Universiteit Leiden
Computer Science

On-line and Incremental Learning

with

Convolutional Neural Networks

Name:	M.G.S. Post
Date:	14/02/2018
1st supervisor:	Dr. M.S.K. Lew
2nd supervisor:	Dr. E.M. Bakker

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Whereas many approaches, that use convolutional neural networks, are designed assuming all training data is available at training time, in many real-life scenarios this is not the case. Examples of this are web search or facial recognition. They cannot use a fixed model because the number of categories (or objects) keeps growing or changing. This type of learning is called on-line learning. The data becomes available over time and the system learns gradually. The more restricted version of this is called incremental learning. These types of learning have their challenges. For example, it has to be able to create a reliable model at each time step and it does not know in advance what the data will look like in the future or how it changes during the training.

In the literature, there is much work available for this problem in neural networks, but not much in convolutional neural networks. The literature we found in our research shows, that the approaches that use convolutional neural networks are using fine-tuning, are boosting based or are a combination of both.

In this work, we propose three main approaches: a fine-tuning approach, combining convolutional neural networks and a boosting based method. We compared several setups of these approaches in our experiments. Our results show that all the approaches have trouble retaining old information when no images of these classes are available at the current time. From these results, we conclude that none of our proposed approaches work well in an incremental environment. The accuracies improve a lot when only a subset of the images of the class remain available. The best results have the fine-tune and boosting-based methods. However, these approaches are not ready to work in a real-life environment yet. More research is needed for that. This work is a good starting point for more research.

Contents

Chapter 1

Introduction

Many approaches using convolutional neural networks assume that all data is available at training time. In real-life, that is not always the case. Data becomes available over time in those cases. This type of learning is called on-line learning.

Real-life examples of this type of learning are among others web search, facial recognition, and medical classifications. In web search, one could think of a search engine. Websites change all the time. A fixed model would not be able to handle this. It is outdated once websites change. After some time results of the model do not reflect the current environment anymore. In facial recognition, an image library can change all the time. In a library are new images added and others are removed. This means that any time new faces can appear and others disappear. More images of already learned faces can be added and a face can also change over time. A fixed model will not be able to handle this without fully retraining it. In medical classification, an example is images of a disease and using that to automatically recognizing this in images from patients. This system could use the opportunity of getting constantly new data in order to improve its accuracy.

1.1 Defining Incremental Learning

Traditionally, in machine learning, models are trained when all training data is available. This means, that the full data set is available when training is started. The majority of algorithms, that use convolutional neural networks, work with this principle. However, in many real-life applications, not all data is available at the start of the training. The data might become available in small batches or one-by-one over time. This type of learning has many names: incremental learning, on-line learning, life-long learning and more. There are no clear overall definitions of these terms. The definitions vary among papers. In this section, we give a compilation of definitions used in the literature. Later in Section 2.1, we give an overview of the definitions we use in this work.

First of all, the difference between on-line and incremental learning is explained in work by A. Gepperth and B. Hammer [1]. They define *on-line learning* (sometimes written as *online* without a hyphen) were examples or data arrives over time as described above and the full data set is not available at the start of training. The system does not know the total number of examples in advance. *Incremental learning* is in [1] defined as a more restrictive form of on-line learning. Incremental learning methods have limited memory resources. Algorithms that store all arriving data cannot be used in an incremental environment. This means an incremental approach has no access to previously trained images.

In a paper by C. Giraud-Carrier [2] are formal definitions of incremental learning and incremental learning algorithms given. These definitions can be found in Definition 1.1 and Definition 1.2. Compared to the definition of on-line and incremental learning in [1], Definition 1.1 is more describing on-line learning in general instead of incremental learning as defined [1].

Definition 1.1. A learning task is incremental if the training examples used to solve it become available over time, usually one at a time.

Definition 1.2. A learning algorithm is incremental if, for any given training sample $x_1, \ldots, x_n$, it produces a sequence of hypotheses $h_0, h_1, \ldots, h_n$, such that h_{i+1} depends only on h_i and the current sample x_i.

There are several other terms used as well. Examples of these are life-long learning or data stream learning. *Life-long learning* [3], as its name already expresses is learning the whole lifespan of a system. This is sometimes also called *continuous learning* [4]. These two terms are similar to the term on-line learning as defined in [1]. *Data stream learning* [5] is learning from a data stream. It is seen as an on-line or incremental learning approach where the examples arrive at a fast pace. The system has a limited time to process them.

In the paper by R. Ade and P. Deshmukh [6], are five criteria mentioned that an algorithm should meet to be called an incremental algorithm. Variants on these criteria can be found in other papers as well. The criteria stated in [6] for algorithms are:

1. It will be able to learn and update with every new data - labeled or unlabeled,
2. It will preserve previously acquired knowledge,
3. It should not require access to the original data,
4. It will generate a new class or cluster when required. It will divide or merge clusters as needed, and
5. It will be dynamic in nature with the changing in environment.

1.2 Challenges in Incremental Learning

On-line and incremental learning approaches face several challenges and problems. These challenges are in the work by A. Gepperth and B. Hammer [1] described. The challenges describe among others reasons why traditional algorithms do not work in an on-line or incremental environment.

The first challenge explained in [1] is *online model parameter adaption*. This challenge is basically the task to create a reliable model, M_t, after each step, t, based on the new sample(s) and the model from the previous step, M_{t-1}. The method should work without knowing the number of samples and steps in advance. This challenge is related to Definition 1.2. It is describing the working of an on-line learning algorithm.

The second one described in [1], is called *concept drift*. This challenge means the changes in data statistics of the distribution of the structure of the data samples over time. There are two types of concept drift. The first one, *virtual* or *covariate concept drift*, is the changes in the distribution of the input samples. This means the samples of a class change. The latter one, *real concept drift*, is the change in the underlying functionality of the sample. This means for example that some images that had label A at step t_a, later at step t_b have label B. Real concept drift is a problem since it leads to conflict in classification. It has an impact on the classification performance of the model until it is adapted to the changed environment.

The survey by J. Gama et al. [7] describes concept drift in more detail. It starts with distinguishing *on-line* and *off-line* learning. It defines off-line learning as learning where the whole training data is available at the time of model training. When training is finished the model can be used for prediction. In contrast, an on-line learning algorithm processes data sequentially and the model should be able to be in operation without having training completed. Training might never finish. The model is continuously updated when new training samples arrive. The paper defines an incremental algorithm as less restrictive as an online one. They process input examples one-by-one or batch-by-batch and update their model after each step. The paper continues by defining concept drift. This definition is shown in Definition 1.3.

Definition 1.3. Concept drift can be defined, between time point t_0 and t_1, as $\exists \mathbf{x} : p_{t_0}(\mathbf{x}, y) \neq p_{t_1}(\mathbf{x}, y)$, where p_{t_0} denotes the joint distribution at time t_0 between the set of input variables $\mathbf{x}$ and the target variable y.

The paper distinguishes three three types of concept drift: *real concept drift*, and *virtual concept drift* as described earlier and *population drift*. In population drift, the population in which future samples will be drawn at test time is different compared to the population on which is trained. According to [7] the drift can happen in several ways. First of all, it can happen *suddenly/abruptly*. It changes in that situation abruptly from one to another concept. The drift can also happen *incrementally*. An incremental drift has many intermediate stages. Another way drift happens is gradual. A gradual drift switches some time between the old and new concept before finally staying with the new one. The drift can also be temporary: it changes to a new concept, but later it changes back to the old one. The challenge in concept drift is detecting it and distinguishing it from outliers or noise because that refers to once-off random deviation or anomaly.

The next challenge in the paper [1], but also mentioned in other papers such as from R. Polikar et al. [8], is known as the *stability-plasticity dilemma*. One can slowly update the model when new data arrives. This way you have slow adaption to a new environment and old information is longer retained. Methods doing this are called *stable* methods: learn slow, but conserve old knowledge longer. One problem, most stable models on this side of the spectrum have, is that they do not learn at all. The opposite of this is to quickly update the model after new data arrives. This way there is a rapid adaption to a new environment, but old information is forgotten faster as well. Methods like this are called *plastic*: fast learning, but no conserving of old data. This dilemma is a well-known constraint in artificial as well as biological learning systems. The plasticity extreme of the dilemma is also called *catastrophic forgetting*. The dilemma has been described well in a paper by M. Mermillod, A.Bugaiska and P. Bonin [9]. The problem of catastrophic forgetting in neural networks has been described in among others by R. French [10] and I. Goodfellow et al. [11]. The paper [9] gives an overview of the dilemma as well as an overview of research into it.

The fourth challenge described in [1] is called *adaptive model complexity and meta-parameters*. This means that on one side that complexity of the model must be variable because the complexity is impossible to determine in advance. It might be even necessary for the method to have a variable model complexity depending on the occurrence of concept drift. However, on the other side is the complexity of the model limited by the available resources. This requires such a model to have intelligent resource allocation when the limit is reached. In batch learning, meta-parameters such as learning rate and regularization are determined prior to learning. Since the incremental nature, this is not suited for incremental learning methods. These methods tend to use few and robust meta-parameters or use meta-heuristics to adapt their quantities during training.

The next challenge in [1] is *efficient memory models*. Incremental methods need to store information provided by observed data in a compact form due to their limited available resources. Methods to do this are via suitable system invariants, the model parameters in implicit form or via an explicit memory model. For incremental algorithms is a careful memory adaption design crucial, because it mirrors the stability-plasticity dilemma.

The last challenge described in [1] is *model bench-marking*. This challenge is essentially how to benchmark a model M. Two possibilities exist to benchmark the performance. These are *incremental versus non-incremental* and *incremental versus incremental*. The first one is used when there is no concept drift and the data distribution is stationary. The incremental method is compared to a batch algorithm. However, due to its streaming data access, the incremental method has restricted knowledge. In the latter possibility, two or more incremental algorithms are compared. This is when concept drift is faced. Such a setting can, for example, be used to compare different constant function or to compare the robustness of the models.

1.3 Image Classification

In this work, we research incremental and on-line learning on the image classification problem. In image classification, we want to classify the object of the image. This is a trivial problem in a large number of applications. Some examples of image classification are among others image search in an image database and processing and identifying objects on satellite images. In the first example application, there is a database containing images. The system searches through this database to obtain images with particular visual content. There are satellite images involved in the second example. In this application, the system needs to classify what the objects on the image or sub-image are.

Classifying images by hand is possible and the human brain is fast in detecting the subject of the image. However, it is not possible to keep up with the amount of work to be done. Also, letting humans search through a database containing millions or maybe even billions of images is too slow. Computers are fast and do not get tired. However, a computer 'sees' the image just as a sequence of bits. It needs some kind of method to classify them. Convolutional neural networks are one family of approaches that can do this.

1.4 This work

In this work, we will do research into on-line and incremental learning. Most current methods using convolutional neural networks are not designed to work in an on-line environment. Many real-life environments are on-line or even incremental environments. Also, very few research has been done for CNN in these types of learning. This means only a few approaches are available at this time.

Our main research question is: "How can we do on-line and incremental learning by using convolutional neural networks?". For this work, we can break down this question into several sub-questions. These are:

- What approaches have been proposed and used in neural networks and convolutional neural networks in the past?
- In what ways can we learn in an on-line or incremental environment with a convolutional neural network?
- Which of these approaches works best?

The aim of this work is however not to find the best on-line or incremental approach that uses CNN's. It is more an exploratory work in this field. It gives an in-depth literature research into on-line and incremental learning in neural networks and convolutional neural network. It explores and implements several approaches using CNN's. And finally, it compares these approaches in several experiments.

In the literature research, we show that there has been done a lot of research in incremental learning in neural networks. However, very little research has been done on convolutional neural networks. Current approaches using CNN's in an incremental environment can be grouped in fine-tuning methods and boosting-based methods.

The knowledge we got from the literature research is used to design three main approaches. These main approaches, we propose, are a fine-tuning approach, one that combines CNN's and a boosting-based approach. The fine-tune approach is the simplest of the three. We can do that approach in two ways: have a large initial CNN or add nodes when new classes arrive. The approach that combines CNN's creates a new CNN each time new data arrives. At test time these CNN's are combined to create a final prediction. This happens in parallel or in series. The boosting-based method uses AdaBoost and the CNN as weak-learner.

In the experiments, we compare several setups of these approaches. We also compare several procedures of fine-tuning a CNN in an incremental environment and two ways we can update our boosting classifier.

1.5 Thesis Overview

This work will continue with a preliminary chapter in Chapter 2 to give some background information on the rest of this work. The preliminaries are followed by a review of classic and recent literature in incremental learning in neural networks and convolutional neural network in Chapter 3. We continue with explaining our approaches in Chapter 4 followed by Chapter 5 with the experiments we did with these approaches. After the Experiments follow the Conclusion in Chapter 7. Finally, we end with Future Work in Chapter 8.

Chapter 2

Preliminaries

This chapter provides background information for the remainder of the thesis. It starts with some definitions and an overview of parameters. Next, it will give an introduction to neural networks, convolutional neural networks and boosting.

2.1 Definitions

In the Introduction, we introduced some terms and their definitions among literature. We will give in this section the definitions that we use. Traditional learning where all training data is from the start available is *off-line learning*. In this type of learning, we do not have to take into account whether data changes or how much data we get. On the other hand, is in *on-line learning* unknown how data will look like in the future or how much data the system will get. The data in such an environment arrives over time during the training. We see *incremental learning* as a restricted version of on-line learning. Where an on-line learning algorithm could just store all previously seen data, an incremental algorithm has limited storage and cannot do that.

2.2 Overview of Used Symbols

In this work, a number of symbols are used in equations and algorithms. The used symbols vary among literature. We tried to use as much as possible the same symbol for the same meaning in different equations and algorithms. The overview of the symbols is shown in Table 2.1. Some symbols are left out of this overview. These symbols are ones that are used just in a single equation of algorithm. Also, some symbols have a different meaning in a different context. Their exact meaning in an equation of algorithm is explained in the text near it.

2.3 Neural Network

A *neural network* is an artificial neural network inspired by its biological equivalents. It is used to estimate or approximate functions. It has similar to other machine learning methods a learning phase where the parameters of the model are trained before it can be used to predict. One of the earliest work into neural networks is an article by W. McCulloch and W. Pitts [12] in 1943. In the years and decades after this work were the perceptron, by F. Rosenblatt [13], and backpropagation, by P. Werbos [14], proposed. The explanation in the rest of this section is based on the book about neural networks and how to do deep learning with them by M. Nielsen [15].

A neural network consists of neurons. The *perceptron*, as proposed by by F. Rosenblatt [13], is the most basic neuron. The allowed input, as well as the generated output of the perceptron, is binary, so either 0 or 1. Each input x_i has a weight α_i. The output is 1, if $\sum_i \alpha_i x_i$ is greater or equal to some threshold value. If it is smaller, the output is 0. This is shown in Equation 2.1. A visualization of the perceptron is shown in Figure 2.1(a).

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j \alpha_j x_j < \text{ threshold} \\ 1 & \text{if } \sum_j \alpha_j x_j \geq \text{ threshold} \end{cases} \tag{2.1}$$

A more complex type of neuron is the *sigmoid neuron*. This type neuron allows real value inputs between 0 and 1. Each input x_i of the sigmoid has, just like the perceptron, a weight w_i. In

Symbol	Meaning
x	Input, sample, data point, image
$\mathbf{x}$	Set of inputs
X	Space of inputs ($x \in X$)
y	Label, target variable
$\mathbf{y}$	Set of labels
Y	Space of labels ($y \in Y$)
t	Iterator for time, step, iterations
T	Total time, total number of steps, total number of iterations
w	Weight of input
$\mathbf{w}$	Set of weights
α	Weight of node
i	Counter, iterator
j	Counter, iterator
b	Bias
h	Hypothesis (boosting)
H	Set of hypotheses (boosting)
o	Output
σ	Sigmoid function
n	Layer number, kernel number
N	Number of layers, number of kernels, number of images (N_x)
s	Stride, step size, data set (training set S_l, test set S_v)
p	Padding
W	Size image
M	Number of feature maps, number of Models
m	Model, feature map
p	Distribution, normalized distribution
ϵ	Error
β	Normalized error
k	Kernel
f	Filter
y'	Prediction
c	Classifier
C	Set of classifiers

Table 2.1: Overview of most of the used symbols in this work. The symbols that are left out are mainly in a single equation or algorithm. Meaning of symbols with multiple meanings depends on context

contrast to the perceptron, has the sigmoid neuron an overall bias. The output generated by the weighted inputs plus the overall bias is shown in Equation 2.2,

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \equiv \frac{1}{1+\exp(-\sum_j \alpha_j x_j - b)}, \tag{2.2}$$

where $\sigma(z) = \sigma(\alpha \cdot x + b)$. A visualization of the perceptron is shown in Figure 2.1(b).

In a neural network are these and other types of neurons used. In Figure 2.2 is a simple neural network shown. This network has three layers. The left-most layer is called the input layer. This is a very simple layer. Each neuron in this layer receives an input value and redirects it to its output without manipulating it. The middle layer is called a hidden layer. The hidden middle layer contains the sigmoid, perceptron or other types of neurons. At the end, the right-most layer is the output layer. This is the output of the neural network. The neurons in this layer either say true or false depending on the criterion they are set on. However, they can also do the final calculation and output the probability that their criterion is true based on the output of the

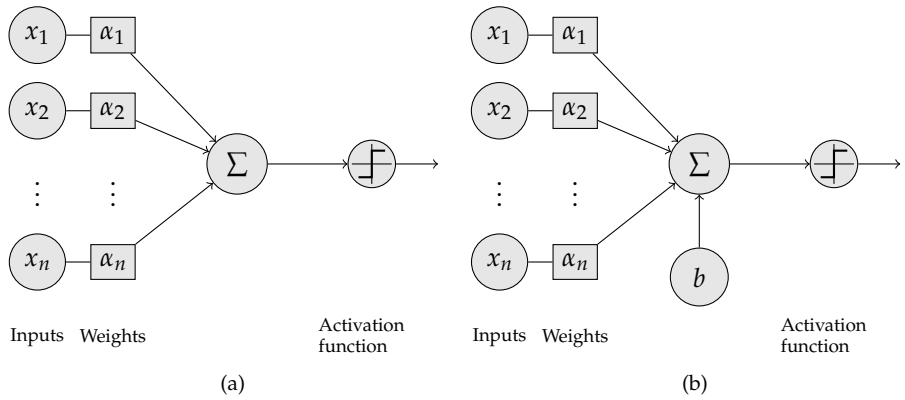

Figure 2.1: Example of the perceptron, (a), and sigmoid neuron, (b). Both neurons have inputs x_1 to x_n. Each input has a weight w_i. The sigmoid neuron also has a overall bias b.

last function. This probability is usually a value between 0 and 1. This network is this example has two input, three hidden and one output neuron.

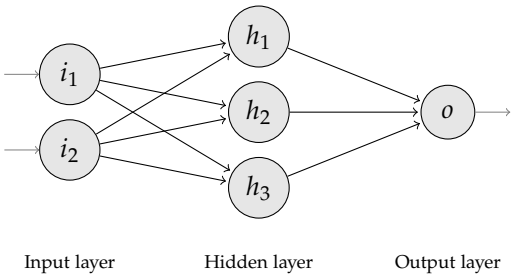

Figure 2.2: Example of a simple neural network. It has two input, three hidden and one output layer.

The neural network in Figure 2.2 is called a single-layer network. A neural network can have more than one hidden layer. One type of such networks is called multi-layer perceptrons (MLPs). Both these types of networks are feed-forward neural networks, which means that there are no loops in the network.

2.4 Convolutional Neural Network

A more complex form of neural networks is the *convolutional neural network* (CNN). CNN's are designed for image and video recognition. A CNN uses a combination of different types of layers such as convolutional and sampling/pooling layer. In this section a basic explanation of CNN's is based on the book by M. Nielsen [15] and papers by D. Ciresan et al [16], A. Giusti [17] and D. Scherer at al. [18].

2.4.1 Convolutional Layer

The *convolutional layer* of a convolutional neural network is one of the main type of layers in a CNN. This layer is in contrast to the layers in a neural network are not fully connected. It gets an image or a feature map from a previous layer as input. The layer has small groups or collections of neurons. These groups of neurons are called *kernels*. The behavior is defined among others by the kernel size, number of maps and step sizes. Each layer has M maps of size $m_x \times m_y$, where m_x is the size in the x-direction and m_y the size in the y-direction. To generate these maps a

kernel of size $k_x \times k_y$ is shifted over the input 'image'. The step sizes s_x and s_y define how the kernel is shifted in the x and y direction. The output maps of a given layer share their weights of the kernel. Based on the input map size, $m_x^{n-1} \times m_y^{n-1}$, the kernel size, $k_x^n \times k_y^n$, and step size, s_x^n and s_y^n, of this convolutional layer, can the size of the output maps determined. This is defined in Equation 2.3, where n indicates the layer number

$$m_x^n = \frac{m_x^{n-1} - k_x^n}{s_x^n + 1} + 1; \qquad m_y^n = \frac{m_y^{n-1} - k_y^n}{s_y^n + 1} + 1 \tag{2.3}$$

The convolutional layers can make use of Rectified-Linear Unit (ReLU). It is an activation function defined as: $f(x) = max(x, 0)$. It computes the output as x if $x > 0$ and 0 if $x \leq 0$. They can also use Local Response Normalization (LRN). That is used to normalize the output of the ReLU. The normalized activity $b_{x,y}^i$ is given by the expression

$$b_{x,y}^i = a_{x,y}^i / \left(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a_{x,y}^j)^2 \right)^\beta, \tag{2.4}$$

where $a_{x,y}^i$ is the activity of kernel i at position (x, y) after applying ReLU, N the number of kernels. The sum in the expression runs over n adjacent kernel maps at the same spatial position. The constants k, n, α and β are hyper-parameters.

2.4.2 Pooling Layer

Some of the convolutional layers are followed by *pooling layers*. The pooling layers reduce the resolution of feature maps generated by the convolutional layers. They aggregate a local neighborhood. There are several pooling methods available for use in convolutional neural networks. One of the most popular ones is *maximum pooling*. Some of the other pooling options are *average* (or *mean*) *pooling* and *stochastic pooling*.

The pooling is done by using the $f_x \times f_y$ filter of the corresponding pooling method. These filters are applied in combination with a certain stride. The stride is the 'step-size' of the filter. A stride of 0 would mean that the filter stays forever at the same place. Using a stride of 2 causes the filter to do steps of 2 in the y and x direction.

For example, assume we have a feature map of size 4×4, a filter of 2×2 and a stride of 2, then we would have four places where the filter is applied. This is shown in Figure 2.3. The resulting reduced feature map is reduced by a factor two and has a size of 2×2.

2	6	8	4
1	3	5	2
2	7	4	9
3	6	3	7

Figure 2.3: Input feature map for feature pooling

In Equation 2.5 the output size of a filter is shown. In this equation is W the length of the input image, k the size of the filter, p the padding and s the stride. Where padding is the number of pixels added at each side of the image. If we would apply this to the feature map of Figure 2.3, then the W is 4, k is 2, p is 0 and s is 2. When we use these number in Equation 2.5, we end up

with $(4-2+0)/2+1$ and thus a output size of 2.

$$o = \frac{W-k+2p}{s}+1 \tag{2.5}$$

One of the most popular filters is the maximum pooling filter. The maximum pooling filter chooses the maximum value of the input for each pooling region. An example is shown of this in Figure 2.4.

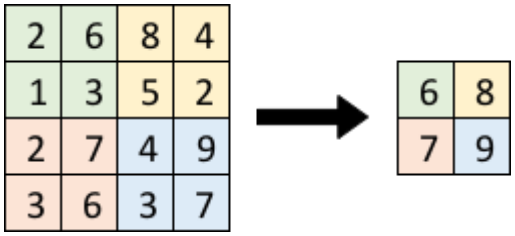

Figure 2.4: Example of maximum pooling. The maximum value of each region is chosen

Another pooling method, average pooling (also called average pooling), works in a similar way. Instead of the maximum value, the mean of the pooling region is determined. The stochastic pooling method is more complex one. It uses a distribution to randomly pick a value of each pooling region.

2.4.3 Classification Layer

The final layers of a convolutional neural network are usually fully connected layers. These layers are also called classification layers. The fully connected layers combine the feature maps from the final convolutional or pooling layer into a one-dimensional vector. The final layer of these fully connected layers outputs a vector containing the probability for each of the categories in case of the classification problem.

An often used technique in fully connected layer (but not limited to this type of layers) is the dropout. Dropout is a technique to reduce overfitting. One used strategy is the one proposed in the paper by Krizhevsky et al. [19]. At training time, it sets the output of each neuron to zero with a probability of 0.5. This reduces the amount of output data by half compared to the amount it got at the input. It also reduces the computation time of following layers. During test time no data is dropped, but the values of all neurons are multiplied by 0.5.

2.5 Boosting

Boosting is a general approach where many 'weak', inaccurate classifiers are combined to a 'strong', much more accurate classifier. One of the first boosting algorithms and one of the most widely used is Adaptive Boosting, usually shortened as AdaBoost. AdaBoost was introduced by Y. Freud and R. Schaphire in [20].

The algorithm has as input a training dataset S_k of N labeled examples $S_k = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, where x_i is an item in some domain or instance space X and each label y_i is in the label set Y. Note that the version of AdaBoost presented in [20] is only designed for binary problems, thus $Y = \{0, 1\}$. Also a weak-learner, referred to as *WeakLearn*, and a integer T, that specifies the number of iterations that the algorithms runs or number of hypotheses generated, are given to AdaBoost. At the start the weights $\mathbf{w}^1$ are initialized by $w_i^1 = 1/N$.

The first step, in each iteration t, is to generate a distribution $\mathbf{p}^t$ by normalizing the weights $\mathbf{w}^t$. Next, WeakLearn is called with dataset S_l and distribution $\mathbf{p}^t$ to generate a hypothesis h_t, h_t:

$\mathbf{x} \rightarrow \{0, 1\}$. Based on the performance of hypothesis h_t is a new weight vector $\mathbf{w}^{t+1}$ determined. For this is first the error of h_t determined,

$$\epsilon_t = \sum_{i=1}^{N} p^t |h_t(x_i) - y_i|.$$

With ϵ_t is the normalized error β_t determined,

$$\beta_t = \frac{\epsilon_t}{(1 - \epsilon_t)}.$$

Finally, the new weights vector $\mathbf{w}^{t+1}$ is calculated with each as

$$w_i^{t+1} = w_i^t * \beta_t^{(}1 - |h_t(x_i) - y_i|).$$

This procedure is repeated T times. At the end the final hypothesis h_f is the weighted sum of hypotheses h_1 to h_T. This is shown in Algorithm 1.

Algorithm 1 Regular AdaBoost

Input: Dataset $S_l = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, T specifying the number of iterations and weak algorithm WeakLearn

Output: hypothesis h_f

1: **Initialize** weight vector $\mathbf{w}^1$ with $w_i^1 = \frac{1}{N}$ for $i \in [0, \ldots, N]$
2: **for** $t = 1, \ldots, T$ **do**
3: Normalize weights $\mathbf{w}^t$, $\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$
4: Call WeakLearn with $\mathbf{p}^t$ and get back a hypothesis h_t: $X \rightarrow \{0, +1\}$
5: Calculate error of h_t, $\epsilon_t = \sum_{i=1}^{N} p_i^t |h_t(x_i) - y_i|$,
6: Determine normalized error, $\beta_t = \frac{\epsilon_t}{(1-\epsilon_t)}$
7: Calculate new weights, $w_i^{t+1} = w_i^t * \beta_t^{(}1 - |h_t(x_i) - y_i|)$
8: **end for**
9: **return** the hypothesis $h_f(x) = \begin{cases} 0 \text{ if } \sum_{t=1}^{T} (\log \frac{1}{\beta_t}) h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \\ 1 \text{ otherwise} \end{cases}$

As stated above, at the start all weights are set equally. In each iteration, AdaBoost adapts adjusts the weights to the errors returned by WeakLearn. The procedure used to generate the new weights $\mathbf{w}^{t+1}$, reduces the probability of examples that are predicted correctly by hypothesis h_t and increases the probability of examples that are incorrectly predicted by h_t. The idea here is that each iteration examples that are predicted incorrectly are more likely to be used during training in the next iteration than correct predicted examples. This adaptive mechanism gives AdaBoost its' name, Adaptive Boosting. The boosting algorithm can use an algorithm that can use the distribution $\mathbf{p}_t$ on the training examples. This means that a neural network or convolutional neural network can be used as weak-learner. One of the advantages of AdaBoost is that it does not need any prior knowledge of the performance of the used weak-learner.

The final hypothesis h_f in Algorithm 1 is decided by majority voting. Each test sample is predicted by all weak-learners. All normalized weights of the weak-learners are summed as shown in step 9 if the weak-learner returns 1. If this is larger than half of the total weights, then the final prediction is 0, otherwise, it is 1.

Chapter 3

Literature Review

This chapter is a review of classical and recent work in incremental learning in neural networks and convolutional neural networks.

3.1 Neural Network

A lot of research has been done in on-line and incremental learning in neural networks. In Section 1.2, we described the stability-plasticity dilemma and catastrophic forgetting in incremental learning. A paper by R. French [10], published in 1999, goes into more detail for that for connectionist networks, another term for neural networks. They examine the causes, consequences and a number of solutions for the problem of catastrophic forgetting in neural networks.

All approaches for on-line and incremental learning in neural networks have to deal with these challenges. In this section, we will go into more detail of some types of approaches. The types we will describe are *Adaptive Resonance Theory* (ART) based methods, boosting based methods, *radial basis function* (RBF) based methods and *self-organizing neural networks* (SOINN). At the end of the section, we also show some other methods.

3.1.1 Adaptive Resonance Theory Based Algorithms

The *Adaptive Resonance Theory* (ART) is introduced by S. Grossberg in 1976 [21]. He describes it as a cognitive and neural theory of how the brain autonomously learns to categorize, recognize, and predict objects and events in a changing world. In [22] by Grossberg ART is explained in detail and in a 2013 paper [23] he gives a review of ARTs' classical and recent developments.

Based on ART are a variety of neural network models developed. The first one, *ART 1*, was introduced by G. Carpenter and S. Grossberg in 1987 [24]. It is an unsupervised approach and only accepts binary input. *ART 2* [25] extends ART 1 by also accepting analog data. The performance improvement in terms of speed was realized with *ART 2-A* in 1991 [26].

ARTMAP or *Predictive ART* is a supervised learning system by G. Carpenter, S. Grossberg and J. Reynolds [27]. It is a type of self-organizing expert system and it is designed for on-line recognition learning, thesis testing and, adaptive naming in response to an arbitrary stream of inputs. ARTMAP consists of two ART modules, ART_a and ART_b. These modules are ART 1 modules with some additions, but they can be of any type of ART module. The ART_a and ART_b read respectively input $[\mathbf{a}^{(p)}]$, a stream of inputs, and $[\mathbf{b}^{(p)}]$, a stream labels given $[\mathbf{a}^{(p)}]$. In between the two ART modules is an inter-ART module, the Map Field. This controls the learning of an associative map from ART_a recognition categories to the ART_b categories.

In [28] by G. Carpenter et al. is ARTMAP extended by describing the ARTMAP dynamics in terms of the fuzzy set-theoretic operations. This method is called *Fuzzy ARTMAP*. Fuzzy sets were introduced by L. Zadeh in 1965 [29]. In classical set theory, an object either belongs or not belongs to a set. In a fuzzy set does the object have a grade of membership, which is a value between 0 and 1. The ART modules in ARTMAP are in the fuzzy variant replaced by Fuzzy ART systems. In ARTMAP leads fast learning typically to different adaptive weights and recognition categories for different orderings of the training set. The fuzzy variant has a voting strategy introduced to prevent that. This based on an ARTMAP system. This system is trained on input sets with different orderings. In a later paper by G. Carpenter et al. [30] is a new method based

on the fuzzy ARTMAP introduced. The method is developed for automatic mapping of Landsat Thematic Mapper (TM) and terrain data; a remote sensing classification problem.

Another Fuzzy based ART system is *Fuzzy ART* by G. Carpenter, S. Grossberg and D. Rosen [31]. This method introduces Fuzzy set theory in the ART 1 modules. The ART 1 dynamics are in a similar way replaced as in Fuzzy ARTMAP. By introducing fuzzy theory into ART, the system can learn from binary as well as analog input. An implementation of this method is given in the paper [32] by the same writers.

In a paper by J. Williamson is the *Gaussian ARTMAP* [33] introduced. It is an incremental learning system for learning supervised learning of analog multidimensional maps. The system combines a Gaussian classifier with an ART neural network. It uses the ART choice function as the discriminant function for the Gaussian classifier.

3.1.2 Boosting Based Approaches

Learn++ was introduced by R. Polikar et al in 2000/2001 [8, 34]. It is a supervised learning method for neural networks such as multilayer perceptrons (MLP). According to the writers, it learns new data, including new classes, without forgetting previously learned information. The method is inspired by Schapire's AdaBoost [20]. It has two key components. The first one is the selection of the subsequent data set. This is based on generating a number of hypotheses using different distributions of the training data. This is a weak learner. Whereas AdaBoost depends on the performance of individual hypothesis h_t, is Learn++ using the performance the overall hypotheses set H_t. The second key component is the way the individual hypotheses are combined. Learn++ uses weighted majority voting to do this.

Based on Learn++ are a variety of new methods developed, which improve or extend it. *Learn++.MT* by M. Muhlbaier, A. Topalis and R. Polikar [35] is a modified version of Learn++. In [35] the "out-voting" problem of Learn++ is described. The original method works for most applications, but in incremental learning problem where new classes are introduced introduce the "out-voting" problem. This means that in the majority voting the new class is in disadvantage compared to the original classes. A new class needs a relatively large number of classifiers so their weight can "out-vote" the classifiers from earlier batches. Learn++.MT uses a different principle to determine the weights. This principle is based on a cross-reference of the classes that have been seen by each classifier during training. During test time the algorithm can dynamically adjust the voting weights for each test sample. The algorithm has a reduced number of classifiers which also improve the performance.

Another approach to solving the "out-voting" problem is Learn++.NC by M. Muhlbaier, A. Topalis, and R. Polikar [36]. It is designed to efficiently learning of multiple *new classes* (NC). According to the writers does Learn++.NC improve performance and stability compared to its predecessor. It uses substantially fewer classifiers than its predecessor. Its' main feature is a new voting mechanism. The method treats its' individual classifiers as intelligent experts. The experts consult with each other and together determine which classifiers have more confidence in identifying a given sample. Then, the voting weights are dynamically set. There is, however, a throwback of Learn++.NC. It cannot handle concept drift problems.

A member that combines ideas from other members of the Learn++ family is *Learn++.UDNC* by G. Ditzler, M. Muhlbaier and R. Polikar [37]. Learn++.UDNC combines ideas from *Learn++.UD* [38] and Learn++.NC [36]. Learn++.UD was introduced by M. Muhlbaier, A. Topalis, and R. Polikar for learning unbalanced data. However, it was not able to learn new classes. Learn++.UD combines algorithms from Learn++.UD and Learn++.NC such as a class-specific weighing method and normalized preliminary confidence measures. The methods also include a transfer function that is used reducing the confidence bias of a group of samples that is trained on a majority class.

Learn++.NSE by R. Elwell and R. Polikar [39] is a Learn++ based method for incremental learning of concept drift, that is characterized by non-stationary environments (NSEs). The method can learn in environments that experience constant or variable rate of drift, addition or deletion of concept classes and also cyclical drift. It learns in batches without making assumptions about the nature or rate of drift. Learn++.NSE makes use of current and past classifiers and combines that with dynamically updated voting weights. These weight updates are based on their time adjusted errors in past and current environments.

Learn++ has also been used in different learning methods. In a paper by D. Medera and S. Babinec [40] is Learn++ used in a convolutional neural network. They state that the error rate achieved by their approach was comparable with non-incremental learning. However, they state as well that some issues remain for further research.

3.1.3 Self-Organizing Neural Network

Self-organizing neural networks were proposed by D. Willshaw and C. von der Malsburg [41] and the principle was later generalized by T. Kohonen [42]. This principle is also called self-organizing maps. In [41] the idea consists of two two-dimensional sheets representing the pre-synaptic and post-synaptic sheet of nerve cells. Axons represent connections between the pre- and post-synaptic sheet. These axons are a mapping between the nodes. Over sufficient training time, these connections spread out over the post-synaptic sheet and gradually narrow down to clusters. Each cluster represents a class of samples.

In the papers [42] and [43] by T. Kohonen is the idea and working of the method is explained in more detail. The method is an unsupervised learning method. The map is a one- or two-dimensional order of neurons. Each neuron has a 2D or 3D position in space. During training the position changes, but the relative position of neighboring neurons remains the same. Every neuron has a weight w_i for each input x_i, where $i \in 1, \ldots, N$. The output of the neuron is $f(\sum_{i=1}^{N} w_i x_i)$. The weights are different for each neuron. The training of the self-organizing map consists of four steps: 1. initialization, 2. sampling, 3. matching and 4. updating. The neurons are initialized with small random weights. In the second step, one of the input samples is drawn. Then the neuron that is the closest to input sample based on its weights is matched. A measure such as Euclidean distance can be used. The neuron with the smallest distance wins. To update the model, the winning neurons and its neighbors are moved towards the input sample. The steps 2 to 4 are repeated until some condition (e.g. iteration limit) is met. A cooling schedule is used in the adaption step.

The problem with the method described above is how to choose a suitable network size and shape in advance, especially in an incremental environment. A solution is described in a paper by B. Fritzke [44]. The paper describes two variants of a self-organizing network that has the ability to find the suitable network structure and size. The first variant performs unsupervised learning and grows until a performance criterion is met. A system is used to determine where to add new neurons and to find ones that are superfluous. The second variant combines the self-organizing network with a radial basis function (RBF; described in section 3.1.4) approach. The number, diameter, and position of RBF units are determined automatically. The positioning of the RBF units and the supervised training of the network are done in parallel. The current classification error is used to determine the locations of new RBF units.

The *self-organizing incremental neural network* (SOINN) is introduced in a paper by S. Furao and O. Hasegawa [45]. It is a two-layered network with a topological structure of the input pattern as the first layer and prototypical nodes of the clusters as the second layer. The algorithm described in [45] adds and removes edges based on the Hebbian rule. To summarize, it connects two closest nodes for each input signal and it removes a connection that has not been refreshed recently. Nodes are removed to separate low-density overlap clusters and nodes are added when after removing edges the number of inputs signals becomes higher than a certain condition.

However, a utility parameter is used to halt the insertion of nodes to prevent overfitting.

The *enhanced self-organizing incremental neural network* (ESOINN) is proposed by S. Furao, T. Ogura and O. Hasegawa [46]. It enhances SOINN by using just one instead of two-layer and making it more stable for on-line learning. In contrast to SOINN, it can also separate clusters with high-density overlap and it only inserts between-class nodes to realize incremental learning. The method uses fewer parameters compared to SOINN as well.

In 2008, presented S. Furao and O. Hasegawa the *adjusted SOINN classifier* (ASC) [47]. This SOINN based method is in contrast to SOINN designed for supervised learning. By using an adaptive similarity threshold, the system can grow incrementally and accommodate input patterns of the incremental data distribution. It deletes the within-class nodes to reduce the number of parameters compared to SOINN. The method can also reduce prototypes that are caused by noise in the data.

Another addition to the SOINN family was done in 2010/11 by S. Furao, H. Yu and K. Sakurai [48]. It is an extension of SOINN. The method is semi-supervised, meaning it can learn labeled and unlabeled samples. Compared to SOINN it has only one layer. It labels 'teacher' nodes and uses them to label all unlabeled nodes. The algorithm does not require knowledge of the number of nodes or classes in advance.

3.1.4 Radial Basis Function

The *Radial Basis Function* (RBF) is a function whose real-valued value depends on the distance to a center point $\mathbf{y}$. As explained by M. Buhmann in [49], the function uses the Euclidean distance in q-dimensional Euclidean space. The standard radial function approximation is defined as,

$$\phi(\mathbf{x}, \mathbf{y}) = \phi(||\mathbf{x} - \mathbf{y}||), \tag{3.1}$$

where $\mathbf{x} \in \mathbb{R}^q$. The radial basis function is often used in function approximations. The problem for such an approximation is described in a paper by D. Broomhead and D. Lowe [50] as in Definition 3.1.

Definition 3.1. Given a set of N distinct vectors (data points), x_i with $i = \{1, 2, \ldots, N\}$ in $\mathbb{R}^q$ and N real numbers f_i with $i = \{1, 2, \ldots, N\}$, choose a function $s : \mathbb{R}^q \to \mathbb{R}$ which satisfies the interpolation conditions: $s(\mathbf{x}_i) = f_i, i = \{1, 2, \ldots, N\}$

This function approximation s can be used in a network. A neural network using radial basis functions is called *radial basis function network*. The RBF networks were introduced by D. Broomhead and D. Lowe [50]. These networks are usually three-layer neural networks: an input layer, a hidden layer, and an output layer. The function s for such the most basic RBF network looks like

$$s(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \phi(||\mathbf{x} - \mathbf{y_i}||), \tag{3.2}$$

where $\mathbf{x}$ are the data points or input of the network, $\phi(||\mathbf{x} - \mathbf{y_i}||)$ the radial basis function in each hidden node i and α_i the weight for the hidden node i and with $i = 1, 2, \ldots, N$.

We described in section 3.1.3 an incremental learning system by B. Fritzke [44]. The second variants of these SOINN use it in combination with a radial basis function. The number, diameter, and position of these RBF units are automatically determined by the system. The radial basis functions used in this network are slightly different from the one showed in Equation 3.2.

In a paper by L. Bruzzone and D. Prieto [51], is an incremental learning radial basis function network approach proposed. It is designed to primary requirements of a robust classifier for remote-sensing images. The method uses a Gaussian RBF neural network. Each neuron in the hidden layer has a Gaussian kernel function $\phi(.)$, characterized by a centre y_i and a width σ_i. The weighted summation of the output neurons for this network is given by

$$o_j(\mathbf{x}_n) = \sum_{i=1}^{N} \alpha_{ij}\phi_i(\mathbf{x}_n) + b_j, \tag{3.3}$$

where m is the number of hidden neurons, w_{ij} is the weight of the hidden neuron i and output o_j and b_j is the bias for output o_j. The method has a self-organizing architecture as well as an adaptive structure.

Another RBF network approach was proposed by S. Ozawa et al. [52]. Their approach is a combination of Incremental Principal Component Analysis (IPCA) and a Resource Allocating Network with Long-Term Memory (RAN-LTM) from earlier work. The Resource Allocating Network is an extension of the Radial Basis Function (RBF) in which the allocation of hidden units is done automatically. The Long-Term Memory (LTM) is used to store items of input-output data. These stored items are retrained with new data to suppress forgetting.

3.1.5 Other methods

The method proposed by B. Ans and S. Rousset [53] aims to suppress catastrophic forgetting and reduce retroactive inference in gradient descent algorithms. The method uses a dual neural network system. Information is transferred between the two networks, which differ in the number of hidden layers and layer sizes. By having two networks, one can retain information, while the other has high ability to generalize.

A semi-supervised method is proposed by H. Al-Behadili et al. [54]. Their approach uses extreme learning machine (ELM) and extreme value technique (EVT). Since the proposed method is semi-supervised and semi-supervised methods are sensitive to false labels, EVT is used to detect outliers.

3.2 Convolutional Neural Network

This section gives an overview of methods used for incremental learning in convolutional neural networks. The work of V. Lomonaco and D. Maltoni [55] compares different learning strategies for incremental learning for convolutional neural network based architectures for real-world applications. As they mention, a naive approach to incremental learning would be to store all seen data. Each time data arrives, from either already known classes or from new classes, this approach would discard its' current model and train a new model from scratch. In real-world applications, this is not practical, because of limitations in memory and computational resources. The solutions described in the paper are framed in three main strategies:

1. training/tuning an ad hoc CNN architecture suitable for the problem,
2. using an already trained CNN as a fixed feature extractor in conjunction with an incremental classifier and
3. fine-tuning an already trained CNN.

They conclude their work that on-line and incremental learning are still scarcely studied in the field of convolutional and deep neural networks. In the remainder of this section, we will go into detail of several approaches applying these main strategies.

3.2.1 Boosting Inspired Methods

A method, that is inspired by AdaBoost and Learn++, is introduced in a paper by D. Medera and S. Babinec [40]. The CNN's are used in this approach as weak-learners for the boosting classifier. The CNN's are kept simple and undersized or with a high error. In the incremental phase, each time new data arrive, a certain number of new CNN's are trained. All existing CNN's are fine-tuned on the new data.

In contrast to AdaBoost [56], [40] uses a distribution $\mathbf{p}^t$ to divide the samples in a training and test set for the CNN. Determining the error of hypothesis h_t works the same as in AdaBoost: all p_i^t of the correct predicted samples are summed and then β_t is calculated. However, hypothesis h_t is discarded when the error ϵ_t is larger than $\frac{1}{2}$. The composition of all hypotheses h_t of all previous t iterations, H_t, is generated by using majority voting,

$$H_t = \arg max_{t:h_t(x)=y} \sum_{t:h_t(x)=y} \log \frac{1}{\beta_t}.$$

The error of the hypothesis H_t determined by summing the p_i^t of incorrectly predicted examples. Here as well, the current h_t is discarded if the error is higher than $\frac{1}{2}$. The error is normalized when the h_t is not discarded. Next, the weights are updated. If the sample was classified correctly, then multiply its weight by the normalized error of H_t, else keep the weight the same. Finally, the final hypothesis is computed by

$$H_{final} = \arg max_{y \in Y} \sum_{t=1}^{T} \sum_{H_t(x)=y} \log \frac{1}{B_t},$$

where B_t is the composite normalized error. This composite normalized error is calculated as

$$B_t = \frac{E_t}{1 - E_t},$$

where E_t is the sum of the weights of the incorrectly predicted samples.

Another boosting method in incremental learning in convolutional neural networks is the *Incremental Boosting CNN* (IB-CNN) method proposed by S. Han et al. [57]. Boosting is in this method integrated into a boosting layer and an incremental boosting layer. These layers are located after the last fully connected layer of the CNN. The model is trained in an iterative manner.

In the model are the outputs from the last fully connected layer used as features. In the boosting layer are each iteration features selected by using the AdaBoost algorithm. The selected nodes are activated, whereas the other nodes remain inactivated. The combined hypothesis of the boosting layer at time step t for feature vector $\mathbf{x}_i$ with dimension K is defined as

$$H^t(\mathbf{x}_i^t) = \sum_{j=1}^{K} \alpha_j h(x_{ij}),$$

where α_j is the weight for node j and $h(x_{ij})$ the hypothesis. In the incremental boosting layer are the selected nodes from the current iteration combined with the activation's of selected nodes from previous iterations. This is defined as

$$H_I^t(\mathbf{x}_i^t) = \frac{(t-1)H_I^{t-1}(\mathbf{x}_i^{t-1}) + H^t(\mathbf{x}_i^t)}{t},$$

where $H_I^{t-1}(\mathbf{x}_i^{t-1})$ is the incremental boosted classifier from time step $(t-1)$ and $H^t(\mathbf{x}_i^t)$ the strong classifier from time step t. During testing time is just the incremental boosting layer

used for the fully connected layer and the boosting layer remains unused. The main idea in this approach is the same as in AdaBoost, with the difference that the weights are used to select nodes from the fully connected layer instead of samples for the weak-learning algorithm.

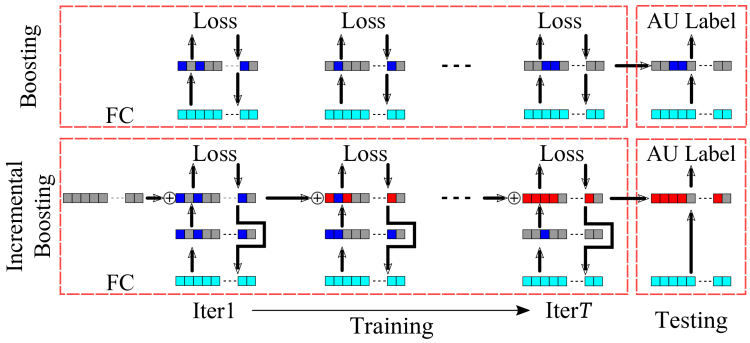

Figure 3.1: *Figure from the paper by S. Han et al. [57] showing the boosting CNN and incremental boosting CNN. The blue nodes are activated nodes at the current iteration and red active nodes from previous iterations. The grey ones are inactive.*

3.2.2 Fine-tuning inspired methods

A different approach to incremental learning with convolutional neural networks is fine-tuning. This method is applied in a paper by C. Käding et al. [4]. The paper focuses on scenarios where new data is added to already known classes and where new classes are added to the current ones. Their method uses warm-start optimization, which means that a pre-trained model is used and the parameters from a previous time step are used for the current time step t. The method has two main steps: 1. the initial learning and training step, and 2. the update step when new samples arrive. The network does not require any modification in the situation where new data of existing classes arrives. However, when new classes are added, the output layer needs additional nodes for the new classes as well as connections to the lower fully connected layer. The parameters for these connections are initialized randomly using a normalization technique.

The *iCaRL* algorithm, proposed by S. Rebuffi, A. Kolesnikov and C. Lampert [58], is a method that resembles fine-tuning. It has three main components,

1. a nearest-mean-of-exemplars classifier, this method requires only storing few examples per class,
2. prioritized sample selection based on herding, and
3. representation learning step to avoid catastrophic forgetting.

The nearest-mean-of-exemplars classifier is a classifier that assigns the label of a class, whose mean is closest to the input image to the input image. However, the used approach in iCaRL is not fully a nearest-mean-of-exemplars classifier. It only stores a sample of examples per seen class. This is, because of limited memory available in practice. The method in [58] uses a flexible number samples per seen class. Assume that M is the memory size and K_t the number of seen classes at step t, then M/K_t samples per class are kept in memory. The examples kept in memory are chosen such that they represent an approximation of the class' mean. The training of iCaRL works on a high level as follows. First, the model parameters are updated with the new dataset and the exemplars sets (sets of examples per class). Next, the current exemplar sets are reduced in size such that sufficient memory is available for examples of the new classes. This is done with a 'herding' method. The examples at the end of the fixed order are removed. The next step is to construct new exemplar sets. For a new set the M/K_t examples that are closest to the class' mean are selected in a fixed order. This means, that the first to be selected, represents the mean

of the class the best, and the last selected, the least best of the M/K_t selected of the set examples. This to ensure the best possible representation after removing examples at a later stage.

The method of updating of the model parameters resembles some extent fine-tuning. The model from the previous step is used as a warm-start and is updated. However, there are two modifications. The input data set is augmented to contain the new examples and the stored examples from previously seen data. The loss function is augmented as well to encourage improvements to the feature representation as well as allowing classifying new classes.

After training the model can be used to classify an image. Classifying an image requires the exemplar sets, $P = (P_1, \ldots, P_K)$, and a feature map. In this paper is a convolutional neural network seen as a trainable feature extractor. The first step is to determine the mean of the examples per class, for each class $y = 1, \ldots, K$ do

$$\mu_y = \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p),$$

where $\varphi(p)$ is the trainable feature extractor. Then the class label of x can be predicted by choosing the class whose mean is closest to x,

$$y^* = \arg min_{y=1,\ldots,K} ||\varphi(x) - \mu_y||.$$

Another, by fine-tuning inspired method, is *Learning without Forgetting* (LwF) proposed by Z. Li and D. Hoiem [59]. In this method, are the parameters of the convolutional neural network split up into shared parameters θ_s, task-specific parameters for all old tasks θ_o and task-specific parameters for the new tasks θ_n. The shared parameters are defined as all parameters of the convolutional, subsampling and fully connected layers. The task-specific parameters, θ_o and θ_n, are the parameters of the classification layer. Here are at time step t, the θ_o parameters the parameters from all tasks at time step $t-1$ and the θ_n the parameters from the tasks that are new at time step t.

Learning without Forgetting starts the initialization by classifying all images, $x_{n,i} \in X_n$, from the new task with the original network. This by using θ_s, θ_o and the randomly initialized θ_n for each new class of images in the new task. The results of each image, $\mathbf{y}_{o,i}$, is a vector of the probabilities per class. For all images, this is defined as Y_o. After the initialization, the training starts witg all tasks and regularization $\mathcal{R}$. The training is done in two steps: first, θ_s and θ_o are frozen and only θ_n is trained until convergence, then all weights θ_s, θ_o and θ_n are trained together until convergence. For the training of the new tasks is multinomial logistic loss used to encourage the predictions $\hat{\mathbf{y}}_n$ for an image to be consistent with the ground truth $\mathbf{y}_n$ of that image,

$$\mathcal{L}_{new}(\mathbf{y}_n, \hat{\mathbf{y}}_n) = -\mathbf{y}_n \cdot \log \hat{\mathbf{y}}_n.$$

For the old tasks is Knowledge Distillation loss used. It increases the weights for smaller probabilities. It is defined as

$$\mathcal{L}_{old} = -H(\mathbf{y}'_o, \hat{\mathbf{y}}'_o) = -\sum_{i=1}^{l} y_o'^{(i)} \log y_o'^{(i)},$$

where l is the number of classes. $y_o'^{(i)}$ and $\hat{y}_o'^{(i)}$ are defined respectively as

$$y_o'^{(i)} = \frac{(y_o^{(i)})^{1/T}}{\sum_j (y_o^{(j)})^{1/T}}$$

and

$$\hat{y}_o'^{(i)} = \frac{(\hat{y}_o^{(i)})^{1/T}}{\sum_j (\hat{y}_o^{(j)})^{1/T}},$$

where T in [59] is set to 2. The new parameters are determined by

$$\theta_s^*, \theta_o^*, \theta_n^* \leftarrow \arg min_{\hat{\theta_s}, \hat{\theta_o}, \hat{\theta_n}} \left(\lambda_o \mathcal{L}_{old}(Y_o, \hat{Y_o}) + \mathcal{L}_{new}(Y_n, \hat{Y_n}) + \mathcal{R}(\hat{\theta_s}, \hat{\theta_o}, \hat{\theta_n}) \right),$$

where λ_o is a loss balance weight to increase the weight to favor old tasks more. It is mostly set to 1.

Less-Forgetting Learning by H. Jung et al. [60] is to some extend similar to Learning without Forgetting. When it generates a new model M_t, at time step t, it copies the model from time step $t-1$, in the paper referred as respectively the domain (M_{t-1}) and target network (M_t). After copying the parameter values of the previous step as initial weights, it freezes the weights of the classification layer to maintain the boundaries of the classifier. Then it is trained to minimize the loss function

$$\mathcal{L}_t(x, \theta_{t-1}, \theta_t) = \lambda_c \mathcal{L}_c(x, \theta_t) + \lambda_e \mathcal{L}_e(x, \theta_{t-1}, \theta_t),$$

where $\mathcal{L}_t$ is the total loss function, $\mathcal{L}_c$ is cross-entropy loss function and $\mathcal{L}_e$ is Euclidean loss function. θ_{t-1} and θ_t are the parameters for the model at $t-1$ and t. Furthermore are λ_c and λ_e tuning parameters and is x a sample from the dataset at time step t. The cross-entropy loss function, $\mathcal{L}_c$, is defined as

$$\mathcal{L}_c(x, \theta_t) = \sum_{i=1}^{K} l_i \log o_i(x, \theta_t),$$

where K is the number of classes, l_i the value of label i and o_i the i-the output of the model at time step t. This loss function helps the network to classify input x correctly. The other loss function, Euclidean loss, $\mathcal{L}_e$, is defined as

$$\mathcal{L}_e(x, \theta_{t-1}, \theta_t) = \frac{1}{2} ||\mathbf{f}_{L-1}(x, \theta_{t-1}) - \mathbf{f}_{L-1}(x, \theta_t)||_2^2,$$

where L is the total number of hidden layers and $\mathbf{f}_{L-1}$ a feature vector of the layer $L-1$. This loss function makes the network learn to extract features which are similar to the features extracted by source network.

The method proposed in a paper by T. Xiao et al. [61] uses fine-tuning in one of its components. Its' model grows organically and hierarchically when data from previous and new classes become available. Seen from a high level, are the classes in the model split in a tree-like way and grouped in groups of similar classes. The different components of the tree use cloning to initialize parameters and use them as a warm-start for training. Thus, to some extent, it is using fine-tuning.

The main procedure in the method, in the paper called ExtendLeafModel, has all classes of this leaf, where at the start all classes are in a single leaf, in one superclass S and they are predicted by one leaf model L_0. When new classes arrive, then the method creates two new models, a flat incremental model, and a set of leaf models. In the flat incremental model L_0' are the new classes added to the top classification layer. In the set of leaf models $\{L_1, \ldots, L_K\}$. Here is the N_0 classes portioned into K superclasses. Each model L_i predicts in its subset of classes. A branch model B is used to direct the prediction to the correct leaf model for a given input. The new leaf models $\{L_1, \ldots, L_K\}$, as well as the branch model B, clone the parameters from the model L_0. These are then trained using them as a warm-start. The classes are split and grouped in superclasses, in this method, based on similarity by using error-driven preview and spectral clustering partition. It generates a confusion matrix, C, where each C_{ij} is the probability that class i is predicted as class j. All K new superclasses have the same topology as the L_0 model

except for the classification layer.

To choose what model to use, the flat incremental model or the leaf models, the method trains all models and then let them compete in accuracy. To make this approach incremental it is generalized and creates a deeper hierarchy. First, the new classes are distributed among the existing superclasses by the error-driven preview, then the algorithm is described above is done for each of these superclasses.

3.3 Experiments in Literature

In the literature of CNN's used in on-line and incremental learning, as described in 3.2, are several types of experiments done. These experiments can be grouped in a few types of experiments. In the first type is the approach in the initial phase trained on a dataset A. That is followed by training the model on a dataset B in the incremental phase. In these experiments is measured what the accuracy is on the test sets of both datasets. The second type is a variant of the first. In this experiment is the training set of dataset B added gradually. After each step in adding the dataset is the model tested on the test set of the dataset A and test set that contains all data of dataset B learned thus far. The third type is a dataset that is split in x parts. These x parts trained gradually the approach. After each or a combination of steps is the model evaluated.

In the Learn++ inspired method by D. Medera and S. Babinec [40] is the third method used. They use the MNIST [62] dataset and create four independent training sets. Each contains 2500 images. The test set of each of these consists of the next 5000 images. As CNN is a small CNN designed which is explained in their paper. The other boosting based method, IB-CNN by S. Han et al. [57], benchmark their approach on four facial AU-coded (action unit) datasets. In their experiments is the IB-CNN compared to a regular CNN, B-CNN (boosting CNN) and a variant of IB-CNN, IB-CNN-S. In the experiments is the CNN based on a modification of cifar10_quick from Caffe [63]. In the paper is not clearly explained whether and how their experiments are on-line or incremental.

The fine-tuning approach by C. Käding et al. uses the AlexNet in their experiments. In their experiments are the weights of the CNN pre-trained using the ILSVRC-2010 [64] dataset. For their experiments are the MS-COCO-full-v0.9 [65] and Stanford40Actions [66] datasets used. The model is initially trained on ten classes with 100 images for each class. In the incremental phase are five classes added one-by-one. In Less Forgetting Learning by H. Jung et al. [60] is the first method used. Here is first trained on one dataset and then in the incremental phase trained on another one. After training on the second one, is the model evaluated on the test set of both datasets. In the experiments are the MNIST and CIFAR-10 in color used as dataset A and SVHN and CIFAR-10 in gray color as datasets B. In Learning Without Forgetting [59] by Z. Li and D. Hoiem is mainly the AlexNet used. In this paper are the first and second type of experiments used. The CNN is initially trained on ILSVRC-2015 [64] or Places365-standard [67] datasets. In the incremental phase is the model trained on either PASCAL VOC 2012 image classification [68], Caltech-UCSD Birds-200-2011 fine-grained classification [69] or MIT indoor scene classification [70]. These new datasets are either at once trained or gradually added to the model. In iCaRL by S. Rebuffi, A. Kolesnikov and C. Lampert [58] are in the experiments the CIFAR-100 [71] and ILSVRC-2012 [64] datasets used and the 18-layer and 32-layer ResNet [72] used as CNN's. The images are added in batches of 2, 5, 10, 20 or 50 classes at a time for CIFAR-100 and in batches of 10 or 100 for the ILSVRC small subset respectively the full dataset. The experiments are of the third type. After each batch of classes is added is the model evaluated. In approach by T. Xiao et al. [61] create two datasets containing the animal classes with more than 100 images from respectively the ImageNet_1K and ImageNet_22K. The model is evaluated after training every 30,000 images.

Chapter 4

Proposed Approach

We have seen in the literature that incremental learning approaches using convolutional neural networks use two main approaches. These are methods based on fine-tuning or boosting. In this project, we propose three main approaches. The first one is fine-tuning. The next one is combining convolutional neural networks. This one is neither fine-tuning nor a boosting approach, although it shows ideas from boosting. Finally, we propose boosting approach.

4.1 Fine-tuning

The most naïve approach of on-line learning with a convolutional neural network is to have a single network and to fine-tune it each time new data arrives. This is a very simple idea and easy to implement. However, soon the problem arises of what size and architecture the initial CNN should have and how new classes of images should be handled. Another problem with this approach is that weights and biases of nodes, depending on the design, can change constantly. The risk of catastrophic forgetting arises. The earlier mentioned paper 'Fine-Tuning in an Incremental Environment" by C. Käding et al. [4] explored this approach. In this work, we will use this as one of the approaches we compare. We will use two setups in fine-tuning a convolutional neural network.

The final connected layer in a CNN has in an off-line environment the same number of nodes as classes. We do not know this number of classes initially in an on-line environment. In our first setup, we use a large initial number of nodes for this fully connected layer. The weights and biases of these nodes are randomly initialized. All nodes are at the start not assigned to a class. In each step, when new classes arrive, more and more nodes get assigned to a class. Each time before the CNN get fine-tuned, all unassigned nodes are re-initialized. This is done because the weights and biases can change during fine-tuning even when these are not assigned to a class. The implementation of this approach is simple since the number of nodes is fixed. The downside, however, is that at some point all nodes are in use and the network cannot learn any new classes. Also, we have the problem that we have to decide at the start what size this layer should be. A too big final layer could also hurt how well the network learns and what accuracy it achieves.

The other main approach is the opposite of the first one. Instead of using a fixed number of nodes, it uses a flexible number. The fully connected layer has always the same number of nodes as the number of classes seen. In each step when new classes arrive new nodes are added. These nodes are initialized randomly. The benefit of this approach over the other is that the network can grow as much as needed. The problem we will encounter at some point is how many classes the network can handle. It cannot grow indefinitely. A network might work well with a thousand classes. One could grow the final layer to accommodate a million classes, but the network might not able to handle this.

Both approaches are shown in Figure 4.1. Here is the top one the final fully connected layer of the large CNN. The gray is the used nodes and the white the unused ones. The bottom one is the final connected layer of the approach that adds nodes when needed. The dark gray shows the added nodes.

Usually, in fine-tuning, all parameters get trained. This strategy might not work in an incremental environment. It could cause catastrophic forgetting. We will use an idea from the paper

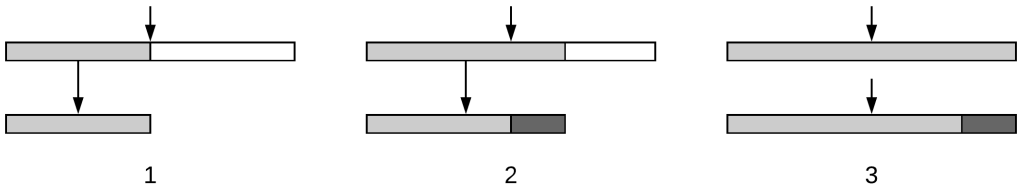

Figure 4.1: Figure showing two fine-tuning approaches. The top one is the initial large CNN and in the bottom one are nodes added when needed. The gray represent the used nodes, the dark gray the added nodes and the white the unused nodes.

"Learning Without Forgetting" by Zhizhong Li and Derek Hoiem [59] in our setups. They split the parameters of a CNN into three groups. First is the set of shared parameters θ_s. Those are the parameters of the convolutional layers and fully connected layers that are not the final fully connected layer. The second set is the set of task-specific parameters of previously learned classes θ_o. These parameters are from the nodes of the final connected layer that are assigned to classes learned in previous steps. The last set is the set of task-specific-parameters of new classes θ_n. These parameters are also in the final fully connected layer but are assigned to the classes that arrived at the current step. The nodes in out large CNN approach that are not assigned to a class are ignored in this model.

We can develop several fine-tuning procedures based on this idea. We can freeze certain sets of parameters to influence the learning in the incremental phase. In this work, we will use four different procedures. The first procedure is to not freeze any parameters during the fine-tuning. All parameters get fine-tuned. We will call this approach the *Classic* approach. In the second approach, that we call *FreezeO*, we freeze the old task-specific parameters θ_o. The shared, θ_s, and new task-specific parameters, θ_n, will be fine-tuned during training. In the third procedure, *FreezeOS*, the old task-specific parameters θ_o as well as the shared ones θ_s are frozen. During training, only the new task-specific parameters are fine-tuned. The last procedure, *FreezeOS+F*, combines the third with the first. It starts by fine-tuning the only new task-specific parameters θ_n. Next, it unfreezes all parameters and fine-tunes again like the classic approach and fine-tunes all parameters.

4.2 Combined Convolutional Neural Network

Instead of adding new classes into the same convolutional neural network as in fine-tuning, we can also train a new CNN when new classes arrive. In this approach, the main idea is to generate a new CNN each time new data arrives. We end up with multiple convolutional neural networks over time. When we want to classify an image after training multiple CNN's, we predict the image in each individual CNN and then combine their predictions. We have two approaches to combine the individual predictions.

In the first one, we do the predictions in parallel. At training time, we train a new CNN each step t when new data arrives. For each image during test time, we let all CNN's make a prediction for it. Each CNN returns its' prediction, the class with the highest probability, and the probability of this prediction. We store the returned predictions and their probabilities in a data structure such that we can remember which prediction had which probability. For the final prediction, we select the prediction with the highest probability.

This strategy is shown in Algorithm 2. It iterates over the M CNN's and get for each m the prediction $y'_{m,x}$ and its probability $p_{m,x}$. For the final prediction, it uses $argmax$ to get the location

of the class with the highest probability overall.

Algorithm 2 Parallel Prediction

Input: An image x, trained CNN's C of combined model, where $m \in 1, \ldots, M$
for $m = 1, \ldots, M$ **do**
    Get prediction and its' probability from CNN $c_m \in C$
    $[y'_{m,x}, p_{m,x}] = c_m(x)$
end for
Make final prediction by choosing class with highest probability.
$j = argmax\mathbf{p}_x$
$y'_{final} = y'_{j,x}$

In the second approach, we add an extra class to each CNN. This class just contains noise. It contains images that do not relate to any of the other classes in the CNN. We call this the *none-of-the-above* (NOA) class. At training time, each CNN is trained the same as in the other approach with the difference that there is an extra class. We start predicting by getting a prediction from the first CNN. If the prediction is one of the classes that is *not* the NOA class, then we choose that as the prediction as final prediction. However, if the prediction is the NOA class, we continue to the next CNN. With this next CNN we do the same: if the prediction is not the NOA class, then the final prediction is that class and if it is the NOA class, then we continue to the next CNN. This is done until a final prediction has been made. If in the last CNN the NOA class the highest probability has, we choose the class with the second highest probability.

This strategy is shown in Algorithm 3. Here, the label of the NOA class is the same as the number of classes in the CNN m, K_m. We iterate over the M CNN's and when the class with the highest probability is not the NOA class, we make the final decision.

Algorithm 3 Prediction using NOA class

Input: An image x, trained CNN's C of combined model and number of classes in each CNN K_m, where $m \in 1, \ldots, M$
Initialize: $m = 1$ and $y'_final = -1$
while $y'_final = -1$ and $m \leq M$ **do**
    Get predictions and their probabilities from CNN $c_m \in C$
    $[\mathbf{p}_{m,x}, \mathbf{y}'_{m,x}] = c_m(x)$
    $j = argmax(\mathbf{p}_{m,x})$
    $o = \mathbf{y}'_{j,x}$
    if $o \neq K_m$ or $m = M$ **then**
        Make final prediction
        $y'_{final} = o_x$
    end if
    $m = m + 1$
end while

The idea of these approaches is very simple. When learning with new classes, no old data is needed to preserve old information. Also, each CNN can be highly optimized to learn new data, each individual CNN can be very stable. In an incremental environment, we can just choose random images for the NOA class. However, how accurate will the combination of results be when the probabilities of many CNN's are combined in parallel? And what will happen if you keep adding CNN's and the prediction has to go through many NOA classes to get to the CNN that contains its class? These are things we will research in the experiments.

4.3 Boosting with Convolutional Neural Networks

Our third main approach is AdaBoost in combination with a convolutional neural network. This approach is inspired by the approach in a paper by D. Medera and S. Babinec [40]. This section goes into more detail how AdaBoost for multiple classes works and how we combine AdaBoost and with CNN's.

4.3.1 Multi-class AdaBoost

In Section 2.5, we described the basic idea of a boosting algorithm and how the particular boosting algorithm AdaBoost works. In short, a boosting algorithm is an algorithm that combines a number of weak classifiers, the weak-learners, into a stronger classifier, the strong-learner. AdaBoost is one of the first algorithms of this kind. However, it is designed for a binary problem. We do image classification with an unknown number of classes in an on-line environment.

Other AdaBoost based approaches reduce the multi-class problem into a multi-binary class problem. Examples for this are in the AdaBoost paper by Y. Freud and R. Schaphire [20] and Schaphire [56]. In [20] is the algorithm the same, but it is handled differently. Instead of asking just a single predicted label from the weak-learner, they ask a set of plausible labels for each sample. The hypothesis of the weak-learner is evaluated based on whether the correct label was included in the set and the number of incorrect labels that were included. Thus, a hypothesis that includes the correct label gets a better evaluation than one that does not. Also, a hypothesis with one incorrect label gets a better evaluation than one that includes ten. In [56] the problem is reduced in a different way. Here is each label in the dataset mapped to a set of binary labels. This set is a unique sequence for each label. Each example is predicted by all weak-learners and the label for the final classification is chosen by the label that is closest to the sequence of predictions. The Hamming distance is used here.

In our algorithm, we use the multi-class AdaBoost algorithm proposed by Zhu et al. [73,74]. The multi-class AdaBoost algorithm presented in [73,74] is almost identical to the original AdaBoost. Although with a small difference. In their paper is also shown that the multi-class algorithm satisfies the same statistical justification as the original AdaBoost.

The multi-class AdaBoost presented in the paper is called Algorithm called SAMME Stage-wise Additive Modeling using a Multi-class Exponential loss function. An overview of the SAMME algorithm is shown in Algorithm 4. Here is c_m classifier m in the AdaBoost classifier and $\mathbf{x}$ a set of training images with labels $\mathbf{y}$. The difference compared to the AdaBoost algorithm in [20] is the extra term $\log(K-1)$ on line 5. When the number of classes $K = 2$ is chosen, then the algorithm reduces to regular AdaBoost. However, when the number of classes is $K > 2$, then the accuracy of the weak classifier only needs to be better than random guessing. The term $(1 - \epsilon_m)$ only needs to be larger than $1/K$ to make α_m positive. In a multi-class environment is random guessing equal to $1/K$, where K is the number of classes.

Algorithm 4 SAMME

1: Initialize weights $w_i = 1/n$, $i = 1, 2, \ldots, n$.
2: **for** $m = 1, \ldots, M$ **do**
3: Fit a classifier $c_m(x)$ to the training data using weights w_i
4: Compute $\epsilon_m = \sum_{i=1}^{n} w_i \mathbb{I}(y_i \neq c_m(\mathbf{x}_i)) \,/\, \sum_{i=1}^{n} w_i$
5: Compute $\alpha_m = \log \frac{1-\epsilon_m}{\epsilon_m} + \log(K-1)$
6: Set $w_i \leftarrow w_i \cdot exp(\alpha_m \cdot \mathbb{I}(y_i \neq c_m(\mathbf{x}_i)))$ for $i = 1, \ldots, n$
7: Re-normalize w_i
8: **end for**
9: Output $C(\mathbf{x}) = \arg max_k \sum_{m=1}^{M} \alpha_m \cdot \mathbb{I}(c_m(\mathbf{x}) = k)$

In each step, it iterates M times to generate M trained weak-learners. It starts by training a weak-learners with weights w_i in an iteration. Then it computes the error rate ϵ_m of the weak-learner on the training data and the current weights. A weighted error rate is the result. Next, it computes the weight of the weak-learner α_m. The new weights are computed with this weight α_m. Finally, the weights are re-normalized.

After all classifiers c_m are fit, the boosting classifier can make predictions. For this final prediction, the image or set of images is inserted into each of the CNN's. Each CNN returns its prediction. In the boosting algorithm is the weight of the CNN added to that class. At the end, the class with the highest value is chosen, e.g. the one that got the highest value of the sum of all the weights it got assigned.

4.3.2 Convolutional Neural Networks as Weak-learners

In our approach, we will use convolutional neural networks as weak-learners and the SAMME multi-class algorithm as boosting algorithm. In step 3 of Algorithm 4 is a CNN trained to fit the dataset using weights w_i. During the training are images of batches chosen based on these weights. After it has been trained, the algorithm continues to step 4 where predictions for all images in the training set are made. The CNN generates a vector of the probability for each image. The class with the highest probability is chosen as final prediction.

4.3.3 Image Selection During Training

In AdaBoost are the weights for all samples in the training data given to the weak-learner during training. The idea is that an image with a higher weight is more likely to be chosen than an image with a smaller weight. During training, this results in that the model sees these higher weighted images more often than others and achieves a higher accuracy on them. That means that the weights are used in the selection of images in the training batches in our approach. An image with a higher weight should appear in more batches and than an image with a smaller weight.

The approach we use to choose images for training batches is shown in Algorithm 5. We iterate over the train set until we have selected M images, where M is the batch size. To select an image, we have a random number p between 0 and 1 and a cumulative probability *cumProb*. This cumulative probability is initially set to 0. The weight of each image we iterate over is added to the cumulative probability. The image that causes the cumulative probability to grow larger than p is selected. After an image is selected, we reset *cumProb* reset to 0 and we choose new random number p. The selection procedure is started again. This is continued until we reach M images.

In this selection procedure, images with a larger weight add more to the cumulative probability *cumProb* than images with a smaller weight. This results that the larger the weight an image has, the higher its chance is that it caused the *cumProb* to be larger than p. This is in line with the idea of these weights in AdaBoost.

4.3.4 Weights per Image and per Class

By default, AdaBoost works with weights per images. The images are selected based on their weight during training of a model. We showed our approach how we select images based on these weights in the last section. We are also interested to see whether the result improves when during updating we do not generate a weight per image, but average the weights of all images of a class. The training set could contain outliers. Weights based per image could cause the algorithm to more train on them and so chance the weights negatively for its class. Also in an incremental environment can concept drift occur. The weights per image could cause the model to react too fast on this drift and cause it to be less stable.

Algorithm 5 Image selection CNN as weak-learner

1: Given M as batch size, $m = 0$ the current number of selected images, n as the number of images to select from and weight vector w
2: Initialize the $cumProb$ to 0 and select a random number p, where $p \in [0, 1]$
3: Initialize counter $i = 0$
4: **while** $m < M$ **do**
5: $cumProb = cumProb + w_i$
6: **if** $cumProb > p$ **then**
7: Add image i to current batch
8: Set m to $m + 1$
9: Reset $cumProb$ to 0 and select a new random number p
10: **end if**
11: $i = i + 1$
12: **if** $i \geq n$ **then**
13: $i = 0$
14: **end if**
15: **end while**

We want to see whether the approach performs better and more stable by using weights per class. Instead of that the weight per image changes on whether it was predicted correct or incorrect, the weight of the class is increased or decreased based on the accuracy of the prediction. The weight increases when more images get predicted incorrectly while it gets decreased when a class as a whole performs better.

Algorithm 6 SAMME with shared weights per class

1: Initialize weights $w_i = 1/n$, $i = 1, 2, \ldots, n$.
2: **for** $m = 1, \ldots, M$ **do**
3: Fit a classifier $c_m(x)$ to the training data using weights w_i
4: Compute $\epsilon_m = \sum_{i=1}^{n} w_i \mathbb{I}\left(y_i \neq c_m(\mathbf{x}_i)\right) / \sum_{i=1}^{n} w_i$
5: Compute $\alpha_m = \log \frac{1-\epsilon_m}{\epsilon_m} + \log\left(K - 1\right)$
6: Set $w_i \leftarrow w_i \cdot exp\left(\alpha_m \cdot \mathbb{I}\left(y_i \neq c_m(\mathbf{x}_i)\right)\right)$ for $i = 1, \ldots, n$
7: Re-normalize w_i
8: **for** $z \in Z$ **do**
9: $W_z = \sum_i^n y_i \left[y_i = z\right] / \sum_i^n 1 \left[y_i = z\right]$
10: **end for**
11: Set $w_i \leftarrow W_{c_i}$ for $i = 1, \ldots, n$
12: Re-normalize w_i
13: **end for**
14: Output $C(\mathbf{x}) = \arg max_k \sum_{m=1}^{M} \alpha_m \cdot \mathbb{I}\left(c_m(\mathbf{x}) = k\right)$

We show in Algorithm 6 our modified SAMME algorithm that uses weights per class instead of weights per image. The new weights per class are computed in lines 8 to 11 after doing all the usual steps. The weights per class are computed by iterating over all weights and calculating the sum of the weights of all images in a class. This cumulative weight is divided by the number of images in the class to compute the average weight. This average weight is assigned to each image in the class.

4.3.5 Incremental Learning in AdaBoost with CNN's

Up to this point, we have mostly described a non-incremental AdaBoost approach. In a non-incremental environment, we generate M CNN's as weak-learners. The M depends on the data and the network the approach is used on. In an incremental environment, we initially also train

M CNN's with the first set of training data. The training of these M CNN's follows the SAMME algorithm. Each time a new set of data arrives, M new CNN's are trained and added to the existing ones. This results in having $t \cdot M$ CNN's at each step t. Each newly trained CNN has enough nodes in the final fully connected layer to accommodate all classes from the new and old data. However, these new CNN's are not trained on previous classes.

The problem we face at this moment is what to do with the existing CNN's. These CNN's are trained on previous data and have a too small final fully connected layer. We iterate over these existing CNN's in the incremental phase and we add nodes to their final fully connected layer to accommodate all classes. Each of these CNN's is fine-tuned using one of the learning procedures described in Section 4.1. Next, we have the choice whether we want to update the weight of the CNN. We can either update the weight using SAMME as we did when we trained the CNN for the first time or we do not update its' weight. Updating the weights could cause the classifier to be more plastic, while not updating might make the AdaBoost classifier more stable.

4.4 Choice of Convolutional Neural Network

In the heart of each approach is a convolutional neural network used. The approaches work in such a way that they do not depend on a certain CNN. The CNN can be swapped out depending on the requirements of the environment. There are two requirements for the CNN. The first is that the CNN can support the input of the approach, which is in our case images and for the boosting approaches a weighted set of images. The other requirement is that the output if the CNN is something the approach can handle. In most cases that would be the top-1 prediction.

The choice of CNN would mainly depend on the problem that the model is used on. There are certain limitations though. A larger CNN might cause a better accuracy than a smaller one, but such a CNN could need more memory and have a longer computation time. When it is applied in an incremental environment, the CNN might be too large to be stored in memory or the weights and biases are too big to be stored. This should be considered when the CNN is used in combination with the combined CNN's and AdaBoost approaches. In these approaches, multiple CNN's are generated and the number of them grows when more data becomes available. The more CNN's are generated, the more storage of these CNN's is needed. In an on-line environment, this is less of a concern.

Chapter 5

Experiments

We do three main experiments with the approaches described in the previous chapter. These experiments are comparing our approaches, comparing several fine-tuning setups and comparing updating the weight of an existing CNN in the AdaBoost classifier versus not updating the weight. Each experiment consists of several setups.

5.1 Implementation

Our implementation of the approaches described in Chapter 4 consists of three parts. The first part is the CNN and the code that does the learning and predicting on the CNN. The second part is an interface for this learning and prediction. It also handles some of the learning procedures. The final part is the code for the approaches themselves. The implementation is split like this to make it more modular and flexible. So, we can reuse the code from the first and second part in the implementation of each of the approaches.

The first part uses the Caffe framework [63]. Caffe is a deep learning framework originally developed by UC Berkeley. The framework is written in C++. A model of a network does not need to be programmed but is defined in a separate text file. The learning parameters can also be defined in a text file. In our experiments, we use the default Caffe. As input in our CNN, we use in most setups the ImageData layer. For the AdaBoost setups, we modified the ImageData layer. It uses the ImageData layer code but adds the image selection algorithm as described in Section 4.3.3. We call it: WeightedImageData layer.

In the second part, we communicate with Caffe. It functions as an interface between our approaches and Caffe. It manages setting up the CNN in Caffe, giving Caffe the training set and saving the trained network. At test time it handles the prediction and can determine the top-1 prediction. For this part, we use the pyCaffe interface to communicate with Caffe.

The last part is the approaches themselves. We implemented the approaches in Python. It handles all the logic of the approaches as described in the last chapter and it uses the interface for the learning and predicting with its CNN's. In the AdaBoost approach, we used the implementation of SAMME in SciKit learn [75] as a basis for our implementation.

5.2 Dataset

In our experiments, we use the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [64] dataset. The ILSVRC-2012 dataset is rather large. It is a subset of the ImageNet database, which contains 15 million images. The data set is often used in image classification. The images are of various sizes and are divided into 1,000 classes. The training set contains about $1,000$ images of each class, which results in about 1.28 million images. The validation set and test set have respectively 50,000 and $150,000$ images with the same 1,000 categories. We do not use the validation or test set in this work.

We will use a subset of the ImageNet dataset for our experiments. To simulate an incremental environment, we feed our approaches data gradually with images. We generate seven sets and four sets containing each ten classes of images to do this. For each class are 800 images randomly selected from the ILSVRC-2012 training set. These 800 images are divided into four groups of

200 images. 4-fold cross-validation is applied on these four sets for more accurate results. In each setup are three sets, 600 images, used for training and one set, 200 images, for testing. To decrease the chance that the order of classes influences the accuracy, we will learn the sets of classes in two orders. If we number the sets from one to seven and one to four respectively, then we apply 2-fold cross-validations and learn the sets in ascending and descending order. For each setup, we average the result over the 4-fold and 2-fold cross-validation.

For each experiment, we use either the seven or four sets of ten classes as described above. In each set, no old information is available. We can see how well the approaches retain old information of previously learned classes in the experiments with this dataset. To also simulate on-line learning with limited ability to store old information, we have a second dataset of the same size. The dataset is exactly the same as the one described above. However, in each set are about $1,000$ images of the previously learned classes. When K_{t-1} is the number of classes learned in all previous steps, then the number of images available per class is $1000/K_{t-1}$. For example, in set 2 are 10 previous classes. The number of images, we choose to store in memory from set 1 is $1,000/10 = 100$. In set 4, there are 30 previous classes from set 1 to 4. Thus the number of images per previously seen class is $1000/30 = 33.33...$. We round that up to 34 images per class of sets 1 to 3. We always round up to have an equal number of images for each class. This causes the total number of previous images to be slightly above $1,000$ sometimes. The images for this old information are selected randomly from the images in the previous training sets.

There are many ways to test the accuracy in an incremental environment. You can test how well the accuracy is for the newly learned classes, but it is also interesting how well the approach performs on previously learned data and how well it retained this data. At each step t, we will test the trained model of the currently added classes as well as on previously learned information. We test the model of each individual test set $S_{v,u}$, where $u \in 1, ..., T$ and on a test set that combines all test sets $S_{v,1}$ to $T_{v,t}$. We test each test set on each model generated by the 4-fold and 2-fold cross-validation and then average over their result.

5.3 Convolutional Neural Network

The network we use in our experiments is the AlexNet network proposed by Krizhevsky et al. [19]. We use the AlexNet since it a simple CNN that is easy and relatively fast to train. Other CNN's are larger and take a longer time to train. The AlexNet has 12 layers. These layers consist of one data layer, five convolutional, three pooling and three fully connected layers. The original network uses 224×224 pixels crops from 256×256 pixels images. In our experiments, we re-sized the images to 128×128 pixels and we use 99×99 pixel crops. Table 5.1 gives an overview of the network and shows the sizes of all the layers.

In the network, all the convolutional layers use ReLU and LRN. We use the default network setup as in [19]. The first convolutional layer uses a kernel size of 11 with a stride of 4. This results in feature maps of 23×23. In the second one is a kernel size of 5, stride of 1 and padding of 2 used. These settings cause the feature maps to be the same size as its input. The other convolutional layers all use a kernel size of 3 and padding and stride of 1. The convolutional layers 1, 2 and 5 are followed by pooling layers. All of these pooling layers are maximum pooling. These layers use a kernel size of 3 and stride of 2. The fully connected layers 6 and 7 both have an output size of 4096. The output of the final fully connected layer depends on which approach it is used. In the fine-tuning where we add nodes, it has the size of the total learned classes. In the initial large fine-tune setup, it has the number of nodes set at the start of the learning. In our combined CNN's approaches, the final fully connected layer has the number of nodes that the training set the CNN is trained on has.

Layer	Contains	Size output
Data	Data augment.	3 99 99
Convolutional 1	ReLU, LRN	96 23 23
Pooling 1		96 11 11
Convolutional 2	ReLU, LRN	256 11 11
Pooling 2		256 5 5
Convolutional 3	ReLU	384 5 5
Convolutional 4	ReLU	384 5 5
Convolutional 5	ReLU	256 5 5
Pooling 5		256 2 2
Fully connected 6	ReLU, dropout	4,096
Fully connected 7	ReLU, dropout	4,096
Fully connected 8		(Depends on setup)

Table 5.1: Overview of the network used in the pooling and augmentation experiments. The first column is the name and type of the layer, the second column is other things that layer contains (ReLU is Rectified-Linear Unit and LRN is Local Response Normalization) and, the third column is the size of the output of the layer (convolutional and pooling layers: number, height and width of feature maps).

5.4 Learning and Fine-tuning Setup

For all experiments, we use the same learning parameters. These parameters are shown in Table 5.2. These values were derived by trial and error where we started with the default values used in [19]. We use a base learning rate of 0.1. When we do fine-tuning this base-learning rate start by a tenth, thus 0.01, of the normal value.

Setting	Value	Description
Base learning rate	0.1	Learning rate at start
Learning policy	fixed	Constant learning rate throughout training
Max. iterations	6,000	Stop learning after 6,000 iterations
Momentum	0.02	
Weight decay	0.0004	Rate that the weight at nodes decay

Table 5.2: Overview settings used in Caffe for training in the pooling and augmentation experiments

When we train a CNN, its' weights are initialized by a Gaussian with a standard derivation of 0.1 and its' biases are all 0. The Local Response Normalization uses a local size of 5, an α of 0.0001 and a β of 0.75. In the fully connected layers 6 and 7 is a drop-out rate of 0.5 used.

5.5 Data augmentation

In the Chatfield et al. [76] and Krizhevsky et al. [19] papers are methods described to reduce over-fitting. These methods are designed to artificially enlarge the used dataset. The methods described are taking random cropped patches and flipping. The first method works by taking a random cropped patch from the original image. The original image has a size of 256×256 pixels. A random patch of 224×224 pixels is extracted during training time. In our experiments, we work with smaller images and thus smaller patches. We take a random 99×99 pixels patch from 128×128 pixels images. Taking a 99×99 sized patch means there are $29 \cdot 29$ available patches and thus increasing the dataset size by a factor of 841. During test time is always the 99×99 center patch used. The second method is simple. It flips the image on its horizontal

axis. So, the top becomes the bottom and vice-versa. This increases the dataset by a factor two. Combining the two methods enlarges the dataset by a factor of 1,682.

5.6 Experimental Setups

In this work, we have three main experiments. The setups and what we research in these experiments are explained in this subsections below.

5.6.1 Comparing Approaches

This is the main experiment of this work. We will compare in this experiment the three main approaches we described in Section 4. The idea of this experiment is one, to see whether the approaches we proposed work and, second, how stable or plastic they are. From the results, we will see which of these approaches works best in an incremental and on-line environment.

We have two setups for each of the three main approaches. In Table 5.3 is an overview of all approaches given. We use fine-tuning with an initial large fully connected layer (FineL) and one that adds nodes (FineA) to that layer when new classes arrive. These setups use FreezeOS on the new only dataset and FreezeS when also old data is available. We decided that based on the results of the experiment of Section 5.6.2. The first combined CNN's approach (CcnnP) is combined CNN's where all predictions are done in parallel and the class with the highest probability is chosen. In the other combined CNN's approach (CcnnN) are the predictions done in series using the 'none-of-the-above' (NOA) class. We use the weights per image (AdaIW) and weights per class (AdaCW) in the AdaBoost setups. The AdaBoost setups use the FreezeOS on the only new and FreezeO on the old and new data during fine-tuning. All setups use the network, learning setup and data augmentation from the sections above.

Name	Approach	Details
FineL	Fine-tuning	Large initial fully connected layer and FreezeOS (new)/FreezeO (new+old)
FineA	Fine-tuning	Add nodes to fully connected layer and FreezeOS (new)/FreezeO (new+old)
CcnnP	Combined CNN's	Combined CNN's using parallel predictions
CcnnN	Combined CNN's	Using NOA class for prediction
AdaIW	AdaBoost CNN	Using weights per image and FreezeOS (new)/FreezeO (new+old)
AdaCW	AdaBoost CNN	Using weights per class and FreezeOS (new)/FreezeO (new+old)

Table 5.3: Overview of approaches, short name we use and overview of their details

In this experiment, we will use the data set with 7 sets of 10 classes. This with all the cross-validation as described in 5.2.For the combined CNN's setup that used the NOA class, we add none-of-the-above class to each of the ten sets. These NOA classes contain images of classes that are not part of the classes from any of the sets.

Each setup is run on both versions of the dataset: the one that only contains new images and the one that also contains old information. With the data set that only contains new images, we test how well the approaches can retain old information without access to that information. This simulates incremental learning. When we use the dataset with old images, we simulate an incremental environment with some more memory or to some extend on-line learning.

5.6.2 Comparing Fine-tune Setups

In this experiment, we compare the different fine-tuning procedures. We want to see how well and which performs the best in an incremental and on-line environment. Some procedures might not work at all incrementally.

An overview of the setups we use is shown in Table 5.4. The first setup is the Classic procedure and it does not freeze any parameters during training. In the second one, FreezeO, are initially all parameters trained, but during the fine-tuning are the old target parameters frozen. In FreezeOS are the old target as well as the shared parameters frozen during training. In the last procedure, FreezeOS+F, is the CNN first fine-tuned like FreezeOS and then again fine-tuned like the Classic approach.

Name	Details
Classic	Not freezing any parameters
FreezeO	Freezing old target parameters
FreezeOS	Freezing old target and shared parameters
FreezeOS+F	Freezing old target and shared parameters, then fine-tune without freezing

Table 5.4: Overview of fine-tune setups.

We use the smaller dataset in this experiment. This dataset has four sets of ten classes. Also in this experiment are the learning settings as described above used. Similar to the first experiment are also here all setups run on both versions of the dataset.

5.6.3 AdaBoost Updating Weights vs Not Updating Weights

In the last experiment, we compare two setups of the AdaBoost approach. We want to compare here the effect of updating the weights of the individual CNN in the AdaBoost classifier during fine-tuning. As described in Section 4.3.4, can the weights of a CNN be updated or not during the incremental phase. When a new CNN is generated, the AdaBoost classifier computes a weight for this CNN. In a next step are new CNN's generated and these previous CNN's are fine-tuned. We can now either update their weights based on their error on the new data or keep their initial weight. Updating their weight could cause the model to be more plastic, while not updating maybe make it more stable.

To test this idea, we have two setups. In the first one, we do update the weights whereas in the other we do not update the weights. In both setups, we use weights for the individual images and do not share them with a class.

Name	Details
Not updating	Not updating weights of CNN. Using FreezeOS (new)/FreezeO (new+old)
Updating	Updating weights of CNN. Using FreezeOS (new)/FreezeO (new+old)

Table 5.5: Overview of AdaBoost setups.

We will use the smaller four set dataset in this experiment. This dataset contains new images will test how stable they are. The other one that does contain old information will test more their plastic side. Also here we use FreezeOS when only new data is available and FreezeO when also old information is available.

5.7 Measurements

In all experiments, we work the image classification problem. We want to classify the topic or category of the image in this problem. This means the model receives an image as input and returns a prediction of the topic of the image. This topic is one of the classes the model is trained on.

A widely-used method is the top-1 and top-5 accuracies and error rates. Due to our implementation, we cannot predict a top-5 at this moment. We will only use the top-1 accuracy. The

top-1 accuracy is the class that is selected by the model. The calculated accuracy is the number of correctly predicted images dived by the total number of images. As described earlier is this accuracy averaged over the two cross-validation procedures we use.

Chapter 6

Results

In this section, we give a detailed overview of the result of the experiments. The results are divided into sections representing the three main experiments. In the main chapter are several tables and figures shown. More detailed results are given in the appendices on this work. These detailed results include the full tables of results for the comparing approaches experiment and tables of the figures shown in this chapter. In the text and captions are the corresponding tables and appendices referenced.

6.1 Comparing Approaches

We compared two setups of each of our main approaches in this experiment. These are the fine-tuning with a large CNN (FineL), fine-tuning that adds nodes (FineA), combined CNN's with parallel prediction (CcnnP), combined CNN using the NOA class for prediction (CcnnN), AdaBoosting with weights per image (AdaIW) and AdaBoost with weights per class (AdaCW). The fine-tuning and boosting setups use FreezeOS when only new data is available and FreezeO when they also have access to old information. The setups are explained in more detail in Chapter 5.6.1.

The detailed results of these experiments are shown in Table A.1 and A.2 of Appendix A. An overview of these results is shown in Figure 6.1 on only new data and in Figure 6.2 where also old information was available. In the plots is each setup represented by its own line. The first plot, (a), shows the accuracies over time of the first set. The next plot, (b), shows the accuracies on previous sets except for the first set. The bottom two plots show results of respectively the newest set and combined sets.

The main differences between the results on the new and old and new data are that the accuracies on the old and new data are in most cases higher. The fine-tune and boosting setups go down to an accuracy of 0 on the first set with only new data. Whereas all approaches have on the new data an accuracy around 10% on the combined sets, the accuracies on previous sets except the first and on the combined sets are all higher when old data is available. The only exception to this difference is the combined CNN's setups. In almost all cases their accuracies are similar when only new and when also old data is available.

Within the results on only the new data, have the Ccnn setups on the first set, previous sets, and combined sets the highest accuracies. Especially on set 1 have the Ccnn setups a much higher accuracy. However, on the newest set is their accuracy the worst and even as low as 0. The other setups do not well on the first set as well as on previous sets. On the newest set, they do perform better. All setups follow same downwards trend in the combined results. Some a bit faster than others. The Ccnn methods have highest combined accuracy at 70 classes and are on top at each step. The FineA setups perform the worst here at 30 classes and higher.

Most setups perform better when old data is available. However, the Ccnn setups performed very similar to when only new data is available. This caused that the other setups have higher accuracies than them in many cases except on the first set. The fine-tune setups have on the first set the lowest accuracy, but on the other scenario's it performs best. The boosting setups perform very similar to them. On the first set, they had a slightly higher accuracy, but on the others slightly smaller than the fine-tune setups.

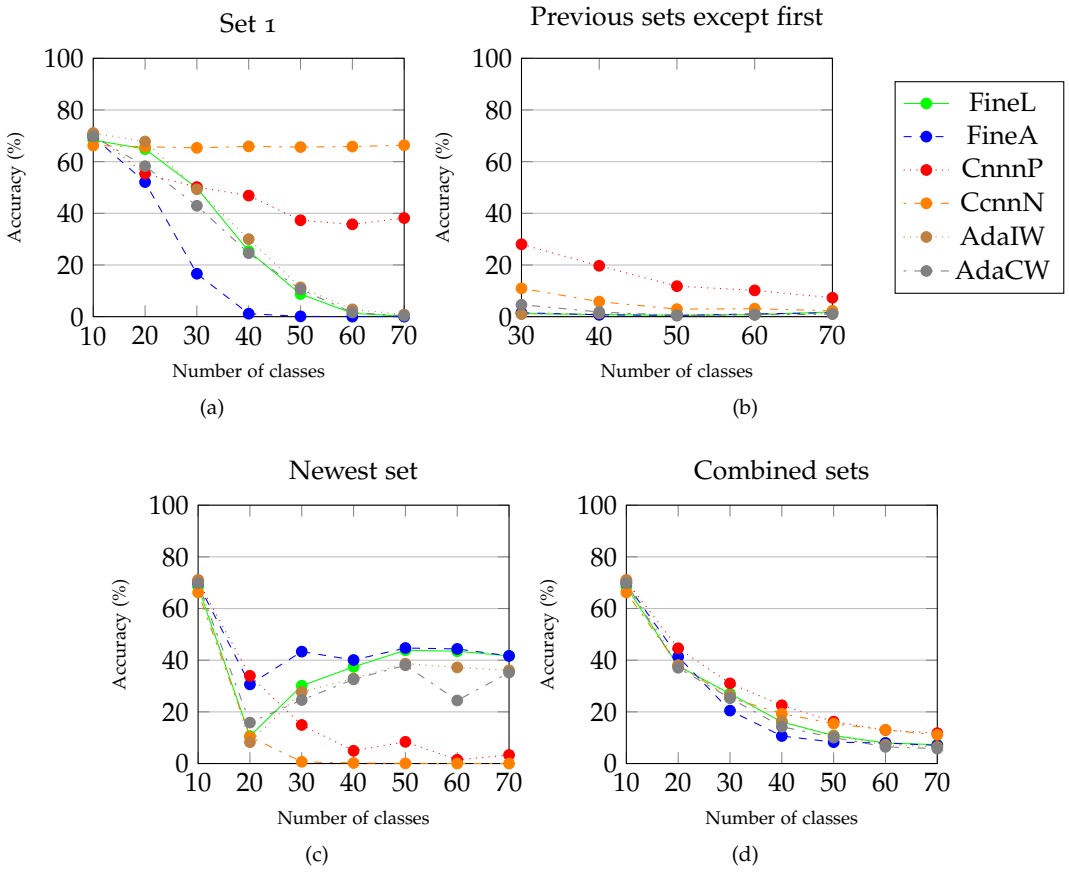

Figure 6.1: Plots showing results of comparing the learning approaches (Comparing Approaches) as described in Chapter 5.6.1 on the dataset with only new data. The data used for each of these plots is shown in Tables B.1, B.2, B.3 and B.4.

6.2 Comparing Fine-tune Setups

In this experiment, we compare the four different learning procedures for the fine-tuning approaches: Classic, FreezeO, FreezeOS, and FreezeOS+F. In this experiment, we used the fine-tune setup that has a large initial CNN. We did this on both datasets with four sets of ten classes. The setup is described in more detail in Chapter 5.6.2. The results of this experiment are shown in Table 6.1. The table is separated into four column groups. In each of these groups is one training set added to the model, Train 1 through Train 4. Each of these groups shows the results on the validation sets. These are the validation sets of the current training set, each previous training set and the validation set that combines them all. The numbers in Table 6.1 are the accuracies of the learning procedure on the validation set. This accuracy is the average over the 4-fold and 2-fold cross-validation as described in Section 5.2.

Plots from this table are shown in Figure 6.3 and Figure 6.4. These are respectively the results on the only new and new and old data. Each line represents one of the learning procedures. The y-axis is the accuracy and the x-axis the number of classes learned. Each training set adds ten new classes to the model. The first plot shows the accuracy of the model on the first set of classes. The second, the accuracy of the set of classes most recently added. And finally, the third shows the combined accuracy of all learned classes. Tables showing the data used in this figures can be found in Chapter C.

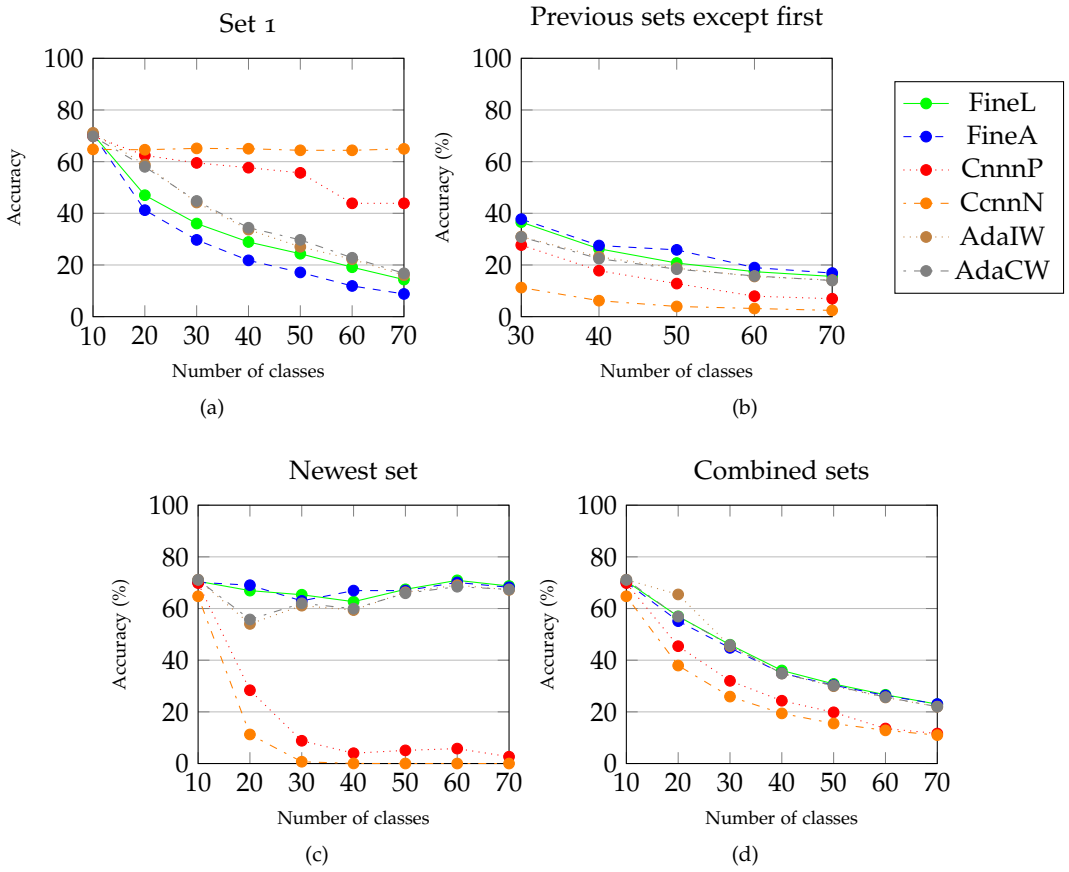

Figure 6.2: Plots showing results of comparing the learning approaches (Comparing Approaches) as described in Chapter 5.6.1 on the dataset containing new and old information. The data used for each of these plots is shown in Tables B.5, B.6, B.7 and B.8.

		Train 1	Train 2			Train 3				Train 4				
		Test 1	Test 1	Test 2	Test 1-2	Test 1	Test 2	Test 3	Test 1-3	Test 1	Test 2	Test 3	Test 4	Test 1-4
New	Classic	65.24%	0%	73.23%	36.62%	0%	0%	73.60%	24.53%	0%	0%	0%	67.45%	16.94%
	FreezeO	66.55%	1.04%	73.37%	37.21%	0.01%	0.18%	73.13%	24.50%	0%	0%	0.07%	66.88%	16.86%
	FreezeOS	63.76%	61.62%	10.69%	36.15%	46.49%	1.96%	32.40%	26.95%	26.59%	0%	2.43%	35.97%	16.25%
	FreezeOS+F	66.23%	0%	73.22%	36.61%	0%	0%	74.09%	24.71%	0%	0%	0%	67.40%	16.85%
Old	Classic	65.53%	35.44%	70.04%	52.80%	22.20%	27.98%	69.24%	39.81%	17.86%	23.09%	26.41%	64.28%	32.91%
	FreezeO	65.01%	44.83%	65.23%	56.31%	32.67%	35.88%	66.11%	45.22%	28.00%	26.36%	30.59%	60.91%	35.71%
	FreezeOS	65.48%	64.90%	3.35%	34.13%	63.78%	2.60%	11.01%	23.60%	63.50%	3.95%	3.64%	4.44%	18.39%
	FreezeOS+F	65.07%	35.34%	70.40%	52.87%	24.03%	30.52%	70.94%	41.83%	18.61%	23.83%	25.98%	64.63%	33.26%

Table 6.1: Table showing overview of results of learning procedures as described in Chapter 5.6.2 in fine-tuning experiments.

The first thing we notice about these results is the difference in accuracy between all approaches that only learned the new images and the ones also having access to old data. Whereas on the new dataset, the approaches accuracies on previous classes are very low to be even down to 0, the results on the old dataset are higher. The accuracy on previous classes is not as high as the just learned classes, but higher than when no old data is available.

When we take a closer look at the results of the procedures on only the new data, then we can see that the accuracies of Classic and FreezeOS+F are very similar. Both procedures have a zero accuracy on previous classes in all sets of the incremental phase. Also, both procedures have a similar accuracy on the current learned set. These are between 65% to 74%. The combined

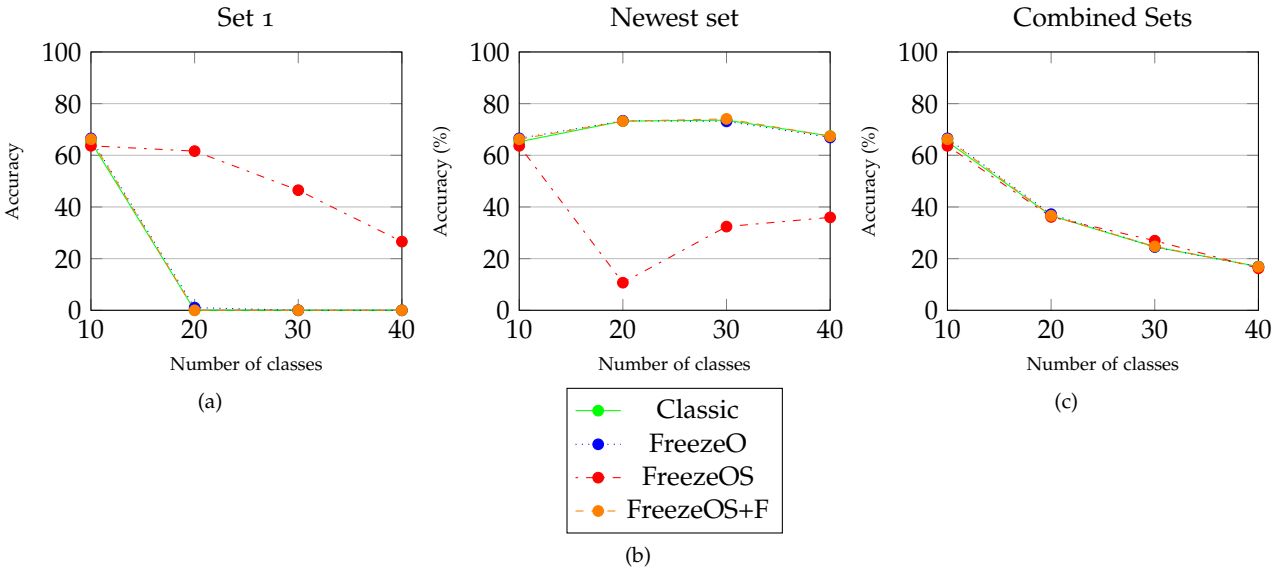

Figure 6.3: Plots showing results of learning procedures of the fine-tuning approach as described in Chapter 5.6.2 on only the first set of classes, the newest set of classes and all sets of classes. The dataset only contains images of new classes. The data used for each of these plots is shown in Tables C.1, C.1 and C.3.

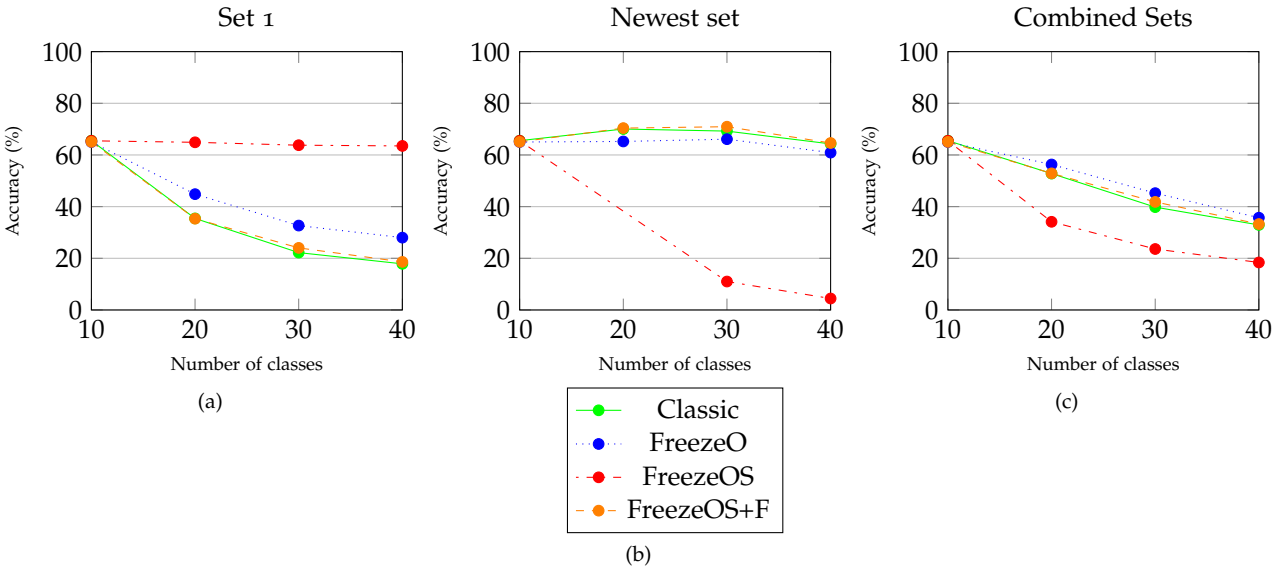

Figure 6.4: Plots showing results of learning procedures of the fine-tuning approach as described in Chapter 5.6.2 on only the first set of classes, the newest set of classes and all sets of classes. The dataset containing images of new and previously learned classes. The data used for each of these plots is shown in Tables C.4, C.4 and C.6.

validation has accuracies that are the currently learned accuracies divided by the number of sets learned. The FreezeO has similar results. It also does not very well on previously learned sets. Its' accuracies are also low, but slightly higher than zero. The FreezeOS procedure shows different results. Its' accuracies in Test 1 is, after learning each new set, the highest. The accuracy of Test 1 does decrease the more sets of classes are learned, but in Train 4 it is still over 25%. The accuracy of the currently learned set is lower than the other procedures with being between 11%

and 36% in this experiment. The accuracies on previously learned sets, that are not the first one, zero or just above zero. On Train 2 and Train 3 has FreezeOS the highest combined accuracy, however, of Train 4, its' accuracy is similar to the other ones.

The results where also old data is available, have higher accuracies. Also here, Classic and FreezeOS+F have similar accuracies. The accuracy of the newest learned set is the highest and the older a set is, the less the accuracy of it is. However, the results are not the same as when only new images are available. The FreezeOS+F has the same pattern as Classic, but its accuracies are slightly higher most of the times. FreezeO follows the same pattern, but its accuracies are higher: around 8% to 10% higher on Test 1 and 3% to 8% on other previous sets. FreezeO also has the highest combined accuracy in each set in the incremental phase. The FreezeOS procedure has the lowest accuracy on the combined validation sets. It does have the highest accuracy on Test 1 in each set, but its accuracies of other previous sets and the current set are low with 2.6% to 4.4% mostly and 11.01% on Test 3 in Train 3.

6.3 AdaBoost Updating Weights vs Not Updating Weights

In these experiments, using the AdaBoost approach, we had one setup where the weights of individual CNN's are updated when new data arrives and one that does not update these weights. The setup is in more detail described in Chapter 6.3. In Table 6.2 is an overview of the results of this experiment shown. The table is the same way build up as Table 6.1.

		Train 1	Train 2			Train 3				Train 4				
		Test 1	Test 1	Test 2	Test 1-2	Test 1	Test 2	Test 3	Test 1-3	Test 1	Test 2	Test 3	Test 4	Test 1-4
New	**No update**	66.08%	62.44%	9.46%	35.95%	43.41%	0.83%	30.51%	24.66%	29.53%	0.05%	1.95%	31.87%	15.85%
	Update	67.94%	0.36%	59.75%	30.06%	15.95%	13.46%	31.78%	20.40%	16.48%	1.10%	2.11%	31.01%	12.59%
Old	**No update**	66.30%	55.17%	53.86%	55.73%	38.11%	29.81%	63.73%	43.88%	31.75%	22.26%	28.45%	60.39%	35.71%
	Update	67.01%	54.84%	53.81%	54.33%	39.38%	29.51%	63.43%	44.11%	31.93%	22.49%	30.07%	60.32%	35.84%

Table 6.2: Table showing overview of results of updating and not updating CNN weights in the AdaBoost classifier as described in Chapter 5.6.3.

The results from this table are plotted in Figure 6.5. In these plots is the result of the new and old and new data combined. In this plots is each line one of the setups on the new or new and old data. The y-axis represents the accuracy while the x-axis the number of classes learned. Each training set adds ten new classes to the model. The first plot shows the accuracy of the model on the first set of classes. The second, the accuracy of the set of classes most recently added. And finally, the third shows the combined accuracy of all learned classes. Tables showing the data used in this figures can be found in Chapter D.

As in the fine-tuning experiments, we can see that accuracies are higher when images from previous classes are available compared to when only new ones are available. The combined accuracies on new data are much lower than on the when also old data is available. The difference is as low as half the accuracy.

In the new dataset, we can see that no updating has a higher accuracy on Test 1 in each learned set. Its accuracies are lower for the current learned and other previously learned sets than the setup that does updating. No updating has also a higher combined accuracy in all sets in the incremental phase. The accuracies with update with only new data are not stable. In Train 2 is the accuracy on Test 1 down to 0.36%, but it goes back up to 16% in the other two sets The other previous sets excluding the first, go down to around 2% in Train 4.

When old data is available, the accuracies are higher. Both updating and not updating perform similarly and similar accuracies. The differences are between 0.05% and around 2%. Also, the combined accuracies are therefore similar.

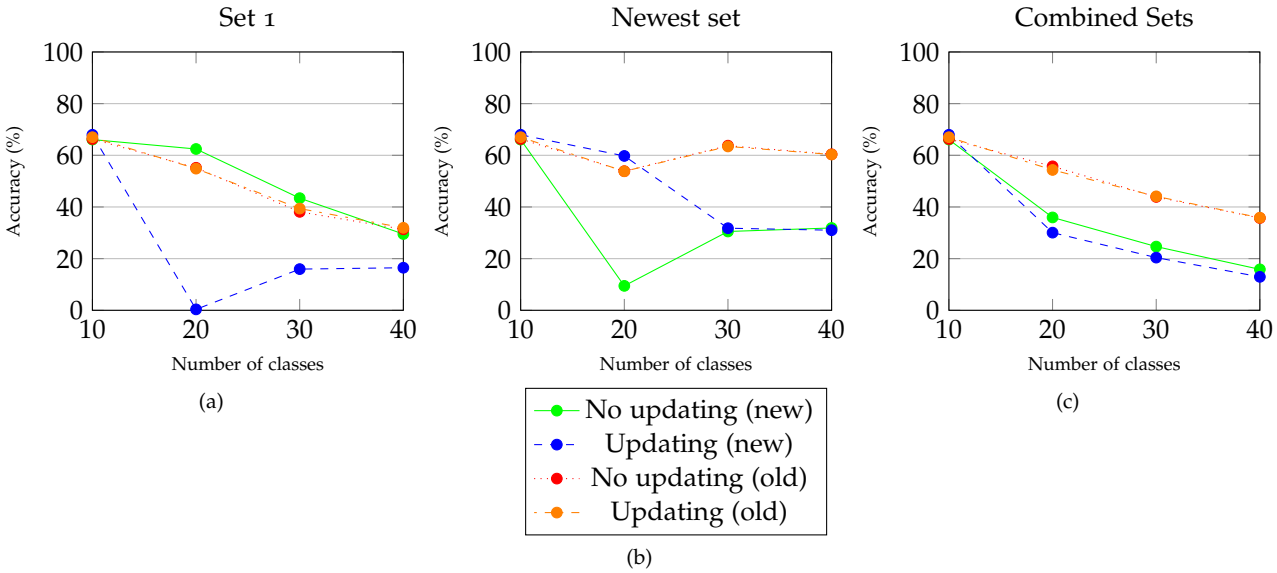

Figure 6.5: Plots showing results of updating and not updating CNN weights in the AdaBoost classifier as described in Chapter 5.6.3 on only the first set of classes, the newest set of classes and all sets of classes. The data used for each of these plots is shown in Tables D.1, D.2 and D.3.

6.4 Experiments in Literature

The experiments in the boosting approach by D. Medera and S. Babinec [40] uses a dataset split parts. This is gradually fed to the model. In every step are four classifiers created. In their results, the accuracy is between 95.52% and 99.96% in the steps on the training set and between 91.95% and 95.59% on the testing set. At the end, after training on all four sets, the model achieves an accuracy of 95.59%. In the IB-CNN approach, by S. Han et al. [57], are the experiments done on four facial action unit datasets. The final results vary from 41.6% to 95.1% on the different datasets. Their performances are similar to the state-of-the-art methods they used to compare their method with.

The fine-tuning approach by C. Käding et al. uses a pre-trained network on the ULSVRC-2010 dataset. In their experiments is the network first trained on ten classes with each 100 images and then in the incremental phase are five additional classes added one by one. Their results only use plots. There are no tables and accuracies in the text mentioned. In their experiments, the best results end up around 70 to 75% accuracy. In Less Forgetting Learning by H. Jung et al. [60] is the model trained on a dataset A and in the incremental phase on a dataset B. In the experiments, there are two versions of the model used. From MNIST to SVHN, it has accuracies of 97.37% and 90.8% respectively on the source dataset and 83.79% and 87.57% on the target dataset. Learning without Forgetting by Z. Li and D. Hoiem [59] has similar experiments as Less Forgetting Learning. The model is also trained on a dataset A first and then incrementally on a dataset B. This is done in the experiments all at once and gradually. The results at the end on learning the dataset B all at once ranging from 49.8% to 57.7% for ImageNet as the source network to 57.7% to 99.3% on the target network, where the 99.3% accuracy was achieved on the MNIST dataset. When the dataset B is added gradually the accuracy of previously learned data decreases gradually. At the end is the result on the source network just below 50% for Places365 and around 54% for ImageNet. The accuracies of the target networks range between 60% to 80%, where earlier added parts have a higher accuracy than later added ones.

The iCaRL approach by S. Rebuffi, A. Kolesnikov and C. Lampert [58] has the images of CIFAR-100 and ILSVRC-2012 added gradually. After each batch is the model evaluated. In the results

are the accuracies decreasing gradually like in our own experiments. On CIFAR-100 is initially an accuracy of about 95% achieved. This reduces to 40 to 60% in the various experiments. The experiments on the ImageNet dataset have similar results. The accuracy decreases when more classes are added. In the end, the accuracy is around 65% on the smaller ILSVRC-2012 and around 45% on the full ILSVRC-2012 dataset. The approach by T. Xiao et al. [61] is also evaluated while the data is gradually added. The animal classes of ImageNet_1K and ImageNet_22K. On the 1K dataset, the accuracy gradually increases from 58.4% after 10 epochs to 63.2% after 40. On the larger dataset is an accuracy of 51.48% achieved.

Chapter 7

Conclusion

In this work, we researched on-line and incremental learning in convolutional neural networks. Most current methods for CNN's are not designed for incremental learning. In on-line and incremental learning data becomes available gradually. This causes some challenges. Examples of these challenges are online model parameter adaption, drift, the stability-plasticity dilemma and model bench-marking. Approaches, who can handle these challenges, have only been researched very limited for CNN's.

For our research into this area, we defined a research question. This question is: "How can we do on-line learning by using convolutional neural networks?". We broke down this into

1. "What approaches have been proposed and used in neural networks and convolutional neural networks in the past?",

2. "In what ways can we learn in an on-line environment with a convolutional neural network?", and,

3. "Which of these approaches works best?"

7.1 What approaches have been proposed and used in neural networks and convolutional neural networks in the past?

In Chapter 3, we did a literature research into previous and current work in on-line and incremental learning in neural networks and convolutional neural networks. Neural network approaches that are used are based on Adaptive Resonance Theory, inspired by AdaBoost, based on Self-Organizing neural networks, based on radial basis function, using two neural networks and using extreme learning machines.

The ART based methods are using the Adaptive Resonance Theory in their neural networks. In the boosting-based methods, is AdaBoost or another boosting algorithm used. Here, multiple neural networks are combined to make a stronger classifier. In the Self-Organizing neural network are the neural networks adapting to their environment. Their connections change over time during training. The radial basis based methods use radial basis functions in their neurons. The dual neural network uses one network to retain information while the other is used to generalize. In the last method are extreme learning machines used to train the neural network.

In convolutional neural networks can incremental approaches be classified into three main groups: 1. training an ad hoc CNN suitable for the problem, 2. using a pre-trained CNN as fixed feature extractor in combination with an incremental classifier and 3. fine-tuning a pre–trained CNN. The naive approach for using a CNN in incremental learning is part of group 1. Such a naive approach would train a whole new CNN each time new data arrives by using all new and previous data. It would cost a lot of time to train a full model each time. This approach does not work in an on-line environment. We found in our literature research two main type of approaches: boosting inspired and fine-tuning approaches. The boosting based methods are in group 2, but can also be in group 3 depending on the implementation. The fine-tune methods are in group 3.

We wrote about two papers on the boosting-based methods. The first one by D. Medera and S. Babinec [40] uses CNN's as weak-learners in their boosting classifier. The CNN's can be kept simple and undersized. Each time when new data arrives a fixed number of new CNN's is trained and added to the boosting classifier. Also, the existing CNN's are fine-tuned. At test time, the image is predicted by each of the CNN's and the final prediction is made based on the weights of the CNN's. The other boosting method is proposed by S. Han et al. [57]. In this approach, only one CNN is used and the boosting is used in the two final layers of the CNN. These are the boosting and incremental boosting layer. Here the outputs in the last fully connected layer are used as input for the boosting layer. During training nodes in the boosting layers are activated. At test time only the incremental boosting layer is used to do predictions.

A basic fine-tuning based method was proposed by C. Käding et al. [4]. It uses a CNN that is being fine-tuned in the incremental environment. The iCaRL method [58] uses fine-tuning as one of its' components. In short, it stores the certain number of images per class. These are chosen based on how well they represent the class. At prediction the class is chosen to which set of images, the input image is the closest to. The CNN is here used as a feature extractor. The Learning without Forgetting [59] method developed a different learning approach. Instead of regular fine-tuning, they split up the parameters in different groups and fine-tune just certain parameters. The Less Forgetting Learning [60] approach is similar to this. It does some things a bit different. One example is the different loss function used. The last approach, proposed by T. Xiao et al. [61], uses a hierarchical model of CNN's'. Here is either a current CNN extended or a new one created when new data arrives.

7.2 In what ways can we learn in an on-line environment with a convolutional neural network?

In the literature research, we have seen the different types of approaches used in past methods. We developed in this work three main approaches for on-line learning using convolutional neural networks. These approaches are fine-tuning, combining convolutional neural networks and AdaBoost using CNN's as weak-learner.

In the fine-tuning approaches we use inspiration from among others [4, 58, 59] and combine this with some of our own ideas. The fine-tuning approaches use two main setups. In the first one, we start with a large fully connected layer in the CNN (FineL) and start using the nodes of that layer as classes arrive. The other approach adds nodes to the last fully connected layer when new classes arrive (FineA).

We proposed four different learning procedures for fine-tuning in these approaches. The parameters of the CNN are split into shared, old target and new target parameters as explained in [59]. In the first procedure, Classic, we fine-tune as usual. In the second, FreezeO, we freeze the old target parameters, during fine-tuning. In the third procedure, FreezeOS, we also freeze the shared parameters. Finally, in the last procedure, FreezeOS+F, we first fine-tune and in the third, but then also fine-tune a second time as in the Classic approach.

In the second main approach, we combine CNN's. We learn a new CNN each time new data arrives. At test time the predictions of these CNN's are combined in two different ways. In the first, all predictions are done in parallel (CcnnP). The class that is the final prediction is the one with the overall highest prediction. The second way uses an additional class, 'none-of-the-above' (CcnnN). At test time the CNN's do a prediction in series. When a CNN predicts the NOA class, we move to the next CNN. If not, then the predicted class gets to be the final prediction.

The final approach is a boosting inspired approach. We use the SAMME [73, 74] algorithm for boosting and a CNN as weak-learner. In the CNN's, we select images based on their weight. For this, we used a selection algorithm. We will experiment with two ways of using weights. In

the first, we use a weight per images (AdaIW), whereas, in the second, we use a single weight of all images of a class (AdaCW). Each time new data arrives, we learn a certain number of new CNN's and also the previously learned CNN's are fine-tuned. In AdaBoost, each weak classifier has a weight. These weights are used during the final prediction. We proposed two ways of handling those weights for previous CNN's when new data arrives. The first is not to update the weight based on its' accuracy on the new data. In the other, we do update its' weight based on the accuracy of the new data.

7.3 Which of these approaches works best?

Based on the approaches we proposed, we set up three experiments. In the first, we compared all approaches, in the second, compared the different learning procedures for fine-tuning and, in the final one, compared updating the weights of CNN's in the boosting classifier or not updating them.

All of the approaches have trouble to work well in an incremental environment when they do not have access to old data. The combined CNN's setups are more stable than the others. The CcnnN setup has the best performance on the first set overall, but the worst on the newest set. Whereas the combined CNN's setups work well on the first set, the fine-tune and AdaBoost setups perform better on the newest set. CcnnP had on Set 1 an accuracy of 38.17% and CcnnN one of 66.36% with 70 classes learned whereas the other approaches had accuracies lower than 1%. However, on the newest set had accuries of 41.66% (FineL), 41.66% (FineA), 36.09% (AdaI) and 35.21% (AdaCW) with 70 classes learned where the combined CNN's approaches had accuracies of 3.31% (CcnnP) and 0% (CcnnN). In the combined score all approaches perform similar. Of these approaches, is no one the best approach for incremental learning. There is just a least worse setup here. In this case, it is CnnnP setup. It remembers the first set almost the best with 38.1%, has the best performance on the other previous sets, with 7.32% with 70 classes learned where the others have accuracies of 2.28% and lower, and performs best overall with 11.76%. However, even this approach is not suited well for incremental learning.

The tables are turned when old data is available. The combined CNN's methods perform similar to before, but the others perform much better. Both the fine-tune and boosting methods perform similarly well on the previous sets except the first, the newest set and the combined sets. Their accuracies with 70 classes learned on the combined sets are 22.97% (FineL), 23.03% (FineA), 21.99% (AdaI) and 22.01% (AdaCW). They outperform the combined CNN's methods here by having about double their accuracy, which is 11.61% (CcnnP) and 11.00% (CcnnN). In the sets 2 and 3 (20 and 30 classes learned), FineA performs worst from the fine-tune and AdaBoost setups, but at the end, it has the highest accuracy with 23.03% versus 21.99% to 22.97% for the other three approaches (FineL, AdaI, and AdaCW). In this on-line environment are the combined CNN's approaches as we implemented them not a recommended approach to use. A reason for this is that it is designed to generate a new CNN each time new data arrives. This causes that model has multiple CNN's that are trained for that class. The other approaches perform better. The fine-tune and AdaBoost setups are a better choice. The difference in accuracy is small with a difference of just 1.04%. The AdaBoost methods are considerably more complex than the fine-tune methods. They have multiple CNN's where the fine-tuning approaches only have one. So, in this environment, the fine-tune methods would be better suited than the AdaBoost methods.

When we look at the different learning procedures in fine-tuning, then we can see that all procedures have trouble in the incremental environment when only images from the newest set of classes are available. The Classic, FreezeO, and FreezeOS+F do not retain any old information. The accuracy of the previous sets is in most cases 0% or close to 0%. They do however learn new data easily. The procedures cause the fine-tuning model to be very unstable. They do suffer from catastrophic forgetting. The FreezeOS procedure retains the information from the first set. The accuracy of this does decrease over time. From 63.76% on set 1 to 26.59% after learning

40 classes. This better performance on the first set does cause it to learn new set less well than the other procedures. It can also not hold information from previous sets that are not the first well. From these observations, we can conclude that none of these procedures in these setups are suitable for incremental learning. The Classic, FreezeO and FreezeOS+F procedures are too unstable, while the FreezeOS is relatively stable for the first set but too unstable for newer sets. In this set of setups the FreezeOS even with its' problems the least worse setup.

The results of the procedures on old and new data look much better. Almost all accuracies are better. Where most procedures were 0% on the first set after learning new classes, when old data is available they retained the information to some extent. After learning 40 classes the accuracies were 17.86% (Classic), 28.00% (FreezeO), 63.50% (FreezeOS) and 18.61% (FreezeOS+F). The FreezeOS procedure does perform better but still suffers the same problem as when only new data is available. This setup retains the information from the first class very well, 26.59% on only new data versus 63.50% when old data is available. The learning of new sets is worse than when only new information is available. On the combined sets it has only an accuracy of 18.39% where the other procedures are between 32% and 36%. It has the worst accuracy in the combined results. The other procedures do much better on the combined sets. They outperform themselves compared to when only new data is available. Their accuracies are almost double. They have accuracies of 32.91% (Classic), 35.71% (FreezeO) and 33.26% (FreezeOS+F) versus accuracies around 16-17% when only new data is available. These procedures follow a similar pattern in learning and retaining data. The Classic procedure performs worst of these three. It learns new data well but retains old data less well than FreezeO and FreezeOS+F. The FreezeOS+F performs similar to Classic but has slightly better accuracies. FreezeO performs the best of the procedures when some old data is available. It has the highest accuracy of them with 35.71% on the combined sets after learning 40 classes compared to 32.91% (Classic) and 33.26% (FreezeOS+F). It has slightly worse performance on the newest set than Classic and FreezeOS+F, but it retains old information. It has the best accuracies on the combined validation sets. In this environment, would FreezeO be the best of these learning procedures for fine-tuning.

In the boosting setups does no updating retain the first set best. After learning 40 classes on only new data, the accuracy of no updating on the combined sets is 15.85% where updating has an accuracy of 12.95%. The accuracy of no updating on the newest set in Train 3 (30 classes) and 4 (40 classes learned) is similar to updating. It does not retain sets that are not the newest or the first well. In Train 3 updating does seem to retain these middle sets better, but in Train 4 it does not do much better than no updating. It has their an accuracy of 1.10% on Test 2 and 2.11% on Test 3 where no updating has respectively 0.05% and 1.95%. Both these setups do not seem to work well for incremental learning without having access to old information.

The setups perform much better when new data is available. Both methods have in almost all cases, after learning 40 classes, about double the accuracy as when only new data is available. In the combined sets, they outperformed the only new by 35.71% (no updating) and 35.84% (updating) when old data is available versus 15.85% (no updating) and 12.95% (updating) when only new data is available. The exception is no updating on only new data on the first set. It has an accuracy of 29.53% where the both with old data available have 31.35% (no updating) and 31.93% (updating). The differences between both setups are small. Both show a similar pattern of learning new set and retaining old. Neither setup shows being better than the other. In the combined sets is the difference at the end just 0.13% after learning 40 classes. One is better after one set is added and the other is better when the next is added.

7.4 Experiments in Literature

It is hard to compare our approaches with the approaches we talked about in the literature. First of all we tested our approaches in an on-line and incremental environment. In the literature it is in many cases not clear whether the model had access to old information during the training of

new data. Also the experimental setups are different and different datasets are used. Some use small datasets like the Learn++ approach by D. Medera and S. Babinec [40] or the fine-tuning approach by C. Käding et al. [4]. Others train initially on a dataset A and then incrementally or on-line on a dataset B such as Less Forgetting learning [60] and Learning without Forgetting [59]. Because of this we cannot compare our approaches with [40], [4], [60] and [59]. However, the fine-tuning approach in [4] is similar to our fine-tuning setups. The IB-CNN by S. Han et al. [57] is an interesting approach, but the dataset is too different from ours to compare it with our approaches. It is proposed as method for facial action units so we cannot relate that to a more general case. The iCaRL approach [58] and approach by T. Xiao et al. [61] have experiments with larger datasets and the data is added gradually. [61] achieves after finishing the on-line/incremental phase of an accuracy of 51.48%. The iCaRL approach does similar well with an accuracy of 45% after learning the full ILSVRC-2012 dataset. However, these experiments were done in niche ways. This means that also these do not lend them themselves to be a reasonable comparison with our experiments.

7.5 Final Conclusion

A variety of types of approaches have been proposed for neural networks, but only a few for convolutional neural networks. In convolutional neural networks, the approaches are based on boosting or fine-tuning. We proposed three main approaches based on this research and our own ideas. From our results, we have seen that learning without access to old data is hard. None of our methods could cope well with this environment. There is no best approach from our approaches we can select. The combined CNN's methods are too stable on the first set, while the fine-tune and AdaBoost approaches seem too unstable. We have seen this same thing in our fine-tune and AdaBoost experiments. Storing some old information gives a boost to the accuracies of most setups. The combined CNN's approaches do not work well here either, but all others do work better. The fine-tune and AdaBoost methods work the best here although also these accuracies would not be good enough for real-life applications. Unfortunately, we could not compare our approaches to the approaches we found in the literature. Their experimental setup or dataset was too different from ours or their experiments were done in a niche way. However, more research is needed in the area of on-line and incremental learning in CNN's.

Chapter 8

Future Work

In the literature, we have seen that limited research has been done into incremental learning in convolutional neural networks. Much more research is needed on the whole spectrum of this area. This work added research to this area, but more could be done among others based on this work.

One thing to do could be to test more setups with the approaches form Section 4. For example, different learning settings. There might be a combination of parameters that causes the results to be a lot better. Also doing experiments on a larger scale could show different results than the ones we got. Another thing that could be researched is more variations of fine-tuning. We compared four different learning procedures, but there might be more. There are also other freezing options that could be researched. Maybe fine-tuning in an incremental setting would benefit from freezing just certain layers in the group of shared parameters.

There are also more methods to develop. One of them could be to develop a dynamic convolutional neural network. In such a CNN nodes and layers could be added, removed and, frozen dynamically depending on how the data changes. It could contain layers in parallel where one route specializes in certain data while the other specializes in different data. Also, the model could adopt ideas from other types of neural networks.

There is also more research needed in using boosting methods in combination with convolutional neural networks in an incremental environment. Currently, only a few papers touched this idea and more research in how to adapt these models to changing environments is needed. We did some research on this by comparing weights per image versus per class and by updating a CNN's weight versus not updating it.

In this work, we only used AlexNet as a CNN. The AlexNet is a simple CNN to learn from scratch and relatively fast to learn. However, more CNN's have been developed. These newer CNN's might work better than the AlexNet we used. Examples for other CNN's are VGG [77] and ResNet [72]. However, for this research area the incremental nature should be kept in mind. A CNN should not take a long time to learn or fine-tune. If it did, the environment could already be changed once it finishes. Also, a large CNN could perform better, but it does need more memory to be stored. Incremental algorithms have to deal with limited memory. Such a large CNN or multiple of them might not be able to be stored in that limited memory. Another research topic could be to design a new CNN from scratch that is designed to work in an on-line or incremental environment.

For our experiments, we used a subset of the ImageNet dataset. In other papers, the writers also designed their own dataset. This makes it hard to compare different approaches without doing the experiments yourself. A big topic in incremental learning with or without CNN's could be to develop a standardized way benchmark approaches. Idealistically this approach would be based on real-life data and consist of different sets of data to test a variety of on-line and incremental environments. We have seen that a lot of research is possible to add knowledge to this field of research. More research can be done based on approaches in this work to new methods. And also more research from using other CNN's to developing a standardized approach to benchmark incremental algorithms.

Chapter A

Full Results Comparing Approaches

In this appendix, we show the full results of the experiment where we compared all approaches. The experiments are described in Section 5.6.1 and the results in Section 6.1. The full results are shown in Table A.1 and A.2. These are the results respectively of the approaches on the new and on the old and new data.

Train	Test	FineL	FineA	CcnnP	CcnnN	AdaIW	AdaCW
Train 1	**Test 1**	68.39%	69.99%	70.67%	66.24%	71.08%	69.78%
Train 2	**Test 1**	64.86%	52.09%	55.34%	65.75%	67.8%	58.25%
	Test 2	10.64%	30.64%	33.93%	10.34%	8.26%	15.86%
	Test 1-2	37.75%	41.37%	44.63%	38.05%	38.03%	37.06%
Tain 3	**Test 1**	49.82%	16.61%	50.16%	65.34%	49.31%	42.91%
	Test 2	1.21%	1.51%	28.01%	10.95%	0.97%	4.62%
	Test 3	30.11%	43.32%	14.93%	0.67%	27.72%	24.58%
	Test 1-3	27.05%	20.48%	31.03%	25.50%	26.33%	25.31%
Train 4	**Test 1**	25.35%	1.14%	45.86%	65.91%	30.01%	24.58%
	Test 2	0.03%	0.08%	26.69%	10.45%	0.08%	1.24%
	Test 3	1.53%	1.36%	12.69%	0.71%	2.58%	2.36%
	Test 4	37.42%	40.07%	4.95%	0.22%	32.93%	32.59%
	Test 1-4	16.08%	10.66%	22.55%	19.32%	16.40%	14.25%
Train 5	**Test 1**	8.75%	0.11%	37.31%	65.64%	11.33%	10.58%
	Test 2	0%	0%	21.63%	10.65%	0%	0.03%
	Test 3	0.03%	0.01%	10.11%	0.83%	0.12%	0.11%
	Test 4	1.52%	1.45%	3.64%	0.15%	2.15%	1.24%
	Test 5	43.85%	44.71%	8.40%	0.06%	38.66%	38.04%
	Test 1-5	10.83%	8.28%	16.22%	15.46%	10.73%	10.04%
Train 6	**Test 1**	1.33%	0%	35.73%	65.85%	2.82%	1.585
	Test 2	0%	0%	18.81%	10.98%	0%	0%
	Test 3	0%	0%	10.73%	0.86%	0%	0.01%
	Test 4	0%	0.06%	3.28%	0.3%	0.05%	0.06%
	Test 5	3.06%	4.16%	7.81%	0.12%	3.55%	2.68%
	Test 6	43.47%	44.36%	1.45%	0%	37.2%	34.41%
	Test 1-6	7.98%	7.93%	12.97%	13.02%	7.27%	6.45%
Train 7	**Test 1**	0.09%	0%	38.17%	66.36%	0.83%	0.37%
	Test 2	0%	0%	15.08%	10.22%	0%	0%
	Test 3	0%	0%	9.51%	0.75%	0%	0%
	Test 4	0%	0%	3.15%	0.31%	0%	0%
	Test 5	0.16%	0.26%	7.28%	0.1%	0.35%	0.16%
	Test 6	9.01%	7.64%	1.59%	0%	5.40%	4.69%
	Test 7	41.66%	41.32%	3.31%	0%	36.09%	35.21%
	Test 1-7	7.28%	7.03%	11.76%	11.11%	6.18%	5.78%

Table A.1: Results of all approaches on only the new data

In the tables is each set of rows a training iteration. Each group has a row for each test set and the combined test set. Each column represents the accuracy results of one setup.

Train	Test	FineL	FineA	CcnnP	CcnnN	AdaIW	AdaCW
Train 1	**Test 1**	70.54	70.14	69.84%	64.74%	71.15%	71.10%
Train 2	**Test 1**	47.01%	41.22%	62.47%	64.61%	58.85%	57.99%
	Test 2	66.93%	69.03%	28.37%	11.25%	54.04%	55.76%
	Test 1-2	56.97%	55.12%	45.42%	37.93%	56.45%	56.88%
Tain 3	**Test 1**	36.04%	29.71%	59.51%	65.13%	44.19%	44.78%
	Test 2	36.59%	37.74%	27.70%	11.91%	30.97%	30.82%
	Test 3	65.34%	66.86%	8.81%	0.7%	61.11%	62.03%
	Test 1-3	45.99%	44.77%	32.01%	25.91%	45.43%	45.88%
Train 4	**Test 1**	28.90%	21.79%	57.66%	65.22%	33.68%	34.42%
	Test 2	26.18%	25.79%	27.45%	11.78%	20.53%	19.84%
	Test 3	26.36%	29.26%	8.14%	0.59%	25.80%	25.08%
	Test 4	62.65%	62.96%	4.00%	0.03%	59.39%	59.94%
	Test 1-4	36.02%	34.95%	24.31%	19.41%	34.85%	34.82%
Train 5	**Test 1**	24.36%	17.12%	55.66%	65.00%	27.04%	29.73%
	Test 2	22.05%	21.11%	26.53%	11.73%	16.935	16.06%
	Test 3	18.84%	21.56%	7.93%	0.65%	17.16%	17.26%
	Test 4	21.38%	34.89%	3.93%	0.04%	22.21%	21.79%
	Test 5	67.43%	66.94%	5.09%	0.01%	66.31%	65.91%
	Test 1-5	30.81%	30.33%	19.83%	15.49%	29.93%	30.15%
Train 6	**Test 1**	19.13%	11.88%	43.87%	64.39%	22.23%	22.79%
	Test 2	21.16%	18.94%	19.08%	11.89%	15.92%	14.94%
	Test 3	15.26%	16.18%	5.95%	0.64%	11.83%	11.82%
	Test 4	12.93%	15.33%	2.29%	0.1%	12.14%	13.08%
	Test 5	20.29%	25.48%	4.36%	0.01%	22.30%	23.19%
	Test 6	70.91%	70.09%	5.78%	0%	69.06%	68.44%
	Test 1-6	26.61%	26.32%	13.55%	12.84%	25.58%	25.71%
Train 7	**Test 1**	14.36%	8.76%	43.93%	64.95%	16.03%	16.77%
	Test 2	18.10%	16.13%	18.38%	11.37%	12.47%	12.11%
	Test 3	13.32%	14.04%	5.64%	0.65%	10.64%	10.24%
	Test 4	11.66%	11.97%	2.08%	0.04%	9.26%	9.52%
	Test 5	13.36%	16.18%	3.27%	0.02%	13.99%	13.84%
	Test 6	21.41%	26.01%	5.34%	0%	24.36%	24.16%
	Test 7	68.71%	68.21%	2.64%	0%	67.21%	67.41%
	Test 1-7	22.97%	23.02%	11.61%	11.00%	21.99%	22.01%

Table A.2: Results of all approaches on the new and old data.

Chapter B

Tables Comparing Approaches

The tables in this chapter show the results used in the Figures 6.1 and 6.2 in Chapter 6.1. The captions below each table refer to which plot the table is related to.

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	68.39%	64.86%	49.82%	25.35%	8.7%	1.33%	0.09%
FineA	69.66%	52.09%	16.61%	1.14%	0.11%	0%	0%
CcnnP	70.67%	55.34%	50.16%	46.86%	37.31%	35.73%	38.17%
CcnnN	66.24%	65.75%	65.34%	65.91%	65.64%	65.85%	66.36%
AdaI	71.08%	67.80%	49.31%	30.01%	11.33%	2.82%	0.83%
AdaCW	69.78%	58.25%	42.91%	24.58v	10.58%	1.59%	0.37%

Table B.1: Data of plot Set 1 on only new data in Figure 6.1 (a).

	Number of added classes				
Approach	**30**	**40**	**50**	**60**	**70**
FineL	1.21%	0.795%	0.516%	0.765%	1.834%
FineA	1.51%	0.72%	0.486%	1.06%	1.504%
CcnnP	28.01%	19.69%	11.79%	10.16%	7.32%
CcnnN	10.95%	5.81%	2.91%	3.07%	2.28%
AdaI	0.97%	1.32%	0.756%	0.90%	1.15%
AdaCW	4.62%	1.80%	0.46%	0.69%	0.97%

Table B.2: Data of plot Previous sets except first set on only new data in Figure 6.1 (b).

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	68.39%	10.64%	30.11%	37.42%	43.85%	43.47%	41.66%
FineA	69.66%	30.64%	43.32%	40.07%	44.71%	44.36%	41.66%
CcnnP	70.67%	33.93%	14.93%	4.95%	8.40%	1.45%	3.31%
CcnnN	66.24%	10.34%	0.67%	0.22%	0.06%	0%	0%
AdaIW	71.08%	8.26%	27.72%	32.93%	38.66%	37.2%	36.09%
AdaCW	69.78%	15.86%	24.58%	32.59%	38.04%	24.41%	35.21%

Table B.3: Data of plot Newest set on only new data in Figure 6.1 (c).

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	68.39%	37.75%	27.05%	16.08%	10.83%	7.98%	7.289%
FineA	69.66%	41.37%	20.48%	10.66%	8.28%	7.93%	7.03%
CcnnP	70.67%	44.63%	31.03%	22.55%	16.22%	12.97%	11.76%
CcnnN	66.24%	38.05%	25.50%	19.32%	15.46%	13.02%	11.11%
AdaIW	71.08%	38.02%	26.33%	16.40%	10.73%	7.27%	6.45%
AdaCW	69.78%	37.06%	25.31%	14.25%	10.04%	6.45%	5.78%

Table B.4: Data of plot Combined sets on only new data in Figure 6.1 (d) of the comparing approaches experiment.

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	70.54%	47.01%	36.04%	28.90%	24.36%	19.13%	14.36%
FineA	70.54%	41.22%	29.71%	21.79%	17.12%	11.88%	8.79%
CcnnP	69.84%	62.42%	59.51%	57.66%	55.66%	43.87%	43.87%
CcnnN	64.74%	64.61%	65.13%	65.00%	64.39%	64.39%	64.95%
AdaIW	71.15%	58.85%	44.19%	33.68%	27.04%	22.23%	16.03%
AdaCW	69.78%	57.99%	44.78%	34.42%	29.73%	22.79%	16.77%

Table B.5: Data of plot Set 1 on new and old data in Figure 6.2 (a) of the comparing approaches experiment.

	Number of added classes				
Approach	**30**	**40**	**50**	**60**	**70**
FineL	36.59%	26.27%	20.76%	17.41%	15.57%
FineA	7.74%	27.53%	25.85%	18.98%	16.87%
CcnnP	27.7%	17.795%	12.796%	7.92%	6.942%
CcnnN	11.25%	6.185%	3.945%	3.16%	2.416%
AdaIW	0.97%	23.165%	18.768%	15.55%	14.144%
AdaCW	0.82%	22.46%	18.37%	15.76%	13.974%

Table B.6: Data of plot Previous sets except first set on new and old data in Figure 6.2 (b) of the comparing approaches experiment.

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	70.54%	66.93%	65.34%	62.65%	67.43%	70.91%	68.71%
FineA	70.14%	69.03%	62.96%	66.94%	66.94%	70.09%	68.21%
CcnnP	69.84%	28.37%	8.81%	4.00%	5.09%	5.78%	2.64%
CcnnN	64.74%	11.25%	0.7%	0.03%	0.01%	0%	0%
AdaIW	71.15%	54.04%	61.11%	59.39%	66.31%	69.06%	67.21%
AdaCW	71.10%	55.76%	62.03%	59.94%	65.91%	68.44%	67.41%

Table B.7: Data of plot Newest set on new and old data in Figure 6.2 (c) of the comparing approaches experiment.

	Number of added classes						
Approach	**10**	**20**	**30**	**40**	**50**	**60**	**70**
FineL	70.54%	56.97%	45.99%	36.02%	30.81%	26.61%	22.97%
FineA	70.14%	55.12%	44.77%	34.95%	30.33%	26.32%	23.03%
CcnnP	69.84%	45.42%	32.01%	24.31%	19.83%	13.55%	11.61%
CcnnN	64.74%	37.93%	25.91%	19.41%	15.49%	12.84%	11.00%
AdaIW	71.15%	65.45%	45.43%	34.85%	29.93%	25.58%	21.99%
AdaCW	71.10%	56.88%	45.88%	34.82%	30.15%	25.71%	22.01%

Table B.8: Data of plot Combined sets on new and old data in Figure 6.2 (d) of the comparing approaches experiment.

Chapter C

Tables Comparing Fine-Tuning Setups Figures

The tables in this chapter show the results used in the Figures 6.3 and 6.4 in Chapter 6.2. The captions below each table refer to which plot the table is related to.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.24%	0%	0%	0%
FreezeO	66.55%	1.04%	0.01%	0%
FreezeOS	63.67%	61.62%	46.49%	26.59%
FreezeOS+F	66.23%	0%	0%	0%

Table C.1: Data of plot Set 1 on only new data in Figure 6.3 (a) of the comparing fine-tune setups experiment.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.24%	73.23%	73.60%	67.45%
FreezeO	66.55%	73.37%	73.13%	66.88%
FreezeOS	63.67%	10.69%	32.40%	35.97%
FreezeOS+F	66.23%	73.22%	74.09%	67.40%

Table C.2: Data of plot newest set on only new data in Figure 6.3 (b) of the comparing fine-tune setups experiment.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.24%	36.62%	24.53%	16.94%
FreezeO	66.55%	37.21%	24.50%	16.86%
FreezeOS	63.67%	36.15%	26.95%	16.25%
FreezeOS+F	66.23%	36.61%	24.71%	16.85%

Table C.3: Data of plot Combined sets on only new data in Figure 6.3 (c) of the comparing fine-tune setups experiment.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.53%	35.44%	22.20%	17.86%
FreezeO	65.01%	44.83%	32.67%	28.00%
FreezeOS	65.48%	64.90%	63.78%	63.50%
FreezeOS+F	65.07%	35.34%	24.03%	18.61%

Table C.4: Data of plot Set 1 on new and old data in Figure 6.4 (a) of the comparing fine-tune setups experiment.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.53%	70.04%	69.24%	64.28%
FreezeO	65.01%	65.23%	66.11%	60.91%
FreezeOS	65.48 %	3.35%	11.01%	4.44%
FreezeOS+F	65.07%	70.40%	70.94%	64.63%

Table C.5: Data of plot newest set on new and old data in Figure 6.4 (b) of the comparing fine-tune setups experiment.

	Number of added classes			
Procedure	**10**	**20**	**30**	**40**
Classic	65.53%	52.80%	39.81%	32.91%
FreezeO	65.01%	56.31%	45.22%	35.71%
FreezeOS	65.48%	34.13%	23.60%	18.39%
FreezeOS+F	65.07%	52.87%	41.83%	33.26%

Table C.6: Data of plot Combined sets on new and old data in Figure 6.4 (c) of the comparing fine-tune setups experiment.

Chapter D

Tables Adaboost Setups Figures

The tables in this chapter show the results used in the Figure 6.5 in Chapter 6.3. The captions below each table refer to which plot the table is related to.

	Number of added classes			
Method	**10**	**20**	**30**	**40**
No updating (new)	66.08%	62.44%	43.41%	29.53%
Updating (new)	67.94%	0.36%	15.95%	16.48%
No update (old)	66.30%	55.17%	38.11%	31.35%
Updating (old)	67.01%	54.84%	39.38%	31.93%

Table D.1: Data of plot Set 1 on only new and new and old data in Figure 6.5 (a) of the AdaBoost comparing updating vs. not updating experiment.

	Number of added classes			
Method	**10**	**20**	**30**	**40**
No updating (new)	66.08%	9.46%	30.51%	31.87%
Updating (new)	67.94%	59.75%	31.78%	31.01%
No update (old)	66.30%	53.86%	63.73%	60.39%
Updating (old)	67.01%	53.81%	63.43%	60.32%

Table D.2: Data of plot Newest set on only new and new and old data in Figure 6.5 (b) of the AdaBoost comparing updating vs. not updating experiment.

	Number of added classes			
Method	**10**	**20**	**30**	**40**
No updating (new)	66.08%	35.95%	24.66%	15.85%
Updating (new)	67.94%	30.06%	20.40%	12.95%
No update (old)	66.30%	55.73%	43.88%	35.71%
Updating (old)	67.01%	54.33%	44.11%	35.84%

Table D.3: Data of plot Combined sets on only new and new and old data in Figure 6.5 (c) of the AdaBoost comparing updating vs. not updating experiment.

Bibliography

[1] Alexander Gepperth and Barbara Hammer. Incremental learning algorithms and applications. In *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[2] Christophe Giraud-Carrier. A note on the utility of incremental learning. *Ai Communications*, 13(4):215–223, 2000.

[3] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in neural information processing systems*, pages 640–646, 1996.

[4] Christoph Käding, Erik Rodner, Alexander Freytag, and Joachim Denzler. Fine-tuning deep neural networks in continuous learning scenarios. In *ACCV Workshop on Interpretation and Visualization of Deep Neural Nets (ACCV-WS)*. Springer International Publishing, 2016.

[5] Haibo He, Sheng Chen, Kang Li, and Xin Xu. Incremental learning from stream data. *IEEE Transactions on Neural Networks*, 22(12):1901–1914, 2011.

[6] RR Ade and PR Deshmukh. Methods for incremental learning : A survey. *International Journal of Data Mining & Knowledge Management Process*, 3(4):119–125, jul 2013.

[7] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37, mar 2014.

[8] Robi Polikar, Lalita Udpa, Satish S Udpa, and Vasant Honavar. Learn++: an incremental learning algorithm for multilayer perceptron networks. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, volume 6, pages 3414–3417. Institute of Electrical and Electronics Engineers (IEEE), 2000.

[9] Martial Mermillod, Aurélia Bugaiska, and Patrick Bonin. The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. *Frontiers in Psychology*, 4:504, 2013.

[10] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

[11] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[12] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[13] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[14] Paul John Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974.

[15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[16] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237. Barcelona, Spain, 2011.

[17] Alessandro Giusti, Dan C Ciresan, Jonathan Masci, Luca M Gambardella, and Jurgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 4034–4038. IEEE, sep 2013.

[18] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks – ICANN 2010*, pages 92–101. Springer Berlin Heidelberg, 2010.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[20] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer Berlin Heidelberg, 1995.

[21] Stephen Grossberg. Adaptive pattern classification and universal recoding: I. parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23(3):121–134, 1976.

[22] Stephen Grossberg. *Adaptive resonance theory*. Wiley Online Library, 2003.

[23] Stephen Grossberg. Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world. *Neural Networks*, 37:1–47, jan 2013.

[24] Gail A Carpenter and Stephen Grossberg. A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, 37(1):54–115, jan 1987.

[25] Gail A. Carpenter and Stephen Grossberg. ART 2: self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26(23):4919–4930, Dec 1987.

[26] Gail A Carpenter, Stephen Grossberg, and David Rosen. ART 2-a: An adaptive resonance algorithm for rapid category learning and recognition. *Neural Networks*, 4(4):493–504, jan 1991.

[27] Gail A Carpenter, Stephen Grossberg, and John H Reynolds. ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural Networks*, 4(5):565–588, jan 1991.

[28] Gail A Carpenter, Stephen Grossberg, Natalya Markuzon, John H Reynolds, and David B Rosen. Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3(5):698–713, 1992.

[29] Lotfi A Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[30] Gail A Carpenter, Marin N Gjaja, Sucharita Gopal, and Curtis E Woodcock. ART neural networks for remote sensing: vegetation classification from landsat TM and terrain data. *IEEE Transactions on Geoscience and Remote Sensing*, 35(2):308–325, mar 1997.

[31] Gail A Carpenter, Stephen Grossberg, and David B Rosen. Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, 4(6):759–771, jan 1991.

[32] Gail A Carpenter, Stephen Grossberg, and David B Rosen. A neural network realization of fuzzy art. Technical report, Boston University Center for Adaptive Systems and Department of Cognitive and Neural Systems, 1991.

[33] James R Williamson. Gaussian ARTMAP: A neural network for fast incremental learning of noisy multidimensional maps. *Neural Networks*, 9(5):881–897, jul 1996.

[34] Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. Learn++: an incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 31(4):497–508, 2001.

[35] Michael Muhlbaier, Apostolos Topalis, and Robi Polikar. Learn++.MT: A new approach to incremental learning. In *Multiple Classifier Systems*, pages 52–61. Springer, Springer Berlin Heidelberg, 2004.

[36] Michael D Muhlbaier, Apostolos Topalis, and Robi Polikar. Learn++.nc: Combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes. *IEEE transactions on neural networks*, 20(1):152–168, 2009.

[37] Gregory Ditzler, Michael D Muhlbaier, and Robi Polikar. Incremental learning of new classes in unbalanced datasets: Learn++.UDNC. In *Multiple Classifier Systems*, pages 33–42. Springer Berlin Heidelberg, 2010.

[38] Michael Muhlbaier, Apostolos Topalis, and Robi Polikar. Incremental learning from unbalanced data. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 2, pages 1057–1062. IEEE, 2004.

[39] Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22(10):1517–1531, 2011.

[40] Dusan Medera and Stefan Babinec. Incremental learning of convolutional neural networks. In *Proceedings of the International Joint Conference on Computational Intelligence*, pages 547–550. SciTePress - Science and and Technology Publications, 2009.

[41] David J Willshaw and Christoph Von Der Malsburg. How patterned neural connections can be set up by self-organization. *Proceedings of the Royal Society B: Biological Sciences*, 194(1117):431–445, nov 1976.

[42] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[43] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[44] Bernd Fritzke. Growing cell structures—a self-organizing network for unsupervised and supervised learning. *Neural networks*, 7(9):1441–1460, jan 1994.

[45] Shen Furao and Osamu Hasegawa. An on-line learning mechanism for unsupervised classification and topology representation. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1. Institute of Electrical and Electronics Engineers (IEEE), 2005.

[46] Shen Furao, Tomotaka Ogura, and Osamu Hasegawa. An enhanced self-organizing incremental neural network for online unsupervised learning. *Neural Networks*, 20(8):893–903, oct 2007.

[47] Furao Shen and Osamu Hasegawa. A fast nearest neighbor classifier based on self-organizing incremental neural network. *Neural Networks*, 21(10):1537–1547, dec 2008.

[48] Furao Shen, Hui Yu, Keisuke Sakurai, and Osamu Hasegawa. An incremental online semi-supervised active learning algorithm based on self-organizing incremental neural network. *Neural Computing and Applications*, 20(7):1061–1074, aug 2010.

[49] Martin D Buhmann. Radial basis functions. *Acta Numerica 2000*, 9:1–38, 2000.

[50] David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document, 1988.

[51] Lorenzo Bruzzone and D Fernandez Prieto. An incremental-learning neural network for the classification of remote-sensing images. *Pattern Recognition Letters*, 20(11-13):1241–1248, nov 1999.

[52] Seiichi Ozawa, Soon Lee Toh, Shigeo Abe, Shaoning Pang, and Nikola Kasabov. Incremental learning for online face recognition. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 5, pages 3174–3179. IEEE, 2005.

[53] Bernard Ans and Stéphane Rousset. Avoiding catastrophic forgetting by coupling two reverberating neural networks. *Comptes Rendus de l'Académie des Sciences - Series III - Sciences de la Vie*, 320(12):989–997, dec 1997.

[54] Husam Al-Behadili, Arne Grumpe, Christian Dopp, and Christian Wöhler. Extreme learning machine based novelty detection for incremental semi-supervised learning. In *2015 Third International Conference on Image Information Processing (ICIIP)*, page 230235. Institute of Electrical and Electronics Engineers (IEEE), dec 2015.

[55] Vincenzo Lomonaco and Davide Maltoni. Comparing incremental learning strategies for convolutional neural networks. In *Artificial Neural Networks in Pattern Recognition*, pages 175–184. Springer International Publishing, 2016.

[56] Robert E Schapire. Using output codes to boost multiclass learning problems. In *ICML*, volume 97, pages 313–321, 1997.

[57] Shizhong Han, Zibo Meng, AHMED-SHEHAB KHAN, and Yan Tong. Incremental boosting convolutional neural network for facial action unit recognition. In *Advances in Neural Information Processing Systems*, pages 109–117, 2016.

[58] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, and Christoph H Lampert. icarl: Incremental classifier and representation learning. *CoRR*, abs/1611.07725, 2016.

[59] Zhizhong Li and Derek Hoiem. Learning without forgetting. In *Computer Vision – ECCV 2016*, pages 614–629. Springer International Publishing, 2016.

[60] Heechul Jung, Jeongwoo Ju, Minju Jung, and Junmo Kim. Less-forgetting learning in deep neural networks. *arXiv preprint arXiv:1607.00122*, 2016.

[61] Tianjun Xiao, Jiaxing Zhang, Kuiyuan Yang, Yuxin Peng, and Zheng Zhang. Error-driven incremental learning in deep convolutional neural network for large-scale image classification. In *Proceedings of the ACM International Conference on Multimedia - MM '14*, pages 177–186. Association for Computing Machinery (ACM), 2014.

[62] Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[63] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, ACM Press, 2014.

[64] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[65] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, Springer International Publishing, 2014.

[66] Bangpeng Yao, Xiaoye Jiang, Aditya Khosla, Andy Lai Lin, Leonidas Guibas, and Li Fei-Fei. Human action recognition by learning bases of action attributes and parts. In *2011 International Conference on Computer Vision*, pages 1331–1338. IEEE, 2011.

[67] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. Places: An image database for deep scene understanding. *CoRR*, abs/1610.02055, 2016.

[68] Mark Everingham, SM Ali Eslami, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111(1):98–136, 2015.

[69] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The caltech-ucsd birds-200-2011 dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[70] Ariadna Quattoni and Antonio Torralba. Recognizing indoor scenes. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 413–420. IEEE, 2009.

[71] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[72] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[73] Ji Zhu, Saharon Rosset, Hui Zou, and Trevor Hastie. Multi-class adaboost. *Ann Arbor*, 1001(48109):1612, 2006.

[74] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class AdaBoost. *Statistics and Its Interface*, 2(3):349–360, 2009.

[75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[76] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *British Machine Vision Conference*, 2014.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.