Tampere University

Ville Parkkinen

CLASSIFICATION OF DAMAGE TYPES IN MOBILE DEVICE SCREENS

Using lightweight convolutional neural networks to detect cracks and scratches

Faculty of Engineering and Natural Sciences
Master's thesis
April 2020

ABSTRACT

Ville Parkkinen: Classification of damage types in mobile device screens
Master's thesis
Tampere University
Science and Engineering, MSc
April 2020

Evaluating the condition of used mobile devices is important part of the process of reselling and recycling smart phones and tablets. Damages on the device screen are usually identified and their severity is classified manually by visual inspection. This can lead to inconsistent and biased results. In this thesis an automated method utilizing a convolutional neural network is proposed to automate this task.

In order to make the neural network classifier usable in practical applications, it must be fast enough to perform within reasonable time even in devices with limited computational resources. The high classification accuracy of convolutional neural networks comes with high computational cost. Lightweight convolutional neural network architectures have been designed to achieve reasonable accuracy with fast inference times.

In this work popular lightweight neural network architectures are described and fine-tuned to classify damages on mobile device screens. Several methods for optimizing the accuracy and inference time are also experimented with. The most accurate network trained in this work classifies damages on mobile device screen with 84.8% accuracy in 3 seconds.

Keywords: convolutional neural network, mobile device screen, crack, scratch

TIIVISTELMÄ

Ville Parkkinen: Vahinkotyyppien luokittelu mobiililaitteiden näytöissä
Diplomityö
Tampereen yliopisto
Teknis-luonnontieteellinen, DI
Huhtikuu 2020

Mobiililaitteen kunnon arviointi on tärkeä osa käytettyjen älypuhelinten ja tablettien jälleenmyynti- ja kierrätysprosessia. Laitteen näytön vaurioiden tunnistaminen ja vaurion vakavuuden luokittelu tehdään usein käsin tarkastelemalla visuaalisesti näytön kuntoa. Tämä voi johtaa epäjohdonmukaisiin ja puolueellisiin arviointeihin. Tässä työssä kehitetään konvoluutioneuroverkkoon perustuva automaattinen menetelmä näytön kunnon arviointiin.

Neuroverkkoon perustuvan luokittelijan käyttö rajoitetun laskentatehon laitteissa, kuten mobiililaitteissa, edellyttää suorituskykyvaatimusten huomioon ottamista jo suunnitteluvaiheessa. Konvoluutioneuroverkot kykenevät hyvään luokittelutarkkuuteen, mutta niiden käyttö vaatii runsaasti laskentatehoa. Kevyiden konvoluutioneuroverkkoarkkitehtuurien kehitys on mahdollistanut sekä kilpailukykyisen luokittelutarkkuuden että nopeuden saavuttamisen myös mobiililaitteissa.

Tässä työssä esitellään suosittuja kevyitä konvoluutioneuroverkkoarkkitehtuureja sekä hyödynnetään niitä mobiililaitteiden näytön vaurioiden tunnistamisessa. Lisäksi työssä kokeillaan useita eri menetelmiä luokittelijan tarkkuuden ja nopeuden parantamiseksi. Tarkin työssä koulutettu neuroverkko pystyy tunnistamaan vauriot mobiililaitteen näytöltä 84,8 %:n tarkkuudella noin kolmessa sekunnissa.

Avainsanat: konvoluutioneuroverkko, mobiililaitteen näyttö, halkeama, naarmu

PREFACE

This thesis was written as an assignment for Piceasoft Ltd. I would like to thank my superiors at Piceasoft, especially Jani Väänänen, Samuli Kivinen and Samuli Ylinen for a chance to work on such interesting and challenging project.

I would also like to thank my supervisors at Tampere University, Professor Tapio Elomaa and Associate Professor Heikki Huttunen, for guidance and teaching during this project and throughout my studies.

Tampere, 24th April 2020

Ville Parkkinen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

C	Total cost value of neural network
α	Width multiplier of MobileNet
λ	Learning rate
$\mathbf{W}$	Weight vector of a network
$\mathbf{w}$	Weight vector of a neuron
ϕ	Compound scaling coefficient of EfficientNet
ρ	Resolution multiplier of MobileNet
a	Activation of a neuron
b	Bias value
d	Depth of a image
g	Loss function
h	Height of a image
k	Kernel size
n^l	Number of neurons in a layer
n_k	Number of kernels in a convolutional layer
t	Expansion factor of inverted residual block
w	Width of a image
z	Output of neuron before applying activation function
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
FLOPs	Floating Point Operations
GPU	Graphics Processing Unit
MSE	Mean Squared Error
SGD	Stochastic Gradient Descent

1 INTRODUCTION

The market for reselling used smartphones and tablets has increased significantly in the past few years. One of the problems in this second-hand market is that the used devices are in varying condition. Accurately estimating the condition of different components is difficult, but also crucial in deciding the resell value of the device. One of the most prominent features of a modern mobile device is the large touch screen in the front part of the device. Any damage on the screen will significantly impact the value of the device or even necessitate a screen replacement before reselling a used device is possible.

In this thesis we propose a method to automatically evaluate the condition of a mobile device screen. The advantages of an automated method over a visual inspection done by a human operator are the objectivity and reproducibility of the screen condition estimation. There are many different actors participating in the process of selling a used mobile device, such as insurance companies, trade-in and repair businesses, the person selling and the person buying the device. They all have different motives regarding the value of the device, and therefore a consistent and objective estimation is necessary.

The automatic classification of the damages will be done based on images taken of the screen. An image of a full mobile device screen is divided into cells, and the condition of each cell will be analysed by a classifier. The final condition of the screen can then be decided based on the number of damaged cells and the severity of that damage. In the recent years, convolutional neural networks have shown unparalleled performance in a variety of image classification tasks. Therefore, the focus of this thesis is to present a convolutional neural network capable of classifying these input image cells into five different categories based on the damages detected in each cell.

The practical implementation of an application utilizing this neural network to perform analysis of device screens is not in the scope of this work, but it sets some restrictions for the network. The application will be run on a smartphone, and the inference will be done locally. This means that the network must be small enough to fit in the memory and fast enough to run inference within a reasonable time on a modern smartphone. Even though the computational power of smartphones in increasing rapidly, the large amount of computation required to a run deep learning neural network may result in a delay that hurts user experience. Due to these restrictions this thesis will focus on some of the most lightweight, yet widely used network architectures available.

In this thesis, we propose a convolutional neural network to classify images of mobile de-

vice screen based on the visible damages. In Chapter 2 the details of the input images, class division and the restrictions set by the application of the classifier are described in detail. Next, Chapter 3 covers the standard theory behind deep learning and convolutional neural networks. Properties of three commonly used lightweight convolutional neural network architectures are introduced, focusing on the aspects that make them suitable for use in mobile devices. Chapter 4 describes the process and methods for training the convolutional neural networks and the metrics used to evaluate their performance. Finally, in Chapter 5 these networks are trained, and their performance and suitability for this task is assessed.

2 DEFINITION OF THE PROBLEM

In this chapter the problem of classifying images of mobile device screens is described in more detail. In Section 2.1 the characteristics of different damage types are defined to get as clear as possible separation between the five classes. The classification algorithm must work in many different environments, so the images will have a wide variation in quality, lighting and background. Properties regarding this variability will be covered in Section 2.2. In Section 2.3 the restrictions set by the application environment are covered.

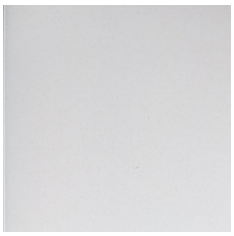
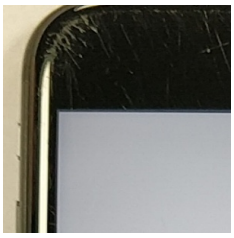
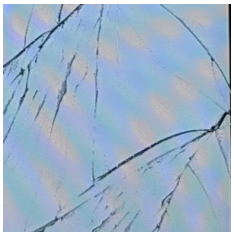
2.1 Damage types in mobile device screens

In the context of second-hand mobile device market, the damages on the screen can affect the both the functionality of the touch screen and the visual appearance of the device. This thesis is focused on detecting the visual damages. For the purposes of this thesis the visual damages will be divided into two distinct damage types, scratches and cracks.

The difference between scratches and cracks is mainly in the depth of the damage. Scratches are usually caused by some sharp object, such as a key, lightly scraping the surface of the screen. Cracks on the other hand are caused by a heavier impact, such as dropping the device on a hard surface. Cracks are deeper and more clearly visible than scratches. From the usability point of view, scratches are usually only a cosmetic problem, but having a badly cracked screen may prevent the use of the device, since cracks might block part of the content displayed on the screen and are more likely to break the touch functionality of the screen.

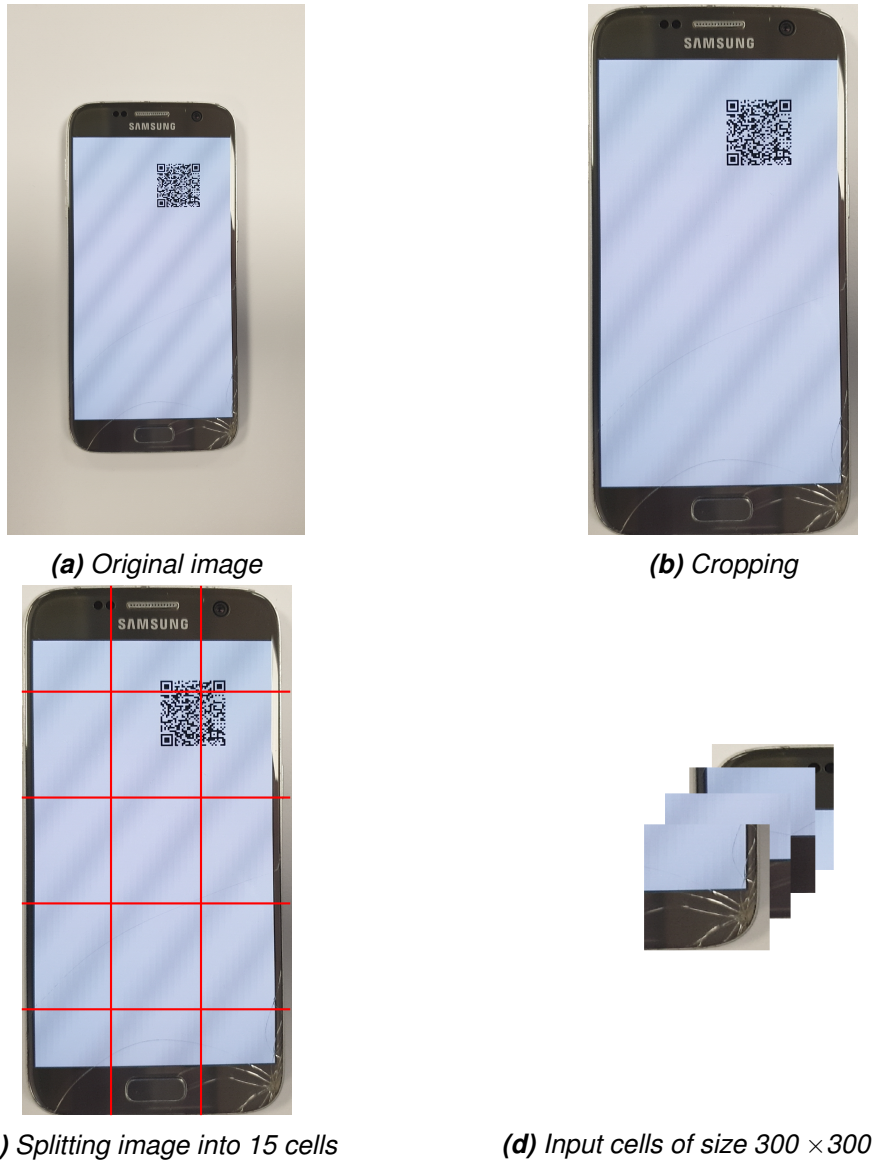
In addition to the depth of the damage, the size of the area impacted by the damage will affect the visual condition of the screen. To be precise enough to meet the needs of second-hand mobile device market, five target classes are defined. Images containing scratches are divided into two classes, based on the extent of the damage. Same is done for images with cracks. In addition to these four classes, one class is assigned for images without any detected damage. If a sample image contains both scratches and cracks, it will be classified as containing cracks, since it is the more prominent of the two damage types. The class indices, their textual descriptions and example images are presented in Table 2.1.

Table 2.1. *Description of damages for samples in each of the five classes.*

Class index	Description	Sample
0	No damage	
1	Minor scratches	
2	Major scratches	
3	Minor cracks	
4	Major cracks	

2.2 Properties of input images

The images are collected by photographing the device that will be analysed. The picture is taken directly above the analysed device with another smartphone that is a running an application developed for this specific use. The application is responsible for cropping the device from the picture and dividing the cropped picture into 15 cells. These 15 cells are resized to 300 $\times$ 300 pixels and will be the input images fed to the neural network. The path from the original raw image to neural network input cell is illustrated in figure 2.1.



(a) *Original image*

(b) *Cropping*

(c) *Splitting image into 15 cells*

(d) *Input cells of size 300 $\times$ 300 pixels*

Figure 2.1. *The inputs for the neural network are obtained by cropping the device from a picture and splitting the cropped image into 15 cells.*

The images of the devices are split into 15 cells instead of feeding the whole image into the classifier for two reasons. Firstly, this gives more flexibility for adjusting the final algorithm for calculating the grade of device screen condition. Different users can set different restrictions based on the classifications, for example requiring a screen replacement for devices with at least two cells containing major cracks. Also, since all damages may not

be visible in the images and the classifier might misclassify a cell, in some use cases it may make sense to allow users to make modifications on the classifications. The number of cells was chosen to be 15, since it is large enough to accurately capture the different areas of the screen, while still being quick and easy for human to verify and adjust the results if necessary. The 15 cells are organized in a 3 $\times$ 5 grid to approximately match the aspect ratio of common smart phones.

As seen from the Figure 2.1, all of the input images contain some part of the front of device body. Images will contain mostly the screen, but damages on the other parts of the body should also be detected. An even white colour is displayed on the screen, but there is also a QR code on the screen each analysed device. This code is used for device identification, and the content of the code is a practical implementation detail that is not relevant to this work. The neural network must be able to classify cells that may contain parts of this QR code. Outside of the screen, the other parts of the device body have greater variability. The colour of the body and the position, size and shape of the cameras, speakers and buttons vary between different device models.

The sample images were taken in varying conditions. Lighting of the setting, properties of the camera taking the picture and screen of the device photographed will cause different kinds of reflections and distortions on the input images. Examples of these effects can be seen in Figure 2.2.

Figure 2.2. *Common disruptions visible in the dataset. On the left a light source directly above the device is causing a big reflection on the screen. The distortions on the right-hand device are caused by the Moiré effect.*

A total of 61 devices were photographed to be used as training material for the neural network classifier. Approximately 4 images were taken of each device in different lighting and with different cameras. Each device had damage on some part of the screen, but

on many devices most of the screen are was intact. This led to imbalance in the training dataset, since most of the training samples belong to class 0. Imbalanced class distribution can negatively impact the classifier performance by over-classifying the majority class [1]. Methods for preventing the negative effects of imbalanced class distribution are discussed in Section 4.1.2.

2.3 Environment for inference

The classification will happen in the same mobile device that takes the pictures. Running a neural network in a mobile device sets some restrictions on the size of the network. In addition to classification accuracy, the speed of inference is the most important quality of the neural network. For each analysed device, the network must be able to classify 15 input cells within a reasonable time, in order to not have negative impact on the user experience. The target set in this work for the inference time in a mid-range smart phone is less than 3 seconds, but anything faster than that is considered an advantage until the classification causes no noticeable delay for the user. The file size of the network must also be small enough to fit in the memory of a smart phone, but given the file size of some of the commonly used convolutional neural networks and the capacities of modern smart phones, the speed of inference will be a much more restricting property.

To summarize, the goal of this thesis will be to build a convolutional neural network, that can classify images of damaged mobile device screens into five classes. The network must also be fast and small enough to run inference on a modern smart phone.

3 IMAGE CLASSIFICATION WITH DEEP LEARNING NEURAL NETWORKS

With the increasing power of modern GPUs, deep learning *artificial neural networks* (ANNs) have gained popularity in the field of machine learning. ANN is a machine learning algorithm loosely based on the way neurons work in an animal brain. They consist of a set of neurons, connected to each other to form a layered structure.

Convolutional neural networks (CNN) are a class of artificial neural networks commonly used in image classification and segmentation tasks. For image classification, the popularity and performance of deep learning convolutional neural networks have surpassed more traditional methods. Since 2012, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been won using CNNs. The ILSVRC is a popular benchmark for image classification using a huge dataset of millions of images from 1000 different categories [2].

Because of the success of CNNs and the nature of the problem at hand, the classifier designed in this thesis will also utilize a convolutional neural network. In this chapter the theory behind CNNs is covered. First, Section 3.1 covers the basic concepts related to artificial neural networks and how they learn. Section 3.2 focuses on properties of convolutional neural networks. In Section 3.3 three lightweight CNN architectures with potential for solving the problem described in Chapter 2 are presented. Finally, in Section 3.4 methods for further optimizing their performance are presented.

3.1 Introduction to deep learning neural networks

Artificial neural network is a collection of interconnected neurons, organized in a layered structure. The activation of a neuron is determined by its inputs and a bias value. Each input also has a weight value that determines how much that input will affect the activation of the neuron. Adapted from equations in [3], the activation a of a single neuron can be expressed with a formula

$$a = f(\mathbf{x} \cdot \mathbf{w} + b), \tag{3.1}$$

where $\mathbf{x}$ is the vector of inputs, $\mathbf{w}$ is the weight vector, b is the bias and f is an activation function. Structure of a neuron is illustrated in Figure 3.1. The activation function f is

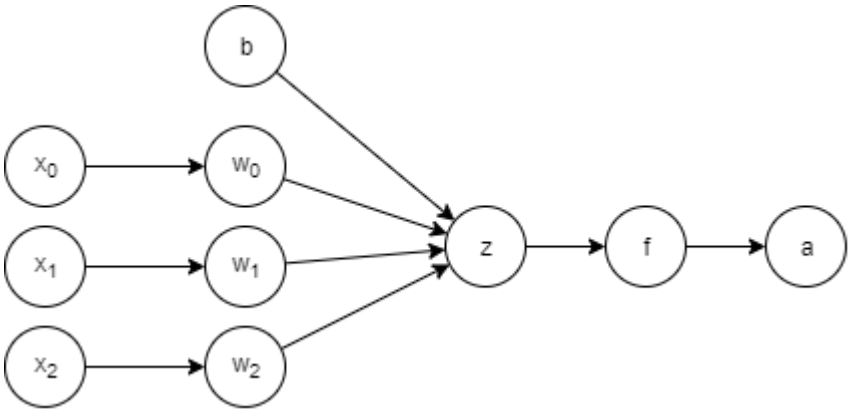


Figure 3.1. *Structure of a neuron with three inputs denoted x_0, x_1 and x_2. Node z is an auxiliary value denoting the activation of a neuron before it is passed through the activation function.*

usually some non-linear, differentiable function such as sigmoid or a rectified linear unit (ReLU). For many years the sigmoid was a widely used activation function, but ReLU has gained popularity since it has been demonstrated to yield better performance in the training of deep networks [4]. ReLU function is defined as

$$f(z) = \max(0, z). \tag{3.2}$$

The output layer of an ANN usually has different kind of activation function than the rest of the layers. In one-class classification tasks, where each input sample has exactly one correct class, it is convenient to have each output neuron present one of the output classes, and the activations of each neuron be the probability that the input belongs to the corresponding class. The classification is then made by choosing the class corresponding to the neuron with the highest activation. This behaviour can be achieved in an neural network by using *softmax* activation function for the final layer [5]. The softmax function is defined as

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{N-1} e^{z_j}}, \tag{3.3}$$

where z_i is the output of the i:th output neuron before passing through the activation function and N is the number of outputs.

The neurons in an ANN can be connected in many different ways. In the simplest neural network topologies, the outputs from one layer are the inputs for the neurons in the next layer. A network is said to be a feedforward network, if outputs from one layer of neurons are the only inputs to the neurons in the following layer. In a fully connected neural network, all neurons in one layer are connected to all neurons in the previous layer [3]. An example of a fully connected feedforward network is presented in Figure 3.2.

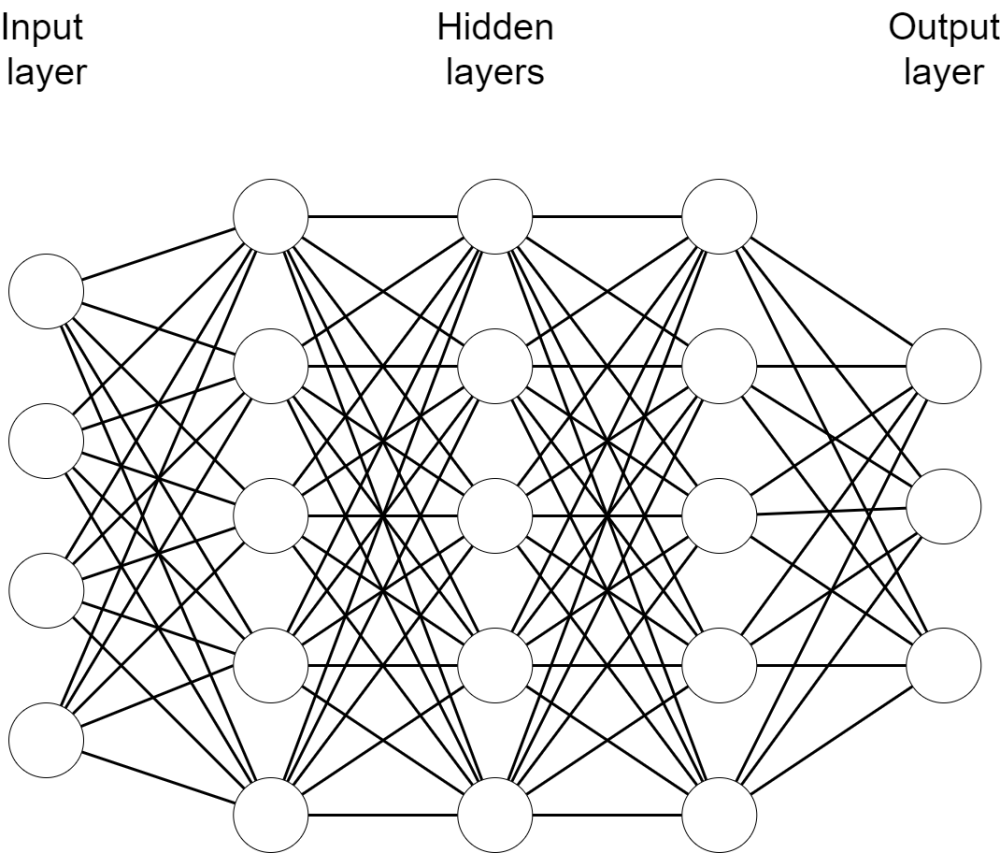


Figure 3.2. *Example of a fully connected feedforward neural network.*

Training a neural network

A neural network is trained by feeding training samples into the network. The training samples consist of a an input such as a vector of numbers or multidimensional array pixel values of an image, and their respective desired output, such as the correct class index for classification tasks. An untrained network will produce some output that is often wrong, but after enough training samples are presented to the network the network will have learned suitable parameters to produce correct outputs.

The learnable parameters in an artificial neural network are the weights and biases. Proper weights and biases are learned as training samples are fed into the network, and their values are adjusted in order to minimize the error between the desired output and the output of the network. Next, we describe the training process in greater detail by deriving equations for how weights and biases should be adjusted based on equations presented by [3] and [6].

Consider a fully connected feedforward neural network with L layers. Each layer has n_l neurons, where l is the index of the layer. For example, the network has n_1 input neurons and n_L output neurons. Activations of the neurons in layer l are denoted by vector $\mathbf{a}^l \in \mathbb{R}^{n_l}$. The neural network learns when multiple training samples $\mathbf{x} \in \mathbb{R}^{n_1}$ and their respective ground truth values $\mathbf{y} \in \mathbb{R}^{n_L}$ are fed into the network. Given input x_i, an untrained network will produce some output $\mathbf{a}_i^L \in \mathbb{R}_L^n$. The correctness of the output is

evaluated using a loss function $g : \mathbb{R}^{n_L} \times \mathbb{R}^{n_L} \mapsto \mathbb{R}$, for example the *mean squared error* (MSE) function

$$g(\mathbf{a}^L, \mathbf{y}) = \frac{1}{n_L} \sum_{j=1}^{n_L} (y_j - a_j^L)^2. \tag{3.4}$$

The training happens when the learnable parameters of the network, weights and biases, are updated in order to minimize the value of the loss function. To find out how each of the parameters in the network should be updated, their effect on the total cost must be computed. The total cost C over N training samples is defined as the mean of the loss function values of each sample with formula

$$C = \frac{1}{N} \sum_{i=1}^{N} g(\mathbf{a}_i^L, \mathbf{y}_i). \tag{3.5}$$

To simplify equations presented later, consider a case where there is only one training sample. In such case the total cost can be written as $C = g(\mathbf{a}^L, \mathbf{y})$. Additionally, an auxiliary value

$$z_k^l = \mathbf{x}_k^l \cdot \mathbf{w}_k^l + b_k^l \tag{3.6}$$

denoting the output of the neuron k in layer l before passing through the activation function is defined. Using z_k^l and setting the input vector $\mathbf{x}_k^l$ as the output of the previous layer, Equation 3.1 for the activation a_k^l of neuron k in layer l becomes

$$a_k^l = f(\mathbf{a}^{l-1} \cdot \mathbf{w}_k^l + b_k^l) = f(z_k^l). \tag{3.7}$$

For the last layer of the network the effect of the weights on the total cost function value can now be presented using the chain rule for partial derivatives as

$$\frac{\partial C}{\partial w_{kj}^L} = \frac{\partial z_k^L}{\partial w_{kj}^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial C}{\partial a_k^L}, \tag{3.8}$$

where w_{kj}^L is the weight of the connection between neuron j in layer $L-1$ and neuron k in layer L. With Equations 3.6, 3.7 and 3.5 the three partial derivatives in Equation 3.8 can be written simply as

$$\frac{\partial C}{\partial w_{kj}^L} = \mathbf{a}^{L-1} f'(z_k^L) \frac{\partial g}{\partial a_k^L}. \tag{3.9}$$

The effect of the bias can be calculated similarly. By using Equation 3.6 it can be shown that $\frac{\partial z_k^L}{\partial b_k^L} = 1$, so the equation for the effect of the bias b_k^L of neuron k in layer L becomes

$$\frac{\partial C}{\partial b_k^L} = \frac{\partial z_k^L}{\partial b_k^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial C}{\partial a_k^L} = f'(z_j^L) \frac{\partial g}{\partial a_k^L}. \tag{3.10}$$

For neurons in any other layer than the last layer the above equations are not as simple. The last term $\frac{\partial C}{\partial a_k^l}$ can only be easily computed using the loss function g for the last layer. Since the activation a_k^l affects the final output of the network through all the connections

the neuron has to the neurons a_j^{l+1} in the next layer, the derivative must be calculated using sum of the effects the activation has on the total cost through all its connections. In the general case the equation becomes

$$\frac{\partial C}{\partial a_k^l} = \sum_{j=1}^{n^{l+1}} \frac{\partial z_j^{l+1}}{\partial a_k^l} \frac{\partial a_j^{l+1}}{\partial z_j^{l+1}} \frac{\partial C}{\partial a_j^{l+1}} = \sum_{j=1}^{n^{l+1}} \mathbf{w}_k f'(z_j^l) \frac{\partial C}{\partial a_j^{l+1}}. \tag{3.11}$$

Since the value of 3.11 is only defined for the last layer, and to calculate it for any other layer l we need the its value for the layer $l+1$, the process of updating the weights and biases must be started from the last layer and then proceed backwards toward the start one layer at the time. This process of moving backwards through the net is called *backpropagation* [6].

With the partial derivatives of all the learnable parameters of the network, the gradient vector of the total cost ∇C can be constructed. The elements in the gradient vector indicate how each of the parameters should be adjusted in order to minimize the total cost. Let vector W be a vector containing all the weights and biases in a network. The weight vector can be updated using *gradient descent* by feeding all the training samples to the network, computing the gradient ∇C and moving the weights slightly in the direction of the negative gradient. This can be expressed with formula

$$W \leftarrow W - \lambda \nabla C, \tag{3.12}$$

where λ is a small real number called the *learning rate*. Choosing a small learning rate may result in very slow training but will yield a smoother gradient descent. A large learning rate may cause the gradient descent to overshoot a minima and prevent the training from converging [7].

Calculating the gradient with all the training data can be a slow process in real world applications. To solve this, the samples are usually fed into the network in randomly selected subsets called *batches*. The gradient is computed and the weights updated using the samples in this batch. This kind of approach is called *Stochastic gradient descent* (SGD). Smaller batch sizes will result in more stable learning and better generalization of the model [8]. In stochastic gradient descent the learning rate is constant for all parameters and stays the as the training progresses. More sophisticated approaches, such as the Adam optimizer, that use different learning rates for each of the parameters and change the rates during training, have been shown to yield better performance [9].

3.2 Convolutional neural networks

When dealing with image data, fully connected neural networks are often not ideal. An image could be flattened into a vector of pixel values and fed into traditional fully connected network described in Section 3.1 but this approach would result in a huge number of neurons, connections and weights in the network. Larger networks are generally more

prone to *overfitting*, meaning they capture really small details from the training data, but do not generalize well [3]. Also, a change in a single pixel value of an image does not usually change what the image represents. A collection of nearby pixels in the image is needed to form *features*, such as edges and shapes, which more effectively describe the contents of the image.

Convolutional neural networks [10] are a type of artificial neural networks, that work especially well on image data. CNNs capture low level features from the input image with an operation called *convolution*. These low-level features are then used to construct more complex features, again using convolution. This process allows the network to more accurately capture the invariant features from the pixels in the input image [11]. In addition to image data, convolutional neural networks have also been applied to other types of machine learning tasks, such as natural language processing [12].

Convolution

The advantage of convolution is that it processes a whole group of adjacent pixels at a time, instead of single pixel values. This is achieved by sliding a *kernel* over the input image and calculating the inner product of the pixel values under the kernel and the values of the kernel itself. Each inner product will produce a single pixel value to the output of the convolution operation. By applying the operation over the whole input, a matrix of these inner product values is produced. This matrix is called a *feature map*. Example of a convolution operation is presented in Figure 3.3. [13]

In image processing, the convolution is used for many tasks, such as blurring or edge detection. Different kernels will extract different kinds of features from the input image. The values for the kernels are the weights of a convolutional neural network, meaning that the kernels will learn to extract features relative specifically to the task at hand during training.

Convolutional layer

The inputs to convolutional neural network can be presented as a three-dimensional array of pixels. Width and height of the array correspond to the width and height of the image, and the depth is equal to the number of colour channels. Usually for colour images, the depth is 3, corresponding to red, green and blue channel, and for grayscale images 1. The kernel is also a three-dimensional array of real numbers. Width and height of kernels can be set as the hyperparameters for each layer, whereas the depth is determined by the depth of input. Applying a kernel of size $k \times k \times d$ on an image of size $w \times h \times d$ will produce a feature map of size $w - (k-1) \times h - (k-1) \times 1$. A *convolutional layer* applies multiple different kernels on the input image, each producing a new feature map. The output of a convolutional layer is an image, where each channel is a feature map produced by one of the kernels. Another convolutional layer can now be applied to this array, with kernel depth equal to the number of channels of the input, that in turn is equal

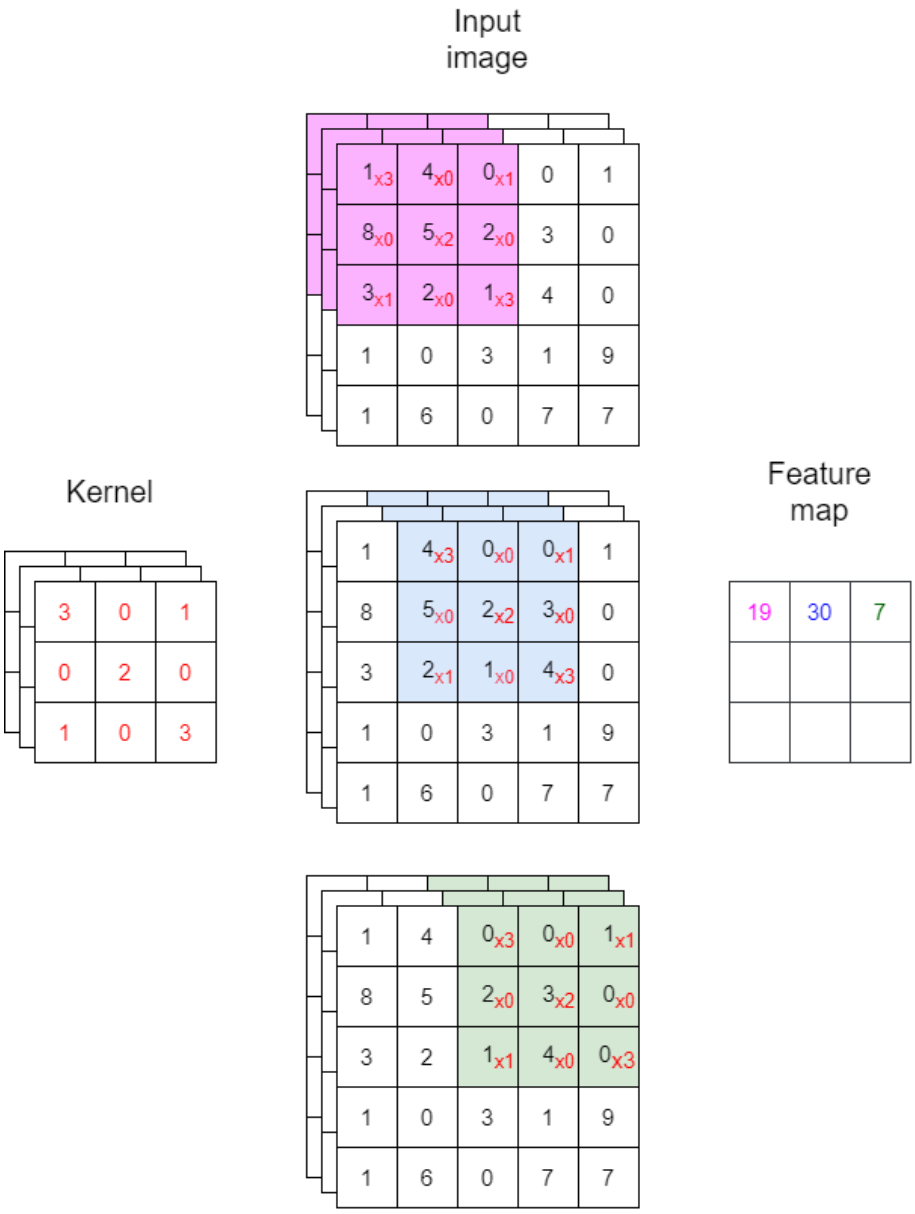


Figure 3.3. *Computation of the top row of a feature map produced by sliding a* $3 \times 3 \times 3$ *kernel over an* $5 \times 5 \times 3$ *input image without padding and stride 1. The kernel is slid over the input image, and the inner product of kernel and the section of input image under the kernel is calculated. The pink, blue and green colors of the numbers in the feature map represent the inner product computed when the kernel is in position denoted by that color over the input. Second and third rows of the feature maps are computed similarly, but sliding the kernel one step lower in the input for each row.*

to the numbers of kernels in the previous layer.

The kernel width and height are set as the parameters of each convolutional layer. The size of the kernel affects the size of the feature map it produces, making the feature map slightly smaller than the width and height of the input. For example, a kernel of size 5×5 applied on an input of size 100×100 will produce a feature map with width and height of 96. This is because there are 96 different positions both vertically and horizontally to fit a 5×5 kernel. Reducing the spatial dimensions of the input is often not desired behaviour. The size of the feature map can be adjusted by using *padding* and *stride*. As the same name suggests, *same padding* adds zeroes at the edges of the input so that the feature maps will have the same width and height as the input. For example, by increasing the size of the example image from 100×100 to 102×102 with padding, the feature map produced by the 5×5 kernel will have a size of 100×100. Stride on the other hand is the step size for the movement of the kernel. By increasing the stride, the kernel will not be applied to every possible location of the input image, hence reducing the width and height of the feature map it produces.

Pooling layer

In addition to the convolutional layer, the other key layer type in a convolutional neural network is the pooling layer. The purpose of a pooling layer is to reduce the spatial size of the feature maps produced by the convolutional layers. This reduces the computational power required to process the data, helps prevent overfitting and adds invariance to small rotational and spatial changes in the input [3]. Pooling also utilizes a similar sliding kernels defined by width, height, padding and stride as the convolutional layer. Although some research has been made on the advantages of trainable pooling layers [14][15], generally the kernels in pooling layers do not have trainable weights.

There are two commonly used types of pooling layers, *max pooling* and *average pooling*. The output max pooling operation picks the largest value under the kernel in each position. Average pooling calculates the average of all values under the kernel. Out of the two methods, max pooling has been a popular choice lately due to its better ability to reduce noise in the input. The outputs of different pooling methods are presented in Figure 3.4

CNN architectures

Typical simple convolutional neural networks for image classification consist of an input layer, subsequent blocks of convolutional and pooling layers, followed by few fully connected layers. The purpose of the convolutional part is to extract as relevant as possible features from the input images. This part of the network is sometimes called the *backbone* of a convolutional neural network. The output of the last convolutional layers is flattened into a one dimensional *feature vector*, and the fully connected layers at the end of the network perform classification based on this vector. The convolutional layers in the network increase the depth of the image, while pooling layers decrease the width and

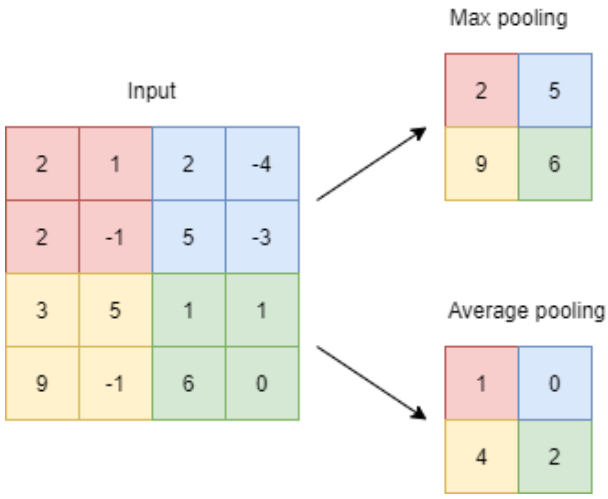


Figure 3.4. *Outputs of Max and Average pooling with kernel size* 2×2 *and stride 2.*

height. In a typical CNN this means that at the first layers the depth of the feature maps is lower and the spatial size large, and in the end the depth has increased, and the spatial size decreased [11]. Typical simple CNN architecture is described in Figure 3.5.

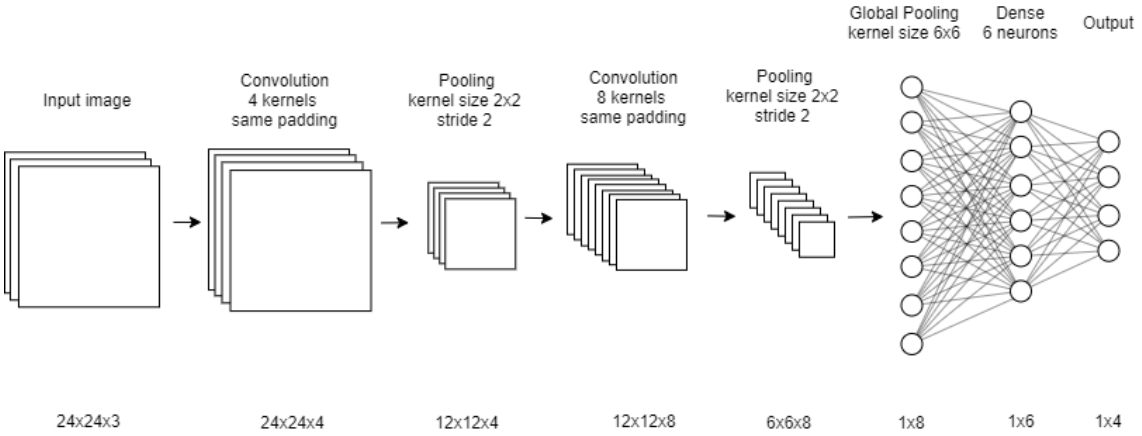


Figure 3.5. *CNN for classifying* 24×24 *color images into 4 categories Each convolutional layer increases the depth of the image, and each pooling layer reduces the width and height. The global pooling layer flattens the 3D image into a 1D vector by pooling each of the channels with a kernel size equal to the image width and height.*

Of course, not all CNNs follow the simple pattern described above. As research progresses, more complex network architectures are developed. Especially the ILSVRC competition has inspired many efficient yet accurate convolutional neural network architectures over the years. Some of these architectures are described in Section 3.3.

Trainable parameters in a CNN

The trainable parameters in a convolutional neural network are the values in the kernels. A kernel with width and height of k and a depth d has $k \times k \times d$ weights and one bias. Since

the same kernel is applied over the whole input, the number of parameters in the network do not increase as the spatial size of the input increases. This reduces the number of parameters in the network, which in turn helps prevent overfitting and decreases the training time.

The number of parameters in each convolutional layer is relative to the number of kernels, spatial size of the kernels and depth of the input and thus also the depth of the kernels. The number of parameters N in one convolutional layer can be written as formula

$$N = (k \times k \times d + 1) \times n_k, \tag{3.13}$$

where n_k is the number of kernels. There is one bias value for each of the kernels, so the volume of the kernel must be incremented by one to get the number of parameters in one kernel.

Regularization of a CNN

Overfitting is a common problem for neural networks and machine learning in general. Overfitting happens when the model learns irrelevant, small details of the training data so that it performs better on the training dataset but does not generalize well to other data [3]. To detect whether a model has been overfit, training dataset is usually split into training and validation sets. As the model is trained, loss values and possibly other metrics for both the training and validations sets are calculated after each *epoch*, meaning every time the whole training set has been fed into the model and the weights have been updated accordingly. Overfitting in terms of the loss function is illustrated in Figure 3.6.

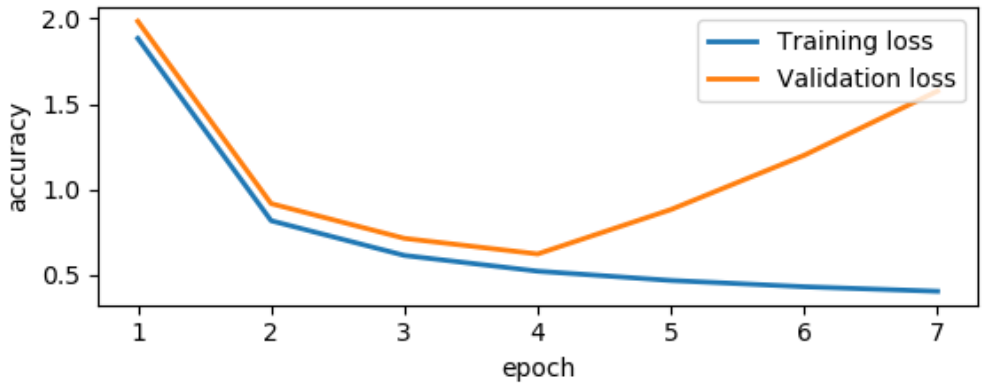


Figure 3.6. *Training and validation losses over a model trained for 8 epochs. In epochs 1 to 3, the model is underfitting, as the validation loss is still decreasing. The point of optimal fit is in epoch 4. After that the model validation loss starts increasing while the training loss keeps decreasing, which means that model is overfitting.*

There are many methods that can be used to prevent and reduce overfitting, such as

cross-validation, *early stopping* and *regularization* [16]. With cross-validation, the model is trained multiple times, using different subsets of the data for training and validation each time. Because the training times for convolutional neural networks are often long, cross-validation is not widely used with them. Early stopping means stopping the training before the model starts to overfit, that is the point of optimal fit illustrated in Figure 3.6.

Regularization can mean any method that aims to reduce overfitting while not increasing the training error [3]. Commonly used regularization methods for convolutional neural networks are *dropout* and *data augmentation*. Dropout layer randomly cuts a certain proportion of the connections between neurons in subsequent layers. Intuitively it might seem like it would reduce the performance of the network, but dropout has been shown to be an effective method to prevent overfitting without a significant effect on the performance [17].

Another popular regularization technique for CNNs that process image data is data augmentation. Augmentation means applying transformations or distortions to the training data to artificially increase the amount of training data available [18]. Augmentation can consist of for example rotating, flipping or changing the brightness of an image. It is important to make sure that the augmentation does not affect the classification of the image. For example, an image of a dog can be flipped horizontally and it will still represent a dog, but rotating a handwritten digit "6" by 180 degrees will change its meaning.

3.3 Lightweight convolutional neural network architectures

When choosing a suitable model for machine learning tasks, accuracy or other similar metrics of the model are not the only factors to consider. In some applications, it is important that the model can make predictions in reasonable time even in environments with limited resources, such as mobile devices or embedded systems. A lightweight CNN architecture aims for good accuracy while being as fast as possible and requiring as little memory as possible. The memory footprint of the network can be estimated by counting the number of parameters in the network, and speed as floating point operations (FLOPs) required to make a prediction [19]. It is worth noting that when talking about the speed of the network, only the time of inference is considered, not the time it takes to train the model, since the training is only done once and can often utilize a more powerful computer than the devices where inference is done.

In this section the accuracy of the models on the ImageNet dataset is often referenced. Even though images in that dataset are vastly different from the images in the problem of this thesis, using convolutional backbones of existing networks and retraining them to learn another classification task is a valid approach. It has been shown that architectures that do well on ImageNet tend to transfer well to image classification tasks as well [20]. The concept of using pre-trained model weights as a basis of learning a new classification task is called *transfer learning*.

3.3.1 MobileNet

One of these CNN architectures specifically designed for resource limited platforms, such as mobile devices, is the MobileNet. MobileNet architecture was published by a team of Google engineers in 2017 [21]. It was the first popular CNN architecture developed specifically for mobile devices with low computing power, and its success has inspired further research towards more lightweight convolutional neural networks. MobileNet achieves a 70.6% accuracy on the ImageNet dataset, which makes it a decent architecture even on complex image classification tasks. There many variants of the MobileNet architecture, with adjustable hyperparameters to trade off between accuracy and size and speed of the network.

The basic building block of the MobileNet is *depthwise separable convolution*. Depthwise separable convolution replaces the traditional convolution operation described in Section 3.2 with a depthwise convolution followed by a pointwise convolution. In the depthwise convolution, one kernel is applied on each of the input channels. This reduces the depth of the kernels into one, which in turn reduces the number of trainable parameters in the network. The computational cost of a depthwise convolution is

$$k \times k \times w \times h \times d, \tag{3.14}$$

where $k \times k$ is the size of the kernel, w, h and d are the width, height and depth of the input, respectively. This makes it efficient compared to the standard convolution, but since the kernels operate only on one channel at a time, depthwise convolution disregards information about the combinations of the channels. The role of the pointwise convolution is to combine the feature maps produced by the depthwise convolution using a 1×1 kernel with a depth equal to the number of channels in the input, with a computational cost of

$$w \times h \times d \times n_k. \tag{3.15}$$

Combining the costs of the depthwise and pointwise convolution and dividing the sum with cost of the standard convolution, it can be shown that the depthwise separable convolution has total cost of

$$\frac{k \times k \times w \times h \times d + w \times h \times d \times n_k}{k \times k \times w \times h \times d \times n_k} = \frac{1}{n_k} + \frac{1}{k^2} \tag{3.16}$$

times the cost of the standard convolution. In the case of the MobileNet, this means 8 to 9 times less computation without a significant loss in accuracy [21].

The convolutional part of the MobileNet consists of 28 layers, followed by one fully connected layer with 1024 neurons and an output layer with softmax activation. First layer in the network is a standard convolutional layer and the last layer of the convolutional part is an average pooling layer. Between the standard convolution and average pooling there are 13 pairs of depthwise and pointwise convolutional layers. Each depthwise and point-

Table 3.1. *Layers of the convolutional backbone of baseline MobileNet architecture. DW and PW convolution stand for depthwise and pointwise convolution operations, respectively.*

Layer type	Stride	Kernel Shape	No. kernels	Input shape
Convolution	2	$3 \times 3 \times 3$	32	$224 \times 224 \times 3$
DW Convolution	1	3×3	32	$112 \times 112 \times 32$
PW Convolution	1	$1 \times 1 \times 32$	64	$112 \times 112 \times 32$
DW Convolution	2	3×3	64	$112 \times 112 \times 64$
PW Convolution	1	$1 \times 1 \times 64$	128	$56 \times 56 \times 64$
DW Convolution	1	3×3	128	$56 \times 56 \times 128$
PW Convolution	1	$1 \times 1 \times 128$	128	$56 \times 56 \times 128$
DW Convolution	2	3×3	128	$56 \times 56 \times 128$
PW Convolution	1	$1 \times 1 \times 128$	256	$28 \times 28 \times 128$
DW Convolution	1	3×3	32	$28 \times 28 \times 256$
PW Convolution	1	$1 \times 1 \times 256$	64	$28 \times 28 \times 256$
DW Convolution	2	3×3	32	$28 \times 28 \times 256$
PW Convolution	1	$1 \times 1 \times 256$	64	$14 \times 14 \times 256$
$5\times$ DW Convolution	1	3×3	512	$14 \times 14 \times 512$
$5\times$ PW Convolution	1	$1 \times 1 \times 512$	512	$14 \times 14 \times 512$
DW Convolution	2	3×3	512	$14 \times 14 \times 512$
PW Convolution	1	$1 \times 1 \times 512$	1024	$7 \times 7 \times 1024$
DW Convolution	2	3×3	1024	$7 \times 7 \times 1024$
PW Convolution	1	$1 \times 1 \times 1024$	1024	$7 \times 7 \times 1024$

wise layer uses ReLU -activation and is followed by a batch normalization operation. It is worth noting that there are no other pooling layers in the network than the one right before the fully connected layer. Downsampling in MobileNet is achieved by setting the stride value to 2 for some of the depthwise convolutional layers. The full MobileNet architecture is described in table 3.1

There are two adjustable hyperparameters in MobileNet, the width multiplier α and the resolution multiplier ρ. These two parameters can make the network even smaller and faster than the baseline version. The width multiplier works by reducing the number of input and output channels in each of the depthwise separable convolution layers. The computational cost of the depthwise separable convolution with width multiplier α is

$$k \times k \times w \times h \times \alpha d + w \times h \times \alpha d \times \alpha n. \tag{3.17}$$

The width multiplier reduces the computational cost and the number of parameters in the network roughly by a factor of α^2. Typical choices for the width multiplier are 1, 0.75, 0.5 and 0.25, with $\alpha = 1$ representing the baseline MobileNet and the others being reduced versions of the network. The resolution multiplier on the other hand affects the width

and height of the input image, and the effect if subsequently carried on to the depthwise separable layers in the network, with a reduced cost of

$$k \times k \times \rho w \times \rho h \times d + \rho w \times \rho h \times \alpha d \times \alpha n. \tag{3.18}$$

Like the width multiplier, the reduced cost by the resolution multiplier is relative to the square of ρ. Typical input resolutions for MobileNet are 224, 192, 160, and 128. The value for ρ is implicitly determined from the input resolution.

3.3.2 MobileNetV2

In 2019, the team behind MobileNet introduced a new, improved version of the MobileNet architecture called the MobileNetV2. The MobileNetV2 utilizes the same depthwise separable convolution operation as the original MobileNet, but adds two new features, linear bottlenecks and inverted residual blocks. On the ImageNet classification challenge, the MobileNetV2 achieves 1.4% higher accuracy than the MobileNet, while reducing the number of parameters in the network by 14% and FLOPs by 30% [22].

When training deep neural networks using backpropagation, the gradient decreases as the partial derivatives are are chained from the back of the network towards the first layers. The gradient can become so small, that the parameters in the first layers of the network essentially don't learn anything or learn very slowly. This common problem in the field of artificial neural networks is called the *vanishing gradient problem*. One of the methods used to prevent the vanishing gradient are *residual block* [23]. Unlike in a feedforward network, where a layer feeds only to the next one, a residual block has a residual connection that skips few layers. These shortcut connections improve the flow the gradient during backpropagation. Diagram of a simple residual block is illustrated in Figure 3.7.

Commonly in convolutional neural networks, the layers connected by the residual connection are ones with high number of channels, and the layers in between are more shallow. In MobileNetV2, the residual connections are between shallow layers, with deeper layers in between. This opposite arrangement of layers in a block is what separates inverted residual blocks used in MobileNetV2 from the standard residual block. This approach leads to more memory efficient networks and slightly higher accuracy on the ImageNet dataset [22].

The varying channel depth is achieved by expanding thin bottleneck layers using pointwise convolution, performing depthwise convolution on these expanded layers and using pointwise convolution to bring them back to a thin bottleneck layer. The expansion from low to high channel depth is controlled by the *expansion factor*. In an inverted residual block with expansion factor t and stride s, input with shape $h \times w \times d$ shape is expanded to $h \times w \times (td)$ using pointwise convolution operation with t kernels. The larger the value of t, the more channels the expanded layers will contain. Depthwise convolution with

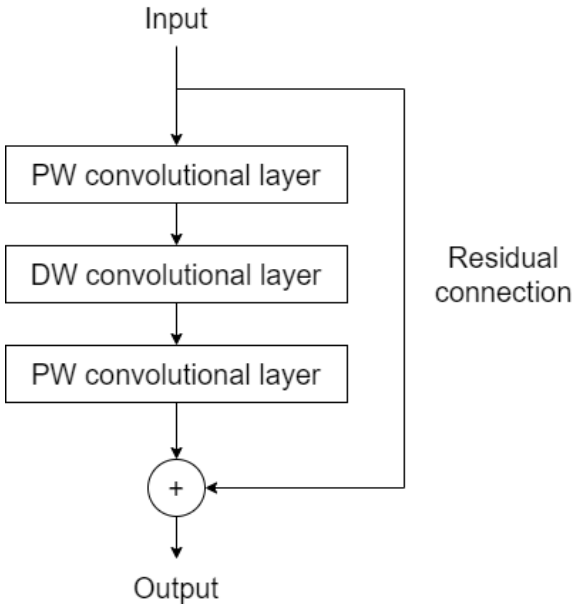


Figure 3.7. *A residual block with a residual connection skipping three convolutional layers. The output of the block is the sum of input and the output of the convolutional layers.*

Table 3.2. *Layers of the baseline MobileNetV2 architecture. IR block stands for inverted residual block. For each row containing multiple identical blocks the stride value corresponds to the stride of the first block in the sequence. Other blocks have stride 1.*

Layer type	Stride	Expansion factor	No output channels	Input shape
Convolution	2	-	32	$224 \times 224 \times 3$
IR Block	1	1	16	$112 \times 112 \times 32$
2 $\times$ IR Block	2	6	24	$112 \times 112 \times 16$
3 $\times$ IR Block	2	6	32	$56 \times 56 \times 24$
4 $\times$ IR Block	2	6	64	$28 \times 28 \times 32$
3 $\times$ IR Block	1	6	96	$14 \times 14 \times 64$
3 $\times$ IR Block	2	6	160	$14 \times 14 \times 96$
IR Block	1	6	320	$7 \times 7 \times 160$
PW Convolution	1	-	1280	$7 \times 7 \times 320$

stride is applied next, producing an image of shape $\frac{h}{s} \times \frac{w}{s} \times (td)$. Using stride value of 1 keeps the spatial size of feature maps the same, and higher stride values reduce them. Finally, pointwise convolution is applied again with d' kernels, resulting in an output of size $\frac{h}{s} \times \frac{w}{s} \times d'$. Value of d' is chosen so that $d' < td$, thus reducing the channel depth from the high value of expanded layers.

The inverted residual block is the basic building block of the MobileNetV2 architecture. There are 17 of these blocks in the network. The layers of the full MobileNetV2 architecture are described in Table 3.2.

Using non-linear activation functions is the key to building multi-layered artificial neural networks. With only linear activation functions, no matter how many layers is added to

neural network, the whole network could represented with a single layer. Adding non-linearity to the network through activation functions is essential to modeling more complex relationships in the input data. However, using non-linear activation function on the final pointwise convolution layers of the inverted residual blocks has been show to lead to loss of information and decreased model performance [22]. Linear bottlenecks in MobileNetV2 means using linear activation for these layers. For the other convolutional layers in the inverted residual block, a variant of ReLU activation function called ReLU6 is used. ReLU6 is defined as

$$f(z) = \text{min}(\text{max}(0, z), 6).$$

3.3.3 EfficientNet

Convolutional neural networks can be scaled up to achieve better accuracy by increasing the depth, width or the image resolution of the network [23][24][25]. Depth scaling corresponds to adding more layers to the network and width scaling to increasing the number of channels in each convolution layer. In 2019 Mingxing and Quoc proposed a new method for scaling CNNs that combines the three scaling methods mentioned above [26]. In the same paper Mingxing and Quoc also introduced a new family of CNN architectures called the *EfficientNets*. The smallest versions of the EfficientNet are suitable for use in environments with limited resources.

The *compound scaling* method scales the model in all three dimensions, depth, width and resolution with fixed scaling coefficients. By scaling all the three dimensions in the specific ratios, better performance was achieved than by arbitrarily scaling on or more dimensions of the model. Suitable scaling coefficients for each model can be found using a small grid search. Existing network architectures, such as MobileNet and ResNet [23] improved more in accuracy when scaled up with compound scaling than with other methods, while still having an equal number number of parameters and FLOPs [26]. In compound scaling, the depth d, width w and resolution r of a network are scaled with their respective scaling coefficients α, β and γ so that the scaled up dimensions of the network are

$$\begin{aligned} d &= \alpha^{\phi} \\ w &= \beta^{\phi} \\ r &= \gamma^{\phi}, \end{aligned} \tag{3.19}$$

where ϕ is a *compound coefficient*, $\alpha \geq 1$, $\beta \geq 1$, and $\gamma \geq 1$. The scaling coefficients are restricted with a equation

$$\alpha\beta^2\gamma^2 \approx 2, \tag{3.20}$$

Table 3.3. *Layers of the EfficientNet-B0 architecture. IR block stands for inverted residual block described in section 3.3.2. For each row containing multiple identical blocks the stride value corresponds to the stride of the first block in the sequence. Other blocks have stride 1.*

Layer type	Kernel size	Expansion factor	No. output channels	Input shape
Convolution	2	-	32	$224 \times 224 \times 3$
BR Block	1	1	16	$112 \times 112 \times 32$
2 $\times$ IR Block	2	6	24	$112 \times 112 \times 16$
2 $\times$ IR Block	2	6	40	$56 \times 56 \times 24$
3 $\times$ IR Block	2	6	80	$28 \times 28 \times 40$
3 $\times$ IR Block	1	6	112	$14 \times 14 \times 80$
4 $\times$ IR Block	2	6	192	$14 \times 14 \times 112$
BR Block	1	6	320	$7 \times 7 \times 192$
PW Convolution	1	-	1280	$7 \times 7 \times 320$

so that incrementing the compound coefficient by one will approximately double the number of FLOPs in the network.

The EfficientNet-B0 is the baseline network for all EfficientNets. The EfficientNet-B0 architecture was developed using neural architecture search optimizing for both accuracy and FLOPS [27]. Other versions, from EfficientNet-B1 to EfficientNet-B7, were obtained by searching the optimal scaling coefficients for EfficientNet-B0 and scaling up the EfficientNet-B0 with compound scaling, with the number in the end of the name corresponding to the compound coefficient used. For reference, the EfficientNet-B0 achieves 6.7% higher accuracy on ImageNet than the baseline MobileNet while requiring less FLOPs [26]. The EfficientNet models are built using the inverted residual blocks introduced by MobileNetV2, with the addition of squeeze and excitation optimization [28]. Layers of the baseline EfficientNet-B0 are presented in Table 3.3.

3.4 Model quantization and pruning

In addition to designing more efficient network architectures, there are other ways to improve neural network performance. In this section two these methods, *quantization* and *pruning*, are described.

Quantization

In general, quantization means mapping values from a large set to a smaller one, such as mapping real numbers to integers via rounding. For neural networks, the model parameters are often represented as 32-bit floating point numbers. By converting the parameters from 32-bit floating point numbers to for example integers, the MobileNet has been shown to achieve higher accuracy on the ImageNet dataset given the same inference time bud-

get [29]. Quantizing the parameters leads to smaller model size and faster inference, but often at the cost accuracy.

The methods for model quantization can be divided into two groups, *post-training quantization* and *quantization aware training* [30]. Post-training quantization can be applied to models after training and does not affect the training process at all, which makes it more easy to use than the latter method. In quantization aware training, the model is trained by emulating the quantization already at training time. Quantization aware training often yields better results, but is more complex to implement, as it does not yet have wide support in the common deep learning frameworks.

Post-training quantization can further be divided into subcategories based on the precision of the quantization output values. According Tensorflow deep-learning platform documentation [31], *Float16 quantization* converts model weights to 16-bit floating point numbers, which reduces the model size by 50% without a significant impact on model accuracy. Float16 quantization can also improve inference time, especially on GPUs. Integer quantization can be done converting either only the weights or both the weights and activations in the network to 8-bit integers. *Dynamic range quantization* converts only the weights, which reduces the model size by 75%, but often leads to loss in model accuracy. *Full integer quantization* utilizes a small set of representative data samples in the quantization process to convert both the weights and activations of the network. Full integer quantization can achieve 75% reduction in model size and up to 3 times faster inference.

Pruning

Pruning works by zeroing out the least important weights during the training process. This reduces the model size without a significant impact on model accuracy. Before training, a fixed target sparsity percentage and pruning schedule are set. During training, the weights with lowest magnitude are gradually set to zero, until the specified sparsity percentage is reached. The pruning happens at epochs dictated by the pruning schedule.

Pruning does not reduce the inference time, but pruned models can be effectively compressed by file compression algorithms to reduce their storage size. Model size can be reduced by up to 6 times with pruning with minimal effect on model accuracy [31].

4 TRAINING A CONVOLUTIONAL NEURAL NETWORK FOR SCREEN DAMAGE CLASSIFICATION

The best CNN model architecture and hyperparameter combination for a given image classification task can often be be found only through trial and error. Experiments with many configurations are run, and various metrics are examined to gain understanding on how the model works and what could be changed in order to gain better performance. Training CNNs is time consuming and the amount of adjustable parameters huge, so testing all possible combinations is not possible.

In this chapter the training of convolutional neural networks on data consisting of pictures of damaged mobile device screens is described. First, in Section 4.1 the detailed description of the dataset is presented. Starting from the raw unlabeled images of device screens, the process of creating training and validation datasets ready for training of a convolutional network is described. Next, Section 4.2 describes the metrics used for model performance evaluation. In Section 4.3 the training process is divided in three stages, where experiments with different sets of parameters and optimization methods are performed.

4.1 Dataset

To create the dataset for the task in this thesis 61 smartphones and tablets with various degrees of damage on the screen were photographed. Each device was photographed approximately four times with multiple different cameras and in varying lighting conditions to get more volume and variability in the dataset. Each image was then split into 15 cells as described in Figure 2.1. The resulting dataset consists of 3360 color images of size 300 by 300 pixels. Since the weights in the backbones are initialized to values obtained by training them on the ImageNet dataset, the input images were preprocessed using the same preprocessing function that was used with the ImageNet dataset. For all three backbones this means normalizing the pixel values from range $[0, 255]$ to range $[-1, 1]$ with function

$$\text{preprocess}(x) = \frac{(x - 127.5)}{127.5}.$$

The images were taken with the same application that will eventually host the final model. This is done to make sure that the images in the training set are as similar as possible to the images that the model will eventually classify in real world application. Simple preprocessing of training also makes it easy to repeat identical preprocessing step in the application the model is used in.

4.1.1 Annotation

The data annotation was done manually using a custom annotation tool. The tool displayed one full image of a device with a 3×5 grid overlaid on top, and a label from 0 to 4 was assigned to each of the 15 cells by clicking the tiles of the grid. The annotation process was repeated two times with a review meeting after the first round, because the class boundaries are not unambiguous, but rather a subjective assessment on the severity of damage. In the review meeting people working on different aspects on the final application utilizing this network gave input on what kinds of damage should be classified in which class.

The variations in image quality caused by differences in lighting and focus made it difficult to make consistent annotations. Especially the distinction between classes 0 (no damage) and 1 (minor scratches) was difficult, since the classifier should be sensitive to very small scratches, but smallest scratches were barely visible in some pictures. Dirt, reflections and distortions could also easily be confused as small scratch.

4.1.2 Class distribution

All devices photographed for the dataset had some damage in their screens. Often the damage only covered small part of the screen, so the majority of image cells were classified as having no damage. The distribution of image cells across the 5 classes is shown in Figure 4.1

There is a clear class imbalance in the data. Class 0 is the largest of the classes, while class 2 is clearly the smallest. This degree of class imbalance can have a negative impact on the performance of the classifier, but there are many methods to address this issue. For convolutional neural networks, *oversampling* has been shown to be an effective method to improve classifier performance without increasing overfitting [32]. In oversampling, samples from the minority classes are replicated in order to have equal number of samples in each class. To generate an oversampled dataset, training samples from classes 1 to 4 were copied until the number of samples in each class was approximately equal to the number of samples in class 0. Oversampling is not performed on the validation set to be able to compare results with models trained without oversampling. Distribution of the oversampled training set is shown in Figure 4.2.

Another widely used used method is setting *class weights*. By setting a higher weight on a specific class, errors made when classifying samples from that class are penalized more

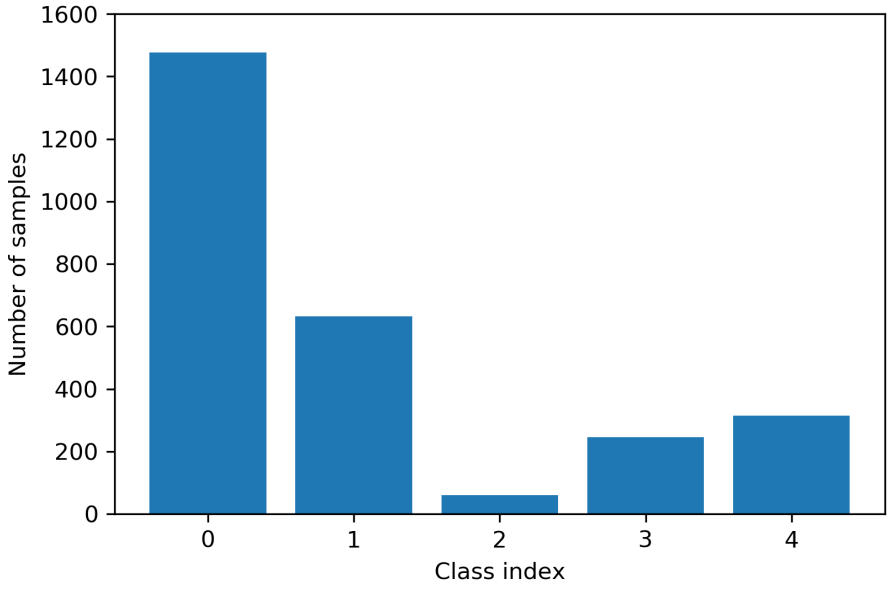


Figure 4.1. *Number of samples in the 5 classes.*

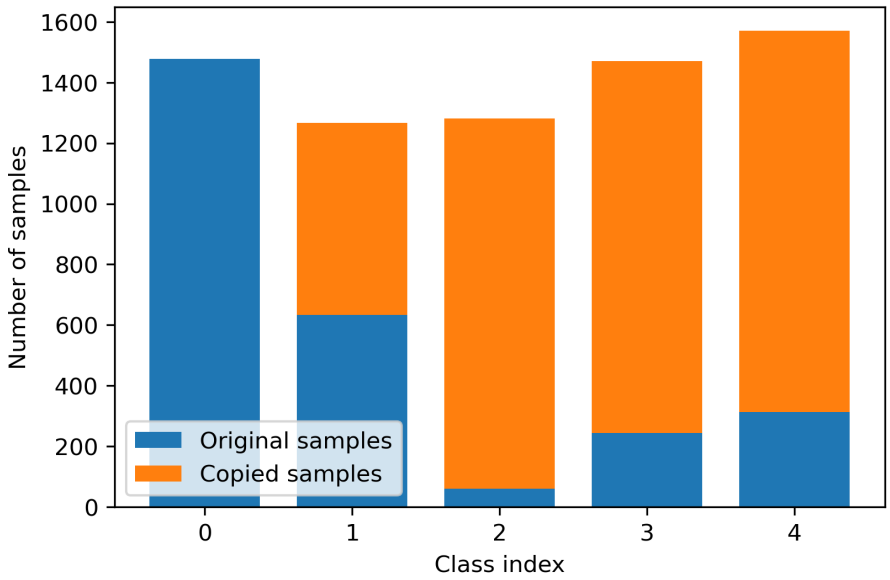


Figure 4.2. *Number of training samples in the 5 classes of the oversampled dataset. Blue color represents training samples in the original dataset, and orange color new samples generated by duplicating the original samples.*

during the training process. This makes the classifier more sensitive to samples from the minority classes. Class weights were set to be balanced, meaning the class weights are inversely proportional to the number of samples in that class. The class weights used in training are listed in Table 4.1.

The dataset was split into training and validation sets by choosing 12 of the original 60 devices and using the images from those devices only for validation. This resulted in approximately 20% of the data set to be used for validation. The training set consists of 2600 images and the validation set of 630 images. The class distribution of the validation

Table 4.1. *Balanced class weights for classes in the dataset. Weights are obtained by finding a values so that the product of class weight and number of samples in a class is constant for all classes.*

Class index	Class weight
0	0.370
1	0.863
2	8.951
3	2.229
4	1.739

set matches roughly the distribution of the whole dataset. The validation split was based on the devices rather than just individual images, since there are multiple images taken of each device. This way the validation set only consists of scratches and cracks that have not been seen by the network during training.

4.1.3 Augmentation

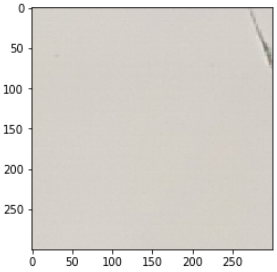
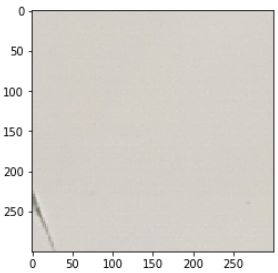
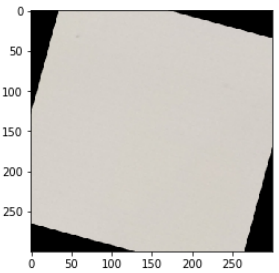
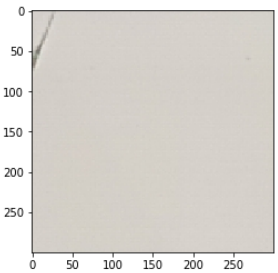
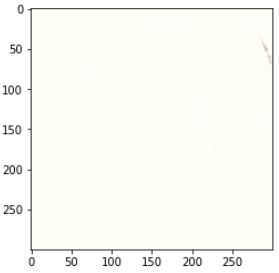
The amount of data available is relatively small for a CNN. There is a high risk of overfitting the model due to small number of training samples. Therefore more data is artificially generated by augmenting the images. When choosing suitable augmentations for this dataset, it is important to note that especially the class 1 images are sensitive to augmentation, meaning that augmentation methods that hide even a small part of the image may remove the only small scratch that makes the difference between classes 0 and 1. Also big changes in brightness or contrast of the image may make very thin scratches invisible. The images are however invariant to rotations and mirroring, which makes it easy to augment them by rotating in multiples of 90 degrees and horizontal flips, without any possibility that the class label might change due to augmentation. Examples of safe and unsafe augmentation methods are illustrated in Table 4.2.

Three levels of augmentation methods are tested in the training process. Lowest level is no augmentation at all. Light augmentation consists of only rotations of 0, 90, 180 or 270 degrees and horizontal flips. Heaviest augmentation level adds also other geometric transformations as well small color and brightness changes. To prevent heavy geometric transformations from hiding relevant features, the images are padded before applying the augmentations. Augmentation sequences and probabilities for each augmentation to be applied for each of the three augmentation levels are presented in Table 4.3. Augmentations are implemented using ImgAug image augmentation library [33].

4.1.4 Class encoding

The most common way to map categorical data into the target values for output neurons in neural network is the *one-hot encoding*. In one-hot encoding the numerical class labels

Table 4.2. *Examples of safe and unsafe augmentations. When the original image is rotated 15 degrees without padding, the damage in the top right corner of the image is lost. When brightness is increased too much, the damage becomes partially invisible.*

Original image	Safe augmentations	Unsafe augmentations
	 ***(a)*** *Rotated 180 degrees*	 ***(b)*** *Rotated 15 degrees*
	 ***(c)*** *Flipped horizontally*	 ***(d)*** *Increased brightness*

are converted into vectors $\mathbf{y} \in \mathbb{R}^n$ were n is the number of classes. The elements of a one-hot encoded vector y for a sample belonging to class j are defined as

$$y_i = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise.} \end{cases}$$

The class indices of the damage types increase as the severity of the damage increases. This ordinal nature of the classes could be utilized by using a class encoding method often used in ordinal regression [34]. In this ordinal mapping scheme the output for a sample in class j is encoded into vector $\mathbf{y} \in \mathbb{R}^{n-1}$ so that

$$y_i = \begin{cases} 1, & \text{if } i < j \\ 0, & \text{otherwise.} \end{cases}$$

It is worth noting that the number of output neurons in the network is different for both of these encoding methods. With one-hot encoding 5 output neurons are used, and with ordinal encoding only 4 are needed. Output vectors for each of the 5 classes encoded with both methods are presented in Table 4.4

Table 4.3. *Operations in the augmentation sequences for each of the three different levels of augmentation. In some of the operations in heavy augmentation, a random value is sampled from the mentioned interval, and operation is applied using that value. For example "Rotate between -5° and 5°" and probability 0.8 means 80% chance for an image to be rotated by a random value between -5 and 5 degrees.*

Augmentation level	Operation	Probability
No augmentation	None	1.0
Light augmentation	Rotate 90°	0.25
	Rotate 180°	0.25
	Rotate 270°	0.25
	Flip horizontally	0.5
Heavy augmentation	Rotate 90°	0.25
	Rotate 180°	0.25
	Rotate 270°	0.25
	Flip horizontally	0.5
	Rotate between -5° and 5°	0.8
	Translate between 0% and 5%	0.8
	Shear between -5° and 5°	0.8
	Add value between -30 and 30 to brightness	0.8
	Add value between -50 and 50 to hue	0.8

Table 4.4. *Encoded output vectors for each of the output classes in one-hot and ordinal encoding.*

Class index	One-hot encoding	Ordinal encoding
0	$[1,0,0,0,0]$	$[0,0,0,0]$
1	$[0,1,0,0,0]$	$[1,0,0,0]$
2	$[0,0,1,0,0]$	$[1,1,0,0]$
3	$[0,0,0,1,0]$	$[1,1,1,0]$
4	$[0,0,0,0,1]$	$[1,1,1,1]$

Conversion from the neural network output back to class indices is straightforward when using one-hot encoding. The index of the highest element in the output vector will be selected as the class index. With ordinal encoding, converting the output of the network to class indices is not as simple. Values of the output vector are scanned from start to end, until a value smaller than a predefined threshold value is found. The class index will be the number of elements with value larger than the threshold found using this method [34]. For example, if the threshold is set as 0.5 an output vector [0.8, 0.9, 0.2, 0.6] will be converted to class index 2.

The loss functions used with the different encoding methods are also slightly different. *Cross-Entropy Loss* is used with both methods. Cross-Entropy loss CE with ground truth

vector y and neural network output a^L is defined as

$$CE = -\sum_{i=0}^{n^L-1} y_i log(a_i^L), \tag{4.1}$$

where n^L is the number of classes and output neurons in the network. For one-hot encoded data, the only non-zero value of the target vector y is the index of the correct class. Assuming j as the index of the correct class, a special case of cross-entropy loss called the *categorical cross-entropy loss* can be written simply as

$$CE = -log(a_j^L). \tag{4.2}$$

For ordinal encoding softmax activation cannot be used, since there are multiple non-zero values in the ground truth vectors. Instead a version of cross-entropy loss commonly used in multi-label classification, called *binary cross-entropy loss* is used. Binary cross-entropy loss treats each output neuron as a two class classification problem, and computes the mean cross-entropy loss over all the output neurons. Binary cross-entropy is defined as

$$CE = -\frac{1}{n^L}\sum_{i=0}^{n^L} c_i, \tag{4.3}$$

where

$$c_i = \begin{cases} -log(a_i^L), & \text{if } y_i = 1 \\ -log(1 - a_i^L), & \text{otherwise.} \end{cases} \tag{4.4}$$

4.2 Evaluation

The most important metrics on which the models performance are evaluated are accuracy and speed of inference. Since the dataset contains a lot of samples where in the annotation phase it is difficult to decide between two neighbouring classes, an auxiliary metric called *off-by-one accuracy* is also defined. Size of the model exported model files is also recorded to make sure they fit within the memory size requirements of the application.

Accuracy is both the simplest and most important metric in this task. It is defined simply as the percentage of correctly classified samples. The accuracy of a model is calculated as the mean accuracy score for the ten best validation accuracy scores during the training epochs.

Speed of inference is measured by running inference on 100 samples and calculating the mean inference time for one sample. Inference is run within a smart phone application, to get as relevant as possible results compared to the real world use case. All models will

be tested on the same device, a modern mid-range smartphone, to eliminate the effect of hardware performance. Before the timed 100 inference runs 10 warm-up inferences are run, since the first runs may include some unrelated processing.

Off-by-one accuracy

Off-by-one accuracy measures the percentage of samples classified correctly or as belonging to one of the classes one step away from the correct one. Off-by-one accuracy A_{obo} is defined as

$$A_{obo} = \frac{1}{N} \sum_{i=0}^{N-1} c_i,$$

where N is the number samples in the validation set and

$$c_i = \begin{cases} 1, & \text{if } \textit{truth}(s_i) - 1 \leq \textit{prediction}(s_i) \leq \textit{truth}(s_i) + 1 \\ 0, & \text{otherwise,} \end{cases} \tag{4.5}$$

where *truth*(s_i) and *prediction*(s_i) are the indices of true and predicted classes for sample s_i.

The motivation for measuring the one-off accuracy is the fact that classification errors are not as severe if they are only one step away from the correct class. If a sample is classified as having no damage, when in truth the screen has major cracks, this misclassification has large impact to the overall estimate on the device condition. On the other hand, the classes are defined only as subjective estimates on the severity of the damage. Therefore there is no clear boundary between neighbouring classes. This does not mean that all errors that are one step off the correct class are acceptable, but combined with accuracy the one-off accuracy gives a more comprehensive understanding on the models performance.

Precision, recall and F1-score

Precision and recall are commonly used metrics originating from the context of information retrieval [35]. In multi-class classification task, precision and recall calculated individually for each of the classes. Precision for a class is defined as the number of correctly classified samples belonging to that class divided by the total number of samples classified to that class. Recall is similar metric, defined as the number of correctly classified samples divided by the total number of samples in that class. [36]

Precision and recall are relevant metrics as such, but they are often combined to a single metric, F1-score. F1-score is defined as the harmonic mean of precision and recall with

formula

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{4.6}$$

Like precision and recall, F1-score is calculated independently for all the classes. In order to compose a single value to describe the F1-score across all the classes, macro-average of F1-score can be computed. Macro-average is simply defined as the arithmetic mean of the F1-scores for all classes. Macro-averaging gives equal value on performance on all classes, unlike its counterpart micro-average, which favors classes with larger number of samples [36]. Therefore macro-averaged F1-score will be used when evaluating the model performance on minority classes.

4.3 Training

Training of the networks was performed using Tensorflow machine learning platform and high level Keras API [37]. Network structure, hyperparameters and training metrics were recorded for each training run. Since there are a lot of adjustable hyperparameters and optimization methods in a convolutional neural network, it is impossible to extensively experiment with all the possible combinations.

The training process consists of two training stages and a post-training optimization stage. In the first stage, models with three different backbones with varying model scaling hyperparameters are trained. The most promising models are chosen for further tests in second stage. In the second stage, several optimization methods, such as data augmentation, non-standard class encoding and class imbalance addressing methods are applied one at a time and their effect is evaluated. After the second training stage, post-training optimization are applied to the trained models. Diagram of the training process is presented in Figure 4.3.

Fixed parameters for all stages

All the tested models converged in 30 epochs, so that was set as the length for every training run. Most of the models started overfitting before this point, so after each epoch the model was saved if it achieved higher scored on validation metrics than the models from previous epochs. This way the best version of the model is always available, even if the performance starts degrading when training is continued.

The configuration of the layers on top of the pre-trained convolutional backbones stayed the same during the training process. The configuration was designed to be as simple as possible, as in preliminary tests adding more layers on was not shown to increase accuracy. Features from the last convolutional layers of the backbones are flattened into feature vector using global average pooling, and a dropout layer is added to reduce overfitting. The output layer of the networks consist of 5 neurons with softmax activation

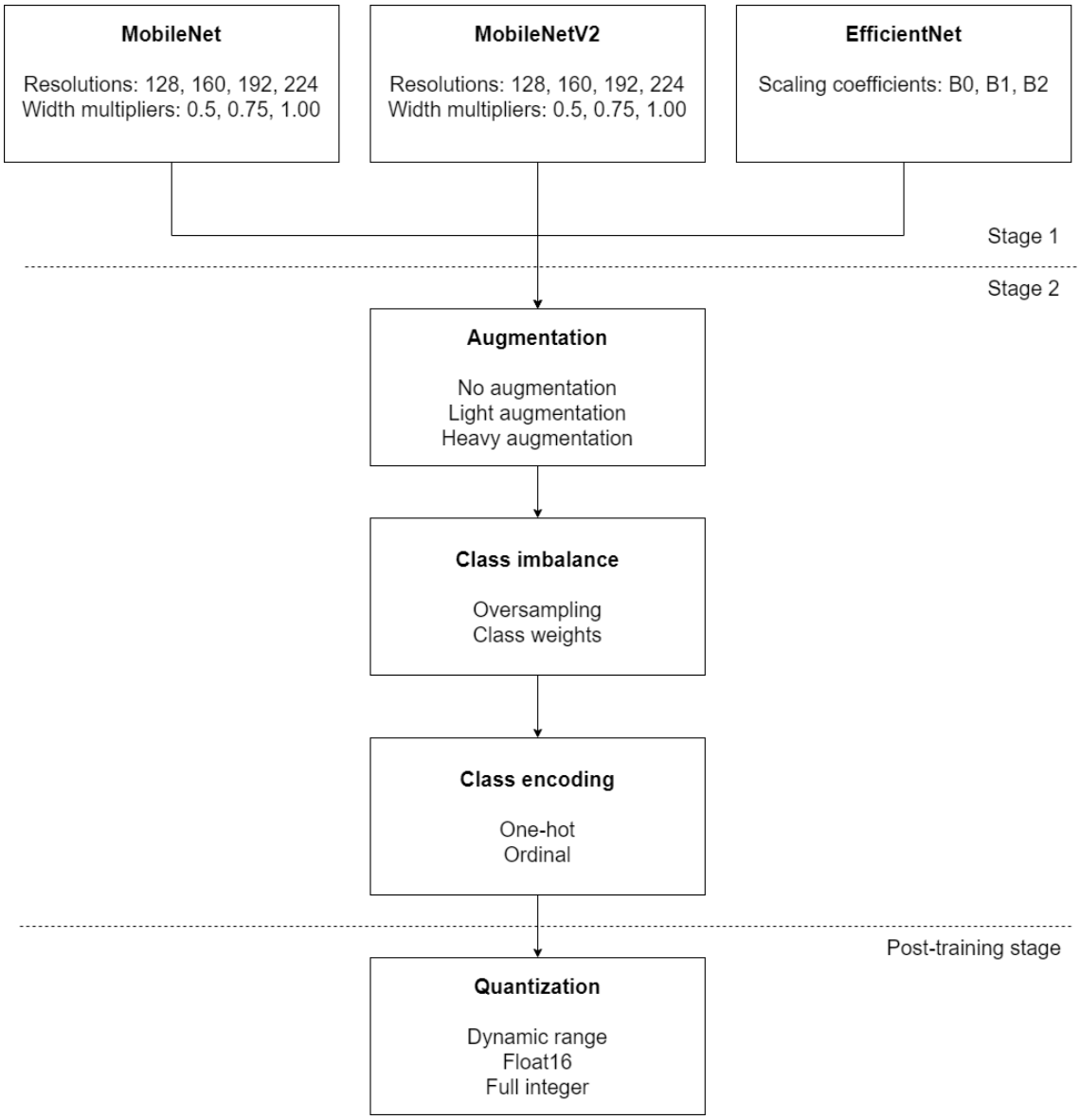


Figure 4.3. *Stages of the training process. In the first stage, models with three different convolutional backbones are trained with varying model scaling hyperparameters. In the second stage, different levels of data augmentation, methods for addressing class imbalance and class encoding schemes are experimented with. In the post-training stage models are optimized using three different quantization methods.*

function, except with ordinal class encoding 4 output neurons and with sigmoid activation was used. The layers added on top of the convolutional backbones described in Tables 3.1, 3.2 and 3.3 are listed in Table 4.5.

The batch size was also fixed for each of the backbones. The limiting factor for the batch size was the amount of dedicated GPU memory available. The batch size was set to be as high as possible while still being able to fit the model into GPU memory. For MobileNet and MobileNetV2 models the batch size was set to 32 and for EfficientNet batch size was 8.

Table 4.5. *The layers added on top of the convolutional backbones. Layers are defined using the layer constructors from tensorlfow.keras.layers package [38]. Layers are added in the order of increasing index value.*

Index	Layer
1	GlobalAveragePooling()
2	Dropout(0.5)
3	Dense(5, activation='softmax')

First training stage

In the first stage of training the backbones were trained with different model specific scaling hyperparameters. MobileNet and MobileNetV2 backbones were tested with input resolutions 128, 160, 192 and 224, and width multipliers 0.5, 0.75 and 1.0. For EfficientNet the versions with the smallest compound scaling coefficients are the ones that could be suitable for use in mobile applications. Compound scaling coefficients from 0 to 2 were chosen, corresponding to backbones EfficientNet-B0, EfficientNet-B1 and EfficientNet-B2.

In the first stage no data augmentation was used and class indices were encoded using one-hot encoding. No oversampling or class weights were used during training, and the models were not optimized with quantization or pruning. The accuracies and inference times on mobile device were recorded, and models with the best performance based on those metrics were selected for second training stage.

Second training stage

The best performing models from the first stage were chosen for further testing. First, models were trained with the three levels augmentation described in Table 4.3. Augmentation was chosen as the first method in the second stage, as it was assumed to have the largest impact on the network accuracy.

Once the optimal augmentation level was discovered, different methods for addressing class imbalance were tested. Models were trained again with oversampling and class weights. The oversampled training dataset and class weights are described in more detail in Section 4.1.2. The last training runs were performed using ordinal class encoding. Training runs with the oversampled dataset, class weights and ordinal class encoding were performed using the best augmentation level based on the previous tests.

Methods in this stage do not affect the models inference time or size, so they were not measured. Instead, more detailed metrics on the classification performance were used. In addition to accuracy, off-by-one accuracy was used to assess the effects of ordinal encoding. Macro-average of F1-score was used to evaluate the effects of class imbalance and its prevention methods.

Post-training stage

After all the training runs were completed, models were further optimized using three different post-training quantization methods, dynamic range quantization, float16 quantization and full integer quantization. Quantization aware training was not performed due to its complicated implementation. Speed of inference is more important factor than the model size, so pruning was not used.

In the post-training stage, the saved model checkpoint from the epoch with highest validation accuracy was used. The model was converted to TensorFlow lite format with and without quantization, and deployed to a mobile device. Inference time and accuracy was measured on the device for each of the quantization methods, and compared to the non-quantized version. Model sizes were also recorded for the quantized models.

5 COMPARISON OF NETWORK PERFORMANCE

In this chapter the performance of convolutional neural networks utilizing backbones described in Section 3.3 is examined. Backbones are compared to one another and the effect of several enhancement methods for each model are evaluated. First, in Section 5.1, results from training each backbone with different model scaling parameters are presented. In Section 5.2 the effect of data augmentation, class encoding, class weight, oversampling and model quantization is evaluated on the best performing models. Finally, in Section 5.3 the advantages and disadvantages of different backbones and optimization methods are discussed and their suitability for the mobile device screen damage classification task is assessed.

The most important metrics on which the models are evaluated in this chapter are the accuracy and inference time. Accuracy is measured on the validation set. The validation accuracy is relatively volatile during training due to small dataset size. To more accurately compare model performance, the mean of accuracy values over the ten epochs with highest values is calculated. The training training and validation accuracies and losses over all 30 training epochs are also recorded and they are used to assess the overfitting tendencies of the models using learning curves. Even when the accuracy scores of the model are computed as the average over ten epochs, there will be some randomness in the results because the models are trained only once with each configuration. This affects the analyzation of the results, as small improvements or deteriorations in metrics may be caused by random noise.

Inference times are measured by converting the models to TensorFlow Lite format and deploying them on an android device, where the models are benchmarked. The device chosen for testing in mobile environment is OnePlus 5. The CPU in that device is a Qualcomm Snapdragon 835 with eight cores and 2.45GHz clock rate [39]. Inference times mentioned in this chapter refer to the time it takes to classify one input cell. There are 15 cells on each device, so in order to get the total time it takes to analyse one device the inference times must be multiplied by 15.

5.1 Backbone comparison

In this section the results obtained from the first training stage described in Section 4.3 are presented. That includes accuracies and inference times from models trained with MobileNet, MobileNetV2 and EfficientNet backbones and varying model scaling hyperpa-

rameters.

MobileNet

Twelve models with a MobileNet backbone and different combinations of width and resolution multipliers were created. The most lightweight of these models is denoted *MobileNet-0.5-128*, corresponding to a model with MobileNet backbone, width multiplier of 0.5 and input resolution of 128. Using the same notation, the largest of these models is the *MobileNet-1.0-224*.

The mean accuracies for all MobileNet variants are presented in Table 5.1. As expected, the larger versions of the model have higher accuracy than the more lightweight ones. The most lightweight variant, MobileNet-0.5-128 achieves a 6.3% lower accuracy score compared to the full MobileNet-1.0-224. In particular, increasing the input resolution has significant impact on the accuracy. With constant width multiplier values, the increase in accuracy between input resolutions 128 and 224 is on average 4.9%. On the other hand increasing the width multiplier while keeping the input resolution constant only adds an average of 2.1% to the accuracy score.

Table 5.1. *Accuracy scores for the MobileNet variants with different input resolutions and width multipliers.*

		Resolution			
		128	160	190	224
α	0.50	70.9	73.2	72.9	76.2
	0.75	71.4	74.1	74.1	76.7
	1.00	73.2	74.3	76.7	77.3

The inference times for the same MobileNet variants are presented in Table 5.2. Significant improvements in the inference speed can be achieved by using a variant with a lower accuracy score. The MobileNet-0.5-128 is approximately ten times faster than the full MobileNet. Significant performance gains can also be obtained by reducing only one of the scaling parameters. As shown in Table 5.1, reducing the width multiplier from 1.0 to 0.5 with input resolution 224 only decreases the classification accuracy by 1.1%, while reducing the inference time to less than one third of the original. To get comparable inference time improvements by reducing only the input resolution, one would have to use the MobileNet-1.0-128, which in turn would reduce the model accuracy by 4.1%.

The trade-off between accuracy and inference time means that when choosing the most suitable model for the final application, the impact of misclassifications and slow analyzation times on the user experience must be taken into account. Therefore an objectively best MobileNet variant for this task cannot be named. Three of the MobileNet models are chosen for further experiments, based on their good values for accuracy and inference time. The models are MobileNet-0.5-160, MobileNet-0.5-224 and MobileNet-1.0-224. In addition to the MobileNet-1.0-224, which achieves the highest accuracy, MobileNet-0.5-

Table 5.2. *Inference times in milliseconds for the MobileNet variants with different input resolutions and width multipliers.*

		Resolution			
		128	160	190	224
α	0.50	12.1	18.5	25.8	34.9
	0.75	23.6	36.6	50.6	68.9
	1.00	39.0	60..9	83.5	112.4

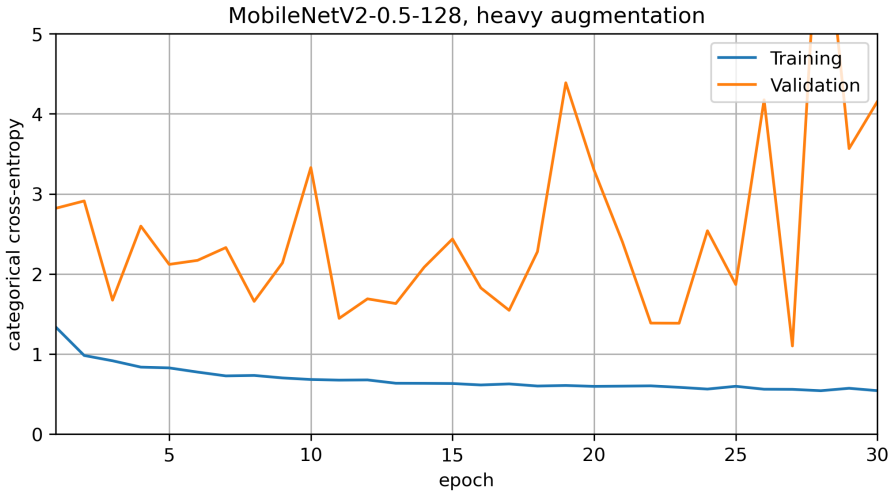


Figure 5.1. *Training and validation losses of MobileNetV2-0.5-128 with heavy augmentation using categorical cross-entropy loss function.*

224 is chosen based on the good balance between accuracy and speed. MobileNet-0.5-160 is almost as fast as the most lightweight variant, but achieves a decent accuracy score of 73.2%, and is a good option when very fast inference is essential.

MobileNetV2

For MobileNetV2 models, none of the variants achieved decent accuracies compared to the other backbone architectures. The accuracies on the validation set were between 60 and 65 percent, which is significantly worse than the accuracies of MobileNet models. The MobileNetV2 models seemed to suffer from heavy overfitting. Even the simplest model, MobileNetV2-0.5-128, with several overfitting prevention methods such as dropout layers and data augmentation started overfitting already after few epochs. The overfitting of MobileNetV2-0.5-128 can be observed from the training and validation loss graphs from Figure 5.1. All the other variants of MobileNetV2 produced similar effect.

The inference times of MobileNetV2 were slightly faster than MobileNet models, but due to the poor accuracy none of the MobileNetV2 variants were chosen for further experiments. Reasons for the poor performance of MobileNetV2 are not clear, as the architecture is very similar to MobileNet and EfficientNet.

EfficientNet

Three models with the most lightweight of the EfficientNet backbones were trained on the dataset. The mean accuracy scores obtained by the three variants were within 0.3% of each other. This implies that even the EfficientNet-B0 is complex enough to perform well on this relatively small dataset, and scaling up the model does not increase accuracy. Inference time on the scaled up versions is significantly slower than on EfficientNet-B0, making it the most suitable version of the EfficientNet for this task. It is the only EfficientNet variant chosen for further experiments.

Table 5.3. *Mean accuracies and inference times achieved by the EfficientNet-B0, EfficientNet-B1 and EfficientNet-B2 models.*

Model	Accuracy (%)	Inference time (ms)
EfficientNet-B0	81.3	204.7
EfficientNet-B1	81.3	347.6
EfficientNet-B2	81.0	470.7

Compared to the MobileNet-1.0-224, the EfficientNet-B0 achieves a 4% higher accuracy with approximately two times slower inference. The total inference time for 15 image cells with EfficientNet-B0 could be reduced to less than 3 seconds with optimizations, which makes it a plausible choice for this task.

It is worth noting that the number FLOPs in EfficientNet-B0 is actually lower than in MobileNet-1.0-224 [26][21]. Still the EfficientNet-B0 has a significantly slower inference time on a mobile device. This could be due to details in the implementation of the models, that allow for more optimization for MobileNet models when converting to TensorFlow Lite format.

5.2 Optimization methods

In this section several methods are applied to the best performing models from the first training stage to further improve their performance. The models chosen for these further tests are MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0.

Augmentation

The mean accuracies with different augmentation levels are presented in Table 5.4. As seen from the accuracies, adding light augmentation increases validation accuracy for all the models by 2-4% compared to using no augmentation at all. Further increasing augmentation gives a smaller improvement on some of the models, but perhaps counterintuitively the more larger models do not improve as much with heavier augmentation. Since augmentation increases the complexity of the training data, it would be expected for the larger models to benefit more from it.

Table 5.4. *Mean accuracies achieved by the MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with different levels of augmentation.*

	Accuracy (%)		
	No augmentation	Light augmentation	Heavy augmentation
MobileNet-0.5-160	73.2	76.6	77.7
MobileNet-0.5-224	76.2	79.3	80.6
MobileNet-1.0-224	77.3	80.9	80.7
EfficientNet-B0	81.3	83.3	83.4

The learning curves in Tables 5.6 and 5.5 visualize how augmentation improves model performance. The graphs present the training and validation accuracies and losses for each in epoch in the training of MobileNet-0.5-160 model.

From the loss graphs in Table 5.5 we can observe that without augmentation, the MobileNet-0.5-160 model starts overfitting already after 5 epochs. As more augmentation is added to the dataset, the model can be trained longer before overfitting starts. With heavy augmentation, small overfitting can be observed only after around 13 epochs.

The accuracy graphs in Table 5.6 show how increasing augmentation reduces model accuracy on the training set, but the gap between training and validation accuracies gets smaller, which translates to higher validation accuracy. Since augmentation improves model accuracy without affecting the inference time, augmentation should be applied to the training dataset when training final model. All training latter training runs in this chapter also include heavy augmentation.

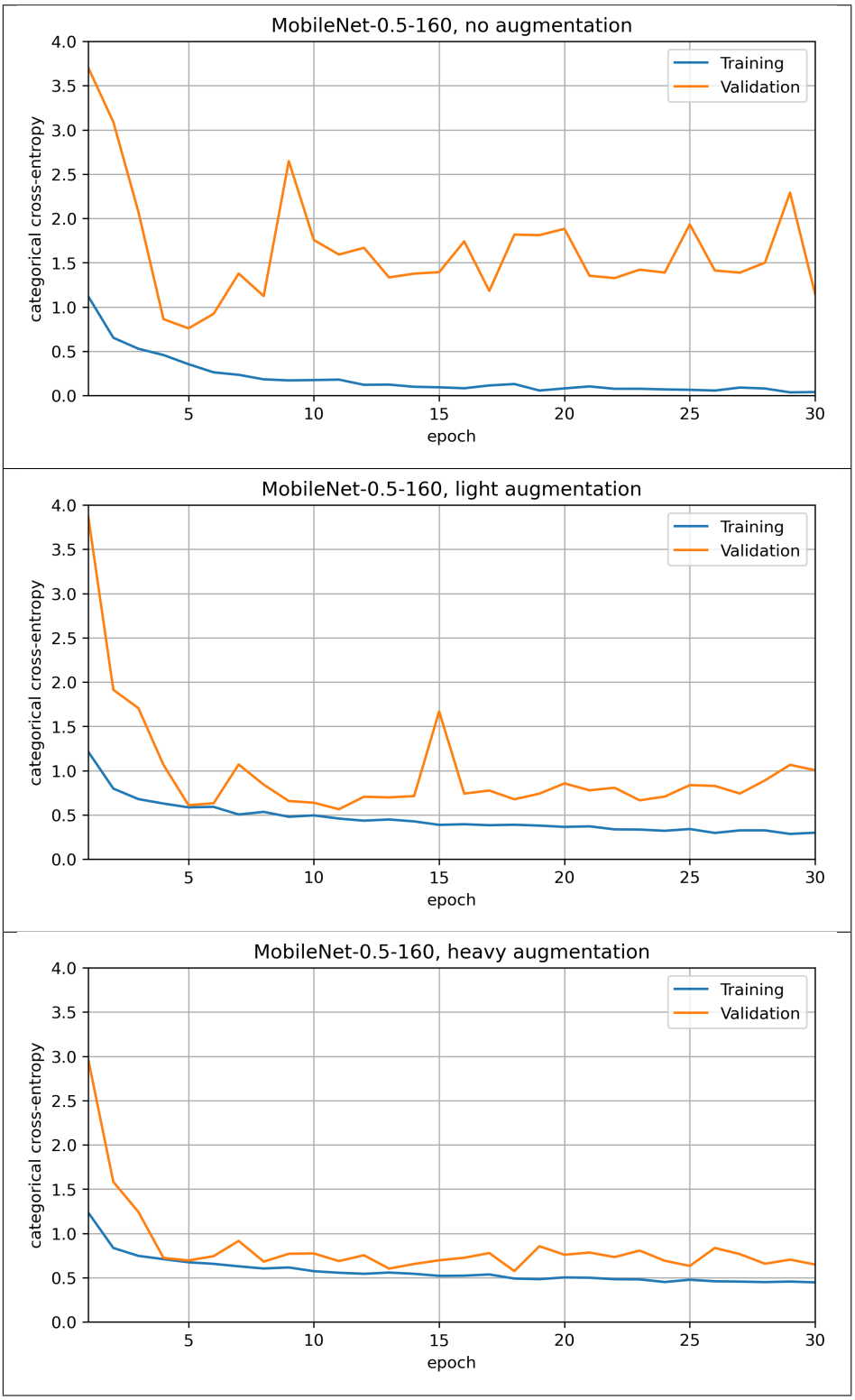
Oversampling and class weights

The negative effects of class imbalance are strongest in the minority classes. High overall accuracy scores can be reached even with very poor performance in minority classes. To better evaluate classifier performance across all classes, we record the macro-averaged F1-score for models trained with oversampled training dataset and weighted classes. Mean validation accuracy from the 10 best epochs is also recorded to measure overall performance. The accuracies of models trained with oversampled dataset and class weights described in Section 4.1.2 are presented in Table 5.7.

While using oversampling or class weights may improve classifier performance in minority classes, it can also have a negative impact on the overall classifier accuracy. In this case, using oversampling had more severe impact on the accuracy score, whereas using class weights had a small positive impact on MobileNet-1.0-224 and EfficientNet-B0 models. For the smallest MobileNet variants, MobileNet-0.5-160 and MobileNet-0.5-224, class weight reduced accuracy, but not as much as oversampling.

Macro-averaged F1-scores from the training run with oversampling and class weights are presented in Table 5.8. The results show improvement in the F1-score in especially the

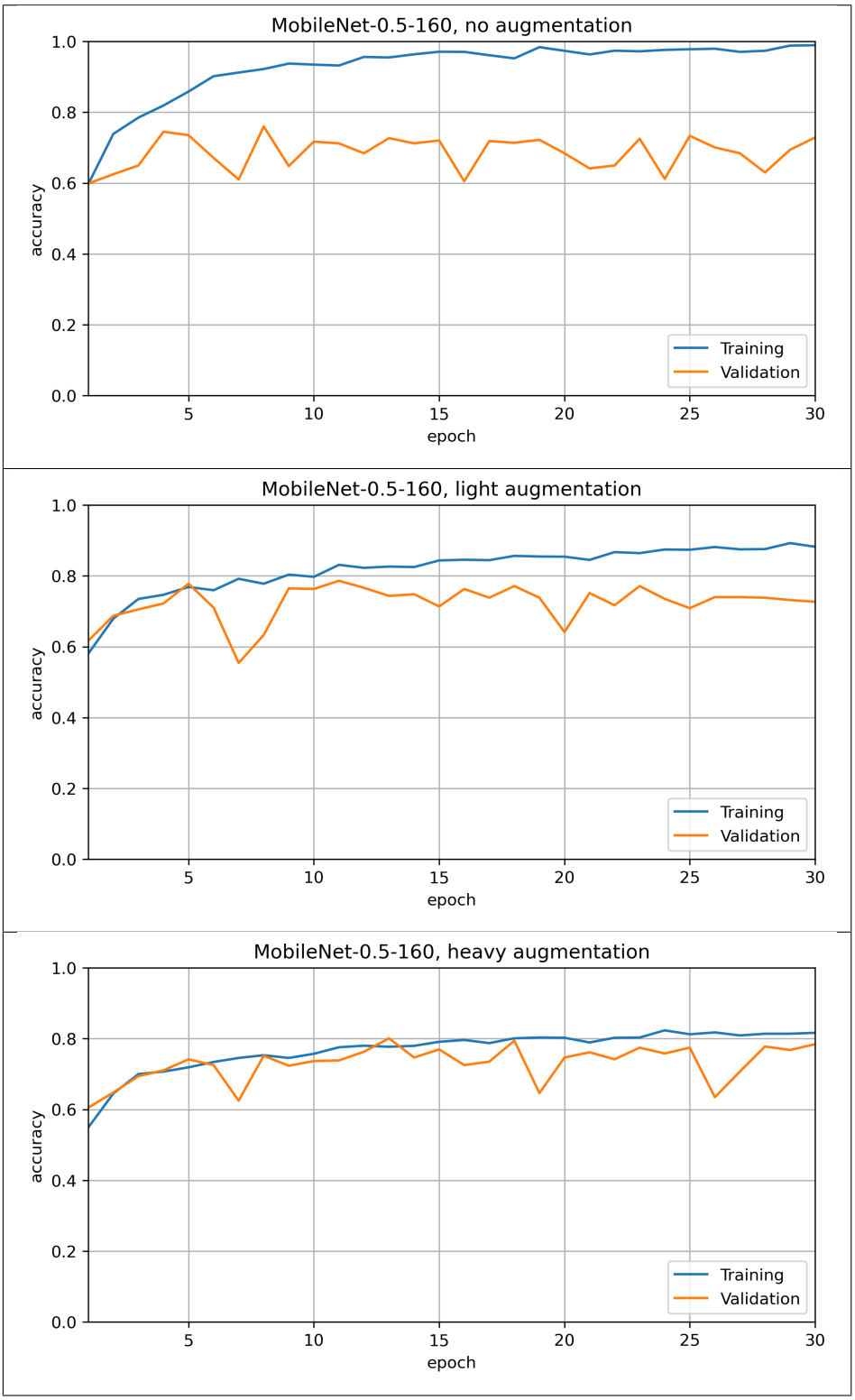
Table 5.5. *MobileNet-0.5-160 training and validation losses with different augmentation levels. Categorical cross-entropy was used as the loss function.*



largest models, MobileNet-1.0-224 and EfficientNet-B0, for both oversampling and class weights.

With imbalanced class distribution, macro-average of F1-score gives a better overview

Table 5.6. *MobileNet-0.5-160 training and validation accuracies with different augmentation levels.*



on classifier performance on all classes than the accuracy. Using balanced class weights with our dataset improves the F1-score macro-average without a big impact on the overall accuracy. The improved performance on minority classes can be seen from the number of correctly classified samples in the smallest of the classes, class 2. There are 12

Table 5.7. *Accuracies of MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models trained with oversampled training dataset and class weights.*

	Accuracy (%)		
	Original	Oversampling	Class weights
MobileNet-0.5-160	77.7	75.6	76.8
MobileNet-0.5-224	80.6	78.6	79.5
MobileNet-1.0-224	80.7	80.5	81.3
EfficientNet-B0	83.4	81.3	83.6

Table 5.8. *Macro-averaged F1-scores for MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models trained with oversampled training dataset and class weights.*

	F1-score, macro-average		
	Original	Oversampling	Class weights
MobileNet-0.5-160	0.66	0.64	0.68
MobileNet-0.5-224	0.68	0.64	0.70
MobileNet-1.0-224	0.64	0.73	0.70
EfficientNet-B0	0.68	0.76	0.76

samples from this class in the validation set, and originally only 3 of them were classified correctly by EfficientNet-B0. With oversampling and class weights, the number of correct classifications went up to 9 and 7, respectively.

Class encoding

The motivation behind ordinal class encoding is to utilize the ordinal nature of the data. The class indices reflect different levels of damage on the screen with class 0 representing no damage and class 4 the heaviest damage. Since the increasing order of class indices match the increasing levels of damage, using ordinal encoding could increase especially the off-by-one accuracy scores of the models.

The accuracies of models trained with the standard one-hot encoded and ordinal encoded classes are presented in Table 5.9. The accuracies are measured as in previous training runs, that is the mean accuracy of 10 epochs with highest validation accuracy.

The class encoding does not seem to have a great impact on the accuracy scores. The accuracy of MobileNet-0.5-224 dropped by 1%, but that is most likely due to randomness in the training process. Considering how high the MobileNet-0.5-224 accuracy is compared to MobileNet-1.0-224 when using one-hot encoding, it is reasonable to assume that the run with one-hot encoding was that much better than with ordinal encoding mostly by chance.

The off-by-one accuracies from the same training runs are presented in Table 5.10. Off-

Table 5.9. *Accuracies of MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with one-hot and ordinal encoding.*

	Accuracy (%)	
	One-hot encoding	Ordinal encoding
MobileNet-0.5-160	77.7	78.1
MobileNet-0.5-224	80.6	79.6
MobileNet-1.0-224	80.7	81.2
EfficientNet-B0	83.4	83.1

by-one accuracy metric is described in detail in Section 4.2.

Table 5.10. *Accuracies of MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with one-hot and ordinal encoding.*

	Off-by-one accuracy (%)	
	One-hot encoding	Ordinal encoding
MobileNet-0.5-160	95.9	97.3
MobileNet-0.5-224	96.4	96.7
MobileNet-1.0-224	96.4	97.3
EfficientNet-B0	96.4	97.4

All models improved their off-by-one accuracy scores with ordinal encoding. The average improvement was 0.9%. The results show that using ordinal encoding instead of one-hot encoding for the classes, the number of misclassification is not reduced, but their severity can be slightly reduced. Misclassifications that are more than one step away from the correct class have can be crucial when evaluating the value of the device, so reducing the number of these big errors is important.

Quantization

The performance of quantized models was measured by exporting the trained models to TensorFlow lite format and deploying them on a mobile device. The exported model was the saved checkpoint file from the epoch with highest validation accuracy during training. Accuracy was measured by predicting classes in the mobile device for samples in the same validation set that was used during training. The representative dataset required by full integer quantization consisted of 200 samples from the validation set.

Accuracies of quantized models are presented in Table 5.11. Dynamic range quantization had severe impact on the accuracy of all the models, making all models but MobileNet-0.5-160 practically unusable. The EfficientNet-B0 also suffered more than 10% reduction in accuracy score when using full integer quantization. Interestingly, full integer quantization did not have similar effect on the MobileNet variants.

Inference times of quantized models in Table 5.12 show that different quantization meth-

Table 5.11. *Accuracies of quantized MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with different quantization methods.*

	Accuracy (%)			
	No quantization	Dynamic range	Float16	Full integer
MobileNet-0.5-160	78.1	76.0	77.9	77.8
MobileNet-0.5-224	80.3	60.0	80.3	80.2
MobileNet-1.0-224	81.1	66.7	81.2	80.6
EfficientNet-B0	82.1	60.1	82.1	70.2

ods have different effect on the model inference time. Full integer quantization had the most significant improvement in speed, while dynamic range quantization actually made the models slower. Float16 quantization did not have effect on the inference times on any of the models. It is worth noting that the tests were run using a CPU, and float16 quantization may reduce the inference if a GPU was available [31]. Full integer quantization had different effect on MobileNet models compared to EfficientNet-B0. For MobileNet models inference time was reduced by approximately 35-45%, and while EfficientNet-B0 also benefited from full integer quantization, the improvement was not as significant.

Table 5.12. *Inference times of quantized MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with different quantization methods.*

	Inference time (ms)			
	No quantization	Dynamic range	Float16	Full integer
MobileNet-0.5-160	18.5	20.2	18.5	12.2
MobileNet-0.5-224	34.9	39.2	34.6	22.8
MobileNet-1.0-224	112.4	120.5	112.8	63.2
EfficientNet-B0	204.7	223.4	203.7	155.9

The sizes of model files after they were converted to TensorFlow lite format using different quantization methods are presented in Table 5.13. Dynamic range quantization reduced model size the most, to approximately 25% of the original. Full integer quantization achieved almost the same file size reduction. The ratios of model file sizes are related to the sizes of data types that are used to store the parameters of the networks. With no quantization, the parameters are stored as 32-bit floating point numbers, and when they are quantized to 16-bit floats the model size is halved. By quantizing parameters to 8-bit integers, model size is reduced to a quarter of the original. For MobileNet variants it is also worth noting that the resolution multiplier does not affect the model size, if width multiplier stays constant. This is because the resolution multiplier does not affect the number of parameters in the network, but increases the number of times the convolution operation must be performed, which can be seen from the increased inference times.

Out of the quantization methods tested in this work full integer quantization showed best

Table 5.13. *File sizes of quantized MobileNet-0.5-160, MobileNet-0.5-224, MobileNet-1.0-224 and EfficientNet-B0 models with different quantization methods.*

	File size (KB)			
	No quantization	Dynamic range	Float16	Full integer
MobileNet-0.5-160	3200	835	1613	1001
MobileNet-0.5-224	3200	835	1613	1001
MobileNet-1.0-224	12518	3179	6172	3518
EfficientNet-B0	15659	4072	7877	5013

results for models with MobileNet backbone. Full integer quantization offered significant improvements in both inference time and model size without significant loss of accuracy. For EfficientNet-B0 however, full integer quantization had a big impact on accuracy. The accuracy of a full integer quantized model was almost 10% less than the accuracy of the non-quantized model. Also the improvements in inference time were not as significant as for MobileNet models. For EfficientNet-B0 the Float16 quantization offers reduction in model size without affecting accuracy, but the inference time was not improved. Dynamic range quantization had the largest reduction in model size, but it greatly reduced the model accuracy in most models. Since model size is not as important metric as accuracy and inference time, dynamic range quantization should not be used. All quantization methods reduced the model size to less than 10MB, which small enough for modern smartphones with several gigabytes of memory.

5.3 Summary

Previously in this chapter multiple methods for improving model accuracy and speed have been discovered. By using all the suitable optimization methods at the same time, the models can be improved even further. This section contains a brief summary of the results presented earlier in this chapter as well as results from combining the best optimization methods discovered for each the models.

The model scaling hyperparameters, width and resolution multipliers in MobileNet and MobileNetV2 architectures and compound scaling coefficient in EfficientNet, allow for an easy way to adjust the model complexity to achieve desired balance between accuracy and speed. From the MobileNet variants, three potential backbones were discovered with unoptimized inference times ranging from 18.5ms to 112.4ms and accuracies from 73.2% to 77.3%. The EfficientNet variants were shown to be significantly slower than the MobileNets, with only the most lightweight EfficientNet variant, EfficientNet-B0, being fast enough for the task with 204.5ms inference time and 81.3% accuracy before applying any optimization methods. The performance MobileNetV2 models was significantly worse than expected, and the problems with their training was the biggest drawback in this work.

Both the classification accuracy and inference time could be improved by applying different kinds of optimization methods. Data augmentation was shown to be the best way of

Table 5.14. *Accuracy, off-by-one accuracy, macro-average of F1-score and inference time for models with different backbones after applying optimizations. For all models, the optimizations applied were heavy augmentation, ordinal class encoding and balanced class weights. Additionally, full integer quantization was applied to MobileNet variants and float16 quantization to EfficientNet-B0.*

	Accuracy (%)	**Off-by-one accuracy (%)**	**F1-score, macro-average**	**Inference time (ms)**
MobileNet-0.5-160	77.5	97.5	0.63	12.2
MobileNet-0.5-224	80.6	97.2	0.77	22.8
MobileNet-1.0-224	81.1	97.3	0.77	63.2
EfficientNet-B0	84.8	97.5	0.82	203.7

improving accuracy, while full integer quantization cut the inference times of most of the models by almost a half. For EfficientNet-B0, the full integer quantization had a severe negative impact on accuracy and float16 quantization should be used instead. Using class weights and ordinal encoding were also beneficial to model performance, even though they did not directly improve overall accuracy. Assigning weights on classes during training produced more balanced performance across all classes. The effect could be seen especially on the accuracy on classes with the least amount of training samples. By utilizing the ordinal nature of the classes through ordinal encoding for the network output, the number of critical mistakes where the predicted class is not even in one of the neighbouring classes of the true class could be reduced. Metrics of the four more thoroughly tested models after combining the best optimization methods are presented in Table 5.14.

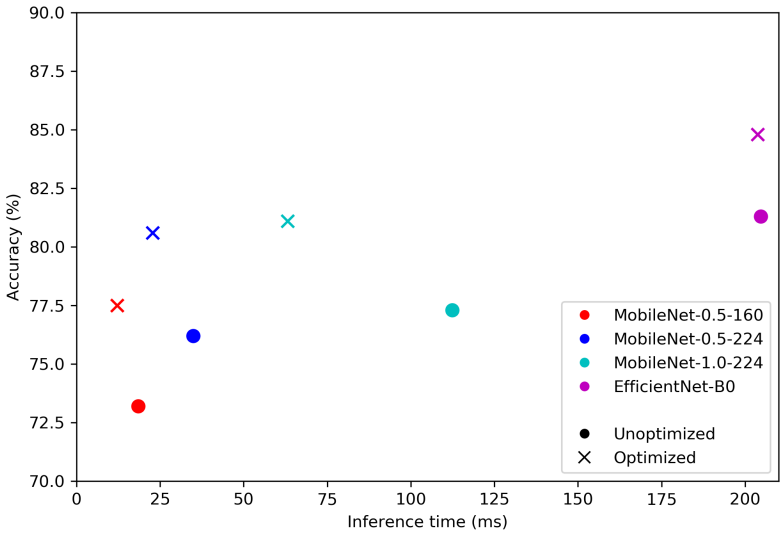


Figure 5.2. *Accuracies and inference times for models with different backbones before and after applying optimizations. For all models, the optimizations applied were heavy augmentation, ordinal class encoding and class weight. Additionally, full integer quantization was applied to MobileNet variants and float16 quantization to EfficientNet-B0.*

Choosing the best CNN model for the classification task described in this thesis is mostly a trade-off between classification accuracy and inference time. Figure 5.2 shows scores of the models on the two most important metrics, inference time and accuracy with and without optimizations. Both the accuracy score and inference time increase as the models grow in complexity. For applications requiring the highest possible accuracy, the EfficientNet is the best choice with an accuracy score of 84.8%. By accepting a 4.2% decrease in accuracy and choosing MobileNet-0.5-224, inference time can be reduced dramatically from 203.7ms to 22.8. Since in the final product the inference must be run 15 times, this corresponds roughly to a difference between a 3 second and 0.3 second delays for the user.

6 CONCLUSIONS

In this work, convolutional neural networks were trained to classify damages on mobile device screens. The networks were able to classify images depicting parts of the screen into 5 categories with 77.5–84.8% accuracy in 12.2–203.7 milliseconds. The networks with higher accuracy were also the ones with slowest inference. Several methods for optimizing network performance were experimented with. Data augmentation, class weights and ordinal class encoding were shown to improve classification performance. Data augmentation was the most efficient method to improve overall accuracy. With model quantization, model size and inference time could be reduced. The most important metrics, accuracy and inference times, are shown in Table 6.1 for the optimized models.

Table 6.1. *Accuracy and inference time for models trained in this thesis.*

	Accuracy (%)	**Inference time (ms)**
MobileNet-0.5-160	77.5	12.2
MobileNet-0.5-224	80.6	22.8
MobileNet-1.0-224	81.1	63.2
EfficientNet-B0	84.8	203.7

Given the ambiguity of the class labels, the accuracy scores achieved by the networks are good and show that CNN based classifier can be used to classify damages in mobile device screens. In an application utilizing this classifier the device screen is divided into 15 parts, and inference is run individually for each part. For the slowest of the networks, EfficientNet-B0, this means that one whole screen can be analysed in 3 seconds, which is a noticeable but reasonable delay for the application user. Networks with lower accuracy can be used for applications requiring instant analyzation.

The training dataset used in this work was relatively small for convolutional neural networks. Even though it is a laboursome task, collecting and annotating more data to use in training is the best way to further improve classifier accuracy in the future. If an application using this classifier is published and data from users can be collected for training purposes, the amount of available data can be increased significantly. Annotating huge amount of pictures is time consuming, but semi-supervised learning methods could be used to reduce the amount of manual labor required. New CNN architectures are constantly being developed, so switching the convolutional backbone of the model to a new, state-of-the-art architecture might be beneficial for the classifier.

This thesis focused only on damages on the screens and fronts of devices, but similar approach could be used also for back panels of smart phones and tablets. The back panels of devices are generally more uneven and have more variations than the front of the device, so detecting damages might be more challenging. If damages on the device need to be located more accurately, image segmentation could potentially be used to detect scratches and cracks with pixel-level precision.

REFERENCES

[1] Johnson, J. M. and Khoshgoftaar, T. M. Survey on deep learning with class imbalance. *Journal of Big Data* 6.1 (2019), 27. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0192-5. URL: https://doi.org/10.1186/s40537-019-0192-5.

[2] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger. Curran Associates, Inc., 2012, 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[3] Goodfellow, I., Bengio, Y. and Courville, A. *Deep learning*. MIT press, 2016.

[4] Glorot, X., Bordes, A. and Bengio, Y. Deep Sparse Rectifier Neural Networks. *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by G. Gordon, D. Dunson and M. Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html.

[5] Bridle, J. S. Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. *Neurocomputing*. Springer Berlin Heidelberg, 1990, 227–236. ISBN: 978-3-642-76153-9.

[6] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning representations by back-propagating errors. *Nature* 323.6088 (1986), 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[7] Murphy, K. P. *Machine learning a probabilistic perspective*. eng. Adaptive computation and machine learning series. Cambridge, MA: MIT Press. ISBN: 9780262018029.

[8] Masters, D. and Luschi, C. Revisiting Small Batch Training for Deep Neural Networks. (20180420).

[9] Kingma, D. and Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (Dec. 2014).

[10] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. and Jackel, L. D. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1.4 (1989), 541–551.

[11] Lecun, Y., Kavukcuoglu, K. and Farabet, C. Convolutional networks and applications in vision. eng. *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2010, 253–256. ISBN: 9781424453085.

[12] Collobert, R. and Weston, J. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, 160–167. ISBN: 9781605582054. DOI: 10.1145/1390156.1390177. URL: https://doi.org/10.1145/1390156.1390177.

[13] Nixon, M. S. *Feature extraction & image processing for computer vision*. eng. 3rd ed. Academic. ISBN: 0-12-397824-6.

[14] Maziarka, Ł., Śmieja, M., Nowak, A., Tabor, J., Struski, Ł. and Spurek, P. Set Aggregation Network as a Trainable Pooling Layer. *Lecture Notes in Computer Science* (2019), 419–431. ISSN: 1611-3349. DOI: 10.1007/978-3-030-36711-4_35. URL: http://dx.doi.org/10.1007/978-3-030-36711-4_35.

[15] Passalis, N. and Tefas, A. Learning Bag-of-Features Pooling for Deep Convolutional Neural Networks. eng. *2017 IEEE International Conference on Computer Vision (ICCV)*. Vol. 2017-. IEEE, 2017, 5766–5774. ISBN: 9781538610329.

[16] Bishop, C. M. *Pattern recognition and machine learning*. eng. Information science and statistics. New York: Springer. ISBN: 0-387-31073-8.

[17] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15 (2014), 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[18] Shorten, C. and Khoshgoftaar, T. M. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6 (2019), 1–48.

[19] Canziani, A., Paszke, A. and Culurciello, E. An Analysis of Deep Neural Network Models for Practical Applications. (May 2016).

[20] Kornblith, S., Shlens, J. and Le, Q. V. Do Better ImageNet Models Transfer Better?: (20180523).

[21] Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (Apr. 2017).

[22] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. (20180112).

[23] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. eng. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 2016-. IEEE, 2016, 770–778. ISBN: 9781467388504.

[24] Zagoruyko, S. and Komodakis, N. Wide Residual Networks. (20160523).

[25] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y. and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. (20181116).

[26] Tan, M. and Le, Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. (20190528).

[27] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A. and Le, Q. V. MnasNet: Platform-Aware Neural Architecture Search for Mobile. (20180730).

[28] Hu, J., Shen, L., Albanie, S., Sun, G. and Wu, E. Squeeze-and-Excitation Networks. eng. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PP.99 (2019), 1–1. ISSN: 0162-8828.

[29] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H. and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-

Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, 2704–2713.

[30] Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. (20180621).

[31] *Tensorflow developers*. *Tensorflow Model Optimization*. https://www.tensorflow.org/model_optimization/guide. (Visited on 03/25/2020).

[32] Buda, M., Maki, A. and Mazurowski, M. A. A systematic study of the class imbalance problem in convolutional neural networks. eng. *Neural Networks* 106 (2018), 249–259. ISSN: 0893-6080.

[33] Jung, A. B., Wada, K., Crall, J., Tanaka, S., Graving, J., Reinders, C., Yadav, S., Banerjee, J., Vecsei, G., Kraft, A., Rui, Z., Borovec, J., Vallentin, C., Zhydenko, S., Pfeiffer, K., Cook, B., Fernández, I., De Rainville, F.-M., Weng, C.-H., Ayala-Acevedo, A., Meudec, R., Laporte, M. et al. *imgaug*. https://github.com/aleju/imgaug. Online; accessed 01-Feb-2020. 2020.

[34] Jianlin Cheng, Zheng Wang and Pollastri, G. A neural network approach to ordinal regression. *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. 2008, 1279–1284.

[35] Kent, A., Berry, M. M., Luehrs, F. U. and Perry, J. W. Machine literature searching VIII. Operational criteria for designing information retrieval systems. eng. *American Documentation* 6.2 (1955), 93. ISSN: 0096-946X.

[36] Sokolova, M. and Lapalme, G. A systematic analysis of performance measures for classification tasks. eng. *Information Processing and Management* 45.4 (2009), 427–437. ISSN: 0306-4573.

[37] Martin 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, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion 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: https://www.tensorflow.org/.

[38] *Tensorflow developers*. *Tensorflow documentation*. https://www.tensorflow.org/api_docs/python/. (Visited on 03/05/2020).

[39] *OnePlus 5 technical specification*. https://www.oneplus.com/support/spec/oneplus-5. Online; accessed 11-Apr-2020.