Neural network based image classification in the context of archaeology

Simon Hohl

546999

MASTER'S THESIS

in

International Media and Computing (Master)

HTW Berlin

March 2016

Supervisors:

Prof. Dr.-Ing. Kai Uwe Barthel

Prof. Dr. Klaus Jung

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Berlin, March 29, 2016

Simon Hohl

Contents

Abstract

This thesis aims to evaluate the application of artificial neural networks on archaeological images for classification purposes. The archaeological object database Arachne of the German Archaeological Institut and the University of Cologne functions as an image source for the evaluation [15].

The analysis is split into two major parts.

One part analyses how well artificial neural networks can be trained with the domain specific images provided by *Arachne*. Because the training of neural networks on larger image datasets is computational quite expensive it requires a corresponding infrastructure. For cases where said infrastructure is missing, the thesis will additionally evaluate techniques that are using already trained networks in order to build more lightweight classifiers for one's own data.

Chapter 1

Introduction

1.1 Motivation

The *Arachne* database is the archaeological object database of the *German Archaeological Institute* (DAI) and the *University of Cologne*. It exists since January 1995, was originally created at the *Cologne Digital Archaeology Laboratory* (CodArchLab)[15] and catalogues the findings of a wide range of different archaeological projects. Its data is structured by different kinds of objects. Besides a multitude of meta information, most objects feature one or multiple images. The objects can be searched by type, location, date, literature, material and many other categories.

Today, the database contains roughly 2.2 million images. These show for example landscapes, excavations, ruins, intact buildings or cities, aerial views, drawings, ground plots, sculptures or pottery to just name a few. Most are part of scientific projects that digitized their photo or reversal film archives or their print publications.

Despite being such a big image collection, so far – besides publishing and curating objects and images – there has been few emphasis on working on the image data itself. This thesis aims to change that by utilizing the capabilities of artificial neural networks in order to create programs that are able to do automatic image classification.

Artificial neural networks mimic the biological neural networks found in the human (or animal) brain and are just one technique in the wider field of machine learning. The basic idea of machine learning is to find structure in- or make predictions about unknown data. More specifically, in machine learning this is not achieved by explicitly defining each step in the prediction or analysis process, but by creating a framework in which the computer is able to learn and find solutions to the problem on its own, using a set of training data.

For classification problems using neural networks this means, that the neural network is presented with a set of training images with class labels

attached to them. Iteratively, the neural network can then be trained to reproduce the given labels by just 'looking' at the image data – basically learning to extract the abstract concepts defining each class out of the image data. The knowledge about these concepts can then be applied to new, unknown image data without any further training. Thanks to improved algorithms and the enhanced processing power of modern GPUs, great advances in image classification have been made using artificial neural networks in the recent years [11, p.4].

Training a neural network using bigger datasets is quite computational expensive, and may require additional hardware infrastructure in order to be done in a reasonable timespan. Letting an already trained network make predictions on the other hand can be done on less powerful machines.

This thesis aims to analyse the possibilities, limits and difficulties when using artificial neural networks for image classification in the specific domain of archaeology. Because the necessary infrastructure for training artificial neural networks may not be at hand for everybody interested in doing image classification, this thesis will analyse two approaches for creating classifiers. The first approach is the standard training procedure for neural networks, the second will utilize parts of an already trained neural network in order to create classifiers for one's own data.

1.2 Overview

The following chapter 2 will first introduce the theoretical basics necessary for the further analysis. It starts out with an overview over the broader field of machine learning, which artificial neural networks are just a part of. This is followed by a more in depths introduction of artificial neural networks themselves. Chapter 3 serves as a short introduction to the different frameworks and technologies used in the analysis. The concept for evaluating the different classification approaches is described in chapter 4, and their implementation in chapter 5. The experimental results for the different classifiers are presented and discussed in chapter 6. Finally, chapter 7 reflects on the analysis and and provides an outlook on possible further research.

Chapter 2

Machine Learning

This chapter aims to introduce those machine learning principles used in the following analysis, but is highly selective and not supposed to be read as an overview of machine learning in general. For a more general overview of existing machine learning techniques please resort to [1], [7], [8], [9] and [10].

Machine learning is a collective term for a wide range of methods that try to make sense of or find new meaning in data. Instead of predefining hypotheses about structures or patterns in existing data, the machine learning approach is to create programs that are able to learn or to create their own hypotheses, hereinafter called **models**. The algorithms learn by using provided training data. Each item of training data is defined by a selection of **features**, also called attributes or covariates.[9, p. 2]

> In the simplest setting, each training input x_i is a D-dimensional vector of numbers, representing, say, the height and weight of a person. [..] In general, however, x_i could be a complex structured object, such as an image, a sentence, an email message, a time series, a molecular shape, a graph, etc.[9, Murphy,p. 2]

Murphy's listing of possible features already shows the diverse area of applications in which machine learning algorithms are used. There are two major types of machine learning: Supervised and unsupervised learning.

2.1 Supervised learning

For supervised learning, a training dataset is given where each item contains input data $\vec{x}$, representing the aforementioned features, and **target values** $\vec{t}$ [9, p. 2]. The training goal for the algorithm is to find a model that is able to predict the target values for all data or at least to minimize the error between its **predicted values** $\vec{y}$ and the target values $\vec{t}$. In machine learning, this

error is also called **loss** or cost. Once the training is finished, the algorithm is supposed to be able to use its learned model to make predictions for new, unknown input features [7, p. 6].

Applied to this analysis, the input training set $\vec{x}$ would be comprised of images imported from *Arachne*, their features would be their dimensions and their RGB colour values. Additionally, each image will have a category label t_i assigned as the target value for the model. The different machine learning methods implemented later will aim to minimize the loss between their predictions $\vec{y}$ and the predefined target categories.

2.1.1 Types of supervised learning

Classification

Supervised learning tasks that try to predict discrete values or categories, like the one described above, are called classification tasks. There is a distinction between binary classification (the image shows either a building or it does not), multiclass classification (the image shows either a building or a sculpture or pottery) and multi-label classification (the image shows both a building and a sculpture, but no pottery) [9, p. 3]. The type of classification done in this analysis will be multiclass classification, but it will also be shown that for archaeological images it may make sense to extend research to multilabel classification techniques.

Regression

An alternative to classification in supervised learning is called regression. For regression tasks, the training targets $\vec{t}$ are not interpreted as discrete but continuous values [9, pp. 8/9].

To again exemplify the idea using the *Arachne* data: The training set may consist of images $\vec{x}$, but this time each with their date of origin assigned as prediction target t_i. A regression algorithm would again search for a model that is able to predict the dates for all image data as predictions $\vec{y}$. But if successful, the final model would then be able to predict the dates for unknown images, even *in between* those dates that were originally trained with.

2.1.2 Supervised learning classifiers

Naive Bayes classifier

The *Naive Bayes* classifier is based on the *Bayes theorem*. The *Bayes theorem* itself will not be discussed in the following section, for further background please refer to [8, p. 154 ff.] As already discussed in section 2.1.1, for classification tasks the goal is to find a model that predicts which category c

out of a set of possible categories C is at hand for a new and unknown data instance D. The *Naive Bayes* classifier is often used in information retrieval [6, p. 238][8, p. 180].

The basic assumption of the *Naive Bayes* classifier is that in order to make a prediction for item D, comprised of K features $d_1 \dots d_k$, the probability of D belonging to a certain category can be calculated as the product of the probabilities for the individual features $d_1, d_2, \dots, d_k$ belonging to said category. This means the algorithm implicitly assumes a general independence between all the features. The probabilities for each feature are calculated using a set of training data. The *Naive Bayes* classifier c_{NB}[8, p. 177] is defined as

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{i=1}^{K} P(d_i|c). \tag{2.1}$$

$P(c)$ denotes the probability that the category c will be observed in the training data in general and, following Manning et al. [6, p. 240], is calculated as

$$P(c) = \frac{\text{Number of all training instances with category } c}{\text{Number of all training instances}}. \tag{2.2}$$

$P(d_i|c)$ on the other hand denotes the probability that the input's feature value d_i is relevant for category c in the training data. This probability is calculated by multiplying the input's feature d_i by a factor f_i^c. f_i^c is representing the given feature's importance for the category in the training data defined as

$$f_i^c = \frac{v_i^c}{V^c} \tag{2.3}$$

where v_i^c represents the summed values of feature i for category c and V^c is the sum of all feature values for c in the training data [6, p. 240].

Calculating the probabilities with products of fractions like this yields problems for cases where $v_i^c = 0$, causing the single feature to dominate the whole product (resulting in a overall probability value of 0 for the category) [6, p. 240]. In order to cope with this, implementations of the *Naive Bayes* use *Laplace Smoothing*

$$P(d_i|c) = \frac{v_i^c + 1}{V^c + K} \tag{2.4}$$

where K is the amount of features as before [6, p. 240].

Another problem when implementing the classifier is the fact that if there are a lot of features (a high K), a lot of potentially very small values are multiplied. In order to prevent floating point underflow, the logarithm instead of the product is calculated [6, p. 239]. The final classifier is then defined by Manning et al. as

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} \log\left(P(c)\right) \sum_{i=1}^{k} \log\left(P(d_i|c)\right). \tag{2.5}$$

k-nearest neighbours classifier

The *k-nearest neighbours* (KNN) algorithm is an instance based classification algorithm, meaning the classifier is not really trained but instead all training data is simply stored and evaluated every time a new prediction is made [8, pp. 230/276].

The algorithm interprets the training data's features as points in multi dimensional space. For a new, unknown data point that has to be classified, KNN searches the k points of training data closest to the input by calculating the Euclidean distance [8, p. 232]. The class predominant between all k neighbours is the one assigned to the unknown data point. The assigned class can change for different values of k using the same training data, as shown in figure 2.1 [8, p. 232]. It is possible to additionally distance-weight the classification results, given the assumption that neighbours closer to the classified point are more relevant [8, p. 233].

Because all processing has to be done for each individual classification, KNN is computational quite expensive. Training cost per classification is $\Theta(1)$, while for testing it is Θ(feature dimensions $\times$ training set size) [6, p. 276].

Artificial Neural Networks

As already mentioned in the introduction, artificial neural networks can also be used for classification tasks and will be explained in more detail in section 2.4.

2.2 Unsupervised learning

In contrast to supervised learning, unsupervised subsumes learning methods that try to find patterns and relations in unstructured data without being given explicit target values. Another way to put it is that compared to supervised classification tasks, some unsupervised methods can be used to to find *possible* categories instead of learning to discern predefined ones.

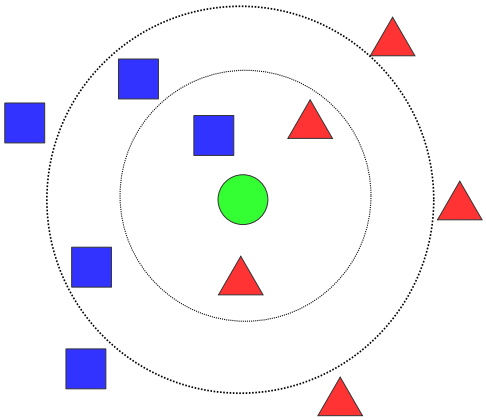


Figure 2.1: Example for the usage of *k-nearest neighbours* in order to classify training and input data with two dimensional feature vectors. Choosing different k for *k-nearest neighbours* may result in different classifications. For a neighbourhood of $k = 3$ the input would be classified as a triangle, for $k = 5$ as a square. Inspired by [8, p. 233]

2.2.1 k-Means clustering

Clustering algorithms are a variant of unsupervised learning algorithms and in general try to divide the given unstructured data into subsets. One of the most common clustering algorithms is *k-Means* [6, p. 331]. Again interpreting the data as a collection of multidimensional feature vectors, *K-means* initializes K cluster centres (also called centroids) in the feature space and tries to minimize the average squared Euclidean distance between the centroids and the data points.

Following Manning et al., the centroids are initialized at the positions of randomly selected training data points. Iteratively, all data points are first assigned to their closest centroid and then all centroids' positions are moved to the mean center of all their assigned data points. The conditions for stopping the algorithm can differ from application to application, but having at least defined a maximum amount of iterations is recommended by Manning et al.. Other criteria can either be that the centroids stopped moving or a given minimum distance threshold was reached. For a more detailed introduction please refer to Manning et al [6, p.332-334].

2.3 Deep learning

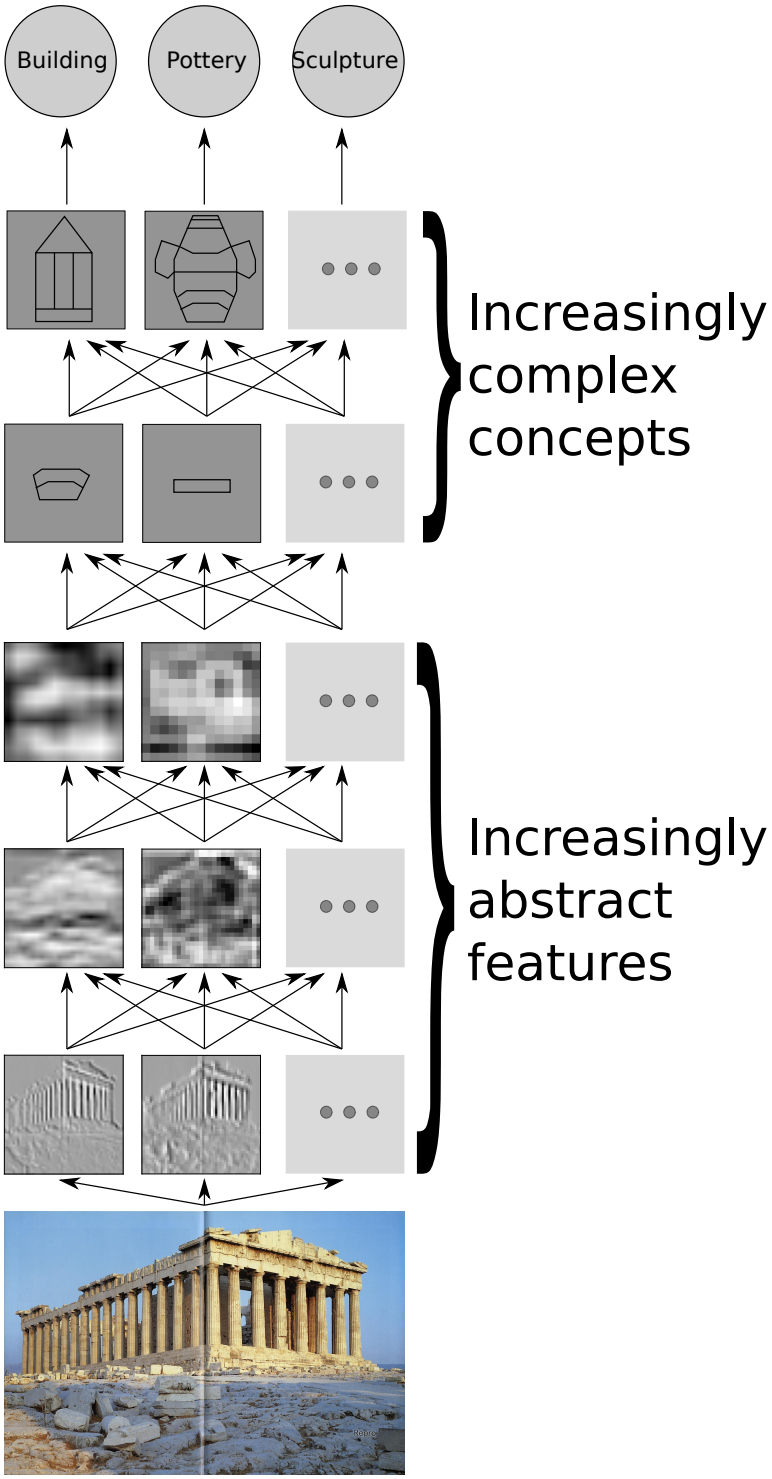
Deep learning can be applied to both supervised and unsupervised learning. In their introduction to deep learning, Goodfellow et al. describe how machine learning applications initially struggled with tasks that humans solve

intuitively. For instance, if asked about an image's content, humans have no problem discerning between relevant and irrelevant parts of the image. This intuitive decision is based on previously learned abstract features about parts and components of objects. Goodfellow et al. describe the problem as follows:

> For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.[3, p. 3]

As a consequence, for a wide range of machine learning tasks, the question of how to provide these abstract features for the algorithms arose. As described by Goodfellow et al., hand crafting these features may be too complex and time consuming in many cases. Problems like this are solved by deep learning techniques, which are able to *automatically* break down the input into increasingly abstract features, and then to recombine those features into more and more complex concepts that can finally be used for reasoning or decision making. Figure 2.2 shows an example of how a potential deep learning algorithm would process images depicting archaeological objects.

Figure 2.2: Deep learning for image classification: The original image gets broken down into abstract features, which are then used to construct concepts. Based on those concepts, the classification decision is made. Figure inspired by Goodfellow et al. [3, p. 6]



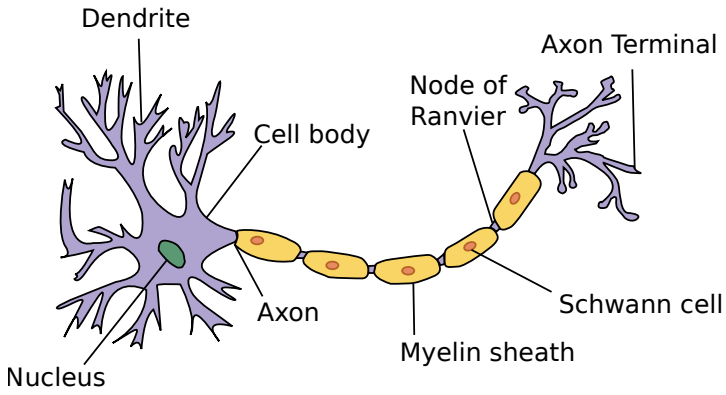


Figure 2.3: Structure of an biological neuron. Figure taken from *Wikimedia Commons*, created by user *Dhp1080* [25].

2.4 Artificial Neural Networks

The analysis done in the following chapters makes use of deep learning, realised with artificial neural networks. The basic concepts for artificial neural networks are quite old and can be traced back to the 1960s, but have only recently started to come to general attention by finally outperforming other machine learning algorithms in a range of different applications due to further improvements and the increased performance of modern GPUs [11, p. 4].

2.4.1 Inspiration

Artificial neural networks are inspired by the human or animal nervous system. The brain is a complex network composed of interlinked neuron cells, passing on electrical impulses.

A typical neuron consists of components as shown in Figure 2.3. Incoming impulses are received at the dendrites and can be either inhibiting or stimulating a subsequent reaction impulse by the neuron [10, p. 11,12]. Each neuron has an activation threshold that has to be surpassed by the sum of all incoming stimulating impulses, while inhibiting impulses reduce the overall activation level [10, p.12 ff.]. If the threshold is surpassed, the neuron itself is producing an impulse, which then runs along the axon towards the axon terminals. The endpoints of the axon terminals are called synapses and are themselves connected to the dendrites of other neurons. The synapses again can either stimulate or inhibit the following neurons' activation potentials [10, p. 18].

While the specific interconnections between individual neurons are also relevant, most tasks like learning or remembering in the brain are accom-

plished by changing the different synapses' stimulating and inhibiting properties in the network [10, p. 20].

Artificial neural networks make use of the same basic principle, but have a reduced structural complexity compared to the brain's neural network and its neurons [10, p. 21]. Research on neurons in the visual cortex of animals, responsible for object recognition, have been the inspiration for the convolutional network layers discussed later in section 2.4.7 [2, p. 195].

As Mitchell points out, historically there have been two different motivations for the research of artificial neural networks: One is concerned with simulating biological neural networks as closely as possible in order to get a better understanding of the brain's biology, the second primarily wants to harness the capabilities of neural networks for machine learning applications [8, p. 82]. The development and building blocks of the latter approach will be outlined in the next sections.

2.4.2 Types of artificial neural networks

There are two main groups of artificial neural networks, namely *feedforward* and *recurrent* neural networks, also called acyclic and cyclic [11, p. 4]. As the name suggests, recurrent neural networks (RNN) feedback produced output as new input, feedforward networks on the other hand are hierarchical and directional [10, p. 29/251].

There is a wide range of artificial neural network architectures of varying complexity and constructed for different types of machine learning problems. All architectures share the idea of being networks of elemental basic units inspired by neurons. Goodfellow et al. name feedforward deep neural networks as the most important example for deep learning architectures [3, p. 5]. Those will also form the basis for this analysis.

2.4.3 Perceptron

One possible basic component in artificial neural networks mimicking a biological neuron is the perceptron. Following Mitchell (see [8, p. 86 ff.]), the perceptron is a unit that

> takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.

He defines the perceptrons output o as

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases} . \qquad (2.6)$$

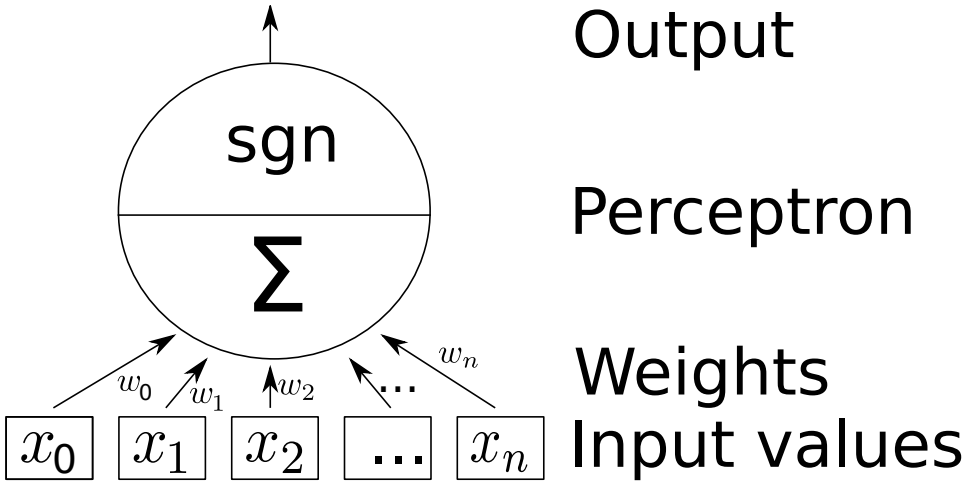


Figure 2.4: A generic perceptron, inspired by Rojas [10, p. 32].

The values $w_1 \dots w_n$ represent weight factors that manipulate the influence of the different values of $\vec{x}$ regarding the sum. The sum has to surpass the given threshold of 0. The weight w_0 is independent of the input and is used to define the threshold. In order to simplify the notation, Mitchell proposes an additional constant input of $x_0 = 1$, which shortens the perceptron's function to

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x}) \tag{2.7}$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases} . \tag{2.8}$$

A schematic of a generic perceptron can be seen in figure 2.4.

As Mitchell further elaborates, most boolean functions can be represented as single perceptrons. For example, assuming 1 means *true* and -1 $false$, and a perceptron has two inputs x_1 and x_2, an AND function could be implemented by setting $w_0 = -0.8$, while setting the other weights as $w_1 = w_2 = 0.5$. Table 2.1 shows the resulting outputs. In order to create an OR function, the threshold weight w_0 can be set to 0.3 while keeping the other weights, as shown in table 2.2. There is of course a wide range of possible weight combinations that could realize the same boolean functions.

Because a single perceptron can only calculate linear combinations of its input, some boolean functions can not be built with only one perceptron.

Table 2.1: *AND* calculation using a perceptron with weights $\vec{w} = (-0.8, 0.5, 0.5)$.

Input x_0	$w_0 \times x_0$	Input x_1	$w_1 \times x_1$	input x_2	$w_2 \times x_2$	**Sum**	Result
1	-0.8	1	0.5	1	0.5	**0.2**	1
1	-0.8	-1	-0.5	1	0.5	**-0.8**	-1
1	-0.8	1	0.5	-1	-0.5	**-0.8**	-1
1	-0.8	-1	-0.5	-1	-0.5	**-1.8**	-1

Table 2.2: *OR* calculation using a perceptron with weights $\vec{w} = (0.3, 0.5, 0.5)$.

Input x_0	$w_0 \times x_0$	Input x_1	$w_1 \times x_1$	input x_2	$w_2 \times x_2$	**Sum**	Result
1	0.3	1	0.5	1	0.5	**1.3**	1
1	0.3	-1	-0.5	1	0.5	**0.3**	1
1	0.3	1	0.5	-1	-0.5	**0.3**	1
1	0.3	-1	-0.5	-1	-0.5	**-0.7**	-1

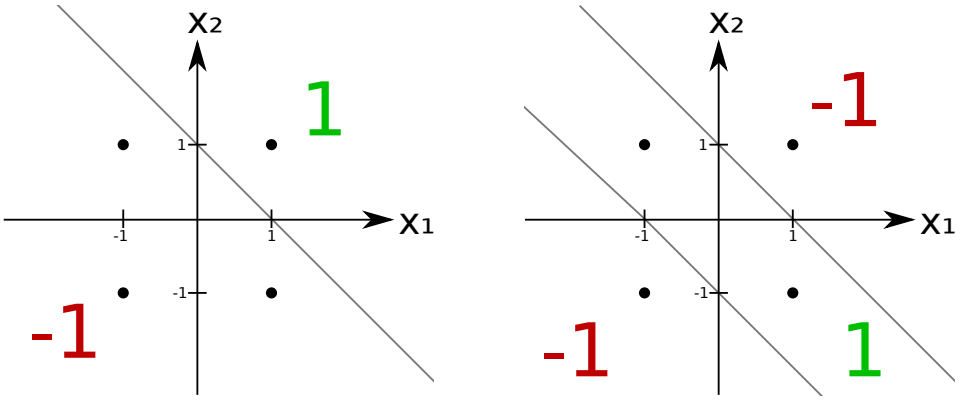


Figure 2.5: The *AND* decision boundary (a) can be expressed linearly, while the boundary for *XOR* (b) can not. Figure inspired by Rojas [10, p. 59].

Mitchell's example for this is the *XOR* function, as figure 2.5 illustrates [8, p. 87]. As he explains further, every boolean function can be expressed with a combination of the boolean functions *AND*, *OR*, *NAND* and *NOR*. Because these four can be expressed using a single perceptron, networks of perceptrons will be able to express every boolean function accordingly [8, p. 87].

2.4.4 Delta rule training using gradient descent

To summarize, networks of perceptrons can be used to express boolean functions by choosing the appropriate weights in order to produce the desired output. In analogy to the inhibiting or stimulating synapse connections in the biological neuron, a network of perceptrons is trained and learns by adjusting its weights.

Rojas describes neural networks in general as 'mapping machines' (German: 'Abbildungsmaschine'), that can be viewed as functions receiving input vectors and producing output vectors. While the initial architecture of the neural network is hand crafted, the final function itself (the already introduced *model* in machine learning terms), expressed by the weight values is impossible to follow for a human in its entirety when using increasingly complex networks. For this reason he states that trained neural networks are often times used as 'black box' functions [10, p. 29].

This means that for more complex models, requiring more complex networks, setting the weights by hand in order to produce the desired outputs as before is not feasible. Instead a learning algorithm is needed that is able to adjust the weights automatically.

As Mitchell explains, the so called delta rule accomplishes just that by using gradient descent. Mitchell first illustrates the delta rule on only one unthresholded perceptron called *linear unit* [8, p. 89].

Referring back to the general machine learning introduction, the vector of weights $\vec{w}$ of a perceptron or linear unit can be thought of as the model in machine learning terms. Also referring back to 2.1, training is done using a set of training data with D items, where each training item consists of a set of features $\vec{x}$ and a target value t.

Following Mitchell, the linear unit outputs the prediction o as

$$o(\vec{x}) = \vec{w} \cdot \vec{x}. \tag{2.9}$$

The training starts with all weights initialized randomly. Next the training error, or loss, is calculated. Mitchell defines the loss as

$$E(\vec{w}) = \frac{1}{2} \sum_{d=1}^{D} (t_d - o_d)^2. \tag{2.10}$$

This means the weights' loss is defined as half the squared distance between the expected values t and the actual predictions o over all training data [8, p. 89]. He points out, that E also depends on the given training set, but it is assumed that the set stays the same while training, so he omits an explicit declaration [8, p. 90]. Figures 2.6 and 2.7 visualize two possible loss functions over different weight values. Because the training goal is to

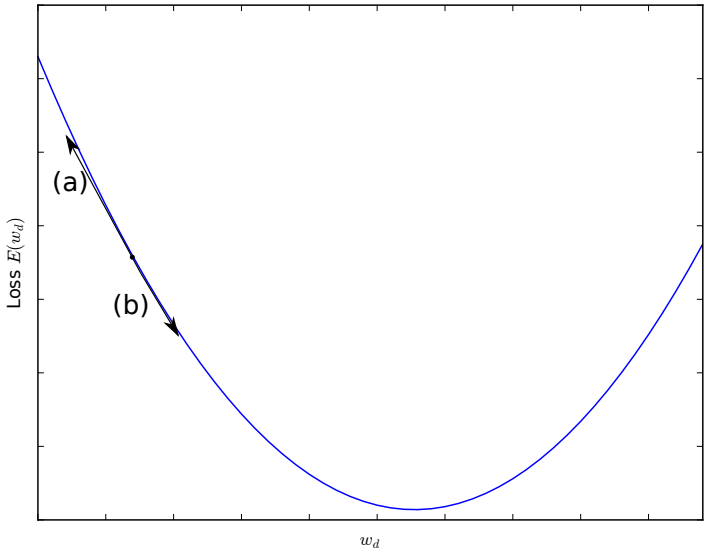


Figure 2.6: The loss E in respect to different values for a single weight w_d. Vector (a) represents the gradient $\frac{\partial E}{\partial w_d}$, while (b) is the descending gradient vector $-\frac{\partial E}{\partial w_d}$, multiplied by learning rate λ. Figure inspired by by Mitchell [8, p. 90]

minimize the loss, Mitchell continues by explaining the delta update rule, which uses gradient descent in order to find the weights with minimum loss [8, p. 91].

> How can we calculate the direction of steepest [drop in loss] [...]? This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the gradient of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$ [8, p. 91].

He defines the gradient vector $\nabla E(\vec{w})$ as

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_o}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right] \tag{2.11}$$

This vector can be interpreted as the direction in weight space in which the loss is increased the most, as shown in figures 2.6 and 2.7. As Mitchell points out, the negative of this vector in contrast will point in the direction of maximum decrease, which is the direction we are interested in.

So, in order to minimize the loss, he defines the delta training rule [8, p. 91] as

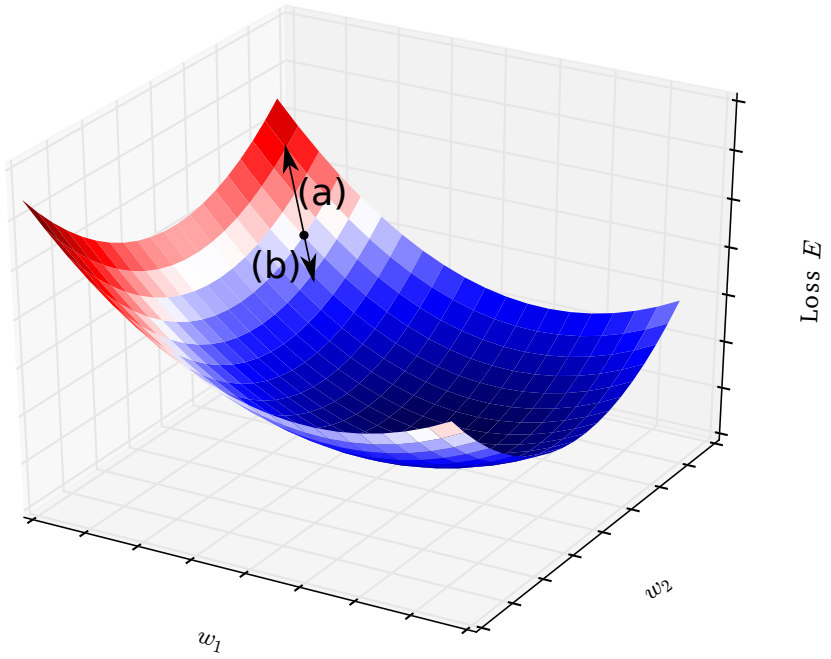


Figure 2.7: The loss E for two weights w_1 and w_2. Vector (a) represents the gradient $\frac{\partial E}{\partial w_d}$, while (b) is the descending gradient vector $-\frac{\partial E}{\partial w_d}$, multiplied by learning rate λ. Figure inspired by by Mitchell [8, p. 90]

.

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \tag{2.12}$$

where

$$\Delta w_i = -\lambda \frac{\partial E}{\partial w_i}. \tag{2.13}$$

λ is a static, positive factor called the **learning rate**, moderating the step size in which the weights are updated. This is done because of the risk that a step size too big causes the gradient descent to overstep the loss minimum as shown in figure 2.8 [9, p.247]. In contrast, choosing a learning rate too small causes the algorithm to run unnecessarily long by forcing it to take very small steps or even causing the descent to stop at local minima. Mitchell notes that in many cases λ is set to a relatively small value of 0.1, then is eventually further reduced over the cause of the training iterations as it nears the loss minimum [8, p. 91/92/94].

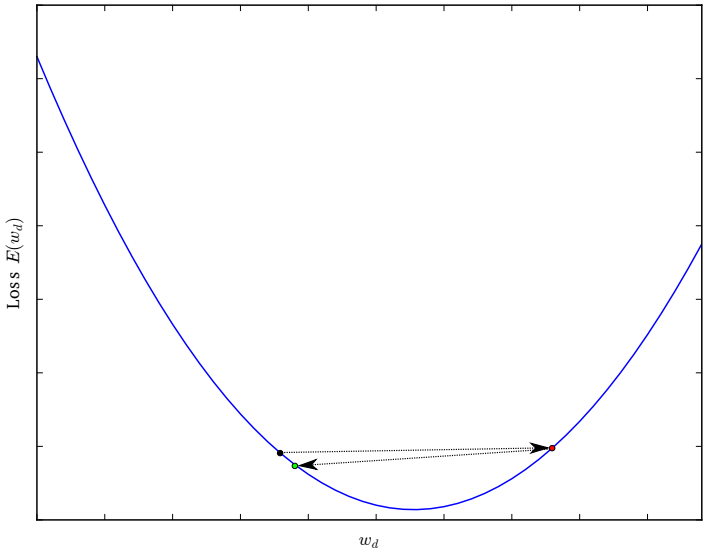


Figure 2.8: If the step size per iteration is too large, the training might overstep the loss minimum. The learning rate λ is used to avoid this. Figure inspired by Murphy [9, p.247].

Mitchell calculates the derivative vector $\frac{\partial E}{\partial w_i}$ from equation 2.10 as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} =& \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d=1}^{D} (t_d - o_d)^2 \\
& \frac{1}{2} \sum_{d=1}^{D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
& \frac{1}{2} \sum_{d=1}^{D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
& \sum_{d=1}^{D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} =& \sum_{d=1}^{D} (t_d - o_d)(-x_{id})
\end{aligned} \tag{2.14}$$

where x_i represents the value of a single feature i of input vector $\vec{x}$. Mitchell defines the final update rule for gradient descent as

$$\Delta w_i = \lambda \sum_{d=1}^{D} (t_d - o_d) x_{id} \tag{2.15}$$

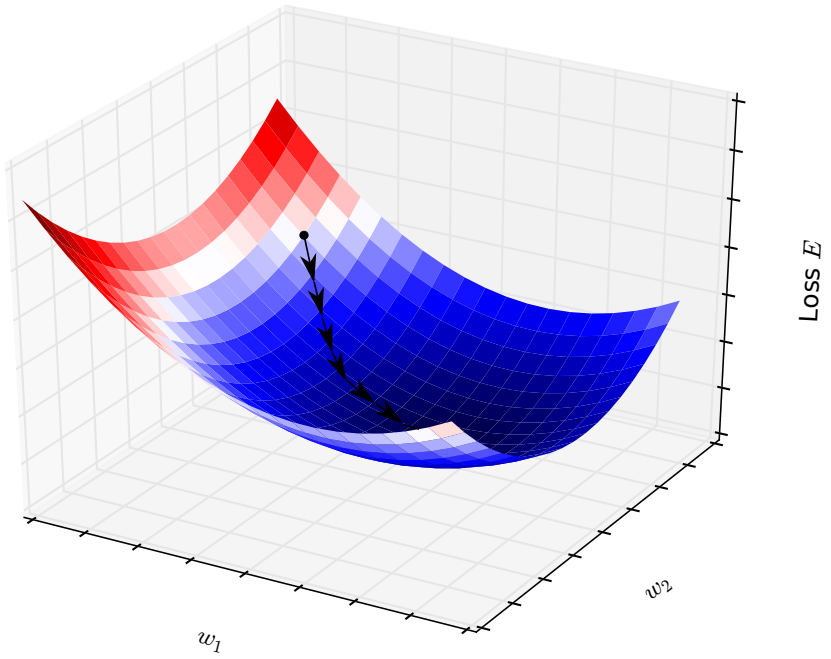


Figure 2.9: The loss E is reduced over multiple iterations for two weights w_1 and w_2. Figure inspired by by Mitchell [8, p. 90].

by inserting equation 2.14 into equation 2.13. The weight adjustments are repeated over several iterations using the delta rule, over time reducing the loss to its minimum, as shown in figure 2.9.

Mitchell emphasises that gradient descent is not just a method for training linear units, but an algorithm also used in other machine learning techniques [8, p. 92] (see also [7, p. 189], [9, p.247] and [9, p. 445]).

Finally, he states that in many practical applications, a variation of gradient descent called stochastic gradient descent is used. Here, instead of calculating the weight gradients over the complete training set, only gradients for single training items or for randomly picked sub-batches of training data are calculated. In consequence, the stochastic gradient descent only approximates the steepest loss decrease, but its calculation is much faster. Additionally, he notes that stochastic gradient can be more robust in avoiding local minima. [8, p. 94]

2.4.5 Backpropagation

Because the previously introduced linear units can only represent linear functions and the perceptron's discontinuous *sgn* function can not be dif-

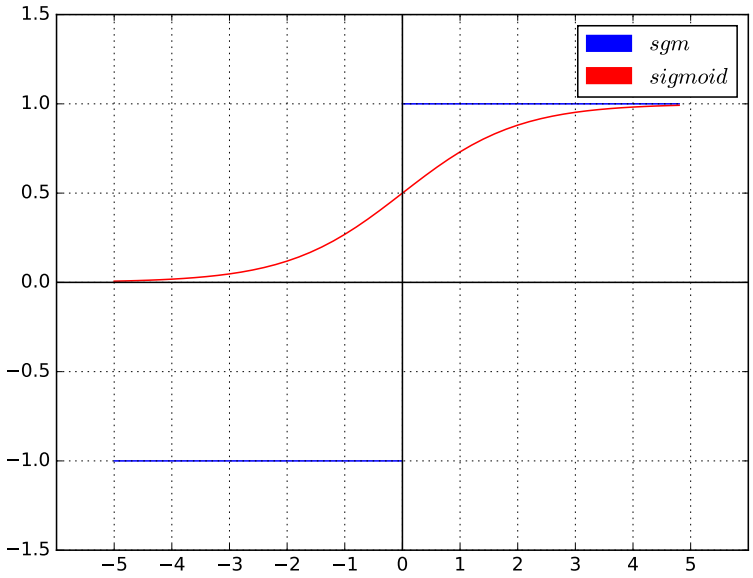


Figure 2.10: The perceptron's *sgn* function is undifferentiable. In order to do gradient descent, the *sigmoid unit* replaces *sgn* with the *sigmoid* function, producing continuous outputs between 0 and 1.

ferentiated (and thus not be used for gradient descent), Mitchell introduces the sigmoid unit, which mimics the perceptron but uses a threshold function that is differentiable.

The sigmoid unit's output is defined by Mitchell as

$$o = \sigma(\vec{w} \cdot \vec{x}) \tag{2.16}$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}. \tag{2.17}$$

Sigmoid units are used for training more powerful networks with multiple layers with the so called backpropagation algorithm [8, p. 95].

Figure 2.12 shows a network containing multiple layers with multiple sigmoid units per layer. The first and the last layer of a network are commonly referred to as the input and output layers, while all layers in between are called hidden layers [10, p.155]. The term 'hidden' again reflects the 'black box' character of neural networks noted by Rojas [10, p.29].

The shown multilayer network has three output units. This means that instead of doing binary classification, the depicted network is able to do

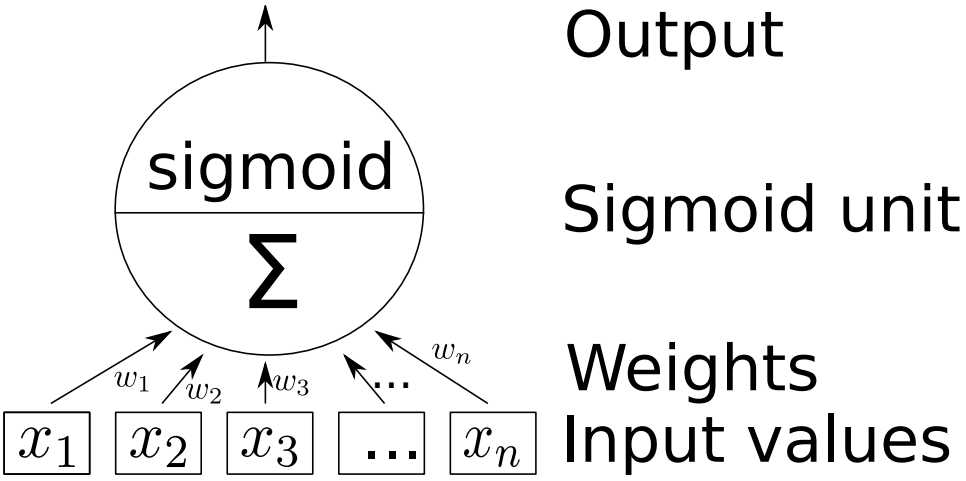


Figure 2.11: The sigmoid unit, the perceptron's *sgn* function got replaced by the *sigmoid* function. Inspired by [10, p. 32]

.

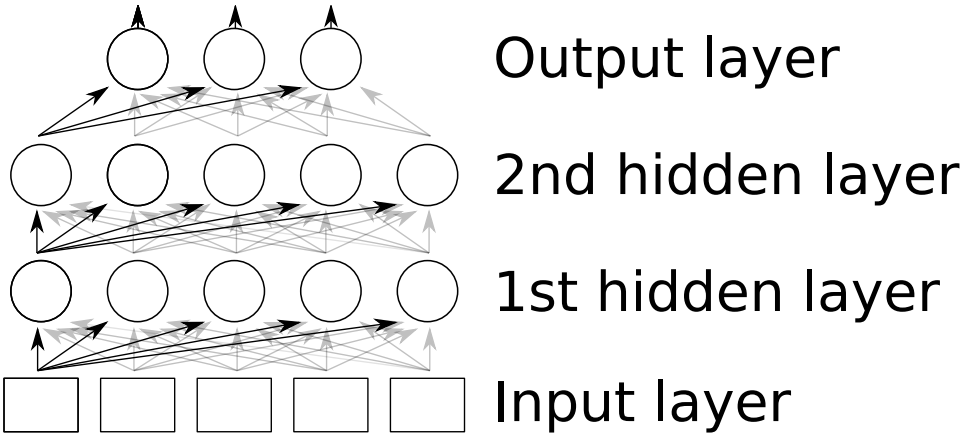


Figure 2.12: A multilayer network with multiple units on each layer. Figure inspired by Mitchell [8, p.98]

multiclass classification. In order to calculate the loss over all output units, the loss function 2.10 is extended to

$$E(\vec{w}) = \frac{1}{2} \sum_{d=1}^{D} \sum_{k=1}^{K} (t_{kd} - o_{kd})^2 \tag{2.18}$$

where K is the number of output units [8, p.97]. For a multiclass classification task, K would be equivalent to the number of classes the network is supposed to learn and tries to classify.

So far, gradient descent was used to optimize the weights by calculating the loss between the expected values and the predicted ones. For multilayer networks, the question arises of how to calculate the loss for the hidden units, because there are no explicit expected values given (those are only present for the K units in the output layer).

The backpropagation algorithm solves this problem by doing so called feedforward and backpropagation passes repeatedly: In the feedforward pass, the the input is passed trough the network from bottom to the top and the loss between produced values and expected values in the output layer is calculated. In the backpropagation pass, the partial contribution of each individual unit in the network to the final loss in the output layer is calculated from top to bottom.

Using the derivation chain rule, Mitchell shows that for the units in the *output* layer the update rule is defined as

$$\begin{aligned} \Delta w_{ji} &= -\lambda \frac{\partial E_d}{\partial w_{ji}} \\ &= -\lambda (t_j - o_j) o_j (1 - o_j) x_{ji} \end{aligned} \tag{2.19}$$

where w_{ji} denotes the weight associated with the ith input to unit j, t_j the target value for the unit j, while o_j represents its produced output and x_{ij} is the ith input to unit j [8, pp. 101-103]. Again applying the chain rule, he shows that the weight updates for units in the *hidden* layers can be computed as

$$\Delta w_{ji} = \lambda \delta_j x_{ji} \tag{2.20}$$

where δ_j denotes the individual loss contributed by the unit j to the overall loss [8, p. 99]. The term δ_j is calculated as

$$\delta_j = o_j (1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj} \tag{2.21}$$

where Mitchell defines Downstream(j) as the set of those units that directly follow unit j and use its output as input [8, p.101].

For the last hidden layer, this means that the different δ_k values are just the loss values generated by the units in the output layer. These can can be calculated as $(t_k - o_k)$ as before [8, p.98]. After the individual loss contribution for all units in the last hidden layer are calculated this way, those values again can be used to calculate the individual loss in the next lower hidden layer.

For a detailed derivation of equations 2.19 and 2.20 please refer to [8, p. 97-103] or [10, p. 154-165]. By running the backpropagation algorithm for

multiple iterations, all weights get gradually optimized and the overall loss of the network should decrease over time.

2.4.6 High bias vs. high variance

When training feedforward networks, there are two conditions that have to be avoided and that can potentially be rectified by changing training parameters like the learning rate.

The first is a high bias, meaning that the network fails to learn weights that produce the desired output. For example, this could mean that the network architecture is too simplistic to represent an adequate model, or that the backpropagation algorithm got stuck in a local minimum while running gradient descent. High bias is indicated by a continual high training loss. The second problem is high variance, also called 'over fitting'. In cases of high variance, good weights were found producing a low training loss, but the weights only work well for the training data. In other words the model is not generalizing well. High variance is indicated by a low loss for the training data but a high loss for test data [8, pp. 108-110].

High bias and high variance are two opposing extremes that both reduce a model's performance. In many machine learning approaches another term for the problem of finding a suitable middle ground is the *bias-variance-dilemma* [7, p. 35].

2.4.7 Convolutional neural networks

In section 2.3 about deep learning, the idea of generating increasingly abstract features from input data was introduced. When using neural networks for image classification, this deep learning concept is often implemented by convolutional deep neural networks.

Convolutional neural networks (CNNs) are feedforward networks, but instead of only using fully connected layers (each unit is connected to each unit on the next higher layer) as depicted in 2.12, the network is separated into two major parts. At first the input data is processed by so called convolution layers, able to extract abstract features, followed by the already introduced fully connected layers that are used to learn concepts [5, p. 253].

The underlying principle of convolution layers is comparable to filter kernels known from image processing [5, p. 253]. But again, instead of predefining these filter kernels by hand, those are learned by the neural network. The first processing step in figure 2.2 shows the application of two such learned filters to an image. The results are very similar to images of x/y gradients used for example to construct *Sobel* filters [33].

A simplified schematic of a convolutional neural network with one convolution and one fully connected layer is shown in figure 2.13. For demonstration purposes, the filter kernels are depicted as one dimensional, instead of

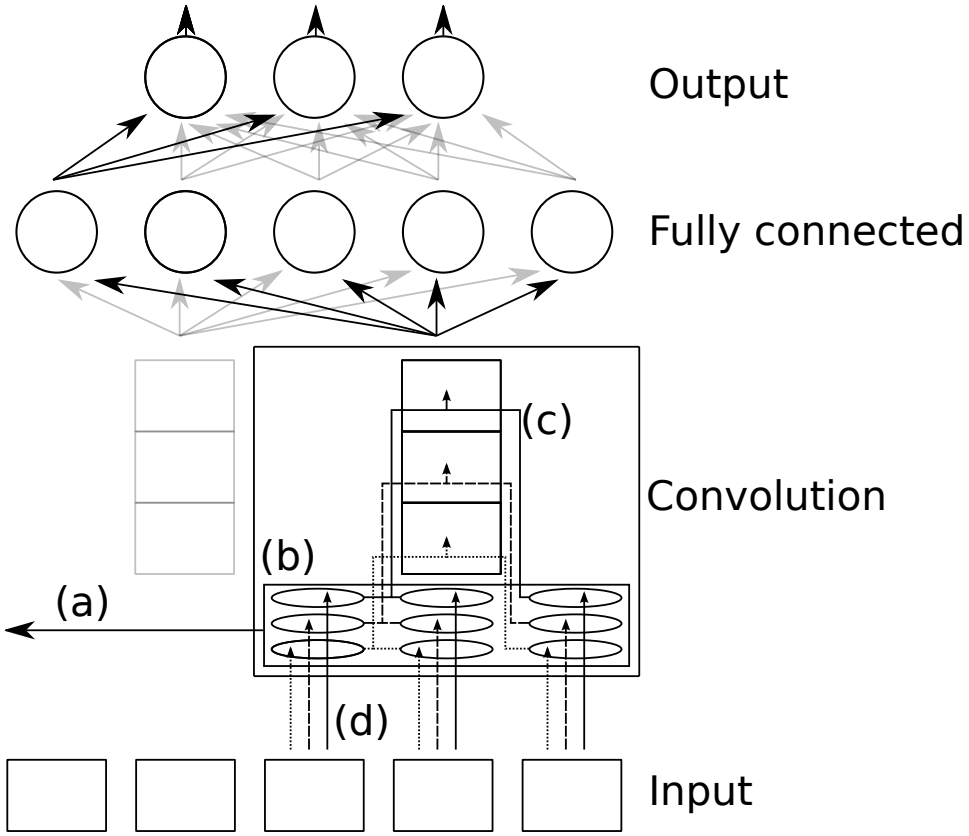


Figure 2.13: The structure of a convolutional feedforward neural network. Figure inspired by [38].

the two (height/width) or three dimensional (height/width/color channels) kernels present in actual implementations for image recognition [5, p. 253]. LeCun et al. refer to both the input and the output of the convolutional layer as feature maps. The input depicted in figure 2.13 is a one dimensional feature map with five values, for example this could be the pixels of an input image. The three filters in the filter bank (b) produce a two dimensional output of three feature maps with two values each (c). The output is produced by sliding the filter bank stepwise over the input (a). The depicted step size is 2, overstepping every second input value.

As already mentioned, this is reminiscent of classic image processing using pre defined filter kernels. But in this case, the kernels are not crafted by hand, but learned by adjusting the weights (d) over many feedforward and backpropagation passes. This allows the neural network to solve the complexity problem of searching for good, abstract features, addressed by Goodfellow et al. 2.3 on its own.

In many practical applications convolutional feedforward neural net-

works are constructed of multiple convolution layers, passing their output feature maps from one to another, followed by multiple fully connected layers (see [4] or [5]).

Chapter 3

Frameworks and technologies

3.1 CUDA

CUDA is a programming interface for GPU processing, developed by NVIDIA and used by most deep learning frameworks [23][22]. CUDA only supports NVIDIA GPUs, meaning that for both deep learning frameworks mentioned in sections 3.2 and 3.3 a machine with one or multiple NVIDIA graphics cards is required to take benefit of the faster computations. For the training described in chapter 5.3, a Dell PowerEdge rack-server with 4 NVIDIA Tesla K20 GPUs was used (see [24] and [32]).

3.2 Deeplearning4j

The open source project *Deeplearning4j* (DL4J)[35] was started in 2014 and is being built with *Java* and *Scala*. Because the *Arachne* back end is also written in *Java*, DL4J was the first deep learning frameworks examined for this project – hoping for a potentially easier integration into the existing infrastructure.

Alongside the deep learning framework itself, the developers are creating a linear algebra library named *ND4J* [36] and a vectorization library named *Canova* [21]. DL4J can be run both on CPUs and CUDA GPUs (via *jCUDA*) [30]. Another option is the use of *Amazon Web Services* [14].

While setting up several tutorials provided by the DL4J website on a local machine the developers proved to be very dedicated to helping newcomers. But when making the first test runs with a bigger datasets, it turned out that, at the time[1], the GPU processing was still work in progress and quite unoptimized. After talking to the developers, they confirmed the observation and referred to the aforementioned *Amazon Web Services*.

Having no prior experience with machine learning, neural networks or

[1] November 2015

deep learning, it was hard to estimate the costs of buying processing power from a commercial cloud service like *Amazon Web Services*. For that reason, the idea of implementing the thesis using DL4J was dropped and instead the *Caffe* framework was chosen.

3.3 Caffe

The deep learning framework *Caffe* was created by the *Berkely Vision and Learning Center* and is written in C++ [17][16]. Working with Caffe does not necessarily require programming, instead the user can set up networks and their training in the form of plain text *Google Protocol Buffer* (*.prototxt) configuration files [27]. After configuring the environment, training can be started using different interfaces, namely the command line, *Python* or *Matlab*. *Caffe* is a rather powerful framework, this chapter will only deal with the parts necessary for explaining the concept and implementation described in sections 4 and 5.3.

3.3.1 Basic components

Blob: Blobs serve as the basic data structures in *Caffe*. They are N-dimensional arrays and are passed through the layers of the neural network.

Network Definition: The network's structure is defined in plain text *Google Protocol Buffer* (*.prototxt) [27]. An example can be found in appendix B.1.

Layer: Layers are the basic building blocks of neural networks. *Caffe* provides a range of standard layer types (like the already mentioned 'Convolution' or fully connected layers). Custom layer types can be implemented for both CPU and GPU computation [18].

Solver: The solver implements the gradient descent for the neural network. There are several different existing solver types to choose from [20]. The solvers are set up in a separate prototxt file (for an example see example code 3.1). Stochastic gradient descent (SGD) is the default solver, if none is explicitly specified in the solver's prototxt.

3.3.2 Fine tuning

Caffe provides the opportunity to use an already trained network (model) for further training on custom data and images. A collection of models can be found in *Caffe's* 'model zoo'[19]. The models consist of the learned network weights and are provided in serialized form as binarized prototxt files.

Program 3.1: Example: solver.prototxt

```
net: "models/bvlc_reference_caffenet/train_val.prototxt" # The
    network definition.
test_iter: 1000 # 1000 batches of test images are used per test run.
test_interval: 1000 # Tests are done every 1000 iterations.
base_lr: 0.01 # The learning rate of 0.01 at the start ...
lr_policy: "step" # ... is reduced step wise ...
gamma: 0.1 # ...by multiplying it by 0.1...
stepsize: 100000 # ...every 10000 iterations.
display: 20 # Every 20 training iterations log the current loss.
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000 # Snapshot of the net being trained.
snapshot_prefix: "models/bvlc_reference_caffenet/caffenet_train"
solver_mode: GPU
```

Fine tuning can be configured to affect the complete network or be restricted to certain layers. It is a powerful tool in cases where there is only a relatively small image data set available. Fine tuning will be discussed in more detail later.

Chapter 4

Concept

In order to evaluate the image classification capabilities of neural networks in the domain of archaeology, there are two main angles to approach the problem: Training a neural network with one's own image data or using an already trained network, called *model*, to build classifiers for said data. The second approach gives the opportunity to build classifiers without additional infrastructure, because no further neural network training is necessary.

Both options require a dataset of training and test images. The source for the images is the archaeological object database *Arachne*[15] of the *German Archaeological Institute* (*Deutsches Archäologisches Institut*, DAI) and the *University of Cologne*.

4.1 Creating the dataset

Arachne contains 2.2 million images[1], with 1.8 million images accessible without an account. Ideally, the download of training- and test data should be accomplished automatically – by defining image queries and associated labels, representing categories, for a downloading script.

The database contains objects, but this term denotes not just single artefacts like pottery or sculptures, but also photographs of cities, ruins, people or landscapes – documenting archaeological sites but also different expeditions' travels. While the final category set was not established before the import implementation, it was quite clear from the start, that the classifiers would have to deal with a wide range of motives and with categories that could potentially overlap, like for example buildings and ruins.

For use with *Caffe*, an image import for training and testing is comprised of the following components:

Images: The downloaded image files.

[1] November 2015

Information about training and test images: Two text files for training and test data respectively. Each line contains the path to an image and the associated label, represented as an index.

```
1 ..
2 image_imports/import_name/images/image_01.jpg 0
3 image_imports/import_name/images/image_02.jpg 0
4 image_imports/import_name/images/image_03.jpg 1
5 image_imports/import_name/images/image_04.jpg 1
6 ..
```

Information about the index mapping: A text file mapping label strings to the label indices in the other two information files.

```
1 Category_1 0
2 Category_2 1
3 Category_3 2
4 Category_4 3
5 ..
```

The import could be realized either by accessing *Arachne*'s SQL database directly for very focussed queries or using its back end's *ElasticSearch*[26] server, potentially giving up some precision. The implementation is described in chapter 5.

4.2 Creating classifiers using existing models

Neural networks for image classification are trained on a given image dataset and its categories. While training a complex network may require additional hardware, using a trained network, called *model*, to make predictions is possible on less powerful machines. While using a model trained for a different domain is possible, the trained categories potentially do not fit the new domain or are imprecise. This poses the question of how to still make use of neural networks if there is neither a model trained with appropriate categories, nor the infrastructure for training a new model at hand.

The strategy suggested hereafter is to create alternative classifiers based on the neural activations in the lower layers of the network. The assumption being, that even though the final categories may not fit the archaeological context, the model is 'smart' enough to detect a wide range of features and patterns in the lower layers. These lower layer activations will be used as features for training less computational expensive machine learning classifiers.

4.2.1 Choosing a model

The first step to build these alternative classifiers is to choose a model that produces the lower level activations. *Caffe* offers a 'model zoo' containing a range of already trained models.

As discussed in section 4.1 the archaeological images are going to have a wide range of motives. The *Hybrid-CNN* model from MIT and Princeton University[13] seemed to be the most promising model for the analysis. It is already trained on the data for the *Large Scale Visual Recognition Challenge 2012* (ILSVRC2012)[31] containing roughly 3.6 million images in 975 object categories and the project's own *Places Database* containing an additional 2.5 million images in 205 categories.

The ILSVRC2012 data set contains categories of a very diverse range of bigger and smaller objects, for example different types and breeds of animals, objects like ships, cars, chairs or different kinds of fruit[28]. There are also some categories concerning landscapes and buildings like 'volcano', 'valley' or 'palace', but the aforementioned types of small objects and technical artefacts are predominant. This is where the *Places Database*'s focus on buildings and landscapes could prove beneficial. Some categories in the *Places Database* even have an archaeological focus, like 'amphitheater' or 'aqueduct', but the majority depict modern places like 'aquarium', 'construction site' or 'movie theater'[29]. The combination of both data sets made the *Hybrid-CNN* the seemingly ideal candidate for the archaeological images. To evaluate this assumption two other models are compared to the *Hybrid-CNN*: *Caffe*'s *BVLC Reference CaffeNet* model, trained on only the ILSVRC2012 images and the *Places-CNN* model trained only on the *Places Database*'s images.

Advantageously, all three models share the same network architecture, namely the *AlexNet* architecture introduced by Krizhevsky, Sutskever and Hinton (see [4]), making results comparable without the need for an additional discussion of different architectures. The *AlexNet* is a convolutional deep neural network, comprised of five convolution layers followed by three fully connected layers [4, p. 4]. Those eight main components are named $conv1, conv2, \ldots, conv5, fc6, fc7, fc8$ respectively in *Caffe*'s implementation of the network architecture (see appendix B.1).

The training and the test images from *Arachne* will be processed by the neural network, with the selected models as weights, and for each image the activations at the fully connected layers *fc6* and *fc7* will be collected and saved together with the images' labels. Both layers contain 4096 units, meaning that after finally being passed trough the layer, each image is mapped to 4096 real number values. The last inner product layer *fc8* will be ignored, because it contains the same amount of units as the respective model's trained categories. Because the different models were trained on different data with different categories, layer *fc8* differs for each model and makes it thus not

comparable.

4.2.2 Classifiers

Different classifiers are going to be tested and compared for each model and layer: *Naive Bayes*, *k-nearest neighbours* and two classifiers based on *k-Means*-clustering.

Naive Bayes: *Naive Bayes* is a probabilistic approach to the classification. For each of the 4096 activation features over all the training images, the algorithm calculates the probability that a feature indicates a certain label. For a given test activation, the combination of the probabilities per feature produce the classification decision. The term $P(c)$ in the classifier definition denotes the prior probability that category c will be observed in the training data (see *Naive Bayes* introduction 2.1.2). Because all categories in the training dataset are expected to be equiprobable, the term is assumed to be the same for all labels and will be omitted in the implementation, shortening the classifier's c_{NB} definition to

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} \sum_{i=1}^{k} \log\left(P(d_i|c)\right). \tag{4.1}$$

k-nearest neighbours: For *k-nearest neighbours*, the activations are interpreted as vectors with 4096 dimensions. For each test activation *k-nearest neighbours* searches the k nearest training activations. The label with most activations between the k neighbours is assigned as the predicted label for the test activation.

k-Means clustering: Because *k-nearest neighbours* is computational expensive, *k-Means* will be used to pre-process the training data by dividing it into k clusters. Afterwards, only the Euclidean distances between test activations and the cluster centres has to be calculated, reducing the amount of necessary computations. In order to do a classification using the clusters, two classifier variants are being implemented.

One creates a small group of k clusters separately for each label in the training set. A test activation is classified with the label of its closest cluster.

The other approach generates clusters over all training activations, with no separation by labels. After the clustering itself is completed, a histogram of associated labels is saved for each cluster. Test activations are assigned the label maximum of its closest cluster.

Both *k-nearest neighbours* and *k-Means* are going to be tested with different values of k in order to find the best results.

4.3 Training a network as classifier

Finally, after evaluating the classifiers created without additional training, training a custom model as a classifier with the *Arachne* images and their labels is going to be evaluated.

The image set imported from *Arachne* is expected to be smaller than the datasets used for training the previously presented models picked from *Caffe*'s 'model zoo'. Due to the time constraints for this thesis, a set of several thousand images instead of millions is more likely. This limitation makes it advisable to use *Caffe*'s fine tuning feature.

The model with the best results evaluated section 4.2 will be used for further fine tuning. Fine tuning will be applied to the complete network and alternatively be restricted to selected layers. Additionally, normal training without any model will be carried out too.

4.4 Test metrics

In order to analyse the different classifiers implemented, the image data from *Arachne* will be split into a set of training- and a set of test images. The classifiers will be trained using the training data and evaluated using the test data. In order to be able to compare the performance of the different classification methods, two different metrics will be used.

Accuracy

The accuracy A [6, p.143] of a classifier is simply defined as

$$A = \frac{\text{Number of test images classified correctly}}{\text{Number of test images}}. \tag{4.2}$$

Mean average precision

The implemented classifiers return predictions for all labels with different certainties. Because some image categories can be considered ambiguous, meaning certain images could be in one category or the other, it would make sense to have a metric that not only considers the final classification decision based on the highest certainty, but to also the remaining categories. This is accomplished using the *mean average precision* (MAP).

In order to do so, the prediction results have first to be sorted by descending certainty. Having done so, the result for a 'river' image classified incorrectly as 'plains', with 'plains' at first position, should be considered better, if the prediction for 'river' came in 2nd instead of 15th place.

Mean average precision is used to judge this ranking over all test images. The metric is commonly used in information retrieval [6, p. 147].

The number of correct predictions returned by the classifiers should be either 0 or 1, because in multiclass classification only one predicted category can be the correct one. For a single test image, the average precision [12, p. 15] can be calculated as

$$AP = \frac{1}{r} \sum_{k=1}^{N} Precision(k) \times rel(k) \tag{4.3}$$

where N is the number of sorted predictions, and r the number of correct predictions within the complete list of sorted predictions [12, p. 15]. Following Turpin et al. the precision of a classification will be defined as

$$Precision = \frac{\text{number of correct predictions}}{\text{number of observed predictions}} \tag{4.4}$$

where the number of observed predictions is defined by k in equation 4.3. In other words, $Precision(k)$ calculates the precision for the first k ranked predictions. $rel(k)$ in equation 4.3 is an indicator function defined as

$$rel(k) \begin{cases} 0 & \text{if prediction at position } k \text{ is wrong} \\ 1 & \text{if prediction at position } k \text{ is correct} \end{cases} . \tag{4.5}$$

Figure 4.1 gives an intuition about how the values are calculated. For multiclass classification, resulting in at most one correct category, the *average precision* may seem to be over elaborated, but the initial idea was to have a metric that could also cope with potential multilabel classification, as illustrated in figure 4.2.

The *mean average precision* for all test images can then finally be calculated as

$$MAP(Q) = \frac{1}{m} \sum_{j=1}^{m} AP(q_j)). \tag{4.6}$$

where in this case Q denotes a set of m test images $q_1, q_2, \ldots, q_m$ [6, p. 147].

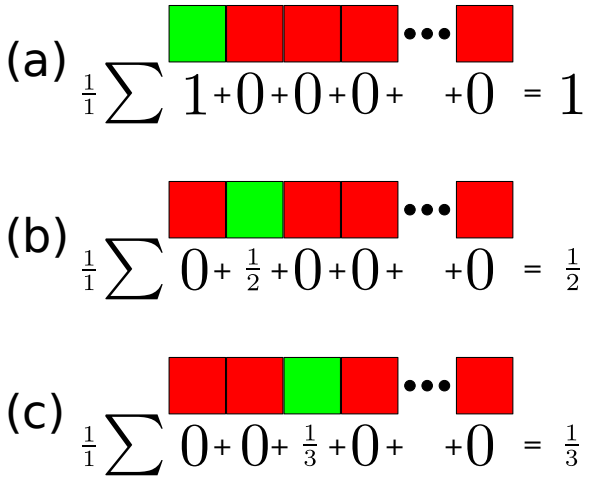


Figure 4.1: The average precision, as defined in equation 4.3 calculated for cases where (a) the correct label was classified, where (b) the correct category was in second place and (c) in third place.

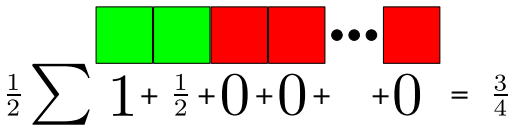


Figure 4.2: The average precision, as defined in equation 4.3 calculated for a case where there are multiple correct labels per image, in this case the correct ones are predicted in first and second place.

Chapter 5

Implementation

All following code will be written in python, utilizing *Caffe*'s python interface where necessary. *Caffe* will be configured using plain text prototxt, as already described in section 3.3.

5.1 Image import from Arachne

5.1.1 SQL based import

The first approach was the direct query of *Arachne*'s mySQL database with the script *import_via_SQL.py*.

The script loads a XML configuration file, itself defining categories and their respective SQL queries. First all queries are run, returning *Arachne*'s internal Id for each image and a url. Next, all image data is downloaded and the three info text files are generated. Every 5th image is automatically saved as a test image.

While working well for small imports with 4-7 categories, increasing the category count made the SQL queries increasingly complex and hard to manage. As a consequence, an alternative was implemented, making use of *Arachne*'s search back end.

5.1.2 Search based import

The search in *Arachne* is provided by an *ElasticSearch*[26] server. The script *import_via_elastic.py* uses *Arachne*'s data interface to run queries and retrieve results in JSON format. Similarly to the SQL based approach, queries and their categories are defined in an external configuration file. To keep formats consistent, the configuration file's format was switched from XML to JSON (see program example 5.1).

An advantage compared to the SQL queries, besides the better readability, is the possibility to get an overview over the results by pasting the query

Program 5.1: Extract: JSON import configuration.

```
{
  "exportName": "elastic_example",
  "queries":
  [
    {
      "query": "fq=facet_image:\"ja\"&q=personengruppe",
      "label": "person"
    },
    {
      "query": "fq=facet_image:\"ja\"&fq=facet_kategorie:\"Einzelobjekte\"&q=einzelperson",
      "label": "person"
    },
    {
      "query": "fq=facet_image:\"ja\"&fq=facet_kategorie:\"Topographien\"&q=landschaft",
      "label": "topography"
    },
    {
     ...
```

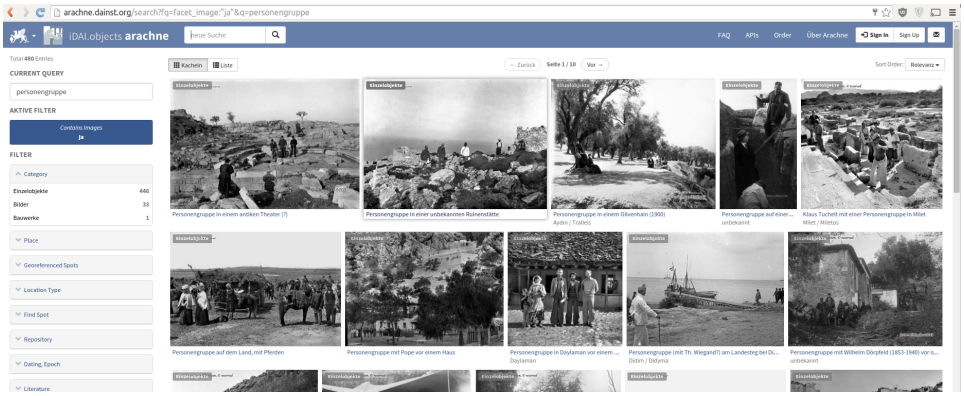


Figure 5.1: Arachne search overview with result grid.

into *Arachne*'s front end search first. Figure 5.1 shows how the results are presented as a grid of image previews, making a quick assessment of query's usefulness possible.

The search returns JSON representations of *Arachne*'s basic data class, the generic entity. The term entity subsumes a wide range of content, for

example different kinds of real world objects (buildings, sculptures, pottery etc.), places or meta data about images. Entities can have image entities attached to them (given the right query, only entities with images will be returned).

In a second step the import script collects the ids of all images associated with the entities and groups the images with their parent entity's category labels. In order to prevent an uneven image count per category, the images are first split by category. Then the category with the fewest images defines the image maximum per category and the surplus in the other categories is discarded.

Finally, the image data itself is downloaded using the remaining ids and the information files required by *Caffe* are created in the process.

The search based approach sacrifices some precision when formulating the queries compared to SQL, but the better work flow – utilizing the front end to have a preview for each query – and the generally more readable queries outweigh that drawback.

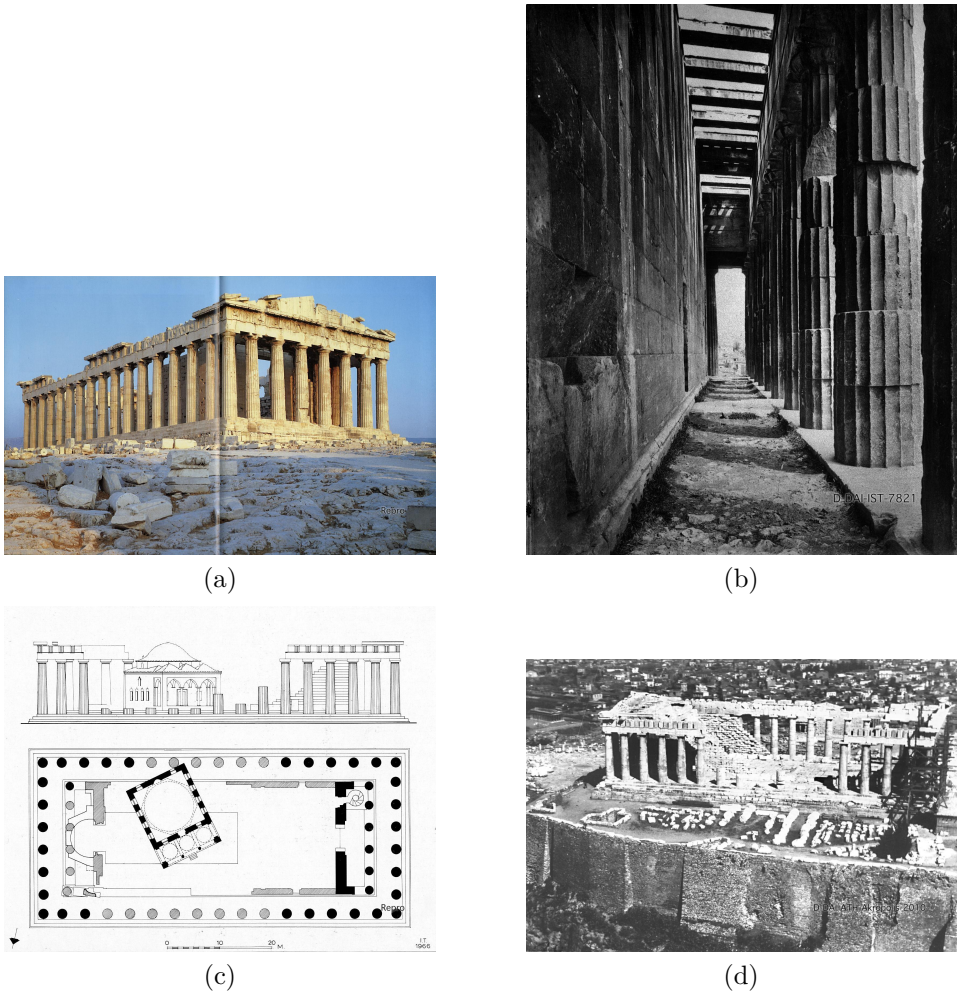
5.1.3 Manual refinishing

One significant problem emerging while implementing both previously introduced methods of image import was the lack of a more general motive description in *Arachne*, especially when it came to images of buildings and landscapes.

Even though it was possible to deduce that an image will show a building (because its parent entity is a building), the decision which kind of depiction is at hand, is not that simple: The image could be any of either the building in profile from ground level, details in the face of the building, of its interior, an aerial view, a ground plot or other schematics, as exemplified in figure 5.2.

All those types of motives are potential categories themselves. In order to achieve the additional category granularity, a search-based import was further sorted by hand into more precise 32 categories. Figure 5.3 shows its final label distribution. The script *prepare_handsorted.py* is used to scan the hand sorted import's sub folders. Each sub folder name is interpreted as a category label. The script then again creates the three information text files for *Caffe*, again picking every 5th image as a test image.

The final image data set for all further analysis contains 7285 training and 1837 test images, with 32 different labels.



(a) (b)

(c) (d)

Figure 5.2: Examples for different image types, taken from the 144 images associated with The Parthenon, Athens. View of the building from ground level (a), interior (b), schematics (c), aerial photography(d).[37]

5.2 Creating classifiers without additional training

The basis for creating classifiers without additional training are the activations in deeper network layers produced by already 'smart' models processing image data. These activations are used as the features for different machine learning algorithms. The concrete activations used for this analysis are the output of layer either *fc6* or *fc7* in *Caffe*'s implementation of the *AlexNet* architecture (see appendix B.1).

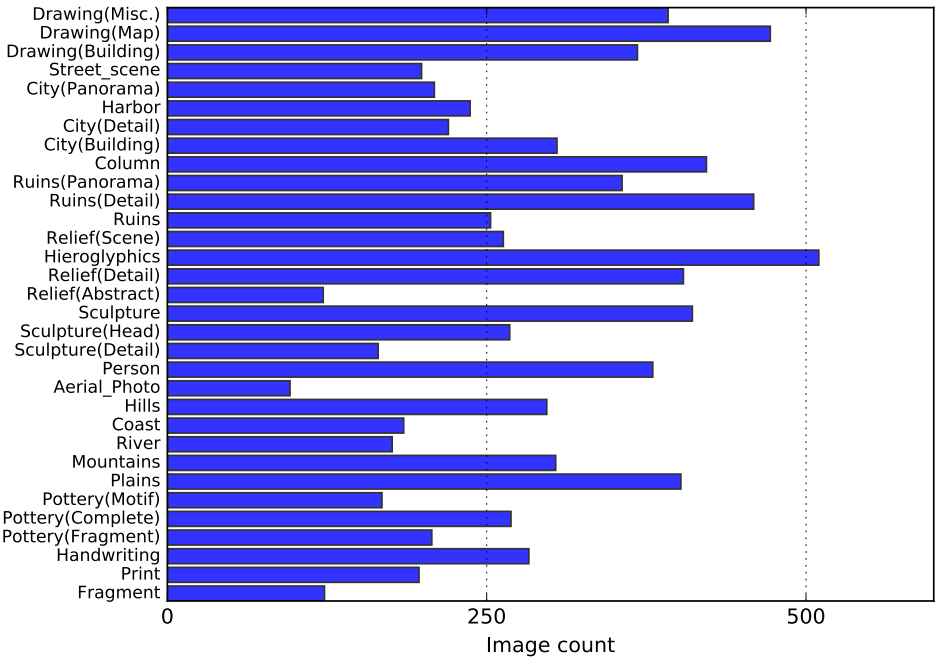


Figure 5.3: Histogram of the image count per category in the handsorted data set. For image examples of each category see appendix A.

5.2.1 Generating activations

Because both layers output 4096 values, each training and test image will be represented as a 4096 dimensional feature vector after being processed by the neural network. Even though the three models evaluated in this analysis create 4096 features for each image, all scripts discussed hereafter are implemented for a generic amount of features and labels.

The feature matrix $\mathcal{F}$, representing the layer activations, will from now on be defined as

$$\mathcal{F} = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1F} \\ f_{21} & f_{22} & \cdots & f_{2F} \\ \vdots & \vdots & \ddots & \vdots \\ f_{N1} & f_{N2} & \cdots & f_{NF} \end{pmatrix} \tag{5.1}$$

where F is the number of features and N the number of images. The label information is stored similarly in matrix $\mathcal{L}$ and will be defined as

$$\mathcal{L} = \begin{pmatrix} l_{11} & l_{12} & \cdots & l_{1L} \\ l_{21} & l_{22} & \cdots & l_{2L} \\ \vdots & \ddots & \ddots & \vdots \\ l_{N1} & l_{N2} & \cdots & l_{NL} \end{pmatrix} \tag{5.2}$$

where L is the number of existing labels and N again the number of images. $\mathcal{L}$ is used to flag the labels, where the value l_{ik} is defined as

$$l_{i,j} = \begin{cases} 1.0 & \text{if the label with id } j \text{ is associated with image } i \\ 0.0 & \text{otherwise} \end{cases}. \tag{5.3}$$

The decision to use a flag matrix for the images' labels, instead of just a vector of index values, was made to be able to potentially expand the project to multilabel classification in the future.

After creating $\mathcal{F}$ and $\mathcal{L}$, both matrices are combined and saved as a single matrix $\mathcal{S}$ for later usage, defined as

$$\mathcal{S} = \begin{pmatrix} \mathcal{F} & \mathcal{L} \end{pmatrix}. \tag{5.4}$$

In the following sections $\mathcal{F}$, F, $\mathcal{L}$, L, N and $\mathcal{S}$ will refer to the matrices and dimensions as defined above.

The first script used is *import_to_numpy*. It processes the training or test image files and produces the matrix $\mathcal{S}$ as a two dimensional numpy array containing the images' activation features and their label flags.

In order to use *Caffe* for predicting no solver (see 3.1) is needed and some adjustments to the network architecture are necessary: The two original *Data* layers pointing to the training and test files have to be replaced with a generic input shape:

```
input: "data"
input_shape {
  dim: 100
  dim: 3
  dim: 227
  dim: 227
}
```

This causes the network to expect an input blob of 100 images, each with 3 channels (RGB) and a size of 227×227 pixels. Images too small or too large are automatically scaled by *Caffe* to fit the dimensions. This change basically hands over the control about which images to load to the python script using *Caffe*'s interface. Additionally, the layers after the target layers (*fc6 / fc7*

for this analysis) are simply removed from the architecture description. The adjusted network architecture is then saved as *deploy.prototxt* and its path stored in the python script as variable *DEPLOY_FILE*.

The path to the trained model is provided by the variable *MODEL_FILE* and has to be adjusted accordingly when creating activations for the three different models discussed in section 4.2.1.

For running the script three arguments have to be provided: The training or test information file, the label mapping file (see 4.1) and the target path and file name for the resulting numpy array. The script parses the image information file, creates batches of images and passes them to the method *evaluateImageBatch* 5.2.

The method utilizes *Caffe*'s python interface to pre-process and commit the image data and run a single forward pass. If the current batch is smaller than previously defined in the network architecture, the *data* input blob is adjusted. The activations in the target layer for each image are returned as one dimensional numpy array of length F. Finally, each activation array is extended by the flag array. All activations for the current batch, including their label flags, are returned as a list.

After all batches are processed, the list of activations is cast to a numpy array, resulting in the two dimensional array $\mathcal{S}$ with the dimensions $N \times (F + L)$, as defined above. For the hand sorted *Arachne* import, this results in one array $\mathcal{S}_{train}$ with the dimensions 7285×4128 for the training data and one array $\mathcal{S}_{test}$ with dimensions 1837×4128 for the test data. The arrays are stored as npy-files for later usage.

Figure 5.4 shows a plot of the activations for the training images.

Program 5.2: The method *evaluateImageBatch* prepares the image batch, runs the forward pass and returns the activations and labels as a list of numpy arrays.

```
1  def evaluateImageBatch(imageBatch, labelCount):
2      global net, transformer
3      # setup Caffe interface with given deploy configuration and model
4      if net == None or transformer == None:
5          setupCaffe()
6
7      batchActivations = []
8      # load and preprocess the image batch as defined in deploy config
9      imageData = map(lambda batchItem: transformer.preprocess('
           data', caffe.io.load_image(batchItem.get('path'))) , imageBatch)
10
11     # adjust the expected batch size for the input blob
12     # keep the other dimensions
13     net.blobs['data'].reshape(len(imageBatch), net.blobs['data'].shape
           [1], net.blobs['data'].shape[2], net.blobs['data'].shape[3])
14
15     # pass the image data to the net
16     net.blobs['data'].data [...] = imageData
17
18     # run a forward pass
19     results = net.forward()
20
21     counter = 0
22     for image in imageBatch:
23         # create the the flag array
24         labelFlags = np.array([0] * labelCount)
25         currentLabelId = np.array(image.get('labelIds'))
26         # flag the current image's label index
27         labelFlags[currentLabelId] = 1
28         # combine activations and label array
29         stacked = np.hstack((results[OUTPUT_LAYER][counter],
           labelFlags)))
30         batchActivations.append(stacked)
31         counter += 1
32
33     return batchActivations
```

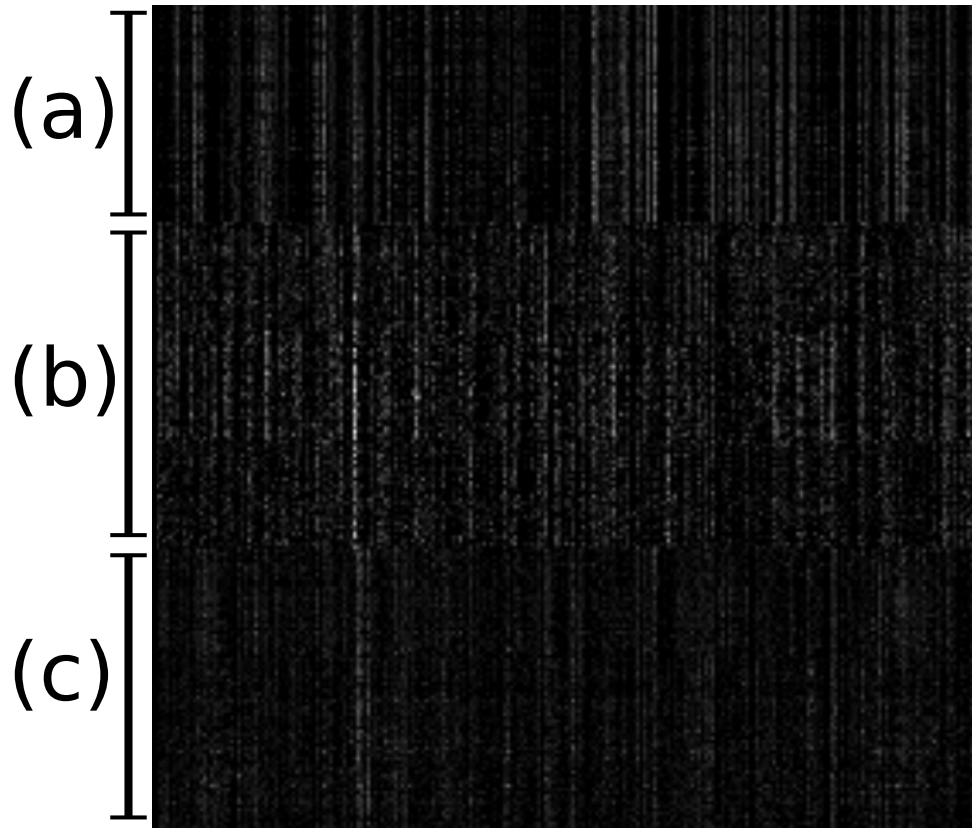


Figure 5.4: Plot of a section of activations on layer *fc7* using the *HybridCNN* model. Every row shows the activation produced by an image, vertically every 35 images are from one category. The group (a) shows the activations for example images with categories 'Print' and 'Handwriting', group (b) for images labeled as 'Pottery (Fragment)', 'Pottery (Complete)' and 'Pottery (Motive)'. The last group (c) contains images with the categories 'Plains', 'Mountains' and 'River'. Even though this is just an magnified section of the overall activations, the boundaries between the labels are visible in the horizontal. These differences between the labels are the foundation for the different classification approaches.

5.2.2 Naive Bayes

Classifier

As already described in section 2.1.2 the *Naive Bayes* classifier operates under the assumption that the probability of an input belonging to a certain category can be calculated by summing the logarithmic individual probabilities of its features belonging to that category. These probabilities are calculated using a set of training data.

The training consists of calculating the relevance of each feature for each

category in the training data. For the implementation, this means an array with dimensions $L \times F$. For an item that is supposed to be classified, first its features are multiplied element wise for each row in the resulting training array. Afterwards, the individual the probabilities per feature are summed up, resulting in an array of size L, containing the probability for each label.

The script *train_bayes* implements the method of the same name *trainBayes* 5.3. The method expects as parameters a training array S_{train} and the number of labels. It calculates the relevance for each feature per label, applies *Laplace Smoothing* and the element wise logarithm, as discussed in 2.1.2. The method returns the array P with the dimensions $L \times F$.

Array P can then be used for classification by first multiplying the input's features with P and then summing the individual probabilities per feature for each label. The index of the maximum value in the final prediction vector p corresponds to the classified label's index.

Testing

The script *evaluate_bayes* is used to apply the classifier to all test images. The script expects four parameters: The path to training array $\mathcal{S}_{train}$, to the test array $\mathcal{S}_{test}$, the path to the import's label mapping and a path used as target folder for the evaluation results.

The classifier expects only feature values $\in \mathbb{R}_0^+$, but some layers in the network may produce feature values $\in \mathbb{R}$. So the first thing the script does is to translate all feature values to co-domain $\mathbb{R}_0^+$. In order to create a confusion matrix later on, all test features are then separated by labels. The main loop iterates over all test features, applies the classifier and creates a confusion matrix and calculates the accuracy and mean average precision. The average precision is calculated as described in equation 4.3 by iterating over a sorted index list of the predictions.

```
1 averagePrecision = 0
2 relevant = 0
3
4 predictedLabelsSorted = np.argsort(predictions)[::-1]
5
6 for idx, value in enumerate(predictedLabelsSorted):
7   indicator = 0
8   if(value == testLabel):
9     indicator = 1
10    relevant += 1
11
12  precision = float(relevant) / (idx + 1)
13  averagePrecision += (precision * indicator)
```

Finally the confusion matrix is plotted and the result metrics are saved

Program 5.3: The method *calculateProbabilities* calculates the feature probabilities for each label and additionally applies *Laplace Smoothing* and the element wise logarithm.

```
1  def trainBayes(STrain, labelCount):
2      # Shape is (N,(F+L)), F = shape[1] - L
3      featureCount = STrain.shape[1] - labelCount
4      featureSumsPerLabel = []
5      counter = 0
6
7      while counter < labelCount:
8          labelIndex = featureCount + counter
9          # select subset by label flag
10         labelSelection = STrain[STrain[:, labelIndex] == 1]
11         # select only features from subset
12         labelSelection = labelSelection [:,0: featureCount]
13         # sum features of subset, result is vector of length F
14         featureSum = np.sum(labelSelection, axis=0)
15         featureSumsPerLabel.append(featureSum)
16         counter += 1
17     # cast to numpy array of shape (L,F)
18     featureSumsPerLabel = np.array(featureSumPerLabel)
19
20     # overall feature sum per label, result is vector of length L
21     overallSumPerLabel = np.sum(featureSumsPerLabel, axis=1)
22     # in order to divide element wise, tile to shape (L,F)
23     devisor = np.tile(overallSumPerLabel,(featureCount, 1)).T
24
25     # Calculating probabilities, apply Laplace Smoothing
26     P = (featureSumPerLabel + 1) / (devisor + featureCount)
27     # Apply element wise logarithm, array shape: (L,F)
28     P = np.log(P)
29     return P
```

to file. The *Naive Bayes* classifier does not have any parameters that could be individually tested.

5.2.3 k-nearest neighbours

Classifier

The *k-nearest neighbours* classifier is implemented in the script *kNearestNeighbours*. Its method *getKNearestNeighbours* expects an array $\mathcal{S}_{train}$ with

training features and an array $\mathcal{S}_{classify}$ containing the features for the set of items that are supposed to be classified. Additionally, the number of nearest neighbours k has to be specified.

Iterating over all input items, their Euclidean distances to all training data are calculated using numpy's *linalg.norm* function. A sorted list of the indices of the k nearest neighbours closest to the item is returned as the result.

The returned list can then be used to determine which label is predominant between an item's neighbours. That label then gets assigned as the label predicted for the item.

Testing

The *k-nearest neighbours* classifier is tested using the script *evaluate_KNN*. Again, paths to the training and the test arrays ($\mathcal{S}_{train}$ and $\mathcal{S}_{test}$) have to be provided, along the path to the import's label-index mapping and a target path for the evaluation results.

The value of k can affect the classifier's overall result (see figure 2.1, therefore the evaluation script will run the classifier with a range of different values for k, the default range being $1, 2, \ldots, \frac{N_{train}}{L}$. Given the default range, first all $\frac{N_{train}}{L}$ nearest neighbours are searched for each test item. Afterwards, the resulting array $\mathcal{N}$ of sorted neighbour relations with the dimensions $N_{test} \times \frac{N_{train}}{L}$ is used for classification multiple times, increasing the number of k considered neighbours. Analogous to testing the *Naive Bayes* classifier, for each iteration a confusion matrix is created and accuracy and mean average precision are saved. Finally, the accuracy and mean average precision for the different values and k are plotted.

5.2.4 k-Means

Classifier

k-Means itself is no classifier, but because *k-nearest neighbours* is a computational expensive classifier, *k-Means* is used to pre-process the training data by clustering it.

The script *kMeans_core* provides the basic functionality for *k-Means*, namely the initialization of new cluster centroids and the subsequent iterative clustering.

The method *runKMeans* returns a requested number of clusters for a training array. Additionally the maximum amount of iterations for running *k-Means* can be specified. The method first initializes the requested number of clusters. Each cluster consists of a centroid, representing its position in feature space, a list of indices indicating the training items assigned to it, and a histogram of the assigned items' labels.

The centroids get initialized at the position of random training items. *runKMeans* then starts to iteratively call the method *kMeansIteration*, which is updating the centroids' positions and then updating the assigned training points afterwards. The iterations stop if either the specified maximum of iterations is reached or the centroids did not change position the last iteration. Finally, clusters without training data assigned to them are removed from the list of clusters. The filtered list is then returned, together with the number of iterations it took to find the final centroids.

There are two types of classifiers using the *k-Means* clusters implemented. The first one creates a smaller number clusters per label, clustering in subgroups containing only one type of label, the second creates a larger amount of clusters, clustering all training activations without regard to their labels. The two types of clustering are implemented by the scripts *kMeans_per_label* and *kMeans_mixed* respectively.

kMeans_per_label first splits the provided training data by label, then calls *runKMeans* with each batch. The generated clusters are collected and saved to a specified target path. *kMeans_mixed* simply serves as an interface to the command line and forwards the specified training data, cluster count, label index mapping and target path to *kMeans_core*. After the clusters have been created, the script also saves them at a specified target path.

The classification decision for an unknown item is made by first evaluating the closest cluster. For that cluster, the label represented most in the cluster's label histogram is the label assigned to the item. For the per-label clusters there is always only one label present in the histogram, meaning the decision could be interpreted as a *1-nearest-neighbour* classification.

Testing

Testing *k-Means* is separated into generating and evaluation of clusters. In order to test different amounts of clusters k for the per-label and the mixed *k-Means*, the script *generate_clusters* automatically runs both types of clustering with different values for k. As default, the per-label clusters will be generated with $k \in (1, 2, 3, \ldots, 8)$, while the mixed clusters will be generated with $k \in (32, 48, 64, \ldots, 128)$.

Because the cluster centroids are initialized on the positions of random training data points, it is possible that those turn out disadvantageous for the formation of good clusters. As a consequence, for each value of k the clustering is done 3 times – trying to lessen the impact of bad picks on initialization for the overall evaluation. All generated clusters are saved to specified target folders for later evaluation. The evaluation itself is implemented in the script *evaluate_clusters*. It scans the folders containing previously generated clusters and utilizes them to classify a provided set of test data as described in the previous section.

Program 5.4: Example: ImageData layer usage

```
layer {
    name: "data"
    type: "ImageData"
    top: "data"
    top: "label"
    include {
        phase: TRAIN
    }
    transform_param {
        mirror: true
        crop_size: 227
    }
    image_data_param {
        source: "./path/to/train_info.txt"
        batch_size: 64
        is_color: true
        new_height: 256
        new_width: 256
        shuffle: true
    }
}
...
```

5.3 Training neural networks in Caffe

5.3.1 Preparing the training input

In order to train the network with the *Arachne* image set, the network's architecture has to be adjusted first.

The two *Data* layers have point to the training and test data. Training and test data can be supplied in different ways. The existing *Data* layers in the reference network architecture expect *Symas Lightning Memory-Mapped Database* (LMDB) files as sources [34]. *Caffe* provides a tool to create LMDB files from image files named *convert_imageset*. Instead of using *Data* layers, it is also possible to use *ImageData* layers. These use the import's training and test information text files directly, as shown in example program 5.4.

Loading the data from LMDB is more efficient, which should be considered given the fact that, while training, batches of training and test images are repeatedly picked and passed through the network. Also, image rescaling can be done by *convert_imageset* prior to training. Further input layers can

be found in *Caffe*'s layer documentation. After first working with *ImageData* layers, all training for the final analysis was done using *Data* layers.

5.3.2 Types of training

Different types of training will be tested, each started with different parameters in order to find the optimal configuration. For fine tuning, the model with best performance evaluated in 5.2 will be utilized. For all training variants, it is assumed the *AlexNet* is used as network architecture.

Training without a model

In order to start learning without an existing model, the only further changes that have to be made to the network structure is to adjust the number of outputs in layer *fc8* to the label count of L. *Caffe* already provides a solver defined for the *AlexNet*, which will be borrowed and has to point to the adjusted network architecture. Training can be started from the console with the following command:

```
1 /path/to/caffe caffe train -solver /path/to/solver.prototxt
```

Fine tuning

For fine tuning a model, the last layer again has to be adjusted to the new label count. But because within the model the weights for the original layer *fc8* are described, the adjusted layer has to be renamed, for example to *fc8_arachne*. Otherwise *Caffe* would fail to load the model. After adjusting and renaming layer *fc8*, fine tuning is started from command line by using the *weights* parameter.

```
1 /path/to/caffe caffe train -solver /path/to/solver.prototxt --
      weights /path/to/hybridCNN.binaryproto
```

Fine tuning only parts of the network

An alternative to fine tuning the complete model is to just fine tune specific layers. For all layers that are supposed to be kept as is, the layer's learning rate in the architecture has to be set to zero. Fine tuning will experimentally be restricted to either the convolutional or the fully connected layers.

Additionally, a last variant will be tested by keeping the complete network up to layer *fc7* static, while changing *fc8* to have 2048 outputs and adding a ninth fully connected layer with again L outputs. Basically this means that the complete original model is kept, while an additional layer is added and fine tuned close to the top of the network.

Chapter 6

Tests

To recapitulate: The experiments done in order to analyse the usability of artificial neural networks for archaeological image data are divided into two main parts. The first part tests the usability of already trained models for building four different classifiers without additional neural network training. The second part will deal with the results of either training a neural net from scratch or of fine tuning an existing model. All approaches will be analysed in detail and compared to one another.

6.1 Creating classifiers from layer activations

6.1.1 Overview

There were four classifiers tested for each of the three models, using two different inner product layers, *fc6* and *fc7*. Table 6.1 shows the results for the *Naive Bayes* classifiers and the best results for *k-nearest neighbours*, *k-Means* with clusters separated by label and *k-Means* with mixed labels. The optimal parameters for KNN and the two *k-Means* classifiers will be discussed later in this section. The overview already allows three observations:

- All classifiers that are created based on the *hybridCNN* model outperform those created using *Caffe*'s reference model and the *placesCNN* model.
- The classifiers KNN and *k-Means* mixed work better on layer *fc7*.
- In contrast, the *Naive Bayes* and *k-Means* per label classifiers generally show better results when trained on layer *fc6*, with *Naive Bayes* showing the bigger difference.

Table 6.2 shows the different runtimes using the 7285 training and 1837 test activations generated by the *hybridCNN* on layer *fc7*. Considering the runtime and classification performance, the mixed *k-Means* classifier is least useful, with training lasting more than an hour and producing the poorest

Table 6.1: Overview over the best results for each classifier, for all three models and their two fully connected layers *fc6* and *fc7*.

		reference		placesCNN		hybridCNN	
		Accuracy	MAP	Accuracy	MAP	Accuracy	MAP
fc6	*Naive Bayes*	0,7082	0.8141	0,6848	0.8043	0,7365	0.8379
	KNN	0,6603	0.7327	0,6543	0.7378	0,7109	0.7807
	k-Means per label	0.7267	0.7871	0.7174	0.7816	0.7632	0.8178
	k-Means mixed	0.6020	0.7230	0.5884	0.7216	0.6434	0.7654
fc7	*Naive Bayes*	0,6864	0.7994	0,6265	0.7639	0,7136	0.8249
	KNN	0,6973	0.7953	0,6880	0.7914	0,7201	0.8186
	k-Means per label	0,7125	0.7862	0,6641	0.7629	0,7468	0.8095
	k-Means mixed	0,6173	0.7423	0,5672	0.7117	0,6668	0.7762

Table 6.2: Overview runtime for training and classification per classifier using the activation features of 7285 training and 1837 test images generated by the *hybridCNN* model on layer *fc7*. The runtime for *k-Means* per label and *k-Means* mixed was measured for the optimal k clusters, as discussed later in sections 6.1.4, 6.1.5 and 6.1.6. All training and classification was done on a single 2.40GHz CPU. The time format is (hh:mm:ss.f).

	Training	Classification
Naive Bayes	00:00:03.38	00:00:03.30
KNN	00:00:00.00	00:24:15.09
k-Means per label	00:03:07.79	00:00:15.42
k-Means mixed	01:14:03.62	00:00:11.09

results overall. *Naive Bayes* and the *k-Means* per label approach are the most promising. The time needed to classify a single image with KNN increases with the amount of training data, making it increasingly impractical if the data set would be extended further.

In order to better visualize and evaluate classification results, confusion matrices can be plotted. A confusion matrix gives an overview over what images belonging to certain categories have been classified as. Figure 6.1 shows the confusion matrix of results for the *Naive Bayes* classifier, applied to the layer *fc7* using the *hybridCNN* model.

Each row shows what the test images belonging to the label specified on the left hand side have been classified as. For example, there are 25 test images for the category 'Fragment', 22 of which have been classified correctly as 'Fragment', but also one as 'Pottery (fragment)', one as 'Sculpture (detail) and one as 'Column'. These mistakes already highlight the fact of blurred categories, as discussed earlier: While the category 'Fragment' contains stone fragments, it is possible that some fragments could also be interpreted as

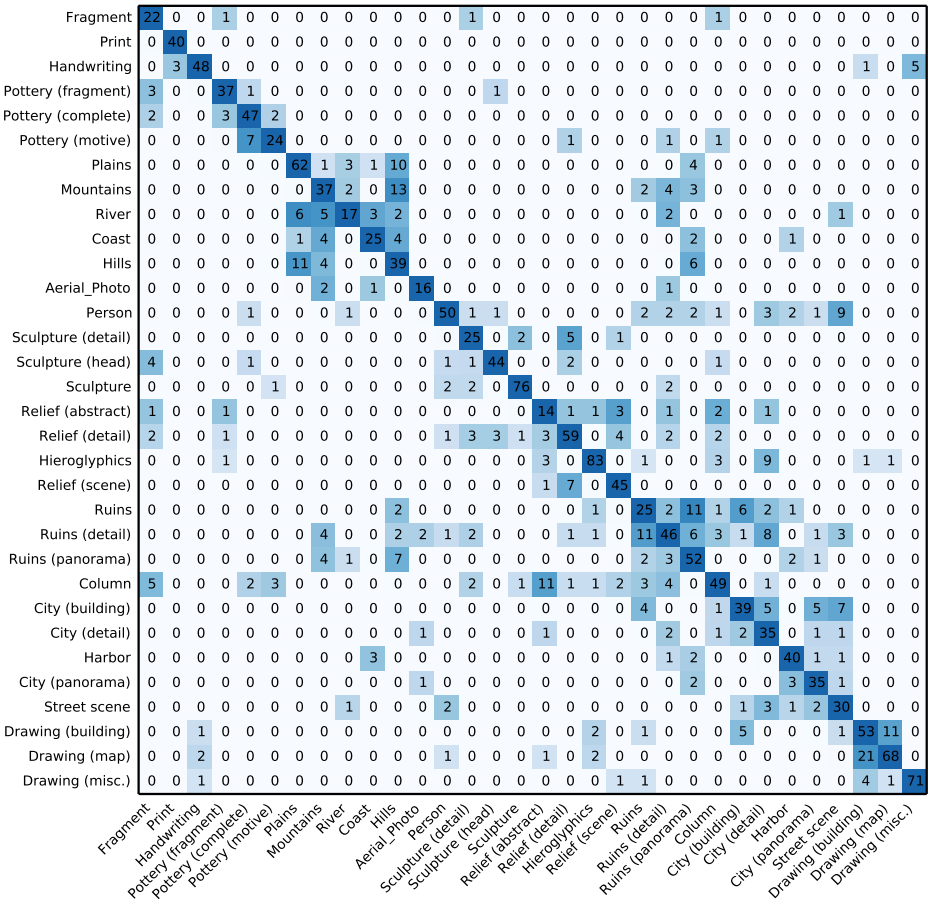


Figure 6.1: Confusion matrix produced by the *Naive Bayes* classifier using the *hybridCNN* model and activations on layer *fc6*.

parts of a sculpture or a broken column.

A perfect classification of all test images would have resulted in a single diagonal line.

6.1.2 Comparing the models

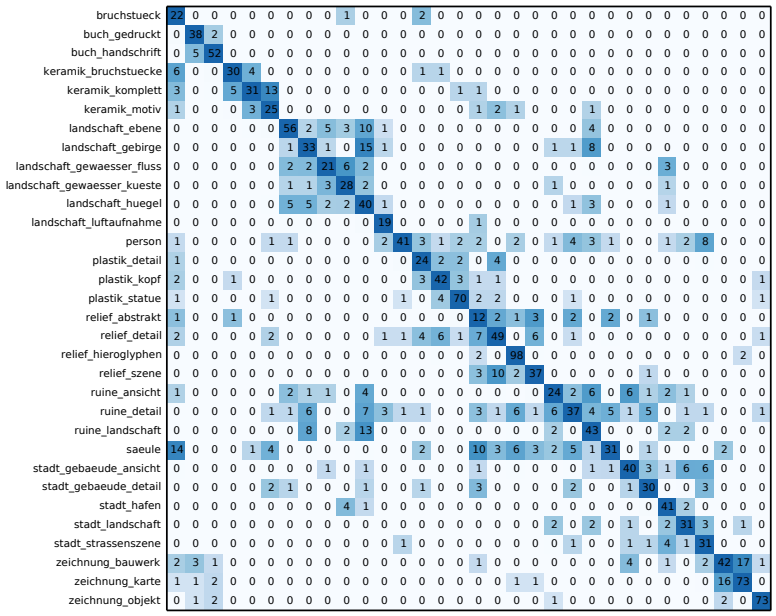
Given the observation that the *hybridCNN* outperforms both other models, the question arises, if the respective specialization of the *placesCNN* model on places and buildings compared to *Caffe*'s reference model is reflected in the classification results. Figure 6.2 shows the confusion matrices of the best results produced by the KNN classifier for both models trained on layer *fc7*.

The difference in correct classifications between both models is also shown in figure 6.3. The *placesCNN* made less mistakes for the labels that are associated with cities, ruins and most landscapes.

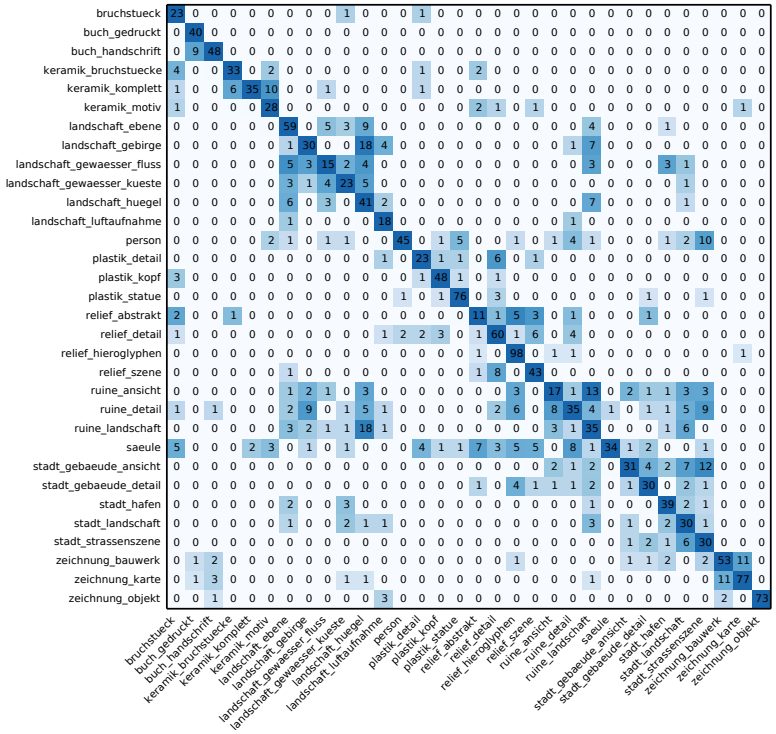
The direct comparison shown in figures 6.4 and 6.5 between the *hybridCNN* model and the other two shows that the *hybridCNN* either surpasses the other two models or diminishes the margin. The *hybridCNN* fails to match either model in the two drawing categories 'Drawing (map)' and 'Drawing (misc.)' as well as categories 'City (detail)' and 'Column'. For the drawing categories one could argue that the result is due to the fact that none of the models had a focus on drawings or sketches in their respective training data sets.

The results would support the initial assumption, that the image data used for training a model influences the performance of classifiers utilizing deeper layer activations.

After having found the *hybridCNN* to be the model with the best performance overall, it will be used in the following sections in order to analyse the different classifiers in detail.



(a)



(b)

Figure 6.2: The confusion matrices for the *k-nearest neighbours* classifier applied to the *placesCNN* (a) and reference (b) models on layer *fc7*.

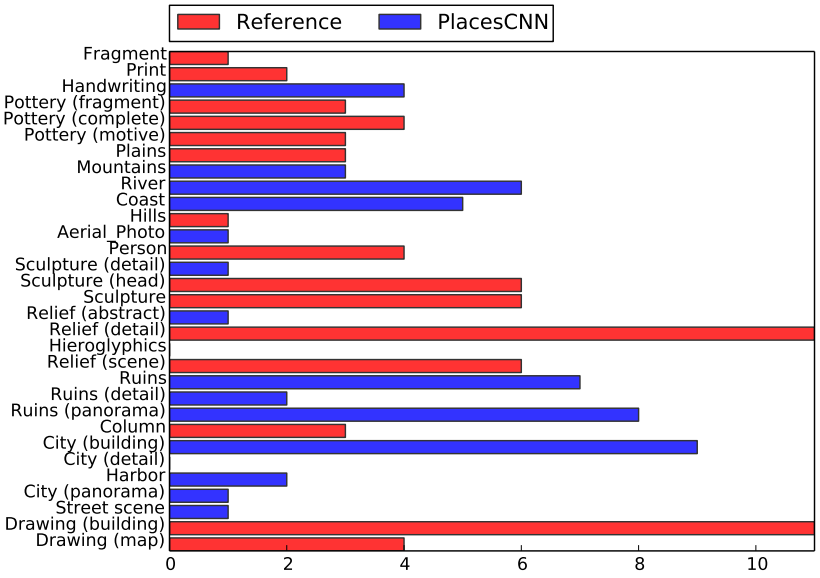


Figure 6.3: Comparison of the correct predictions produced by *k-nearest neighbours* using the *placesCNN* model and the reference model on layer *fc7*. Each bar denotes the positive margin in correct predictions compared to the other model.

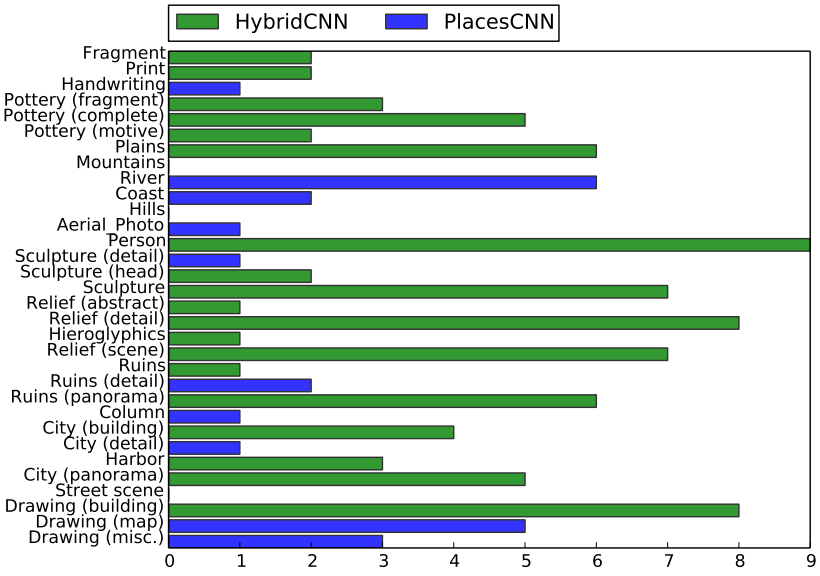


Figure 6.4: Comparison of the correct predictions produced by *k-nearest neighbours* using the *hybridCNN* model and *placesCNN* model on layer *fc7*. Each bar denotes the positive margin in correct predictions compared to the other model.

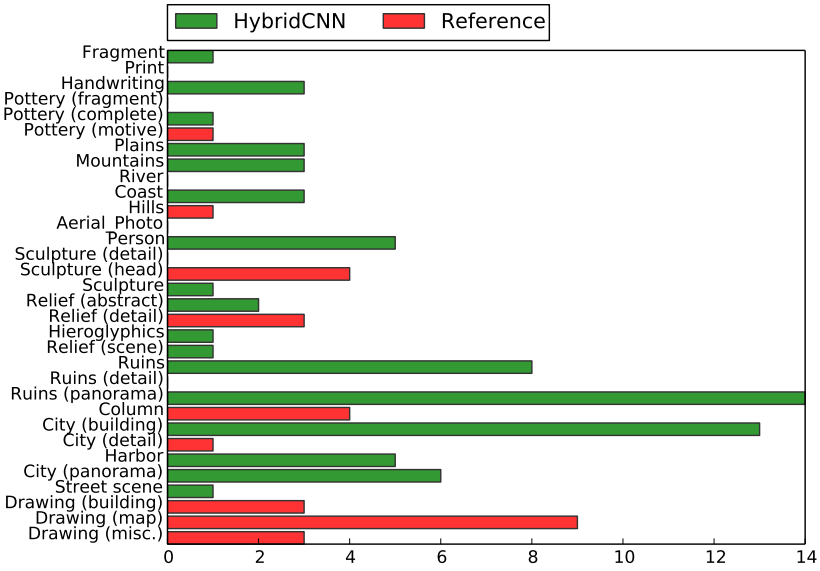


Figure 6.5: Comparison of the correct predictions produced by *k-nearest neighbours* using the *hybridCNN* model and reference model on layer *fc7*. Each bar denotes the positive margin in correct predictions compared to the other model.

6.1.3 Naive Bayes

Plotting the two confusion matrices produced by the *Naive Bayes* classifier on layer *fc6* shows how the classifier struggles with the fuzzy categories.

There are two big blocks along the diagonal, one formed in the area of, in a broader sense, 'landscape' categories, the other around categories that concern buildings. Besides these bigger blocks some 'Person' images are put into city categories and some 'Landscape' categories are classified as 'Ruins' or 'Ruins (panorama)'.

These decisions could be interpreted as sensible, as one could argue that lacking multilabel (per image) classification, those errors were expected to occur. The relatively high mean average precision of 0.8379 and 0.8249 (see table 6.1) may support the argument that the correct predictions came atleast close to being the first choice.

As already mentioned, the *Naive Bayes* classifier produces better results using the activations of layer *fc6*. This could be explained by the fact that the *Naive Bayes* classifier implicitly assumes that all features, in this case the layer activations, are independent from one another.

As described in sections 2.3 and 2.4.7, convolutional neural networks implement deep learning by learning to first break down the input data into abstract features, and then recombining those abstract features into increas-

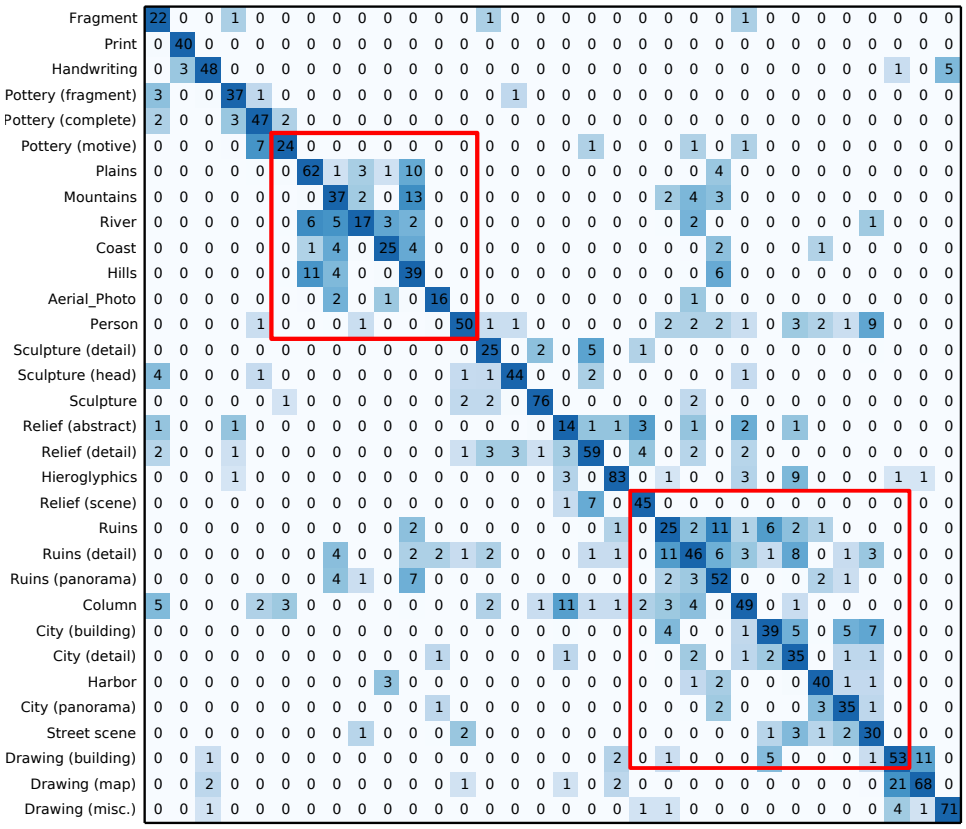


Figure 6.6: Confusion matrix for the *Naive Bayes* classifier using the *hybridCNN* model and activations on layer *fc6*. Groups of similar categories causing confusion are marked in red, namely categories concerning landscape or architecture

ingly complex concepts. This could explain the fact, that the *Naive Bayes* classifier works better close to the convolutional layers, that are responsible for creating abstract features. The more fully connected layers follow those abstract features, the learned concepts get more complex, which could also mean that the different activations get increasingly correlated to one another, forming patterns of activation that get interpreted by the following layers - thereby also increasingly contradicting the *Naive Bayes* classifier's assumption.

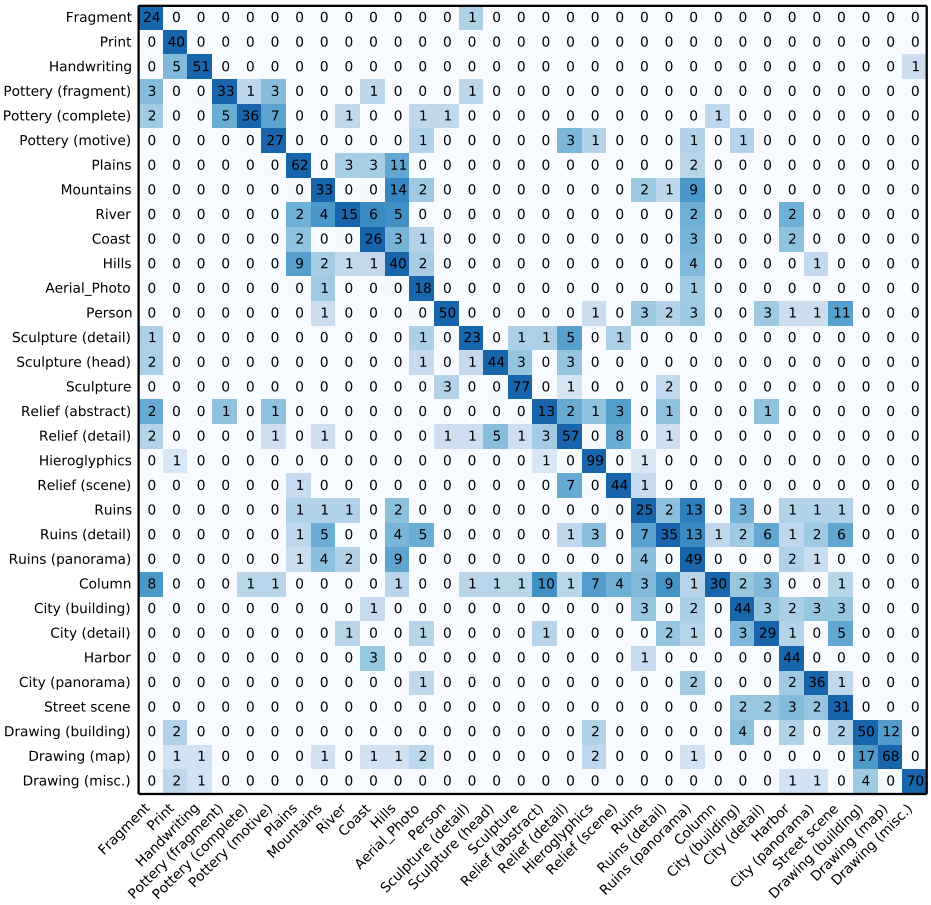


Figure 6.7: Confusion matrix produced by the KNN classifier, $k = 18$, using the *hybridCNN* model with layer *fc7*. The dashed line marks the maximum values.

6.1.4 k-nearest neighbours

The confusion matrix for *k-nearest neighbours* shows patterns similar to those produced by the *Naive Bayes* classifier. Again, the classifier fails classifying blurry categories, but the relatively high mean average precision may indicate that the correct category on average was close to being picked.

Plotting the accuracy and mean average precision over the different values of k (see figure 6.8) shows a relatively constant accuracy up until $k \approx 33$. The two categories with least images in the test dataset have only 20 and 25 images. This could mean that at around 33 neighbours, the smaller categories are starting to be classified incorrectly because of too big 'neighbourhoods', where they do not make up a majority any more.

The mean average precision increases up to a maximum for $k = 18$, then

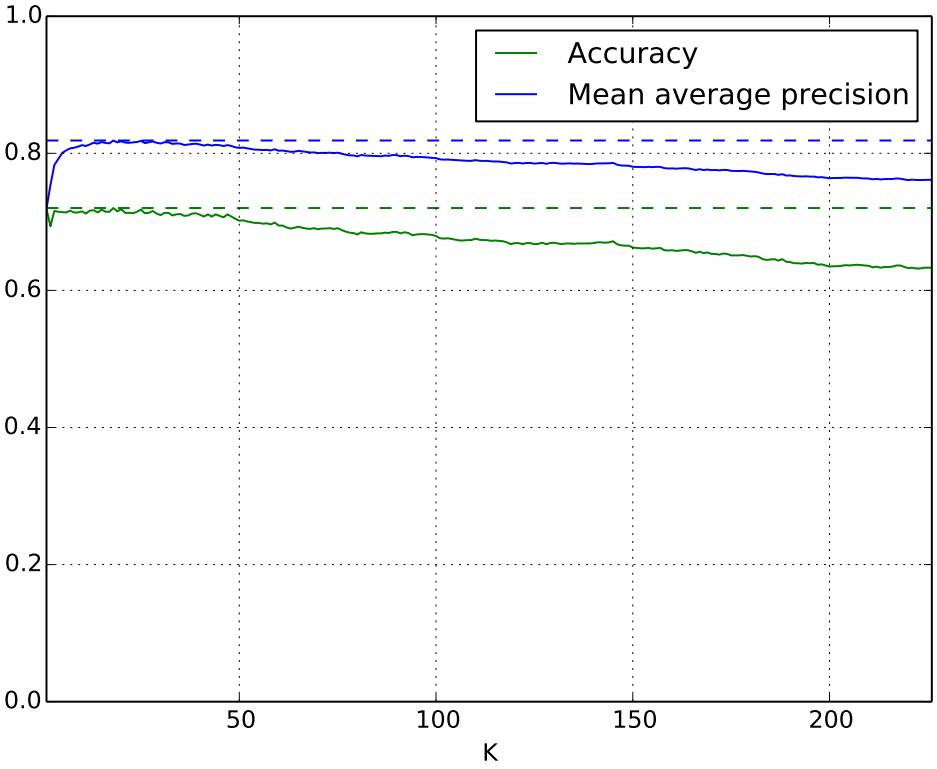


Figure 6.8: Accuracy and mean average precision over different k for the *k-nearest neighbours* classifier using the *hybridCNN* model with layer *fc7*.

starts to decrease too. Given the relative constant accuracy up to $k = 30$, $k = 18$ could be interpreted as the best value for classification. A high mean average precision would mean, that for the test cases where the final prediction was wrong, the correct prediction atleast came close to being chosen in the classification decision.

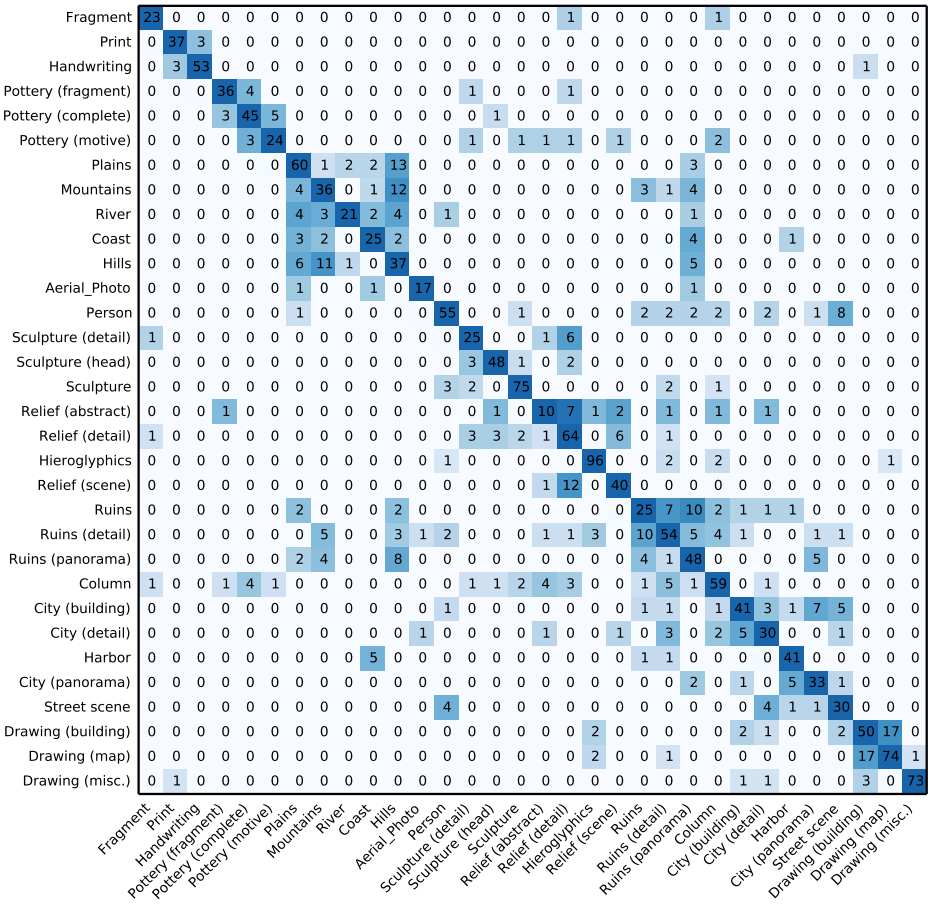


Figure 6.9: Confusion matrix produced by the *k-Means* per label classifier, $k = 8$, using the *hybridCNN* model with layer *fc6*.

6.1.5 k-Means per label

The confusion matrix shown in figure 6.9 produced by the *k-Means* per label classifier again is similar to the ones produced by the other two classifiers.

Figure 6.10 shows that increasing the amount of k clusters per label is only marginally increasing the accuracy. This would suggest that the images per category are grouped close to each other in feature space, forming just one or only very few clusters.

The figure also shows that the mean average precision is falling when increasing the number of clusters per label. This observation would also support the assumption of few clusters per category: While one cluster per label yields good results for most images with that label, adding more clusters may help to correctly classify more exotic outliers (hence the marginally increased accuracy) correctly, while at the same time being in further Eu-

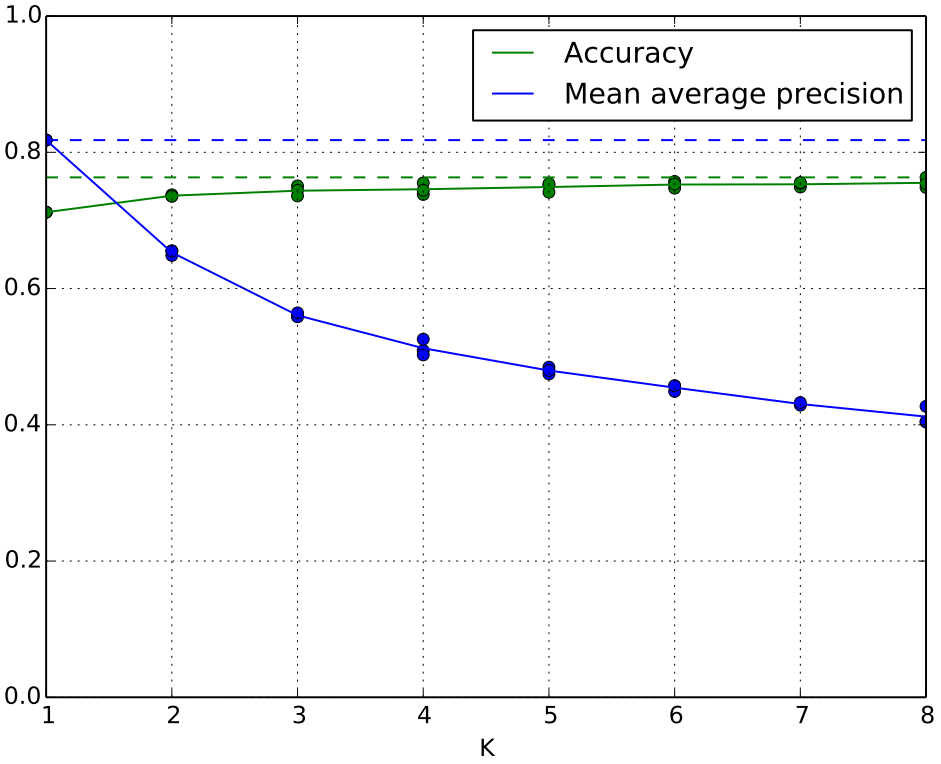


Figure 6.10: Accuracy and mean average precision over different amount of clusters k for the *k-Means* per label classifier using the *hybridCNN* model with layer *fc6*.

clidean distance from the bulk of images in the category, reducing their average precision.

To give an intuition about the results, we first recapitulate (see 4.3), that the average precision AP was defined as

$$AP = \frac{1}{r} \sum_{k=1}^{N} Precision(k) \times rel(k).$$

Assuming only one cluster per label and that the bulk of images per label would get classified correctly. The average precision for each image in that majority would then be calculated as

$$\begin{aligned} AP &= \frac{1}{1}(\frac{1}{1} \times 1, \frac{1}{2} \times 0, \ldots, \frac{1}{N} \times 0) \\ &= 1. \end{aligned}$$

In contrast, given three clusters per label, there may be two clusters representing just a small amount to outliers. These outlying clusters will have their centroids further away in Euclidean space from the majority of images for that label. The average precision for the majority of test images in that category may then be given as

$$\begin{aligned} AP &= \frac{1}{3}(\frac{1}{1} \times 1, \frac{1}{2} \times 0, \ldots, \frac{1}{30} \times 1, \ldots, \frac{1}{56} \times 1, \ldots, \frac{1}{N} \times 0) \\ &\approx 0,35 \end{aligned}$$

where the outlying clusters come in 30th and 56th place when sorting the centroids' distances from the majority of the labels test images.

The mean average precision MAP is defined as the sum over the average precisions of all test images, divided by the number of test images. The improved average precision for some outliers has few impact on the overall MAP, given that the average precision for the majority of test images has in contrast drastically decreased.

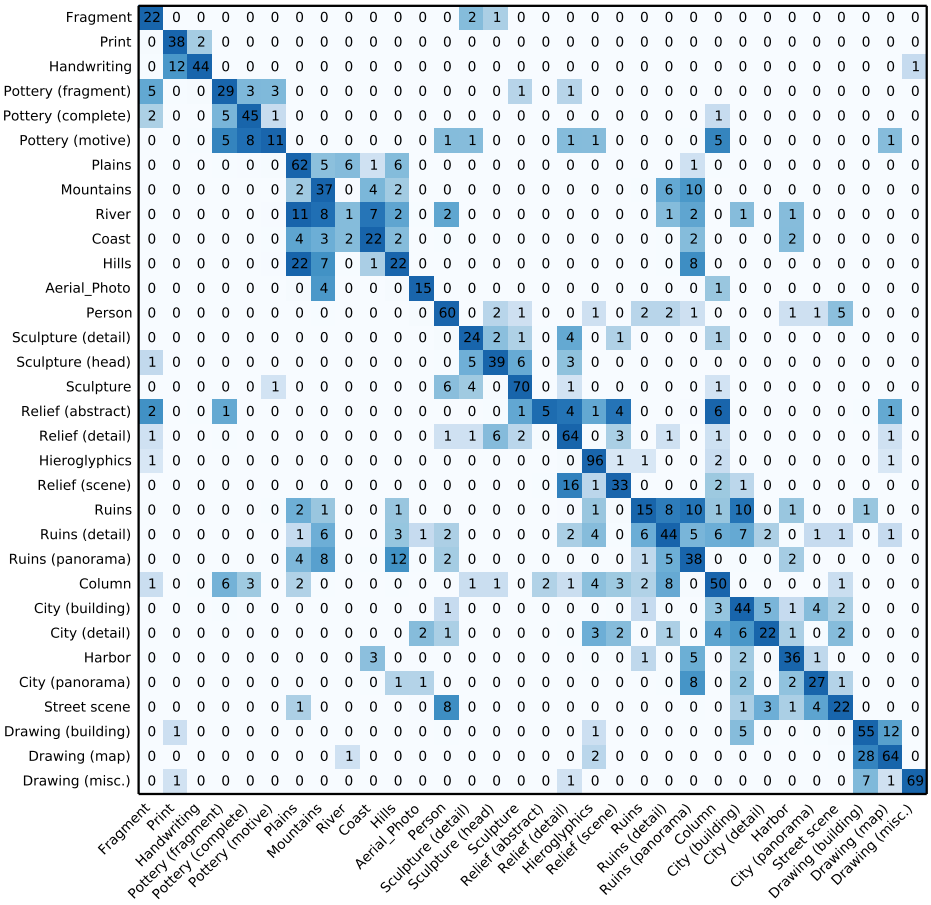


Figure 6.11: Confusion matrix produced by the mixed *k-Means* classifier, $k = 120$, using the *hybridCNN* model with layer *fc7*.

6.1.6 mixed k-Means

Again, the confusion matrix generated by the mixed *k-Means* classifier is similar to the previous ones, as shown in figure 6.11.

The classifier's performance can be improved by adding more clusters, as shown in figure 6.12. But in comparison to *k-Means* per label, using the same model and layer and 32 clusters, the mixed classifier achieves an accuracy of only 0.5329, while the per label classifier produces an accuracy of 0.6929 (one cluster per label). Combined with the longer training time, the mixed *k-Means* classifier seems not to be the best choice for the task.

The reason for the poorer performance is probably the implementation of the classification decision. The mixed *k-Means* classifies by first evaluating the closest cluster to a test image, and then choosing the category that was assigned most to that cluster. As can be seen in figure 6.13, when using only

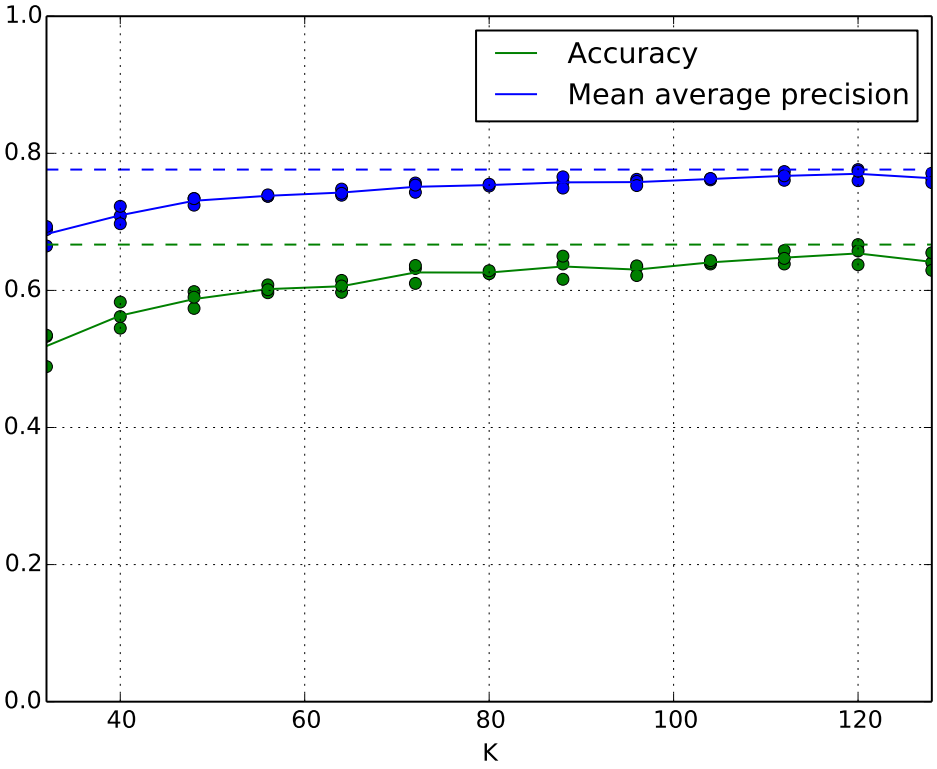


Figure 6.12: Accuracy and mean average precision over different amount of clusters k for the mixed *k-Means* classifier using the *hybridCNN* model with layer *fc7*.

32 clusters, some labels like 'Pottery (complete) make up the maximum in two clusters, causing others like 'Relief (abstract)' to be a minority in all clusters, making a classification decision for 'Relief (abstract)' impossible.

Increasing the cluster count helps to rectify that error, as shown in figure 6.14, but more clusters also mean a longer training- and classification time. The same figure also illustrates, that the confusion observed in the classification tests caused by blurry categories is already present while clustering. The landscape categories for example form a block similar to the ones already observed in the confusion matrices, first encountered in figure 6.6. The number of training images per category can also be observed: Categories with more images are dominant in more clusters, see figure 5.3 for comparison.

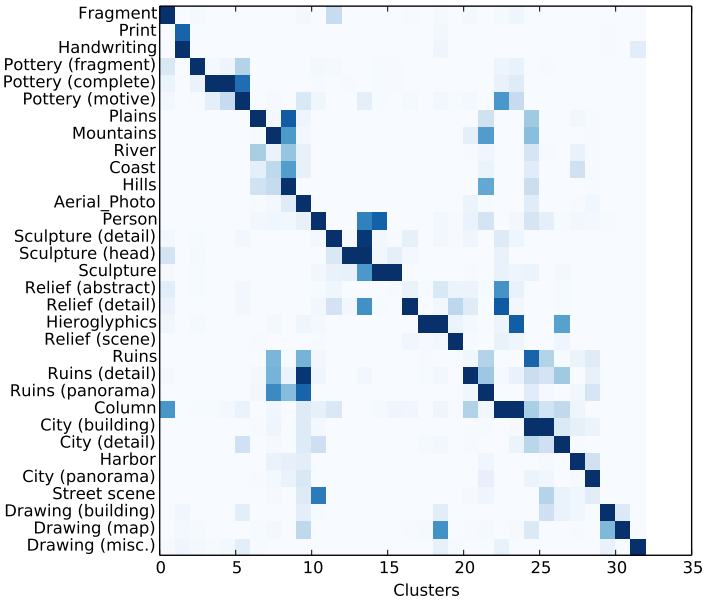


Figure 6.13: The distribution of label in 32 clusters, as generated by training the mixed *k-Means* classifier using the *hybridCNN* model with layer *fc7*. The darker the colour, the more the label is present in the given cluster.

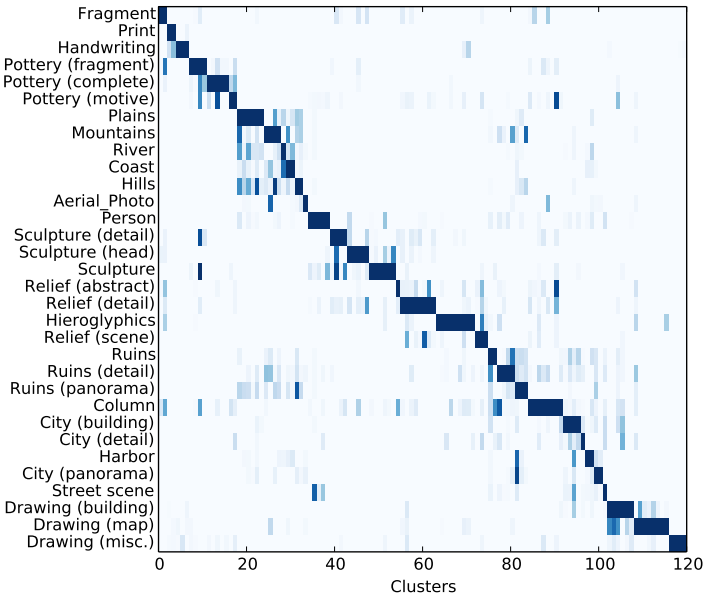


Figure 6.14: The distribution of labels in 120 clusters, as generated by training the mixed *k-Means* classifier using the *hybridCNN* model with layer *fc7*. The darker the colour, the more the label is present in the given cluster.

Table 6.3: Accuracy and mean average precision results for the different fine tuning configurations after 10000 training iterations.

	Accuracy	MAP
Training without a model	0.6761	0.7822
Fine tuning complete model	0.8094	0.8876
Fine tuning only fully connected layers	0.8051	0.8842
Fine tuning only convolutional layers	0.7991	0.8794
Fine tuning added layers	0.7838	0.8734

6.2 Training a network as classifier

6.2.1 Overview

Table 6.3 shows the best results for each of the different training and fine tuning configurations evaluated. Training without using some model produces the poorest results, while the fine tuning approaches achieve comparable accuracy and mean average precision, with fine tuning additional layers falling behind the other three. The best results are achieved by fine tuning all network weights.

6.2.2 Training without a model

The first type of training tested was the training without using a pre-trained model. The *AlexNet* architecture used for originally training the *hybridCNN*, *placesCNN* and *Caffe* reference model was designed for the *Large Scale Visual Recognition Challenge 2012* (ILSVRC2012), meaning it was designed to be trained on millions of images. Because the image data set imported from *Arachne* is much smaller, the problem of over fitting was foreseeable. The complexity of the network architecture is such, that for smaller training data sets the network may 'memorize' the training data perfectly, but then will be unable to make meaningful predictions for new data. This was already introduced as the high variance, or over fitting problem.

Figure 6.15 illustrates this problem: Using the *AlexNet* architecture without changing anything except the last layers in order to accommodate the different label count in the *Arachne* data set, the figure shows the training progress over 10000 iterations. More specifically, the figure shows plots of the training data's and test data's loss and the prediction accuracy for the test data.

At first both training and test data loss decrease rapidly while the accuracy for predicting the test images increases. After roughly 1000 iterations the two loss graphs diverge: While the training data loss continues to decrease to almost 0 and stays low, the test data loss starts to increase again

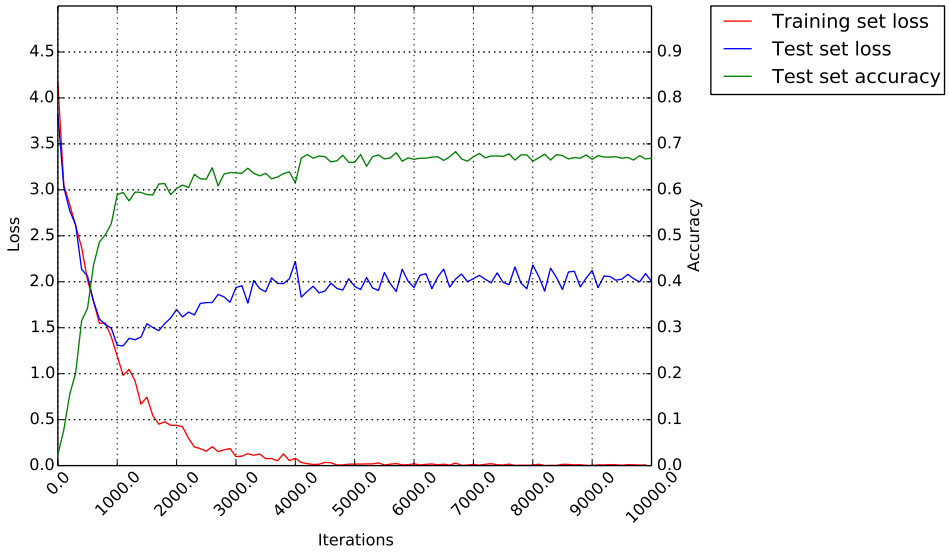


Figure 6.15: The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

and then keeps fluctuating around a loss value of 2. The accuracy flattens out at ≈ 0.67. This means that even though the network is able to predict its training data almost perfectly (training loss ≈ 0), the network does not generalize well and is unable to predict the unknown test images with an accuracy above 0.67.

Using more image data from *Arachne* would have been a possible solution to this problem, but was not feasible due to time constraints for this thesis. To counter this, it was decided to use *Caffe*'s fine tuning features. Instead of training the network 'from scratch', the already trained *hybridCNN* model is used to initialize the network's weights and then trained further using the *Arachne* data.

6.2.3 Fine tuning the complete model

Using the same basic configuration as described in the previous section, the network was trained again, but instead of letting the weights be initialized randomly, the *hybridCNN* model was used. Figure 6.16 again shows the loss values and accuracy over 10000 training iterations.

Using this configuration, the loss values both increase dramatically to a value of ≈ 88 after only a few iterations, while the accuracy fluctuates around ≈ 0.04. These results are obviously far worse than those produced by training without a model. Both a high training and test set loss indicates a high bias: The network is unable to find a good model for making predictions, neither for the training nor for the test images. The fact that the loss values

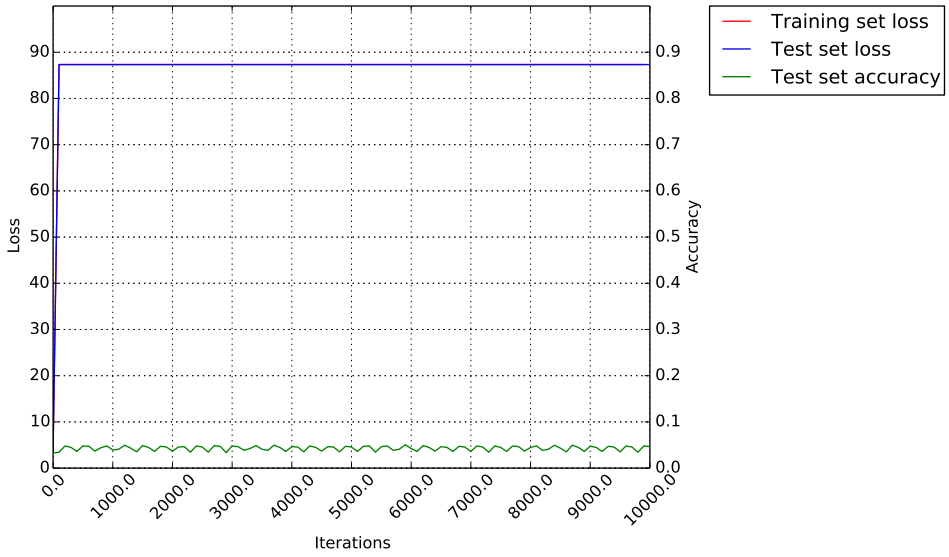


Figure 6.16: The *hybridCNN* was used for fine tuning the *AlexNet*. The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

jump rapidly to such a high value could indicate that the learning rate is too high.

As discussed in section 2.4.4, if the gradient descent steps are too large, it is possible that they overstep the minimum the algorithm is supposed to reach. In this case, it seems like the gradient descent is not only slowed down by overstepping the minimum (as depicted in figure 2.8), but in fact the steps are so large, that they are escalating and causing the weight values to move further and further away from those that would produce the loss minimum.

In *Caffe* the initial learning rate at the training start is defined in the *solver.prototxt*. The *AlexNet* starts with a default learning rate of 0.01, which is then reduced every 100000 iterations by a factor of 0.1. In order to use the *hybridCNN* model for fine tuning, using the original learning rate is not viable, considering that the network already was trained 700000 iterations and thus finished with a learning rate of $0.1 \times (0.1^6) = 0,0000001$. In order to find the optimal initial value for the training rate, several training runs were started, while decreasing the initial learning rate each time.

The best result was achieved using an initial training rate of 0.0001, the progress is shown in figure 6.17. Both training and test loss fall rapidly to ≈ 0.13 at iteration 1000. Afterwards the networks again begins to overfit the training data, reducing the training loss further but producing a flattened test loss output. The test accuracy keeps fluctuating around 0.81 after the

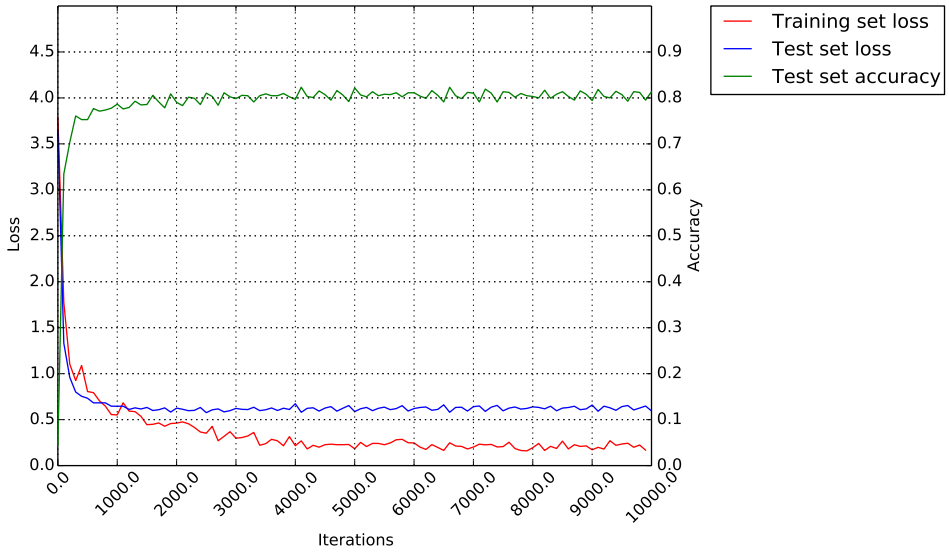


Figure 6.17: The *hybridCNN* was used for fine tuning the *AlexNet* with a reduced initial learning rate of 0.0001. The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

first 3000 iterations.

The strategy of changing the initial learning rate stepwise was applied to all following types of configuration in order to produce better results. Only those best results are presented in the following sections.

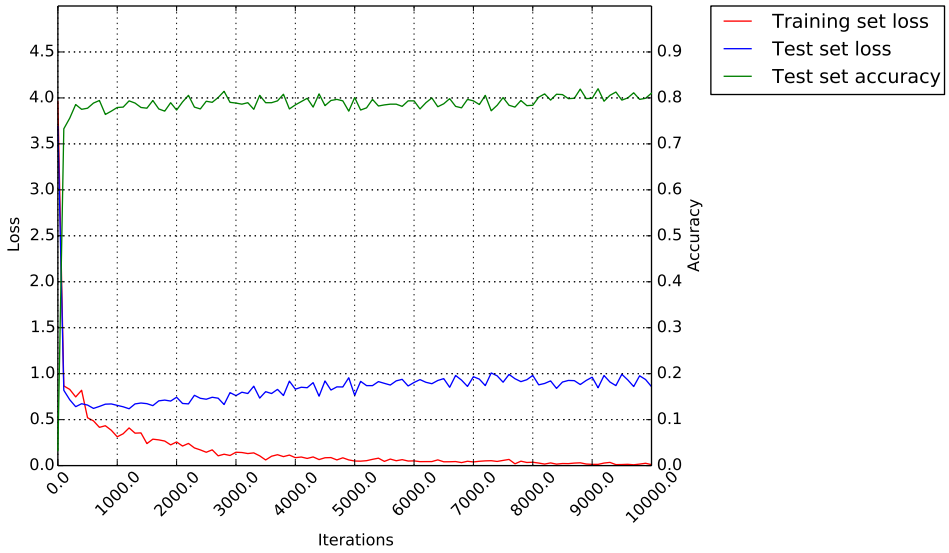


Figure 6.18: The *hybridCNN* was used for fine tuning the *Convolution* layers with a reduced initial learning rate of 0.001. The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

6.2.4 Fine tuning only convolutional layers

Fine tuning only the *Convolution* layers produces an accuracy comparable to fine tuning the complete network. This approach tends stronger towards over fitting, with a final training loss of ≈ 0, and a higher test loss than the previous approach.

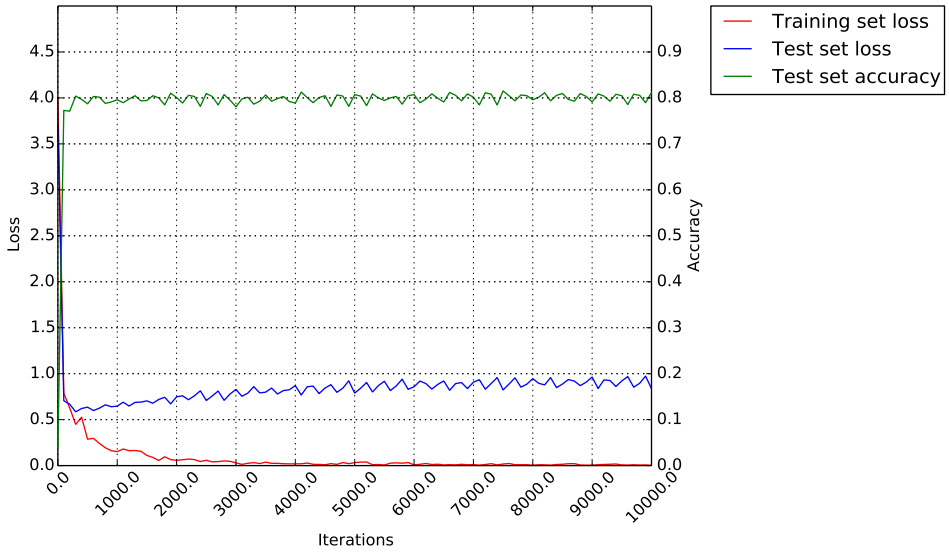


Figure 6.19: The *hybridCNN* was used for fine tuning the *InnerProduct* layers with a reduced initial learning rate of 0.001. The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

6.2.5 Fine tuning only fully connected layers

Fine tuning only the fully connected layers, responsible for deriving concepts from the features produced by the convolutional layers, also yields an accuracy comparable to the fine tuning of the complete network, as shown in figure 6.19. The over fitting on the other hand is more pronounced, with a final training loss of ≈ 0 and a test loss of ≈ 1.9.

Training only the convolutional or only the fully connected layers causes more over fitting than fine tuning the complete network. All three approaches produce roughly the same accuracy. In other words, even though the fraction of correct predictions is almost the same for all resulting models, the two partially fine tuned models are 'less certain' about their results – making the completely fine tuned network the potentially more reliable classifier.

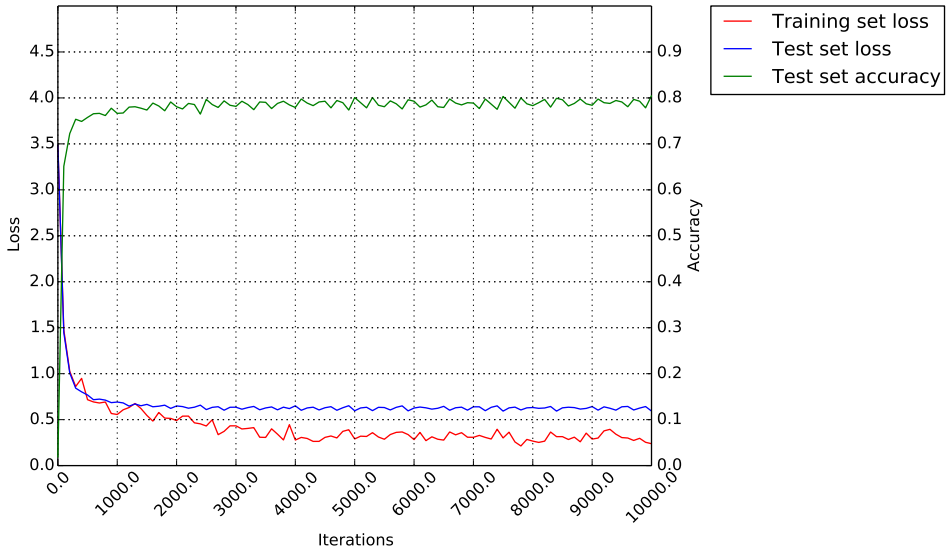


Figure 6.20: The *hybridCNN* was used for fine tuning an extended version of the *AlexNet*, featuring an additional *InnerProduct* layer, with a reduced initial learning rate of 0.001. The figure shows plots of the training- and test loss, as well as the test accuracy over 10000 training iterations.

6.2.6 Fine tuning only added layers

The last training variant was the fine tuning of an enhanced *AlexNet*. Instead of fine tuning layers *fc1* to *fc7*, a new fully connected layer *fc8_arachne_added* with 2048 outputs was introduced, followed by a last *InnerProduct* layer with outputs the number of labels. The best variation of this network configuration had a a learning rate of 0.001, as shown in figure 6.20, with an accuracy of ≈ 0.79. The over fitting is less severe than observed while fine tuning only specific layer types.

6.2.7 Best result

Fine tuning the complete network was the configuration producing the best accuracy results. The confusion matrix for the configuration is shown in figure 6.21. As before with the classifiers based on deeper layer activations, the blurry categories are responsible for the bulk of misclassification instances. In general, when plotting the confusion matrix for the different configurations discussed in this section, the same basic pattern emerged. Together with an accuracy of ≈ 0.8 for all fine tuning approaches, this suggests that an improvement of accuracy will rather be accomplished by reworking the data set than by evaluating more fine tuning configurations.

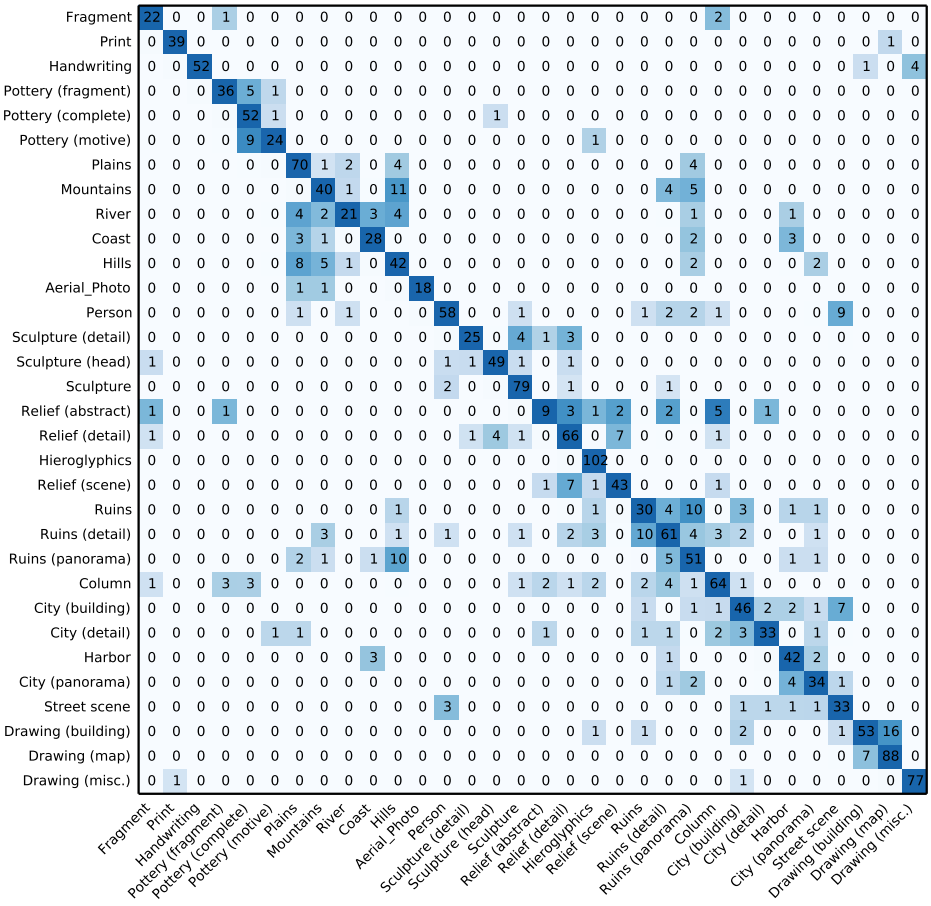


Figure 6.21: The confusion matrix produced by fine tuning the *AlexNet*'s *InnerProduct* layers using the *hybridCNN* model and an initial learning rate of 0.0001

6.3 Results

A comparison of the best results per classifier type is shown in figure 6.22. Fine tuning models yields better results than the classifiers trained on deeper layer activations using existing models. Fine tuning the complete network produced the overall best results. Nevertheless, the performance gap between the *Naive Bayes* and *k-Means* per label classifier on the one hand and the fine tuned networks on the other turned out smaller than initially expected.

For the fine tuning approaches, there is few difference between fine tuning the complete network, fine tuning only the convolutional layers or only the fully connected layers – keeping in mind that the latter two are more prone to overfitting. All three achieve an accuracy of roughly 0.8 and a mean average precision of around 0.88. Training the network without using a pre-trained model produces decent results, but is, as already anticipated, inferior to the fine tuning approaches. The reason could be the relatively small dataset, for which the *AlexNet* may be overbuilt.

The *Naive Bayes* classifier produces the best mean average precision of all classifiers based on deeper layer activations, while the *k-Means* per label classifier achieves the best accuracy in that category. Both produced better results when applied to the deeper fully connected layer *fc6*, which could mean that layer activations closer to the features produced by the convolutional layers are better suited to train classifiers. The mixed *k-Means* classifier falls behind in runtime and prediction results, making it the overall weakest one. Initially the hope was to see more differences between the probabilistic approach of the *Naive Bayes* classifier and the three feature space based classifiers.

All fine tuning attempts seemed to be capped at an accuracy of ≈ 0.8. Combined with the observation that *all* tested classifiers show a similar confusion pattern (see figure 6.23), it could be argued that the image data's diffuse categories are the reason that no higher accuracy was achieved.

Reworking the image dataset and its categories could be a valid strategy for improving accuracy.

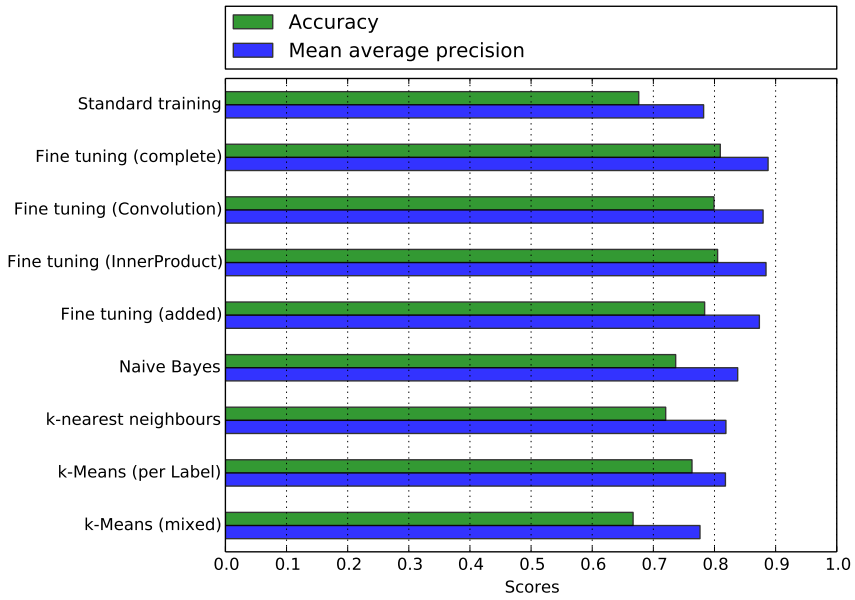


Figure 6.22: Overview of the best classification results per classifier type.

6.4 Problem analysis

The overall good mean average precision values indicate that the correct categories atleast came close to being the predicted category. For overlapping categories this means that many of the classifiers' wrong predictions could nevertheless be considered reasonable. Because the archaeological images at hand could potentially be assigned to multiple categories each, it may be a good idea to proceed by evaluating multilabel classification techniques in the future. Figure 6.24 shows a test image that was initially categorized as 'River', but got finally classified as 'Plains' by the fine tuned model. It could be argued that both categories are valid, and they actually are the top two predictions.

The selection of categories used in this analysis was far from complete. Figure 6.25 shows how the fine tuned network fails to classify an image from without the dataset. This is because a matching category (for example 'Vegetation') simply does not exist in the training- and test data used to fine tune the network.

The intuition when choosing the model for classifiers without additional network training originally was to choose a model whose training data best reflected one's own image data. The test results indicate that this intuition seems to be correct. The *hybridCNN* outperformed both the *placesCNN* and *Caffe*'s reference model with all tested classifiers.

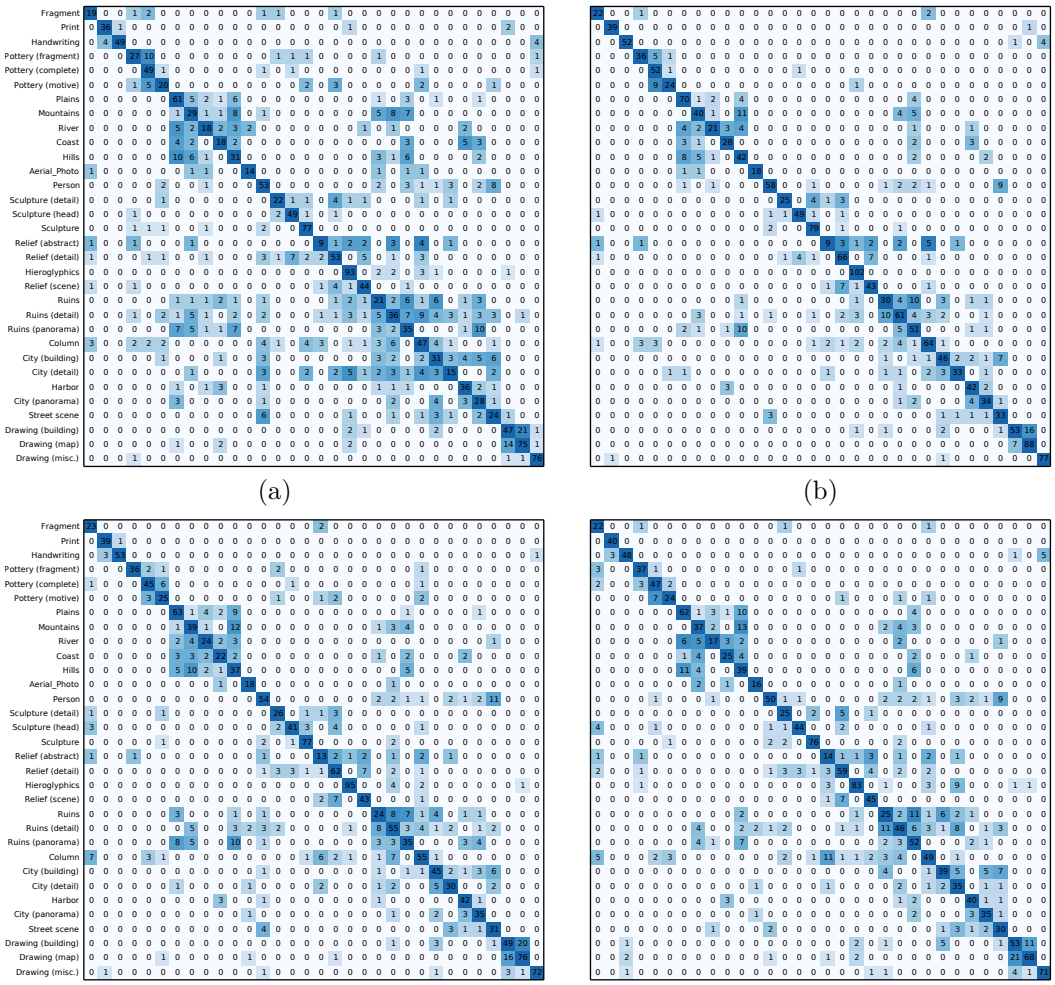


Figure 6.23: The confusion matrices produced by (a) training the network without fine tuning, (b) fine tuning the complete network using the *hybridCNN*, (c) the *k-Means* per label classifier and (d) the *Naive Bayes* classifier, where the last two were trained on the *InnerProduct* layer *fc6* using the *hybridCNN* model.

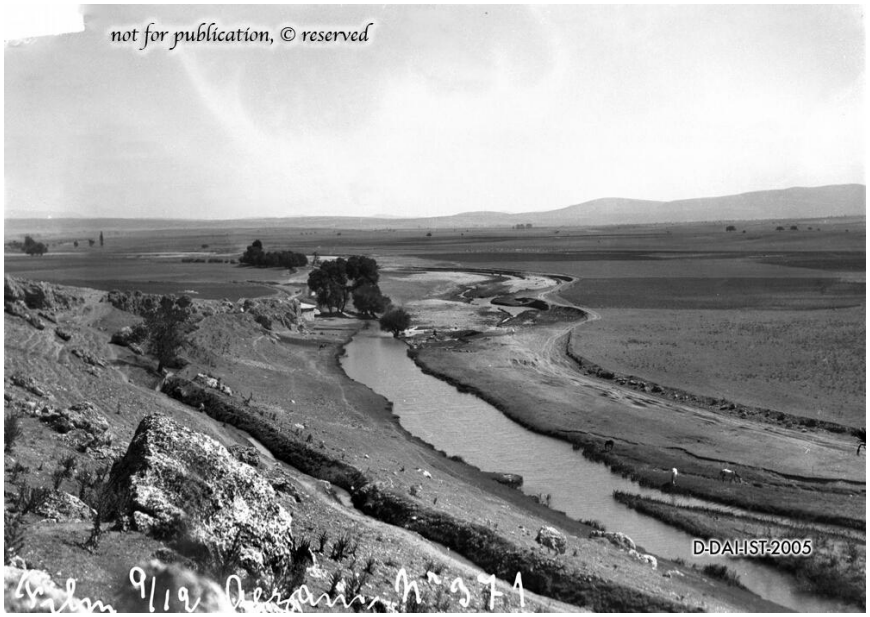


Figure 6.24: Image classified as 'Plains' (64%), 'River (32%)' or 'Ruins(Panorama) (1%)' by the fine tuned neural network. This is a good example for the potential of further extending this analysis by using multilabel- instead of multiclass classification techniques.

Figure 6.25: Image classified as 'Mountains' (68%), 'Ruins (22%)' or 'Ruins(Detail) (6%)' by the fine tuned neural network. A category 'Vegetation' simply is missing in the training- and test dataset.

Chapter 7

Summary

This thesis aimed to evaluate the potential usage of neural networks to do classification of archaeological image data. On the one hand neural networks were trained, while on the other hand classifiers were built using existing network models. Both approaches were finally compared, in order to assess if building classifiers from existing models is a valid alternative for researchers that lack the necessary hardware infrastructure for training their own networks.

At first the hope for the analysis was to find a way of automatically generating training and test datasets from *Arachne*. Even though the *Arachne* database has a lot of information about its objects, the meta information present about the motives of single images was rather sparse and not as structured as the information about the depicted objects themselves. This made an additional sorting by hand necessary and cut into the overall time available for the actual classification tests. The final data set used for the analysis hence ended up smaller than initially expected.

Two frameworks have been considered. The *Deeplearning4j* framework seemed promising and had a friendly and quickly responding developer community, but in the end its reliance on commercial cloud computing services at the time made it impractical for a first project concerned with machine learning and artificial neural networks. The *Caffe* framework proved to be a very powerful tool for the task of classifying images. Especially its fine tuning feature was advantageous, given the smaller image dataset.

It could be shown that there are multiple ways that artificial neural networks can be utilized to classify archaeological images.

For smaller image datasets, using fine tuning when training networks with one's own data seems to be advisable - instead of training new, randomly initialized model weights. In cases where the hardware infrastructure for efficiently training the neural networks themselves is missing, using existing models and their lower layer activations proved to also be viable way of building classifiers. Of the four alternative classifiers evaluated in this

regard, the *Naive Bayes* and the *k-Means* per label classifiers seem to be the most promising in both results and runtime. Both these classifiers work better on lower layers. The experiments seem to suggest, that using layer activations close to the networks convolutional layers is advisable.

The experiments also suggest that classifiers trained on network activations also perform better if the activations are created using models whose original training images are closer to one's own domain and dataset.

The performance difference between the activation based classifiers and the fine tuned/trained networks was smaller than initially expected. This may be due to the fact that all classifiers were constructed to do multiclass classification, meaning that each image only had one assigned category. It turned out that the archaeological image data used for the evaluation had a lot of images that could have fit in several of the existing categories. This posed difficulties for both the neural network- and the activation based training approaches. Based on this insight, it seems to be advisable to either reconsider the categories used in this analysis or, maybe preferable, to expand the classifiers to do multilabel classification in the future.

7.1 Outlook

The data set created for this analysis was rather small and featured only 32 categories. For any future work the most obvious task would be to expand both the image count and their categories. As already mentioned, creating a multilabel instead of multiclass dataset also seems to be desirable. Only a few of potentially hundreds of possible categories were evaluated in this analysis. This also means that the classifiers trained are still far from being usable in general applications.

Another motivation for creating such an extended image dataset is the observation that activation based classifiers produce better results when using models trained on similar images. An extended image dataset could be used to train a domain specific model for archaeology. So instead of using the *hybridCNN* as in this analysis, future researchers could use an domain specific model, if they lack the infrastructure to train their own.

Given classifiers with a sufficient amount of different categories, there are quite a range of potential applications. One example could be the automatic tagging of new images when importing them into a database like *Arachne*, another would be the possibility to search a databases by example image.

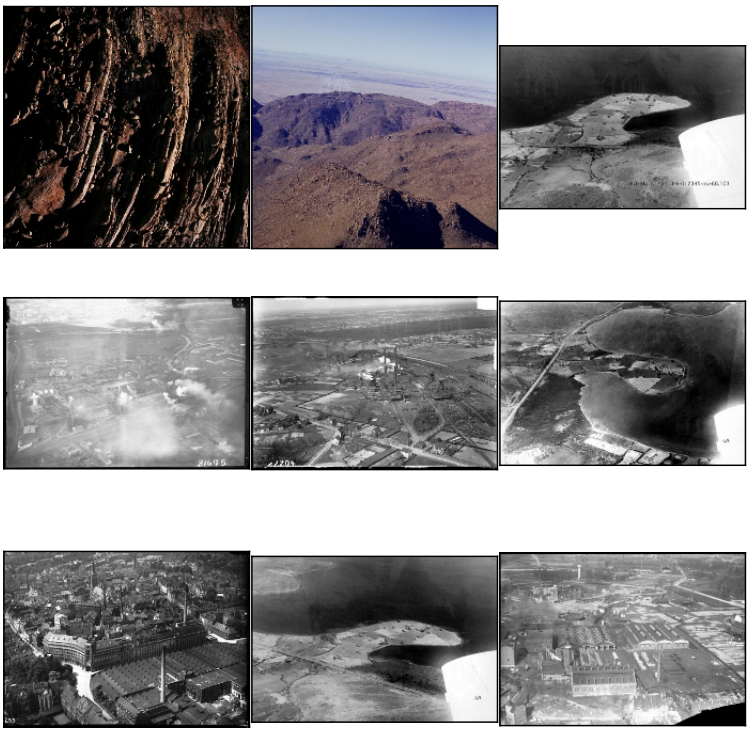
To conclude, there is good reason for considering the application of artificial neural networks in the domain of archaeology. Even without additional infrastructure, it could be shown that building image classifiers is possible. The image data used in this thesis highlighted some difficulties, especially blurry image categories. But overall there is quite a lot of room for im-

provements and ideas for continued research, most of which could not be implemented in this first analysis simply due to time constraints.

Appendix A

Overview: Handsorted data set

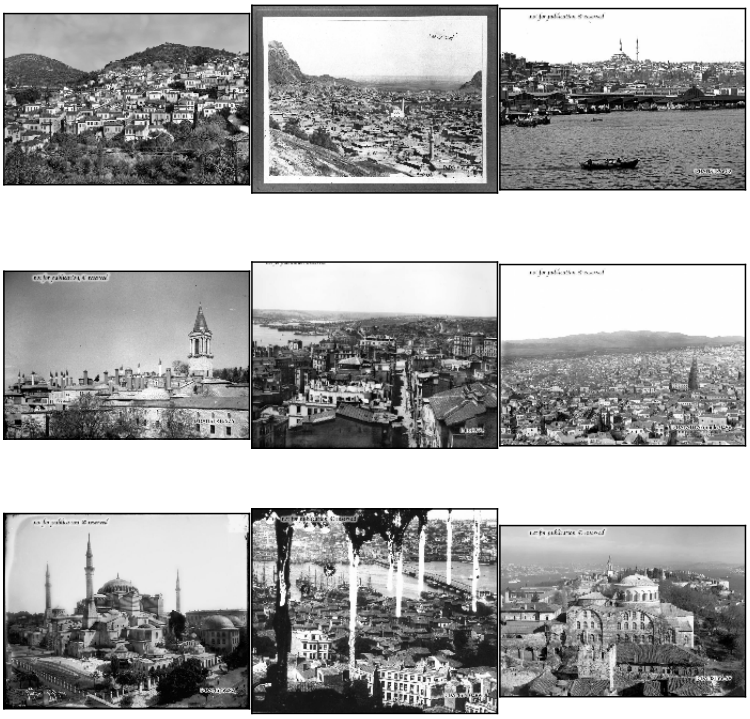
A.1 Aerial photo



A.2 City (building)

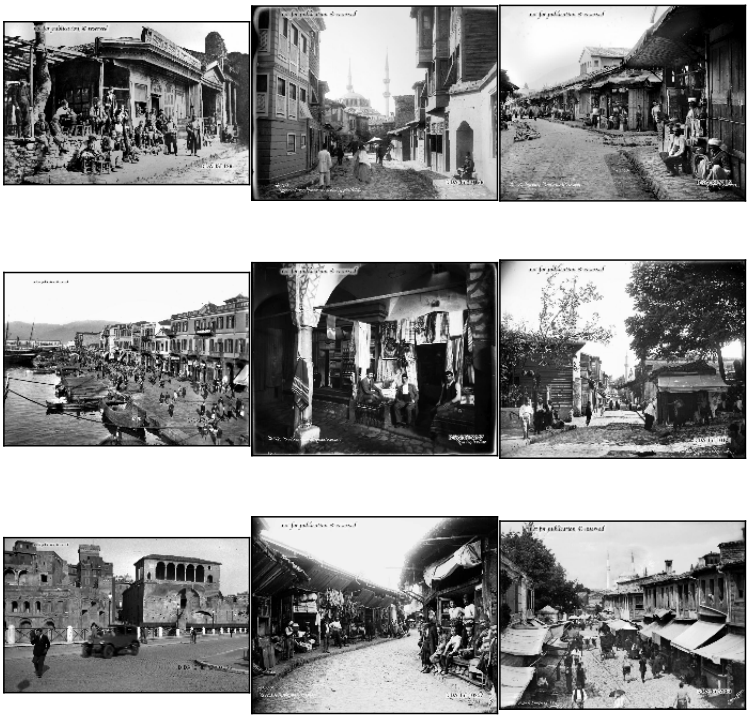
A.3 City (detail)

A.4 City (panorama)

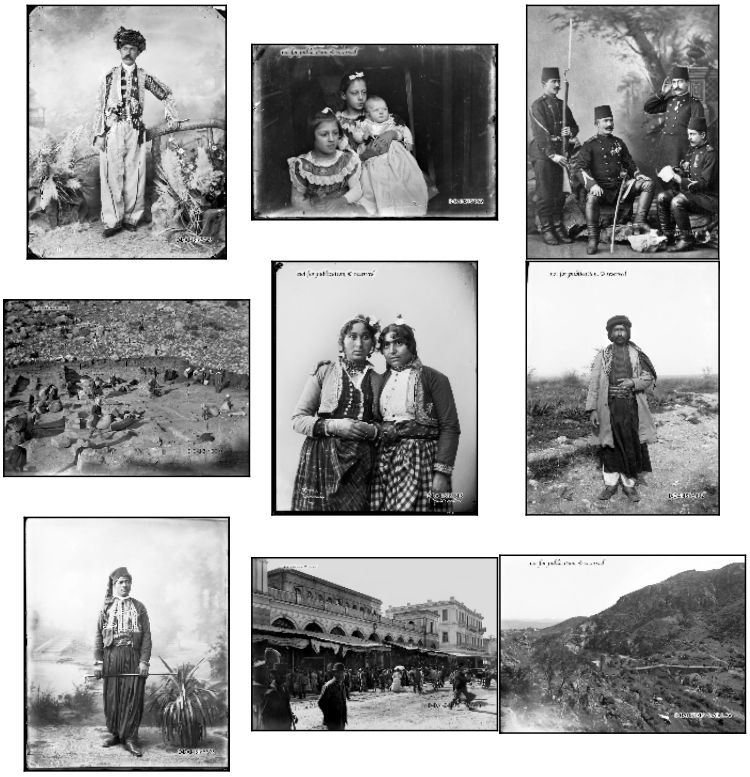


A.5 Harbour

A.6 Street scene



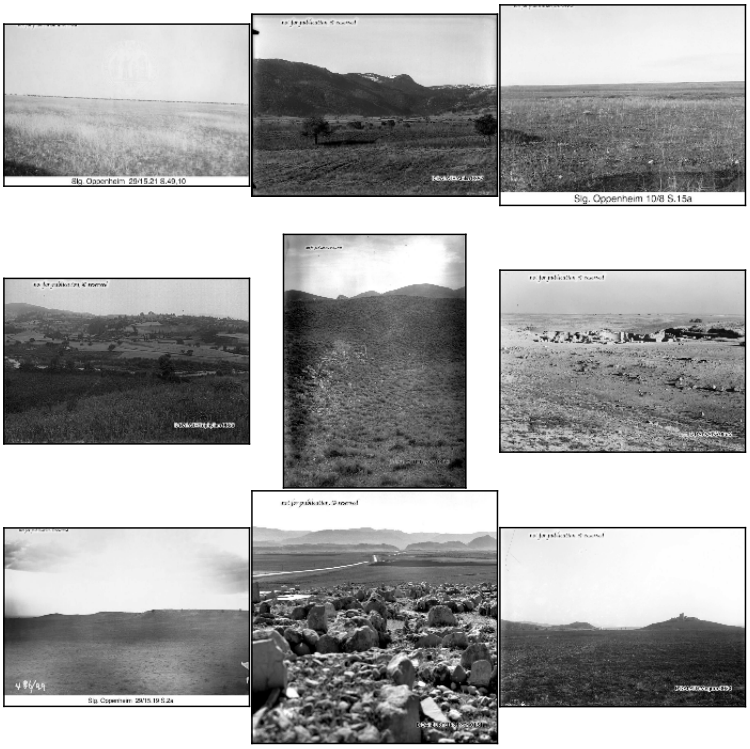
A.7 Person



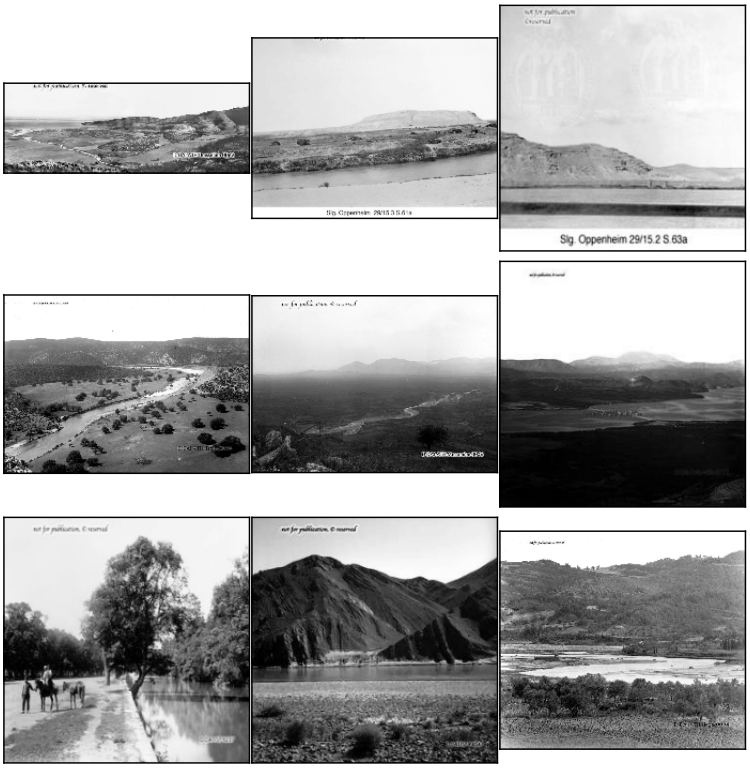
A.8 Hills

A.9 Mountains

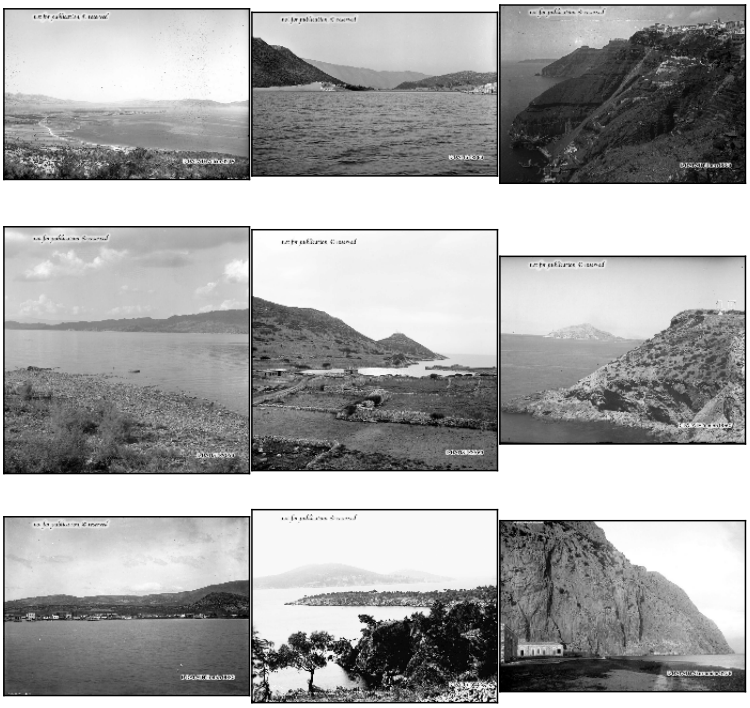
A.10 Plains



A.11 River

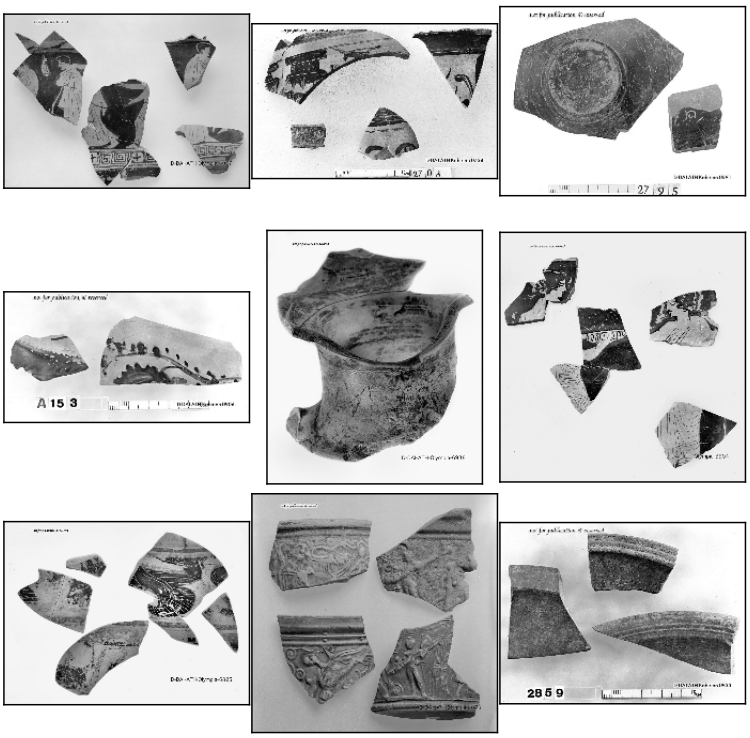


A.12 Coast

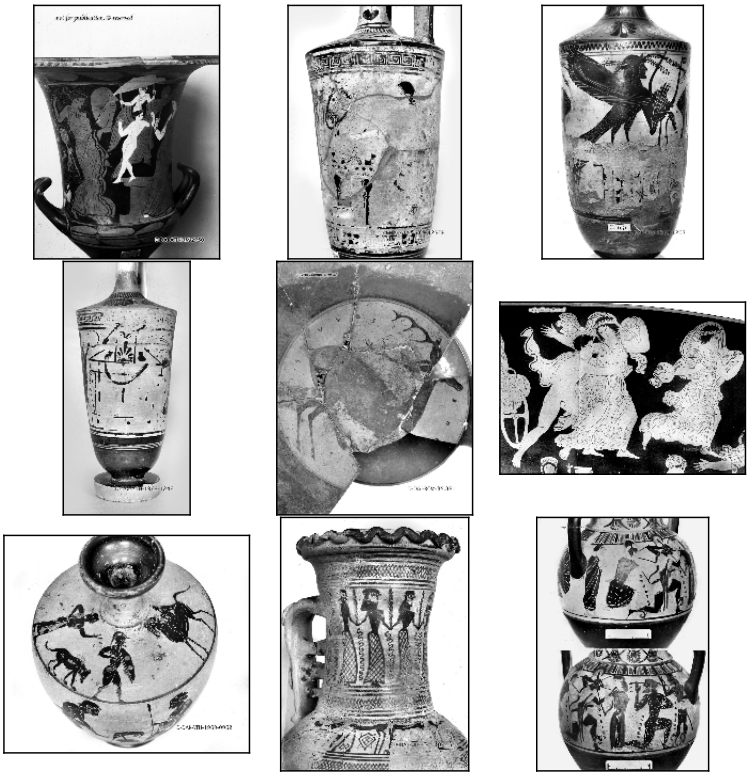


A.13 Pottery

A.14 Pottery (fragment)

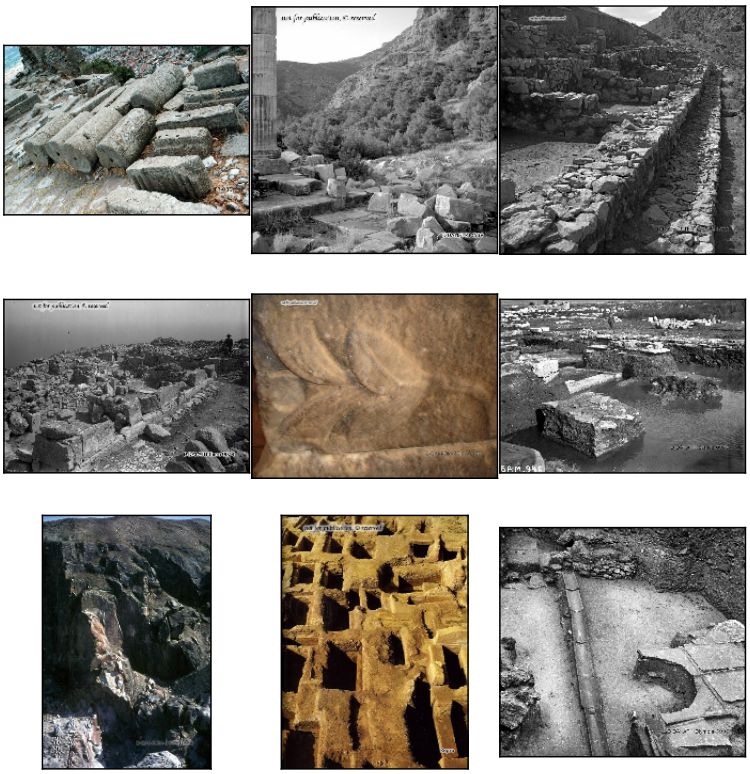


A.15 Pottery (motif)



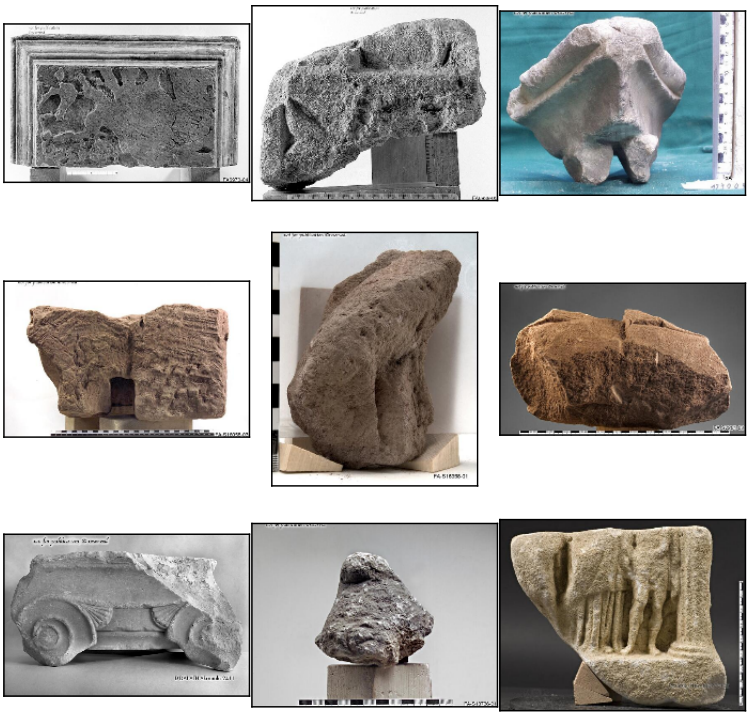
A.16 Ruins

A.17 Ruins (detail)

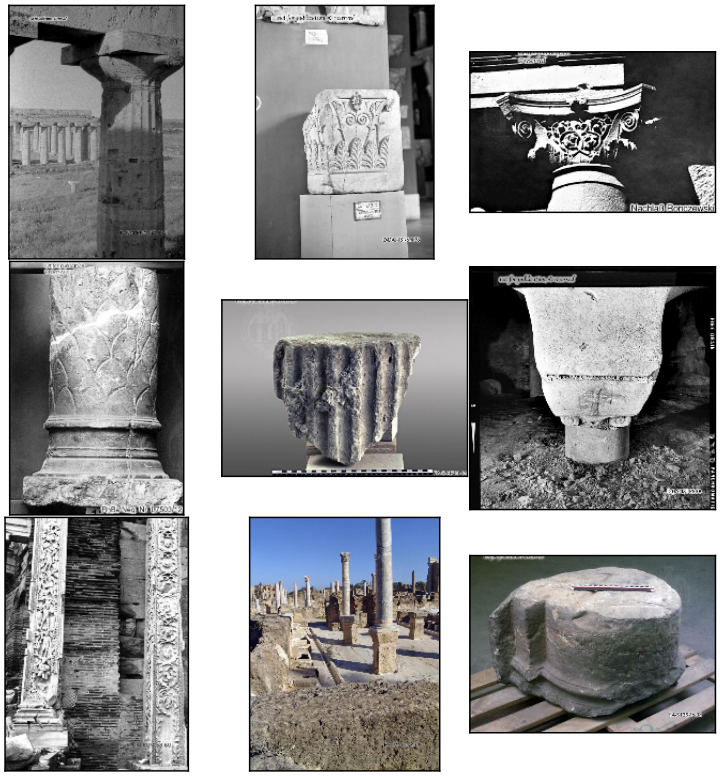


A.18 Ruins (panorama)

A.19 Fragment

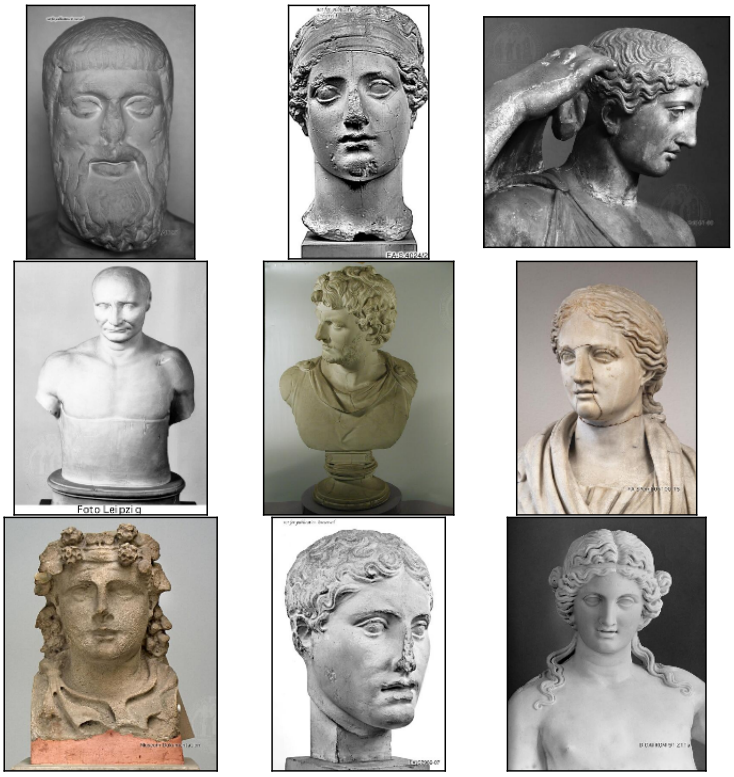


A.20 Column



A.21 Sculpture

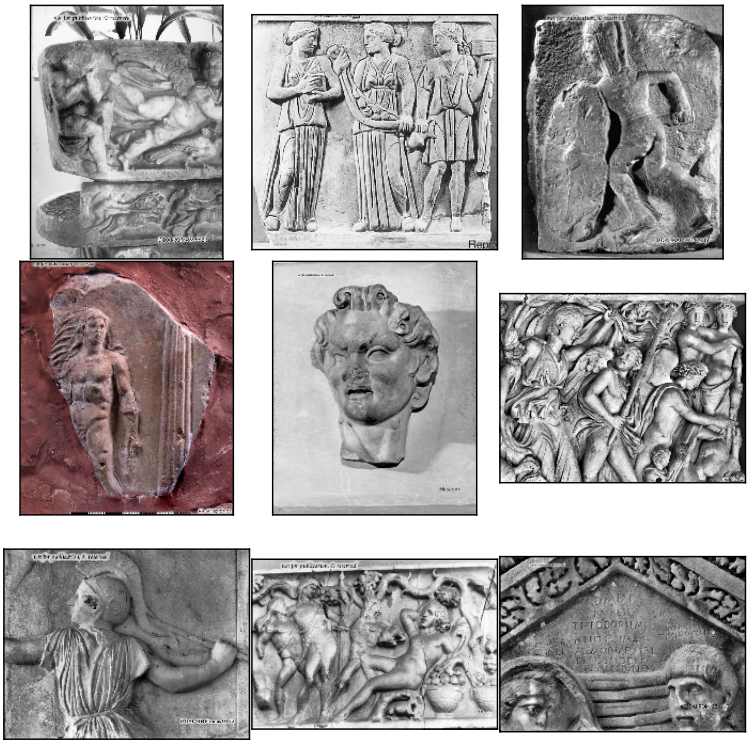
A.22 Sculpture (head)



A.23 Sculpture (detail)

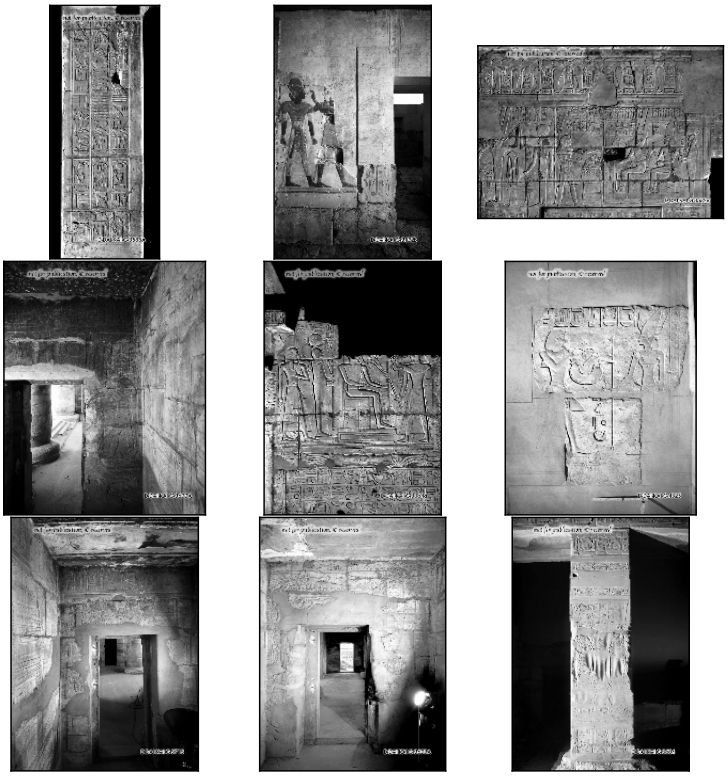
A.24 Relief (scene)

A.25 Relief (detail)

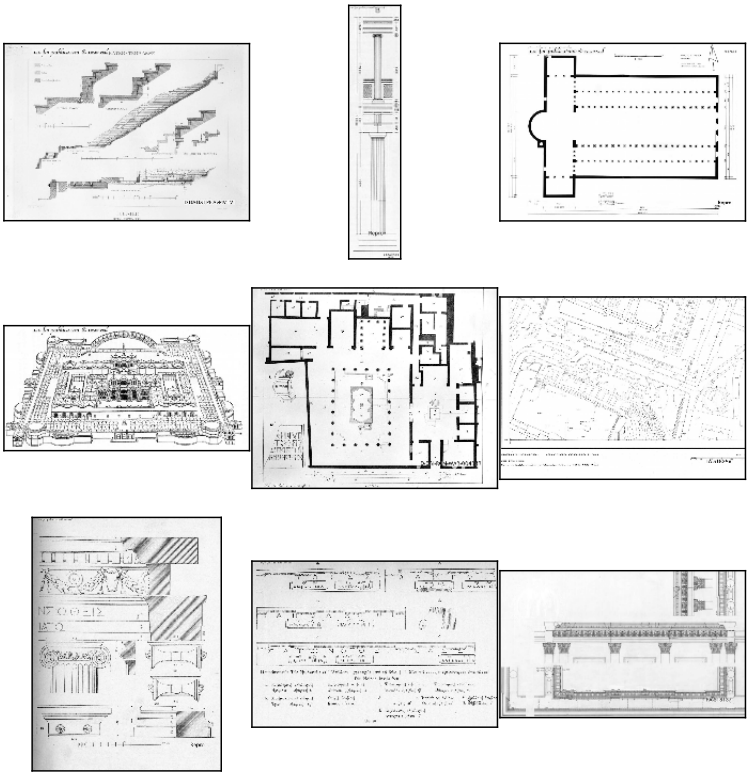


A.26 Relief (abstract)

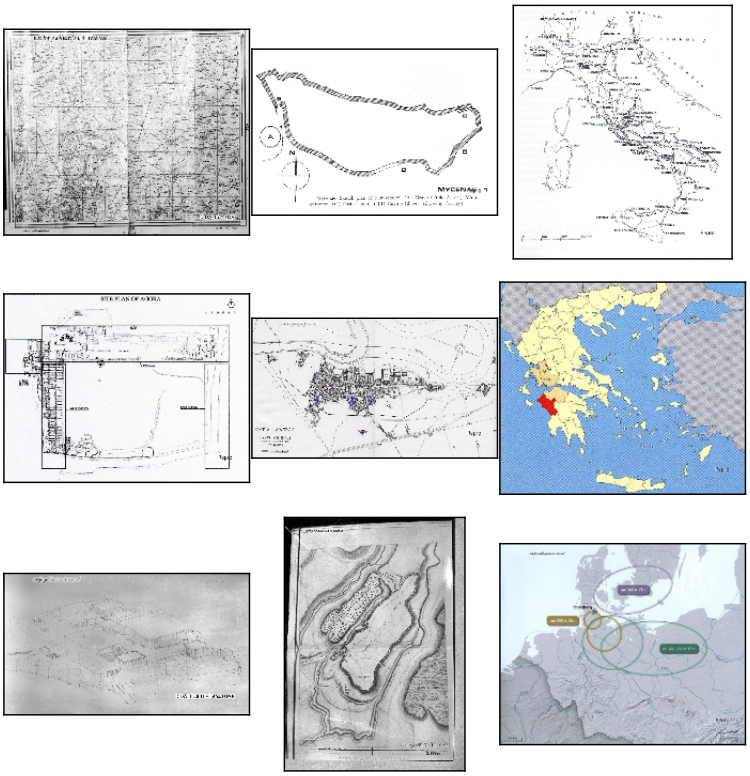
A.27 Hieroglyphics



A.28 Drawing (building)

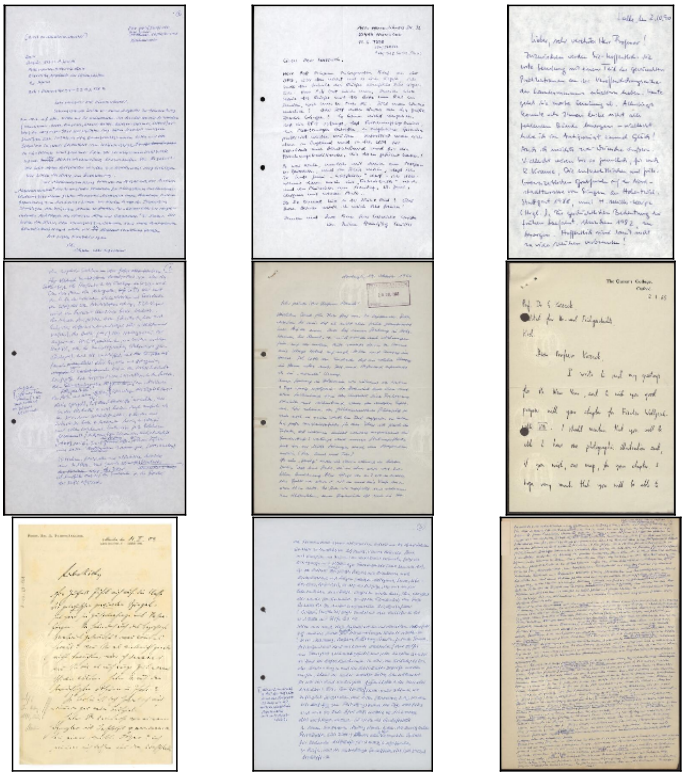


A.29 Drawing (map)

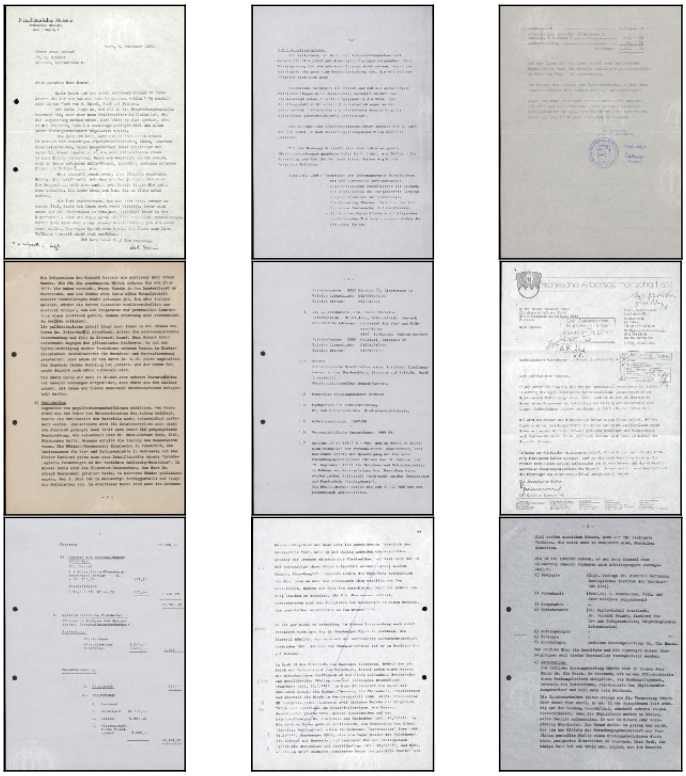


A.30 Drawing (misc.)

A.31 Handwriting



A.32 Print



Appendix B

Source code

B.1 Example: The AlexNet as implemented in Caffe in prototxt notation.

```
name: "CaffeNet"
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
```

```
    phase: TEST
  }
  transform_param {
    mirror: false
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_val_lmdb"
    batch_size: 50
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
```

```
71    top: "conv1"
72  }
73  layer {
74    name: "pool1"
75    type: "Pooling"
76    bottom: "conv1"
77    top: "pool1"
78    pooling_param {
79      pool: MAX
80      kernel_size: 3
81      stride: 2
82    }
83  }
84  layer {
85    name: "norm1"
86    type: "LRN"
87    bottom: "pool1"
88    top: "norm1"
89    lrn_param {
90      local_size: 5
91      alpha: 0.0001
92      beta: 0.75
93    }
94  }
95  layer {
96    name: "conv2"
97    type: "Convolution"
98    bottom: "norm1"
99    top: "conv2"
100   param {
101     lr_mult: 1
102     decay_mult: 1
103   }
104   param {
105     lr_mult: 2
106     decay_mult: 0
107   }
108   convolution_param {
109     num_output: 256
110     pad: 2
111     kernel_size: 5
112     group: 2
113     weight_filler {
114       type: "gaussian"
```

```
115         std: 0.01
116       }
117       bias_filler {
118         type: "constant"
119         value: 1
120       }
121     }
122 }
123 layer {
124   name: "relu2"
125   type: "ReLU"
126   bottom: "conv2"
127   top: "conv2"
128 }
129 layer {
130   name: "pool2"
131   type: "Pooling"
132   bottom: "conv2"
133   top: "pool2"
134   pooling_param {
135     pool: MAX
136     kernel_size: 3
137     stride: 2
138   }
139 }
140 layer {
141   name: "norm2"
142   type: "LRN"
143   bottom: "pool2"
144   top: "norm2"
145   lrn_param {
146     local_size: 5
147     alpha: 0.0001
148     beta: 0.75
149   }
150 }
151 layer {
152   name: "conv3"
153   type: "Convolution"
154   bottom: "norm2"
155   top: "conv3"
156   param {
157     lr_mult: 1
158     decay_mult: 1
```

```
159   }
160   param {
161     lr_mult: 2
162     decay_mult: 0
163   }
164   convolution_param {
165     num_output: 384
166     pad: 1
167     kernel_size: 3
168     weight_filler {
169       type: "gaussian"
170       std: 0.01
171     }
172     bias_filler {
173       type: "constant"
174       value: 0
175     }
176   }
177 }
178 layer {
179   name: "relu3"
180   type: "ReLU"
181   bottom: "conv3"
182   top: "conv3"
183 }
184 layer {
185   name: "conv4"
186   type: "Convolution"
187   bottom: "conv3"
188   top: "conv4"
189   param {
190     lr_mult: 1
191     decay_mult: 1
192   }
193   param {
194     lr_mult: 2
195     decay_mult: 0
196   }
197   convolution_param {
198     num_output: 384
199     pad: 1
200     kernel_size: 3
201     group: 2
202     weight_filler {
```

```
203         type: "gaussian"
204         std: 0.01
205       }
206       bias_filler {
207         type: "constant"
208         value: 1
209       }
210     }
211 }
212 layer {
213   name: "relu4"
214   type: "ReLU"
215   bottom: "conv4"
216   top: "conv4"
217 }
218 layer {
219   name: "conv5"
220   type: "Convolution"
221   bottom: "conv4"
222   top: "conv5"
223   param {
224     lr_mult: 1
225     decay_mult: 1
226   }
227   param {
228     lr_mult: 2
229     decay_mult: 0
230   }
231   convolution_param {
232     num_output: 256
233     pad: 1
234     kernel_size: 3
235     group: 2
236     weight_filler {
237       type: "gaussian"
238       std: 0.01
239     }
240     bias_filler {
241       type: "constant"
242       value: 1
243     }
244   }
245 }
246 layer {
```

```
247   name: "relu5"
248   type: "ReLU"
249   bottom: "conv5"
250   top: "conv5"
251 }
252 layer {
253   name: "pool5"
254   type: "Pooling"
255   bottom: "conv5"
256   top: "pool5"
257   pooling_param {
258     pool: MAX
259     kernel_size: 3
260     stride: 2
261   }
262 }
263 layer {
264   name: "fc6"
265   type: "InnerProduct"
266   bottom: "pool5"
267   top: "fc6"
268   param {
269     lr_mult: 1
270     decay_mult: 1
271   }
272   param {
273     lr_mult: 2
274     decay_mult: 0
275   }
276   inner_product_param {
277     num_output: 4096
278     weight_filler {
279       type: "gaussian"
280       std: 0.005
281     }
282     bias_filler {
283       type: "constant"
284       value: 1
285     }
286   }
287 }
288 layer {
289   name: "relu6"
290   type: "ReLU"
```

```
291   bottom: "fc6"
292   top: "fc6"
293 }
294 layer {
295   name: "drop6"
296   type: "Dropout"
297   bottom: "fc6"
298   top: "fc6"
299   dropout_param {
300     dropout_ratio: 0.5
301   }
302 }
303 layer {
304   name: "fc7"
305   type: "InnerProduct"
306   bottom: "fc6"
307   top: "fc7"
308   param {
309     lr_mult: 1
310     decay_mult: 1
311   }
312   param {
313     lr_mult: 2
314     decay_mult: 0
315   }
316   inner_product_param {
317     num_output: 4096
318     weight_filler {
319       type: "gaussian"
320       std: 0.005
321     }
322     bias_filler {
323       type: "constant"
324       value: 1
325     }
326   }
327 }
328 layer {
329   name: "relu7"
330   type: "ReLU"
331   bottom: "fc7"
332   top: "fc7"
333 }
334 layer {
```

```
335   name: "drop7"
336   type: "Dropout"
337   bottom: "fc7"
338   top: "fc7"
339   dropout_param {
340     dropout_ratio: 0.5
341   }
342 }
343 layer {
344   name: "fc8"
345   type: "InnerProduct"
346   bottom: "fc7"
347   top: "fc8"
348   param {
349     lr_mult: 1
350     decay_mult: 1
351   }
352   param {
353     lr_mult: 2
354     decay_mult: 0
355   }
356   inner_product_param {
357     num_output: 1000
358     weight_filler {
359       type: "gaussian"
360       std: 0.01
361     }
362     bias_filler {
363       type: "constant"
364       value: 0
365     }
366   }
367 }
368 layer {
369   name: "accuracy"
370   type: "Accuracy"
371   bottom: "fc8"
372   bottom: "label"
373   top: "accuracy"
374   include {
375     phase: TEST
376   }
377 }
378 layer {
```

```
379     name: "loss"
380     type: "SoftmaxWithLoss"
381     bottom: "fc8"
382     bottom: "label"
383     top: "loss"
384 }
```

References

Literature

[1] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering. Algorithms and Applications*. New York: CRC Press, 2014 (cit. on p. 3).

[2] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". *Biological Cybernetics* 36 (April 1980), pp. 193–202. URL: http://link.springer.com/article/10.1007/BF00344251 (cit. on p. 11).

[3] Yoshua Bengio Ian Goodfellow and Aaron Courville. "Deep Learning". Book in preparation for MIT Press. 2016. URL: http://goodfeli.github.io/dlbook/ (cit. on pp. 8, 9, 11).

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cit. on pp. 24, 30).

[5] Yann LeCun, Koray Kavukcuoglu, Clément Farabet, et al. "Convolutional networks and applications in vision." In: *ISCAS*. 2010, pp. 253–256. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-iscas-10.pdf (cit. on pp. 22–24).

[6] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. New York: Cambridge University Press, 2008 (cit. on pp. 5–7, 32, 33).

[7] Stephen Marsland. *Machine Learning. An Algorithmic Perspective*. 2nd ed. New York: CRC Press, 2015 (cit. on pp. 3, 4, 18, 22).

[8] Tom M. Mitchell. *Machine Learning*. Singapore: McGraw-Hill Book Co, 1997 (cit. on pp. 3–7, 11, 13–16, 18–22).

[9] Kevin P. Murphy. *Machine Learning. A Probabilistic Perspective*. Cambridge: MIT Press, 2012 (cit. on pp. 3, 4, 16–18).

[10] Rául Rojas. *Theorie der neuronalen Netze. eine systhematische Einführung*. 4th ed. Berlin: Springer, 1996 (cit. on pp. 3, 10–14, 19–21).

[11] Jürgen Schmidhuber. *Deep Learning in Neural Networks: An Overview*. Tech. rep. IDSIA-03-14 / arXiv:1404.7828 v4 [cs.NE]. Galleria 2, 6928 Manno-Lugano, Switzerland: The Swiss AI Lab IDSIA, University of Lugano & SUPSI, Oct. 2014. URL: http://arxiv.org/pdf/1404.7828v4.pdf (cit. on pp. 2, 10, 11).

[12] Andrew Turpin and Falk Scholer. "User Performance versus Precision Measures for Simple Search Tasks". In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2006, pp. 11–18. URL: https://researchbank.rmit.edu.au/view/rmit:2446/n2006001961.pdf (cit. on p. 33).

[13] Bolei Zhou et al. "Learning Deep Features for Scene Recognition using Places Database." In: *Advances in Neural Information Processing Systems 27 (NIPS) spotlight*. 2014. URL: http://places.csail.mit.edu/places_NIPS14.pdf (cit. on p. 30).

Online sources

[14] *Amazon Web Services*. URL: https://aws.amazon.com (cit. on p. 25).

[15] *Arachne*. URL: http://arachne.dainst.org (cit. on pp. vii, 1, 28).

[16] *Berkely Vision and Learning Center*. URL: http://bvlc.eecs.berkeley.edu/ (cit. on p. 26).

[17] *Caffe*. URL: http://caffe.berkeleyvision.org (cit. on p. 26).

[18] *Caffe: Layer catalogue*. URL: http://caffe.berkeleyvision.org/tutorial/layers.html (cit. on p. 26).

[19] *Caffe: Model Zoo*. URL: http://caffe.berkeleyvision.org/model_zoo.html (cit. on p. 26).

[20] *Caffe: Solver*. URL: http://caffe.berkeleyvision.org/tutorial/solver.html (cit. on p. 26).

[21] *Canova: A Vectorization Lib for ML*. URL: http://deeplearning4j.org/canova.html (cit. on p. 25).

[22] *Comparison of deep learning software*. URL: https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software (cit. on p. 25).

[23] *Cuda*. URL: http://www.nvidia.com/object/cuda_home_new.html (cit. on p. 25).

[24] *Dell PowerEdge product page*. URL: http://www.dell.com/us/business/p/poweredge-r720/pd (cit. on p. 25).

[25] Dhp1080. *Biological neuron*. Originally Neuron.jpg taken from the US Federal (public domain) (Nerve Tissue, retrieved March 2007), redrawn by User:Dhp1080 in Illustrator. Source: "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program. 2006. URL: https://commons.wikimedia.org/wiki/File:Neuron.svg (cit. on p. 10).

[26] *Elastic Homepage*. URL: https://www.elastic.co (cit. on pp. 29, 35).

[27] *Google Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (cit. on p. 26).

[28] *Image category overview for ILSVRC2012*. URL: http://image-net.org/challenges/LSVRC/2012/browse-synsets (cit. on p. 30).

[29] *Image category overview for Places Database*. URL: http://places.csail.mit.edu/browser.html (cit. on p. 30).

[30] *jCUDA*. URL: http://jcuda.org (cit. on p. 25).

[31] *Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)*. URL: http://image-net.org/challenges/LSVRC/2012/ (cit. on p. 30).

[32] *NVIDIA Tesla product page*. URL: http://www.nvidia.com/object/tesla-workstations.html (cit. on p. 25).

[33] *Sobel operator on Wikipedia*. URL: https://en.wikipedia.org/wiki/Sobel_operator (cit. on p. 22).

[34] *Symas Lightning Memory-Mapped Database*. URL: http://symas.com/mdb (cit. on p. 48).

[35] Deeplearning4j Development Team. *Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0*. URL: http://deeplearning4j.org (cit. on p. 25).

[36] ND4J Development Team. *ND4J: N-dimensional arrays and scientific computing for the JVM, Apache Software Foundation License 2.0*. URL: http://nd4j.org (cit. on p. 25).

[37] *The Parthenon in Arachne*. URL: http://arachne.dainst.org/entity/5584 (cit. on p. 38).

[38] *Tutorial at deeplearning.net for convolutional neural networks*. URL: http://deeplearning.net/tutorial/lenet.html#shared-weights (cit. on p. 23).