Karlsruhe Institute of Technology

Sample-Incremental Meta-Learning: Weight-Mapping with Deep Learning

Master's Thesis of

Christian Huber

at the Interactive Systems Lab
Institute for Anthropomatics and Robotics
Karlsruhe Institute of Technology (KIT)

Reviewer:	Prof. Dr. Alex Waibel
Second reviewer:	PD Dr. Gudrun Thäter
Advisor:	M.Sc. Juan Hussain

26. November 2019 – 25. May 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, May 25, 2020

. . . . . .

(Christian Huber)

Abstract

We propose a new incremental learning scheme which is based on a meta-learning approach. The knowledge of the system is stored in the weights of a base-model and a mapping-model is learned to adjust these weights when new data is available to improve the performance of the system. Different methods of training the system are compared and analyzed. During inference the system requires only little data. For a linear base-model this approach outperforms fine-tuning and it also works for more complex base-models.

Zusammenfassung

Wir stellen ein neues inkrementelles Lernschema vor, das auf einem Meta-Learning-Ansatz basiert. Das Wissen über das System wird in den Gewichten eines Basis-Modells gespeichert und ein Mapping-Modell wird gelernt, um diese Gewichte anzupassen, wenn neue Daten verfügbar sind, um die Leistung des Systems zu verbessern. Verschiedene Methoden zum Training des Systems werden verglichen und analysiert. Während der Inferenz benötigt das System nur wenige Daten. Bei einem linearen Basis-Modell übertrifft dieser Ansatz Fine-tuning, und er funktioniert auch bei komplexeren Basis-Modellen.

Contents

1. Introduction

1.1. Motivation

In the last several decades computers have become more and more powerful due to the increase in transistor density. This is called Moore's law – rather an observation than a physical law. Furthermore, concepts and faster algorithms have been developed increasing the tasks a computer can solve. Such a concept are neural networks. Although first approaches go back to the 1940s, in the last decade they have proven to be powerful when combined with large datasets and a huge amount of compute.

However, in some aspects neural networks are not yet on a human level of performance. Two of these aspects are self-assessment and self-description. Humans are able to evaluate their performance and know if they do not know an answer to a given question are able to articulate this. Socrates put this in the phrase "I know that i know nothing". Furthermore, humans are able to efficiently learn in an incremental manner with little data given, and to selectively forget things, e.g. if one is told and explained that some knowledge one possesses is wrong.

This thesis focuses on incremental learning aspect with little given data.

1.2. Goals

The goal of this thesis is to develop a system for image classification. This system should be able to learn from little data during inference, e.g. at a certain timestep one image per class could be given and the system should use this data to improve the performance of the system. In other words, the system should be able to incorporate new knowledge in an incremental manner. To implement such a system, neural networks are used, as they provide state-of-the-art performance for image classification. Furthermore the incremental learning should be achieved by some mechanism which does not just save all seen data and retrain a neural network with that data.

1.3. Outline

In the next chapter, the theory used in this work is explained, i.e. concepts of neural networks and encoder-decoder models. In chapter three, the work related to this thesis is described, i.e. incremental learning. Furthermore, an introduction to meta-learning is given. Chapter four specifies the sample-incremental meta-learning method of this work. This includes the training and the inference of the system. In Chapter five the results of the experiments are shown. Chapter six contains a review of the work and an outlook to future work. In the appendix all results are given in detail.

2. Theory

In this chapter the methods used in this work are explained. This includes the development of artificial neural networks, the application of neural networks to image classification and sequence-to-sequence models.

2.1. Neural networks

Neural networks, or more precise artificial neural networks, are inspired by biology. The intention was to model the neurons and synapses of the brain with the goal to reproduce the functionality. The question was how the brain retrieves, processes and stores information. Furthermore the model should be able to learn and adapt to new situations.

The first approaches describing this were published in the 1940s. McCulloch and Pitts proposed a mathematical model of the human brain [MP43]. In their model the inputs and the output of nerves are binary, resulting in binary logic.

2.1.1. Perceptron

In 1958, Rosenblatt introduced the so called perceptron [Ros58]. For an input $x \in \mathbb{R}^n$, $n \in \mathbb{N}$, $w \in \mathbb{R}^n$, $b \in \mathbb{R}$ and a function $\varphi : \mathbb{R} \to \mathbb{R}$ the perceptron (see figure 2.1) computes the output $y \in \mathbb{R}$ given by

$$y \coloneqq \varphi(w^T x + b).$$

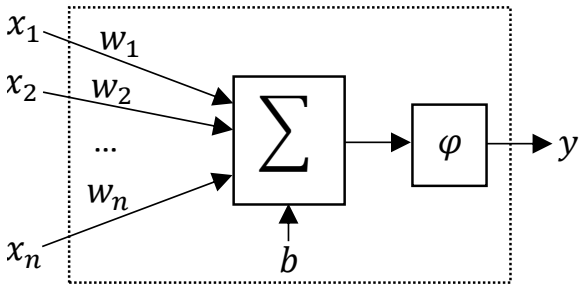

Figure 2.1.: Perceptron

In this context n is the number of inputs to the perceptron, the elements of w are called weights, b bias and φ activation function.

For a more compact notation, the bias b can be omitted by increasing the length of w and x by one and setting $x_0 := 1$. Furthermore when using multiple perceptrons, say $m \in \mathbb{N}$, one can describe this by a weight matrix $W \in \mathbb{R}^{m\times(n+1)}$. The outputs $y \in \mathbb{R}^m$ of the perceptrons are then computed by

$$y := \varphi(Wx), \tag{2.1}$$

where the activation function is applied pointwise. The transformation in equation (2.1) is called layer. In sections 2.2 and 2.4.3 more advanced layers are introduced.

2.1.2. Activation functions

Common activation functions are

$$\varphi_1(x) := \begin{cases} 0, & \text{for } x < 0, \\ 1, & \text{for } x \geq 0, \end{cases} \qquad \text{Heaviside step function,}$$

$$\varphi_2(x) := \frac{1}{1+e^{-x}}, \ x \in \mathbb{R}, \qquad \text{Sigmoid function,}$$

$$\varphi_3(x) := \begin{cases} 0, & \text{for } x < 0, \\ x, & \text{for } x \geq 0, \end{cases} \qquad \text{Rectified linear unit (ReLU).}$$

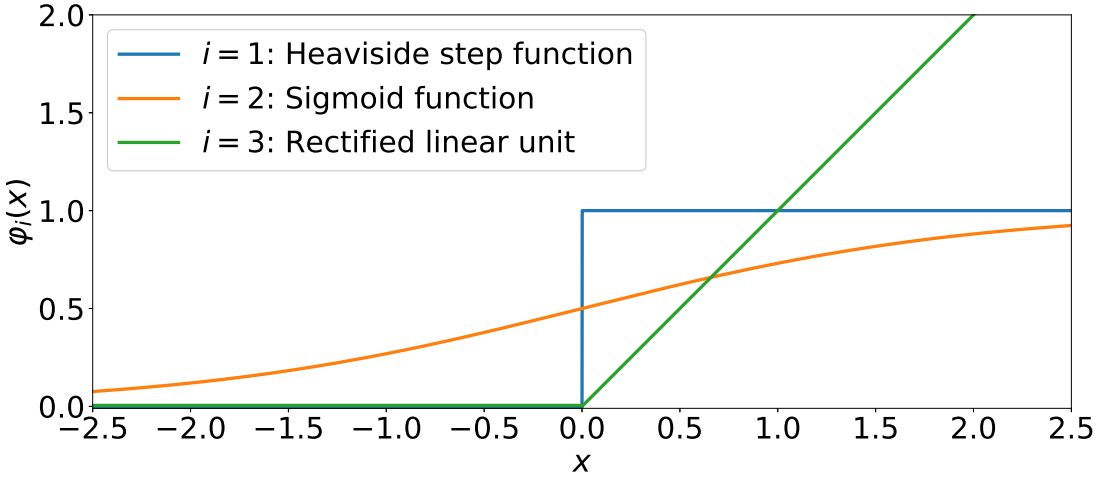

Figure 2.2.: Activation functions

By using the Heaviside step function the perceptron only activates, i.e. it does not output zero, if the (weighted) input is above some threshold. The ReLU function is similar, but if the perceptron is activated the output also depends on the (weighted) input. The sigmoid function maps the input smoothly between zero and one. A further comparison between these activation functions is discussed later.

2.1.3. Multi-layer perceptron

The perceptron can be used to separate data points. This is done by applying it and distinguish between values larger or equal and less than zero. A problem that emerges by using a perceptron with Heaviside step activation function (or any other monotonic activation function) is that it is only capable of separating linearly separable data, i.e. if there exists a hyperplane where one part of the data lies on one side and the other one on the other side. For instance the XOR-data (table 2.1) can not be separated by such a perceptron.

x	y	$x \bigoplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1.: XOR data

To overcome this issue, a so-called feedforward neural network can be used, also called multi-layer perceptron (MLP). The output of perceptrons is used as input for other perceptrons. This results in deeper networks, i.e. networks with more than one layer. For instance the model defined by equation (2.1) has one layer and the model defined by equation (2.3) has two layers.

2.1.4. Loss functions

The function represented by a MLP depends on the weights of the MLP. These weights can be adjusted such that the perceptron approximates a given function or given data. Dependent on the task which should be learned, a so-called loss function is chosen, which is minimized during training. The process of adjusting the weights by minimizing the loss function is called training. The loss function is chosen dependent on the task. For given inputs $x_1, \dots, x_k \in \mathbb{R}^n$, $n, k \in \mathbb{N}$, corresponding labels $y_1, \dots, y_k \in \mathbb{R}^m$, $m \in \mathbb{N}$ and MLP f (with output in $\mathbb{R}^m$) the most common loss functions are:

$$
\begin{aligned}
l_{\text{regression},y}(x) &:= \sum_{j=1}^{m} \frac{1}{k} \sum_{i=1}^{k} (f(x_i) - y_i)_j^2 && \text{Mean squared error (regression task),} \\
l_{\text{classification},y}(x) &:= -\sum_{i=1}^{k} \sum_{j=1}^{m} (y_i)_j \log(f(x_i)_j) && \text{Cross entropy (classification task),}
\end{aligned}
\tag{2.2}
$$

For $m = 1$ and $y_i = e_{l_i}$, i.e. if the labels are unit vectors, these expression simplify to

$$l_{\text{regression},y}(x) = \frac{1}{k}\sum_{i=1}^{k}(f(x_i) - y_i)^2,$$

$$l_{\text{classification},y}(x) = -\sum_{i=1}^{k}\log(f(x_i)_{l_i}).$$

In the case of the classification task y_i is a probability distribution over the m classes, $i \in \{1, \dots, k\}$, (for instance the unit vector corresponding to the class label) and f has a softmax activation function at the final layer, i.e.

$$\varphi_{\text{softmax}}(y)_i := \frac{e^{y_i}}{\sum_{j=1}^{m} e^{y_j}}, \qquad y \in \mathbb{R}^m, i \in \{1, \dots, m\},$$

as a final activation function, thus the cross entropy loss is well defined.

2.1.5. Backpropagation algorithm

To train MLPs efficiently, the backpropagation algorithm [RHW86] was proposed. If a MLP is composed of differentiable functions, i.e. it uses only differentiable activation functions, the backpropagation algorithm can be used to calculate the derivative of the loss function w.r.t. each weight. Instead of calculating the derivatives numerically which would suffer from numerical instability and a high computational complexity, the backpropagation algorithm computes the derivatives analytically by iteratively applying the chain rule.

Let us illustrate this algorithm for a two-layer perceptron f and loss function l

$$\begin{aligned} f(x) &:= z_3 := \varphi_1(W_1\varphi_2(W_2x + b_2) + b_1), \\ l(x) &:= (f(x) - y)^2 = (z_3 - y)^2, \end{aligned} \tag{2.3}$$

for $x \in \mathbb{R}^n$, $n \in \mathbb{N}$, $W_1 \in \mathbb{R}^{1\times m}$, $m \in \mathbb{N}$, $W_2 \in \mathbb{R}^{m\times n}$, $b_1 \in \mathbb{R}$, $b_2 \in \mathbb{R}^m$, $y \in \mathbb{R}$ and activation functions $\varphi_1, \varphi_2 : \mathbb{R} \to \mathbb{R}$ for a regression task. Defining $z_0 := W_2x + b_2$, $z_1 := \varphi_2(z_0)$, $z_2 := W_1z_1 + b_1$, we get by the chain rule for instance

$$\begin{aligned} \frac{\partial l}{\partial z_3} &= 2(z_3 - y), \\ \frac{\partial l}{\partial z_2} &= \frac{\partial l}{\partial z_3}\frac{\partial z_3}{\partial z_2} = \frac{\partial l}{\partial z_3} \cdot \varphi_1'(z_2), \\ \frac{\partial l}{\partial z_1} &= \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial z_1} = W_1^T\frac{\partial l}{\partial z_2}, \\ \frac{\partial l}{\partial z_0} &= \frac{\partial l}{\partial z_1}\frac{\partial z_1}{\partial z_0} = \frac{\partial l}{\partial z_1} \cdot \varphi_2'(z_0), \\ \Rightarrow \frac{\partial l}{\partial W_2} &= \frac{\partial l}{\partial z_0}\frac{\partial z_0}{\partial W_2} = \frac{\partial l}{\partial z_0}x^T = 2W_1^T(z_3 - y) \cdot \varphi_1'(z_2) \cdot \varphi_2'(z_0)x^T, \end{aligned}$$

where $\cdot$ denotes the pointwise multiplication. In the same way the derivatives w.r.t. the other weights can be computed.

We obtain, that the Heaviside step function does not fulfill the requirement of being differentiable and is therefore not used in combination with backpropagation. Although the ReLU activation function is not differentiable at $x = 0$, it is used together with the backpropagation algorithm. The derivative at $x = 0$ is set to zero.

With the calculated derivative of the loss function w.r.t. each weight we are able to minimize the loss function in an iterative way starting with random initial weights w_0:

$$w_{n+1} := w_n - \eta \nabla l(w_n), \qquad n \in \mathbb{N},$$

where $\eta > 0$ is called learning rate, w_n are the weights of the MLP after training n steps, $n \in \mathbb{N}$, and l the loss function. This procedure is called gradient descent. Furthermore, more advanced methods have been developed to increase the speed of the iteration process, e.g. stochastic gradient descent, where the gradient is approximated over a small random part of the training data, and Adam [KB14].

2.1.6. Weight initialization

As stated in the last section the initial weights are drawn randomly from some distribution. The scaling of these weights is explained in this section.

Let $x = (x_i)_i \in \mathbb{R}^n$, $n \in \mathbb{N}$, be an input for a layer with matrix $W = (w_{i,j})_{i,j} \in \mathbb{R}^{m \times n}$, $m \in \mathbb{N}$. Furthermore let $x_i, w_{i,j}$ be independent with mean zero, all x_i have variance one and the variance of all $w_{i,j}$ be σ^2. Then one can calculate that the expectation value of $(Wx)_i$ is zero and

$$\mathbb{V}\left((Wx)_i\right) = \sum_{j=1}^{n} \mathbb{V}(w_{i,j}) = n\sigma^2.$$

To prevent exploding or vanishing outputs after the application of W one wants $(Wx)_i$ to have variance one. The equation

$$\sigma^2 = \frac{1}{n}. \tag{2.4}$$

follows. When backpropagating through the matrix multiplication with W one multiplies with W^T. Therefore one wants a similar equation to hold:

$$\sigma^2 = \frac{1}{m}.$$

These constraints are possible only if $n = m$. A compromise is $\sigma^2 = \frac{2}{n+m}$ which leads to the following weight initialization:

$$w_{i,j} \sim U\left[-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right], \qquad \text{Glorot initialization [GB10].}$$

When using ReLU activation functions it can be better to use $\sigma^2 = \frac{2}{n}$ or $\sigma^2 = \frac{2}{m}$. This initialization scheme is called He initialization [He+15].

2.1.7. Generalization

When we deploy a model, we want it to generalize well, i.e. have a good performance on unseen data. To achieve this and prevent the model from just memorizing the data (called overfitting), the available data is split into (at least) two sets: A training set whose data is used to adapt the weights of the model and a validation set whose data is used to measure the performance on data which is not used to learn the weights. The goal is to minimize the validation loss.

Another method to prevent a model from overfitting is regularization. There are numerous variants of regularization. A frequently used one is Dropout [Sri+14]. During training time each neuron is randomly set to zero with probability $p \in (0, 1)$, while during test time the weights are multiplied by p. The idea behind setting some neurons randomly to zero is that the model can not focus on specific neurons to calculate the output and therefore should generalize better.

Another important topic is the available data. The more data is used for training the better the data distribution is approximated resulting in less overfitting. Therefore the model is able to perform better and/or a model with more parameters can be trained. Since the data is limited, the goal is to find an architecture which does not have too many parameters but is able to approximate the given data.

2.1.8. Deeper networks

In the last years neural networks with more and more layers were used. This seems counterintuitive since it is well known that a MLP with only one hidden layer, i.e. a MLP with two layers, can approximate any continuous function on a compact set arbitrarily well. However, [Tel16] suggests that deeper models (w.r.t. the number of layers) can be better than having more neurons in each layer. Their statement is that for each $k \in \mathbb{N}$ "there exist neural networks with $\Theta(k^3)$ layers, $\Theta(1)$ nodes per layer, and $\Theta(1)$ distinct parameters which can not be approximated by networks with $O(k)$ layers unless they are

exponentially large — they must possess $\Omega(2^k)$ nodes"[Tel16]. Therefore it can be better to make models deeper instead of increasing the number of nodes per layer.

2.2. Image classification

An image can be represented as $I \in \mathbb{R}^{n \times m \times 3}$, $n, m \in \mathbb{N}$, where $I_{i,j}$ is the RGB-value of the pixel (i, j), $i \in \{1, \dots, n\}$, $j \in \{1, \dots, m\}$. In image classification the task is to classify a given image into a given (finite) set of classes.

The performance of an image classification model is measured by accuracy, i.e. the percentage of correctly classified images. For the top-n accuracy an image is considered correctly classified if the class label of the image is within the n most likely classes of the probability distribution output of the model. The error rate is defined by one minus the accuracy.

2.2.1. Convolution layers and pooling layers

Major progress in image classification was made in the last decade. In 2012 [KSH12] published a new architecture. They used a deep network of convolution and pooling layers and the ReLU activation function. Using these advanced architectures results in a reduction of learnable parameters compared to a perceptron and the model can be trained faster.

Convolution layers are the two-dimensional pendent to time-delay neural networks (TDNNs) [Wai+89] which operate on one-dimensional inputs. Both methods are based on the same idea to deduce shift-invariant features through local filters exploiting the structure of the input. For an input image $I \in \mathbb{R}^{n \times m}$, $n, m \in \mathbb{N}$, and a kernel $ker \in \mathbb{R}^{k \times k}$ of weights, $k \in \mathbb{N}$ odd, the output of the convolution layer (without padding) is defined by

$$(I * ker)(x, y) := \sum_{i=1}^{k} \sum_{j=1}^{k} I(x + k - i, y + k - j) ker(i, j),$$

$$x \in \{1, \dots, n - k + 1\}, y \in \{1, \dots, m - k + 1\}.$$

This formula can be generalized for instance for inputs with more channels, i.e. for inputs $I \in \mathbb{R}^{n \times m \times c}$, $c \in \mathbb{N}$, for padding $p \in \mathbb{N}$, i.e. the input of the layer is padded at all borders with p columns/rows of values, e.g. zeros, to obtain an input $I \in \mathbb{R}^{(n+2p) \times (m+2p)}$, and for more than one filter, i.e. $ker \in \mathbb{R}^{d \times k \times k}$, $d \in \mathbb{N}$.

Pooling layers are used to reduce the spacial dimension. Due to the structure of images and the usage of local filters adjacent outputs of the convolution layer may be similar.

For $n \times n$-Pooling, $n \in \mathbb{N}$, $n \geq 2$, the image is split into $n \times n$ blocks and a function $f : \mathbb{R}^{n \times n} \to \mathbb{R}$ is applied to every block resulting in a smaller spacial dimension. Frequently used functions f are taking the maximum or average of the input.

[KSH12] also chose to use the ReLU activation function instead of the sigmoid activation function because the ReLU function has better gradient properties. In the backpropagation algorithm the derivative of the activation function is used. The derivative of the sigmoid function is

$$\varphi_2'(x) = \varphi_2(x)(1 - \varphi_2(x)), \qquad x \in \mathbb{R}.$$

We can conclude from this equation, that if $|x|$ is large, then $\varphi_2'(x)$ is near zero resulting in small gradients (saturation) and the model does not learn. In praxis "networks with ReLUs consistently learn several times faster"[KSH12].

2.2.2. Residual connections and batch normalization

In 2015 [He+16] improved the results again by introducing residual connections to the convolution layers and using batch normalization [IS15]. These concepts work as follows:

Let g be a convolution layer with same input and output shape and x an input to the convolution layer. The output using a residual connection is then $x + g(x)$. This has the advantage that the layer g only has to learn higher order features pertubating the input slightly. This helps for deeper network architectures. Another way to look at this construction is that it helps to maintain the gradient flow during backpropagation since that gradient is transmitted undisturbed by the identity part of the residual connection.

In a batch normalization layer the output of another layer is normalized component-wise to mean zero and variance one. Using batch normalization layers can results in a model which can be trained faster compared to a model not using batch normalization layers.

Both [KSH12] and [He+16] trained their models on ImageNet [Den+09] achieving top-5 error rate 17.0% and 3.57%, respectively. The task was to classify an image into one of 1000 classes. For comparison the top-5 error rate of humans is around 5.1% [Rus+15].

If the amount of data given for image classification is small, it may be advantageous to use a model pre-trained on ImageNet and retrain the weights of the last few layers/last layer on the given data while fixing the rest of the weights.

2.3. Recurrent neural networks

Feedforward models are not well suited for sequence inputs like sentences or audio. To overcome this issue, recurrent neural networks (RNNs) have been invented. If a RNN is applied to an input, it produces a hidden state and an output. The hidden state is used when the RNN is applied to the next input. This can be seen as a circular connection between nodes and allows for an internal memory of the network.

2.3.1. Elman RNN

In 1990 Elman proposed a RNN. Nowadays it is called Elman RNN [Elm90]. At each timestep $t \in \mathbb{N}$ it takes a vector $x_t \in \mathbb{R}^n$, $n \in \mathbb{N}$, and produces a hidden state $h_t \in \mathbb{R}^m$, $m \in \mathbb{N}$, and an output $y_t \in \mathbb{R}^k$, $k \in \mathbb{N}$, provided an initial hidden state $h_0 \in \mathbb{R}^m$, defined by

$$\begin{aligned} h_t &:= \varphi_h(W_h x_t + U_h h_{t-1} + b_h), \\ y_t &:= \varphi_y(W_y h_t + b_y), \end{aligned}$$

where $W_h \in \mathbb{R}^{m \times n}$, $U_h \in \mathbb{R}^{m \times m}$, $b_h \in \mathbb{R}^m$, $W_y \in \mathbb{R}^{k \times m}$, $b_y \in \mathbb{R}^k$ are weights and φ_h and φ_y are activation functions.

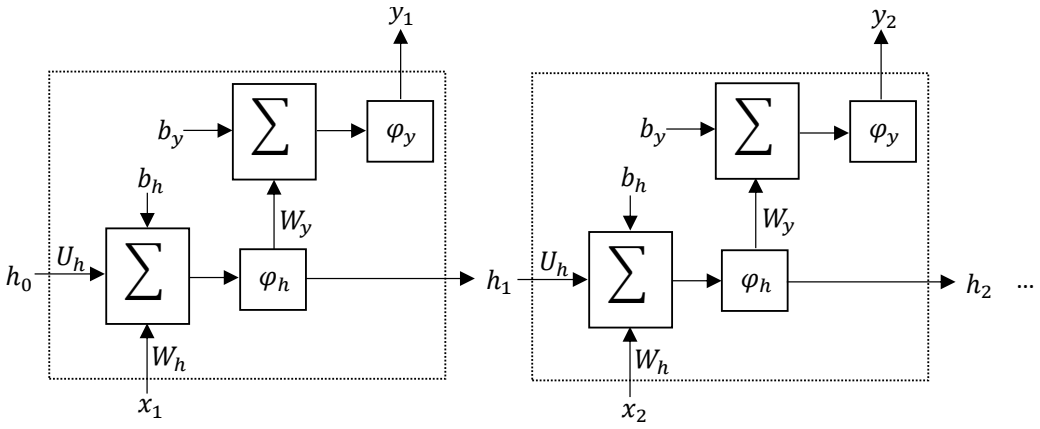

Figure 2.3.: Unfolded Elman RNN: Labeled arrows indicate a multiplication with the label, boxes denote an operation applied to the inputs.

Similar to the backpropagation algorithm a method called Backpropagation through time [Wer+90] was invented to learn the weights of a RNN. The RNN is unfolded through time as shown in figure 2.3 and the backpropagation algorithm is applied.

The Elman RNN faces the problem of exploding and vanishing gradients. This occurs during backpropagation due to the fact that the error is backpropagated multiple times through the matrix multiplication with U_h, e.g. backpropagating from y_t to x_1 to calculate

the derivative w.r.t. W_h this derivative contains $t-1$ factors U_h^T. This repeated application of the same matrix leads to either exploding or vanishing gradients. Exploding gradients can be dealt with by clipping the gradients if it is to large, for the problem of vanishing gradients an architectural improvement has been made.

2.3.2. Long short-term memory

To deal better with vanishing gradients, the Long short-term memory (LSTM) [HS97], a RNN, was proposed. Instead of the one hidden variable of the Elman RNN, the LSTM has two hidden variables. The LSTM takes at each timestep $t \in \mathbb{N}$ an input $x_t \in \mathbb{R}^n$, $n \in \mathbb{N}$ and calculates two hidden states $h_t \in \mathbb{R}^m$ and $c_t \in \mathbb{R}^m$, $m \in \mathbb{N}$, and an output $y_t \in \mathbb{R}^k$, $k \in \mathbb{N}$, provided the initial hidden states $h_0 \in \mathbb{R}^m$ and $c_0 \in \mathbb{R}^m$, defined by

$$\begin{aligned}
\begin{pmatrix} f \\ i \\ o \\ g \end{pmatrix} &:= \begin{pmatrix} \varphi_2 \\ \varphi_2 \\ \varphi_2 \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}, \\
c_t &:= f \cdot c_{t-1} + i \cdot g, \\
h_t &:= o \cdot \tanh(c_t), \\
y_t &:= \varphi_y(W_y h_t + b_y),
\end{aligned}$$

where $W \in \mathbb{R}^{4m \times (m+n)}$, $W_y \in \mathbb{R}^{k \times m}$ and $b_y \in \mathbb{R}^k$ are weights and φ_y is an activation function.

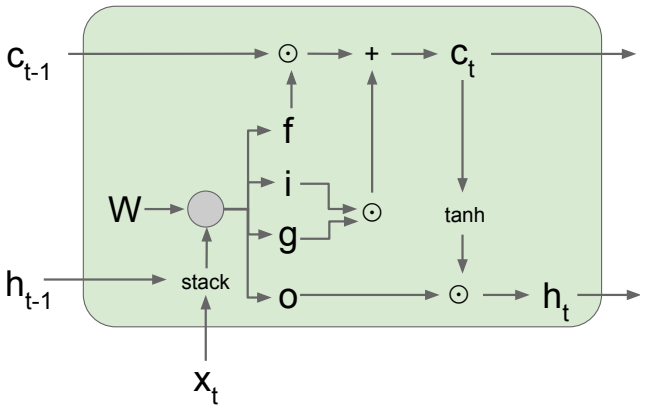

Figure 2.4.: LSTM architecture, $\odot$ denotes the pointwise multiplication [LJY17].

Compared to the Elman RNN the LSTM has a better gradient flow since in each timestep when backpropagating from c_t to c_{t-1} one has a pointwise multiplication (with not necessarily equal vectors per timestep) instead of a matrix multiplication with always the same matrix. Thus the gradient w.r.t. c_t has better properties w.r.t. vanishing gradients. This carries over to the gradient w.r.t. W. Therefore by using LSTMs the model is able to use the context of longer sequences.

2.4. Encoder-decoder models

In this section encoder-decoder models are described. An encoder-decoder model is a special sequence-to-sequence model, i.e. a model that maps an input sequence of arbitrary length to an output sequence of arbitrary length. Sequence-to-sequence models can be used for instance in machine translation and speech recognition.

2.4.1. RNN Encoder-decoder model

[Cho+14] and [SVL14] published the RNN encoder-decoder model. The model is based on a RNN used as encoder and a RNN used as decoder and trained on a machine translation task. Due to the reasons discussed above LSTMs are used over Elman RNNs.

The model uses a source vocabulary and a target vocabulary. A vocabulary is a finite sets of tokens (identified with $\{1, \dots, k\}$, $k \in \mathbb{N}$). The model is trained with data consisting of an input sequence of tokens in the source vocabulary $(\hat{x}_1, \dots, \hat{x}_T)$, $T \in \mathbb{N}$, and a target sequence of tokens in the target vocabulary $(\hat{z}_1, \dots, \hat{z}_{T'-1})$, $T' \in \mathbb{N}$. The target sequence is padded with a start of sequence token $\hat{z}_0$ and an end of sequence token $\hat{z}_{T'}$ to $(\hat{z}_0, \dots, \hat{z}_{T'})$. These sequences are transformed into sequences of vectors $(x_1, \dots, x_T)$, $x_t \in \mathbb{R}^n$, $n \in \mathbb{N}$, $t \in \{1, \dots, T\}$, and $(z_0, \dots, z_{T'})$, $z_t \in \mathbb{R}^{n'}$, $n' \in \mathbb{N}$, $t \in \{0, \dots, T'\}$, via an embedding layer (one vector of weights is assigned to each token in the vocabularies).

The model (see figure 2.5) then processes the input sequence $(x_1, \dots, x_T)$ and encodes it with a RNN f (given $h_0 \in \mathbb{R}^{n_h}$)

$$h_t \coloneqq f(x_t, h_{t-1}), \qquad t \in \{1, \dots, T\},$$

into fixed-length vector $s_0 \coloneqq h_T$. Another RNN g takes s_0 as the initial hidden state and the target sequence $(z_0, z_1, \dots, z_{T'})$ and decodes it (except $z_{T'}$) into a sequence $(y_1, \dots, y_{T'})$, $y_t \in \mathbb{R}^m$, $t \in \{1, \dots, T'\}$, where $m \in \mathbb{N}$ is the size of the target vocabulary. Both RNNs can be jointly trained to minimize

$$\sum_{t=1}^{T'} l_{\text{classification},e_{\hat{z}_t}}(y_t), \tag{2.5}$$

where the output layer of the RNN g uses a softmax activation function, $l_{\text{classification},\cdot}$ is defined in equation (2.2), and e_l, $l \in \mathbb{N}$, denotes the l-th unit vector. Equation (2.5) can be interpreted that at each timestep the model is trained to predict the next token of the sequence provided the sequence until the current position.

During test time the input sequence is used to calculate h_T. Next, h_T and the start of sequence token z_0 is given to the RNN g, the output y_1 is calculated, from the distribution y_1 is sampled and the embedding of the sampled token is fed back to the RNN to calculate

y_2. This is iterated until the end of sequence token is sampled. The sampled tokens form the output sequence. There are also more sophisticated decoding schemes, e.g. beam search.

2.4.2. Attention-based encoder-decoder model

One problem resulting from the construction in section 2.4.1 is that each input sequence is encoded into a fixed-length vector regardless of its length. This results in an information-flow bottleneck. [BCB14] proposed as solution an attention mechanism (see figure 2.5). The core idea is that h_j contains information about the sequence $(x_1, \dots, x_j)$, but should focus on the part of the sequence near x_j. Based on the vector s_{i-1} the model is able to attend to these h_j's allowing the model to search for parts of the source sequence that seem relevant in the current step. Mathematically this is done by

$$
\begin{aligned}
e_{i,j} &:= v^T \tanh(W s_{i-1} + U h_j), \\
\alpha_{i,j} &:= \frac{\exp(e_{i,j})}{\sum_{k=1}^{T} \exp(e_{i,k})}, \qquad\qquad i \in \{1, \dots, T'\},\ j \in \{1, \dots, T\}, \\
c_i &:= \sum_{j=1}^{T} \alpha_{i,j} h_j,
\end{aligned}
$$

where $v \in \mathbb{R}^{n_a}$, $n_a \in \mathbb{N}$, is a weight vector and $W, U \in \mathbb{R}^{n_a \times n_h}$ are weight matrices. For fixed i the $e_{i,j}$'s form an alignment model between s_{i-1} and the h_j's. The RNN g of the attention-based encoder-decoder model has as input in addition to s_{i-1} also c_i.

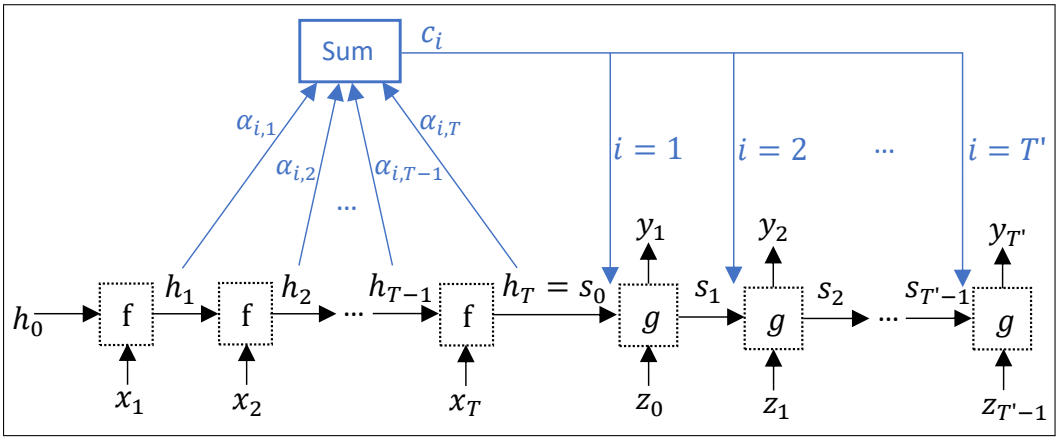

Figure 2.5.: Attention-based RNN encoder-decoder model

2.4.3. Transformer model

In 2017 [Vas+17] published the transformer model, an encoder-decoder model. The idea of using an attention mechanism is pursued resulting in an attention layer. Similar to the

RNN encoder-decoder model the transformer model consists of an encoder and a decoder (see figure 2.7), but it uses these attention layers instead of RNNs.

The attention layer has two matrices as input. One representing the memory $M \in \mathbb{R}^{T \times d_1}$, $d_1 \in \mathbb{N}$, $T \in \mathbb{N}$, to which the layer attends and one the context $C \in \mathbb{R}^{T' \times d_2}$, $d_2 \in \mathbb{N}$, $T' \in \mathbb{N}$. The first step of the attention layer is to apply linear transformations to C to obtain a matrix denoted by $Q \in \mathbb{R}^{T' \times d_k}$ (Query), $d_k \in \mathbb{N}$, and to M to obtain matrices denoted by $K \in \mathbb{R}^{T \times d_k}$ and $V \in \mathbb{R}^{T \times d_v}$ (Key and Value), $d_v \in \mathbb{N}$. The output of the attention layer is then given by

$$\text{Attention}(Q, K, V) := \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V,$$

where the softmax is taken w.r.t. the rows of the matrix. The factor $\frac{1}{\sqrt{d_k}}$ is a scaling factor. The intuition behind this construction is that first the input gets transformed into Query and Key, projecting the features in a space where the dot product is a good similarity measure, and then mix similar information from context and memory.

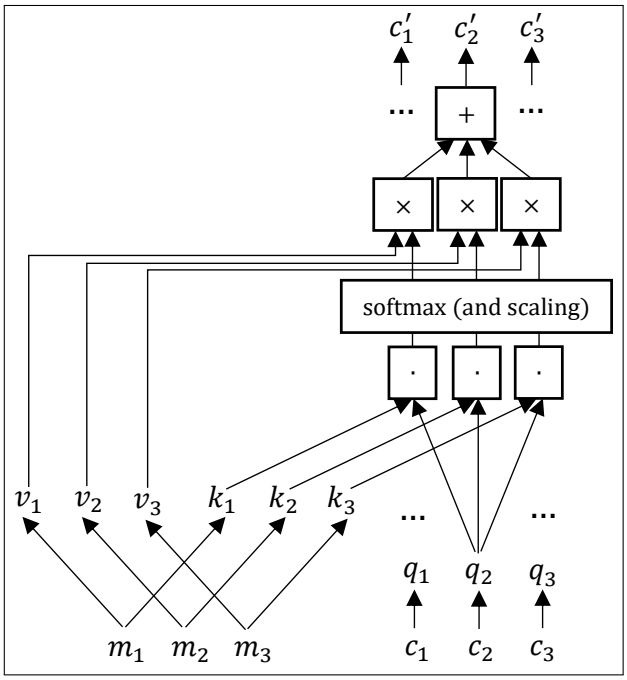

Figure 2.6.: Attention layer for memory and context sequences of length three. Index i denotes the i-th row of the corresponding matrix.

This described attention layer is permutation invariant, i.e. the layer is not able to distinguish the order of the input sequence. For instance, if in figure 2.6 the vectors m_1 and m_3 are interchanged the output would be the same. Of course the order of the input sequence may be relevant, therefore a positional encoding is added to the input sequence.

The encoder consists of multiple encoder layers. Each encoder layer consists of a self-attention layer, i.e. an attention layer where as memory and context matrices the output of

the previous encoder layer (or the input sequence for the first encoder layer) is used, and a pointwise feedforward network. Both layers are employed with residual connections and a layer normalization [BKH16].

The decoder mimics a language model and consists of multiple decoder layers. Each decoder layer consists of a masked self-attention layer, an encoder-decoder attention layer, and a pointwise feedforward network. The masked self-attention layer has as memory and context matrices the output of the previous decoder layer (or the target sequence for the first decoder layer). The encoder-decoder attention layer has the output of the encoder as memory and the output of the masked self-attention layer as context. Similar to the encoder residual connections and layer normalization are used. By masking out subsequent positions in the masked self-attention layer it is ensured that the model is autoregressive, i.e. one output of the decoder only depends on the encoder output and target sequence up to the current position, and during inference the target sequence can be decoded iteratively similarly to the RNN encoder-decoder model.

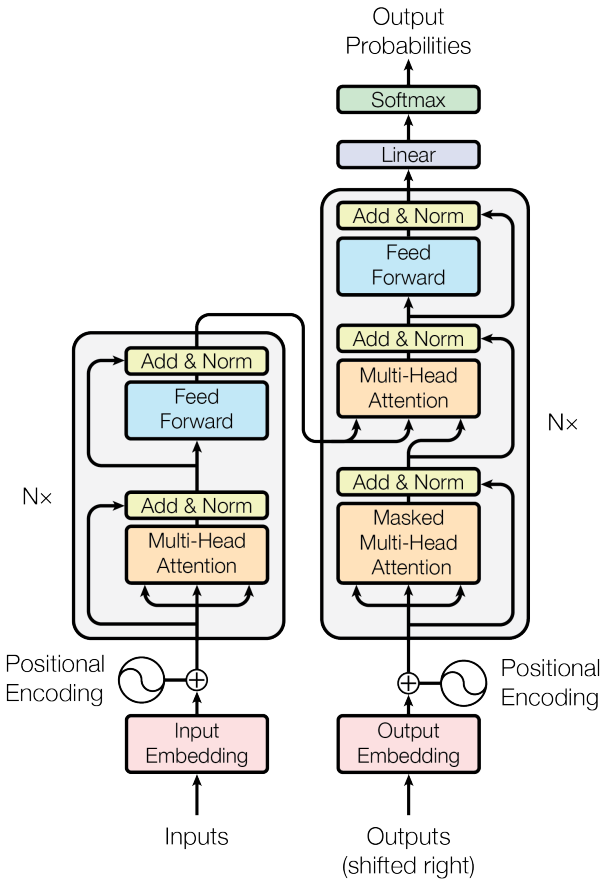

Figure 2.7.: Transformer architecture [Vas+17].

Instead of one attention layer it can be beneficial to use multiple attention layers (with smaller dimensions) on the same input, stack the outputs and apply a linear transformation to the stacked outputs. This is called multi-head attention and allows the model to attend to multiple parts of the input sequence.

3. Related work

In the last decade major progress has been made with neural networks, surpassing human performance in many tasks, e.g. image classification [He+16] and object detection [Rus+15] and games like go [Sil+17b], chess and shogi [Sil+17a], where reinforcement learning is applied. However, the applied methods have the drawback that a lot of data is needed, the model gets trained, then deployed and is fixed. For the reinforcement learning methods the whole environment can be simulated and all possible actions are known a priori.

On the other hand, humans face different tasks each day and are able to adapt rapidly with only little data given. For neural networks this is still a very challenging problem. Retraining a neural network on a new task results in catastrophic forgetting [Fre99] of the old task. Therefore more advanced techniques have been developed.

3.1. Incremental learning

This section will give an introduction to incremental-learning methods for image classification. The problem can be described as follows (similar to [De +19]):

At each timestep $t \in \mathbb{N}$ training data X_t and labels Y_t are given (sampled from sets $\mathcal{X}_t, \mathcal{Y}_t$, i.e. $(X_t, Y_t) \sim (\mathcal{X}_t, \mathcal{Y}_t)$) and can be used to learn the system. The goal is to minimize the following error

$$\sum_{t=1}^{T} \mathbb{E}_{(\mathcal{X}_t, \mathcal{Y}_t)} \left[l(f_t(\mathcal{X}_t, \theta), \mathcal{Y}_t) \right]$$

with limited or no access to the data with $t < T$, where T is the number of seen tasks so far. The symbol l denotes a loss function, $f_t(x, \theta)$ represents the output of the network for task t and an input x, and θ are the parameters of the network.

Following [Hsu+18] the methods for incremental learning can be categorized dependent on the data $(X_t, Y_t)_t$ into methods for incremental domain learning, methods for incremental class learning and methods for incremental task learning.

In incremental domain learning the distribution of the training data X_t changes but the classes remain the same, e.g. a class in X_t could be vehicles, in timestep one one gets bicycles, in timestep two one gets motorcycles etc.

In incremental class learning the class labels $(Y_t)_t$ are disjoint but the goal is to train a system which classifies all classes seen so far (without the information of which task the input to classify is), e.g. in timestep one one gets different vehicles, in timestep two one gets different animals. After learning with the animal data one wants a classifier which distinguishes between all vehicles and animals.

In incremental task learning the class labels $(Y_t)_t$ are disjoint and the goal is to train a classifier for each seen task, e.g. for the example above the information if the input is a vehicle or an animal is provided. Therefore incremental class learning can be seen as a superset of incremental task learning.

3.1.1. Incremental class learning

According to [De +19] methods for incremental class learning can be divided into replay-based methods, regularization-based methods and parameter isolation-based methods. For more details refer to [De +19] or the corresponding source.

In replay-based methods, e.g. [Reb+17], [LR17], a subset of the samples of old tasks is stored. These samples can be stored as raw data and used for retraining the network when new tasks arrive. This is called rehearsal. It is also possible to store the data compressed in a generative model, e.g. [Shi+17], [VT18], and use images generated by this generative model while training on new tasks, called pseudo rehearsal. For example [Shi+17] works as follows: For the initial task a generative model and a classification model are trained. When a new task arrives, samples from the latest generative model are mixed with current data to train a new generative model. Next, samples of the latest generative model together with labels produced by applying the latest classification model to them and the current data are used to train the new classification model. Since generative models are trained this method requires a certain amount of data per task.

In regularization-based methods no samples of the old tasks are stored. Reasons to do this are privacy issues and memory efficiency. Instead of storing samples the output of the previous model is used as soft labels for the previous task, e.g. [LH17], [Zha+20]. There are also methods penalizing the change of important parameters of certain tasks during learning of new tasks, e.g. [Kir+17].

In parameter isolation-based methods, e.g. [ML18], “different subsets of the model parameters are dedicated to each task”[De +19]. This can mean that for new tasks new branches are created or freezing parts of the parameters during training of new tasks. Most parameter isolation-based methods need a task oracle, i.e. they are restricted to

incremental task learning. An exception is for instance [Raj+19]. In [WSS89] a parameter isolation-based method is used to fuse two classifiers using additional connections called "connectionist glue". However, the resulting classifier is trained with all data used to train the two classifiers.

The methods used in incremental domain learning are similar to the methods used in incremental class learning. State-of-the-art are rehearsal-based methods ([Shi+17], [VT18]) with generative models [Hsu+18].

3.2. Meta-learning

In this section an introduction to meta-learning similar to [Ber+18] is given since our method is based on a meta-learning approach.

The meta-learning method consists of a base-model and a meta-model. The base-model is used within one episode, i.e. an image classification task with a small amount of given data. The meta-model learns from many such episodes to learn the structure of the problem and leverage this to improve the performance of the base-model.

Mathematically this can be done by denoting the set of training episodes by E and the set of validation episodes by E'. An episode $\mathcal{E}$ of these sets consists of inputs $x \in \mathbb{R}^m$, $m \in \mathbb{N}$, and corresponding outputs $y \in \mathbb{R}^n$, $n \in \mathbb{N}$, and can be partitioned in a training set $\mathcal{E}_1$ and a test set $\mathcal{E}_2$: $\mathcal{E} = \mathcal{E}_1 \dot\cup \mathcal{E}_2$.

Let

$$\phi_\omega : \mathbb{R}^m \to \mathbb{R}^e, \qquad \text{feature extractor,}$$

$e \in \mathbb{N}$, be a feature extractor depending on weights ω. We denote the base-model (dependent on episodic-specific weights $\omega_\mathcal{E}$) by

$$f_{\omega_\mathcal{E}} : \mathbb{R}^e \to \mathbb{R}^n, \qquad \text{base-model,}$$

the meta-model by Λ_ρ and a loss function by $L : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. The meta-model depends on weights ρ and maps the features of the inputs and the corresponding labels to episodic-specific weights ω_ϵ. If the functions ϕ_ω, f_{ω_ϵ} and Λ_ρ are differentiable one can learn the weights ω and ρ by minimizing the training error

$$\frac{1}{|E|} \sum_{\mathcal{E} \in E} \frac{1}{|\mathcal{E}_2|} \sum_{(x,y) \in \mathcal{E}_2} L(f_{\omega_\epsilon}(\phi_\omega(x)), y), \qquad \text{where } \omega_\epsilon = \Lambda_\rho(\phi_\omega(\mathcal{E}_1)), \tag{3.1}$$

with backpropagation, where $\phi_\omega(\mathcal{E}_1) \coloneqq \{(\phi_\omega(x), y) \mid (x, y) \in \mathcal{E}_1\}$. The meta-model has as input the training part of an episode and the error is calculated with the test part. The performance of the meta-model is evaluated by the validation error which is obtained by replacing E by E' in equation (3.1).

3.2.1. Examples for meta-learning algorithms

In this section a few meta-learning algorithms with their meta-models are described. For more details refer to [Ber+18] or the corresponding source.

In [And+16] an optimizer is learned. The meta-model works as follows: From the input data the gradient of the loss function w.r.t. the weights of the neural network is computed and given to an coordinate-wise LSTM. The LSTM outputs updates for the weights, then these updates are applied resulting in new weights. This process is iterated for a certain number of steps.

[RL16] is similar to [And+16]. However the loss and the old weights are additionally given to the LSTM and the LSTM outputs not weight updates but new weights. Furthermore the initial hidden state of the LSTM consists of learned parameters of the meta-model.

In [Ber+16] the meta-model uses a single image as input and maps it to weights of a network. This network uses an image as input and evaluates if it is similar to the input image of the meta-model. To reduce the number of parameters of the meta-model, factorizations of linear transformations and convolution layers are proposed.

In [FAL17] initial weights of a network are learned such that these weights are adjustable on new tasks with only a few steps of an optimizer using the gradient of the loss function w.r.t. the weights. Therefore the meta-model is the application of the optimizer with the computed initial weights.

4. Methods: Sample-incremental meta-learning

In this work we consider an incremental-learning method for image classification. More specifically, we want to obtain an image classifier from little data (for a fixed number of classes) whose performance can be improved with more data in an incremental manner. In [Hsu+18] this type of incremental learning is called incremental domain learning. To achieve this we use a meta-learning method (see section 3.2). The core idea of our method is that we train a mapping-model which adjusts weights of a base-model to incrementally incorporate new knowledge.

The sample-incremental meta-learning approach of this work consists of a meta-model (see section 4.2) which outputs weights of a base-model given little data and the base-model is used together with a feature extractor to classify input images into $n \in \mathbb{N}$ classes.

The meta-model consists of a mapping-model (see section 4.3) which is used for incremental learning to improve the performance of the system. In section 4.4 this incremental learning during inference is described in detail. In section 4.5 the methods to train the meta-model are specified.

4.1. Classification model

We use an on ImageNet pretrained ResNet34 model [He+16] with removed last layer as feature extractor and a linear transformation as base-model. The composition of the feature extractor and the base-model is called classification model (see figure 4.1).

ResNet models with more layers are not used since they have a larger feature dimension 2048 instead of 512 and therefore the used meta-model would have to be larger. More complex base-models are discussed in section 5.4.

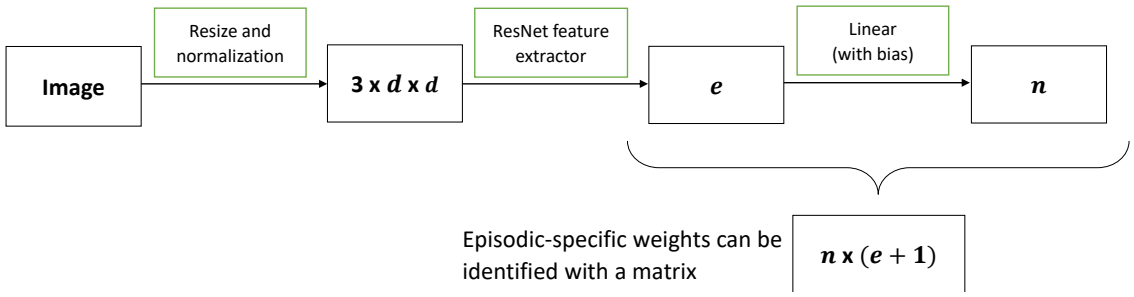

Figure 4.1.: Classification model

The feature extractor uses an input dimension $d = 224$ and outputs a feature vector of dimension $e = 512$.

4.2. Meta-model

In the following we describe the meta-model. It is designed to take features of input images as input and map them to weights which can be used for the base-model.

As presented in section 3.2 the meta-model learns from many episodes. We write such an episode $\mathcal{E}$ as

$$\mathcal{E} = \underbrace{\left\{(x_{i,1}, e_1)_i, \ldots, (x_{i,n}, e_n)_i \mid i \in \{1, \ldots, k\}\right\}}_{=\mathcal{E}_1} \dot{\cup}\, \mathcal{E}_2 \tag{4.1}$$

for some $k \in \mathbb{N}$, i.e. the episode is split such that the training set of the episode is balanced over the different classes. Images are denoted by $x_{i,j} \in \mathbb{R}^{d \times d}$, $i \in \{i, \ldots, k\}$, $j \in \{1, \ldots, n\}$, the l-th unit vector is denoted by e_l, $l \in \{1, \ldots, n\}$, and n denotes the number of classes.

From the first part $\mathcal{E}_1$ of $\mathcal{E}$ in equation (4.1) the input for the meta-model Λ_ρ is created. The images $x_{i,j}$, $i \in \{i, \ldots, k\}$, $j \in \{1, \ldots, n\}$, are given to the feature extractor ϕ_ω and matrices s_i containing the feature vectors in the rows are calculated:

$$s_i \coloneqq \left(\phi_\omega(x_{i,1}) | \ldots | \phi_\omega(x_{i,n})\right)^T, \qquad i \in \{1, \ldots, k\}.$$

The meta-model Λ_ρ (see figure 4.2) is based on a mapping-model $M_{\rho_2} : \mathbb{R}^{n \times e} \times \mathbb{R}^{n \times (e+1)} \to \mathbb{R}^{n \times (e+1)}$ (described in section 4.3) and it depends on weights ρ. These weights ρ consist of weights $\rho_1 \in \mathbb{R}^{n \times (e+1)}$, which are used as initial weights for the mapping, and the weights ρ_2 of the mapping-model. The initial weights ρ_1 are similar to an initial hidden state of a RNN consisting of learned weights.

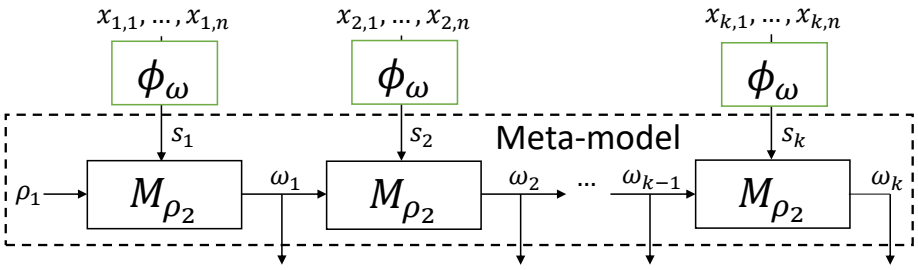

Figure 4.2.: Meta-model Λ_ρ

The outputs ω_i, $i \in \{1, \ldots, k\}$, of the meta-model Λ_ρ are defined by

$$\begin{aligned} \omega_0 &:= \rho_1, \\ \omega_i &:= M_{\rho_2}(s_i, \omega_{i-1}). \end{aligned}$$

The idea behind the approach of the mapping-model is that the mapping-model gets weights as input and additional data (features) and should adjust these weights such that they correspond to weights one would obtain by adding the additional data to the knowledge the system already has. Therefore the output weights of the mapping-model should perform better for the classification than the input weights.

The labels of the input feature vectors are encoded in the order of them and exactly one input sample of each class is used for the mapping-model input. Furthermore the weights ρ_1 are randomly initialized similar to an initialization of the base-model (see section 2.1.6). We use as initialization scheme an uniform distribution with variance as in equation (2.4).

4.3. Mapping-model

The mapping-model M_{ρ_2} has as input a matrix of feature vectors (one feature vector per class) and a matrix of weights of the base-model and maps them to a matrix of weights which again can be used as weights for the base-model (see figure 4.3).

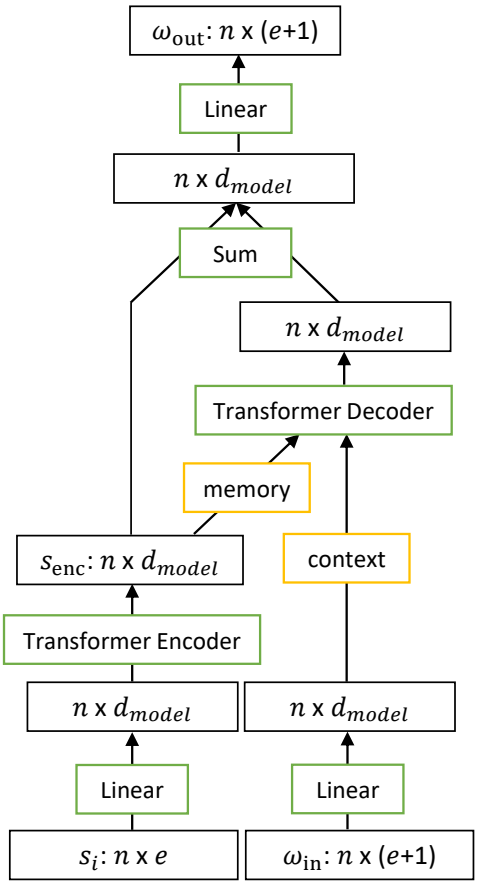

Figure 4.3.: Mapping-model M_{ρ_2}: Green boxes denote the applied functions, black boxes contain the shape of the processed data (and the names if they are present).

More abstractly this can be seen as

$$\begin{aligned} s_{\text{enc}} &= \text{encode}(s_i), \\ \omega_{\text{out}} &= \text{decode}(\text{context} = \omega_{\text{in}}, \text{memory} = s_{\text{enc}}) + s_{\text{enc}}, \end{aligned} \tag{4.2}$$

where s contains the feature vectors and ω_{in} and ω_{out} are the matrices of input weights and output weights, respectively.

A transformer model is used within the mapping-model since this mapping can be seen as a sequence-to-sequence task. The sequence of feature vectors is used as source sequence and the sequence of weights (corresponding to each class) are used as target sequence. The transformer decoder in the model has a self-attention layer instead of a masked self-attention layer since during inference no decoding is done but a forward path through the transformer is done and therefore it is not necessary to mask the target sequence. The output of the encoder and decoder are summed to achieve a better alignment between the source and target sequence.

Furthermore it is important to pay attention to the scaling of the output. Since the output of the mapping-model consists of weights it should be scaled such that applying the linear

transformation with these weights to feature vectors should result in a reasonably scaled output. Therefore we scale the output weights of the mapping-model such that they have the same variance as the initial weights ρ_1.

4.4. Inference

During inference the model is able to learn with one sample per class as input to the meta-model Λ_ρ. As a result one gets episodic-specific weights ω_1 and is able to use them together with the weights ω of the feature extractor for the classification model. Given an image x to classify the system outputs the following probability distribution over the classes:

$$f_{\omega_1}(\phi_\omega(x))$$

If more samples per class are available one can use the mapping-model again to get episodic-specific weights ω_i, for $i > 1$. This is possible in an incremental manner, i.e. if the samples are delivered one after each other the ones already processed by the mapping-model do not need to be stored. An illustrative example can be found in figure 5.9. Furthermore this mapping is a lot faster than fine-tuning or retraining a complete model since only a forward path of the mapping-model has to be performed.

4.5. Training

The meta-model and the feature extractor can be trained with different loss functions and data. In the following we will consider these methods.

For all following training methods the validation error is calculated (similar to equation (3.1)) by

$$\frac{1}{|E'|}\sum_{\mathcal{E}\in E'}\frac{1}{k}\sum_{i=1}^{k}\frac{1}{|\mathcal{E}_2|}\sum_{(x,y)\in\mathcal{E}_2} L_{CE}(f_{\omega_i}(\phi_\omega(x)),y). \tag{4.3}$$

Training method 1:

For training method 1 (see figure 4.4) the cross entropy loss is chosen.

Training method 1.1:

The output weights ω_i of the meta-model are used in the base-model and the base-model is trained with the data $(\phi_\omega(x), y)$ for $(x, y) \in \mathcal{E}_2$, i.e. the test part of $\mathcal{E}$. This means that every ω_i is trained with the same data and this data is not used as input of the meta-model.

The training error is similar to the validation error (equation (4.3)) and defined by

$$\frac{1}{|E|}\sum_{\mathcal{E}\in E}\frac{1}{k}\sum_{i=1}^{k}\frac{1}{|\mathcal{E}_2|}\sum_{(x,y)\in\mathcal{E}_2} L_{CE}(f_{\omega_i}(\phi_\omega(x)), y),$$

where the weights ρ of the meta-model are learned (the weights ω_i depent on ρ).

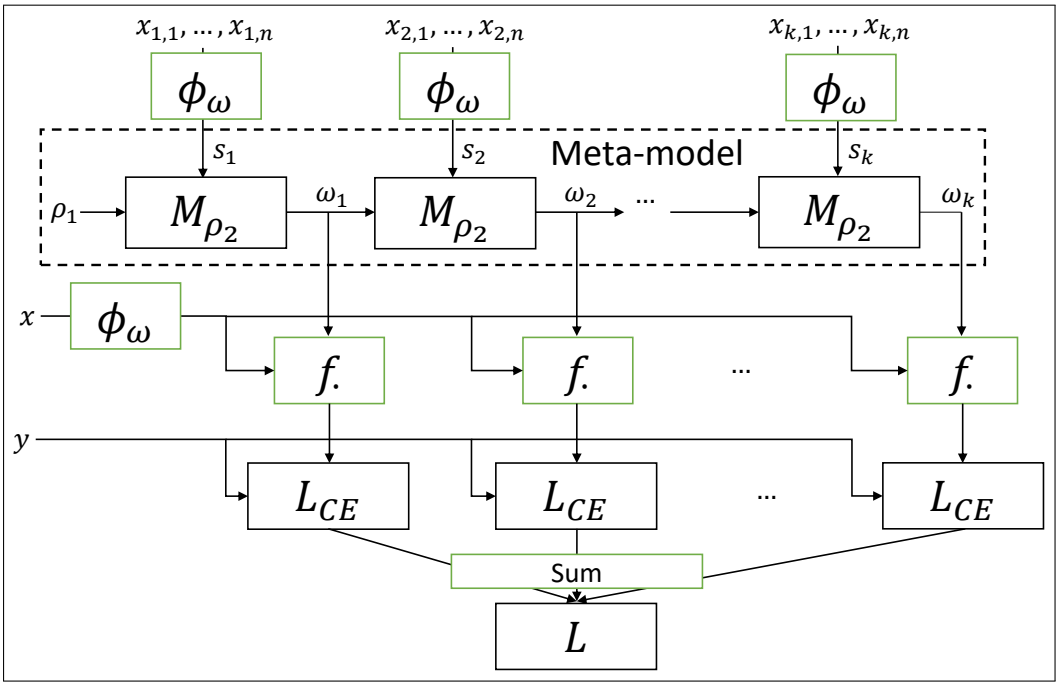

Figure 4.4.: Training method 1: The output of the meta-model is given to the base-model and the base-model is trained with a cross entropy loss.

The intuition behind this training method is that the weights in ω_i depend on the inputs $x_{j,1}, \ldots, x_{j,n}$, $j \in \{1, \ldots, i\}$, and therefore when all weights get trained with the same data, weights with a higher index i should deliver a better performance. As a result the mapping-model should learn to approximate an incremental-learning function.

Training method 1.2:

The output weights ω_i of the meta-model are given to the base-model and the base-model is trained with the data $((\phi_\omega(x), y)$ for $(x, y) \in \{(x_{j,1}, e_1), \ldots, (x_{j,n}, e_n) \mid j \in \{1, \ldots, i\}\}$, $i \in \{1, \ldots, k\}$. This means that the weights are trained with the input to the meta-model and weights ω_i with higher index i are trained with more data.

The following expression is minimized w.r.t. ρ:

$$\frac{1}{|E|}\sum_{\mathcal{E}\in E}\frac{1}{k}\sum_{i=1}^{k}\underbrace{\frac{1}{i}\sum_{j=1}^{i}\frac{1}{n}\sum_{l=1}^{n}L_{CE}(f_{\omega_i}(\phi_\omega(x_{j,l})),e_l)}_{=:L^i}. \tag{4.4}$$

In this approach the base-model with the weights ω_i is trained with only the data available to the weights ω_i through the meta-model. Therefore the mapping-model is trained to map the input weights to weights which arise when adding the mapping-model feature inputs to the training set, i.e. the mapping-model is trained to approximate an incremental-learning function.

Training method 2:

This method is based on training method 1.2 and starts with a precomputation step. For each episode $\mathcal{E}$ weights $\widetilde{\omega}_i$, $i \in \{1,\dots,k\}$, are precomputed by minimizing the following validation error w.r.t. $\widetilde{\omega}_i$

$$\begin{aligned}
&\frac{1}{i}\sum_{j=1}^{i}\frac{1}{n}\sum_{l=1}^{n}L_{CE}(f_{\widetilde{\omega}_i}(\phi_\omega(x_{j,l})),e_l), && \text{training error,}\\
&\frac{1}{|\mathcal{E}_2|}\sum_{(x,y)\in\mathcal{E}_2}L_{CE}(f_{\widetilde{\omega}_i}(\phi_\omega(x)),y), && \text{validation error.}
\end{aligned}$$

This is equivalent to the training of the base-model. As initial weights for $\widetilde{\omega}_i$ the weights $\widetilde{\omega}_{i-1}$ after the precomputation step are used, $i \in \{1,\dots,k\}$, and $\widetilde{\omega}_0$ is randomly initialized.

For the training method 2 (see figure 4.5) the mean squared error is chosen. For this method a fixed feature extractor is needed and it is not possible to jointly train meta-model and feature extractor. After the precomputation step the meta-model is trained to minimize

$$\frac{1}{|E|}\sum_{\mathcal{E}\in E}\frac{1}{k}\sum_{i=1}^{k}\underbrace{L_{MSE}(\omega_i,\widetilde{\omega}_i)}_{=:L^i} \tag{4.5}$$

w.r.t. ρ.

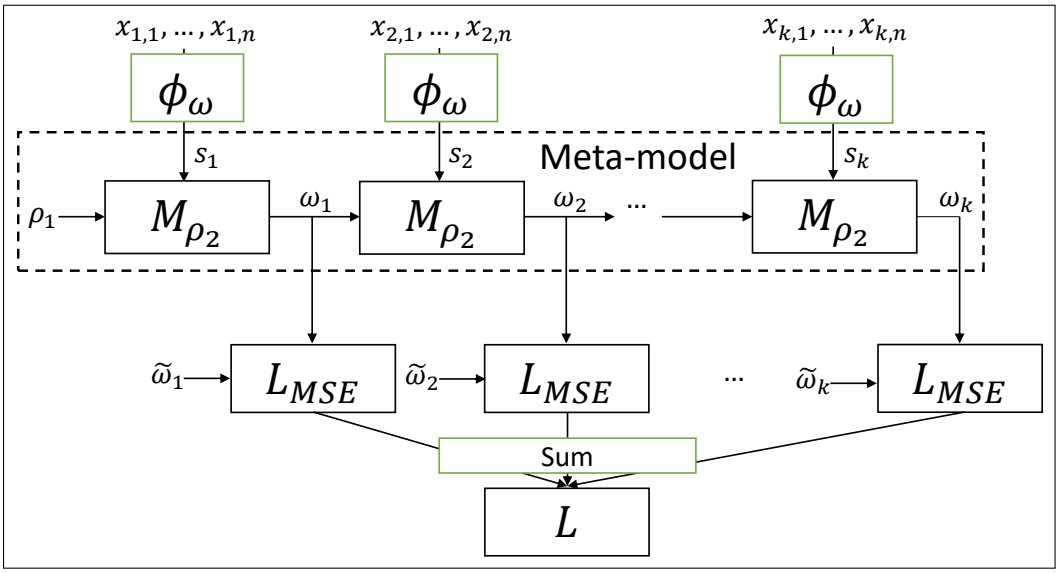

Figure 4.5.: Training method 2: The output of the meta-model is trained with the precomputed weights and a mean squared error.

One problem that emerges from the construction in the first two training methods is a vanishing or exploding gradient (see figure 4.6) since the same model is applied multiple times similar to an Elman RNN (see section 2.3).

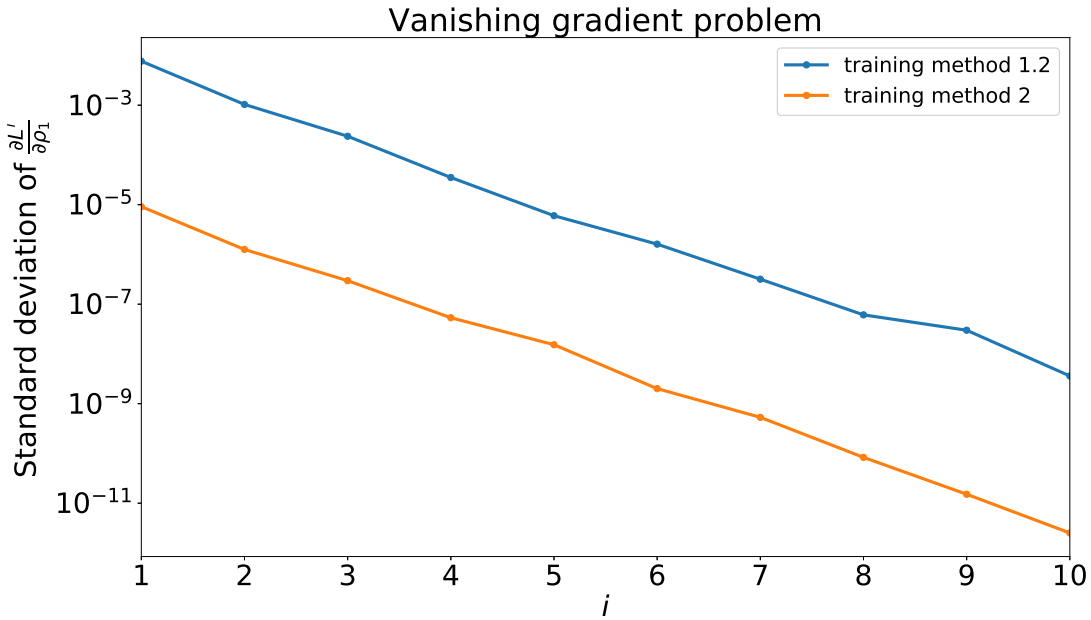

Figure 4.6.: Vanishing gradient problem: Standard deviation (averaged over 128 episodes) of the i-th loss part L^i w.r.t. ρ_1. For hyperparameters see appendix A.1.

Despite this vanishing-gradient problem the model is able to learn due to bootstrapping since there is not only the k-th part L^k of the loss but also the rest.

Training method 3:

This method is based on training method 2 but avoids the vanishing-gradient problem via teacher forcing. The precomputation step of training method 2 is adopted. During training the output of the mapping-model is not given back to the mapping-model. Instead the precomputed ground truth is given to the mapping-model. The weights ω_i (to compute the training error) are defined by

$$\omega_i := M_{\rho_2}(s_i, \widetilde{\omega}_{i-1}), \quad i \in \{1, \dots, k\},$$

and $\widetilde{\omega}_0 := \rho_1$.

Training method 3.1: (see figure 4.7)

As in training method 2 the meta-model is trained to minimize the term in equation (4.5).

Training method 3.2: (see figure 4.8)

As in training method 1.2 the meta-model is trained to minimize the term in equation (4.4).

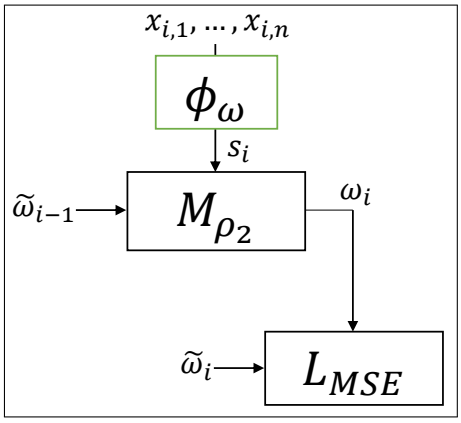

Figure 4.7.: Training method 3.1: Teacher forcing and a mean squared error are used.

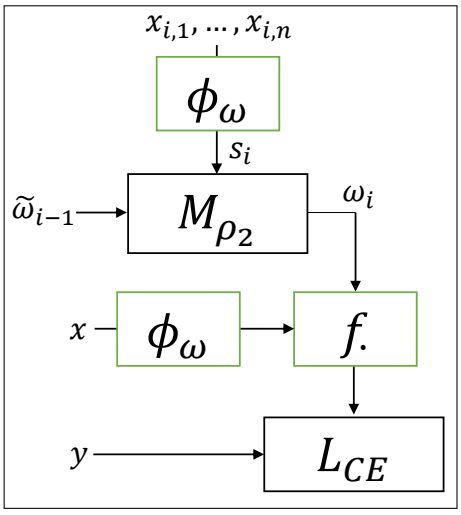

Figure 4.8.: Training method 3.2: Teacher forcing is used, the output ω_i is given to the base-model and the base-model is trained with a cross entropy loss.

training method	loss	mode	precomputation
1.1	CE	chain	no
1.2	CE	chain	no
2	MSE	chain	yes
3.1	MSE	teacher forcing	yes
3.2	CE	teacher forcing	yes

Table 4.1.: Summary of the training methods. Note that for the training methods 1.1 and 1.2 the training data is used in different ways.

4.6. Dataset

For training data a subset of ImageNet [Den+09], mini-ImageNet [Vin16], is used. It consists of 100 classes with 600 colored images of size 84 x 84 per class. The classes are split into 64, 16 and 20 classes for training, validation and test, respectively. This dataset is more complex than the CIFAR dataset [KH10] but smaller than ImageNet allowing for faster experiments.

The episodes are sampled from this data by randomly choosing n classes of the respective set and within this classes the data is randomly split into training and test sets.

5. Results

In this chapter the methods introduced in chapter 4 are compared with each other and to training and fine-tuning. For the training method the weights of the last step are taken (or initialized randomly at the begin, respectively) and trained with all the data the meta-model had as input, i.e. all of the old images are saved and used for updating the weights. This is used as a baseline. For the fine-tuning method the weights of the last step are taken (or initialized randomly at the begin, respectively) and trained with the current input of the meta-model, i.e. none of the old images are saved.

As mentioned in section 3.1, rehearsal-based methods ([Shi+17], [VT18]) with generative models [Goo+14] deliver state-of-the-art performance. However, these methods need a lot of data for the generative model to train, e.g. [VT18] trains with approximately 6000 samples per class, and can therefore not be compared to our method which learns with only one sample per class.

The number of classes $n = 4$ is chosen for the following experiments. The shown accuracies are obtained by testing the methods using the mini-ImageNet testset, i.e. the classes used during testing are not seen during training of the meta-model. For hyperparameters/results not mentioned in this chapter see appendix A.1/appendix A.3.

5.1. Evaluation of training and fine-tuning

Since during inference (see section 4.4) only one image per class is needed/used to improve the performance of the system, no validation set is available. To compare our methods to training and fine-tuning in a meaningful way also no test part (of the episode) should be provided during inference. Therefore during training the number of epochs to train/fine-tune during inference are calculated:

Weights ω_0 of the base-model are initialized randomly. To get weights ω_i from the weights ω_{i-1} which have the best accuracy in the last step, $i \in \{1, \ldots, k\}$, the base-model is trained with the data s_j, $j \in \{1, \ldots, i\}$, or fine-tuned with the data s_i for the training or fine-tuning method, respectively. This is done until test accuracy does not increase anymore for a certain number of epochs $e_{\text{Change}}(= 5000)$. This procedure is calculated for 100 episodes.

Next, the numbers of epochs needed to train/fine-tune the model to obtain the weights ω_i from ω_{i-1} are averaged over the episodes and this number of epochs is used during inference to obtain the weights ω_i from ω_{i-1}.

This can be seen as a meta-model. As a result no test set is needed during inference to get the best point of stopping the training/fine-tuning. These numbers of epochs to train/fine-tune can be seen in the figures 5.1, 5.2 and in appendix A.2.

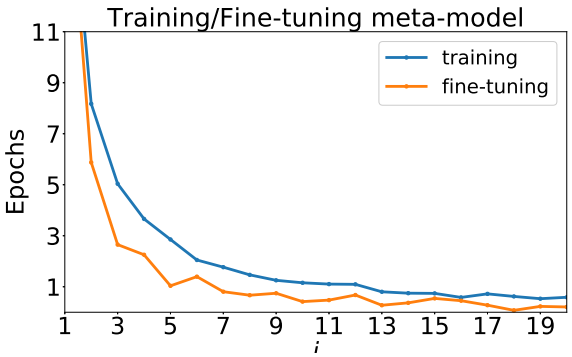

Figure 5.1.: Training/Fine-tuning meta-model 1: Number of epochs (in 1k) used for the training method comparison experiment (figures 5.3 and following).

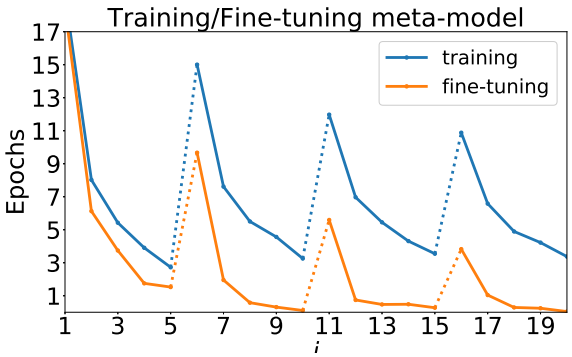

Figure 5.2.: Training/Fine-tuning meta-model 2: Number of epochs (in 1k) used for the input distribution change experiment (figures 5.7 and following).

One can see that the number of epochs decreases for larger i (if there is no distribution change in the data) and for the fine-tuning method there are less epochs required compared to the training method.

Note that the k in this context can be different to the k in the context of our own methods since we can evaluate our methods independently from the k used during training.

5.2. Comparison between training methods

Training method 1:

The weights of the feature extractor are fixed since training the weights of the feature extractor jointly with the meta-model gave no performance benefit for training method 1. A reason could be that the meta-model is trained with the miniImageNet data and not the full imageNet data.

Training method 1.1:

The training method 1.1 is evaluated for $k \in \{5, 10, 20\}$ and $|\mathcal{E}_2| \in 4 \cdot \{10, 40, 105, 230, 480\}$, i.e. 10,40,105,230 or 480 training images per class. The value $k = 20$ with 480 training images per class performed best.

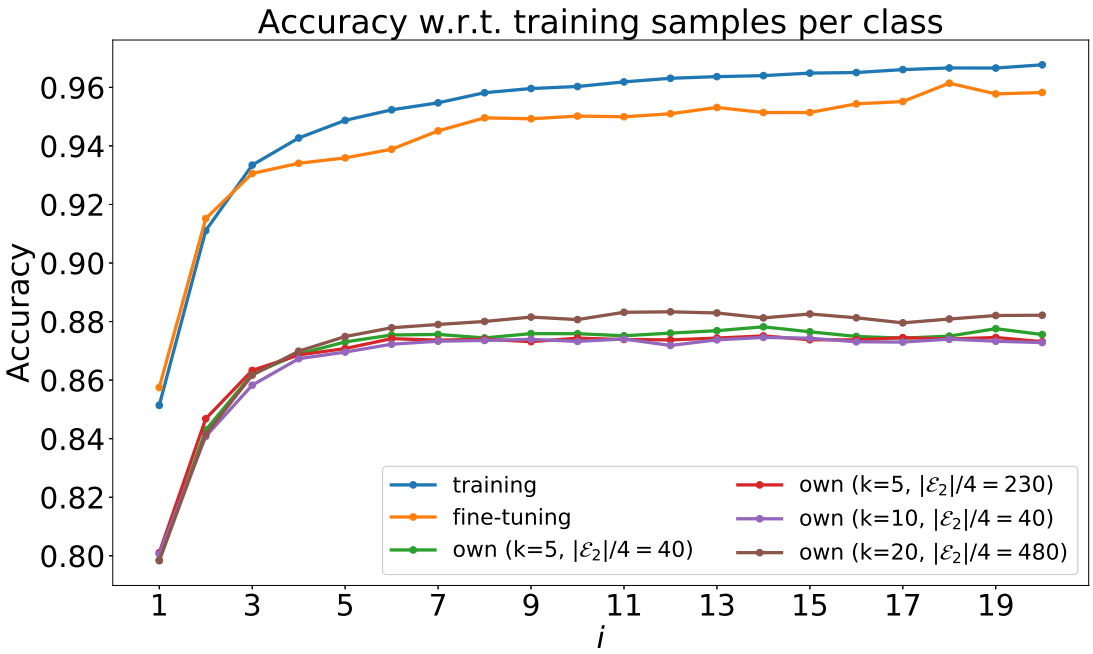

Figure 5.3.: Evaluation of training method 1.1

One can see that the usage of more data delivers slightly better accuracy but training method 1.1 is substantially worse than fine-tuning.

Training method 1.2:

The training method 1.2 is performed for $k \in \{5, 10, 20\}$.

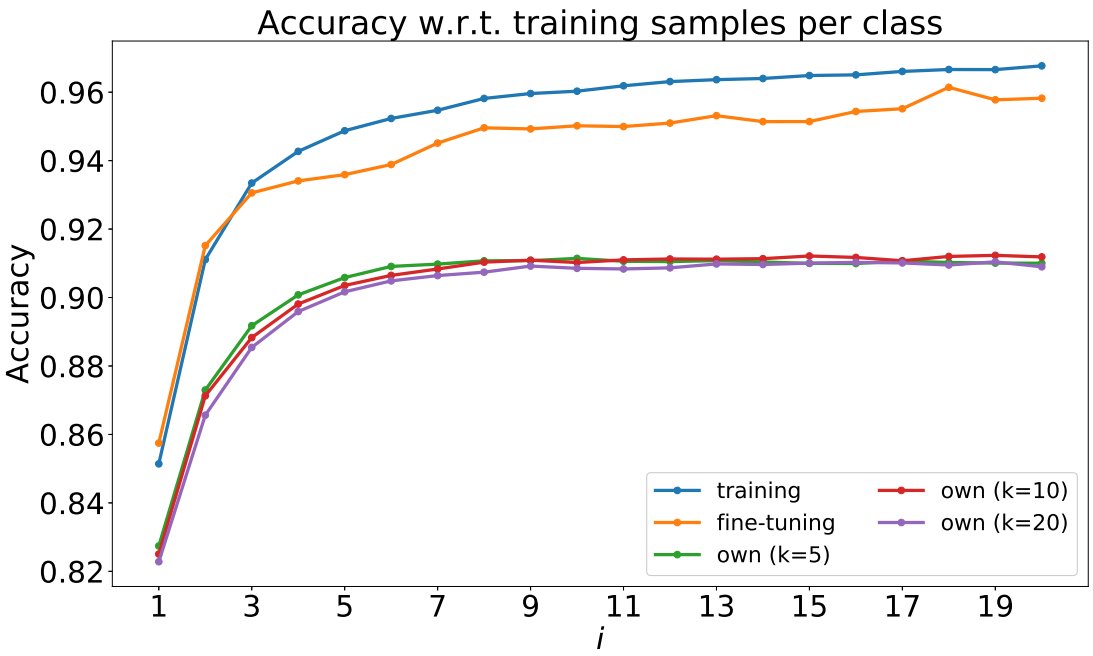

Figure 5.4.: Evaluation of training method 1.2

Compared to training method 1.1 this method performs significantly better, but is still worse than fine-tuning. Furthermore, there is little difference between different values of k.

Training methods 2 and 3:

These training methods are also evaluated for $k \in \{5, 10, 20\}$ (see figure 5.5). For training method 2 and 3.2 the value $k = 5$ performed best. For training method 3.1 the values $k = 5$ and $k = 20$ performed similarly well and better than $k = 10$.

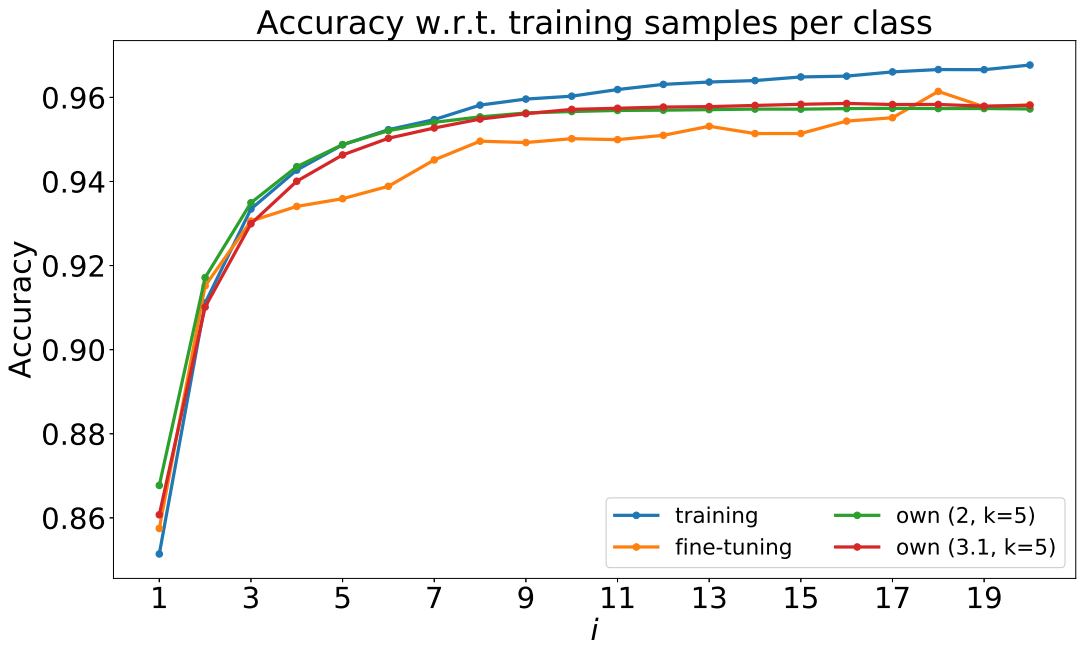

Figure 5.5.: Evaluation of training methods 2 and 3

We obtain that training methods 2 and 3.1 work similarly well and better than fine-tuning. Due to the bootstrapping mentioned in section 4.5 training method 2 needs approximately an order of magnitude more training steps than training method 3.1. Training method 3.2 has a worse performance than the other two (see appendix A.3).

5.3. Distribution change in the input data

A very interesting aspect of incremental learning is the case where the distribution of the input data changes during the incremental-learning process. This can happen, for instance, when the classes consist of multiple subclasses and in the first few training steps the given samples belong to the first subclasses and afterwards to other subclasses.

In the following such an experiment is performed. Each of the $n = 4$ classes consists of four subclasses which are assembled randomly. The learning process is arranged such that in the first five learning steps the samples are chosen from the first subclasses of each class. For the second five learning steps the samples are chosen from the second subclasses of each class and so forth. After five learning steps the input distribution of the samples changes. An illustrative example is given in section 5.3.1.

In this experiment we additionally compare our method to fine-tuning where additional connections and neurons are added similar to [WSS89]. Whenever the input distribution changes a hidden layer with two neurons is added, employed with a ReLU activation

function (see figure 5.6). The number of weights in the original network is approximately twice the number of weights added when adding a new hidden layer.

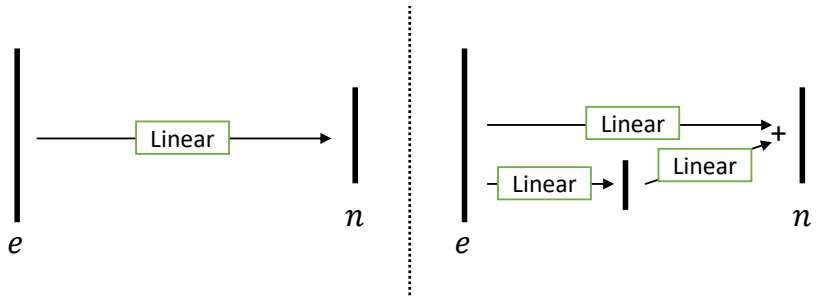

Figure 5.6.: A vertical line denotes a vector. Left: Original base-model: Linear transformation from the feature vector dimension $e = 512$ to the number of classes $n = 4$, Right: Additional hidden layer added. The output of the original transformation and the output of the new hidden layer(s) are summed.

We tried two versions of this approach: A parameter isolation-based method (fine-tuning + GLUE + freeze), where only the last added (hidden) layer is fine-tuned and the others are frozen, and a version (fine-tuning + GLUE) where all weights are fine-tuned. The first version performed poorly, the second version performed approximately as well as the fine-tuning method (see appendix A.3). The additional connections and neurons did not have a positive impact on the performance. A reason could be that during fine-tuning not enough data is available to leverage the increased capacity of the network.

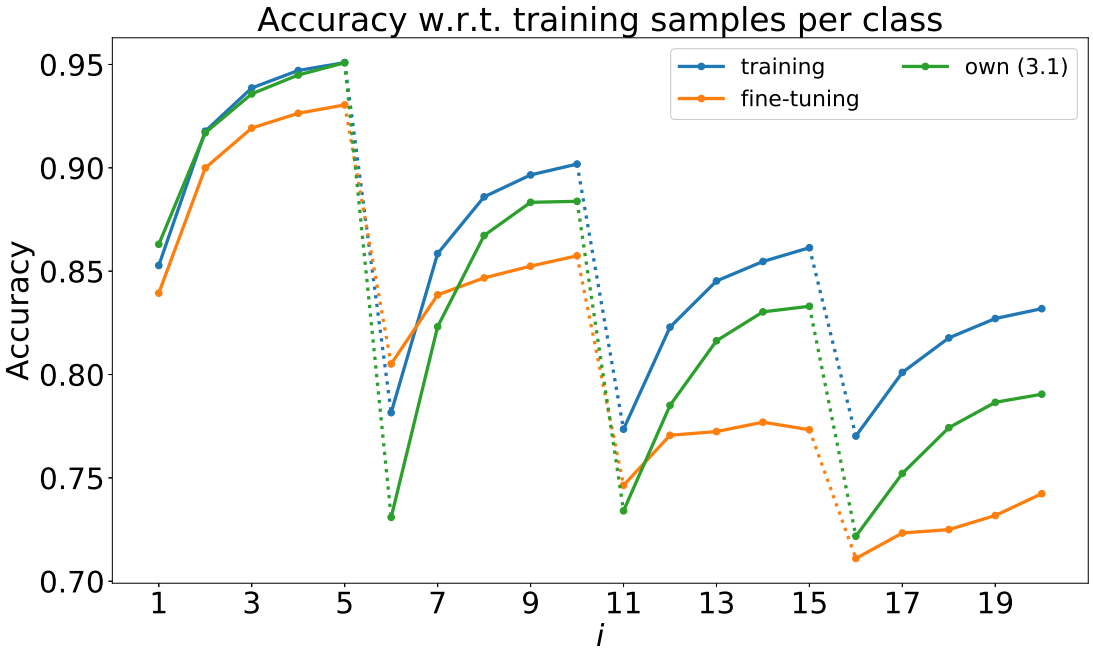

Figure 5.7.: Evaluation of distribution change in the input data: Accuracy evaluated on the subclasses of which samples are used as input to the meta-model, i.e. for $i \in \{1, \ldots, 5\}$ evaluated on the first subclasses of each class, for $i \in \{6, \ldots, 10\}$ evaluated on the first and second subclasses of each class and so forth.

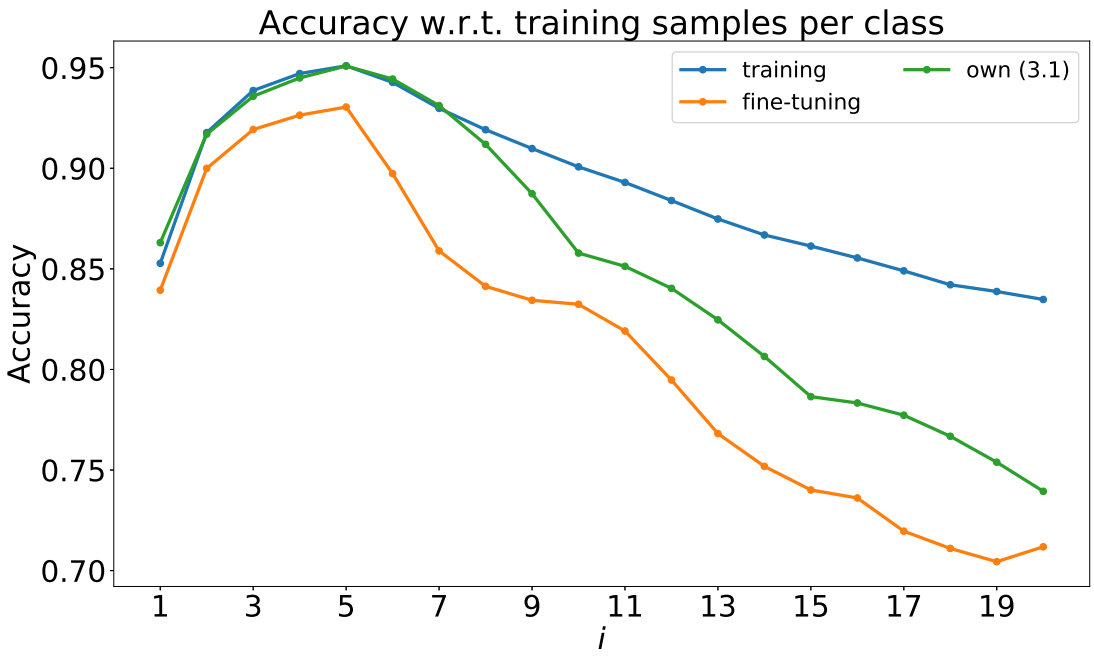

Figure 5.8.: Evaluation of distribution change in the input data: Accuracy evaluated on the first subclass of each class.

The accuracy of the training method decreases for $i > 5$ since the base-model has to deal with multiple subclasses per class but has the same capacity. For $i = 6$ the fine-tuning method performed best in figure 5.7. This can occur due to the fact that the fine-tuning method adapts rapidly to the new task and the accuracy is calculated as the mean of both tasks accuracy's. For the training methods (and our training method 3.1 which is derived from the training method) the training data is not balanced, this effect reduces for $i \in \{7, \ldots, 10\}$. We obtain that there is a catastrophic forgetting [Fre99] for $i > 5$ of the old data for the fine-tuning method (see figure 5.8) as expected. Our own method performs better and the accuracy is between the training method (which does not learn incrementally) and the fine-tuning method.

5.3.1. Illustrative example

Let's say we want to classify the following classes (of the CIFAR-dataset) which are divided into subclasses:

classes	subclasses 1	subclasses 2	subclasses 3	subclasses 4
vehicles	bicycle	pickup truck	motorcycle	bus
people	boy	woman	man	girl
large carnivores	bear	wolf	tiger	lion
food containers	bottle	plate	cup	can

Table 5.1.: Illustrative example: Classes

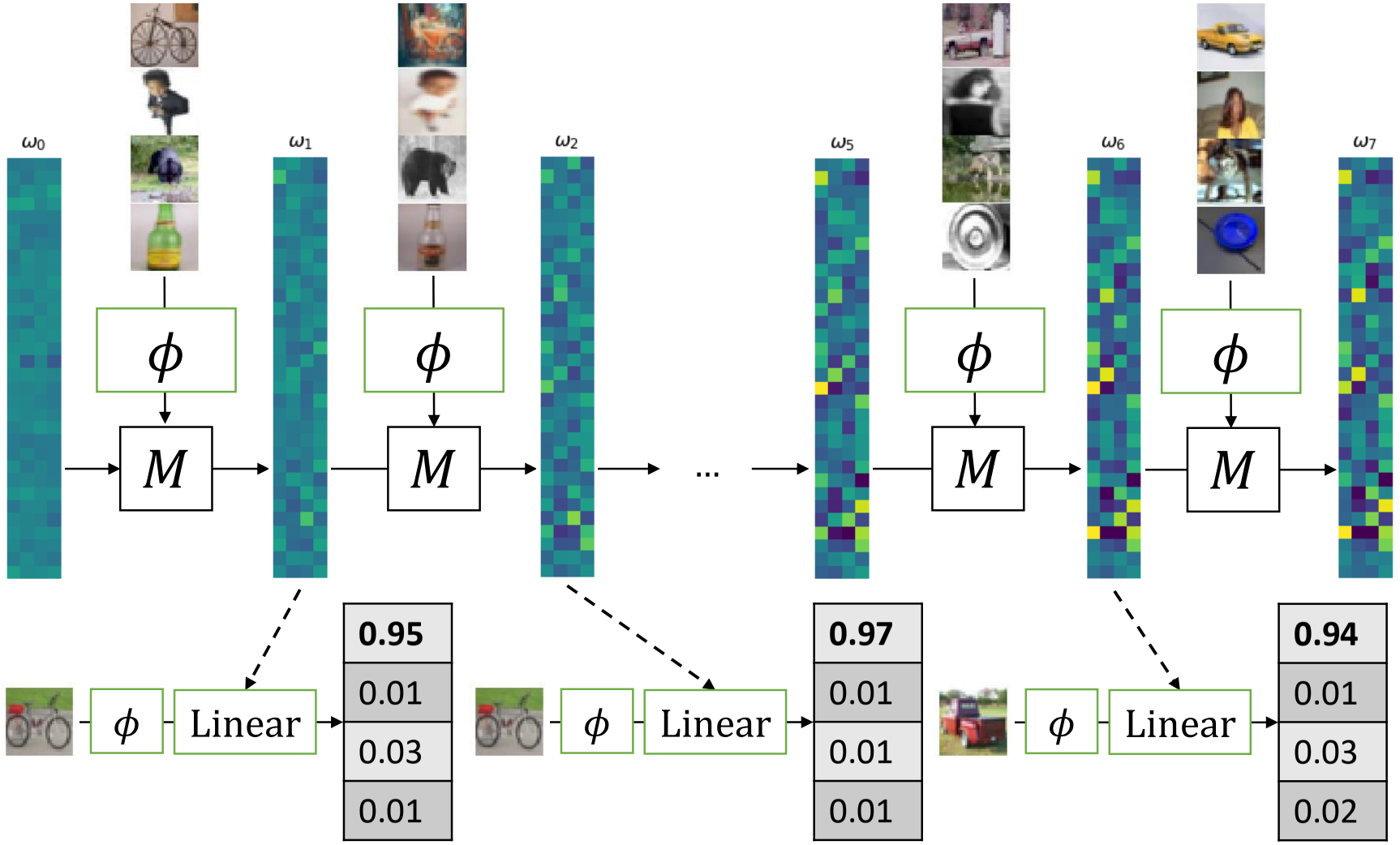

Figure 5.9.: Illustrative example: Incremental learning during inference. ϕ denotes the feature extractor, M denotes the mapping-model, $\omega_0 = \rho_1$ are learned weights of the meta-model, the plotted weights are a subset of all weights of the base-model.

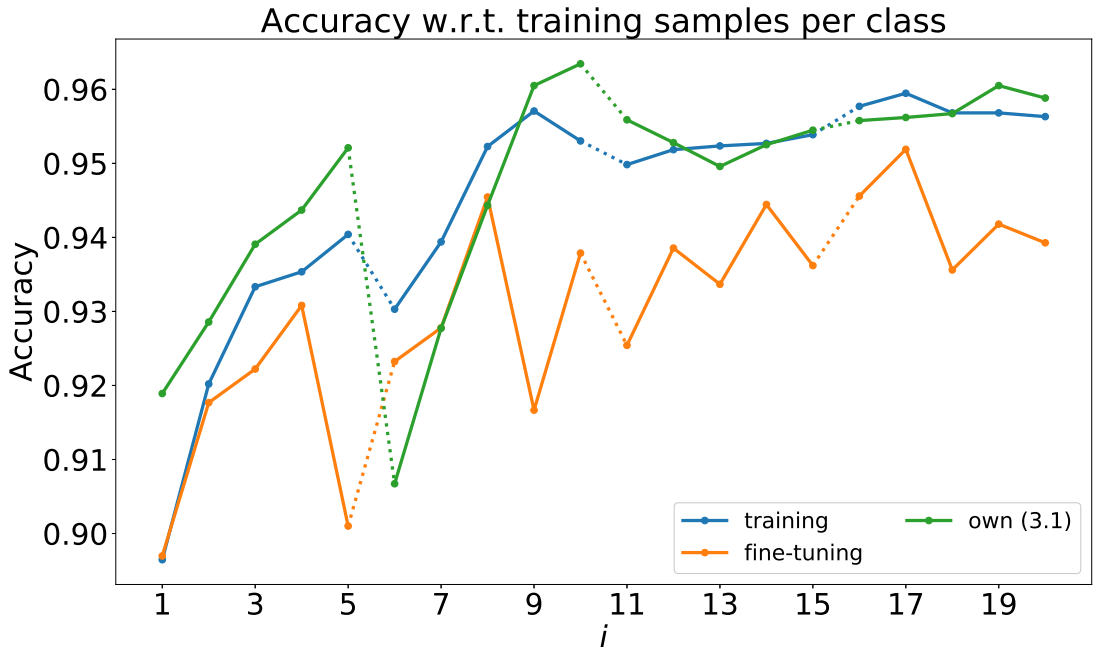

Figure 5.10.: Illustrative example: Accuracy evaluated as in figure 5.7. In this example CIFAR images are used for the evaluation.

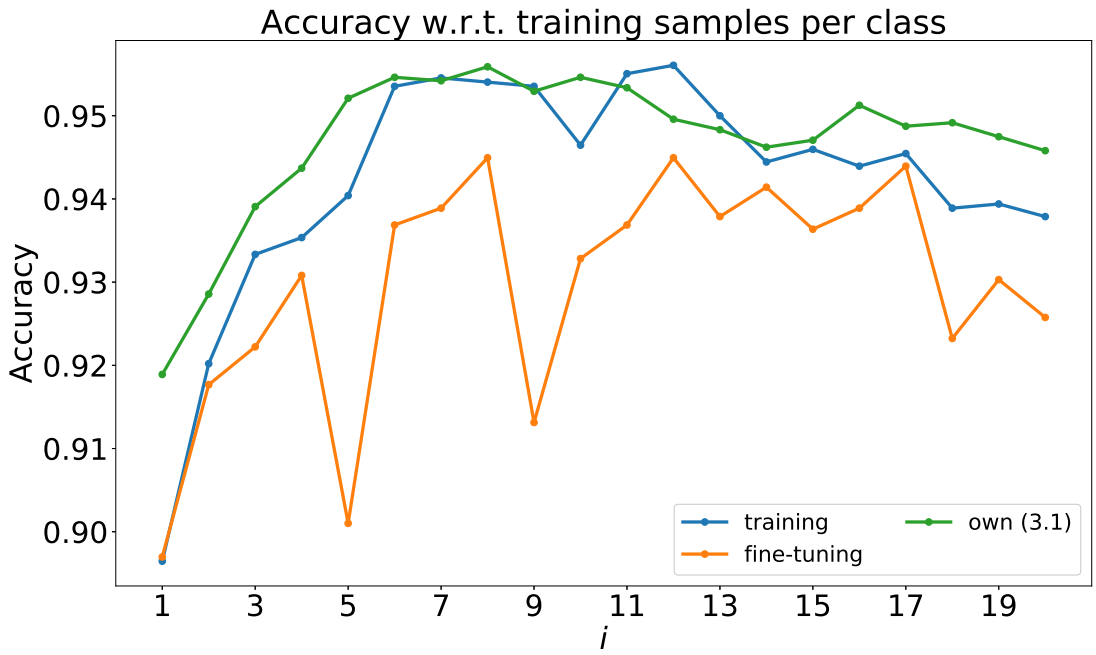

Figure 5.11.: Illustrative example: Accuracy evaluated on the first subclass of each class. In this example CIFAR images are used for the evaluation.

We obtain that in this specific example our method works better than the fine-tuning method, in some parts of the incremental-learning process even better than the training method. This is possible since our method has access to more data during training of the meta-model than the training method.

5.4. More complex base-models

In the previous sections the base-model is a linear transformation. Therefore, only linearly separable data is able to be classified. To address this issue a more complex base-model is used. A linear transformation with 128 output neurons is applied to the output of the feature extractor, followed by a ReLU activation function and another linear transformation with $n = 4$ output neurons.

Among other approaches we tried the following mapping-model (similar to equation (4.2)) which maps the weights of the two linear transformations together with the input features to other weights of the two linear transformations:

$$\begin{aligned} s_{\text{enc}} &= \text{encode}(s_i), \\ [w_{1,\text{out}}, w_{2,\text{out}}] &= \text{decode}(\text{context} = [\omega_{1,\text{in}}, \omega_{2,\text{in}}], \text{memory} = s_{\text{enc}}) + [0, s_{\text{enc}}], \end{aligned}$$

where s_i contains the feature vectors. The weights of the first and second linear transformation of the base-model are denoted by ω_1 and ω_2, respectively. An additional subscript in/out denotes the input/output of the mapping-model.

For this combination of base-model and meta-model training method 1.2 attained 86.2% accuracy, the training methods 1.1 and 3.2 attained 51.8% and 75.6% accuracy, respectively. The training methods 2 and 3.1 did not learn anything useful. For comparison, the training and fine-tuning methods attained 95.4% and 93.8% accuracy, respectively. It seems that the used mapping-model is not capable of mapping the weights appropriately, compared to the training and fine-tuning methods.

6. Conclusion

6.1. Review

We proposed a new incremental-learning scheme for image classification which requires only little data during inference and does not store samples. It uses a meta-learning approach. A mapping-model is learned to adjust weights of a classification model when new data is available to improve the performance of the system.

The mapping-model we use is based on the transformer architecture. To learn the system, different training methods have been compared and analyzed.

For a linear base-model our approach outperforms fine-tuning, for more complex base-models it also works, but together with the used mapping-model it is not as good as fine-tuning.

6.2. Future work

The main point of future work is to use base-models with more layers and to find advanced mapping-models which works with them. More complex base-models are needed to increase the amount of knowledge that can be stored.

The main focus of this work lies in incremental learning. However, one could also be interested in the other direction, i.e. one does not want to learn specific knowledge but selectively forget knowledge. This could happen if the system recognizes or is told that some of the acquired knowledge is outdated or wrong.

A similar approach to the one proposed in this work could be applied. Instead of learning a mapping-model to map to weights where new knowledge is added, one could train a mapping-model which maps to weights where the knowledge to forget is removed. This allows for a selective forgetting.

Bibliography

[And+16] Marcin Andrychowicz et al. “Learning to learn by gradient descent by gradient descent”. In: *Advances in neural information processing systems*. 2016, pp. 3981–3989.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[Ber+16] Luca Bertinetto et al. “Learning feed-forward one-shot learners”. In: *Advances in neural information processing systems*. 2016, pp. 523–531.

[Ber+18] Luca Bertinetto et al. “Meta-learning with differentiable closed-form solvers”. In: *arXiv preprint arXiv:1805.08136* (2018).

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[Cho+14] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[De +19] Matthias De Lange et al. “Continual learning: A comparative study on how to defy forgetting in classification tasks”. In: *arXiv preprint arXiv:1909.08383* (2019).

[Den+09] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[Elm90] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. “Model-agnostic meta-learning for fast adaptation of deep networks”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1126–1135.

[Fre99] Robert M French. “Catastrophic forgetting in connectionist networks”. In: *Trends in cognitive sciences* 3.4 (1999), pp. 128–135.

[GB10] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[Goo+14] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[He+15] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hsu+18] Yen-Chang Hsu et al. "Re-evaluating continual learning scenarios: A categorization and case for strong baselines". In: *arXiv preprint arXiv:1810.12488* (2018).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KH10] Alex Krizhevsky and Geoff Hinton. "Convolutional deep belief networks on cifar-10". In: *Unpublished manuscript* 40.7 (2010), pp. 1–9.

[Kir+17] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LH17] Zhizhong Li and Derek Hoiem. "Learning without forgetting". In: *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017), pp. 2935–2947.

[LJY17] Fei-Fei Li, Justin Johnson, and Serena Yeung. *Lecture 10: Recurrent Neural Networks*. 2017. URL: `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf` (visited on 01/21/2020).

[LR17] David Lopez-Paz and Marc'Aurelio Ranzato. "Gradient episodic memory for continual learning". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6467–6476.

[ML18] Arun Mallya and Svetlana Lazebnik. "Packnet: Adding multiple tasks to a single network by iterative pruning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7765–7773.

[MP43] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[Raj+19] Jathushan Rajasegaran et al. “Random path selection for incremental learning”. In: *arXiv preprint arXiv:1906.01120* (2019).

[Reb+17] Sylvestre-Alvise Rebuffi et al. “icarl: Incremental classifier and representation learning”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2017, pp. 2001–2010.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[RL16] Sachin Ravi and Hugo Larochelle. “Optimization as a model for few-shot learning”. In: (2016).

[Ros58] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[Rus+15] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[Shi+17] Hanul Shin et al. “Continual learning with deep generative replay”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 2990–2999.

[Sil+17a] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[Sil+17b] David Silver et al. “Mastering the game of go without human knowledge”. In: *Nature* 550.7676 (2017), pp. 354–359.

[Sri+14] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[Tel16] Matus Telgarsky. “Benefits of depth in neural networks”. In: *arXiv preprint arXiv:1602.04485* (2016).

[Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[Vin16] O Vinyals. “Matching networks for one shot learning. Vinyals O., Blundell C., Lillicrap T. Kavukcuoglu K. & Wierstra D.(2016). Matching networks for one shot learning”. In: *Neural Information Processing Systems conference, Barcelona, Spain, December*. 2016, pp. 5–10.

[VT18] Gido M van de Ven and Andreas S Tolias. “Generative replay with feedback connections as a general strategy for continual learning”. In: *arXiv preprint arXiv:1809.10635* (2018).

[Wai+89] Alex Waibel et al. “Phoneme recognition using time-delay neural networks”. In: *IEEE transactions on acoustics, speech, and signal processing* 37.3 (1989), pp. 328–339.

[Wer+90] Paul J Werbos et al. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[WSS89] Alex Waibel, Hidefumi Sawai, and Kiyohiro Shikano. “Modularity and scaling in large phonemic neural networks”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.12 (1989), pp. 1888–1898.

[Zha+20] Junting Zhang et al. “Class-incremental learning via deep model consolidation”. In: *The IEEE Winter Conference on Applications of Computer Vision*. 2020, pp. 1131–1140.

List of Figures

List of Tables

A. Appendix

A.1. Hyperparameters

Transformer model: $d_{\text{model}} = 512$, $d_{ff} = 2048$ with $N = 6$ layers and $h = 8$ heads.

Dropout base-model (before the linear transformation(s)): 0.5, dropout mapping-model: 0.3 (0.1 for the two-layer base-model experiment)

Batch size: The batch size is maximized such that the available memory of the graphics card is not exceeded.

Optimizer and learning rate: The optimizer is adopted from [Vas+17], the learning rate is chosen similarly, but with a factor 0.2 and $warmup\_steps = 133$. The learning process is stopped after the validation error increases.

Features: The feature vectors of the input images are precomputed if the training method allows.

Evaluation: The accuracies shown in chapter 5 are averaged over 1024 episodes for our own methods and over 100 episodes for the training/fine-tuning methods. The precomputation step is done for 1000 episodes.

Number of classes: $n = 4$ classes are used since the mini-ImageNet validation dataset consists of 16 classes and with 4 subclasses per class in one experiment, the maximum number of classes that can be used with this dataset is 4.

A.2. Training/Fine-tuning meta-models

The following numbers of epochs are used in the meta-models of the training and fine-tuning method:

Training:

First experiment (no change in input distribution):

19600, 8172, 5037, 3659, 2854, 2049, 1767, 1466, 1252, 1156, 1104, 1094, 802, 748, 740, 582, 721, 620, 531, 587

Change in input distribution after every fifth learning step:

19209, 8021, 5424, 3906, 2724, 15005, 7614, 5502, 4563, 3256, 11974, 6969, 5450, 4311, 3547, 10880, 6582, 4895, 4221, 3372

Two-layer base-model (no change in input distribution):

10615, 6212, 4094, 2942, 2124, 1700, 1517, 1305, 978, 880, 843, 755, 678, 827, 609, 560, 564, 459, 515, 457

Fine-tuning:

First experiment (no change in input distribution):

19270, 5874, 2652, 2254, 1037, 1396, 806, 668, 745, 417, 476, 676, 275, 369, 543, 453, 277, 75, 230, 207

Change in input distribution after every fifth learning step:

18445, 6127, 3742, 1751, 1526, 9669, 1954, 580, 311, 107, 5591, 745, 472, 484, 274, 3825, 1042, 290, 243, 42

Fine-tuning + GLUE + freeze (change in input distribution): 18755, 4970, 3774, 2390, 2182, 3013, 650, 34, 5, 3, 563, 125, 46, 34, 37, 807, 143, 146, 308, 37

Fine-tuning + GLUE (change in input distribution): 18223, 4839, 3216, 2450, 1915, 9738, 1060, 1198, 288, 250, 4982, 1042, 261, 134, 182, 2437, 1246, 420, 204, 178

Two-layer base-model (no change in input distribution):

10954, 4327, 2630, 1286, 1303, 794, 1082, 550, 392, 225, 469, 384, 330, 361, 129, 313, 328, 302, 240, 212

A.3. Detailed results

A.3.1. Comparison between training methods

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.1	91.1	93.3	94.3	94.9	95.2	95.5	95.8	96.0	96.0	96.2	96.3	96.4	96.4	96.5	96.5	96.6	96.7	96.7	96.8	95.1
fine-tuning	85.7	91.5	93.1	93.4	93.6	93.9	94.5	95.0	94.9	95.0	95.0	95.1	95.3	95.1	95.1	95.4	95.5	96.1	95.8	95.8	94.3

Table A.1.: Results: Training/Fine-tuning methods

Own methods:

k	data \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
5	10	79.1	83.4	85.3	85.7	85.8	86.1	85.9	85.9	86.0	86.1	86.2	86.2	86.0	86.2	86.0	86.0	86.1	86.2	86.1	86.1	85.5
5	40	79.9	84.3	86.2	86.9	87.3	87.5	87.6	87.4	87.6	87.6	87.5	87.6	87.7	87.8	87.7	87.5	87.4	87.5	87.8	87.6	86.9
5	105	79.6	83.8	85.6	86.0	86.5	86.4	86.6	86.8	86.9	86.7	86.8	86.7	86.6	86.8	86.9	86.7	86.7	86.8	86.6	86.7	86.1
5	230	**80.1**	**84.7**	**86.3**	86.9	87.1	87.4	87.4	87.4	87.3	87.4	87.4	87.4	87.4	87.5	87.4	87.4	87.5	87.4	87.5	87.3	86.8
5	480	79.7	84.2	86.1	86.8	86.9	86.9	87.0	87.1	87.0	87.2	87.2	87.2	87.0	86.9	87.0	87.1	87.1	87.2	87.0	87.0	86.5
10	10	78.3	82.4	84.1	84.9	85.1	85.5	85.4	85.3	85.2	85.3	85.1	85.2	85.3	85.4	85.3	85.3	85.5	85.4	85.4	85.6	84.7
10	40	**80.1**	84.1	85.8	86.7	87.0	87.2	87.3	87.4	87.4	87.3	87.4	87.2	87.4	87.5	87.4	87.3	87.3	87.4	87.3	87.3	86.7
10	105	78.7	83.4	85.3	86.2	86.6	86.8	86.8	86.9	86.8	86.9	86.9	87.1	86.9	86.7	86.8	87.0	86.9	86.9	87.0	87.0	86.2
10	230	79.4	83.8	85.8	86.4	86.9	86.9	87.0	87.0	87.1	87.1	87.2	87.2	87.1	87.3	87.2	87.2	87.4	87.4	87.2	87.2	86.5
10	480	79.6	83.8	85.7	86.6	86.9	87.2	87.3	87.4	87.4	87.4	87.5	87.6	87.4	87.4	87.5	87.4	87.5	87.5	87.5	87.4	86.7
20	10	78.7	83.1	84.7	85.6	85.9	86.2	86.2	86.3	86.3	86.2	86.2	86.4	86.4	86.4	86.3	86.4	86.3	86.3	86.4	86.4	85.6
20	40	79.2	83.5	85.2	86.0	86.4	86.5	86.5	86.8	86.7	86.5	86.7	86.7	86.8	86.7	86.7	86.6	86.7	86.6	86.8	86.8	86.0
20	105	78.4	83.4	85.2	86.2	86.6	86.7	86.8	86.9	87.0	86.9	87.1	87.1	87.2	87.0	86.9	87.1	87.1	87.1	87.1	87.1	86.3
20	230	79.8	84.1	85.8	86.6	87.0	87.0	87.1	87.2	87.1	87.3	87.3	87.0	87.1	87.1	87.2	87.1	87.0	87.0	87.1	87.0	86.5
20	480	79.8	84.1	86.2	**87.0**	**87.5**	**87.8**	**87.9**	**88.0**	**88.2**	**88.1**	**88.3**	**88.3**	**88.3**	**88.1**	**88.3**	**88.1**	**88.0**	**88.1**	**88.2**	**88.2**	**87.3**

Table A.2.: Results: Training method 1.1

k \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
5	**82.7**	**87.3**	**89.2**	**90.1**	**90.6**	**90.9**	**91.0**	**91.1**	**91.1**	**91.1**	**91.1**	**91.1**	**91.1**	91.0	91.0	91.0	**91.1**	91.0	91.0	91.0	**90.3**
10	82.5	87.1	88.8	89.8	90.4	90.6	90.8	91.0	**91.1**	91.0	**91.1**	**91.1**	**91.1**	**91.1**	**91.2**	**91.2**	**91.1**	**91.2**	**91.2**	**91.2**	90.2
20	82.3	86.6	88.5	89.6	90.2	90.5	90.6	90.7	90.9	90.9	90.8	90.9	91.0	91.0	91.0	91.0	91.0	91.0	91.0	90.9	90.0

Table A.3.: Results: Training method 1.2

k \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
5	**86.8**	**91.7**	**93.5**	**94.4**	**94.9**	**95.2**	**95.4**	**95.5**	**95.6**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**95.7**	**94.8**
10	86.2	91.2	93.1	94.1	94.7	95.0	95.2	95.4	95.5	95.6	95.6	95.6	95.6	**95.7**	**95.7**	95.6	95.6	95.6	95.6	95.6	94.6
20	85.9	90.6	92.5	93.5	94.1	94.4	94.6	94.8	94.8	94.9	94.9	94.9	94.9	94.8	94.7	94.7	94.6	94.5	94.5	94.4	93.9

Table A.4.: Results: Training method 2

k \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
5	**86.1**	**91.0**	**93.0**	**94.0**	**94.6**	**95.0**	**95.3**	**95.5**	**95.6**	**95.7**	**95.7**	**95.8**	**95.8**	**95.8**	95.8	**95.9**	95.8	95.8	95.8	95.8	**94.7**
10	85.2	90.2	92.2	93.3	94.0	94.4	94.8	95.0	95.2	95.3	95.4	95.5	95.5	95.6	95.6	95.6	95.7	95.7	95.7	95.7	94.3
20	85.1	89.9	91.9	93.1	93.7	94.2	94.6	94.9	95.1	95.3	95.5	95.6	95.7	**95.8**	**95.9**	**95.9**	**96.0**	**96.0**	**96.1**	**96.1**	94.3

Table A.5.: Results: Training method 3.1

k \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
5	82.1	**86.3**	**87.7**	**88.2**	**88.5**	**89.0**	**89.1**	**89.3**	**89.4**	**89.3**	**89.3**	**89.3**	**89.3**	**89.1**	**89.2**	**89.1**	**89.2**	**89.2**	**89.1**	**89.2**	**88.6**
10	**82.2**	86.2	87.0	87.7	87.7	87.8	87.9	87.8	88.1	87.9	87.9	88.1	88.1	88.0	88.1	88.1	87.9	87.9	88.0	87.9	87.5
20	81.9	85.7	87.5	88.0	88.4	88.8	88.8	88.7	88.7	88.8	88.8	88.8	88.9	88.8	89.0	89.0	88.9	88.8	88.8	88.9	88.2

Table A.6.: Results: Training method 3.2

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.1	91.1	93.3	94.3	94.9	95.2	95.5	95.8	96.0	96.0	96.2	96.3	96.4	96.4	96.5	96.5	96.6	96.7	96.7	96.8	95.1
fine-tuning	85.7	91.5	93.1	93.4	93.6	93.9	94.5	95.0	94.9	95.0	95.0	95.1	95.3	95.1	95.1	95.4	95.5	**96.1**	95.8	95.8	94.3
2, k=5	**86.8**	**91.7**	**93.5**	**94.4**	**94.9**	**95.2**	**95.4**	**95.5**	**95.6**	**95.7**	**95.7**	95.7	95.7	95.7	95.7	95.7	95.7	95.7	95.7	95.7	**94.8**
3.1, k=5	86.1	91.0	93.0	94.0	94.6	95.0	95.3	**95.5**	**95.6**	**95.7**	**95.7**	**95.8**	**95.8**	**95.8**	95.8	**95.9**	95.8	95.8	95.8	95.8	**94.7**
3.1, k=20	85.1	89.9	91.9	93.1	93.7	94.2	94.6	94.9	95.1	95.3	95.5	95.6	95.7	**95.8**	**95.9**	**95.9**	**96.0**	96.0	**96.1**	**96.1**	94.3

Table A.7.: Results: Best-to-best comparison

A.3.2. Distribution change in the input data

For each method in this section there are two tables. In the first one the accuracy is evaluated on data from the subclasses of the classes already seen as input to the meta-model. In the second one the accuracy is evaluated only on data from the first subclasses of the classes and therefore it can be evaluated how much of the old data each method forgets.

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.3	91.8	93.9	94.7	95.1	78.2	85.9	88.6	89.7	90.2	77.4	82.3	84.5	85.5	86.1	77.0	80.1	81.8	82.7	83.2	85.7
fine-tuning	83.9	90.0	91.9	92.6	93.0	80.5	83.9	84.7	85.2	85.7	74.6	77.1	77.2	77.7	77.3	71.1	72.3	72.5	73.2	74.2	80.9
fine-tuning + GLUE + freeze	85.1	90.5	92.0	92.8	93.3	37.1	41.2	42.1	42.2	42.2	32.1	31.0	31.0	30.8	30.9	28.1	28.7	29.4	30.8	31.1	48.1
fine-tuning + GLUE	84.4	90.1	92.3	93.1	93.5	78.8	82.3	83.2	84.1	84.0	74.9	77.2	76.6	77.8	77.6	70.3	71.5	72.1	71.2	71.0	80.3

Table A.8.: Results: Training/Fine-tuning methods(1)

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.3	91.8	93.9	94.7	95.1	94.3	93.0	91.9	91.0	90.1	89.3	88.4	87.5	86.7	86.1	85.5	84.9	84.2	83.9	83.5	89.0
fine-tuning	83.9	90.0	91.9	92.6	93.0	89.7	85.9	84.1	83.4	83.2	81.9	79.5	76.8	75.2	74.0	73.6	72.0	71.1	70.4	71.2	81.2
fine-tuning + GLUE + freeze	85.1	90.5	92.0	92.8	93.3	24.5	24.1	24.4	24.5	24.5	25.3	24.7	25.1	25.1	25.3	24.7	24.8	25.1	25.9	26.1	41.4
fine-tuning + GLUE	84.4	90.1	92.3	93.1	93.5	88.5	85.1	81.6	81.6	80.8	78.3	75.4	74.1	74.5	73.5	72.6	69.4	68.4	66.5	65.5	79.5

Table A.9.: Results: Training/Fine-tuning methods(2)

Own methods:

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
2	86.0	91.5	93.1	94.2	94.7	**73.2**	81.6	85.5	86.7	86.6	71.2	75.1	77.6	78.6	78.6	67.9	69.9	71.1	71.4	71.1	80.3
3.1	**86.3**	**91.7**	**93.6**	**94.5**	**95.1**	73.1	**82.3**	**86.7**	**88.3**	**88.4**	**73.4**	**78.5**	**81.6**	**83.0**	**83.3**	**72.2**	**75.2**	**77.4**	**78.7**	**79.0**	**83.1**
3.2	78.9	84.8	86.5	87.3	88.0	69.2	67.0	62.9	60.6	59.2	56.0	54.3	51.5	49.8	48.6	48.5	47.3	45.3	43.9	42.9	61.6

Table A.10.: Results: Own methods(1)

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
2	86.0	91.5	93.1	94.2	94.7	93.7	92.1	89.8	87.0	83.8	83.0	81.5	79.6	77.5	75.1	74.3	73.0	71.3	69.3	67.2	82.9
3.1	**86.3**	**91.7**	**93.6**	**94.5**	**95.1**	**94.4**	**93.1**	**91.2**	**88.7**	**85.8**	**85.1**	**84.0**	**82.5**	**80.6**	**78.6**	**78.3**	**77.7**	**76.7**	**75.4**	**73.9**	**85.4**
3.2	78.9	84.8	86.5	87.3	88.0	72.4	53.3	41.0	34.2	30.6	29.3	28.0	26.8	26.0	25.7	25.6	25.9	26.0	25.8	25.8	46.1

Table A.11.: Results: Own methods(2)

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.3	91.8	93.9	94.7	95.1	78.2	85.9	88.6	89.7	90.2	77.4	82.3	84.5	85.5	86.1	77.0	80.1	81.8	82.7	83.2	85.7
fine-tuning	83.9	90.0	91.9	92.6	93.0	**80.5**	**83.9**	84.7	85.2	85.7	**74.6**	77.1	77.2	77.7	77.3	71.1	72.3	72.5	73.2	74.2	80.9
3.1	**86.3**	**91.7**	**93.6**	**94.5**	**95.1**	73.1	82.3	**86.7**	**88.3**	**88.4**	73.4	**78.5**	**81.6**	**83.0**	**83.3**	**72.2**	**75.2**	**77.4**	**78.7**	**79.0**	**83.1**

Table A.12.: Results: Best-to-best comparison(1)

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	85.3	91.8	93.9	94.7	95.1	94.3	93.0	91.9	91.0	90.1	89.3	88.4	87.5	86.7	86.1	85.5	84.9	84.2	83.9	83.5	89.0
fine-tuning	83.9	90.0	91.9	92.6	93.0	89.7	85.9	84.1	83.4	83.2	81.9	79.5	76.8	75.2	74.0	73.6	72.0	71.1	70.4	71.2	81.2
3.1	**86.3**	**91.7**	**93.6**	**94.5**	**95.1**	**94.4**	**93.1**	**91.2**	**88.7**	**85.8**	**85.1**	**84.0**	**82.5**	**80.6**	**78.6**	**78.3**	**77.7**	**76.7**	**75.4**	**73.9**	**85.4**

Table A.13.: Results: Best-to-best comparison(2)

Illustrative example:

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	89.6	92.0	93.3	93.5	94.0	**93.0**	**93.9**	**95.2**	95.7	95.3	95.0	95.2	**95.2**	**95.3**	**95.4**	**95.8**	**95.9**	**95.7**	95.7	95.6	94.5
fine-tuning	89.7	91.8	92.2	93.1	90.1	92.3	92.8	94.5	91.7	93.8	92.5	93.9	93.4	94.4	93.6	94.6	95.2	93.6	94.2	93.9	93.1
3.1	**91.9**	**92.9**	**93.9**	**94.4**	**95.2**	90.7	92.8	94.4	**96.1**	**96.3**	**95.6**	**95.3**	95.0	**95.3**	**95.4**	95.6	95.6	**95.7**	**96.1**	**95.9**	**94.7**

Table A.14.: Results: Best-to-best comparison(3)

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	89.6	92.0	93.3	93.5	94.0	95.4	**95.5**	95.4	**95.4**	94.6	**95.5**	**95.6**	**95.0**	94.4	94.6	94.4	94.5	93.9	93.9	93.8	94.2
fine-tuning	89.7	91.8	92.2	93.1	90.1	93.7	93.9	94.5	91.3	93.3	93.7	94.5	93.8	94.1	93.6	93.9	94.4	92.3	93.0	92.6	93.0
3.1	**91.9**	**92.9**	**93.9**	**94.4**	**95.2**	**95.5**	95.4	**95.6**	95.3	**95.5**	95.3	95.0	94.8	**94.6**	**94.7**	**95.1**	**94.9**	**94.9**	**94.7**	**94.6**	**94.7**

Table A.15.: Results: Best-to-best comparison(4)

A.3.3. More complex base-models

Two-layer base-model (no change in input distribution):

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
training	84.3	91.6	93.7	94.7	95.2	95.7	95.9	96.1	96.3	96.4	96.5	96.6	96.7	96.8	96.8	96.9	96.9	97.0	97.0	97.0	95.4
fine-tuning	84.8	91.0	92.3	93.1	93.6	93.7	94.3	94.2	94.1	94.5	94.9	95.1	95.0	95.3	94.9	95.3	95.4	95.4	94.8	95.2	93.8

Table A.16.: Results: Training/Fine-tuning methods for the two-layer base-model

version \ i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	mean
1.1, (k=5,data=495)	47.3	49.6	50.9	51.8	52.1	52.2	52.3	52.3	52.2	52.3	52.2	52.2	52.3	52.2	52.2	52.3	52.2	52.2	52.1	52.1	51.8
1.2, k=5	78.6	83.3	85.4	86.4	86.8	87.1	87.0	87.1	87.1	87.2	87.0	86.9	86.8	86.9	86.7	86.7	86.7	86.6	86.7	86.7	86.2
3.2, k=5	75.9	75.2	75.6	75.6	75.7	75.8	75.9	75.7	75.2	75.1	75.2	75.6	76.0	75.8	75.9	75.8	75.4	74.9	75.6	75.4	75.6

Table A.17.: Results: Own methods for the two-layer base-model

Note that for the training methods 2 and 3.1 the mean squared error of both weight matrices is weighted such that the error is approximately equal. These methods did not learn anything useful, i.e. the accuracy was around $\frac{1}{n}$.