Automatic Car Damage Recognition using Convolutional Neural Networks

Author:

Jeffrey de Deijn

Internship report
MSc Business Analytics

March 29, 2018

Abstract

In this research convolutional neural networks are used to recognize whether a car on a given image is damaged or not. Using transfer learning to take advantage of available models that are trained on a more general object recognition task, very satisfactory performances have been achieved, which indicate the great opportunities of this approach. In the end, also a promising attempt in classifying car damages into a few different classes is presented. Along the way, the main focus was on the influence of certain hyper-parameters and on seeking theoretically founded ways to adapt them, all with the objective of progressing to satisfactory results as fast as possible. This research open doors for future collaborations on image recognition projects in general and for the car insurance field in particular.

Keywords: Computer vision, image recognition, deep learning, convolutional neural networks, transfer learning, ImageNet.

PricewaterhouseCoopers Pensions,
Actuarial & Insurance Services B.V.
Postbus 90351
1006 BJ Amsterdam

Supervisor PwC:
M. Oeben, MSc

Vrije Universiteit Amsterdam
Faculty of Science
De Boelelaan 1081a
1081 HV Amsterdam

Supervisors VU:
Dr. M. Hoogendoorn
Prof.dr. R.D. van der Mei

Preface

The master programme of Business Analytics turned out to be an excellent preparation for starting a professional career in the current data-driven world. All skills a data scientist is supposed to have nowadays are developed, resulting in a solid basis to enter the professional field. Also the combination with the Econometrics and Operations Research curriculum has proven its value. This has revealed my passion in operations research, mathematical economics and data science, for which I'm very grateful to everyone who contributed to this. This includes all teachers, supervisors and peer students at the university and my former colleagues at PwC PAIS and i2i. Special thanks are going to my supervisors Marvin Oeben (PwC) and Mark Hoogendoorn (VU), who have been a great help during this graduation project, and to SURFsara for providing me access to their Lisa GPU cluster, enabling me to perform my experiments. Finally, I also want to express my big gratitude to my family and friends and, especially, my dear Romée for the great support during my whole study time. It was not always easy, but they were always there for me when I needed them and I couldn't have done it without them.

Jeffrey de Deijn, Amsterdam, March 29, 2018

Contents

1 Introduction

1.1 Research goals

The field of computer vision has greatly developed during the last decade, mainly because of the gain in computing power and available image datasets. In this research, we will explore and apply current state-of-the-art techniques in this field to answer the question:

How accurately can we predict whether a car on a given image is damaged or not?

We do this research in cooperation with the Pensions, Actuarial & Insurance Services (PAIS) department of PwC Nederland, which is concerned with advisory and mediation activities in the fields of pensions and insurances (PwC, 2015). PAIS also contains a small, but strong and ambitious team devoted to data science consulting, which wants to gain more experience in deep learning applications (such as computer vision). The purpose of this desire is that it helps to convince (new) clients of the capabilities of data science, both in general and specifically at PwC PAIS. Our research clearly contributes to this goal by initiating work in computer vision within PAIS, potentially attracting clients such as – but not only – insurance companies that want to automate the processing of car damage claims.

1.2 Literature review

In fact, Jayawardena (2013) already dedicated his PhD thesis completely to automating vehicle damage detection. He even developed prototype software that led to *Control€xpert's EasyClaim app* (Control€xpert, 2015). As we will see more extensively in Section 2.1, his approach requires 3D computer-aided design (CAD) models of the considered vehicle to identify how it would look like if it were undamaged. The fact that we cannot obtain such 3D models (of sufficiently high quality) is only one of the reasons why we are not able to replicate this research. However, ever since Jayawardena finished his thesis there have been great developments in the application of so-called *convolutional neural networks* (ConvNets) in computer vision. In particular, ConvNets have proven their power in object recognition tasks, for which the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) serves as a benchmark (Russakovsky et al., 2015). We will see in Section 2.4 that, from the moment the ConvNet of Krizhevsky et al. crushed all competition in 2012, no ILSVRC contestant has managed to obtain competitive results without using ConvNets yet (ImageNet, 2017). For that reason, we choose to focus on applying ConvNets. As far as we know, this is the first time that ConvNets are being applied in the specific context of car damage recognition. Hence, our scientific contribution is to examine the capabilities of ConvNets in classifying car damage. In this process, we also test some strategies for learning and demonstrate how certain hyper-parameters influence learning as well as the performance of the final model.

1.3 Project plan

Being unaware of any past implementations of ConvNets in the context of car damage recognition, it was difficult to predict a priori how far we could get within this research. We therefore decided to divide the damage classification process into multiple steps, so that we can start with a relatively easy task and increase complexity when we progress. That is, we will first develop a method to classify whether a given image contains a car or not. Since the ILSVRC dataset also includes images of many different types of vehicles and ConvNets perform well here, we expect this first task to be a good warm-up task to examine our methods. After that, we proceed to our main task, which is to classify whether a car is damaged or not. Since damages may look very different depending on the type, location and severity of the damage, we expect this task to be much harder than the first one. If we can perform relatively well on this task we can already be quite satisfied. In the end, we will also do a few experiments to classify, localize and quantify a damage in order to get an idea of how complexity increases here with respect to our main task. In the following, Section 2 first discusses the related PhD thesis of Jayawardena (2013) in more detail, after which it provides some background on all that is relevant for us to apply ConvNets. In addition, we will elaborate on the ILSVRC and some of its top-performing contestants. Section 3 then states our research methodology, which is inspired on the CRISP-DM standard, after which a description of the data is given in Section 4. In Section 5 we specify our research approach, including a detailed description of our tasks and a specification of our models and the way we implement them in our experiments, of which we discuss the results in Section 6. Finally, we end with a conclusion and some recommendations in Section 7.

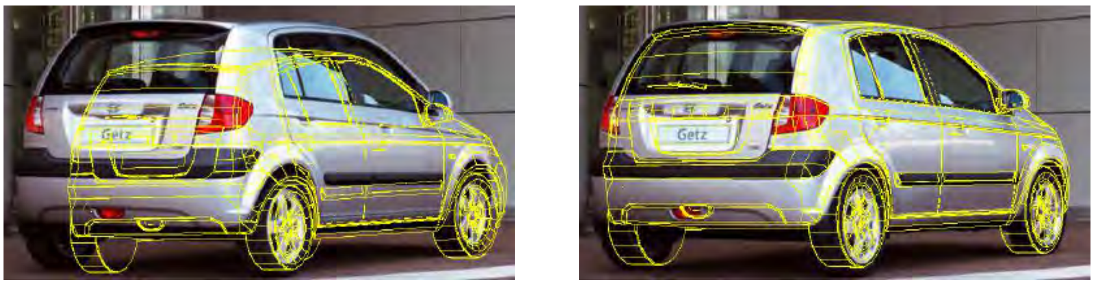
2 Background

We will start this section by expanding on the most relevant related work, which is the PhD thesis of Jayawardena (2013) introduced in Section 1.2. We will explain his overall approach and why we are obliged to choose another direction. Then, we will give a brief introduction of machine learning in Section 2.2, after which we discuss everything we need to know about ConvNets in Section 2.3. Finally, Section 2.4 elaborates on ImageNet and some top-performing models of its visual recognition challenge.

2.1 Jayawardena's approach

As far as we know, the PhD thesis of Jayawardena (2013) is the only work in literature that aims directly at automating vehicle damage detection using photographs. He considered the application of standard computer vision techniques a very challenging task for this purpose and therefore proposed the following approach.

3D pose estimation. Given an image of a known damaged vehicle, first a predefined 3D computer-aided design (CAD) model of that vehicle is registered over the photograph. This results in a model projection that serves as ground truth information, i.e., it identifies how the vehicle would look like if it were undamaged. After removing the background of the image using the *GrabCut* method, the optimal 3D pose is determined by minimizing a distance measure between the adapted image and a full perspective 2D projection of the 3D model[1]. An illustration of this subtask is given in Figure 2.1. Note that the given result is good but not 'perfect', while the 3D model seems quite detailed. Visually perfect matches are achieved when using laser scanned 3D models that are even more detailed. This indicates the importance of the quality of the available 3D model.



(a) Initial rough pose. (b) Final pose.

Figure 2.1: An experimental result of the robust 3D pose estimation procedure developed by Jayawardena (2013), showing in yellow the poses before and after optimization.

[1]The 3D pose has seven degrees of freedom: three for both 3D rotation and shifting and one for scaling/zooming.

3D model assisted segmentation. The second step is to use the recovered 3D pose to identify components of the vehicle, such as doors and fenders. Since each part p is known for the 3D model, it can be projected at the 3D pose to obtain a 2D outline o_p. After applying an erosion morphological operator to get it inside the real boundary, this outline is used to initialize an evolutionary ('level set') method. This method makes the initial curve $\phi_{0,p}$ 'evolve' towards the real boundary of p by exploiting the fact that the gradient on this boundary theoretically tends to infinity. The resulting curve $\phi_{r,p}$ then identifies the desired part of the vehicle in the original image. An illustration of this is given in Figure 2.2. Whereas the 3D pose estimation is sensitive to the chosen GrabCut margin, a potential problem in the segmentation process is the necessity of smoothing to prevent image noise from affecting the performance of the level set method, while over-smoothing may cause the boundary to be missed completely.

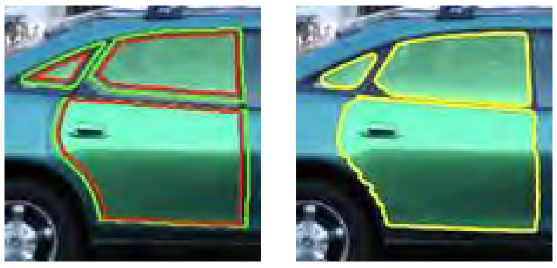

(a) Initialization. (b) Result.

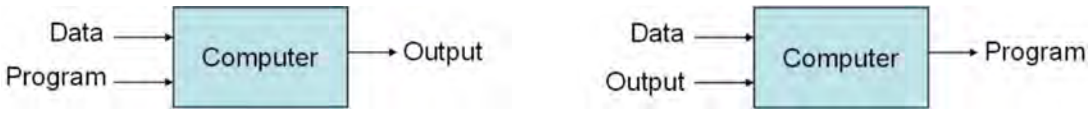
Figure 2.2: An experimental result of the 3D model assisted segmentation procedure developed by Jayawardena (2013), showing the outlines o_p in green, the initial curves $\phi_{0,p}$ in red, and the resulting curves $\phi_{r,p}$ in yellow.

Reflection detection. The final step is to compare the identified components of the vehicle with the 3D model projection. It is assumed that image edges that are not present in the model can either be classified as damage, or as *inter-object reflection*, which is often present due to the highly reflective metallic bodies of vehicles. The unknown scene environment, including illumination and the presence of surrounding objects, causes the amount and appearance of inter-object reflection to be very divergent. For this reason, Jayawardena (2013) claims that applying standard computer vision techniques is very challenging here. Instead, he proposes to first obtain corresponding edge points between two photographs of the vehicle taken from different view points. Then, he seeks to estimate a homography transformation H such that $Hx_i \approx x'_i$ for every pair of corresponding points (x_i, x'_i) that is on the surface of the vehicle body, rather than being inter-object reflection. Based on this, multiple variants of logistic regression are applied to classify whether an edge point is caused by inter-object reflection. Finally, the remaining points can then be compared with the segmentation results to isolate *mild* damage to the vehicle, such as scratches and peeled off paint. It turns out that this method succeeds in recognizing damage as such, but also tends to incorrectly classify reflection edge points as damage. Moreover, the obtained point correspondences tend to be rather noisy at surfaces with a repetitive pattern (such as grilles), which often causes them to be misclassified as reflection. Hence, there is still room for improvement. Unfortunately, only visual results are reported while we will mainly look at numeric metrics, so therefore we can only compare our performance with that of Jayawardena to a very limited extent.

Discussion. Since we do not possess a library of (high-quality) 3D CAD models for every vehicle make and model, we are not able to replicate the method of Jayawardena (2013). However, the fact that they followed this approach does not mean that no better approaches exist. We already stated some weaknesses and limitations of their method, including their focus on *mild* damage only. They do not explain this choice, but a plausible reason would be to exclude intrinsic damage that cannot be deduced from images, making it easier to quantify the damage. Despite this argument, we choose not to restrict ourselves to mild damages, because it is not (yet) ruled out that the possibilities of convolutional neural networks reach beyond this limited area. In the following subsections, we will extensively describe this machine learning model and explain the choice for this approach.

2.2 Machine learning

Machine learning is a major subfield of artificial intelligence (AI) that provides systems the ability to learn to do some task from experience (i.e. training data) without being explicitly programmed (Koza et al., 1996). It can be separated into (i) unsupervised learning, in which a function is learned to describe patterns in the *unlabelled* training data, (ii) reinforcement learning, in which successful strategies are learned from rewards and punishments (trial-and-error), and (iii) *supervised learning*, in which a computer program is learned from *labelled* training data in order to predict the true labels of ‘new’ data (see Figure 2.3). Clearly, our task belongs to the latter category.



(a) Traditional programming (b) Supervised learning

Figure 2.3: Illustration of the idea of supervised learning (Domingos, 2017).

Depending on the type of output, supervised learning is often divided into regression (numeric output) and *classification* (categorical output). Nowadays, many state-of-the-art models for classification come from *deep learning*, which is the family of machine learning methods that are based on *neural networks*. For example, human performance can be approached (or beaten) on typical AI tasks like speech recognition (e.g. Graves et al. (2013), using recurrent neural networks), natural language processing (e.g. Collobert et al. (2011), using unified neural networks) and visual perception, for which we will see in Section 2.4 that current state-of-the-art models are often based on ConvNets. In addition, the broader application of deep learning is also encouraged by the recent technological developments, enabling the processing of increasing amounts of data (in reasonable time) as well as the use of more complex models. Hence, it seems reasonable to believe that ConvNets can potentially perform well in our image recognition task, while still being efficient enough to evaluate in practical applications as well.

2.3 Convolutional neural networks

In discussing convolutional neural networks (ConvNets), we focus on the type of machine learning that we will apply, which is classification. A *classifier* is a program that implements a so-called *score function*. That means, given a data instance, it computes a score for all C possible *classes*. The class with the highest score is then predicted to be the true class for that data instance. In the special case that $C = 2$ (e.g. damaged versus undamaged), we call this *binary classification*. An algorithm that returns a classifier based on a set of labelled training data (illustrated as the computer in Figure 2.3b) is called a *learner*. In order to obtain some structure in choosing a learner from the countless amounts of possibilities, Domingos (2012) characterizes learning as a combination of three components: *representation*, *evaluation* and *optimization*. In the following, we use this structure to discuss the components of ConvNets. We end this subsection with some regularization and ensemble methods that apply to ConvNets in Sections 2.3.4 and 2.3.5.

2.3.1 Representation

Commonly referred to as the model, the representation of the learner is its most characteristic component. It determines the *hypothesis space* of the learner, i.e., the set of classifiers it can learn. The representation of a ConvNet is given by the architecture of the network. That is, a ConvNet is represented by a set of nodes[2] ordered in one or more *layers* that are connected in a *feed-forward* manner (so without any cycles). Figure 2.4 gives an example of a regular neural network with a *depth* of 3 (the number of non-input layers), where the two hidden layers each are of *width* 4 (the number of nodes in a layer).

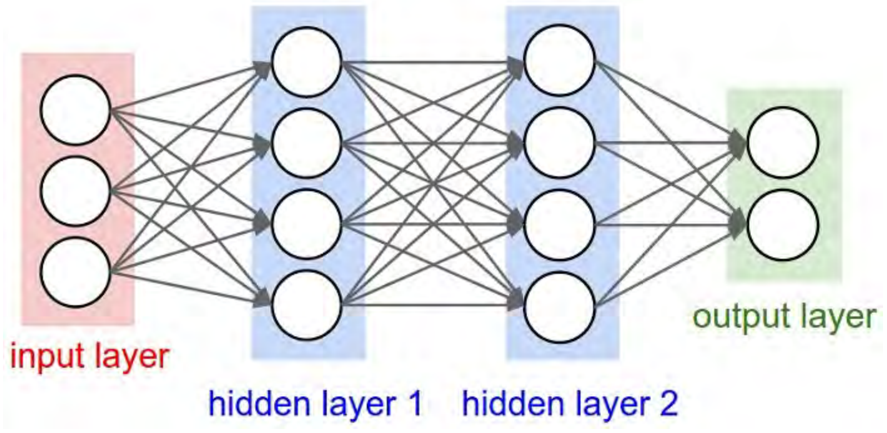


Figure 2.4: A three-layer neural network with three inputs, two hidden layers of four nodes each, and two outputs. Notice that all connections are directed (so we may call them *arcs*) in a feed-forward fashion (from left to right) and there are no connections between nodes that are in the same layer.

[2]By using the term 'nodes', we stress the network structure of the model. It is also common to use 'neurons', stressing the analogy between neural networks and brains, or 'units' instead.

Feed-forward computation. Let each arc (i, j) have a *weight* $w_{i,j} \in \mathbb{R}$, where the receiving node also has a *bias* $b_j \in \mathbb{R}$ and an *activation function* $f_j : \mathbb{R} \to \mathbb{R}$. In the example of Figure 2.4, this means that the number of weights is $3 \cdot 4 + 4 \cdot 4 + 4 \cdot 2 = 36$. Including the biases, this gives a total of $36+10 = 46$ parameters, which is often used as a measure for the *size* of neural networks. Let V_ℓ be the set of nodes in layer ℓ, then the activation x_j of node $j \in V_\ell$ can be computed by $x_j = f_j(b_j + \sum_{i \in V_{\ell-1}} w_{i,j} x_i)$. A big advantage of this structure is that we only need one matrix-vector multiplication per layer, after which we can compute the activation function for all nodes in that layer at once. Note here that for efficiency purposes, it is important to assume that all nodes within each layer have the same activation function.

Activation functions. In order to understand the importance of applying a *non-linear* activation function between layers, note that repetitive linear matrix-vector operations, e.g. $A_2(A_1 x)$, can also be done in one step, e.g. Ax with $A = A_2 A_1$. Hence, multi-layer neural networks only make sense when the activation functions used are non-linear. Moreover, Cybenko (1989) shows that we can arbitrarily well approximate *any function* using a neural network with only one hidden layer and any continuous sigmoidal ('S-shaped') activation function. An obvious example here is the sigmoid (or logistic) function $\sigma(x) = (1 + e^{-x})^{-1}$, which has a nice interpretation for binary decisions as it ranges from 0 to 1. However, for optimization purposes the hyperbolic tangent function $\tanh(x) = 2\sigma(2x) - 1$ (a scaled variant of the sigmoid function) is slightly preferred. Nevertheless, Ramachandran et al. (2017) find in a recent study that the current most successful and widely-used activation function is the rectified linear unit (ReLU) function $f(x) = \max(0, x)$ introduced by Nair and Hinton (2010). In the same study, Ramachandran et al. report promising results with the so-called *Swish* function $f(x) = x \cdot \sigma(x)$, which is similar to the ReLU function, but with the advantages of being smooth as well as having a non-zero left tail derivative. Plots of all mentioned functions are displayed in Figure 2.5. Of course, many more activation function exist, but for a more extensive review on this we refer to other literature, e.g. Schmidhuber (2015).

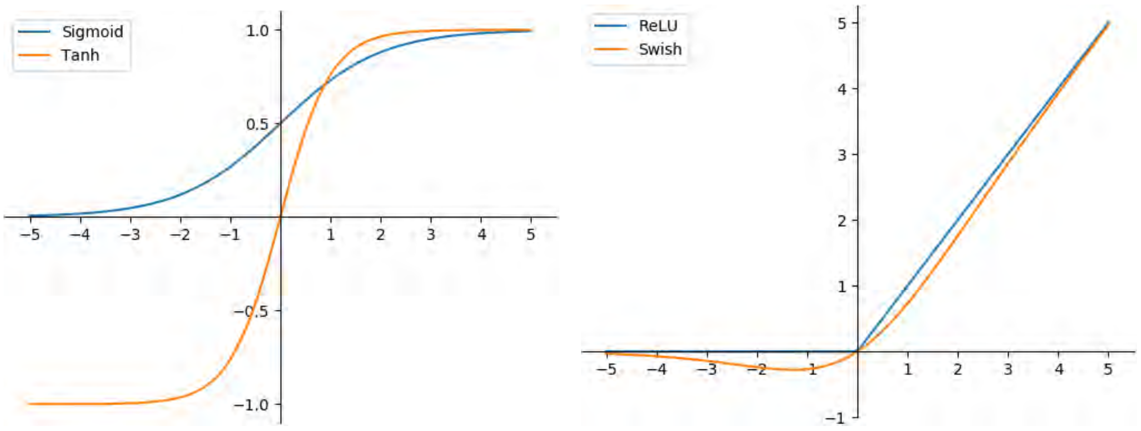


Figure 2.5: **Left**: The sigmoid and hyperbolic tangent function. **Right**: The ReLU and Swish function.

The input layer. We now continue with describing the different types of layers existing in ConvNets, starting with the input layer. This layer contains one node for every feature we have in our dataset. In computer vision, it is common practice to just use the raw Red-Green-Blue (RGB) pixel values, possibly zero-centered and/or normalized, as input. This means we generally have 3D inputs of size $width \times height \times 3$. No further feature engineering is required to compare images other than pixel-wise, because the structure of the model enables the learner to identify valuable features automatically (which we will see later). Note that pixel-wise comparison is very inappropriate, because in this way shifted, rotated or darkened versions of a picture are likely to be classified as (very) different from the original picture, which obviously is not desirable.

Fully-connected layer. For regular neural networks (as in Figure 2.4), the input layer is usually followed by one or more *fully-connected layers*, which means that every node is connected to all nodes in the previous layer. The last fully-connected layer is called the output layer and has width C, equal to the total number of classes so that the output of this layer represents class scores from which we can predict the correct class. A big disadvantage of fully-connected layers is that the number of weights quickly explodes when using image data. For example, suppose that we use (rescaled) images of size $224 \times 224 \times 3$, then we would have over 150 000 weights for *every node* in the first layer. Therefore, other methods are required to reduce the number of weights in some way. In the following, we discuss the two main layer types for ConvNets that can do this.

Convolutional layer. In the convolutional layer, introduced by LeCun et al. (1989), the number of weights is reduced by linking each node to only a limited part of its layer's input. For example, consider a convolutional layer with an input volume of size $W_1 \times H_1 \times D_1$. Every node in this layer corresponds to a filter (or 'kernel') that connects it to a local region of the image, as shown in Figure 2.6. Typically, this filter is a square of size $F \times F \times D_1$, where F is called the *receptive field*. Note that the depth of the filter is always equal to the depth of the input volume, so the connection is only limited in width and height. Every filter is used multiple times by sliding it over the whole input volume (we refer to this as *parameter sharing*). Besides that this heavily reduces the number of learnable parameters (with respect to computing different filters for every region), it is also reasonable to expect that a feature that is useful to compute for one region, is also likely to be useful for other regions. The way we slide the filters over the input volume is specified by:

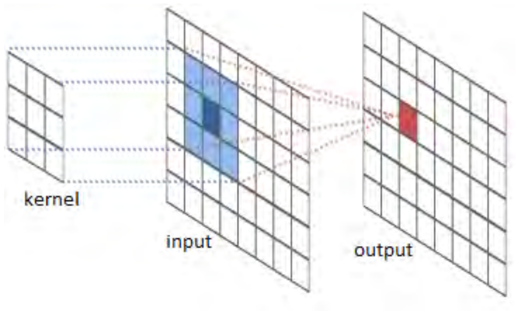


Figure 2.6: An illustration from Intel Labs (2016) of a convolutional filter (the kernel), which computes a weighted sum of every local region of pixels it slides over.

1. the *stride* S, determining the number of (say) pixels we move at a time, and;
2. the amount of *zero-padding* P on the border (see Figure 2.7 for an example), which can be used to control the width and height of the layer's output volume (most commonly in such a way that these are equal to the width and height of the input volume).

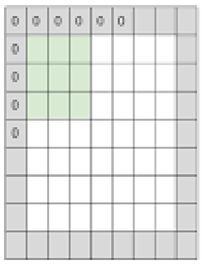
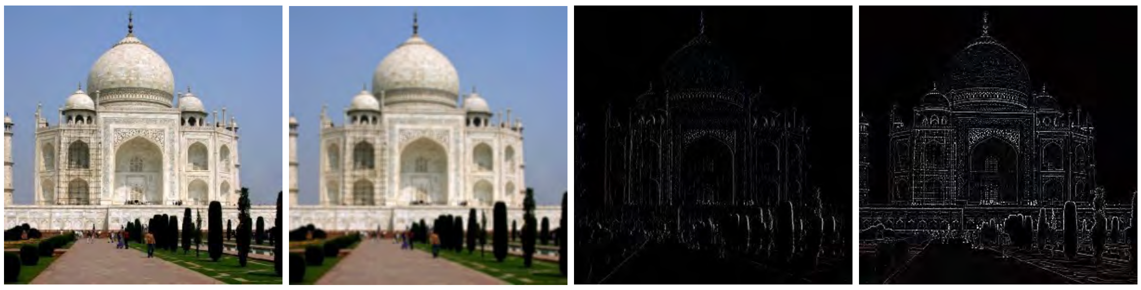


Figure 2.7: Consider an input volume of size $7 \times 7 \times D_1$ (in white, depth not illustrated), a 3×3 filter (in green) with stride 1, and 1 border of zero-padding (in grey). Thanks to the zero-padding, all pixels of the input volume (even those on the border) are once the centre of the filter, resulting in an output volume of size $7 \times 7 \times 1$ containing dot products between the filter and each of the connected regions.

In general, we can compute any number of filters K within a convolutional layer. Hence, the total number of parameters (weights plus biases) in such a layer is $(F^2 D_1 + 1)K$. It can be shown that the layer produces an output volume of size $W_2 \times H_2 \times D_2$, where

$$W_2 = \frac{W_1 - F + 2P}{S} + 1, \quad H_2 = \frac{H_1 - F + 2P}{S} + 1 \quad \text{and} \quad D_2 = K \text{ (one depth slice per filter).}$$

Note that the hyper-parameters F, S and P must be such that W_2 and H_2 are integers. Typical choices are to take a small odd value for F, stride $S = 1$, and zero-padding of $P = \frac{F-1}{2}$ pixels. These settings ensure that the output volume is equal to the input volume. Figure 2.8 gives some examples of what convolutional filters can (learn to) do. Finally, as for fully-connected layers, the output of each convolutional layer is 'activated' by some non-linear function.



(a) Original image.

0	0	0
0	1	0
0	0	0

(b) Blurred.

1	1	1
1	1	1
1	1	1

(c) Detect vertical edges.

0	0	0
-1	1	0
0	0	0

(d) Detect all edges.

0	1	0
1	-4	1
0	1	0

Figure 2.8: Examples from Steiner (2013) of some convolutional filters and their effect on the input image. Every input pixel is once the centre of the filter ($S = P = 1$), so the first kernel does nothing to the original image; The second averages over the surrounding pixels, creating a blurred variant; The third detects differences with the left neighbour-pixels, which enhances vertical edges; The fourth detects differences with all direct neighbour-pixels, which enhances all edges.

Pooling layer. Similar to the convolutional layer, the pooling layer connects nodes to local regions of the input, using some receptive field F and stride S (usually no zero-padding). However, instead of computing one or more filters, a pooling layer just computes a fixed function of each connected region, independently per depth slice and without taking any dot products first. Hence, no extra parameters are introduced in pooling layers. Whereas it formerly was more common to compute averages, it has been shown that computing the maximum (*max-pooling*) works better in practice, especially for cluttered images (Boureau et al., 2010). Supposing an input volume of size $W_1 \times H_1 \times D_1$, pooling results in an output volume of size $W_2 \times H_2 \times D_2$, where

$$W_2 = \frac{W_1 - F}{S} + 1, \quad H_2 = \frac{H_1 - F}{S} + 1 \quad \text{and} \quad D_2 = D_1 \text{ (reduction only in width and height).}$$

Note again that F and S must be such that W_2 and H_2 are integers. Karpathy (2017) claims that it is most common to use receptive field $F = 2$ and stride $S = 2$, so that the input volume is partitioned in disjunct 2×2 squares over which the maximum is computed (see Figure 2.9). It is easy to see that this halves both width and height, resulting in a 75% reduction of the activations. Besides this, he also claims that the only other pooling setting seen regularly in practice is with $F = 3$ and $S = 2$ (called overlapping pooling). Pooling layers are usually inserted periodically in-between successive convolutional layers (as we will see in Section 2.4.2). Although many of the best performing ConvNets nowadays still use pooling, it seems likely that it will be used less in the future. As an alternative, Springenberg et al. (2014) proposes to use convolutional layers with a larger stride once in a while.

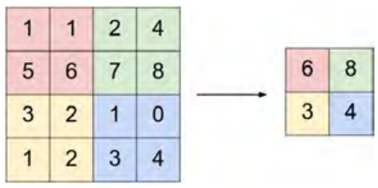


Figure 2.9: An illustration of max-pooling with $F = S = 2$ over a single depth slice.

2.3.2 Evaluation

The fundamental goal of machine learning is to *generalize* beyond the examples on which we train a model (Domingos, 2012). An interesting consequence of this, is that we do not need to fully optimize the training performance of the model. Doing this may even lead to *overfitting*, which is the act (or 'pitfall') of fitting the noise of individual training examples instead of the underlying patterns that are hidden in the full dataset (as in the right graph of Figure 2.10). This problem can be tackled by splitting the available dataset in three separate sets (Ripley, 1996). First we need a *training set* for optimizing the parameters of a specified model. Then, a *validation set* can be used to provide an unbiased evaluation of the resulting classifier. Based on this, the hyperparameters of the model (e.g. the number of layers of a neural network) can be tuned in order to improve performance. However, this validation performance also becomes more biased the more the model has been tuned on the validation set. Therefore, a *test set* can be saved until the very end, so that it can then be used to obtain an unbiased evaluation of the fully-specified classifier.

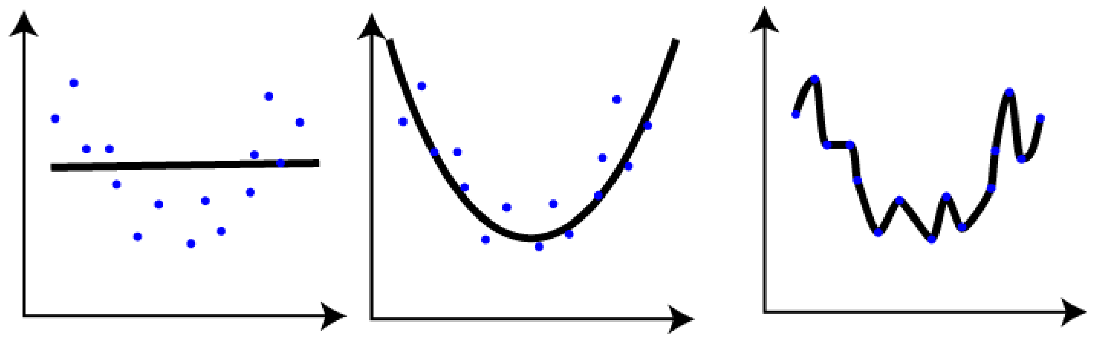


Figure 2.10: **Left**: A line is not flexible enough to capture the underlying pattern of this set of points (underfitting). **Center**: A parabola follows the general pattern reasonably well while still having relatively low complexity. **Right**: A high-order polynomial may fit the data perfectly, but is much more complex and, moreover, it is likely to predict new data poorly (overfitting). The graphs are taken from Johnson (2013).

Cross-validation. Instead of reserving a part of the training data for validation, one can also apply *k-fold cross-validation*. Here, we split the training data into k sets and use each of them as a validation set once while training a model on the other ones. An advantage of this method (especially when the training set is small) is that we do not lose part of the training set for validation. A major disadvantage is that it requires training k models to do one validation. When training is computationally expensive – as will often be the case for ConvNets – it is therefore not a good idea to take k large. On the other hand, the smaller we take k, the less accurate/smooth the results are, so a good balance should be found here. However, to prevent these issues at all, it may *in this case* be better to just use separate sets for training, validation and testing.

Loss functions. Besides the evaluation methods described above, we also need to determine what evaluation function to use for comparing the performance of different classifiers. We make a distinction between the evaluation function used internally by the learner for optimization (to which we refer as the *loss function*) and the one used externally for reporting the performance of the resulting classifier (to which we refer as the *metric function*).[3] Obviously, both these functions must be similar to each other in the sense that a classifier that 'optimizes' the loss function should also perform well according to the metric function. For the loss function, it is important to choose a differentiable function, because we will see in Section 2.3.3 that most common optimizers are gradient-based. Examples include the so-called (squared) hinge loss, mean squared error (MSE) and mean absolute percentage error (MAPE), but we will mainly use the *mean cross-entropy loss*. For any data sample $\{x_i\}_{i=1}^N$, the mean cross-entropy loss L is defined as

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \quad \text{with } L_i = -\log q_{y_i}.$$

[3]Different terminology is used ambiguously across the existing literature for referring to the (internal and external) evaluation function. Other frequently used terms are cost-, score-, error- and objective function.

Here, q_{y_i} represents the predicted probability of x_i being of the *true* class y_i. Since we obviously want to maximize (the logarithm of) this probability, we need the minus to obtain a loss function to *minimize*. Recall from Section 2.3.1 that a ConvNets usually outputs class scores $\{s_j\}_{j=1}^C$, so we need to convert these to class probabilities before can compute cross-entropies. In most studies we will discuss in Section 2.4.2, this is done using the *softmax* function, which gives

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right).$$

In case of binary classification (with $C = 2$ classes), the softmax probabilities q reduce to

$$q_1 = \frac{e^{s_1}}{e^{s_1} + e^{s_0}} = \frac{e^{s_1 - s_1}}{e^{s_1 - s_1} + e^{s_0 - s_1}} = \frac{1}{1 + e^{-s}}, \quad \text{where } s = s_1 - s_0,$$

and $q_0 = 1 - q_1$. Hence, for binary classification it satisfies to output only one class score s, which we can activate using the *sigmoid* function to obtain $q_1 = \sigma(s) = (1 + e^{-s})^{-1}$. Since the scores s_j are a weighted sum of the output layer's input and we maximize the log-likelihood of $\{q_{y_i}\}_{i=1}^N$, this is equivalent to training a *(multinomial) logistic regression* classifier in the output layer.

Metric functions. Whereas the loss function needs to be appropriate for optimization, the metric function should primarily give an easy to interpret measure of the model's performance and is therefore not necessarily differentiable. Batista et al. (2004) observe that metric functions are often based on *confusion matrix* elements (see Figure 2.11). Possibly the most straightforward and well-known metric function is the *accuracy*, measuring the fraction of all data instances predicted correctly, i.e.,

$$\mathit{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

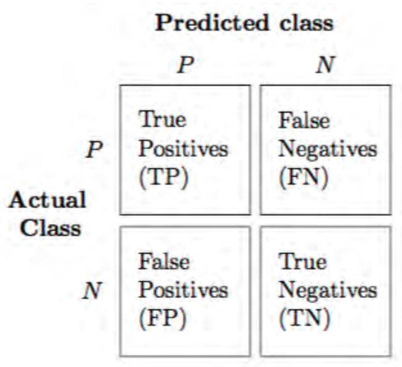


Figure 2.11: A confusion matrix summarizes the performance of a classifier. This illustration, taken from Raschka (2014), assumes that either there are only two classes or that we focus on the performance for identifying a specific class.

However, the accuracy is not always the best choice. For example, it gives a biased representation of the performance when the dataset is *unbalanced*, which is the case when the great majority of the instances is of one particular class. In that case, a classifier predicting only this class already achieves very high accuracy, so it is better to choose another metric function here. A popular choice here is the F_1-*score* (see e.g. Chawla (2009)), which is defined as the harmonic mean of *recall* (the fraction of true examples successfully recognized) and *precision* (the fraction of true predictions actually being true), i.e.

$$\mathit{Recall} = \frac{TP}{TP + FN}, \quad \mathit{Precision} = \frac{TP}{TP + FP}, \quad \text{and } F_1 = 2 \cdot \frac{\mathit{Recall} \cdot \mathit{Precision}}{\mathit{Recall} + \mathit{Precision}}.$$

Figure 2.12 shows visually that both recall and precision must be high in order to achieve a good F_1-score. Note that recall and precision have contradicting goals. For example, a recall of 1 can simply be achieved by always predicting true, but this will give a maximal number of false positives, resulting in poor precision. Achieving high performance on both these metrics is therefore an indication for good overall performance. Another metric function suitable also for unbalanced data is the *Matthews correlation coefficient (MCC)* introduced by Matthews (1975). Given some predicted and true distributions P and Y, this is defined as

$$MCC = \frac{\mathrm{Cov}(P,Y)}{\sqrt{\mathrm{Var}(P)\,\mathrm{Var}(Y)}}.$$

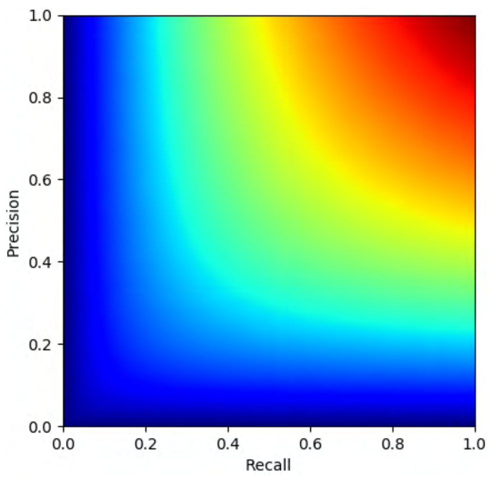


Figure 2.12: An illustration of the F_1-score in terms of recall and precision, where the colors represent its value, from 0 (dark blue) to 1 (dark red).

Although a general definition for multiclass problems is given by Gorodkin (2004), we will only consider the binary implementation, in which case we can also define the MCC as

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}.$$

Note that, while the previous metrics all range between 0 and 1, the MCC ranges between -1 and 1 as it represents a correlation. That means, for example, that an F_1-score of 0.9 is (roughly) as good as an MCC of 0.8. We must keep this in mind when interpreting the results in Section 6.

2.3.3 Optimization

When the network architecture has been determined, the next step is to search for the best classifier within the defined hypothesis space. That is, we search for the set of parameters that minimizes the loss function. This can be interpreted as searching for the bottom of a high-dimensional optimization landscape. The applied optimization algorithm determines where we end up on this landscape as well as how fast we get there. Because ConvNets generally induce complex, non-convex loss functions, it is difficult to optimize them and to prevent getting stuck in a (poor) local optimum (see Figure 2.13). Judd (1990) even shows that there is no polynomial algorithm that guarantees to optimize a given neural network to such extent that it produces the correct output for more than (only) *two-thirds* of the training data. As a consequence, we are also unable to prove whether a given solution is optimal. However, we are often able to approach a sufficiently good local optimum by running as many iterations of *gradient descent* as possible.

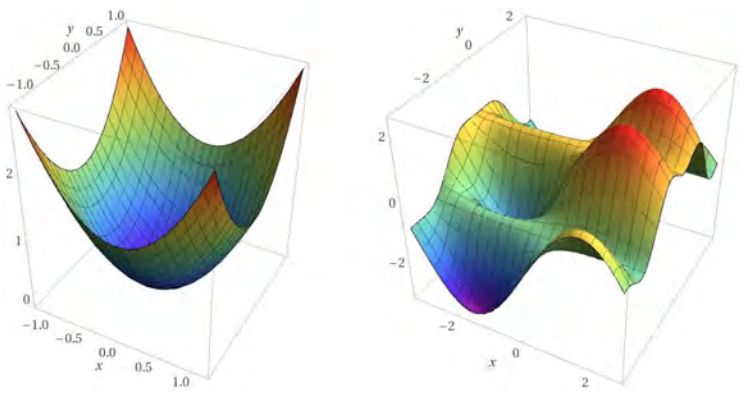


Figure 2.13: Convex functions (example left) are easier to minimize than non-convex functions (example right), since any local minimum is then globally minimal as well (Zadeh, 2016).

Mini-batch gradient descent. The idea of gradient descent algorithms is that, given some point on the optimization landscape, we check in which direction we need to go in order to obtain the steepest descent, assuming an infinitesimal step size. Since datasets for deep learning models are often large, it is often too computationally expensive to compute each of these steps based on the full dataset. On the other hand, using only one training example per step is not efficient either, because of the efficiency advantages of vectorized coding. Therefore, it is common to use *batches* of some size B that balances both efficiency issues (usually some power of 2). The number of *epochs* measures how many times we have gone through the full dataset during training. Now, before we can start training, He et al. (2015) propose to initialize the weights randomly, using a zero-mean Gaussian distribution with standard deviation $\sqrt{2/n}$ for each weight matrix, where n is the number of weights. Then, for each step we use one batch of training data to compute the direction that is locally optimal. This direction relates to the gradient of the loss function, which we can compute analytically in an efficient way using a method called *back-propagation*.

Back-propagation. For a given training step, the back-propagation method introduced by Rumelhart et al. (1986) starts by computing the loss function of the inputs in a systematic way, using a *computational graph*. This is called the *forward-pass*. Figure 2.14 gives an example for the case the loss function is $L(x, y, z) = (x + y)z$, where we first compute $q = x + y$ and only then $L = qz$. Now that the loss is known, we can compute the gradient of the loss function by repeatedly applying the *chain rule*. That is, first we compute the gradi-

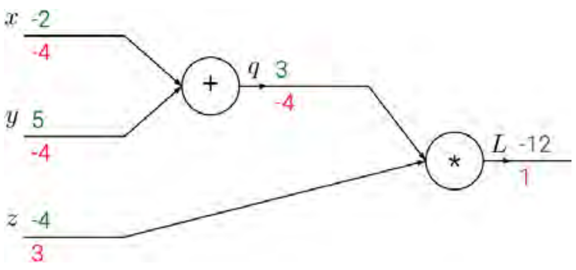


Figure 2.14: A back-propagation example with loss function $L(x, y, z) = (x + y)z$ and inputs $(x, y, z) = (-2, 5, -4)$. The green numbers represent the forward-pass, e.g. $q = x + y = -2 + 5 = 3$. The red numbers represent the backward-pass, e.g. $\partial L/\partial q = z = -4$ and $\partial L/\partial x = \partial L/\partial q \cdot \partial q/\partial x = -4 \cdot 1 = -4$.

ent with respect to q, which we can then use to compute the others, e.g. $\partial L/\partial x = \partial L/\partial q \cdot \partial q/\partial x$. This is called the *backward-pass*. Since the previous example is a bit over-simplified, we now consider a more general and realistic case.[4] Let $x_1 \in \mathbb{R}^{m\times 1}$ be an input vector for a two-layer ConvNet and $W_1 \in \mathbb{R}^{m\times n}$ a weight matrix for the first layer. The output $y_1 \in \mathbb{R}^{n\times 1}$ of this layer can be computed by $y_1 = W_1^\mathsf{T} x_1$, which we then activate using function f_1 (elementwise) to obtain the input $x_2 = f_1(y_1)$ for the second layer.[5] Similarly, we compute the output y_2 of the second layer as $y_2 = W_2^\mathsf{T} x_2$ and the final loss L as $L = f_2(y_2)$. Now, the goal is to retrieve the gradient of the loss with respect to the (vectorized) weights. Starting at the end of the network and using the chain rule repetitively, we get

$$\frac{\partial L}{\partial\,\mathrm{vec}(W_2)} = \frac{\partial f_2}{\partial y_2}\frac{\partial y_2}{\partial\,\mathrm{vec}(W_2)} \quad \text{and} \quad \frac{\partial L}{\partial\,\mathrm{vec}(W_1)} = \frac{\partial f_2}{\partial y_2}\frac{\partial y_2}{\partial x_2}\frac{\partial x_2}{\partial y_1}\frac{\partial y_1}{\partial\,\mathrm{vec}(W_1)}.$$

The exact expressions of the resulting products of partial derivatives are not relevant for now, but what is most important here is that they are all known. Hence, we can use this method to compute the required gradients analytically in a relatively efficient way. These gradients can then be used to update the weights to, hopefully, get closer to a (sub)optimal classifier (in terms of the loss). That is, applying the regular gradient descent algorithm, we take a small step of size α (called the *learning rate*) in the *negative* gradient direction (negative to go 'downhill'). Besides this *gradient step*, it is almost always possible to improve the convergence rate by using *momentum* in the parameter update as well. The idea here is to repeat a fraction $\mu \in (0, 1)$ of the step V taken in the previous iteration, since we expect that the gradient direction does not change drastically over only one iteration. Assuming we take this momentum step μV anyway, it then makes more sense to compute the gradient at $W + \mu V$ instead of W. This is the idea of the *Nesterov accelerated gradient (NAG)* update, introduced by Nesterov (1983), of which a simple implementation developed by Sutskever (2013) is commonly applied in practice. Other, more advanced, *adaptive* update algorithms exist that can tune the learning rate per parameter in each training step, based on the magnitude of the gradient. A popular example is Adam (derived from adaptive moment estimation), introduced by Kingma and Ba (2014). We refer interested readers to this article for the details of this update algorithm.

Scaling the input data. A last remark on optimization is that the back-propagation algorithm clarifies the importance of scaling the input data. For example, note that the local gradients of q and z in Figure 2.14 can be simply computed by switching their input values, i.e., $\partial L/\partial q = z$ and $\partial L/\partial z = q$. This is the case for any pair of nodes preceding a so-called *multiplication gate*. Therefore, if the input values are of very different scales, we may end up with a large gradient for a small input value. As a result, we must be very conservative in specifying the learning rate, which may significantly slow down learning, especially for the values having a larger scale.

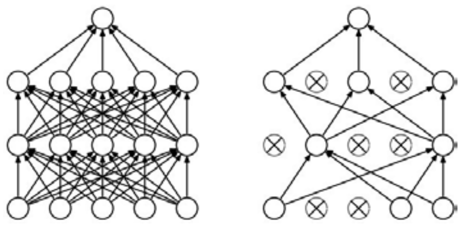
[4]For a numeric example, we refer to Jay (2017).

[5]For simplicity, we assume appropriate dimensions of variables introduced from here on in this example.

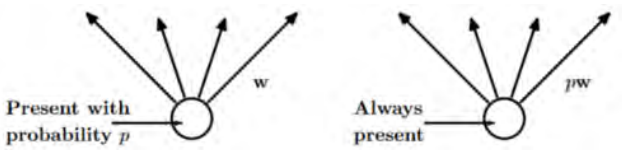
2.3.4 Regularization

Regularization refers to controlling (or 'regulating') the complexity of the learned model, generally with the objective to prevent overfitting (recall Section 2.3.2). The most traditional way is to add a *regularization term* to the loss function. Common choices here are to add $\lambda_1|w|$ (L_1 regularization), $\frac{1}{2}\lambda_2 w^2$ (L_2 regularization, multiplied by $\frac{1}{2}$ to simplify the derivative) or a combination of both (Elastic net regularization) for every weight w in the network. Karpathy (2017) claims that L_2 regularization can be expected to give the best performance in case we are not concerned with explicit feature selection (which is the case for us). Another regularization technique is to directly control the magnitude of the weights, so that their values stay within specified boundaries. This is normally implemented by putting a *max norm constraint* on the weight vector of every neuron, i.e., $\|\vec{w}\|_2 < c$ with c typically around 3 or 4. In the next paragraphs, we additionally discuss some more advanced regularization techniques that are frequently used in present practice for neural network learners.

Dropout. Figure 2.15 illustrates the idea of *dropout* as introduced by Hinton et al. (2012). That is, at every training step we sample from the full network by keeping each node independently with some probability p (i.e., by 'dropping out' each node with probability $1-p$). For example, we then compute the activation of some node y as $y = f(\sum_{i=1}^{N} a_i w_i x_i)$, where a_i ($i = 1, \ldots, N$) is the outcome of a Bernoulli random variable A with $\Pr(A = 1) = p$. Consequently, only the weights w_i for which $a_i = 1$ are updated by the back-propagation algorithm. During testing, no dropout is applied, so in order to match the expected output of each node to that during training, we multiply each weight by p. Experiments of, among else, Dahl et al. (2013) and Simonyan and Zisserman (2014) indicate that dropout may be very effective in some cases. It is common to apply dropout after every fully-connected layer, with $p = 0.5$ as a reasonable default. As a last practical note, Srivastava et al. (2014) claim that even higher improvements can be reached when using dropout alongside max-norm regularization, large decaying learning rates and high momentum.



(a) Neural network before (left) and after (right) applying dropout.



(b) Dropout during training (left) and testing (right).

Figure 2.15: Illustrations of dropout taken from Srivastava et al. (2014).

Batch normalization. An important problem that slows down training of neural networks, is that the distribution of every layer's input changes during training, due to the changing parameters in the previous layers. Ioffe and Szegedy (2015) address this problem (which they refer to as the *internal covariate shift*) by making the normalization of layer inputs part of the model architecture. They call this *batch normalization* and the idea of this mechanism is to force the activations throughout a network to better resemble a unit Gaussian distribution at the beginning of training. It is therefore most logical to apply batch normalization immediately before every activation function. This results in that we can be less careful about the weight initialization and that we can use a higher learning rate, speeding up learning (recall also the last paragraph of Section 2.3.3). Because each training example is normalized differently in every epoch, depending on which other examples are in the same mini-batch, an additional effect of batch normalization is that it regularizes the model, sometimes eliminating the need for dropout.

Stochastic depth. In order to speed up training for *very* deep networks (say over 100 layers), Huang et al. (2016) propose a training procedure called *stochastic depth*.[6] The idea is similar to that of dropout, but instead of determining which nodes to keep during each training step, we 'survive' each layer ℓ with some probability p_ℓ and otherwise we bypass them with the identity function. During testing no layers are dropped, but (similarly to dropout) we need to multiply the activation input of layer ℓ by p_ℓ in order to match the expected output during training.

2.3.5 Ensemble methods

Finally, we can almost always improve performance a few percent by training multiple independent models and then combining them using so-called *ensemble methods*. The hypothesis space of an ensemble generally contains hypotheses that cannot be learned by the individual models, which makes it more flexible. Usually, higher flexibility increases the risk of overfitting the training data (recall Figure 2.10). However, Ueda and Nakano (1996) show that ensemble methods can reduce the (squared) bias and/or variance of the predictions, so these methods are now even used for generalization purposes (see e.g. Yang and Browne (2004)). Moreover, Sollich and Krogh (1996) find analytically that, in large ensembles, it is even advantageous to overfit the individual models in order to maximize the variance-reduction effects. The disadvantage of ensembles is that they are more computationally expensive to evaluate. This may especially become an issue when one wants to deploy such ensembles. In that case, one may consider to follow the approach of Hinton et al. (2014), who attempt to 'distil' the ensemble back to a single model by incorporating the individual log-likelihoods into a modified objective. In the following paragraphs, we will discuss three popular methods for ensemble learning. However, keep in mind here that many more ensemble methods exist (and are applied in practice as well).

[6]Since training such deep ConvNets is not required for our purposes, this paragraph is just for interested readers.

Bagging. Bootstrap aggregating (often abbreviated to *bagging*) is a method proposed by Breiman (1996) that mainly improves the stability of the predictions. The idea of bagging is to bootstrapping the training data, i.e., to take random samples (often with replacement) of the training data. Typically, these samples are of the same size as the original dataset, but also smaller samples can be used when the training set is large. Then, we fit a model on each of the samples and apply, typically, *majority voting* (voting with equal weights) over all individual predictions in order to determine the prediction of the ensemble. Only if perturbing the training set can significantly change the constructed classifier, then bagging can improve accuracy as well.

Boosting. The idea of *boosting* is to create a single strong classifier from multiple weak ones. Many algorithms have been developed that can be considered an instance of the boosting family and their approach generally resembles bagging in many aspects. The main difference is that sampling from the training set is not performed randomly. That is, boosting involves an iterative process where every new sample contains the training instances that are most likely to be misclassified by the previously trained models. This causes boosting to be better at reducing the bias than bagging, but with the disadvantage of having a higher tendency to overfit the training data, because it is more sensitive to noise and outliers. A very popular boosting algorithm is the *AdaBoost* (Adaptive Boosting) algorithm introduced by Freund and Schapire (1995). Here, in each step a new model is trained on just the full training set, but with a weight on each of the N instances. Initially, these weights are all set to $1/N$ and are then modified in each iteration: a weight is increased when the corresponding data instance is misclassified and decreased otherwise. Finally, the predictions of all weak classifiers are combined through a *weighted* majority vote.

Stacking. Stacked generalization (or *stacking*) is a more generic term referring to training a model to combine the predictions of multiple weak classifiers (Wolpert, 1992). Note that bagging and boosting can therefore be considered special cases of stacking as well, although here the weak classifiers are combined by means of a fixed function. In stacking, the combiner model can be, for example, a simple logistic regression model. To illustrate the potential performance of such methods, the two best performing contestants of the *Netflix prize*[7] both used a form of stacking (Sill et al., 2009). However, this is not the type of task that we need to do. In the next subsection, we therefore deepen into the visual recognition challenge of ImageNet.

[7]The Netflix prize is a competition that run from October 2006 to July 2009 in which one million US dollars were awarded to the first contestant that improved the performance of Netflix's own algorithm (called Cinematch) by at least 10% on predicting the user ratings for films, just based on previous ratings (so without any other information about the users of films).

2.4 ImageNet

Quoting ImageNet (2017), “ImageNet is an ongoing research effort to provide researchers around the world an easily accessible image database”. Their dataset is organized according to the semantic hierarchy of WordNet’s English lexical database (Miller, 1995), which groups synonyms into 117 000 unordered sets (*synsets*). Every synset indentifies a distinct concept and all are interlinked by means of conceptual relations, such as super-subordinate (e.g. motor vehicle – car) and part-whole relations (e.g. chair – seat). ImageNet aims to provide around 1000 images for each of the 80 000 synsets corresponding to nouns (we refer to these synsets as ‘categories’). As of August 2014, ImageNet provides an average of 650 manually verified images for 21 841 categories, with a total of 14 197 122 images (Russakovsky et al., 2015).

2.4.1 The challenge

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was first held in 2010. By then, the ImageNet database contained 3.2 million images of 5247 categories (Deng et al., 2009). A core task in the ILSVRC is *image classification*, i.e., to classify which object categories are present on the provided images. As of 2011, the challenge additionally consists of the task to specify locations of the present categories (by means of bounding boxes), but we will focus only on image classification for now. Every year, the ILSVRC provided images of 1000 (varying) object categories for classification, with a total of over 1.2 million images for training, 50 000 for validation and (at least) 100 000 for testing (Russakovsky et al., 2015). It is ensured that the included categories contain images of high diversity in terms of (among else) object scale/size, shape and color distinctiveness, and the amount of texture and image clutter. A submission was supposed to contain a list of at most five category labels for each test image. The error for an image equals the sum of incorrect classifications (both false positives and false negatives) divided by the number of categories. The final score is the average error over all test images.

2.4.2 Top-performing algorithms

Table 2.1 summarizes how the performance on image classification has improved since the start of the ILSVRC in 2010. In the first two years, no ConvNets were used at all, with XRCE happening to be the best in these years. As the best non-ConvNet algorithm in 2012 (ISI, with 26.17%) scored lower than XRCE, it seemed that no major improvements could be made. Luckily, SuperVision showed differently, opening new doors with their relatively small ConvNet model. In the following paragraphs, we will briefly describe some of the top-performing ConvNet architectures, beginning with that of SuperVision. In the end, we should be able to see the general structure of successful ConvNet implementations, which may be inspiring for constructing our own.

Table 2.1: All winners of the ILSVRC image classification task, including references to their corresponding publications (no published work of Trimps-Soushen seems to exist).

Year	Winning team	Error
2010	NEC-UIUC (Lin et al., 2011)	28.19%
2011	XRCE (Sánchez and Perronnin, 2011)	25.77%
2012	SuperVision (Krizhevsky et al., 2012)	16.42%
2013	Clarifai (Zeiler and Fergus, 2013)	11.74%
2014	GoogLeNet (Szegedy et al., 2014)	6.66%
2015	MSRA (He et al., 2016)	3.57%
2016	Trimps-Soushen	2.99%
2017	WMW (Hu et al., 2017)	2.25%

SuperVision (2012). The groundbreaking algorithm of SuperVision (now also known as *AlexNet*) is a ConvNet with only five convolutional layers followed by three fully-connected layers with a total of 60 million parameters. The output of each layer is activated by a ReLU function and the activations of the first two layers are normalized as well. In addition, max-pooling is applied after the first, second and fifth convolutional layers. The last layer's output is activated by a 1000-way softmax function to obtain a probability distribution over all classes. For more details, we refer to Krizhevsky et al. (2012).

Clarifai (2013). The algorithm of Clarifai (now also known as *ZFNet*) is largely based on AlexNet. Zeiler and Fergus (2013) present a way to visualize the features that intermediate layers in a ConvNets have learned. They find that increasingly complex features are learned, from just edges in the first convolutional layer(s) to shapes, collections of shapes, and so on in later layers. They use these visualizations to detect weaknesses in SuperVision's model, so that they can improve upon these specific aspects. Most importantly, they decreased the stride and receptive field of the first convolutional layer and increased the number of filters of the middle convolutional layers.

VGG (2014). The runner-up of 2014 (scoring 7.33%) was the first to recognize the importance of a ConvNet's depth for its performance as well as the advantage of using receptive fields of at most $F = 3$ in convolutional layers. This can be explained by the possibility to replicate any convolutional layer with $F = 5$ (and odd) by two consecutive convolutional layers with $F = 3$. This even increases the expressiveness of the model, because of the extra non-linearity between these layers. Similarly, we can replicate a convolutional layer with $F = 7$ by 3 consecutive convolutional layers with $F = 3$, and so on. Although they did not win the challenge, the *VGGNet* of Simonyan and

Zisserman (2014) is worth mentioning because of its appealingly homogeneous architecture. It stacks five blocks of multiple convolutional layers, followed by three fully-connected layers and a softmax output. A ReLU activation is applied after every layer and, in addition, each of the five blocks is followed by a max-pooling layer. A downside of this ConvNet is that it has around 140 million parameters, making it computationally expensive to train and evaluate the model. Nevertheless, due to its simple but effective architecture, VGGNet has been an inspiration for many later architectures.[8]

GoogLeNet (2014). The architecture of GoogLeNet is of similar depth as that of VGGNet, but containing the more advanced *Inception modules*, introduced by Szegedy et al. (2014). Simply stated, these modules compute multiple convolutional (and max-pooling) layers at once and concatenate the results, so that the model can determine itself which convolutions are most informative. Together with the use of less fully-connected layers (which are very expensive), this results in a relatively efficient ConvNet of only 5 million parameters. GoogleNet also provides the basis of popular models developed by Szegedy et al. (2015, 2016), where the latter also uses the ideas of the 2015 ILSVRC winner.

MSRA (2015). Whereas all algorithms discussed so far use dropout around their last fully-connected layer(s), the residual network (*ResNet*) of MSRA heavily uses batch normalization instead, following the practice of Ioffe and Szegedy (2015). Although this allows deeper networks to converge, it also exposes an additional problem called *degradation*: as the depth of networks increases, both the training and test error tend to get saturated and then decrease (rapidly). He et al. (2016) tackle this problem using *residual learning*: instead of aiming for a layer to learn an output $H(x)$, we aim for it to learn the change $F(x) = H(x) - x$ with respect to the input x. This can be done by short-cutting the input (using an identity mapping) alongside the actual (often convolutional) layer. This solution allows MSRA to train a very deep ConvNet of over 150 layers. As their ResNet ends with only one fully-connected layer (with softmax output), the number of parameters is limited to just over 60 million, so it is computationally cheaper than VGGNet.

WMW (2017). The squeeze-and-excitation networks (*SENets*) of Hu et al. (2017) contribute to the existing literature by focussing on channels (the depth dimension). They introduce the so-called *SE block*, which explicitly models interdependencies between the channels. For details of this model we refer to the given article.

[8]Besides the revolutionary paper of Krizhevsky et al. (2012), the paper of Simonyan and Zisserman (2014) is cited more on *Google Scholar* than that of any other ILSVRC winner (over 43% more than its runner-up He et al. (2016) as of Febraury 27, 2018).

Conclusion. Although very wide, but shallow ConvNets are good at fitting the training data, they often do not generalize well. The advantage of multiple layers is that they can learn increasingly complex features, so the current trend is to develop techniques that allow the training of deeper ConvNets, often using 'network-in-network' architectures. Nevertheless, the global architecture stays more or less the same, stacking multiple blocks of successive convolutional layers with activated (ReLU) output, followed by max-pooling. After these blocks, the model is often ended with a few fully-connected layers or, more recently, with an average-pooling layer and only one fully-connected layer. In both cases, the output of the last layer is often activated by a softmax function to obtain a probability distribution over the possible classes. Finally, batch normalization is often required for training very deep ConvNets (with tens or hundreds of layers) and dropout is still an appreciated technique for (further) reducing symptoms of overfitting.

3 Research methodology

A typical machine learning project is an iterative process that can be divided in multiple phases. The most well-known and widely-used model, called the *cross-industry standard process for data mining (CRISP-DM)*, is developed by Shearer (2000) and identifies six major phases, as illustrated in Figure 3.1. We will briefly discuss for every phase what (generic) tasks it involves, both in general and for us specifically. Besides the original paper of Shearer, we also use the explanation of SV-Europe (2016) and the review of Wirth and Hipp (2000) as references for the following paragraphs.

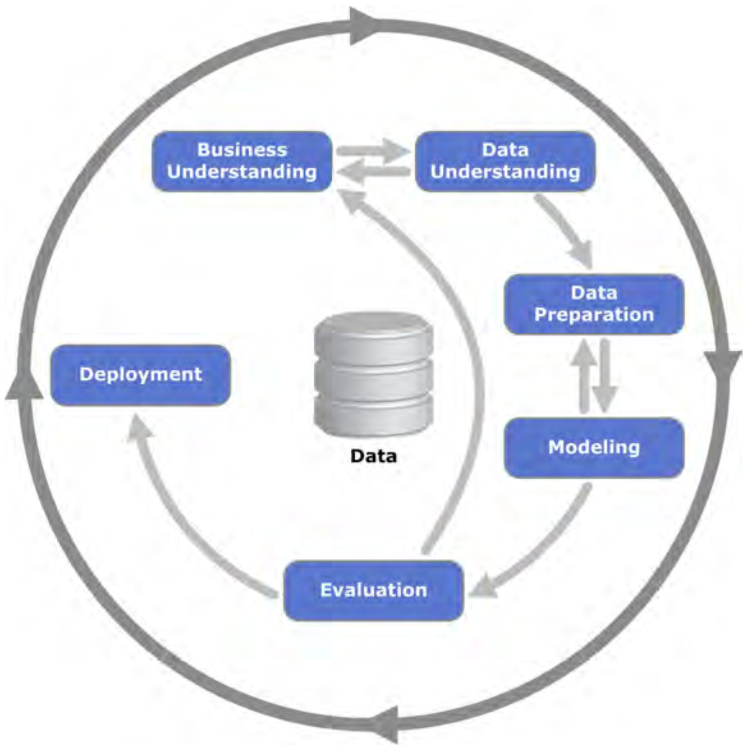


Figure 3.1: Process diagram from Wikipedia (2017) showing the relationship between the different phases of CRISP-DM as well as the cyclic nature of such processes as a whole. For example, if we succeed in recognizing cars, we will restart (part of) the process for recognizing car damage.

Business understanding The first goal is to determine the objectives of the project from a business perspective. The importance of this initial phase is sometimes underestimated, which may lead to putting a great deal of effort into producing the right answers to the wrong questions. Usually, this phase results in a *project plan* that states the current situation (including the available resources, constraints and assumptions), the goals from both a business and technical perspective, and of course a clear plan of how to achieve these goals. As we have already discussed most of this in Section 1, we can be short on this here. We will discuss the available resources in Section 5.2.

Data understanding After the project plan is (largely) completed, the second phase starts with the initial data collection. The acquisition methods and data sources must be reported clearly so that future repetition is possible. Normally, the data must be explored extensively in order to get familiar with its characteristics and to identify possible data quality problems. For us, however, the data exploration part is very limited since the dataset just consists of images, which are to be collected ourselves.

Data preparation In many data mining projects, the data preparation phase is the most time consuming one, especially when the quality of the data is poor. This phase involves data selection and cleaning as well as *feature engineering*, which is the process of creating new attributes (out of the existing ones) that may contain valuable information and possibly improve the performance of the model. This requires a great deal of domain knowledge, data understanding and creativity. For us, the data preparation phase is also very time consuming, but this is mainly caused by the necessity to label the data. Since ConvNets just require the (pixel representation of the) original images as input, no data cleaning or feature engineering is required. We will discuss all our data collection and preprocessing activities in Section 4.

Modelling When the data is ready, the modelling phase starts with splitting it into separate sets for training, validation and testing. Then we can determine our learning approach (representation, evaluation and optimization), optimize the parameters on the training data and use the validation data to assess the resulting model. Since the fitted model may differ based on the settings of the chosen learner (also called its *hyper-parameters*), we often keep iterating this process of model building and assessing until we reach satisfactory performance. This is an important part of our research, since it is interesting to observe how the performance changes as we tweak certain hyper-parameters. The test data must remain untouched until the very end, so that it can be used to test the final model without any *overfitting* bias. We will discuss all modelling choices in Section 5 and perform the actual experiments and results throughout Section 6.

Evaluation The evaluations done in the modelling phase were all of a technical nature, resulting in a model that appears to be of high quality from a data analysis perspective. In this phase, the results as well as the whole process towards them are evaluated from a business perspective, which should in the end yield a decision on the use of these results. We will do this in Section 7.

Deployment In this final phase, a strategy is determined so that (typically) the user is able to deploy the results as well as to monitor and maintain them. Although our results will not be immediately deployed, we will give some recommendations on possible follow-up steps in Section 7. In this case, also this report and the final presentation can be considered a part of deployment.

4 Data

Regarding our stepwise classification procedure, we need to obtain three datasets, containing respectively images without cars, with undamaged cars, and with damaged cars. In the following paragraphs, we describe the collected datasets.

1 – Images without cars. For images without cars, we use the *Caltech-256* dataset of Griffin et al. (2007). This dataset originally contains 30 607 images of 257 different categories (including one 'clutter' category). We exclude nine categories[9], remaining with 29 030 images without cars. All these images are collected by downloading examples from *Google Images* and manually screening out all inapplicable ones. The developers ensured that there are at least 80 images of every category (with an average of 119) and that no rotated variants are contained. There exist other datasets as well, but Caltech-256 seems most suitable for our purposes regarding its size and diversity.

2 – Images with undamaged cars. The best and largest car image dataset that is openly available is the *Cars* dataset of Krause et al. (2013). It contains 16 185 images of 196 different car models and is originally created as a dataset for *fine-grained* classification, meaning that the classes are very similar. For that reason, it only contains regular (passenger) cars, so no bigger vehicles like vans, buses or trucks are included. This is a minor limitation of our research, which is also subtly expressed in the title of this report, stating *car* instead of vehicle damage recognition.

3 – Images with damaged cars. No dataset of images with damaged cars has been found, so we needed to create our own dataset here. Following the method of Griffin et al. (2007) for creating Caltech-256, we 'scraped' Google Images (using *Python*) and downloaded all images yielding from different queries. For the diversity of our dataset, we ensured that the obtained car damages are of different types and severities. We manually checked all collected images and deleted the inapplicable or duplicate ones. This was a very time consuming task, since the majority of the images are cartoons, advertisements, featuring people too much or have other reasons for being inappropriate. We stopped collecting after obtaining 1007 useful images, because at this point it became hard to find queries that yield a substantial number of images that are not contained in the dataset yet. In order to model the location, size and type of car damage as well, we manually assigned labels to the resulting dataset, which is summarized in Table 4.1. We assigned only one label of every category to each of the images. When, for example, the car on an image is both dented and scratched, we assigned the one that stands out most. Note therefore that this

[9]Excluded categories: `031.car-tire`, `050.covered-wagon`, `130.license-plate`, `145.motorbikes`, `146.mountain-bike`, `178.school-bus`, `224.touring-bike`, `229.tricycle`, `252.car-side`.

labelling contains some subjectivity. This also holds for labelling damages close to the (imaginary) boundary between the side and front/rear of the car and, even more, for labelling the damage size. Hence, we have to take this subjectivity bias into account when assessing the results in Section 6.

Table 4.1: Assignment of the 1007 car damage images to the labels of each category.

Type		**Location**		**Size**	
- dent:	391	- front:	358	- large:	282
- glass:	158	- rear:	263	- medium:	426
- hail:	83	- side:	373	- small:	299
- scratch:	375	- top:	13		

Data augmentation. Before preprocessing the data, we apply some minor *data augmentation*, which is the creation of additional images based on existing ones. This is especially useful for the images of damaged cars, of which there are relatively few. A popular method is taking random crops, but this is risky here because we may also crop the actual damage if it is close to the border. Many other augmentation methods exist, such as rotating or shearing images and adapting colours, but we only apply *horizontal flipping* in order to prevent that we overfit too much.

Data preprocessing. The only preprocessing we do before feeding the data into our models is zero-centering and scaling, for reasons similar of that of batch normalization (recall Section 2.3.4), which are also connected to the argument given at the end of Section 2.3.3. Therefore we divide all pixel values by 255 (their maximal possible value) to get them all between 0 and 1, and then we subtract the mean for each mini-batch we feed to our models. More preprocessing techniques exist, such as decorrelating or normalizing the data using respectively principal component analysis (PCA) and whitening, but it is not common to apply these to ConvNets (Karpathy, 2017).

5 Research approach

This section describes our approach of implementing ConvNets to tackle our tasks. First, Section 5.1 explains generally how we can benefit from ConvNets trained on the ImageNet dataset using transfer learning. Then, Section 5.2 gives more details on how we will initiate our experiments and on which platforms we execute them. Finally, in Section 5.3 we discuss strategies on how to assess a trained model in order to determine adaptations that are likely to improve it.

5.1 Transfer learning

In practice, it is very uncommon to have sufficient data and resources to successfully train a full ConvNet from scratch, i.e., with random weight initialization. Karpathy (2017) advices:

> *"Don't be a hero: Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on ImageNet, download a pre-trained model and fine-tune it on your data. You should rarely ever have to train a ConvNet from scratch or design one from scratch."*

Generally, *transfer learning* refers to storing the knowledge gained while solving one problem and then use this to address another problem that is similar to the former one. In the context of ConvNets, this comes down to obtaining complete (pre-trained) architectures and retraining (or fine-tuning) a part of the model to serve our own purpose. We will follow the given advice and choose from the ILSVRC top-performing models described in Section 2.4.2. That is, we will initially use the *VGGNet* of Simonyan and Zisserman (2014) with sixteen 'weight-layers' (excluding, e.g., pooling layers introducing no additional weights), because of its relatively simple, but effective architecture. We will refer to this representation as *VGG16*.

VGG16. The architecture of the original VGG16 model trained for the ILSVRC 2014 is summarized in Table 5.1. Note that the majority of the 138 million parameters (almost 90%) comes from the last three fully-connected layers. The model is optimized using mini-batch gradient descent with batch size 256 and momentum 0.9. For preventing overfitting, L_2 regularization with $\lambda_2 = 5 \cdot 10^{-4}$ is used as well as dropout with probability 0.5 in the first two fully-connected layers. The initial learning rate of 10^{-2} is divided by 10 every time the validation accuracy stopped improving. Since we will use similar model architectures, these optimization settings are likely to work well for us as well. However, besides experimenting with the optimization settings, we also need to change the model architecture in order to be applicable for our specific tasks. The next paragraph discusses different approaches for applying transfer learning, after which we will elaborate on our own implementation in Section 5.2.

Table 5.1: A detailed description of the original VGG16 architecture for the ILSVRC. All images are rescaled to 224×224 RGB pixels and zero-centered (no normalization). Then five blocks of two or three convolutional layers with the same receptive field (e.g. $F = 64$ for the two convolutional layers in Block 1) and each of these blocks is ended by 2×2 max-pooling with stride 2, halving the width and height dimensions. The output of all convolutional and fully-connected (FC) layers is activated by a ReLU function, except from the output layer (FC 3). Here, a softmax function is computed to get a probability distribution over the 1000 possible classes.

Layer	**Description**	**Output shape**	**# Parameters**
Input	rescale image	$224 \times 224 \times 3$	0 M
Block 1	2xConv-64	$112 \times 112 \times 64$	<1 M
Block 2	2xConv-128	$56 \times 56 \times 128$	<1 M
Block 3	3xConv-256	$28 \times 28 \times 256$	1 M
Block 4	3xConv-512	$14 \times 14 \times 512$	6 M
Block 5	3xConv-512	$7 \times 7 \times 512$	7 M
FC 1	width-4096	4096	103 M
FC 2	width-4096	4096	17 M
FC 3	width-1000	1000	4 M

Possible implementations. A potential disadvantage of VGG16 is that training it can take a long time due to its relatively large size. However, we can solve this issue by retraining only a limited part of the parameters. One possible way of doing this is to obtain the VGG16 model pre-trained on ImageNet and drop only the output layer. The remaining network can then serve as a *feature extractor* for our dataset. That is, for every training image we then compute a 4096-dimensional feature vector by feeding it to the network, after which we can use these features to train a simple (e.g. linear SVM or softmax) classifier. As an example, Razavian et al. (2014) successfully address several visual recognition tasks with this approach. On the other hand, it is also possible to drop more than only the last layer, or to just use the pre-trained model as an initialization and fine-tune the parameters on the new dataset (or a combination of both). The best strategy depends on the size of our dataset and its similarity to the ImageNet dataset on which the model is trained. For example, when the new dataset is small, fine-tuning increases the risk of overfitting. Also, the more the new dataset differs from the one on which the pre-trained model is based, the less likely it is that especially the features learned by the later layers are *transferable*, i.e., useful for the new task. This is demonstrated by Yosinski et al. (2014), who also show that successive convolutional layers may contain *fragile co-adapted features*, meaning that this co-adaption could possibly not be relearned by either of the layers when fixing the other, resulting in a performance drop. Not only can this be recovered using fine-tuning, they even show that *fine-tuning transferred features can improve generalization* (provided that there is sufficient data).

5.2 Experimental setup

In this subsection, we first give a more detailed description of the tasks we will tackle and which datasets we will use for this. In addition, we explain our validation method and specify how we will apply transfer learning in our context. Finally, we also discuss what tools and platforms we used for implementation.

Tasks. As mentioned before, we divide our damage classification process into multiple steps. For clarity, all different tasks and their possible outcomes are listed below.

Task 1. Recognizing cars: car or no car.

Task 2. Recognizing damage on car images: damaged or undamaged.

Task 3. Classifying the damage on car damage images:

- a. Type: dented, glass damage, hail damage or scratched.
- b. Location: at the front, rear, side or top of the car.
- c. Size: large, medium or small.

For the first task, we will not use our dataset with damaged cars for training. The reason for this is that images of damaged cars generally look somewhat different than those of undamaged cars, e.g. more zoomed-in or even with heavily deformed cars. It is therefore interesting to test how our model will perform on such images when it is trained to recognize (mostly) completely visible and undamaged cars. Obviously, for the other tasks we exclude the images without cars, and for the last three (sub)tasks also the images of damaged cars.

Data splitting. Since training ConvNets is computationally expensive, using cross-validation to assess our models would take too much time. Therefore, we decided to split the data into separate sets for training (50%), validation (30%) and testing (20%). Due to the small size of our dataset, we need a relatively large part for validation and testing in order to keep enough examples to get reliable performance indications. In the splitting of the dataset with images of damaged cars, we ensured that all labels represent the given percentages as well as possible. The final splits are summarized in Tables 5.2 and 5.3.

Table 5.2: Number of images per category for training, validation and testing (50-30-20% split).

Image category	*Total*	Training	Validation	Testing
No cars	*29 030*	14 515	8 709	5 806
Undamaged cars	*16 185*	8 093	4 855	3 237
Damaged cars	*1 007*	504	302	201

Table 5.3: Number of images per car damage category for training, validation and testing (50-30-20% split, accurate to 0.1% weighted average absolute deviation).

		Total	Training	Validation	Testing
Type	- dent	*391*	196	117	78
	- glass	*158*	79	48	31
	- hail	*83*	41	25	17
	- scratch	*375*	188	112	75
Location	- front	*358*	179	107	72
	- rear	*263*	132	79	52
	- side	*373*	187	112	74
	- top	*13*	6	4	3
Size	- large	*282*	141	85	56
	- medium	*426*	213	128	85
	- small	*299*	150	89	60
	Total	*1007*	*504*	*302*	*201*

Learner specification. As announced in Section 5.1, we will apply transfer learning on VGG16. Since our dataset is relatively similar to that of ImageNet, we expect that a big part of the pre-trained model is transferable to our application. Therefore, we choose to keep the complete convolutional part of the model as it is and only retrain the fully-connected layers. Initially, we also do not apply fine-tuning on the convolutional layers in order to prevent overfitting on our relatively small dataset. For the first two fully-connected layers, we use a width of only 1024 in order to reduce the number of parameters in these layers with almost 78%, from 120 to 27 million. Obviously, the last fully-connected layer also becomes much smaller, since we have at most $C = 4$ classes in our applications (instead of $C = 1000$). When $C = 2$, as for recognizing cars and car damage, we use a width of 1 instead of C in the last layer with sigmoid activation (as explained in Section 2.3.2). All fully-connected layers are initialized randomly as no obvious transfer of weights is possible due to their size being different from their respective original counterparts. We stick to this framework in all our tasks and leave experimenting with other architectures for future research. As stated in Section 1, we focus on tweaking certain hyper-parameters in order to find out how they can influence the model's performance. In particular, we will mainly experiment with the batch size B, the learning rate α, and the amount of regularization (mainly dropout). In addition, we will do a few experiments with the number of fully-connected layers, the types of data preprocessing, and the fine-tuning of the convolutional part. In all cases, we will use a NAG optimizer with Nesterov momentum parameter $\mu = 0.9$ and a cross-entropy loss function, similar to what Simonyan and Zisserman (2014) used for (pre-)training VGG16.

Implementation details. For developing our models, we decided to use Keras, which is a high-level deep learning library written in Python that runs on top of (in our case) TensorFlow.[10] It enables fast experimentation with (recurrent and convolutional) neural networks and also contains a collection of pre-trained deep learning models, among which VGG16. We can therefore just import this model pre-trained on the ILSVRC-2014 dataset, drop the last fully-connected layers and train a new classifier on top of the remaining network. We started computations on a local Intel Core i7 CPU with 12GB RAM. However, the VGG16 model contains tens of millions of weights, of which at least 25 million are to be optimized every time we want to train a model under certain specifications. This takes so many iterations of our gradient descent optimizer to converge to a good classifier, that training a model on our local CPU takes over a week. We therefore searched for a GPU computing platform to speed up learning, which we found at SURFsara. They were testing a new GPU cluster on their *Lisa* system and provided us access to this cluster (for which we are very grateful). It has 23 Intel Xeon Bronze 3104 GPUs with 11GB GDDR5X (VRAM). We could store up to 200GB of files on our account and training a model now took only up to a few hours. For more details on the Lisa system, we refer to the website of SURFsara.

5.3 Learning strategy

In all experiments following in Section 6, our overall approach will be to start by training some specified initial model, matching (largely) the initial learner as given in the previous subsection. We train this model for k epochs, where k is small, but large enough for allowing the learner to roughly converge. In this way, we seek to prevent spending a large amount of time on model specifications that already turn out to be bad in a relatively early stage of training. As long as the validation performance of the trained model is not satisfactory, we adapt the hyper-parameters of the learner and restart training for k epochs (again with random initialization of the fully-connected layers). Once the validation performance is good, we train the corresponding model somewhat longer (possibly with a reduced learning rate) to give it a chance to converge even closer to the (local) optimum. Then, we take the model at the point at which the validation performance is best as our final model. Finally, the test set is fed to this final model to get an unbiased estimate of the model's performance. This overall approach is illustrated in Figure 5.1.

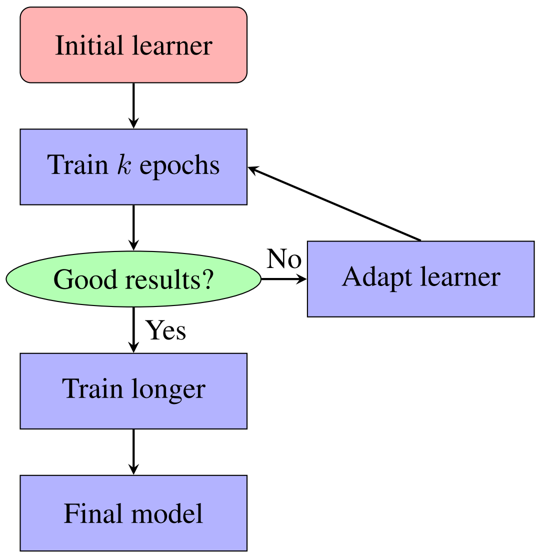


Figure 5.1: Illustration of our learning approach.

[10]We use Keras version 2.1.3, in Python 2.7.14, with TensorFlow 1.5.0 as backend.

It is important to note that we train every learner only once. Therefore, it could be that one learner (incorrectly) comes out better than another due to chance, since both initialization and optimization involve randomness. Multiple trials would be necessary to enable stronger conclusions on some learner being (significantly) better than another. Unfortunately, time constraints make this impossible for us. In order to save more time, we use some diagnostics to adapt the hyper-parameters in a theoretically founded way, hopefully leading us to a good learner faster. In the next paragraphs, we discuss some good practices for this given by Karpathy (2017). In particular, this will allow us to tweak the amount of regularization, the learning rate and the batch size more effectively.

Detecting overfitting. We can get an indication of the extent to which a classifier overfits the training data by plotting the chosen metric function (e.g. accuracy) for both the training and validation data and observing the difference between the two, as shown in Figure 5.2. When the model overfits too much, it is probably desirable to increase regularization or, if possible, to collect more data. On the other hand, when the performances for training and validation are very similar, it is affordable to overfit somewhat more in order to improve the overall performance (e.g. by adding complexity to the model). Obviously, the loss function can be used similarly for this purpose as well.

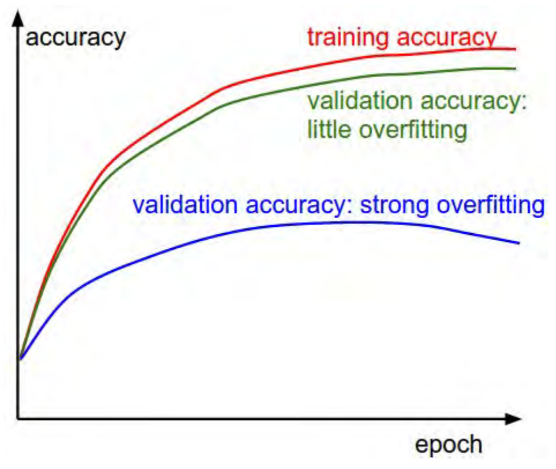


Figure 5.2: Illustration of how the performance develops under strong or little overfitting (Karpathy, 2017).

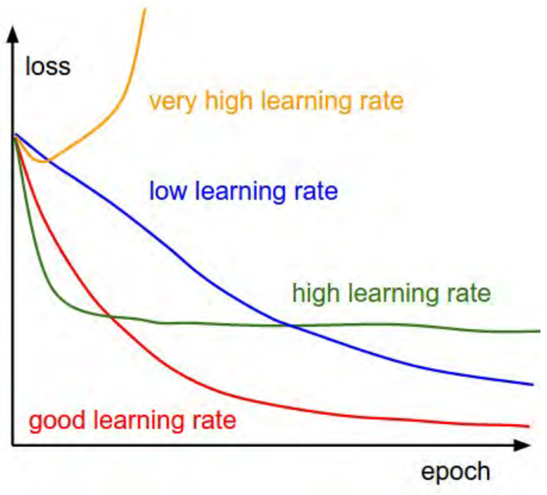


Figure 5.3: Illustration of how the loss develops under different learning rates (Karpathy, 2017).

Detecting a bad learning rate. Figure 5.3 shows how we can also derive from the loss development whether we need to change the learning rate. When it is too low, the learner often just converges to the closest *local* optimum and, moreover, does so very slowly. High learning rates often induce a quick initial decay of loss, but are unlikely to end up in a (local) optimum, because the steps taken in the optimization landscape are too big. Karpathy (2017) also states that, as a rule-of-thumb, the average *relative* parameter update r should satisfy $r \approx r^* = 10^{-3}$. We will therefore adapt the learning rate α after every epoch of training as $\alpha_{new} = \alpha_{old} \cdot (10^{-3}/r)$.

Detecting a bad batch size. Finally, it is interesting to note that the development of the loss function is also useful for assessing the batch size. In Figure 5.4, a loss function is plotted *per mini-batch* (so it shows the loss after every step taken by the gradient descent algorithm). The trend in this plot indicates that the learning rate is probably good, but the graph fluctuates heavily. This fluctuation indicates that the batch size can be taken a little higher, resulting in a smoother loss function development due to each gradient descent step being based on more data.

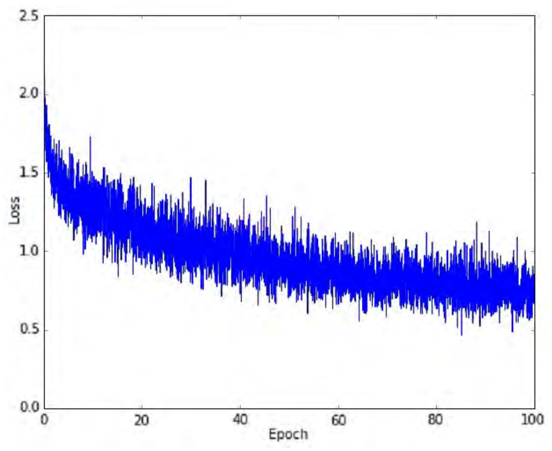


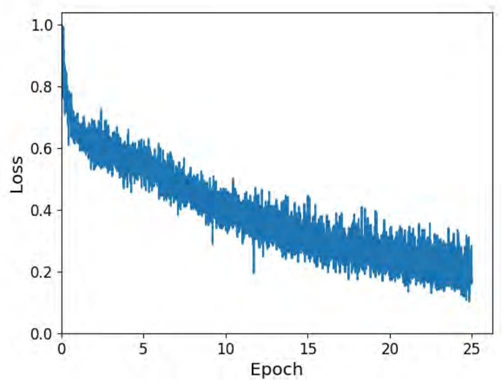
Figure 5.4: Example of how the loss per batch may develop under a good learning rate, but somewhat low batch size (Karpathy, 2017).

6 Experiments and results

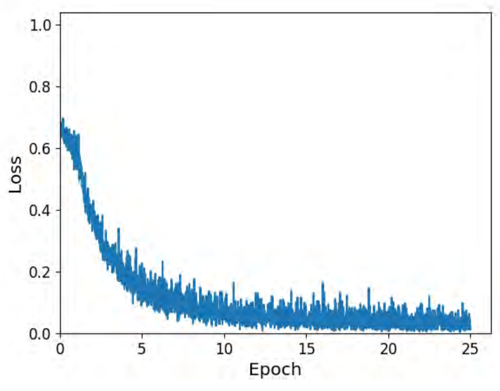
In this section we perform the actual experiments, starting with recognizing cars in Section 6.1 and then increasing the complexity to recognizing car damage in Section 6.2 and, finally, classifying also the type, location and severity of a recognized damage in Section 6.3. The results are given and discussed along the way. In all experiments, we use the learner specification and strategy as described in the previous section. Only the exact values of some hyper-parameters with which we will experiment are given in this section, when we apply the corresponding model.

6.1 Recognizing cars

Initial model and adjusting the batch size and learning rate. Initially, we use an initial learning rate of $\alpha = 10^{-6}$, a batch size of $B = 64$ and we apply dropout with probability $p = 0.75$ of a node being present during training in the first two fully-connected layers. Furthermore, we only scale the input data (no zero-centering) and we train only the three fully-connected layers at the end of the network. Finally, we assess our models using the *accuracy* metric, giving around 64% performance when always predicting 'no car'. Using these specifications, we train the models for $k = 25$ epochs. A plot of the loss function per mini-batch for the initial model is given in Figure 6.1a. It looks similar to the one in Figure 5.4 regarding the amount of fluctuation, so we can conclude here that we may increase the batch size. In addition, it stands out that the decreasing trend looks linear for quite some time and weakens only after about 15 epochs, resembling the blue curve in Figure 5.3. This indicates that the learning rate may be increased. Based on these observations, we restart training the same model, but now with batch size $B = 128$ and aiming at a higher average relative parameter update $r^* = 10^{-2}$, with an initial learning rate of $\alpha = 10^{-5}$. The loss function of this second attempt is given in Figure 6.1b.



(a) $B = 64$, $\alpha = 10^{-6}$, $r^* = 10^{-3}$.



(b) $B = 128$, $\alpha = 10^{-5}$, $r^* = 10^{-2}$.

Figure 6.1: The loss per mini-batch for the initial model as well as after the first adjustments as given.

Comparing the two loss functions in Figure 6.1, multiple interesting differences can be observed. First, we see the consequence of the random initialization of the fully-connected layers, as the second loss function clearly starts lower than the first one. However, the first loss function has already catched up in the first epoch, despite of that it has a smaller initial learning rate. This indicates that a bad random initialization is not necessarily problematic, certainly when using a small learning rate. On the other hand, the larger the learning rate, the higher is the risk of diverging to a hopeless region of the optimization landscape when the random initialization is bad. This can be detected quickly, since the loss function 'explodes' in this case (recall Figure 5.3). We can replicate this behaviour by training three new models for a few epochs, all with the same specification except from the learning rate. From the result in Figure 6.2, we conclude that it seems safe to assume that, as long as the loss function does not explode, a single run of training gives a reasonably good indication of the quality of a learner, regardless of its initialization.

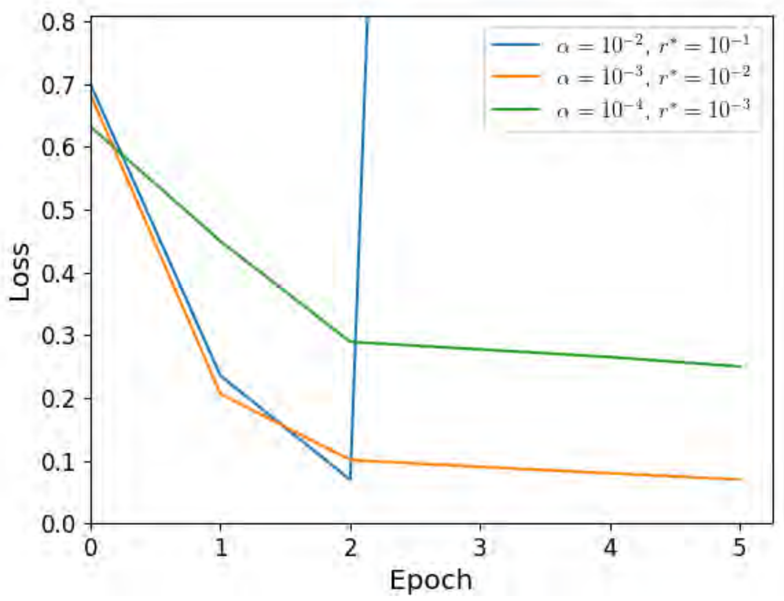
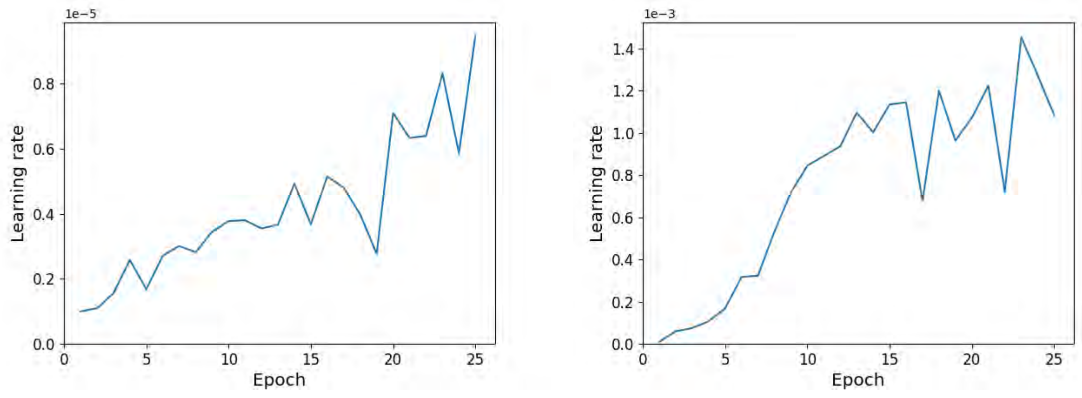


Figure 6.2: Loss functions for three models, all with the same specifications as in Figure 6.1b except from the learning rate, which is specified in the legend. Clearly, the learning rate corresponding to the blue line is too large, that of the green one is somewhat small, and that of the orange one is good.

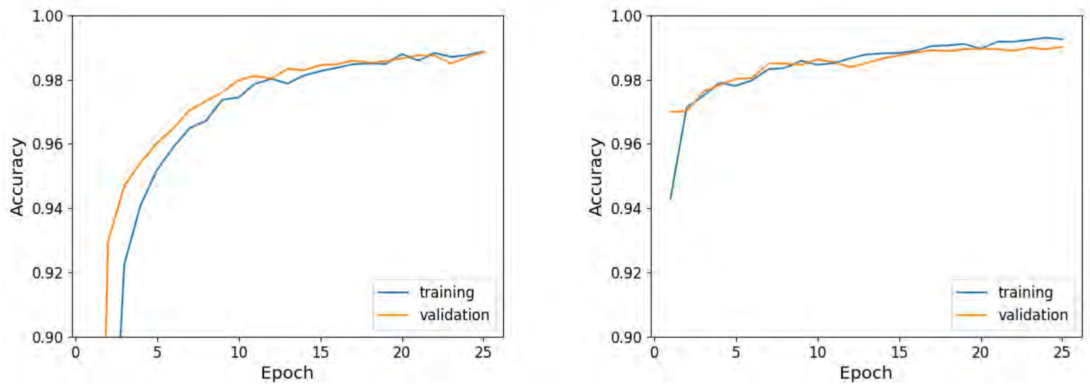
Another observation from Figure 6.1 is that the second loss function does not improve much during the first epoch. This suggests that the initial learning rate may be too small with respect to the desired average relative update r^*. This is confirmed by Figure 6.3b, showing an increasing pattern for our learning rate, instead of a large, decreasing learning rate, which is commonly used in literature (e.g. Srivastava et al. (2014); Simonyan and Zisserman (2014)). From Figure 6.3a we conclude that the same holds for our initial model, so in both cases we could probably better use a higher initial learning rate α. Since the overall trend in Figure 6.1b looks good, it seems better to aim at an average relative parameter update of $r^* = 10^{-2}$ in this case.



(a) $B = 64$, $\alpha = 10^{-6}$, $r^* = 10^{-3}$. (b) $B = 128$, $\alpha = 10^{-5}$, $r^* = 10^{-2}$.

Figure 6.3: The learning rate we used per epoch in training our first two models as given.

Drop fully-connected layer. Since the second model already has a validation accuracy of 98.75%, we only try one more model. Here, we use one less fully-connected layer, as it is found that this can be done without a significant performance downgrade (see e.g. Hoffer et al. (2018)), while it reduces the number of (trainable) parameters. As for the second model, we use batch size $B = 128$ and we aim at average relative update of $r^* = 10^{-2}$, but now with an initial learning rate of $\alpha = 10^{-3}$. Figure 6.4b shows that the model heavily benefits from this higher initial learning rate. With respect to the model from Figure 6.4a, it already converges to a reasonable fit during the first epoch, resulting in a higher performance after 25 epochs as well, with a validation accuracy of 99.01% (recall from Figure 5.3 that this is not necessarily the case). Since this last model performs best, we choose to train this one for some longer, aiming now at a smaller average relative update $r^* = 10^{-3}$ as we already seem to be close to a (local) optimum. Based on the validation data, the best model is reached after a total of 40 epochs, with an accuracy of 99.14%.



(a) $\alpha = 10^{-5}$, $r^* = 10^{-2}$, 3 FC layers. (b) $\alpha = 10^{-3}$, $r^* = 10^{-2}$, 2 FC layers.

Figure 6.4: The accuracy per epoch for the two given models. Note that, especially in earlier epochs, the validation accuracy is sometimes higher than the training accuracy, which may indicate that the validation data has lower inner variance, making it ‘easier’ to predict.

Results. The final model as described above is evaluated on the test set, resulting in the confusion matrix in Figure 6.5. From this figure, we can extract that the test accuracy is 99.04%. Regarding that our goal is only to demonstrate the potential power of (partially transferred) ConvNets in this context, this is a satisfactory performance. Although the images of damaged cars are not used to train this model, it also recognizes 89.1% of them correctly as cars. Taking a look at the false predictions, we see that the model generally has difficulties with images of very low resolution, car types that are uncommon in the training data (e.g. jeeps or sports cars), and car images with the viewpoint above the car. Regarding the non-car false predictions, we see that four fire engine images are contained, which is a category we apparently missed when excluding vehicle-like categories from the non-car dataset. In addition, we find some images that are labelled as 'no car', but which do contain a car (see Figure 6.6). Our model cannot really be blamed for being wrong here.

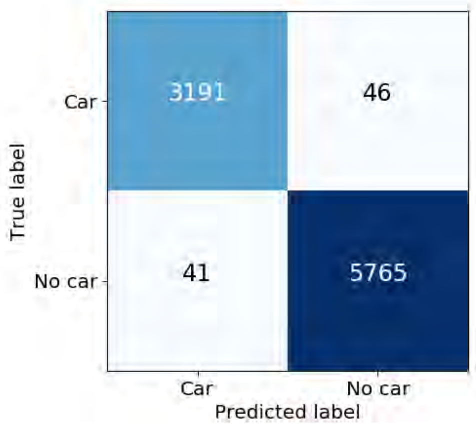
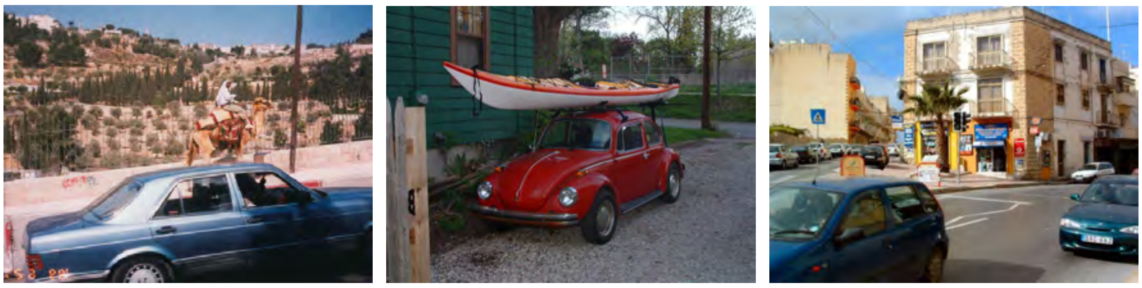


Figure 6.5: Confusion matrix representing how well our final model predicts whether the test images (exluding those of the damaged cars) contain a car or not.



(a) Camel. (b) Kayak. (c) Traffic light.

Figure 6.6: Some non-car labelled images that have been predicted as car images. The original Caltech-256 label is given below the images. We can conclude that our model is not very wrong here.

6.2 Recognizing damage

Initial model and zero-centering. Because of our observations in the previous subsection, we immediately use only two fully-connected layers here. Since we only have 504 training images with *damaged* cars, we initially use no dropout or other regularization technique. We first want to see if the model can learn to perform well on the training set and what it then does for the validation set. We start with a batch size $B = 64$, we aim at an average relative update $r^* = 10^{-3}$ with initial learning rate $\alpha = 10^{-4}$ and we scale the input data as before. In our first experiments here, we test the effect of zero-centering the input data. We train the models for $k = 25$ epochs, after which we look how the different models perform based on the *MCC* metric, since only 5.8% of the images contains a *damaged* car. The first results, shown in Figure 6.7, do not show big differences between applying zero-centering on the input data or not. Since the variant *with* zero-centering has a slight advantage at the end (despite of a worse initialization) and because Simonyan and Zisserman (2014) also use it in their VGGNet implementation, we decide to apply zero-centering in all our further experiments. An extensive analysis of whether this makes a significant difference is beyond the scope of this thesis, since our main goal is to show the potential of ConvNets in this context, rather than the exact specifications that work best. Finally, Figure 6.8 indicates that we may also experiment with a higher batch size and learning rate again.

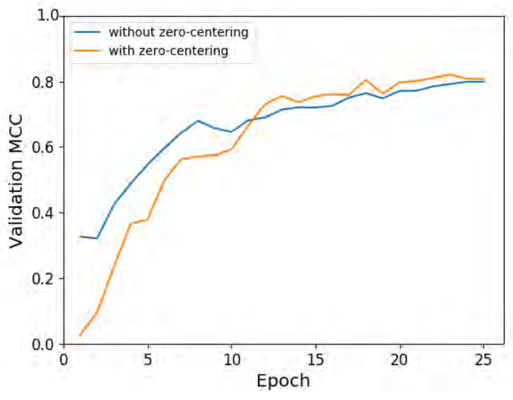
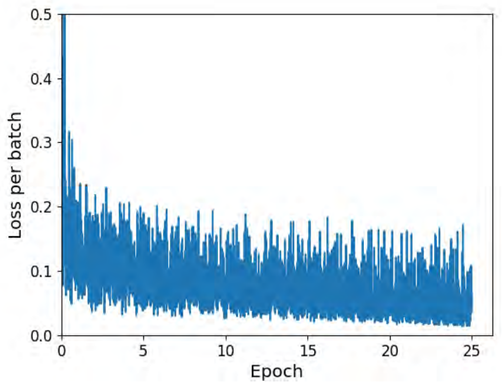
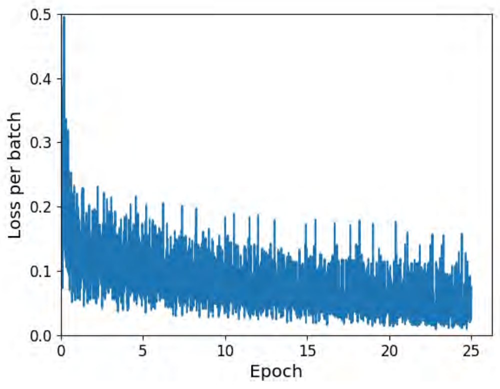


Figure 6.7: The validation MCC of the initial model, with and without zero-centering the input data. Both end up around 0.8 after 25 epochs.



(a) Without zero-centering.



(b) With zero-centering.

Figure 6.8: The loss per mini-batch for the initial model. In both cases, the high amount of fluctuation and slow decay indicate that we may experiment with larger batch sizes and learning rates.

Adding regularization. First, the results in Figure 6.9 show that the initial model with a higher batch size and learning rate quickly yields very promosing performance. However, it is also clear that this model overfits the training data to such extent that the validation loss even tends to start increasing. We therefore add dropout (with $p = 0.5$) in the first fully-connected layer and retrain the model again. In addition, we also recover the model from Figure 6.9 after 16 epochs of training (when its validation performance was highest) and continue training from there on, but now with dropout ($p = 0.5$) and relative update $r^* = 10^{-3}$. Figure 6.10 shows that this regularization has (in both cases) not the desired effect of increasing the validation performance. Figure 6.11 shows that the gap between training and validation performance did get smaller with respect to the case without regularization (recall Figure 6.9), but primarily due to a lower training performance. Nevertheless, model A here slightly improves to a validation MCC of 0.9328 after 46 epochs.

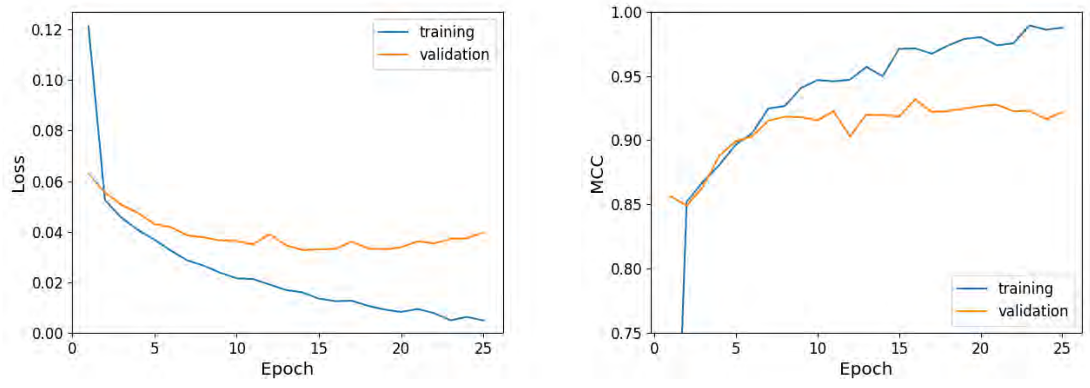


Figure 6.9: The loss (left) and MCC (right) of the initial model with zero-centering, but with higher batch size $B = 128$, relative update $r^* = 10-2$ and initial learning rate $\alpha = 10^{-3}$.

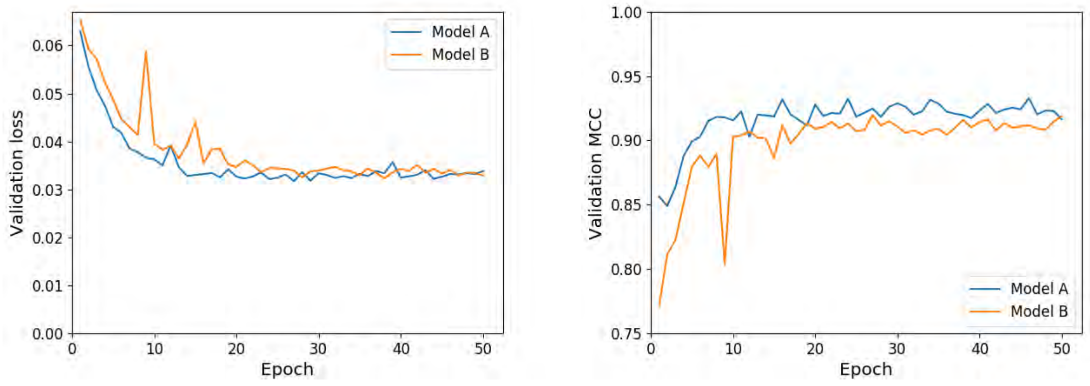


Figure 6.10: The loss (left) and MCC (right) for two models similar to the one in Figure 6.9, but with dropout ($p = 0.5$) in the first fully-connected layer. The only difference is that one model (A) is initialized using the model from Figure 6.9 after 16 epochs (from that point onwards using $r^* = 10^{-3}$) and the other (B) is initialized randomly in the fully-connected layers and uses $r^* = 10^{-3}$ after 25 epochs.

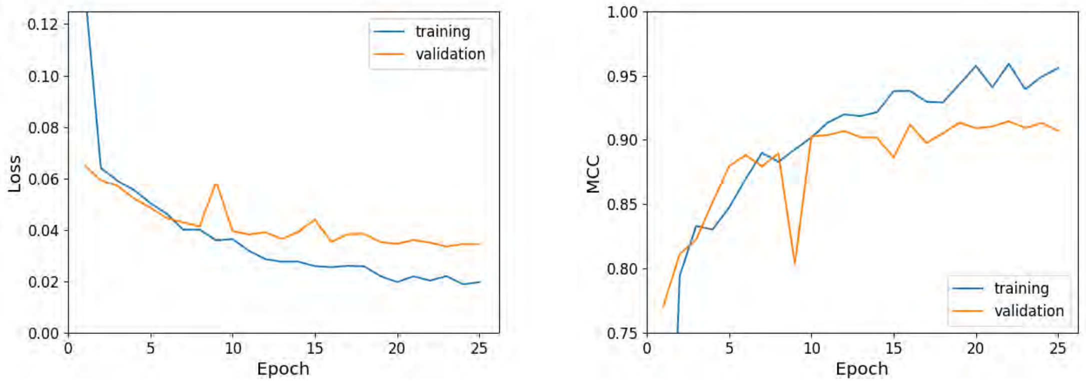


Figure 6.11: The loss (left) and MCC (right) of model B from Figure 6.10. The gap between training and validation performance is smaller than for its equivalent without dropout from Figure 6.9, but this is mainly due to a lower training performance. On the other hand, the validation loss does not tend to increase here.

Applying fine-tuning. The fact that regularization only reduces the training performance for the previous models indicates that, in order to make it generalize better, we should: (i) collect more data, (ii) make our model more complex (e.g. by adding more layers), or (iii) make the current model more flexible by *fine-tuning* also the convolutional part of the network. Since collecting more data is not really contributing to the goals of this research (as we do not need to deliver a deployable tool) and we prefer to stick to our current architecture, we choose to try fine-tuning the full model. This increases again the risk of overfitting, so we probably need strong regularization in this case. We therefore use dropout with probability $p = 0.25$ of keeping a node. We further use the same specifications as for the previous model, with the only other difference that we use batch size $B = 64$ for implementation purposes (we get memory issues if we use a larger batch size). Figure 6.13a shows that this model (C) is quite unstable and even diverges after some time. A new model (D) with dropout ratio $p = 0.5$ (so weaker regularization), eventually diverges as well, but it gives a very good validation MCC of 0.9377 after 6 epochs. Hence, we decide to retrain this model again from this point onwards, but now with a lower relative update $r^* = 10^{-3}$. Figure 6.13b shows that this results in a more stable model, but further improvements using fine-tuning seem hard to achieve. Nevertheless, we note that the validation MCC after 6 epochs is slightly higher than we achieved earlier with model A (0.9377 versus 0.9328). However, for model D the peak

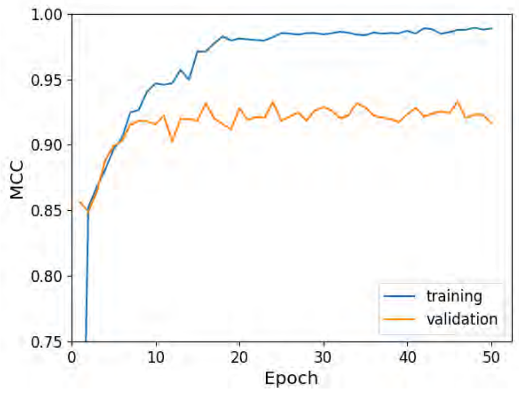
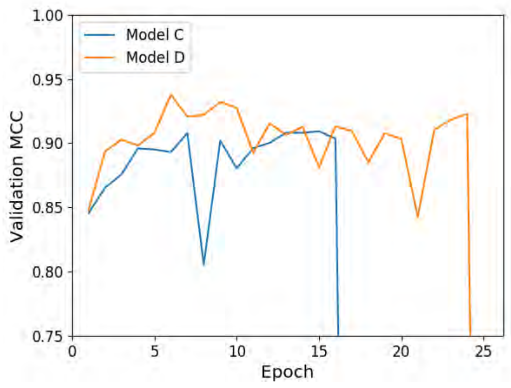
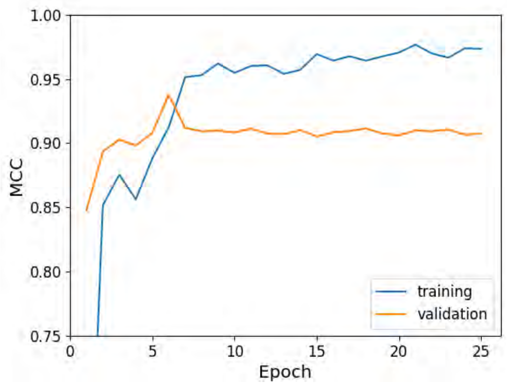


Figure 6.12: The MCC of model A (recall Figure 6.10). The validation MCC has its peak after 46 epochs at 0.9328, where the training MCC is 0.9876.

in validation MCC can also be partly due to luck, since it is ‘much’ higher here than in any other epoch and even higher than the training MCC at that point as well. Since the performance of model A looks more reliable in that sense (see Figure 6.12), we argue that this model is more likely to generalize better. Hence, we choose model A (after 46 epochs of training) as our final model for recognizing car damage.



(a) Models C and D.



(b) Model D retrained after 6 epochs.

Figure 6.13: The MCC of models C and D with respective dropout rates $p = 0.25$ and $p = 0.5$ (left), and the retrained variant of model D after 6 epochs (right) with a lower learning rate.

Results. The confusion matrix in Figure 6.14 shows the test performance of the final model as described above. From this figure, we can extract that the test MCC is 0.9037. This scales to a 95.2% performance, which is beyond our expectations. Despite that only 5.8% of our images contain *damaged* cars, our model has a recall of 89.6%, meaning that it recognizes this percentage of the damaged cars as such. The test accuracy is 99.0%, but, as mentioned before, this is not a very reliable metric here as always predicting ’undamaged’ would already give 94.2% accuracy. Looking at some examples of wrongly predicted images (see e.g. Figure 6.15), we observe that the dataset contains images that are hard to predict (sometimes even for humans). These include, for instance, images of oddly posed, shaped or illuminated cars and images with overlapping advertisements. Supposing that insurance companies are able to obtain a much better image dataset, both in size and quality, we believe that our approach has the potential to create models that perform well enough for deployment. Table 6.1 summarizes all experiments performed in this subsection.

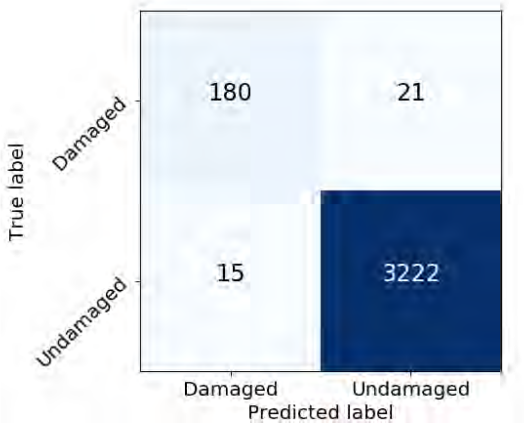
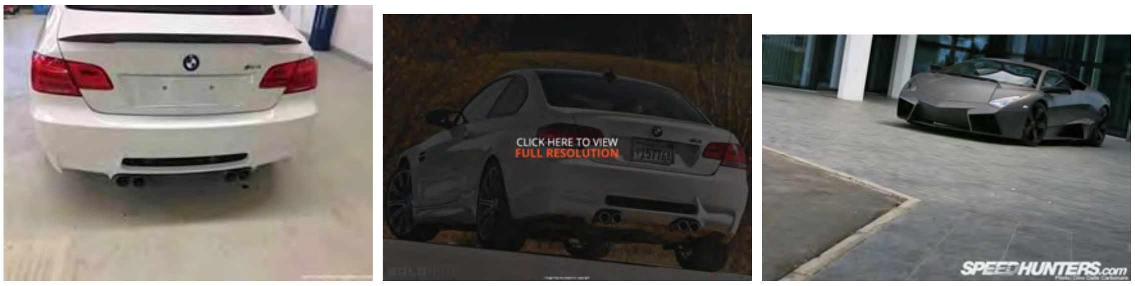


Figure 6.14: Confusion matrix representing how well our final model predicts whether the cars on the test images are damaged or not.

Table 6.1: Summary of the experiments. More details of the learners are specified in the text.

	Attempt 1	Attempt 2	Attempt 3	Model A	Model B	Model D
epochs (k)	25	25	25	50	50	25
batch size (B)	64	64	128	128	128	64
average relative update (r^*)	10^{-3}	10^{-3}	10^{-2}	$10^{-2}/10^{-3}$	$10^{-2}/10^{-3}$	10^{-2}
initial learning rate (α)	10^{-4}	10^{-4}	10^{-3}	10^{-3}	10^{-3}	10^{-3}
dropout (p)	1	1	1	1/0.5	0.5	0.5
zero-centering	×	✓	✓	✓	✓	✓
fine-tuning	×	×	×	×	×	✓
validation MCC	0.7989	0.807	0.9218	0.9328	0.9188	0.9377



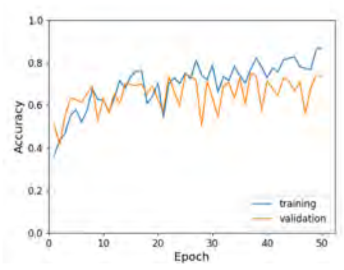
(a) Missing license plate. (b) Advertisement. (c) Odd pose.

Figure 6.15: Some undamaged labelled images that have been predicted as damaged car images. This illustrates that our dataset contains images that are hard to predict (for varying reasons).

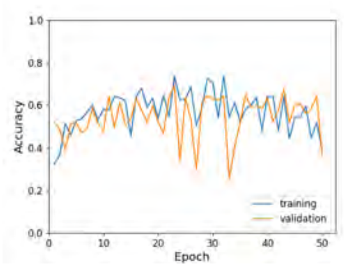
6.3 Classifying, localizing and quantifying damage

Initial model. Since we have only 504 training images for the final three tasks (recall Table 5.3), we assume that fine-tuning is not an option here. Moreover, we probably need strong regularization even when we do not fine-tune the convolutional part of our models. To start with, we use dropout with $p = 0.5$ in the first fully-connected layer of our usual architecture with only two fully-connected layers. We apply both zero-centering and scaling on the input data, we use batch size $B = 64$ and we aim at a relative update of $r^* = 10^{-2}$ with initial learning rate $\alpha = 10^{-3}$. Because of the limited data, an epoch of training can be done much faster now, but we also need to train for more epochs to converge. Initially, we therefore train all learners for 50 epochs here, which can be done in around 15 minutes per learner. Since the data is not extremely unbalanced, the accuracy metric suffices for now.[11] From Figure 6.16, we can conclude that the results are not (yet) as desirable, and that using a higher batch size $B = 128$ and a lower update ratio $r^* = 10^{-3}$ with initial learning rate $\alpha = 10^{-4}$ may stabilize the learning process, hopefully leading to higher performances. These changes indeed turn out to have the desired effects, so we continue training these models for a total of 500 epochs, giving the results as shown in Figure 6.17.

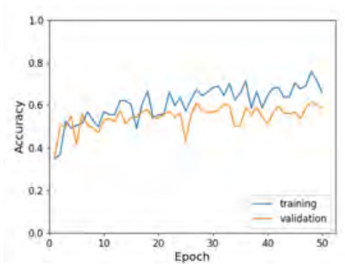
[11] In practice, other metrics are would be preferred in case we have under-represented categories (such as damage on 'top' of the car here). For example, generalizations of the MCC and F_1-score exist for multi-class problems.



(a) Type.

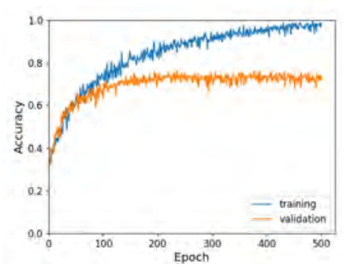


(b) Location.

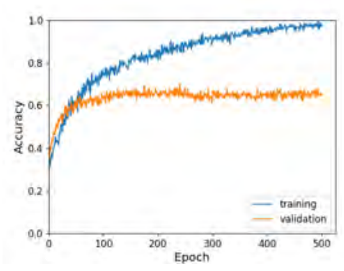


(c) Size.

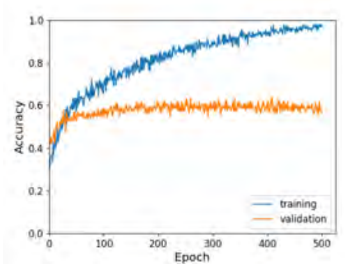
Figure 6.16: The accuracy of the initial model for classifying the type, location and size of the car damage. All plots are very unstable, so we may need a higher batch size and/or use a lower learning rate.



(a) Type.



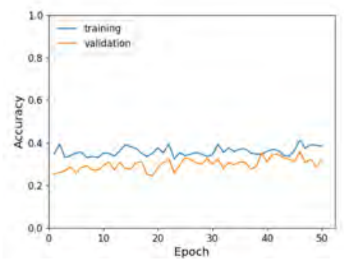
(b) Location.



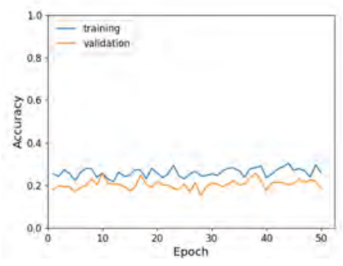
(c) Size.

Figure 6.17: The accuracy of the initial model for classifying the type, location and size of the car damage, but with higher batch size and lower learning rate than in Figure 6.16. The validation accuracy for classifying the damage type peaks at 76.2% (after 235 epochs), for the location at 69.9% (after 240 epochs), and for the size at 63.7% (after 408 epochs).

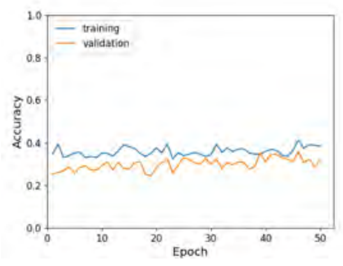
Adding regularization. We observe from Figure 6.17 that the previous model is able to fit the training data very well, but without generalizing well enough to also achieve high validation performance (although the results are already quite good regarding the amount of available data). Therefore, the only experimenting we will do here is with the amount of regularization. As a first attempt, we add batch normalization in the fully-connected layers, but the results in Figure 6.18 indicate that this is not a successful approach here. Apparently, this batch normalization deforms the transferred features from the convolutional part in such a way that they lose a large part of their information. We have to be careful with generalizing this result, but it at least suggests that, although batch normalization can be a good regularizer when training models from scratch, it is not always directly applicable when transferring models for which batch normalization was not part of the original architecture. Hence, we choose to try dropout with a stronger rate $p = 0.25$, keeping the rest of the model specifications equal to the model corresponding to Figure 6.17. The results in Figure 6.19 show smaller gaps between training and validation accuracy, but only for the type and location of damage the validation accuracy (slightly) increases using stronger dropout.



(a) Type.

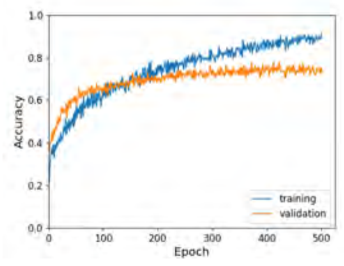


(b) Location.

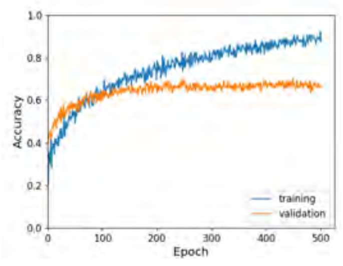


(c) Size.

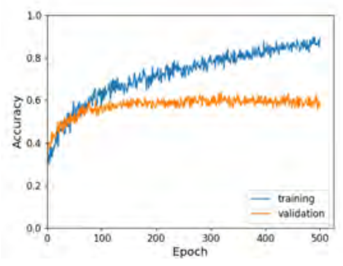
Figure 6.18: The accuracy of a similar model for classifying the type, location and size of the car damage as in Figure 6.17, but with an extra fully-connected layer and batch normalization in the first two fully-connected layers. We needed to use a lower batch size $B = 64$ here to prevent memory issues again.



(a) Type.

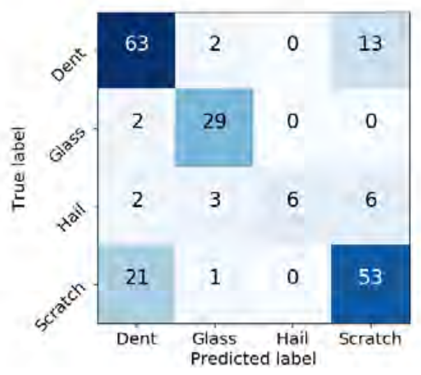


(b) Location.



(c) Size.

Figure 6.19: The accuracy of a similar model for classifying the type, location and size of the car damage as in Figure 6.17, but with dropout rate $p = 0.25$. The validation accuracy for classifying the damage type peaks at 77.7% (after 412 epochs), for the location at 70.3% (after 449 epochs), and for the size at 63.3% (after 318 epochs).



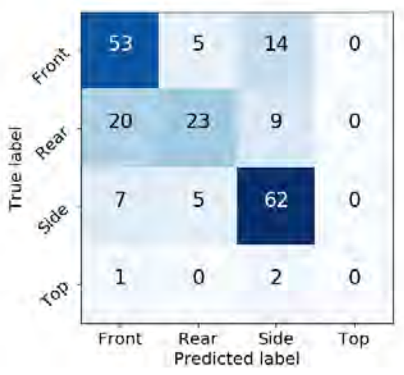


Figure 6.20: Confusion matrix representing how well our model predicts the *type* of car damage.

Figure 6.21: Confusion matrix representing how well our model predicts the car damage *location*.

Results. Figures 6.20 to 6.22 show the confusion matrices resulting from evaluating the test data. For all three tasks, we use the respective models with dropout rate $p = 0.25$ as described in Figure 6.19. Although the model with $p = 0.5$ has a slightly higher validation accuracy for predicting the size, we expect the one with $p = 0.25$ to generalize better in this case as well, since it overfits less. For these multi-class problems, we can compute the accuracy by counting the number of correctly predicted examples and divide it by 201, the size of our test set. This results in that our model predicts the car damage type, location and size with respectively 75.1%, 68.7% and 54.2% accuracy. Apparently, there is a big difference between the difficulties of these three tasks. This can be explained by the fact that the different types of damage are more discriminative than that of the size. For example, the difference between a dent and broken glass is very clear, whereas the boundary between medium and small damages is hard to define. In the latter case, there is also a bias caused by subjectivity in the manual labelling process. This is the case for more categories, such as dents and scratches, which often appear together, while we can only assign a single label to each example. Also, it is hard to define the boundaries between a damage at the side of a car and one at the front or rear. These biases clearly come back in the confusion matrices as well. The only pair of categories that turns out difficult to distinct without having this subjectivity bias, is damages at the front versus ones at the rear of a car, which is also not surprising since these can look rather similar. Furthermore, Figure 6.23 give some examples of incorrectly predicted images. The left one is just dented, showing a human mistake in labelling the image. The middle one has a large scratch at the side (which may be hard to see), but the model predicts 'rear'. This indicates that the model may not predict the location of the damage itself, but rather the view of the car on the image. The right one is an example of the described subjectivity issue, since we could also label this image as 'medium' damage. Despite of these issues, the results give good hope that ConvNets can potentially perform well on these tasks when there is more and better data.

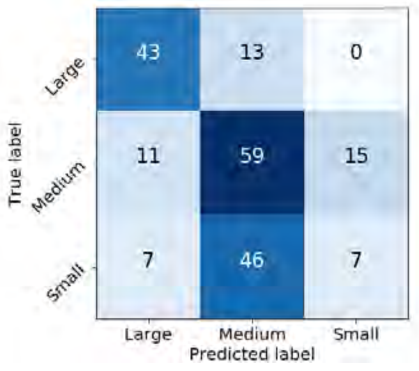
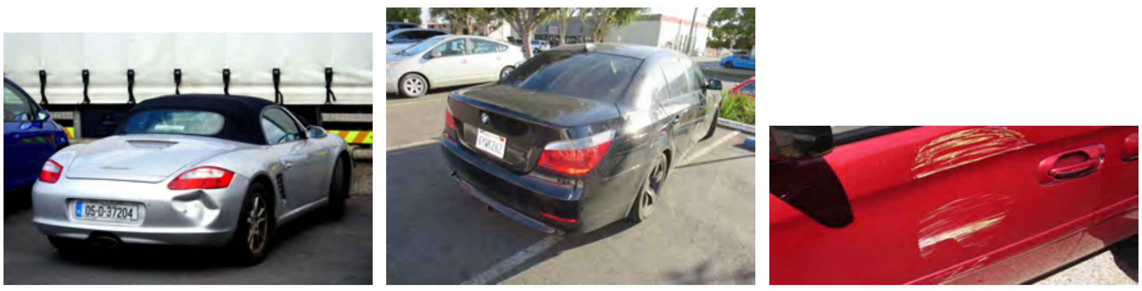


Figure 6.22: Confusion matrix representing how well our model predicts the *size* of a car damage.



(a) Type: hail. (b) Location: side. (c) Size: large.

Figure 6.23: Some examples with their original label that have been predicted incorrectly.

7 Conclusions and recommendations

7.1 Conclusions

Recognizing car damage. We have taken some good first steps for the application of ConvNets in the context of car damage recognition. Although recognizing cars has been done before, we showed that it is relatively easy to achieve very good results here (99.0% accuracy) using transfer learning. Recognizing car damage is a bit more difficult, partly because pre-trained models are usually not trained on doing this task, but mainly because our image dataset for damaged cars was very limited in size. Nevertheless, we achieved a Matthews correlation coefficient of 0.904 using our approach with transfer learning VGG16, so this has great potential to give excellent performance when the dataset is richer, both in size and quality. In that sense, our mission to show the potential of ConvNets in this context can be considered a success. We have even attempted to take one step further: we classified the type, location and size of the damage with respectively 75.1%, 68.7% and 54.2% accuracy. This is much better than random (which would give around 33% accuracy in all cases), but large improvements have to be made here in order to get results that are good enough for deployment. A larger and better image dataset is expected to enable the training of much better models. Preferably, such a dataset should contain images that are actually used for car damage claims, with labels representing more exact classifications of the damage. Not only would this heavily reduce the subjectivity bias in the labels, but it would also allow more detailed classifications than the few classes we used in our experiments.

Influence of hyper-parameters. We have seen that we can benefit from adapting the hyper-parameters in a theoretically founded way, based on diagnostics. This allowed us to obtain good results with a limited number of experiments. Especially, good specifications of the learning rate, batch size and the amount of regularization can be determined efficiently by keeping track of the loss and metric function during training. The smaller the learning rate and the larger the batch size, the more stable the learning process is. On the other hand, a small learning rate does not necessarily give better results and a large batch size is more computationally expensive. Dropout is a very strong regularizer, but in future research it may also be interesting to try using dropout alongside a (L_2) regularization term. Batch normalization is probably only useful when training a ConvNet from scratch or when the transferred model also used this during training. Fine-tuning transfered parameters can give great results according to literature, but this was not the case for us due to our limited dataset. Finally, we did not observe the added value of zero-centering the data beforehand, while scaling was very important. When using transfer learning, our advice would be to use the same types of data preprocessing as for pre-training the transfered model. Besides the mentioned hyper-parameters, it may be interesting to experiment with other model architectures and optimization algorithms in future work as well.

7.2 Recommendations

Adding bounding boxes. Although we have attempted to give a few classifications of the damage, our model for recognizing damage is still a bit of a 'black box', because it does not give an indication of the reason why it gives a certain prediction. A possible approach for improving the interpretability of the model is to add *bounding boxes* to our predictions. That is, when the model predicts that the car on an image is damaged, we can add a box to indicate what rectangular area of the image causes the model to predict 'damaged'. This can be done relatively easy by evaluating this image multiple times, each time making another rectangular area of the image black. When the model changes its prediction to 'undamaged', we know that the presumed damage lies in this covered area. Bounding boxes are also an additional element in the ILSVRC (since 2013 this task is referred to as *object detection*), so inspiration of how this exactly works can be obtained, for example, from the work of its best contestants again.

Combining ConvNets with Jayawardena's work. Although we are not able to replicate the full method of Jayawardena (2013) without the availability of 3D CAD models, we are able to replicate the reflection detection part of their method. As mentioned in Section 2.1, this procedure at least succeeds in classifying actual damages as edges on the surface of vehicle panels, which means that all image edges that are classified as inter-object reflection are (almost) certainly not damage. A possible approach in future research may therefore be to first apply local smoothing on images around edges that are classified as reflection, before using these images as input for ConvNet learners. This can be seen as a pre-processing operation that may help preventing the learner to incorrectly classify reflections as damage.

Ensemble learning. We included a subsection on ensemble methods, because we expect that they can further improve the results we achieved so far. We have not been able to apply these methods yet, but especially boosting is intuitively a good addition to our approach as it can stimulate the learning of specific patterns that do not appear frequently, but that do need to be recognized as damage. Future work can probably benefit from such methods.

Embrace and take advantage of the ample opportunities. Transfer learning provides great opportunities for applying ConvNets to a wide variety of problems in practice. We have (once again) demonstrated the only requirements here are that the right resources and sufficient data are available. When, for example, clients in the car insurance industry are willing to cooperate in providing primarily the required data, then PwC now has gained valuable knowledge and tools for doing very interesting projects in increasing the efficiency and quality of, among else, parts of the car damage claim handling process.

References

Batista, G. E., R. C. Prati, and M. C. Monard (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter 6*(1), 20–29.

Boureau, Y.-L., F. Bach, Y. LeCun, and J. Ponce (2010). Learning mid-level features for recognition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 2559–2566. IEEE.

Breiman, L. (1996). Bagging predictors. *Machine learning 24*(2), 123–140.

Chawla, N. V. (2009). Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pp. 875–886. Springer.

Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research 12*(Aug), 2493–2537.

Control€xpert (2015). EasyClaim - claims settlement in no time. Available: www.controlexpert.com [Last accessed: 16 January 2018].

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*(4), 303–314.

Dahl, G. E., T. N. Sainath, and G. E. Hinton (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8609–8613. IEEE.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE.

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM 55*(10), 78–87.

Domingos, P. (2017, October). Lecture notes in data mining / machine learning. Week 1: Introduction. *Paul G. Allen School of computer science & engineering*. Available: www.courses.cs.washington.edu [Last accessed: 5 December 2017].

Freund, Y. and R. E. Schapire (1995). A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pp. 23–37. Springer.

Gorodkin, J. (2004). Comparing two k-category assignments by a k-category correlation coefficient. *Computational biology and chemistry 28*(5-6), 367–374.

Graves, A., A.-r. Mohamed, and G. Hinton (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645–6649. IEEE.

Griffin, G., A. Holub, and P. Perona (2007). Caltech-256 object category dataset.

He, K., X. Zhang, S. Ren, and J. Sun (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

He, K., X. Zhang, S. Ren, and J. Sun (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hinton, G., O. Vinyals, and J. Dean (2014). Dark knowledge. *Presented as the keynote in BayLearn 2*.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hoffer, E., I. Hubara, and D. Soudry (2018). Fix your classifier: the marginal value of training the last weight layer. In *International Conference on Learning Representations*.

Hu, J., L. Shen, and G. Sun (2017). Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*.

Huang, G., Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger (2016). Deep networks with stochastic depth. In *European Conference on Computer Vision*, pp. 646–661. Springer.

ImageNet (2017). ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Available: www.image-net.org [Last accessed: 16 January 2018].

Intel Labs (2016). Bringing parallelism to the web with River Trail. Available: www.intellabs.github.io [Last accessed: 24 February 2018].

Ioffe, S. and C. Szegedy (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456.

Jay, P. (2017). Back-propagation is very simple. Who made it complicated? Available: www.becominghuman.ai [Last accessed: 28 February 2018].

Jayawardena, S. (2013). Image based automatic vehicle damage detection.

Johnson, J. (2013, March). General regression and over fitting. Available: www.shapeofdata.wordpress.com [Last accessed: 18 December 2017].

Judd, J. S. (1990). 4. The intractability of loading; 5. Subcases. In *Neural network design and the complexity of learning*, pp. 38–55. MIT press.

Karpathy, A. (2017, Spring). Course notes of convolutional neural networks for visual recognition. *Stanford University class CS231n*. Available: www.cs231n.github.io [Last accessed: 8 December 2017].

Keras (2018). Keras: The python deep learning library. Available: www.keras.io [Last accessed: 28 February 2018].

Kingma, D. and J. Ba (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koza, J. R., F. H. Bennett, D. Andre, and M. A. Keane (1996). Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design'96*, pp. 151–170. Springer.

Krause, J., M. Stark, J. Deng, and L. Fei-Fei (2013). 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation 1*(4), 541–551.

Lin, Y., F. Lv, S. Zhu, M. Yang, T. Cour, K. Yu, L. Cao, and T. Huang (2011). Large-scale image classification: fast feature extraction and svm training. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pp. 1689–1696. IEEE.

Matthews, B. W. (1975). Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure 405*(2), 442–451.

Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM 38*(11), 39–41.

Nair, V. and G. E. Hinton (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, Volume 269, pp. 543–547.

PwC (2015, January). Onze juridische structuur. Available: www.pwc.nl [Last accessed: 16 January 2018].

Python (2018). Python. Available: www.python.org [Last accessed: 28 February 2018].

Ramachandran, P., B. Zoph, and Q. V. Le (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*.

Raschka, S. (2014). Confusion matrix. Available: www.rasbt.github.io [Last accessed: 24 January 2018].

Razavian, A. S., H. Azizpour, J. Sullivan, and S. Carlsson (2014). Cnn features off-the-shelf: an astounding baseline for recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pp. 512–519. IEEE.

Ripley, B. D. (1996). *Pattern recognition and neural networks*, pp. 354. Cambridge university press.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning representations by back-propagating errors. *nature 323*(6088), 533.

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision 115*(3), 211–252.

Sánchez, J. and F. Perronnin (2011). High-dimensional signature compression for large-scale image classification. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pp. 1665–1672. IEEE.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks 61*, 85–117.

Shearer, C. (2000). The crisp-dm model: the new blueprint for data mining. *Journal of data warehousing 5*(4), 13–22.

Sill, J., G. Takács, L. Mackey, and D. Lin (2009). Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*.

Simonyan, K. and A. Zisserman (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Sollich, P. and A. Krogh (1996). Learning with ensembles: How overfitting can be useful. In *Advances in neural information processing systems*, pp. 190–196.

Springenberg, J. T., A. Dosovitskiy, T. Brox, and M. Riedmiller (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research 15*(1), 1929–1958.

Steiner, J. (2013). The GIMP documentation. Available: www.docs.gimp.org [Last accessed: 25 February 2018].

SURFsara (2017). Description of the Lisa system. Available: www.surfsara.nl [Last accessed: 29 January 2018].

Sutskever, I. (2013). Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*.

SV-Europe (2016). What is the CRISP-DM methodology? *Smart Vision Europe*. Available: www.crisp-dm.eu [Last accessed: 5 December 2017].

Szegedy, C., S. Ioffe, V. Vanhoucke, and A. A. Alemi (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, Volume 4, pp. 12.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2014). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna (2015). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826.

TensorFlow (2018). TensorFlow: an open-source machine learning framework for everyone. Available: www.tensorflow.org [Last accessed: 28 February 2018].

Ueda, N. and R. Nakano (1996). Generalization error of ensemble estimators. In *Neural Networks, 1996., IEEE International Conference on*, Volume 1, pp. 90–95. IEEE.

Wikipedia (2017). Cross-industry standard process for data mining. Available: www.wikipedia.org [Last accessed: 27 February 2018].

Wirth, R. and J. Hipp (2000). CRISP-DM: Towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, pp. 29–39.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks 5*(2), 241–259.

Yang, S. and A. Browne (2004). Neural network ensembles: combining multiple models for enhanced performance using a multistage approach. *Expert Systems 21*(5), 279–288.

Yosinski, J., J. Clune, Y. Bengio, and H. Lipson (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pp. 3320–3328.

Zadeh, R. (2016, November). The hard thing about deep learning. Available: www.oreilly.com [Last accessed: 12 December 2017].

Zeiler, M. D. and R. Fergus (2013). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833. Springer.