3.3.3.

The number of steps per epoch, i.e. the number of gradient updates per epoch, was calculated in such a way that all images in the set were passed to the model in one epoch. Since the model uses a mini-batch of 16 images to compute one update step, the number of steps could be calculated as the total number of images in the set divided by the batch size. This resulted in $80022/16 = 5001$ steps for the training set and $26805/16 = 1675$ steps for the validation set.

Preprocessing of images

Before the images could be used for training, they were preprocessed. Preprocessing the images can help improve the model accuracy and shorten training time.

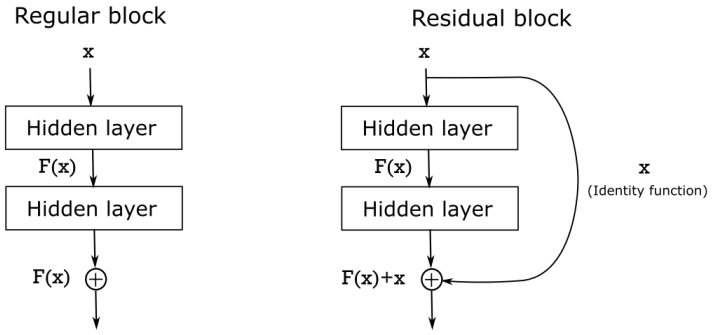


Figure 3.2: A common block in a regular stacked network (left) and a residual block in the ResNet architecture (right). The special shortcut connection in the residual block connects the input of one layer to the output of another layer.

Firstly, all images were rescaled to a size of 224X224 pixels as this is the default input size for the pre-trained models and training on larger images take longer time. While Keras can use generators to rescale the images upon loading them into the model, this proved to be a computational bottle-neck on the CPU, leading to the GPU not being used to its full capacity. For this reason, all images were rescaled before starting the training process, effectively removing the bottle-neck.

Since the network was initialized with weights from ImageNet and following the implementation of CheXnet, the images were normalized based on the mean and standard deviation of the images in the ImageNet dataset. Also following the implementation of CheXnet, augmentation in the form of random horizontal flipping was applied to the images. The augmentation was used to artificially increase the size of the data set, providing the model with more training examples to learn from.

Cost function

Because the binary relevance method results in a number of binary classification models, the cost function used for each model was binary crossentropy. As explained in Section 2.4.4, binary cross entropy is a special case of cross entropy (defined in Eq. 2.7) where the number of classes equals to 2. This means that binary cross entropy can be defined using Eq. 3.1, where y and $\hat{y}$ are the ground truth and predicted labels respectively and $N = 2$ is the number of classes. Binary cross entropy is implemented in Keras as the function `binary_crossentropy`[3].

$$J(y, \hat{y}) = -\sum_{i=1}^{N=2} y_i log(\hat{y}_i) = -y_1 log(\hat{y}_1) - (1 - y_1) log(1 - \hat{y}_1) \tag{3.1}$$

Optimizer

The optimizer chosen for this approach was the Adam optimizer[21], for which there exists an implementation in Keras[4]. As mentioned in Section 2.4.5, Adam is usually a good default choice for training a CNN. Adam was also used by Rajpurkar et al. in their implementation of CheXnet, which further motivated choosing it as the optimizer. As Rajpurkar et al. achieved

[3]Source code: `https://github.com/keras-team/keras/blob/master/keras/losses.py`
[4]Source code: `https://github.com/keras-team/keras/blob/master/keras/optimizers.py`

good results by using the default parameters of Adam (i.e. (initial) $lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$), the default parameters were also used in this project.

Decaying learning rate scheme and early stopping

Models can often benefit from lowering the learning rate by a factor once learning has stagnated, as this ensures that the minimum is not missed[2]. Usually, the learning rate is lowered if the validation cost has not improved over n epochs (the parameter n is referred to as a patience parameter). Rajpurkar et al. lets their initial training rate of 0.001 decay by a factor of 10 each time the validation loss plateaus after an epoch. In this project, the learning rate was set to decay by a factor of 10 with a patience of 5, meaning that if the validation cost did not decrease after 5 epochs, the learning rate decayed. A minimum for decaying the learning rate was set to 1^{-8}. In Keras, this type of decay can be used during training with the help of the callback function `ReduceLROnPlateau`.

Early stopping was also implemented to store the best weight configurations in case the models start to overfit during training. Early stopping, as explained in Section 2.4, stores the weights each time the model improves (here defined as every time the validation loss decreases). These "best" weights can then be retained and used for the final model even if the model starts to overfit. In Keras, early stopping can be used during training with the help of the callback function `EarlyStopping`.The patience parameter for early stopping was set to 10, meaning that if validation loss did not decrease over 10 epochs, the model was assumed to either have converged or overfit. In this case, training of the current model was cut short and training of the next model could start early. Consequently, the different models were trained different amounts of time.

Summary of hyperparameters

The values of different hyperparameters used during training are summarized in Table 3.5. While different values were experimented with, the values in Table 3.5 were the ones used to produce the results in Chapter 4.

Table 3.5: Hyperparameters values used in training

Hyperparameter	Value
Base model	ResNet50
Cost function	Binary crossentropy
Output activation	sigmoid
Optimizer	Adam
Learning rate	0.001(initial)
Epochs	40(maximum)
Batch size	16
Steps per epoch (training)	5001
Steps per epoch (validation)	1675
Learning rate decay factor	10
Learning rate decay patience	5
Early stopping patience	10

3.3.4 Experiment using a single "Sick" class

After training the 14 individual models, an additional model was trained that essentially combines approach 1 and 3. In this experiment the model uses the same method and implementation as approach 3, but the different pathologies were again combined into a singles "Sick" class like in approach 1. This experiment was done to test if the first approach could benefit from using the same type of preprocessing, base model, etc., used in approach 3. It

was also done since it would give an interesting point of comparison of the two different approaches. This experimental model was trained using the same hyperparameters listed in Table 3.5. The model was also evaluated the same way as the 14 individual models, see Section 3.4 and the results can be found in Section 4.

3.4 Evaluating the models

After the models had been trained they were evaluated with the help of the test set achieved from splitting the CheXpert subset in Section 3.2. The evaluation metrics used were:

- Precision
- Recall
- F1 score
- ROC curves with corresponding AUC score
- Precision-Recall curves with corresponding AUC score (average precision)

These metrics were chosen because they are all suitable for binary classification problems. The definitions of the metrics above can be found in Section 2.5. The reason that several different metrics were chosen was that they show the model performance from different points of view, e.g. precision and recall. Like explained in Section 2.5, some metrics can give an overly positive view of the model performance, such as the ROC curve which may be very good when the model is bad as a result of imbalanced classes (as is the case with this project). PR curves, which takes class imbalance into account, can therefore be used to confirm the results of the ROC curve.

The metrics were computed using the *scikit-learn* library, which is a machine learning library written in Python[38]. *scikit-learn* contains function implementations for each of the evaluation metrics above, which made it a good choice for this project. The different metrics were computed for one model at a time, storing the numeric results in a file corresponding to the model. Graphs of the ROC and PR curves were also plotted and stored. The results from this can be seen in Chapter 4.

3.4.1 Evaluating the models on a previously unseen data set

While the test set was kept separate during training as to not influence the values of the weights, the test set comes from the same distribution as the training- and validation-set, which could mean that model has an easier time classifying the images in the test set as the images are similar. To further test the generalization abilities of the different models, a completely new data set was used. The data set used in this additional testing phase was the ChestX-ray14 data set[39] which contains 108,948 frontal-view chest x-ray images of 32,717 unique patients labeled with 14 different lung diseases.

In order to test the models, the labels of ChestX-ray14 had to transformed to match the class labels found in CheXpert. This was done by retrieving images that had overlapping labels between the two datasets, and disregarding the others. The labels between CheXpert and ChestX-ray14 that overlapped were No Finding, Cardiomegaly, Edema, Consolidation, Atelectasis, and Pneumothorax. The value of the overlapping labels were set to 1 while the remaining CheXpert labels were set to 0. This resulted in a subset of 83830 images from ChestX-ray14, see Table 3.6. From this subset, 20000 images were randomly chosen to form a new test set that could be used to evaulate the different models. In order to test the experimental binary model introduced in Section 3.3.4, a second version of this test set where all class labels except "No Finding" were combined into a singular "Sick" class was also created.

Table 3.6: The resulting test set from ChestX-ray14. Because there are labels that did not overlap between ChestX-ray14 and Chexpert, several of the classes have 0 positive examples. The Sick class summarizes all class labels (except No Finding) into one class.

Label	Positive	Negative
No Finding	14420	5580
Enlarged Cardiomediastinum	0	20000
Cardiomegaly	629	19371
Lung Opacity	0	20000
Lung Lesion	0	20000
Edema	575	19425
Consolidation	1119	18881
Pneumonia	0	20000
Atelectasis	2704	17296
Pneumothorax	1267	18733
Pleural Effusion	0	20000
Pleural Other	0	20000
Fracture	0	20000
Support Devices	0	20000
Sick	5580	14420

Because there were labels that did not overlap between the two data sets, several labels in this new test set had 0 positive examples. This meant that there were no true positives, which in turn made the evaluation metrics ill-defined for those particular class labels. For this reason, only the overlapping labels could be evaulated and used for comparison.

3.5 Visualising the model using Grad-CAM

Grad-CAM (Gradient-weighted Class Activation Maps) by Selvaraju et al.[40] is a method for visualizing how a CNN makes its decisions. Using Grad-CAM, it is possible to produce a coarse localization map that indicates which regions in the image were important when the model predicts a particular class. For example, if the model predict the class "Cat" for a photo of a cat and a dog, the region of the image containing the cat should be more important to that decision than the region containing the dog. Essentially, the localization map gives an indication of what the model "sees" when it looks at an image.

Grad-CAM uses class-specific gradient information flowing through the last convolutional layer to determine the importance of each neuron when predicting the specific class. The feature map from the final convolutional layer has every channel weighted with the gradient of the class with respect to the channel. The results of this shows how much the input image activates the different channels, which in turn shows how important that channel is for the particular class. The final result is a localization map that can then be upscaled and overlaid on the original image to show the important regions in the form of a heat map.

This technique can be very useful for debugging CNNs, as it can be difficult to interpret what is actually happening under the hood of the model. Grad-CAMs were therefore used during debugging and analysis of the final results. To create the Grad-CAMs, the *Keras-vis*[41] toolkit was used. *Keras-vis* is a toolkit for visualizing neural networks created with Keras.

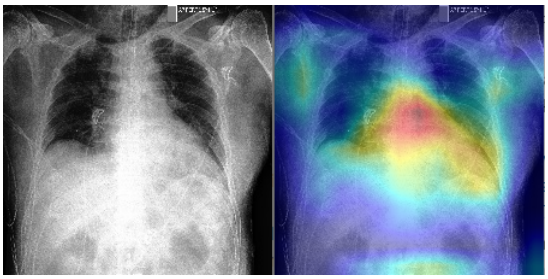


Figure 3.3: Example of a heatmap produced by Grad-CAM.

4 Results

This chapter presents the results from training and testing the CNNs described in Sections 3.3.3 and 3.3.4. Results from testing the model on the ChestX-ray14 data set is also presented.

4.1 Results from training

This section shows how the models learned during training.

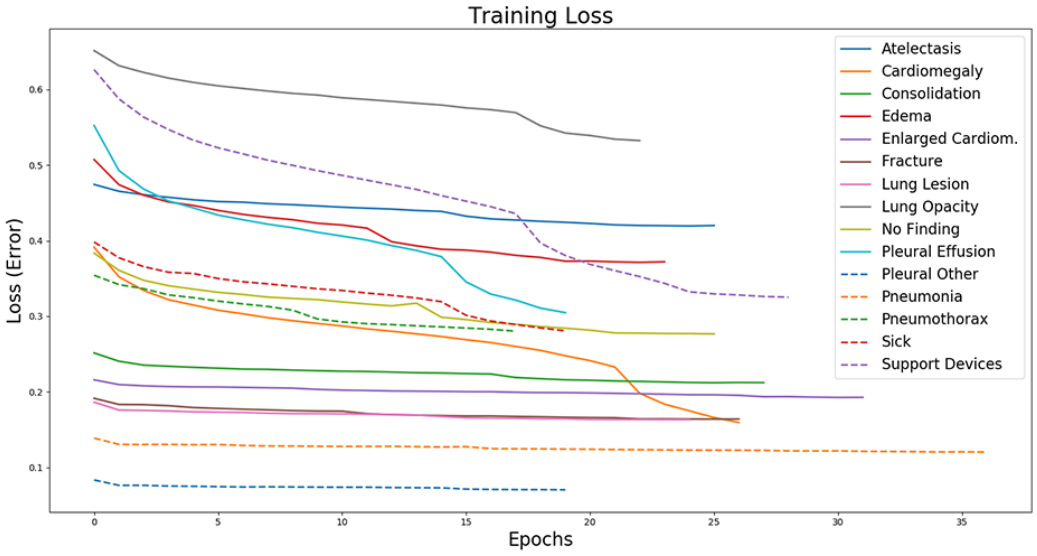


Figure 4.1: The changing training loss for the different models during training

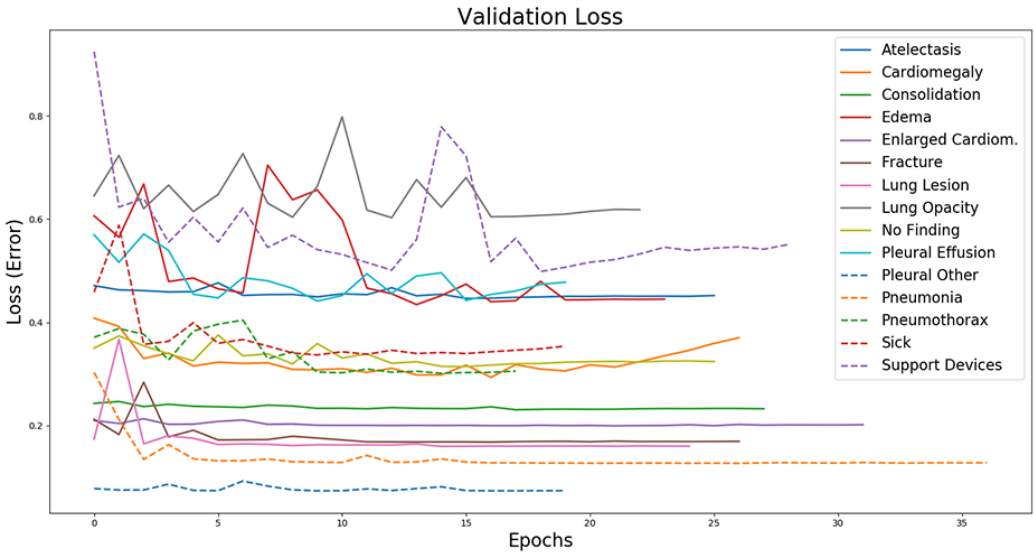


Figure 4.2: The changing validation loss for the different models during training

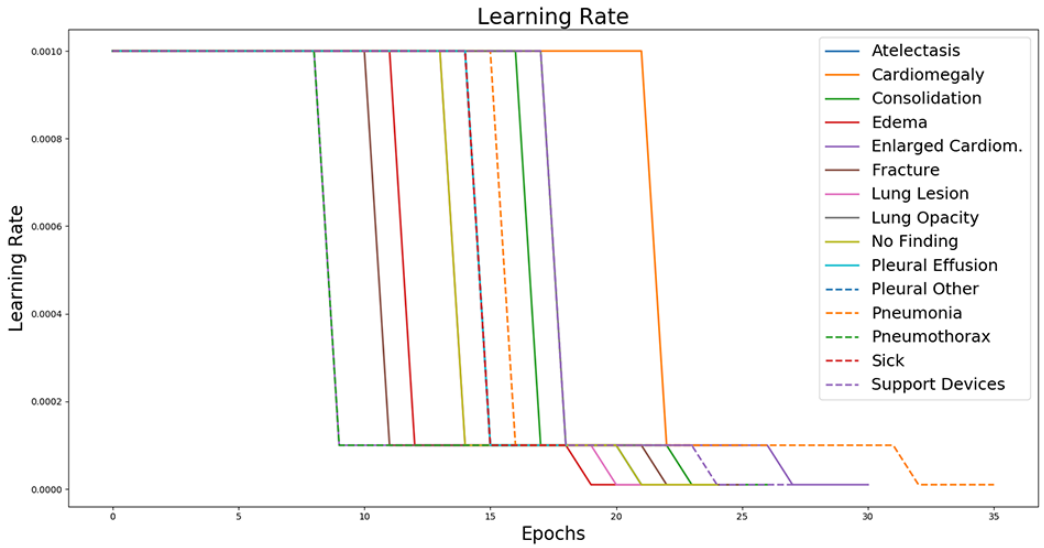


Figure 4.3: The changing learning rate for the different models during training

4.2 Numeric evaluation metrics

The tables in this section presents the numerical results from evaluating the different models using the metrics precision, recall, F1 score (see Table 4.2), ROC AUC score and Average Precision (AUC score for the precision-recall curve) (see Table 4.1)

Table 4.1: AUC scores (ROC and Precision-Recall) for each model.

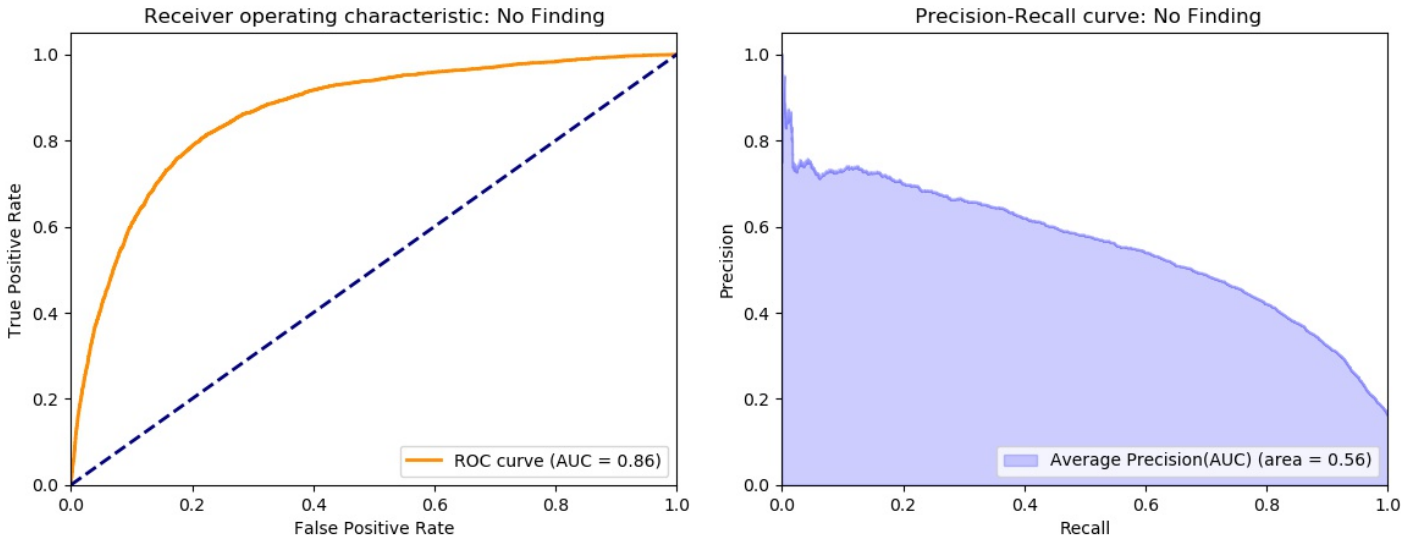
Class	ROC AUC score	Average Precision score (AUC)
No Finding	0.86	0.56
Enlarged Cardiomediastinum	0.6	0.09
Cardiomegaly	0.84	0.51
Lung Opacity	0.71	0.59
Lung Lesion	0.67	0.08
Edema	0.84	0.65
Consolidation	0.68	0.13
Pneumonia	0.64	0.05
Atelectasis	0.65	0.27
Pneumothorax	0.77	0.36
Pleural Effusion	0.87	0.82
Pleural Other	0.69	0.03
Fracture	0.68	0.11
Support Devices	0.84	0.85
Sick	0.86	0.96

Table 4.2: Results from using the evaluation metrics Precision, Recall and F1 score with five different classification thresholds.

Class	Metric	Threshold				
		0.4	0.5	0.6	0.7	0.8
No Finding	Precision	0.552235	0.600309	0.655	0.706622	0.712794
	Recall	0.559018	0.442347	0.327723	0.191722	0.0620878
	F1 score	0.555606	0.509362	0.436865	0.30161	0.114226
Enlarged Cardiomediastinum	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Cardiomegaly	Precision	0.610709	0.669280	0.7348066	0.783486	0.820895
	Recall	0.3500138	0.259607	0.183854	0.1180536	0.060824
	F1 score	0.444991	0.37410358	0.2941176	0.2051899	0.113256
Lung Opacity	Precision	0.542103	0.5948257	0.668947	0.687943	0.0
	Recall	0.727776	0.492690	0.186416	0.008808	0.0
	F1 score	0.621366	0.538962	0.2915779	0.01739286	0.0
Lung Lesion	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Edema	Precision	0.6068215	0.6583604	0.704451	0.755215	0.8166023
	Recall	0.624516	0.503869	0.3761747	0.245163	0.1169154
	F1 score	0.6155418	0.570847	0.490450	0.3701617	0.204545
Consolidation	Precision	0.2	0.0	0.0	0.0	0.0
	Recall	0.000569	0.0	0.0	0.0	0.0
	F1 score	0.001135	0.0	0.0	0.0	0.0
Pneumonia	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Atelectasis	Precision	0.398892	0.6	0.0	0.0	0.0
	Recall	0.0300187	0.001876	0.0	0.0	0.0
	F1 score	0.055835	0.0037406	0.0	0.0	0.0
Pneumothorax	Precision	0.498286	0.565495	0.6160267	0.6720867	0.677272
	Recall	0.2343649	0.1711799	0.1189555	0.079948	0.048033
	F1 score	0.3187897	0.262806	0.199405	0.142898	0.089705
Pleural Effusion	Precision	0.727736	0.7616847	0.794222	0.827190	0.861784
	Recall	0.817527	0.7579176	0.679826	0.568590	0.4165727
	F1 score	0.7700229	0.759796	0.732585	0.673934	0.5616518
Pleural Other	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Fracture	Precision	0.166666	1.0	0.0	0.0	0.0
	Recall	0.000808	0.0008078	0.0	0.0	0.0
	F1 score	0.001608	0.001614	0.0	0.0	0.0
Support Devices	Precision	0.767411	0.794826	0.821242	0.845107	0.869980
	Recall	0.840936	0.7985678	0.7505636	0.687707	0.580294
	F1 score	0.802493	0.79669257	0.7843137	0.7583257	0.696205
Sick	Precision	0.8462537	0.873181	0.900423	0.921451	0.943093
	Recall	0.969320	0.935532	0.891337	0.834482	0.763031
	F1 score	0.903616	0.903282	0.895857	0.875813	0.843560

4.3 ROC- and Precision-Recall curves

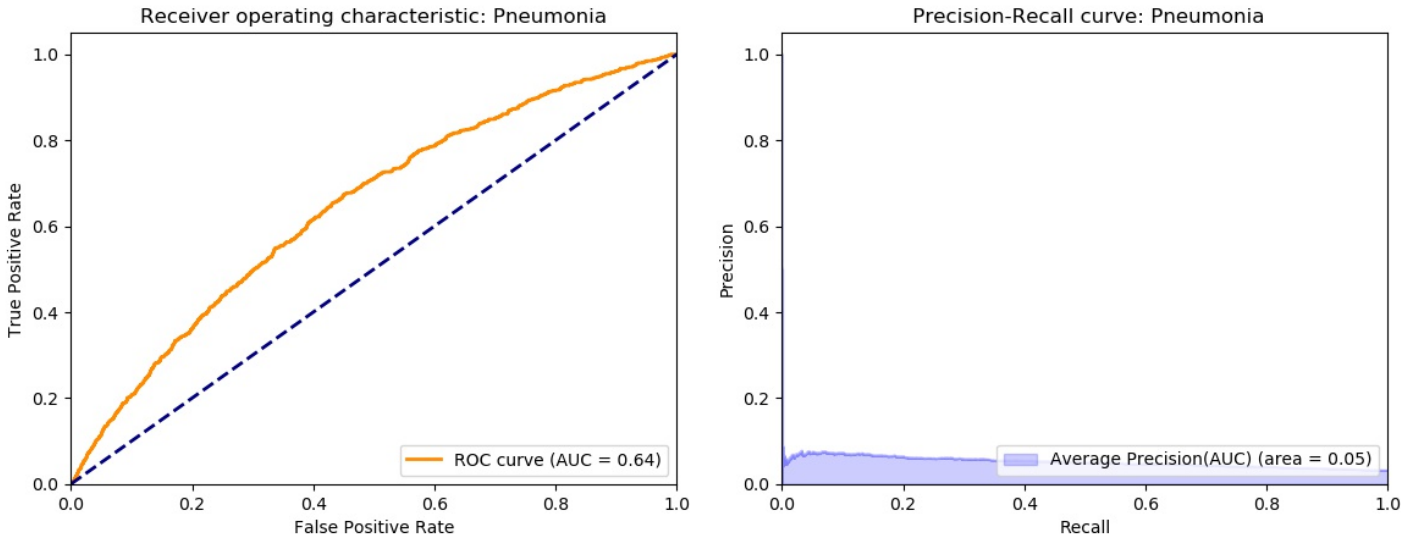
This section shows graphs of the ROC curve and Precision-Recall curves for some of the trained classification models: "No Finding", "Pneumonia", "Pleural Other", "Support Devices" and "Sick". The results from the remaining models can be found in Appendix A (Figures A.1-A.10).



(a) ROC curve

(b) PR curve

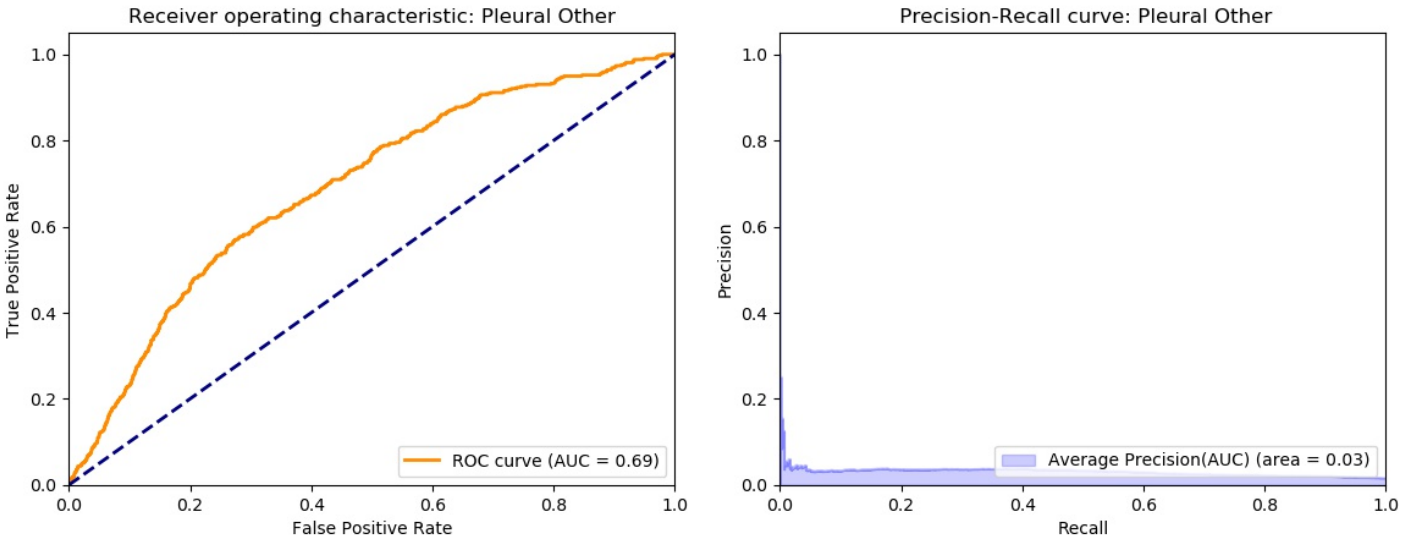
Figure 4.4: Graphs for the No Finding class.



(a) ROC curve

(b) PR curve

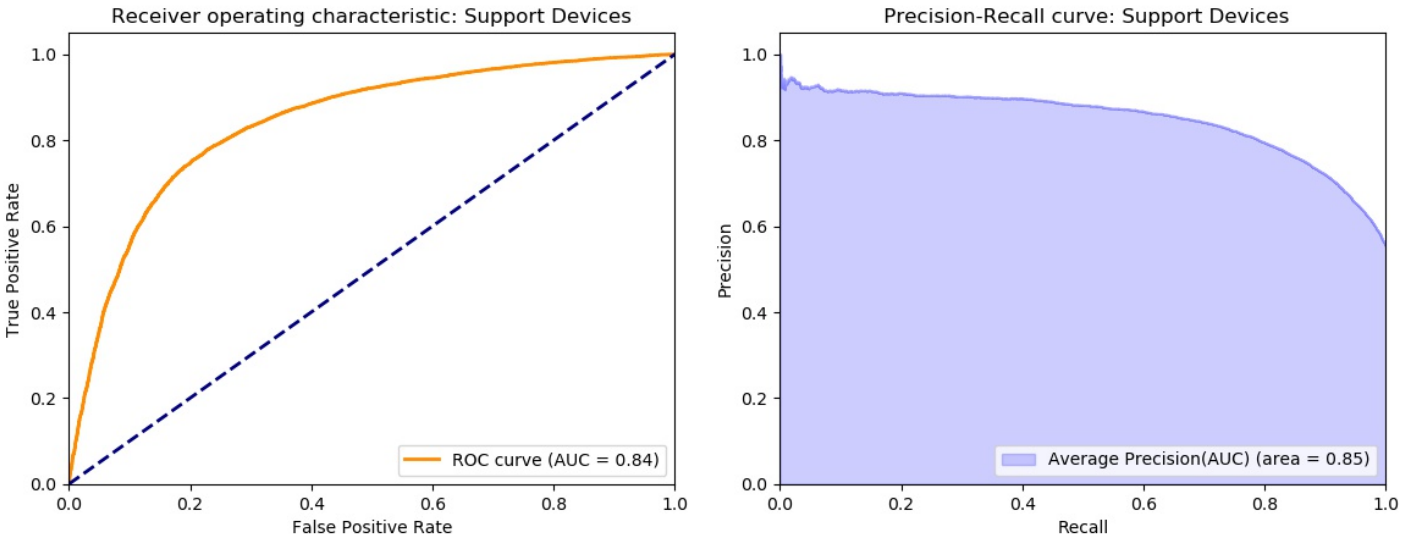
Figure 4.5: Graphs for the Pneumonia class.



(a) ROC curve

(b) PR curve

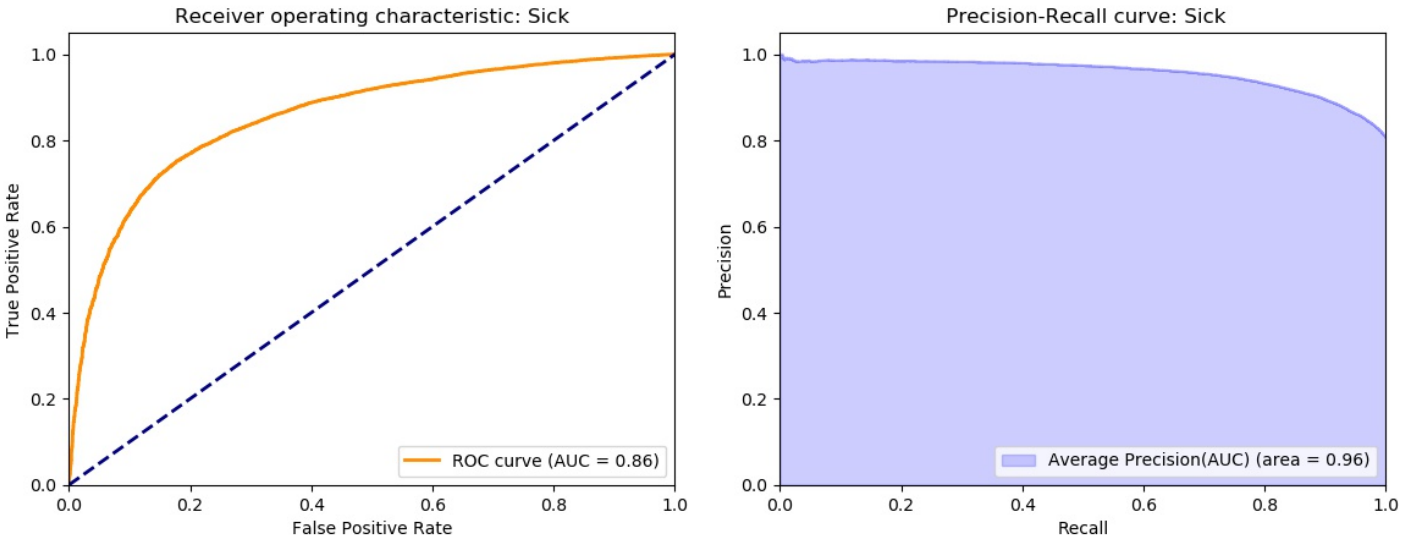
Figure 4.6: Graphs for the Pleural Other class.



(a) ROC curve

(b) PR curve

Figure 4.7: Graphs for the Support Devices class.



(a) ROC curve

(b) PR curve

Figure 4.8: Graphs for the Sick class.

4.4 Testing the models on the ChestX-ray14 data set

As explained in Section 3.4.1, some of the models were also tested on the ChestX-ray14 data set. This section presents the results from testing these models on the ChestX-ray14 data set. The models tested were for the classes "No Finding", "Cardiomegaly", "Edema", "Consolidation", "Atelectasis", "Pneumothorax" and "Sick".

4.4.1 Numerical evaluation metrics

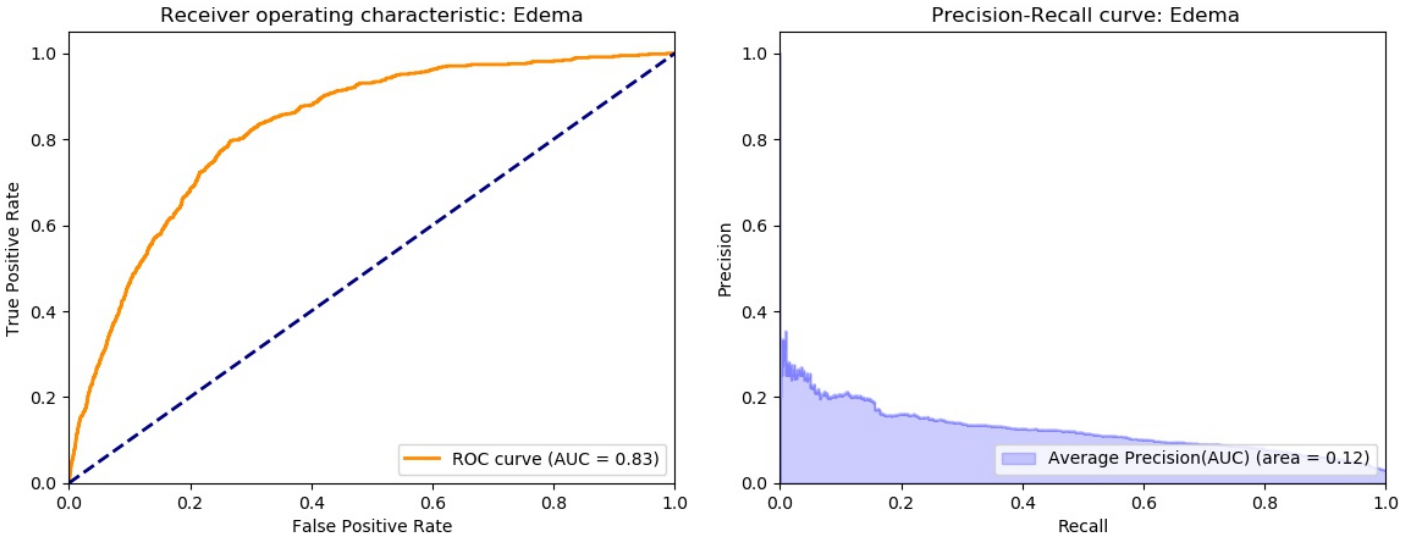
Table 4.3: AUC scores (ROC and Precision-Recall) for each model tested on the ChestX-ray14 data set.

Class	ROC AUC score	Average Precision-Recall score (AUC)
No Finding	0.74	0.88
Cardiomegaly	0.84	0.23
Edema	0.83	0.12
Consolidation	0.72	0.13
Atelectasis	0.65	0.22
Pneumothorax	0.67	0.15
Sick	0.74	0.48

Table 4.4: Results from using the evaluation metrics Precision, Recall and F1 score with five different classification thresholds, tested on the ChestX-ray14 data set.

Class	Metric	Threshold				
		0.4	0.5	0.6	0.7	0.8
No Finding	Precision	0.867168	0.883532	0.903111	0.928231	0.958835
	Recall	0.615257	0.513454	0.386546	0.229612	0.066227
	F1 score	0.719809	0.649474	0.541375	0.368155	0.123897
Cardiomegaly	Precision	0.306034	0.413127	0.496599	0.61039	0.742857
	Recall	0.225755	0.170111	0.116057	0.074722	0.041335
	F1 score	0.259835	0.240991	0.188144	0.133144	0.078313
Edema	Precision	0.089701	0.100939	0.116307	0.132248	0.160167
	Recall	0.704348	0.598261	0.497391	0.353043	0.2
	F1 score	0.159136	0.172734	0.18853	0.192417	0.177881
Consolidation	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Atelectasis	Precision	0.0	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0	0.0
	F1 score	0.0	0.0	0.0	0.0	0.0
Pneumothorax	Precision	0.357143	0.6	0.8	0.75	0.666667
	Recall	0.007893	0.004736	0.003157	0.002368	0.001579
	F1 score	0.015444	0.009397	0.006289	0.004721	0.00315
Sick	Precision	0.378132	0.403792	0.426170	0.447774	0.476748
	Recall	0.868280	0.797670	0.731362	0.661470	0.576882
	F1 score	0.526831	0.536168	0.538533	0.534037	0.522056

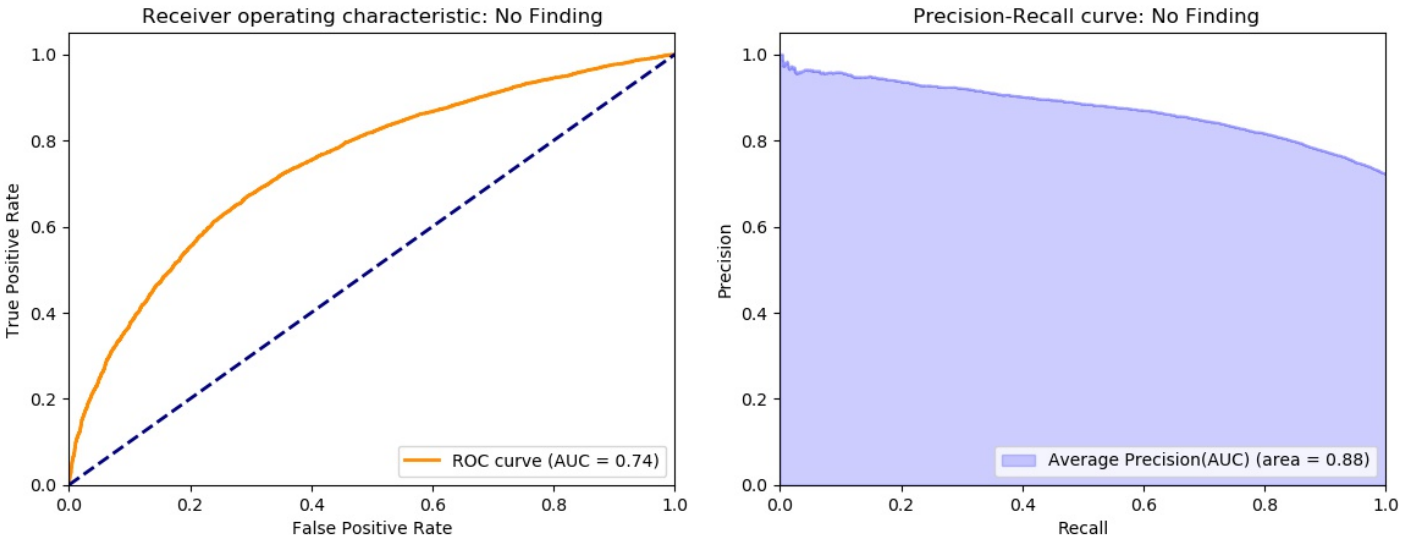
4.4.2 ROC- and Precision-Recall curves



(a) ROC curve

(b) PR curve

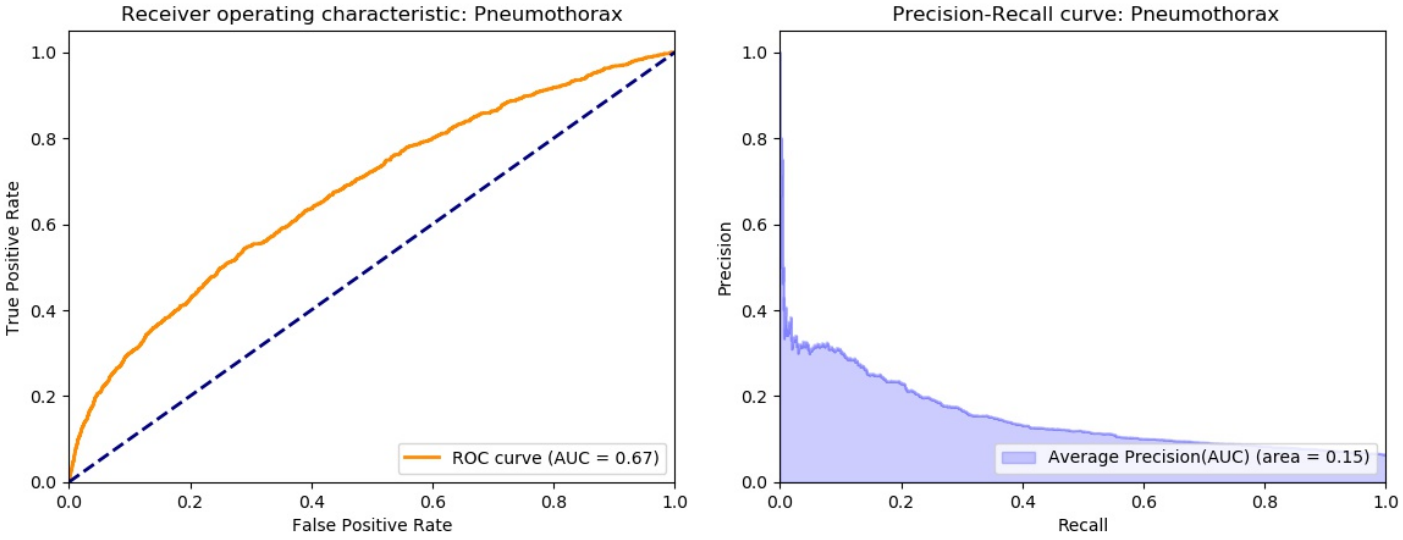
Figure 4.9: Graphs for the Edema class tested on the ChestX-ray14 data set.



(a) ROC curve

(b) PR curve

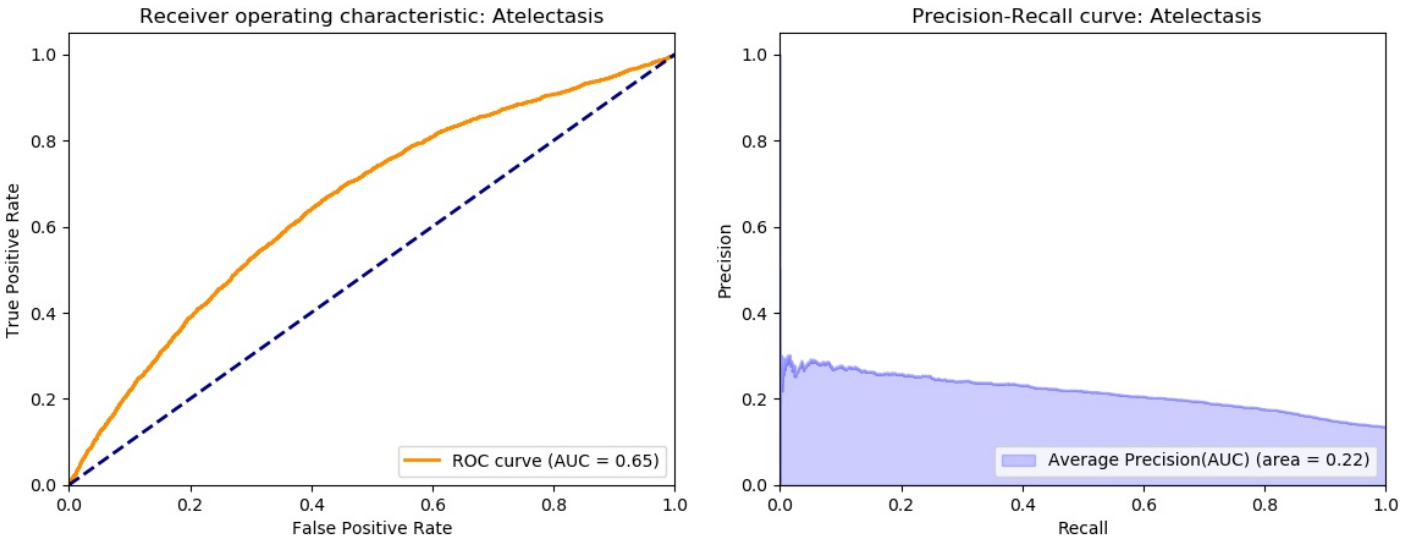
Figure 4.10: Graphs for the No Finding class tested on the ChestX-ray14 data set.



(a) ROC curve

(b) PR curve

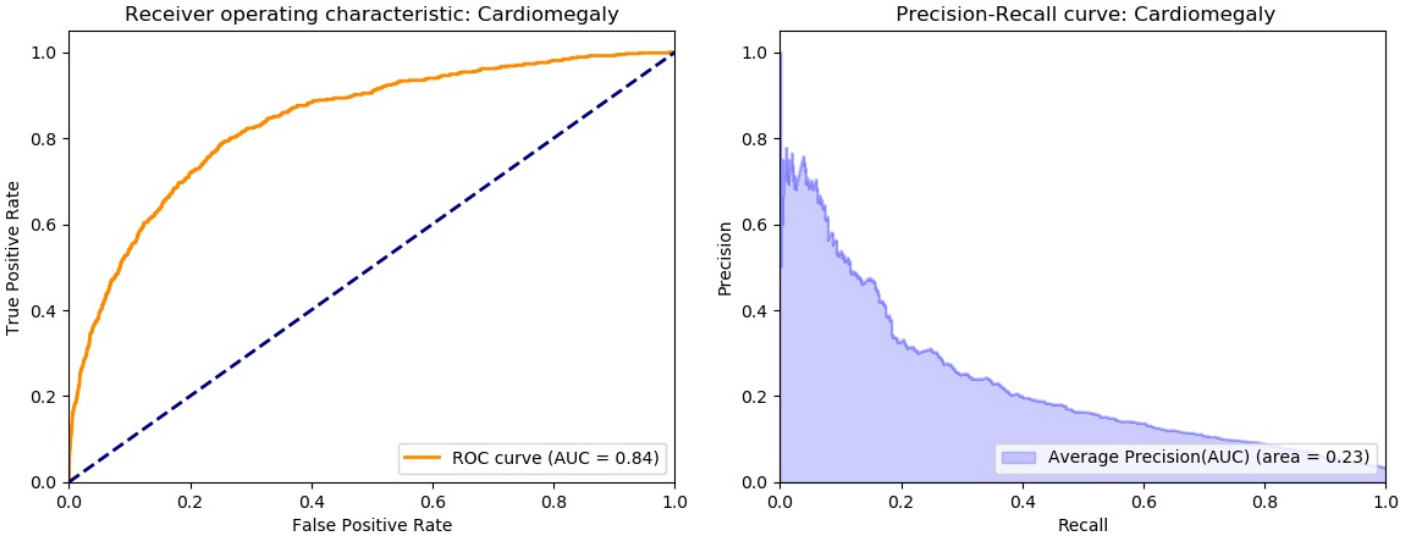
Figure 4.11: Graphs for the Pneumothorax class tested on the ChestX-ray14 data set.



(a) ROC curve

(b) PR curve

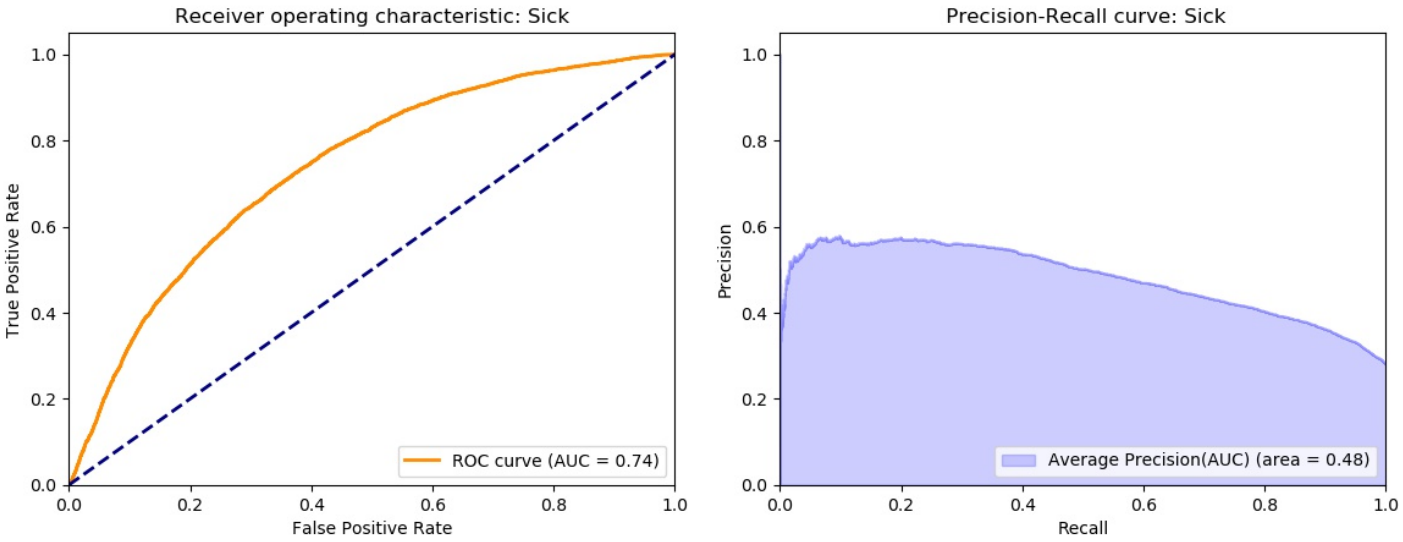
Figure 4.12: Graphs for the Atelectasis class tested on the ChestX-ray14 data set.



(a) ROC curve

(b) PR curve

Figure 4.13: Graphs for the Cardiomegaly class tested on the ChestX-ray14 data set.



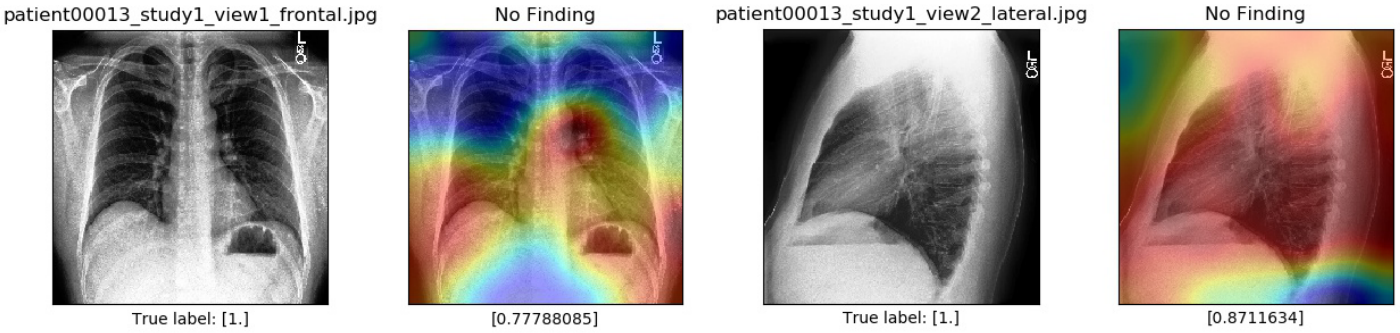
(a) ROC curve

(b) PR curve

Figure 4.14: Graphs for the Sick class tested on the ChestX-ray14 data set.

4.5 Visualization using Grad-CAM

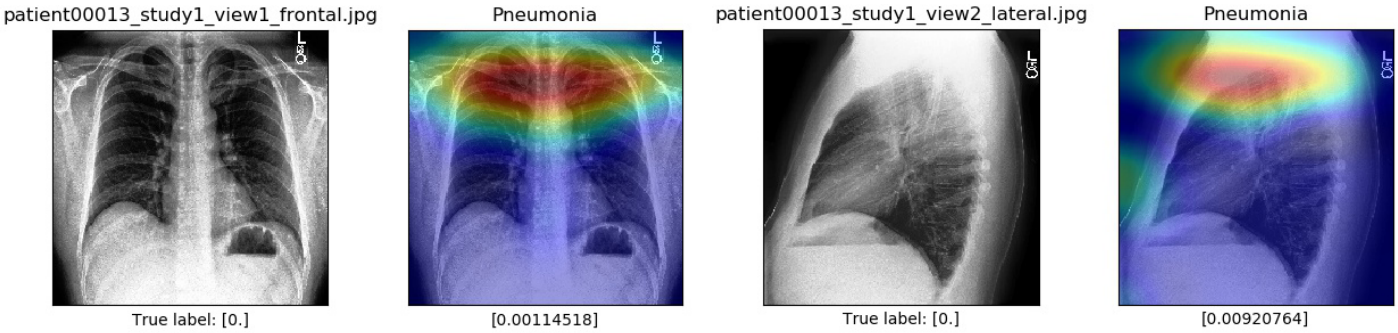
Grad-CAM (see Section 3.5) was used to produce heatmaps to visualize how the models make their decisions. The following images show Grad-CAMs generated from some example images in the test set, both lateral and frontal. The original image and the true label for that image is shown on the left, while the generated heatmap for the image can be seen on the right in each subfigure of Figure 4.15. The name of the model used to generate the heatmap can be seen above the heatmap, while the model's prediction for the image can be seen below it. Note that Grad-CAMs for the classes "Fracture", "Lung Opacity", "Lung Lesion" and "Pleural Other" are missing because the Grad-CAM generation failed for those models, which could be an issue with the *Keras-vis* toolkit. Additional examples can be found in Appendix A.



(a) Frontal Grad-CAM generated from the No Finding model.

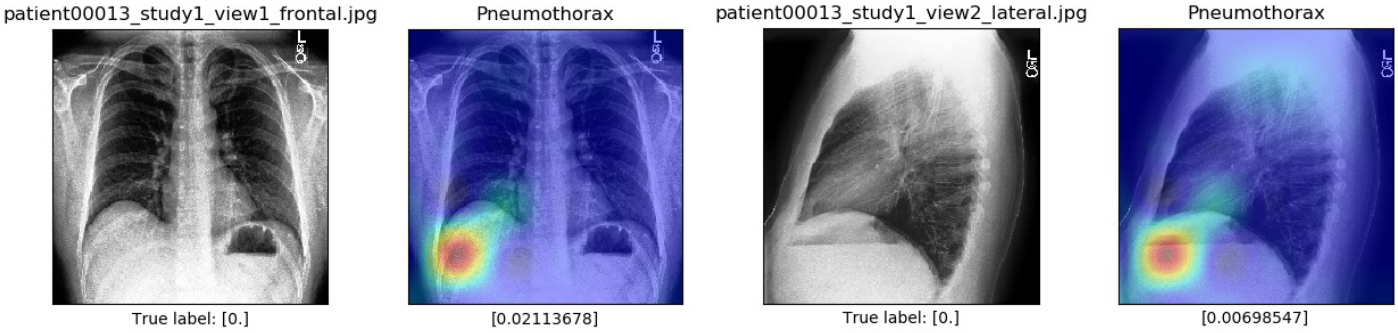
(b) Lateral Grad-CAM generated from the No Finding model.

Figure 4.15: Examples of generated Grad-CAMs.



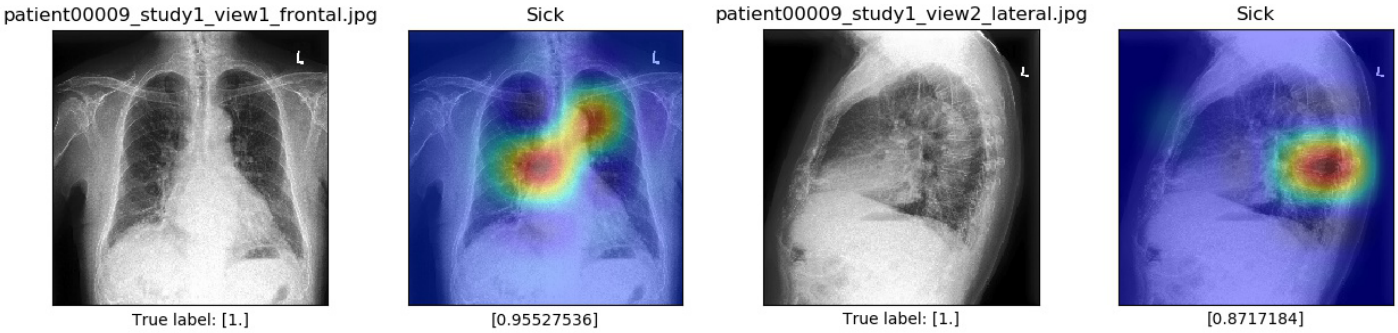
(c) Frontal Grad-CAM generated from the Pneumonia model.

(d) Lateral Grad-CAM generated from the Pneumonia model.



(e) Frontal Grad-CAM generated from the Pneumothorax model.

(f) Lateral Grad-CAM generated from the Pneumothorax model.



(g) Frontal Grad-CAM generated from the Sick model.

(h) Lateral Grad-CAM generated from the Sick model.

Figure 4.15: Examples of generated Grad-CAMs (continuation).

5 Discussion

This chapter contains a discussion around the results from the different models and the method used to train the models. The work in a wider context is also discussed.

5.1 Results

5.1.1 The training progress

The progress of training the different models can be seen in Figures 4.1, 4.2 and 4.3. From Figure 4.1 it is possible to see how the training error for the different models decreases over the number of epochs. Some of them decrease quickly while some decrease very slowly, for example "Pneumonia". One can also see how some of the graphs momentarily decrease even more as a result of decaying the learning rate by a factor of 10. This is especially noticeable for the "Support Devices" and "Cardiomegaly" models whose learning rates decay at the start of epoch 17 and 22 respectively. The learning rate decay for all models can be seen in Figure 4.3. The initial training error varies a lot between the models, most likely as a result of the difference in the number of positive and negative example images in each class: as the model starts by guessing randomly, it is not unreasonable for the initial error to reflect the class distribution in the training set.

Figure 4.2 shows how the models performed on the validation set during training. The validation error for some of the models fluctuated a lot more compared to the training error in some cases validation error. Some of the models overfitted during training, for example the "Support Devices"- and "Cardiomegaly" models whose validation error can be seen to increase at around epoch 20. This indicates that the models may have a poor generalization ability.

5.1.2 Results from evaluation

The models from training were tested on the test set and the result from the evaluation metrics used can be seen in Tables 4.1-4.2 and Figures 4.4-4.7. Most of the models did not perform well on the test set: in Table 4.2 it is possible to see that many of the models had really poor values for the metrics precision, recall and F1 score for five different example thresholds. Notably, the worst metric for most of models is recall (which in turn influences the F1 score)

which is usually a lot lower than precision for the same model. An example of this is the "Cardiomegaly" model, which achieves a best precision of 0.82 while the recall for the same threshold is as low as 0.06.

Table 4.2 also shows how changing the classification threshold affects the metrics: precision generally increases with a higher threshold while recall decreases. Note that some models have the value 0.0 for several metrics in Table 4.2, e.g. "Lung Lesion" or "Pneumonia". This is most likely due to all predictions made by those models falling below the example thresholds, resulting in $TP = 0$.

The poor results of the bad models can also be seen in their respective ROC and Precision-Recall curves (Figures 4.4-4.7). At first glance the ROC curves may look okay but the Precision-Recall curve for the same model shows a more negative result. An example of this can be seen in the models for "Pleural other" and "Pneumonia", see Figures 4.6 and 4.5 respectively, who has very low Precision-Recall curves for all thresholds, leading to an AUC score close to 0. As the bad results are most noticeable in the Precision-Recall curve, it indicates that it may be caused by class imbalance.

While several of the models had very poor results (mainly the models for "Atelectasis", "Consolidation", "Enlarged Cardiomediastinum", "Fracture", "Lung Lesion", "Pleural Other" and "Pneumonia"), some models achieved okay to good results from the evaluations. The results for the "Pneumothorax" and "Cardiomegaly" models are better in terms of ROC and Precision-Recall curves compared to the previously mentioned models, having a larger AUC scores by comparison (see Table 4.1) but they are still not very good models as their metrics are not near the optimum values. The "No Finding", "Lung Opacity" and "Edema" models fall somewhere on the line between good and bad, achieving a ROC AUC score over 0.7 but an average precision of at around 0.6.

The models who achieved good results from the metrics were "Support Devices" and "Pleural Effusion". Compared to the other models, these two got high values for all the metrics in Tables 4.1 and 4.2. Both models have good ROC and Precision-Recall curve with AUC scores above 0.8.

A possible correlation can be observed between the results and the class imbalance/number of positive and negative example images in the training set for a particular model. The models that achieved a good result from the evaluation has many more positive examples compared to the other classes, see Table 3.3, meaning that the class imbalance when training those models were not as severe. The models that performed okay to good have over 20,000 positive examples with the rest being negative, e.g. "No Finding" or "Lung opacity". Notably the best performing models, "Pleural Effusion" and "Support Devices", has the two largest number of positive examples of all their classes. In comparison, the worst models have below 10,000 positive examples, meaning that their imbalance is very high. This explains why the ROC curve for these models may look okay but the Precision-Recall curve is bad (as explained in Section 2.5, Precision-Recall curves take class imbalance into account). Note that there are some exceptions, for example the "Atelectasis" class which achieved bad test results despite having a large number of positive examples.

The bad results could be attributed to either the lack of positive data to learn from or simply to the difference in the number of positive and negative examples. It is possible that the models could have been trained better by simply reducing the class imbalance through removing some of the negative examples. On the other hand, there simply could have been not enough positive data for the model to learn how to detect the class in question.

When testing the five overlapping models on the ChestX-ray14 data set, they performed even more badly, showing that these models have poor generalization abilities. The only good result when testing on this data set is for the "No Finding" model which actually achieves a higher average precision than when tested on the test set. However, at the same time the ROC AUC score for the ChestX-ray14 data set is slightly less compared to the test set (0.86 VS 0.74), meaning the model had a bit more difficulty in separating between the two classes for this data set. This could be due to the images being slightly different between the

two data sets, in terms of contrast for example. One way this perhaps could be improved is by applying slightly different preprocessing to make the images as similar to the images in CheXpert as possible.

5.1.3 The experimental "Sick" model

Perhaps the most interesting results comes from the experimental model introduced in Section 3.3.4 that combines all classes except "No Finding" in one "Sick" class. The results from this model exceeds all of the other models obtained from the binary relevance approach, as can be seen in Table 4.1 and 4.2 as well as Figure 4.8. This model achieves the highest average precision score of 0.96. In Table 4.2 it can be seen that this models has consistently high values for the metrics over all the experimental thresholds.

While the "Sick" model performed well on the test set, its results for the ChestX-ray14 data is not as good. The average precision dropped by 50%, see Table 4.3, and in Table 4.4 it can be seen that precision metric is much lower in comparison to the test set. This means that many of the positive predictions made by the model turned out wrong for this data set. A factor in this could be that classes such as "Support Devices" are included in the "Sick" class: as mentioned in Section 3.2, a healthy patient can still have support devices, as that in and of itself is not a disease. This may make it very difficult for the model to learn features that indicates sickness, as these kind of features can exists in both healthy and sick images. This makes it interesting to repeat this experiment but with a redefined "Sick" class with classes such as "Support Devices" excluded.

Interestingly, the "Sick" model is actually the inverse of the "No Finding" model. The only difference is what is defined as positive: healthy images or sick images. Despite being so similar, the "Sick" model achieves much better results than the "No Finding" model. The "Sick" model should learn to find patterns that indicate disease, while the "No Finding" model technically should find the "absence" of these patterns. It could be that it is easier for the model to learn if a feature exists rather than if it does not. This brings the necessity of the "No Finding" model in question, as the question of whether or not an image is healthy can technically be answered but better by a "Sick" model.

5.1.4 Visualization of the model predictions

One of the biggest challenges with deep learning is that it can be extremely difficult to interpret how the models actually work and what they have learned during training[40]. A model trained to detect tumours may actually not look at the tumours when it makes its decision, which is a problem.

To visualize how the models work, a number of Grad-CAM localization maps were generated from images in the test set, see Figure 4.15. The heat maps give some explanation to the poor results of the models. For some models, e.g. "Consolidation" or "Enlarged Cardiomediastinum", the heat maps show that models have learned some feature occurring outside of the lungs, even outside of the body, which is not a good result when the aim is to locate lung diseases. Relating back to the discussion in the previous section about the "No Finding" model, Figures 4.15a and 4.15b shows that almost all pixels in the image are important to classify the images as instances of the class "No Finding". Does this mean that the model looked everywhere and could not find a trace of disease? Or does it mean that it finds an upper body x-ray in the image? It is difficult to tell either way.

This shows that the models can learn "wrong" features to represent a class. For example, the "Pneumonia" model focuses a lot on the clavicles of the patient(see Figure 4.15c). Most likely all images positive for pneumonia are going to have the patients clavicles in the image, which causes the model to associate clavicles with the "Pneumonia" label. This explains the bad results of the "Pneumonia" model.

An example of a better Grad-CAM belongs to the "Sick" model which appears to have learned features present within the lung area. This also reflects the good results of the "Sick" model.

5.1.5 Improving the results

The results obtained from the models could be improved by further fine-tuning of the models. Due the amount of time it took to train all 14 models, there was not enough time to do a lot of fine tuning. Tuning the different hyperparameters, such as batch size or number of epochs, could therefore lead to better models. One thing to consider would be to tune the models separately, as a certain kind of tuning may be beneficial for one model but detrimental for another. Another thing that could possibly improve the models is some sort of regularization.

As previously discussed, a big factor in the results of the trained models probably lies in the data and the imbalance between classes. The models could therefore improve by evening out the class imbalance (either by adding or removing data) or by simply training with more data.

Another thing that potentially could benefit the models is different preprocessing of the images. The difference preprocessing can make is evident when considering the "Sick" model. This model did not well in the approach described in Section 3.3.1 which used little to no preprocessing. By training this model using the preprocessing and a slightly different implementation, it improved drastically. Hence, it could be interesting to explore if some other preprocessing could improve the models further.

5.2 Method

This section discusses the method presented in Chapter 3. As previously explained, multiple approaches were investigated with the binary relevance approach being used as the final approach.

5.2.1 The binary relevance approach

The final method used was to train multiple models using the binary relevance method. Binary relevance is a *problem transformation method*[42] since it transforms a multi-label classification into several binary classification problems. The main advantages of using this method is that its implementation is straightforward and it is much easier to evaluate the models' performance. Another advantage of using this method is that it was also possible to classify individual disease labels. Obtaining information about exactly which disease the model has located can be helpful (even though it is not technically need to create a triage) and this is something that would not be possible if all sickness labels were combined into a single class.

One of the disadvantages of the binary relevance method is that it assumes absolute independence between the different classes[18]. While for some situations this is true, there often exist some kind of dependency or relationship between classes. By treating them as independent from each other the data can be misrepresented. Dependencies between class labels can also be exploited to improve the model's performance[43]. In the case of this thesis, it is possible that one observation or disease can indicate the presence of some of the other classes. For example, pleural effusion is a condition where the area around the lungs are filled with an unusual amount of fluid[44]. This condition has many different causes, one of them infections like pneumonia. This means that the presence of the "Pleural Effusion" class could also indicate the presence of the "Pneumonia" class.

Another disadvantage of the used method is that, while the training and evaluation became simpler, training 14 models naturally took longer time compared to training only a single binary or multi-label classifier. Since all weights in the models were trained, this further extended the training time. Each model took somewhere between 15-25 hours to train.

This meant that the training was carried out over several days. In this regard, successfully training a single multi-label classifier to give 14 different outputs simultaneously would have been a lot faster.

Another major drawback of having multiple models is the time it takes to actually predict sortable for a bulk of images. Because of how Keras works, it is not possible to hold several models in memory simultaneously. This means that to classify the images, the images would first need to be prepared in the same way as described in Section 3.3.3, load the first model, let the model make a prediction on the images, store the predictions and lastly remove the first model from memory before repeating the process for all remaining models. The main time consuming factor here is having to load and remove the models from memory. Predicting values for a bulk of images is not that bad since the prediction can be done reasonably fast once the model has been loaded. In contrast, using a multi-label classification model would have made it possible to get predicted values for all class labels at once, meaning the model only has to load once which would have been a lot faster. It is possible that this issue could be improved by using multi-threading and running the models on parallel threads.

Having to work with multiple models quickly becomes less flexible. The result using this approach are 14 separate predictions for one image. As mentioned previously, the benefit of using different models is that it is possible to gather information about which disease the model has detected. However, in order to make a triage, only a single value per image is needed . This means that there needs to be some sort of extra rule or strategy in order to assign the image with only one sortable value. One such strategy could be to simply take the maximum predicted value of all the models and use it to definitively classify the image (to compare the diseases to each other they will have to be treated as a singular class in the end anyway). Note that if the maximum value comes from the "No Finding" class, this value must be reinterpreted if it is to be compared to other disease classes, i.e. the predicted value should be somehow inverted. Therefore, there needs to be a special check if the "No Finding" model is used in conjunction with the other models.

The inflexibility of having multiple models is the biggest issue of the binary relevance approach compared to the other approaches investigated in Chapter 3. Training one multi-label model would produce the same results. Training multi-label classification models have been done before[9], so it there is value in investigating this approach further: while there were issues using it in this thesis, the results could be improved, perhaps by using different data or fine-tuning the models better. The experimental "Sick" model (which achieved the best results) also indicates that there is value in investigating that approach further, since with better preprocessing and better implementation it proved to work fairly well. As it stands, it is feasible to use the binary relevance approach to create a triage, but doing so can be cumbersome.

5.2.2 The usage of pre-trained model architectures

As explained in Chapter 3, the method used pre-trained model architectures. The choice to use already existing architectures was made because building the the models from scratch would have been an entire project in its own right. While it is possible that it could have been an advantage to design the network from bottom up (making it tailor-made for the task in this thesis), the existing architectures have been shown to work very effectively for a large variety of tasks and they have been studied in great detail by experts. All in all, three different architectures were tested during the implementation: VGG16, DenseNet121 and ResNet50.

The first model tested was DenseNet121. This model was initially chosen because it had been used in similar projects[9] and it is a very deep network. Since the images in this task are fairly complex (even for a human expert it can be challenging to determine diseases in chest x-rays), a deeper network should in theory be able to detect advanced enough features to classify the chest x-rays. However, in the approaches it was tested, DenseNet121 did not

appear to perform well: it usually overfitted quickly and took a long time to train. Overall, DenseNet121 was very challenging to train and was not very accurate.

VGG16 was tested as a contrast to DenseNet121: it is a much smaller network, which meant it could be trained a lot faster and thus, if it could learn to classify chest x-rays even with a smaller number of layers it would be considered a better model. While VGG16 was easier to train, it was not accurate.

The final model used was ResNet50 which was chosen as a middle point between the two previous models. It was easier to train than DenseNet121 but still had considerable depth and appeared to work well for at least some of the classes.

These models were initially trained using transfer learning and weights from ImageNet. The ImageNet data set contains 1000 different classes, most of them very different than chest x-rays, e.g. different species of animals. When using transfer learning, all layers but the output layer were frozen. This means that there could have been filters specialized for finding cats for example, which does not really help when classifying x-rays. This could be the reason why the transfer learning approach did not work very well. It is possible that it could have worked better by unfreezing and training more layers.

There are several possible reasons as to why training classifiers using these models as base models failed, and it does not necessarily have to do with the architectures themselves. The preprocessing of the images, the hyperparameters chosen, the use of transfer learning, class imbalance in the data, etc., could all be contributing factors in the poor performance. Hence, the VGG16 and DenseNet121 should not be ruled out as bad models without further testing.

5.2.3 The CheXpert data set

The CheXpert data set[34] used in this thesis is a data set of anonymized chest x-rays. The data was collected from Stanford Hospital during the time period between October 2002 and July 2017, resulting in a large data set.

One of the main reasons for choosing to use CheXpert in this thesis was its size: typically, it is difficult to find large data sets containing medical data due to patient privacy and security. Medical data is sensitive information and therefore all images must be gathered with the consent of the patient and anonymised so they cannot be traced back to the patient. CheXpert was collected and anonymised with the intention of being used in research, thus making it a good choice for this thesis.

The CheXpert data set contains over 200,000 images but only about half of them were used in this thesis. This was mostly due to the preprocessing done to handle uncertainty labels described in Section 3.2. By removing all images containing an uncertainty label, over 100,000 images were removed. Removing images meant removing important positive examples from classes, leading the models to have less data to learn from (see Table 3.3). This could definitively have an effect the both the training and results achieved from the models.

This type of preprocessing meant that the full potential of the CheXpert data set could not be realized. By using a different strategy to handle the uncertainty labels, it could have been possible to use more of the data when training the models, which could have helped improve the model performance. Irvin and Rajpurkar et al. presents a number of different strategies for dealing with the uncertainty labels[34]. In any future work, it would be interesting to see how this would affect the training of the models.

While the CheXpert has several advantages, it is not without criticism. One criticism that could be directed towards CheXpert is the way the images are labeled. The images are labeled using Natural Language Processing to extract labels from reports corresponding to the images. This means that the labels are not based on information in the x-ray images themselves, which means that the classes in the data set may not be possible to infer from the images alone (which is how the CNN models work). There is also a problem when considering the variability in the data, as a large number of the images comes from the same patients, meaning that a lot of images are very similar. Oakden-Rayner expresses these as concerns in

a first impressions-review of the data set, but also comments that the CheXpert data set fixes a lot of faults present in similar earlier data sets[45].

One of the advantages of CheXpert is that it contains chest x-rays taken from different patient positions. The data set contains both lateral and frontal views, as well as AP (anteroposterior) and PA (posteroanterior) views (referring to the direction from which the x-rays passes through the body, i.e. from the front and back respectively). The major benefit from this is that some diseases or injuries can only be seen from specific views, such as from the side. This also means that images that are labeled the same can look quite different, especially considering frontal and lateral images. The models in this thesis were trained on all different view-types at once. This could potentially be an issue since the model could start to learn a pattern from the frontal views, which it then cannot apply to an image taken from the side. An example of where this might be the case can be seen in Figures 4.15e and 4.15f: the model seem to detect something the bottom left corner of the lungs on the frontal view but when it applies the same logic on the lateral view, it focuses on an area outside of the lung. It is possible that mixing images taken from different views could have negatively impacted the results. It possible that the results could be improved by separating the data based on the view direction and training individual models for the specific view direction. Rubin et al. used a similar approach: they trained separate models for different view-types and fused them to produce one output in a DualNet architecture[8]. It could therefore be of interest to investigate this further in future work.

5.2.4 The usability of the method

To summarize, the method used in this thesis can work but is inaccurate for most of the classes. While the models themselves could potentially be improved to be more accurate through further fine-tuning, the method of using binary relevance to train multiple models can inflexible and slow to work with. It may therefore be interesting to further investigate the other approaches to the problem and see if they can be useful with a different method.

5.3 The work in a wider context

The rapid advancement of technology over the past century has changed and is changing many parts of both industry and general society. Automatization of routine tasks means that many jobs previously performed by humans can be replaced by computers and algorithms. There is thus a concern that sometime in the future, AI will cause mass-unemployment[46]. Frey et al. did a study on the probabilities of different professions becoming automated and found that professions foremost within transportation and logistics are leading the highest risk of being overtaken by the rapid advancement of technology[47]. They conclude that as technology advances further, workers will have to adapt to tasks requiring social and creative intelligence.

5.3.1 Will AI-algorithms replace doctors?

The growing usage of deep learning within the medical field has many posing the question if doctors will one day be replaced by AI. There are many differing opinions about this question, but even if deep learning cannot completely replace doctors, it will still change the tasks a doctor has to perform.

R. Susskind and D. Susskind state that there are still things that can only be done by a human and not computers, such showing creativity, empathy and performing other non-routine tasks[48]. Breaking down the job of a doctor into many parts shows that many tasks do not require these skills and can therefore be replaced using computers. However, because doctors have human interactions with their patients, these are still skills that are useful within the field of medicine.

Davenport and Dreyer discusses the future of the radiologist profession in their article *AI Will Change Radiology, but It Won't Replace Radiologists* and says that while the use of AI and disease detecting neural networks (like those in this thesis) will change the profession, it will not replace it[49]. They attribute this conclusion to the fact that the task of a radiologist covers more than analysing images, such as consulting with other experts, discussing treatments with patients, etc. Even if deep learning solutions become more reliable in the future, radiologists can focus their attention to other important tasks. They conclude that radiologists will need to adopt new skills to not be replaced by machines.

5.3.2 Patient security

One of the most important aspects when creating a tool for medicinal use is to analyse the potential negative effects it may have on a patient. This is especially important when considering AI-based tools, as an AI cannot be guaranteed to make the same analysis and decisions as a trained human professional would. If the tool does not work as intended it could lead to severe consequences, such as misdiagnosis, personal injury or even death. Hence, it is important to evaluate what kinds of effects the lung triage may have on the patient.

As explained in Section 1.1, the purpose of the lung triage is to sort a bulk of images based on the degree of sickness in order to help a radiologist find patterns more quickly. As explained in Section 1.4, the lung triage is limited from giving any sort of diagnosis based on the x-ray images and can at most give an indication to the radiologist what to look for in the image (if the model outputs a distinct pathology label). The task of actually diagnosing the patient falls on the radiologist or doctor examining the x-rays. It should also be noted that many lung diseases are diagnosed with the help of both chest x-rays and other clinical information, something the models themselves do not have access to, which makes the role of the radiologist all the more important.

Because the triage is used to sort a bulk of images, at worst the triage would result in an arbitrary sorting of the images, which is no worse than what it is today. All the images would still need to be examined, but with the triage some abnormality could potentially be found faster. One thing to consider is if the model shows which pathology it thinks the image is, this could influence the opinion of the radiologist for better or worse.

5.4 Source criticism

Most of the literature used a base for this thesis has been collected from different scientific databases. The databases used were mainly IEEE Xplore Digital Library[1], Google Scholar[2] and arXiv.org e-Print archive[3].

The base for the theory presented in this report comes from mainly two sources: the book *Deep Learning* by Goodfellow et al.[2] and the paper *Deep Learning* by LeCun et al.[3]. The authors of the book *Deep Learning* are Ian Goodfellow: researcher working in machine learning currently employed at Apple Inc, Yoshua Bengio: a professor in the Department of Computer Science and Operational Research at the University of Montreal, and Aaron Courville: an assistant professor in the Department of Computer Science and Operational Research at the University of Montreal. The authors of the paper *Deep Learning* are Yann LeCun: VP and Chief AI Scientist at Facebook, Geoffrey Hinton: professor in Computer Science dividing is time between working at the University of Toronto and Google, and Yoshua Bengio. These sources have been cited over 8000 times, hence they are considered reliable.

[1] IEEE Xplore Digital Library: `https://www.ieee.org/`
[2] Google Scholar: `https://scholar.google.se/`
[3] arXiv.org: `https://arxiv.org/`

6 Conclusion

This thesis investigates how well convolutional neural networks can be used to create a lung triage that can prioritize chest x-rays based on a degree of disease. Using AI to sort a bulk of chest x-ray images from sickest to healthiest could help radiologists find abnormalities more quickly, which can help improve the workflow.

To solve this task a total of 14 different CNN models were trained using the binary relevance method as well as an additional experimental model trained on a combined "Sick" class. The 14 models were trained to detect common observations in chest x-rays. The models were trained from scratch using the pre-existing ResNet50 model architecture. The results obtained from most of the models are not satisfactory and do not compare to state-of-the-art models, probably due to insufficient data or insufficient fine-tuning of the models. The models with more balanced data performed comparatively better. The best model obtained from training was the single binary "Sick" classifier model.

Using the binary relevance approach to create a lung triage is feasible, but it can be inflexible to work with due to having to handle multiple models. Because multiple predictions are obtained for every image, some strategy must also be used to decide the definitive predicted label for the image before it can be sorted. The usefulness of the lung triage depend heavily on the accuracy of the underlying models, meaning that since the models trained in this thesis achieved unsatisfactory results, the results from the triage are also lacking.

While the results from the CNN models are far from satisfactory, previous works as well as the experimental "Sick" model indicates that there is still a possibility for training CNNs to detect diseases in lungs accurately. Using more data, removing class imbalance in the data and further fine-tuning of the models are all measures that could be taken to improve the models in the future.

Other approaches to tackle the task were investigated: multi-label classification and single binary classification. These approaches could not initially produce useful results, mostly due to implementation and evaluation issues. However, with changes to the implementation and an experimental model, the single binary classifier approach in the end produced the best model. Using the other approaches would automatically solve some the inflexibility issues present in the binary relevance approach. With better preprocessing, data and fine-tuning both the single binary approach and the multi-label approach could become more viable, as indicated by both previous works and the experimental "Sick" model. This makes it interesting to investigate these approaches further.

The models were tested on two different data sets: CheXpert and ChestX-ray14. This was to evaluate how well the models would be able to perform on data from different distributions. The results from testing the overlapping models on ChestX-ray14 revealed that the models performed worse on the new data set compared to the data set used during training. This indicates that the models have further difficulty generalizing to data from other distributions. The generalization error for new data sets could possibly be lowered through more extensive preprocessing of the image, making them more like the images in the training set.

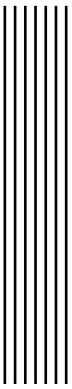
6.1 Future work

As previously explained, many things can be done in order to try and improve the results of the models in this thesis, for example further fine-tuning and more labeled data. In any future work, other combinations of hyperparameters should be tested to try and improve the models.

One of the major problems in the method is the handling of data, which was more or less halved in size during the preprocessing. Further development that can be done in this project is therefore to investigate other strategies for dealing with the uncertainty labels in CheXpert, as this would lead to more available data which is beneficial to the models.

One thing that should evaluated more are the classes in CheXpert and their actual usefulness when creating a lung triage. For example, is there really a need for the "No Finding" class, since its results could be implicitly obtained from the just training the disease models? Or what is the necessity of classes like "Support Devices" and "Fractures", both of which are not lung diseases, when creating a lung triage?

As mentioned earlier, it would be interesting to further investigate the single binary approach and multi-label approach, as they fix a lot of inflexibility issues present with the binary relevance approach. Another thing that could be interesting to investigate is more extensive preprocessing of the images. This could help improve the model results and also help the model to generalize better on data from other distributions.



Bibliography

[1] Tom M. Mitchell. *Machine Learning*. en. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[4] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. "A survey on deep learning in medical image analysis". In: *Medical image analysis* 42 (2017), pp. 60–88.

[5] Yaniv Bar, Idit Diamant, Lior Wolf, Sivan Lieberman, Eli Konen, and Hayit Greenspan. "Chest pathology detection using deep learning with non-medical training". In: *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*. Apr. 2015, pp. 294–297. DOI: 10.1109/ISBI.2015.7163871.

[6] Jia Deng, Wei Dong, Richard Socher, Li-jia Li, Kai Li, and Li Fei-fei. "ImageNet: A large-scale hierarchical image database". In: *CVPR09*. 2009.

[7] Hoo-Chang Shin, Kirk Roberts, Le Lu, Dina Demner-Fushman, Jianhua Yao, and Ronald M. Summers. "Learning to Read Chest X-Rays: Recurrent Neural Cascade Model for Automated Image Annotation". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 2497–2506. DOI: 10.1109/CVPR.2016.274.

[8] Jonathan Rubin, Deepan Sanghavi, Claire Zhao, Kathy Lee, Ashequl Qadir, and Minnan Xu-Wilson. "Pneumothorax Detection and Localization in X-Ray Images Given Richer Annotation Information". In: (Apr. 2018).

[9] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, Matthew P. Lungren, and Andrew Y. Ng. "CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning". In: *arXiv:1711.05225 [cs, stat]* (Nov. 2017). arXiv: 1711.05225. URL: http://arxiv.org/abs/1711.05225 (visited on 02/01/2019).

[10] Andrew Ng. "CS229 Lecture notes". In: *CS229 Lecture notes* 1.1 (2000), pp. 1–3.

[11] Christopher M. Bishop. *Pattern recognition and machine learning*. en. Information science and statistics. New York: Springer, 2006. ISBN: 978-0-387-31073-2.

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html.

[13] Yi-Tong Zhou and Rama Chellappa. "Computation of optical flow using a neural network". In: *IEEE 1988 International Conference on Neural Networks*. IEEE, July 1988, 71–78 vol.2.

[14] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. Aug. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[15] Francois Chollet. *Deep Learning with Python*. 1st. Greenwich, CT, USA: Manning Publications Co., 2017. ISBN: 1617294438, 9781617294433.

[16] David Stutz. "Understanding Convolutional Neural Networks". In: *Lehr- und Forschungsgebiet Informatik VIII Computer Vision*. 2014.

[17] Min-Ling Zhang and Zhi-Hua Zhou. "A Review on Multi-Label Learning Algorithms". In: *IEEE Transactions on Knowledge and Data Engineering* 26.8 (Aug. 2014), pp. 1819–1837. ISSN: 1041-4347. DOI: 10.1109/TKDE.2013.39.

[18] Mohammad S Sorower. *A literature survey on algorithms for multi-label learning*. Tech. rep. 2010.

[19] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020, 9780262018029.

[20] Boris Polyak. "Some methods of speeding up the convergence of iteration methods". In: *Ussr Computational Mathematics and Mathematical Physics* 4 (Dec. 1964), pp. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

[21] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Dec. 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 02/25/2019).

[22] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[23] Geoffrey Hinton. *Neural networks for machine learning*. 2012.

[24] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[25] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. "The Marginal Value of Adaptive Gradient Methods in Machine Learning". In: *arXiv e-prints*, arXiv:1705.08292 (May 2017), arXiv:1705.08292. arXiv: 1705.08292 [stat.ML].

[26] D. Randall Wilson and Tony R. Martinez. "The General Inefficiency of Batch Training for Gradient Descent Learning". In: *Neural Networks* 16.10 (Dec. 2003), pp. 1429–1451. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(03)00138-2. URL: http://dx.doi.org/10.1016/S0893-6080(03)00138-2.

[27] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[29] Foster J. Provost, Tom Fawcett, and Ron Kohavi. "The Case Against Accuracy Estimation for Comparing Induction Algorithms". In: *Proceedings of the Fifteenth International Conference on Machine Learning*. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 445–453. ISBN: 1-55860-556-8. URL: http://dl.acm.org/citation.cfm?id=645527.657469.

[30] Jesse Davis and Mark Goadrich. "The Relationship Between Precision-Recall and ROC Curves". In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania, USA: ACM, 2006, pp. 233–240. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143874. URL: http://doi.acm.org/10.1145/1143844.1143874.

[31] François Chollet et al. *Keras*. https://keras.io. 2015.

[32] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[33] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[34] Jeremy Irvin, Pranav Rajpurkar, Michael Ko, Yifan Yu, Silviana Ciurea-Ilcus, Chris Chute, Henrik Marklund, Behzad Haghgoo, Robyn Ball, Katie Shpanskaya, Jayne Seekins, David A. Mong, Safwan S. Halabi, Jesse K. Sandberg, Ricky Jones, David B. Larson, Curtis P. Langlotz, Bhavik N. Patel, Matthew P. Lungren, and Andrew Y. Ng. "CheXpert: A Large Chest Radiograph Dataset with Uncertainty Labels and Expert Comparison". In: *arXiv:1901.07031 [cs, eess]* (Jan. 2019). arXiv: 1901.07031. URL: http://arxiv.org/abs/1901.07031 (visited on 02/04/2019).

[35] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks". en. In: *arXiv:1608.06993 [cs]* (Aug. 2016). arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993 (visited on 03/04/2019).

[36] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv:1409.1556 [cs]* (Sept. 2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556 (visited on 05/22/2019).

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[38] Fabian Pedregosa, Gael Varoquaux, Alexander Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Eduard Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[39] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M. Summers. "ChestX-Ray8: Hospital-Scale Chest X-Ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases". en. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, July 2017, pp. 3462–3471. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.369. URL: http://ieeexplore.ieee.org/document/8099852/ (visited on 02/04/2019).

[40] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization". In: *arXiv e-prints*, arXiv:1610.02391 (Oct. 2016), arXiv:1610.02391. arXiv: 1610.02391 [cs.CV].

[41] Raghavendra Kotikalapudi and contributors. *keras-vis*. https://github.com/raghakot/keras-vis. 2017.

[42] Grigorios Tsoumakas and Ioannis Katakis. "Multi-label classification: An overview". In: *International Journal of Data Warehousing and Mining (IJDWM)* 3.3 (2007), pp. 1–13.

[43] Grigorios Tsoumakas and Min-ling Zhang. *LEARNING FROM MULTI-LABEL DATA*. 2009.

[44] April Kahn and Ana Gotter. *Fluid in the Chest (Pleural Effusion)*. Apr. 2018. URL: https://www.healthline.com/health/pleural-effusion (visited on 06/09/2019).

[45] Luke Oakden-Rayner. *Half a million x-rays! First impressions of the Stanford and MIT chest x-ray datasets*. en. Feb. 2019. URL: https://lukeoakdenrayner.wordpress.com/2019/02/25/half-a-million-x-rays-first-impressions-of-the-stanford-and-mit-chest-x-ray-datasets/ (visited on 02/27/2019).

[46] James Bessen. "AI and Jobs: The role of demand". In: *The Economics of Artificial Intelligence: An Agenda*. University of Chicago Press, 2018.

[47] Carl Benedikt Frey and Michael A Osborne. "The future of employment: how susceptible are jobs to computerisation?" In: *Technological forecasting and social change* 114 (2017), pp. 254–280.

[48] Richard E Susskind and Daniel Susskind. *The future of the professions: How technology will transform the work of human experts*. Oxford University Press, USA, 2015.

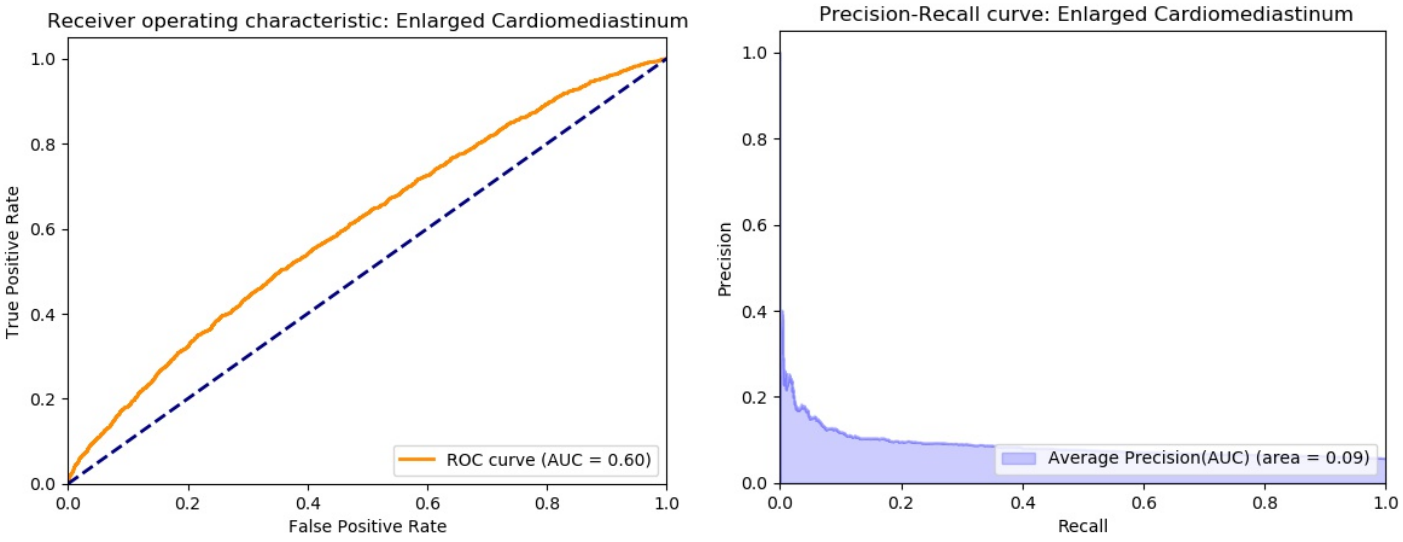
[49] Thomas H. Davenport and DO Keith J. Dreyer. "AI Will Change Radiology, but It Won't Replace Radiologists". In: *Harvard Business Review* (Mar. 2018). ISSN: 0017-8012. URL: https://hbr.org/2018/03/ai-will-change-radiology-but-it-wont-replace-radiologists (visited on 06/09/2019).

A Additional Results

This appendix contains additional results in the form of ROC curves, Precision-Recall curves and grad-CAM visualizations for the trained models.

A.1 ROC- and Precision-Recall curves

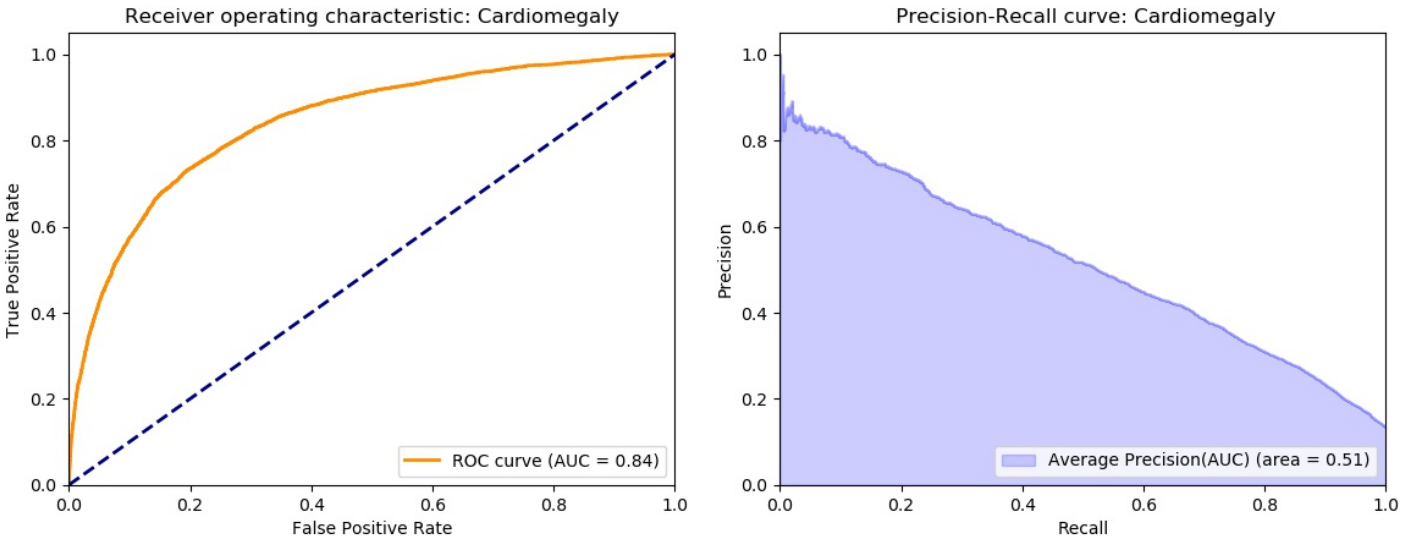
This section shows the resulting ROC and Precision-Recall curves for the remaining trained models.



(a) ROC curve

(b) PR curve

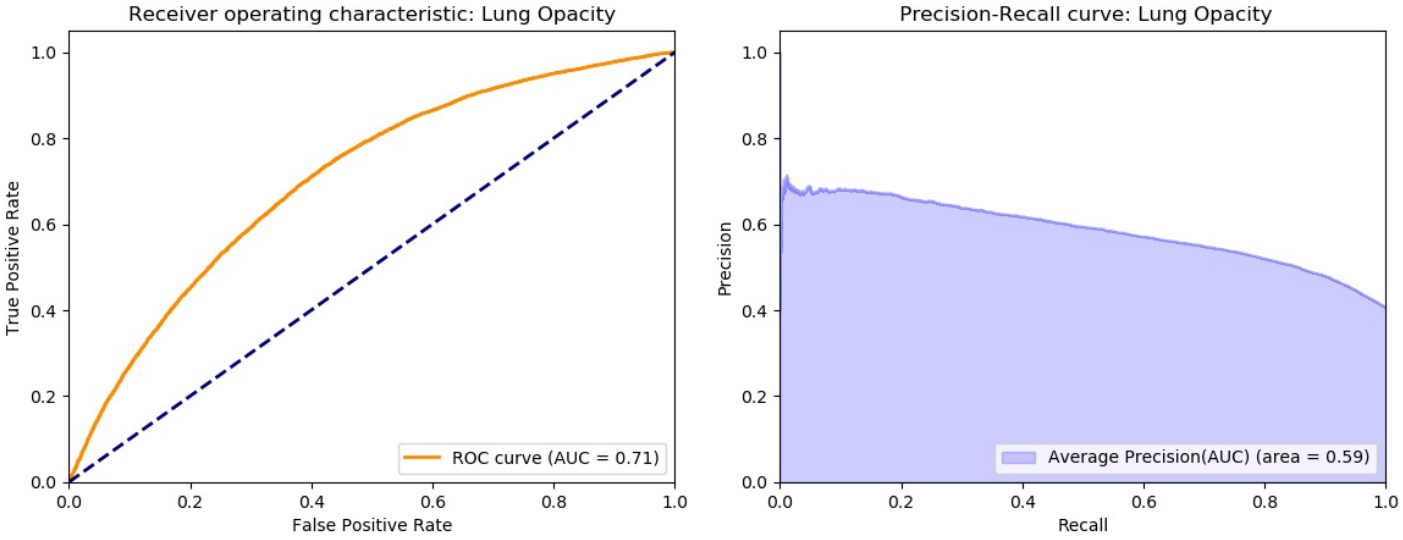
Figure A.1: Graphs for the Enlarged Cardiomediastinum class.



(a) ROC curve

(b) PR curve

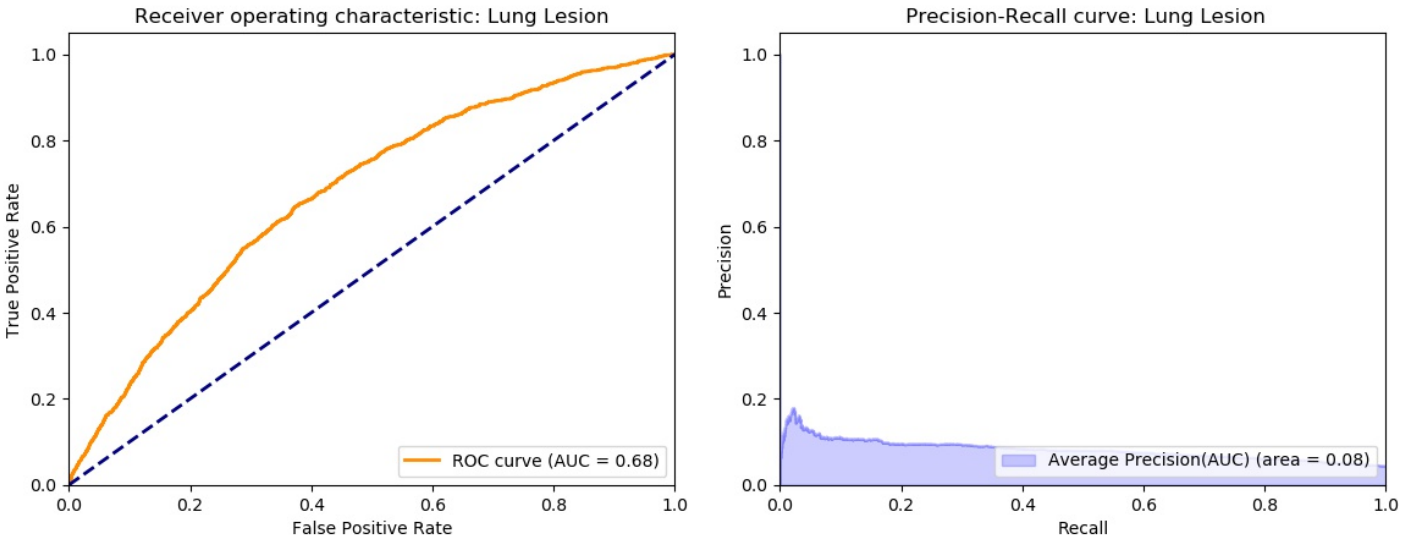
Figure A.2: Graphs for the Cardiomegaly class.



(a) ROC curve

(b) PR curve

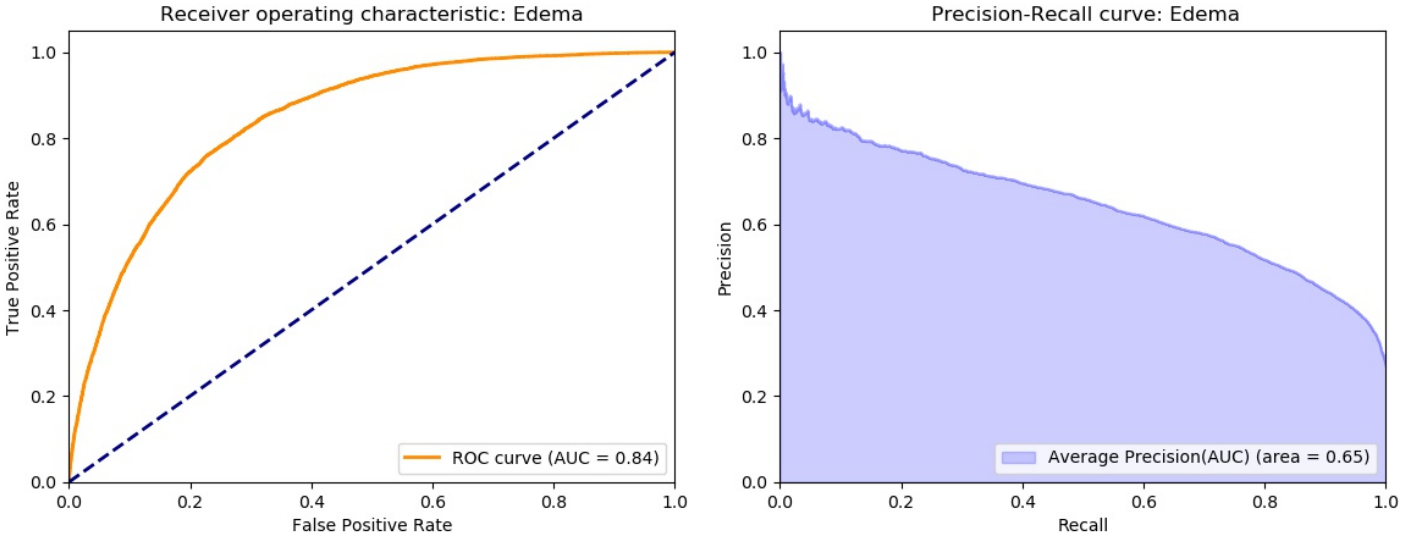
Figure A.3: Graphs for the Lung Opacity class.



(a) ROC curve

(b) PR curve

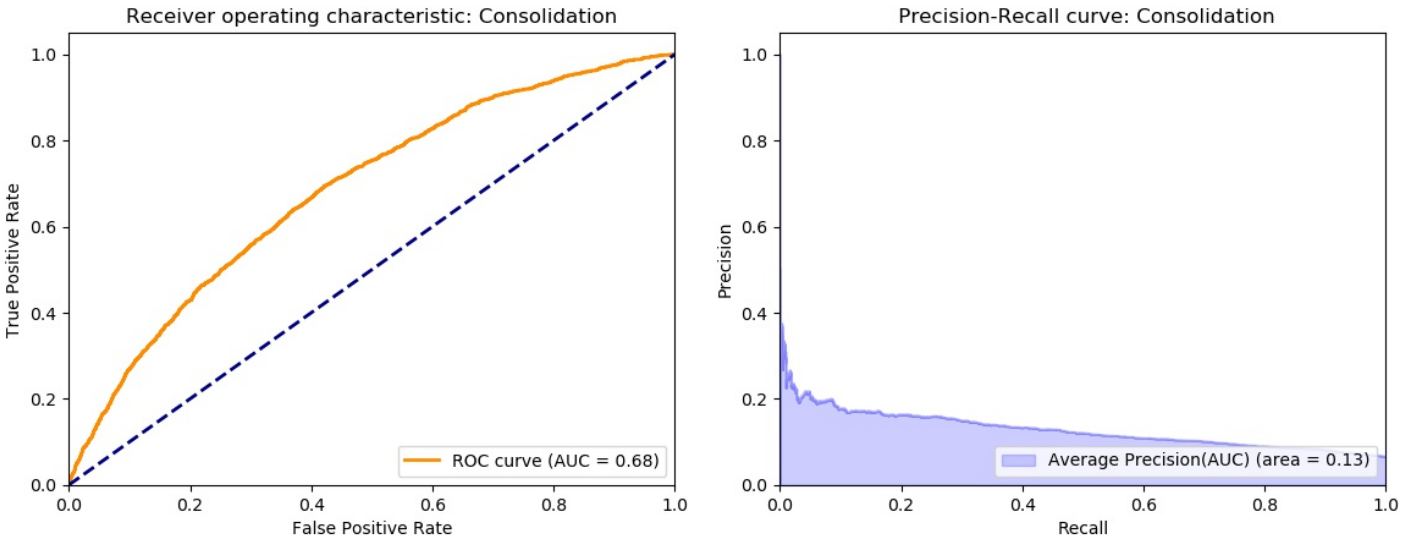
Figure A.4: Graphs for the Lung Lesion class.



(a) ROC curve

(b) PR curve

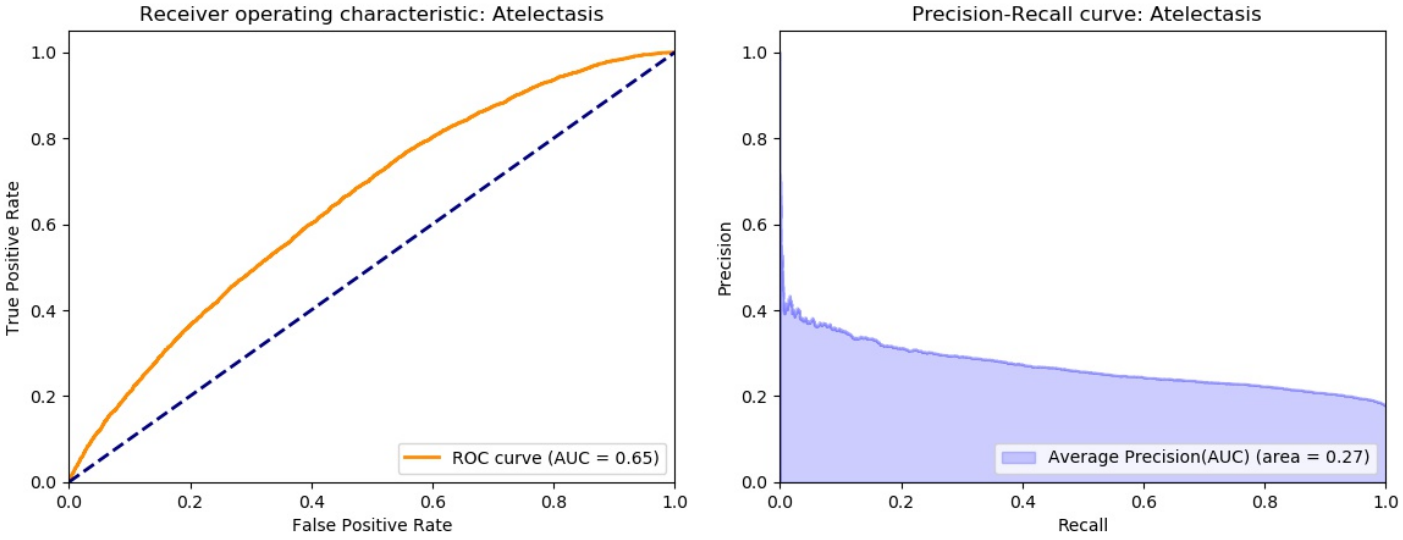
Figure A.5: Graphs for the Edema class.



(a) ROC curve

(b) PR curve

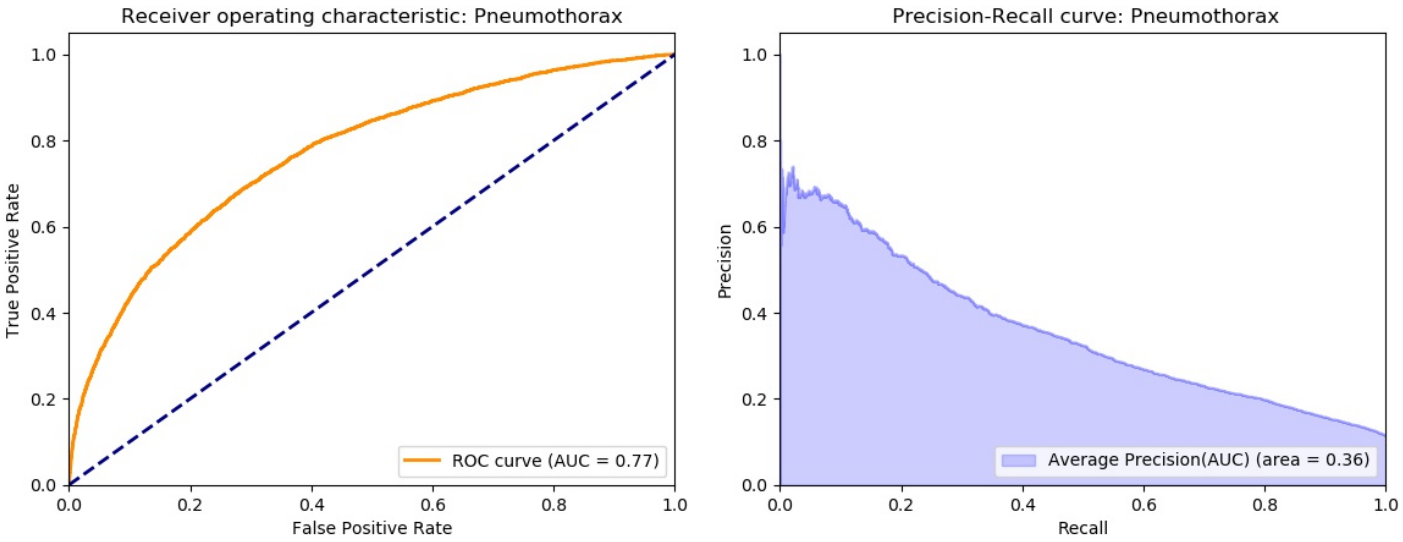
Figure A.6: Graphs for the Consolidation class.



(a) ROC curve

(b) PR curve

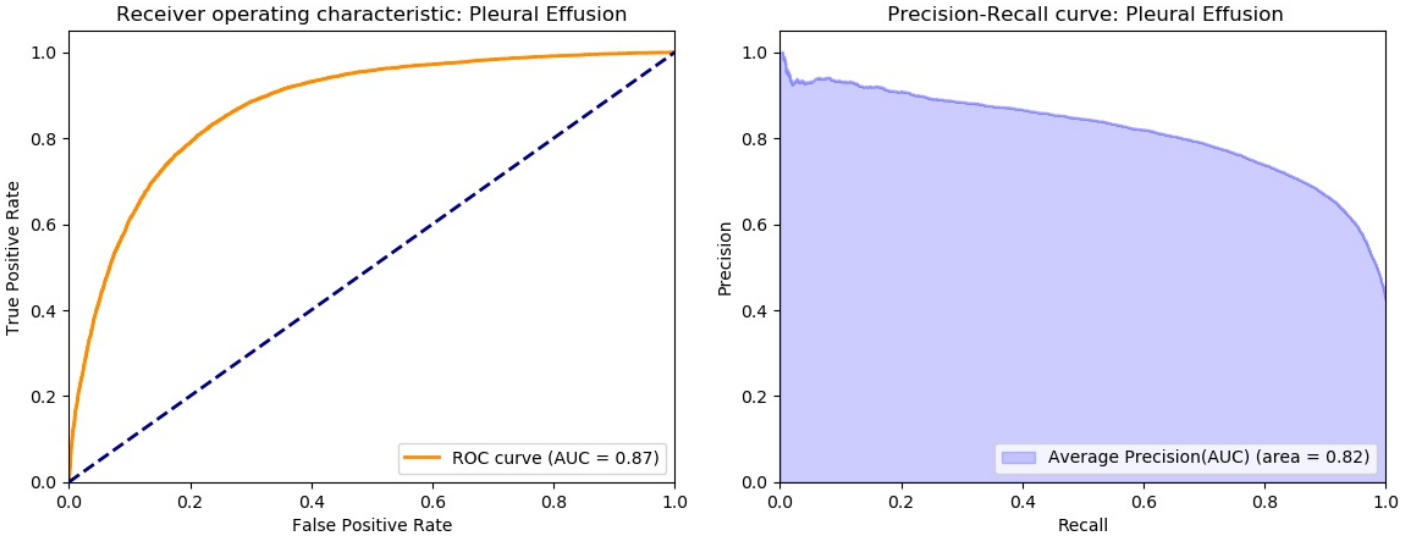
Figure A.7: Graphs for the Atelectasis class.



(a) ROC curve

(b) PR curve

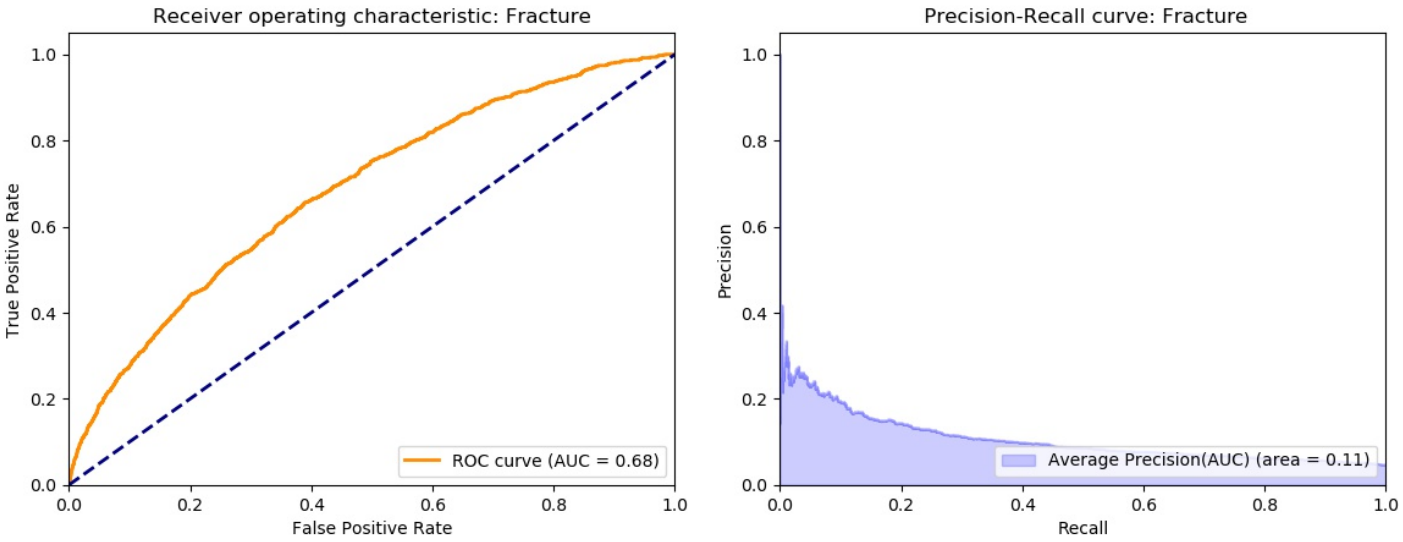
Figure A.8: Graphs for the Pneumothorax class.



(a) ROC curve

(b) PR curve

Figure A.9: Graphs for the Pleural Effusion class.



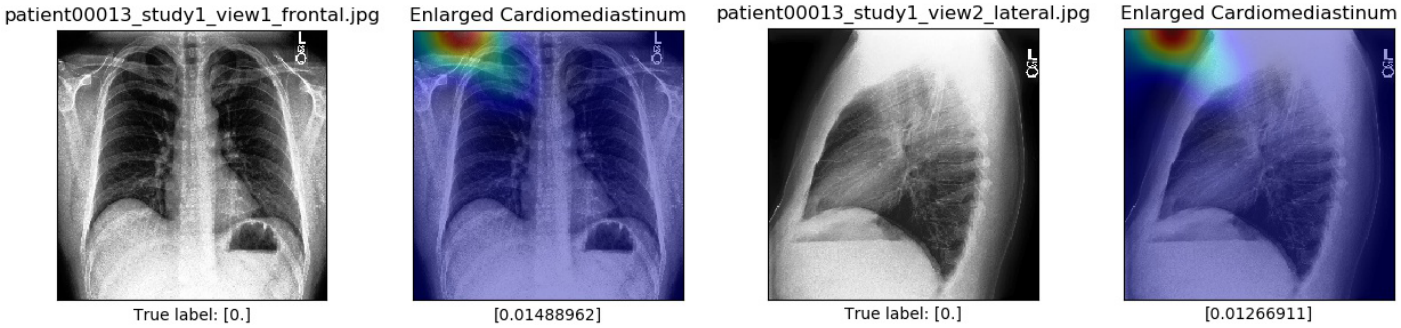
(a) ROC curve

(b) PR curve

Figure A.10: Graphs for the Fracture class.

A.2 Additional examples of grad-CAM visualization

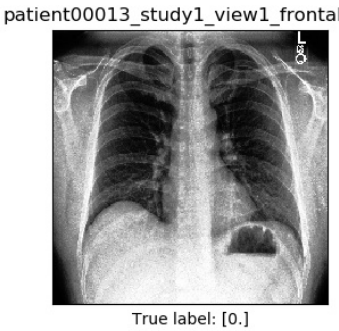
A.11 shows additional examples of grad-CAM visualization generated from the trained models.

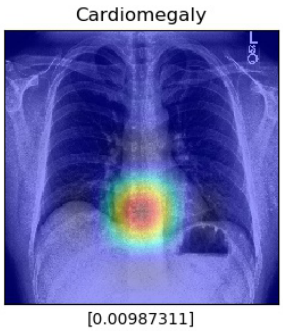


(a) Frontal Grad-CAM generated from the Enlarged Cardiomediastinum model.

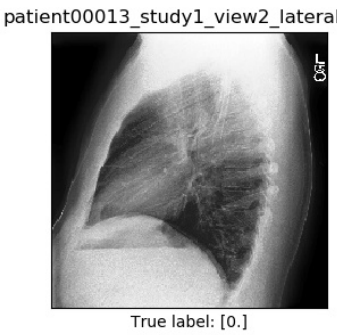
(b) Lateral Grad-CAM generated from the Enlarged Cardiomediastinum model.

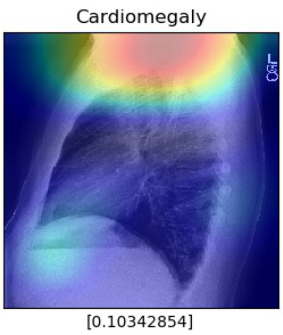
Figure A.11: Examples of generated Grad-CAMs.



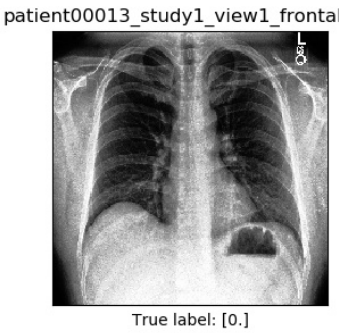


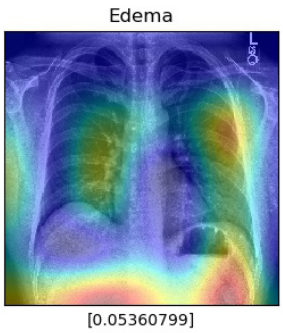
(c) Frontal Grad-CAM generated from the Cardiomegaly model.



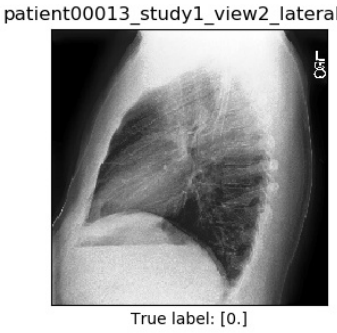


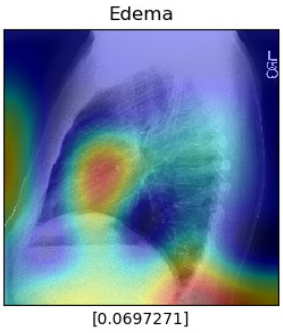
(d) Lateral Grad-CAM generated from the Cardiomegaly model.



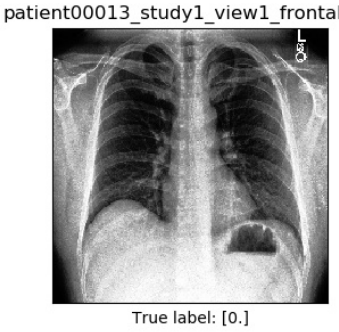


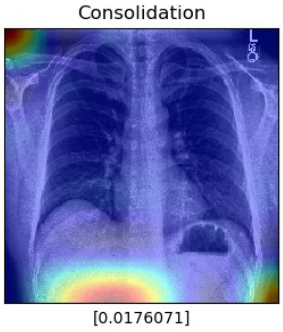
(e) Frontal Grad-CAM generated from the Edema model.



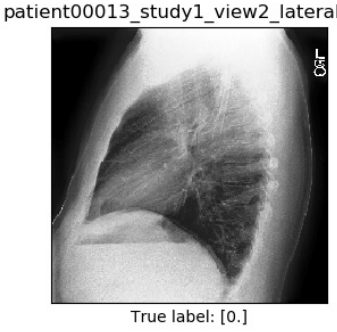


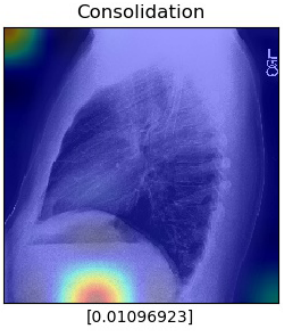
(f) Frontal Grad-CAM generated from the Edema model.





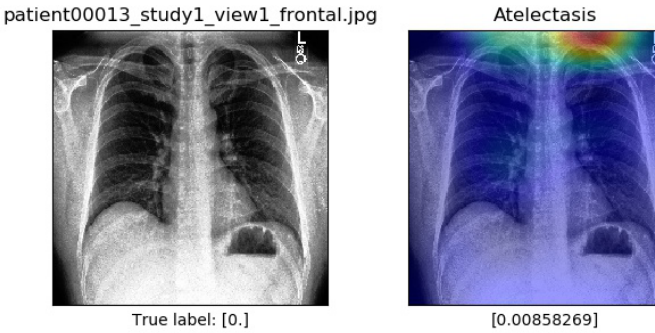
(g) Frontal Grad-CAM generated from the Consolidation model.



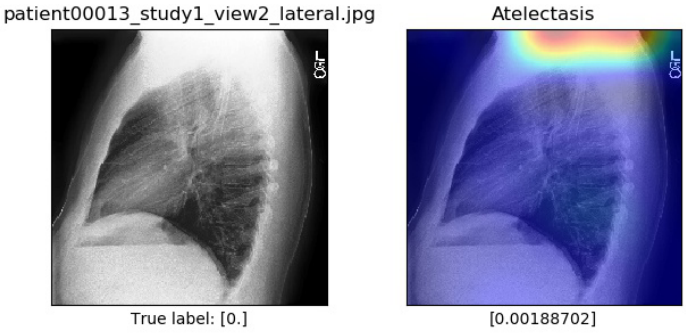


(h) Lateral Grad-CAM generated from the Consolidation model.

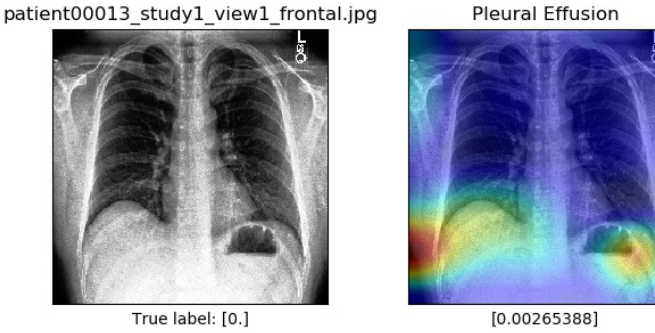
Figure A.11: Examples of generated Grad-CAMs (continuation).



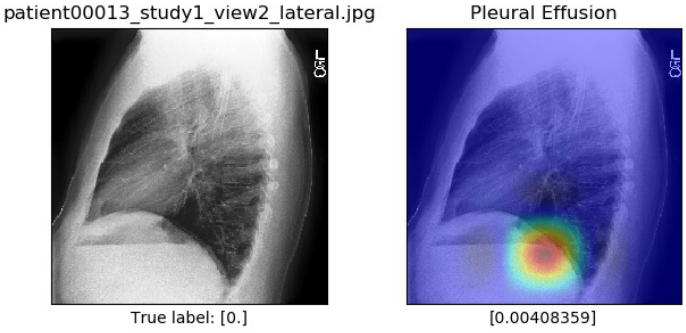
(i) Frontal Grad-CAM generated from the Atelectasis model.



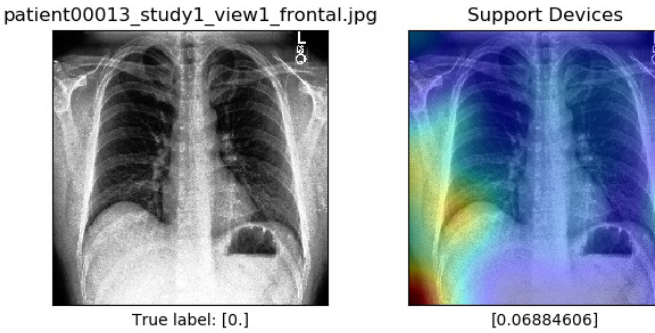
(j) Lateral Grad-CAM generated from the Atelectasis model.



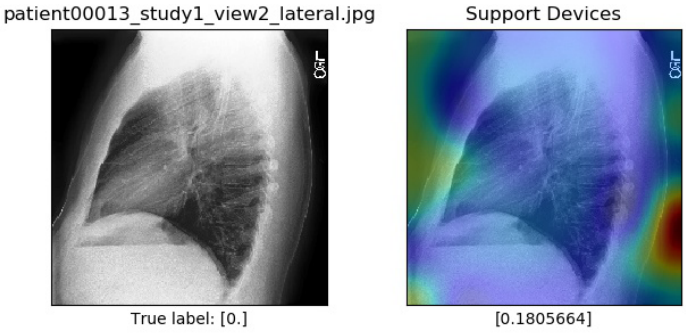
(k) Frontal Grad-CAM generated from the Pleural Effusion model.



(l) Lateral Grad-CAM generated from the Pleural Effusion model.



(m) Frontal Grad-CAM generated from the Support Devices model.



(n) Lateral Grad-CAM generated from the Support Devices model.

Figure A.11: Examples of generated Grad-CAMs (continuation).