university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Master's Thesis

University of Groningen

Department of Artificial Intelligence

Improving Offline Handwritten Text Recognition Using Conditional Writer-Specific Knowledge

First Supervisor:
Prof. dr. L.R.B. Schomaker

Author:
Tobias van der Werff
s4314719

Second Supervisor:
M. Dhali

July 4, 2022

Abstract

Modern neural networks suffer from a major flaw: an inability to deal with changing data distributions. In the field of handwritten text recognition (HTR), this shows itself in poor recognition accuracy for writers that are not similar to those seen during training. In this research, the question is asked whether explicit information regarding writer identity can be used to condition a neural network to a writer-specific distribution. For example, writer information can be presented in the form of a small batch of labeled examples originating from a particular writer. Two state-of-the-art HTR architectures are used as baseline models (writer-unaware), using a ResNet backbone along with either an LSTM or Transformer sequence decoder. Using these base models, various methods are proposed to make them writer-adaptive, based primarily on 1) an idea originating from automatic speech recognition known as *speaker codes*, and 2) model-agnostic meta-learning (MAML), an algorithm commonly used for tasks such as few-shot classification that has gained much attention in recent years. Results show that an HTR-specific version of MAML known as MetaHTR improves performance compared to the baseline with a 1.4 to 2.0 improvement in word error rate. Furthermore, it is shown that a deeper model lends itself better to adaptation using MetaHTR than a shallower model. Speaker codes do not show a concrete benefit for writer-aware adaptation.

Keywords: offline handwritten text recognition, writer adaptation, few-shot adaptation, conditionality

Acknowledgements

Working on this thesis has been an exhilarating journey for me. It has made me realize how much fun research can be (even if it does not always lead to the conclusions you would have liked). I would like to thank Lambert Schomaker for his insightful comments and critical thought process, which often challenged me and forced me to question my assumptions. I am grateful for the various interesting conversations we had on the topic of deep learning and machine learning more broadly. I also would like to thank Maruf Dhali, who was always ready to help out and provide guidance whenever I needed it. Over the last few years that I have been working with Maruf, I have come to value him both as a teacher and as a friend. Lastly, I thank the Center for Information Technology of the University of Groningen for providing access to the Peregrine high performance computing cluster.

Contents

Chapter 1

Introduction

Deep learning has achieved tremendous success in recent years. However, neural networks are still lacking when it comes to adapting to novel environments. This manifests itself in an inability to deal with changing data distributions [58]. Most modern neural networks tend to be *one-trick ponies*, in the sense that they work very well for specific task and data conditions, but fail – sometimes catastrophically – when such conditions are violated. In some sense, this is understandable, as the high dimensionality of the input space for common applications such as images leads to an inconceivable amount of possible variations. Arguably, much of the modern success of neural networks can be attributed to collecting massive amounts of data to cover as many parts of the underlying data distribution as possible, combined with a proportional increase in computing power and model size [53].

However, such a brute-force approach to learning is not always practical, e.g., in domains where there is limited data for training a high-capacity neural network. In such cases, more efficient use of data and reusability of previously learned representations suddenly play an important role. An example of a possible solution is *transfer learning*, where previously learned model parameters are reused for a new but related task that has only a modest amount of training data. This approach has led to notable successes in fields such as natural language processing [22] and computer vision [74]. In the ideal case, neural networks should be able to autonomously *adapt* in a flexible manner to various input values. Or, put slightly differently, neural networks should have the ability to be properly *conditioned* on various possible input values.

In this thesis, we explore notions of conditionality and adaptivity in the realm of deep learning, zooming in on the field of offline handwritten text recognition (HTR) as a concrete use case. Offline handwritten text recognition refers to the process of automatically turning images of handwritten text into letter codes. Within the deep learning framework, this is presented as an image-to-sequence problem: A handwriting image (input) is converted into a sequence of characters (output) transcribing the handwritten text in the image.

Handwriting recognition, which encompasses a broader body of research than just text recognition, has seen major successes using deep learning. This includes areas such as handwritten text recognition [71, 3], writer identification [114, 41], binarization [23], and word

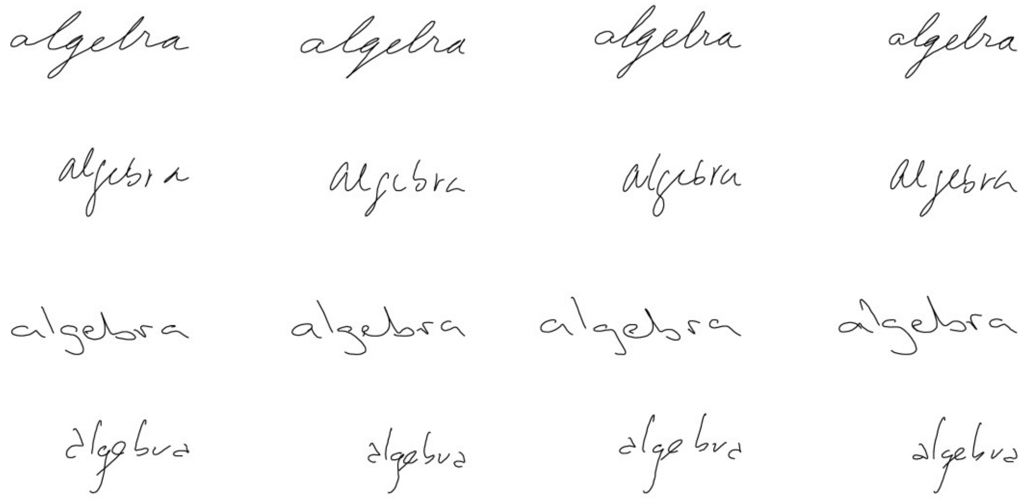


Figure 1.1: The word "algebra" written by different writers. Each row contains handwriting for a single writer, recorded at four different times. Note that variation manifests itself between writers but also within individual writers. Figure taken from [87].

spotting [17]. Accurate and reliable handwritten text recognition remains a challenging problem, mainly due to the large number of possible handwriting variations. The underlying biomechanical process for handwriting involving coordinated control of the arm, wrist, and finger joints effectively leads to an infinite amount of possible style variations. When it comes to disentangling the factors of variation in handwriting, a distinction can be made between *within-writer variability* and *between-writer variation*. Within-writer variability refers to the inherent variation that a single writer can produce due to changing circumstances such as pen grip, writing hand, etc. Between-writer variation refers to larger differences in handwriting that are a result of a change in writer. This is illustrated in Fig. 1.1, where within-writer variability is displayed along the columns and between-writer variation along the rows.

When it comes to collections of handwritten documents, it is common to have access to additional information regarding a piece of handwritten text. One could refer to such additional information as *metadata*, e.g., who wrote the text, what year it was written, what the script is, etc. For example, consider the Monk collection[1] [102], a large and diverse database of historical handwritten documents containing large amounts of variation in style, language, and script. The material contained in the Monk collection varies from Western medieval, Western administrative and diaries from 1400-1900, to Chinese, Hebrew, Arabic, and Egyptian hieroglyphs, as well as low-resource language variants. The images in the Monk collection often contain additional metadata provided by human labelers.

We ask the question how modern HTR models can be designed in such a way as to be flexible in taking into account additional metadata that comes along with handwriting images.

[1]`http://monk.hpc.rug.nl/`

Specifically, we consider one of the most common kinds of metadata for handwriting, namely *writer identity.* This can be the real-life identity of the writer, or simply an anonymous label indicating that a set of handwriting images is produced by the same writer. We thus ask the question how additional information about the writer of a handwriting image can be used to efficiently improve recognition. This is useful because it is generally not possible to capture all relevant handwriting variations in a single dataset. The writer identity provides an additional signal source that can be used to constrain the expected variation in pixel and character sequence space. Essentially, there are two problems at hand: Improving recognition accuracy for writers that have been seen during training, and improving recognition accuracy for novel writers whose handwriting may lie outside the training distribution. We will be focusing mainly on the latter problem.

A change in writer is a manifestation of *distributional shift*, i.e., a change in the joint probability distribution of pixels and character sequences. For example, certain letters exhibit more variation across writers than others. The letter "k" is more likely to exhibit variation across writers than a more "neutral" letter such as the "o". In the ideal case, we would capture this distributional shift by considering distributions *conditioned* on the writer identity. This notion of conditionality is elegantly expressed within the framework of Bayesian statistics. Generically speaking, Bayesian statistics deals with updating a current hypothesis H as additional evidence E comes in. This is commonly known as *Bayes' rule*:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)} \tag{1.1}$$

where $P(H|E)$ is a conditional probability indicating the probability of H given E. We can use the same notation for the case of HTR. If we assume that the writer identity w for a handwritten image $\mathcal{I}$ can meaningfully inform a transcription T, we can express this as a change in the data distribution by including w:

$$P(T|\mathcal{I}) \neq P(T|\mathcal{I}, w) \tag{1.2}$$

In other words, we now model the target probability conditioned on the input image *and* the writer identity. The Bayesian notation thus provides a flexible way to include various kinds of conditionals into a modeling framework. Unfortunately, Bayesian statistics applied in practice tends to become problematic for high-dimensional probability distributions, where the computational complexity of modeling such a distribution becomes prohibitively expensive. However, the underlying conceptual idea is still highly relevant.

The issue of how to effectively include writer identity into a HTR neural network as a conditioning variable as shown on the right side of Eq. 1.2 is one of the central questions of this research. It is important to note that the potential benefit of explicitly including writer identity as a conditional variable cannot easily be decoupled from architectural choice. For example, Hidden Markov Models [10] have been a common choice for HTR in the past, and methods have been developed to include writer identity in such models. However, these methods are often not useable for modern approaches to HTR using deep neural networks, which make use of powerful hierarchical representations that outperform methods of the past. In this sense, the more suitable question to ask is whether state-of-the-art deep learning approaches to HTR can benefit from explicit writer information in the first place. For example, a convolutional neural network (specialized to process images) may learn convolutional kernels that are good at detecting cursive letter shapes. If the cursive style is modeled well enough, the added value of adaptation based on information about whether a style is cursive or not may prove negligible. Thus, in order to adapt effectively based on style information, there is a clear need to identify *what exactly the model has not learned yet*. In the case of writer-based adaptation, the question can be formulated as "what novelty does this new writer introduce, that is not effectively handled by the neural network?". At the same time, the question then also rises what signal source can be provided to allow for adaptation, as well as the non-trivial question of how to effectively include such information into a HTR model.

Using a conditional variable to steer the model output is commonly seen in generative networks such as Generative Adversarial Networks (GAN) [32]. The quintessential example of this is the Conditional GAN (or cGAN) [72], an extension of the generator and discriminator models used in a traditional GAN where an additional conditional variable is part of the input. For generative models, this can serve as a way to constrain the stochastic output space. For example, in the case of generating images of handwritten digits, the conditional variable can indicate what number from 0 to 9 should be generated. However, generative models generally perform a fundamentally different task than discriminative models (which we are concerned with in this research), and the benefit of explicit conditionality for discriminative models is less clear.

Recently, a paper was published by Bhunia et al. [13] which employs *meta-learning* to flexibly adapt HTR models to different writers, seemingly with great success. Meta-learning (also known as learning-to-learn) is currently an active area of research [44]. Broadly speaking, meta-learning is concerned with improving the learning algorithm itself. Oftentimes, the idea is to adapt a learning algorithm to a new task based on a small number of task-specific examples. The aim is to learn underlying *meta-knowledge* that can be transferred to various tasks, even those unseen during training. The paper by Bhunia et al. (2021) makes use of a modified form of model-agnostic meta-learning (MAML) [28] which they call MetaHTR. This work formed the initial inspiration for the current research. We will be exploring several versions of the MAML approach in later chapters and will test its ability to perform writer-specific adaptation.

Additionally, we experiment with several other approaches. The first approach is based on *writer codes*: Compact vector representations of individual writers that are supposed to capture the most relevant information about a writer to allow for effective adaptation. A writer code can be learned, or explicitly given as part of the model input. The codes can be inserted into a trained HTR model by adjusting the parameters of batch normalization layers. We experiment with several approaches to creating such a writer code: One based on learned feature vectors, and one based on traditional handcrafted features used for writer identification. Lastly, we employ an approach originating from literature on domain adaptation, where statistics for intermediate normalization layers are dynamically adjusted based on individual writer statistics.

Let us now formulate the primary research question for the current research:

Can state-of-the-art deep learning-based HTR models benefit from writer identity as a conditioning variable?

To help answer the primary research question, we pose three secondary research questions:

1. How can writer information be represented in a way that is effective for facilitating adaptation and improving recognition performance?
2. What is a suitable method to include writer information in a state-of-the-art deep learning-based HTR architecture?
3. Does architectural choice play a meaningful factor in facilitating effective adaptation?

These research questions will be answered in the following chapters. We summarize the contributions in this thesis as follows:

- We show that MAML-based methods applied to a trained HTR model can lead to improved recognition accuracy, showing an improvement between 1.4 and 2.0 word error rate compared to a naive finetuning baseline;
- We test the capability of MetaHTR to perform writer-specific adaptation, finding that it leads to an improvement of 0.7 word error rate for a deep HTR model, but shows no significant effect for smaller models;
- We analyze how a trained HTR model can be effectively adapted based on writer-specific vector representations, finding that finetuning batch normalization scale and bias parameters can be an effective way to obtain additional performance gains, even without writer-specific information;
- Finally, we find no concrete benefit to including writer-specific vector representations into a trained HTR model.

This thesis is structured as follows. In Chapter 2, we provide background information regarding relevant topics, as well as related work. In Chapter 3, we propose several methods for writer-adaptive HTR, as well as experiments to verify their performance. In Chapter 4, we show results for the proposed methods, and finally, in Chapter 5, we discuss the results and future work. Code used for this research will be released at `https://github.com/tobiasvanderwerff/master-thesis`.

Chapter 2

Background

2.1 Neural networks and deep learning

The current section will highlight some key aspects of deep learning and neural networks that the reader should be aware of. Explanations of various concepts will be brief, and are not intended as a comprehensive introduction to neural networks. For this, the reader is referred to [31].

In a mathematical sense, neural networks are powerful function approximators capable of modeling complex probability distributions. At a conceptual level, deep learning revolves around creating neural networks that are sufficiently *deep* (referring to the number of layers in the network) to learn complex concepts in a hierarchical manner. In this way, complex representations are built out of other, simpler representations. For example, a convolutional neural network (CNN) – a neural network architecture often used for image processing –, builds upon crude visual shapes such as corners and contours in order to represent more complex shapes deeper into the network (such as the notion of a cat or dog). The quintessential deep learning model is the *multilayer perceptron* or MLP (shown in Fig. 2.1), which can be represented as a mapping $\mathbf{y} = f_\theta(\mathbf{x})$, where θ is a vector of learned parameters, also referred to as the *weights* of the network.

The output of a single layer l of a MLP can be represented as a matrix-vector multiplication, followed by the addition of a bias vector and a non-linear activation function,

$$\mathbf{z}^l = \sigma(\mathbf{W}^l\mathbf{z}^{l-1} + \mathbf{b}^l) \tag{2.1}$$

where $\mathbf{z}^{l-1}$ is the output from the previous layer in the network, $\mathbf{W}^l$ and $\mathbf{b}^l$ are learnable parameters for layer l, and σ is a non-linear *activation function.* The activation function serves to explicitly make the layer transformation non-linear, in order to increase its modeling capacity.

At the most basic level, deep neural networks consist of many consecutive layers such as the one shown in (2.1). Besides the MLP architecture, many specialized architectures exist, such as Convolutional Neural Networks for image processing, and LSTMs [42] or Transformers [103] for sequence processing tasks. We will not cover these architectures in depth here.

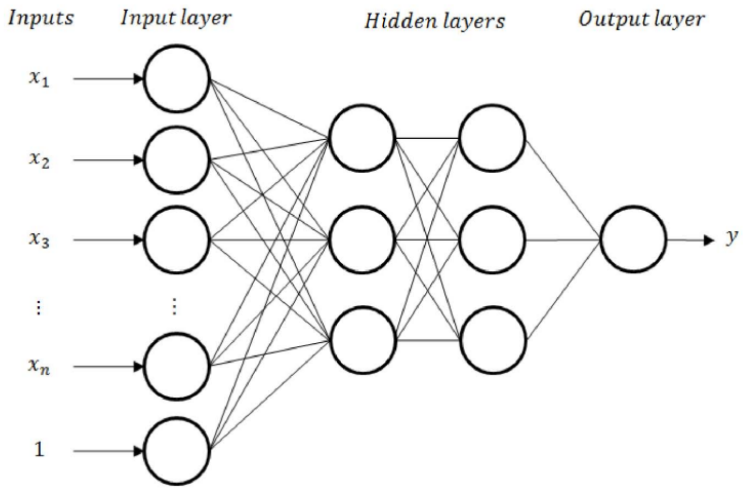


Figure 2.1: Graph representation of a simple multilayer perceptron with three layers. Taken from [36].

In order to learn the correct mapping from input to output, the learned parameters at each layer have to encapsulate representations that are most relevant for the task at hand. For example, a task could consist of mapping an input image of a dog (fed into the neural network using the raw pixel values) to the dog species that the dog belongs to. The representations learned inside the neural network will have to contain meaningful information about various dog features that allow the network to make a correct classification of the species. This is therefore also known as *representation learning*, where features are *learned* inside the model itself. This approach contrasts with an earlier approach to machine learning building on so-called *handcrafted* features, relying on human judgment and manual effort to design appropriate features for a task. Instead, neural networks make use of learned features trained using gradient descent (introduced in Section 2.1.2). In other words, we do not explicitly tell a model how input data should be processed to produce a desired outcome. Instead, we simply design the computational infrastructure for the input/output mapping, and let the model itself learn the most suitable way to process the data. Given enough relevant data, a deep neural network will be able to learn representations that are appropriate for performing a particular task, such as classifying the content of an image.

2.1.1 Cost functions

Training a neural network requires specifying a *cost function*: A differentiable function of the learned parameters that measures the performance of the model. The particular cost function of choice depends on the task at hand. Most neural networks employ the *cross-entropy* cost function, which assumes that the output is a probability function $p(y|x;\theta)$, where y is a target variable and x the input. The cost (or *loss*) for a single input-output pair is then of the form

$$\mathcal{L}(x, y; \theta) = -\log p(y|x; \theta) \tag{2.2}$$

Or for an entire dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$,

$$\mathcal{L}(\mathcal{D}; \theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p(y_i|x_i; \theta) \tag{2.3}$$

An important requirement of any cost function is that its gradient must be large enough to serve as a guide for the learning algorithm. This is an important principle in designing cost functions, but also for designing neural network architectures. Cost functions or activation functions that *saturate* (become very flat) are generally not a good choice in that they violate this principle. An example of this in practice is the replacement of the *logistic sigmoid* activation function (which saturates for large regions of the input space), with the *rectified linear unit* (ReLU) activation function (which is nearly linear and has more stable gradients of larger magnitude).

Although the cost function acts as an optimization metric for training a neural network, it is usually not used directly to measure model performance. Instead, a specialized performance metric is chosen that more directly measures the model performance, e.g., the number of correctly classified examples. The performance metric of choice depends on the task at hand (for the task in this research, we define appropriate metrics in Section 3.2).

In order to get a more accurate picture of model performance on unseen data, datasets are commonly split into three disjoint sets: a training set, validation set, and a test set. The training set is used to train the model; the validation set is used for the purpose of tuning hyperparameters, and the test set is used as a final evaluation of the model performance.

A common phenomenon occurs when a neural network memorizes spurious correlations present in the training set instead of learning representations that generalize to unseen data. This is known as *overfitting*, and can be measured by considering the gap in training performance and validation performance. Overfitting provides yet another reason why performance is not measured directly on training data, but on another set not seen during training. The way overfitting is handled is an aspect of deep learning which differs from the field of statistical learning theory (e.g. [37]), a field from which many of the principles of deep learning are derived. A point of departure between the two fields lies in the idea that model complexity, expressed as the number of parameters, can be increased to large numbers without catastrophic overfitting. Modern neural networks can consist of millions of parameters, commonly exceeding the number of data points available for training.

Commonly, the cost function will include an additional term called the *regularization term*, designed to mitigate overfitting. In the context of machine learning, this is commonly referred to as *weight decay*. The total loss function is then of the following form

$$\mathcal{L} = \mathcal{L}(\mathcal{D}; \theta) + R \tag{2.4}$$

where R is a regularization term. For example, in the case of weight decay, this term is a function of the magnitude of the parameters θ. Other regularization methods exist for neural networks that are not included in the cost function – most commonly the *dropout* method [95].

2.1.2 Stochastic gradient descent

Modern neural networks are trained through an iterative, gradient-based procedure called *stochastic gradient descent* (SGD). It optimizes a specified cost function using iterative updates. A single update is of the form

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L} \tag{2.5}$$

where α is a predefined *learning rate*, controlling the magnitude of the updates. An important aspect of these updates is that the loss function is not measured over the entire training dataset, but rather over randomly sampled batches of data, also called *minibatches*.

Since minibatches are generally much smaller than entire datasets (generally chosen to range between 1 to a few hundred examples), computing a single gradient descent update is much less computationally expensive. By randomly sampling batches of data from the dataset, the cost function gradient is an approximation of the true gradient. By sampling minibatches randomly – and not in the same order every time – the approximation of the gradient is not influenced by the ordering of the examples in the dataset.

Although SGD is an important algorithm for training neural networks, it is generally not used in the form presented in (2.5). Instead, many methods have been proposed to make gradient-based training run more smoothly. One of the most commonly used methods is Adam [56], but many other methods exist. We will not provide further details on such methods here; the reader is referred to [31] for an overview. The question of what method is generally preferable is far from settled, and often depends on the nature of the task at hand, as well as the personal preference of the practitioner.

2.1.3 Normalization methods

Normalization layers are commonly used in modern neural networks. As they will play a prominent role in future chapters – the Batch Normalization method in particular –, we cover them here. In a general sense, neural network normalization involves normalizing intermediate layer activations of a neural network to have zero mean and unit variance.

Batch normalization (Batch Norm or BN) [47] is one of the most prominent normalization methods, used in well-known computer vision architectures such as ResNet [39] and Inception-v3 [97]. For Batch Norm, normalization statistics are calculated over a full minibatch of activations. Applied to a minibatch of activations $B = \{x_{1,\ldots,m}\}$, batch normalization layers are of the following form:

$$y_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \cdot \gamma + \beta \tag{2.6}$$

where

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.7}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.8}$$

γ and β are learnable parameter vectors of size equal to the number of channels in the input. The ϵ parameter is a small constant added for numerical stability.

As an example, consider an intermediate layer input for a CNN network, with shape (N, C, H, W), where N is the batch axis, C the channel axis, and (H, W) the spatial axes. Batch normalization then calculates μ and σ along the (N, H, W) axes and normalizes each channel independently. Fig. 2.2 displays this graphically, along with a few other common normalization strategies.

The learnable parameters γ and β are an important component of Batch Norm because they enable the layer output to represent the identity transform if needed (negating the normalization).

The original explanation for the effectiveness of Batch Norm was that it reduces "internal covariate shift", referring to changes in a layer's input distribution during training. However, more recently, evidence has been put forward indicating that Batch Norm instead leads to reduced sensitivity to weight initialization [83]. This manifests itself in smoother optimization landscapes, which in turn allow for easier training with higher learning rates and reduced sensitivity to hyperparameter settings.

One of the downsides of Batch Normalization is its dependence on batch size. For smaller batch sizes, the estimate of the mean and variance will be less accurate. It is standard practice to use the statistics in a single batch for normalization during training, while stored statistics accumulated during training are used at test time to reduce the dependence on batch size. The discrepancy between train and test time statistics used for normalization can lead to performance differences between the two modes. To reduce the dependence on batch size, several modifications have been proposed, most notably batch renormalization [46] and group normalization [110].

2.1.4 Other ingredients for successful neural network training

Neural networks are not a recent invention, dating back to the late 50s to Frank Rosenblatt's brain-inspired *perceptron* [79]. However, it is only until relatively recently that deep networks have been successfully trained on complex tasks (e.g. [59, 39]). Factors like numerical instability, too slow convergence, or poor model quality largely prevented the successful application of deep networks. The relatively recent success of large neural networks is mostly due to several important innovations in the field. We here provide a brief overview of some of the most important innovations.

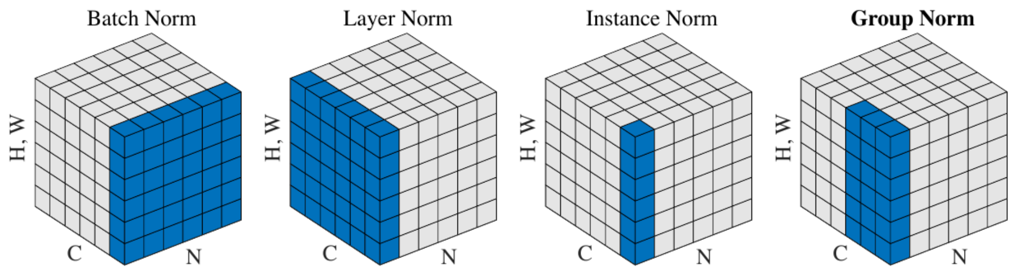


Figure 2.2: Different normalization methods used in neural networks. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Image taken from [110]

- *Backpropagation for gradient descent.* The error backpropagation algorithm [80] serves as a way to efficiently calculate gradients for neural network training via gradient descent, through the application of the chain rule. It served as a key innovation for training deeper neural networks.

- *Better methods for weight initialization.* Due to a large number of sequential matrix-vector multiplications, intermediate layer outputs, as well as their corresponding gradients, can drastically increase or decrease in magnitude, leading to numerical instability issues. This is most commonly known as *vanishing* or *exploding gradients.* Due to the high sensitivity to weight initialization, the choice of initial weights can determine whether a model will converge at all. Modern weight initialization schemes are often simple and heuristic (e.g., [30, 38]).

- *Adaptive learning rates.* The learning rate (shown in Eq. 2.5) is widely established to be one of if not *the* most important hyperparameter to tune, because it has a significant impact on model performance. However, using the same scalar learning rate for all parameters ignores differences in optimization sensitivity along different parameter axes. Algorithms for adaptive learning rates address this issue by using per-weight learning rates that are dynamically updated during training. At this moment, the Adam algorithm [56] is one of the most widely used methods for this. It is regarded as being fairly robust to the choice of hyperparameters, which makes it a relatively easy choice for most tasks.

Other important factors include the increasing capacity and decreasing price of computing hardware, as well as the availability of large datasets. A common thread throughout deep learning research is that innovations are often found and verified experimentally, rather than providing hard theoretical guarantees about new techniques. In this sense, deep learning research is very much an experimental science.

2.2 Offline handwritten text recognition

Given these neural network essentials, we now turn to the task of the current research: Offline handwritten text recognition – commonly referred to in this research as HTR. The goal of offline handwritten text recognition is to recover the handwritten text represented in an input image, containing one or several handwritten words and/or letters. Another way to describe the task is that of turning images of handwritten text into letter codes. Depending on the context, input images can contain handwritten text at the word, line, or even document-level [93]. Research on handwritten text recognition dates back to the early sixties [70], but we will only consider modern approaches to HTR making use of deep neural networks, which have been most successful thus far.

From a probabilistic interpretation, an HTR model f_θ – corresponding to a deep neural network –, is trained to maximize the probability of the correct transcription given an input image,

$$\theta^* = \arg\max_\theta \sum_{(\mathcal{I},Y)\in\mathcal{D}} p(Y|\mathcal{I};\theta) \tag{2.9}$$

Here, $p(.)$ refers to the probability distribution modelled by the neural network given its weights θ, $\mathcal{I}$ represents the input image and $Y = (y_1, y_2, \dots, y_L)$ the ground truth character sequence, where each y_i is picked from a vocabulary V (such as ASCII characters). The full-sequence probability used during training is the multiplication of the invididual character probabilities conditioned on the previous ground truth characters in the sequence:

$$p(Y|\mathcal{I};\theta) = \prod_{t=1}^{L} p(Y_t = y_t|y_{<t}, \mathcal{I};\theta) \tag{2.10}$$

The training dataset $\mathcal{D} = \{(\mathcal{I}_1, Y_1), (\mathcal{I}_2, Y_2), \dots, (\mathcal{I}_N, Y_N)\}$ consists of tuples containing an image $\mathcal{I}_i$ and the corresponding character sequence Y_i. The cost function is derived from cross-entropy, which, for a single example, is of the following form:

$$\begin{aligned} \mathcal{L}(\mathcal{I}, Y;\theta) &= -\frac{1}{L}\log(p(Y|\mathcal{I};\theta)) \\ &= -\frac{1}{L}\sum_{t=1}^{L}\log p(Y_t = y_t|y_{<t}, \mathcal{I};\theta) \end{aligned} \tag{2.11}$$

Modern architectures for HTR most commonly consist of a CNN backbone for initial processing of the image, followed by a sequence modeling architecture such as a Recurrent Neural Network (RNN) or Transformer, sometimes combined with an additional decoding module based on CTC or attention, to predict a sequence of output tokens in an autoregressive manner.

We now discuss two state-of-the-art models for offline handwritten text recognition that will be used throughout this research. In particular, we will be focusing on HTR for word

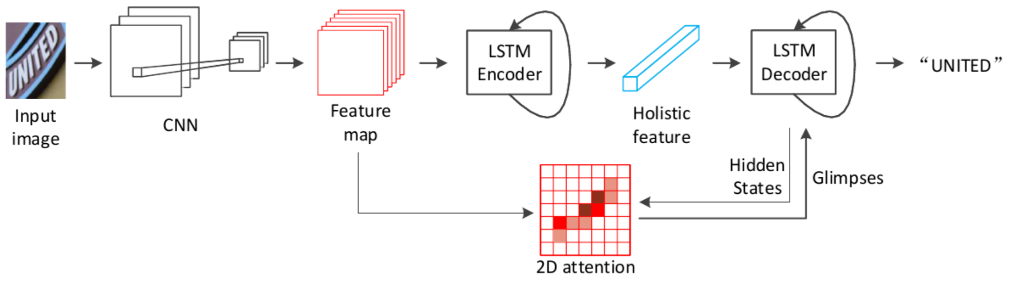


Figure 2.3: Schematic of SAR, the LSTM-based model used in [61].

images. We will refer to the architectures that follow as *base models* because they will be used as a starting point for most of the methods discussed in Chapter 3.

2.2.1 LSTM-based model

Our first base model [61] is based on the Long short-term memory (LSTM) architecture [42]. The full architecture is displayed in Fig. 2.3 and consists of a ResNet image processing backbone, LSTM encoder, LSTM decoder, and a 2-dimensional attention module. We will refer to this architecture as SAR (Show, Attend, and Read), and add a suffix indicating what image encoder is used (e.g., SAR-31 indicates that a ResNet with 31 layers is used). As the name of the paper suggests (a "simple and strong baseline"), the architecture itself is relatively straightforward, consisting for the most part of standard neural network building blocks. Although the architecture is presented for the task of natural scene text recognition, the generality of the architecture makes it such that it can just as easily be applied to handwritten text.

CNN backbone

The CNN backbone consists of a modified ResNet [39] with the final average-pooling and linear projection layer removed. The authors use a 31-layer ResNet, with small modifications specifically for the HTR task [91]. It outputs a 2-dimensional feature map, which is used by the consecutive LSTM encoder to extract a holistic feature vector for the whole image, and also serves as context for the 2D attention network. The ResNet architecture consists of 31 layers and 2×1 max pooling for some layers as proposed in [91]. The modified pooling kernels preserve more information along the horizontal axis and are supposed to benefit the recognition of narrow-shaped characters (e.g. 'i', 'l'). However, due to the large number of parameters that this architecture has (see Appendix B.1), we also make use of a standard ResNet-18 architecture. This allows for faster experimentation and fewer potential problems with fitting all the necessary components into GPU memory.

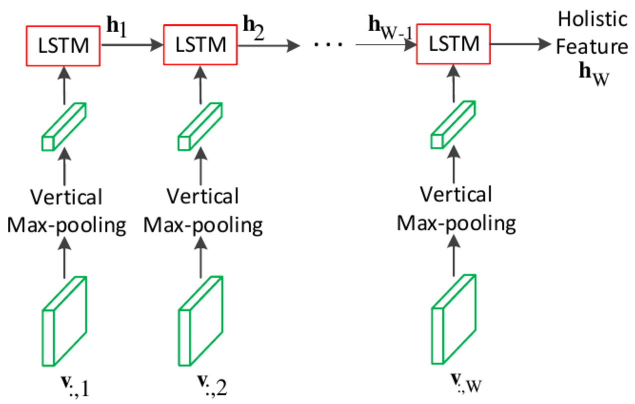


Figure 2.4: Structure of the LSTM encoder used in SAR. Here, $\mathbf{v}_{:,i}$ represents the i'th column of the 2D feature map $\mathbf{V}$. At each time step, a single column feature is max-pooled along the vertical direction, and then fed into the LSTM. Figure taken from [61].

LSTM encoder

The LSTM encoder processes the extracted feature map as a sequence of feature vectors. The 2D feature map $\mathbf{V}$ extracted from the CNN is reduced to its horizontal dimension by applying max-pooling across the vertical dimension, after which it is sequentially fed into an LSTM. This is displayed visually in Fig. 2.4. After all image columns have been consumed by the LSTM, the final hidden state $\mathbf{h}_W$ is used as a fixed-size representation of the input image and provided for the decoding that follows.

LSTM decoder

The LSTM encoder is followed by another LSTM, used for decoding. This is shown in Fig. 2.5. The last hidden state $\mathbf{h}_W$ of the LSTM encoder is used as the initial input for the decoder. Then for the first time step, a special start-of-sequence token (<SOS>) is fed as input to the LSTM. At each timestep of the LSTM, a new character is sampled autoregressively. Each input at the timesteps that follow is either 1) the previous character from the ground truth character sequence (also known as *teacher forcing*), or 2) the sampled character from the previous timestep (at test time). If the latter is the case, the end of the sampling procedure is signified by sampling a special end-of-sequence token (<EOS>).

All token inputs are fed in as vector representations, followed by a linear transformation, $\psi(.)$. After being fed through an LSTM cell along with the previous hidden state, the timestep prediction is then calculated as

$$\boldsymbol{y}_t = \phi(\boldsymbol{h}'_t, \boldsymbol{g}_t) = \mathrm{softmax}(\boldsymbol{W}_o[\boldsymbol{h}'_t; \boldsymbol{g}_t]) \tag{2.12}$$

where $\boldsymbol{h}'_t$ is the current hidden state and $\boldsymbol{g}_t$ is the output of the attention module. $\boldsymbol{W}_o$ is a linear transformation, which maps the features to a vector whose size is equal to the number of character classes.

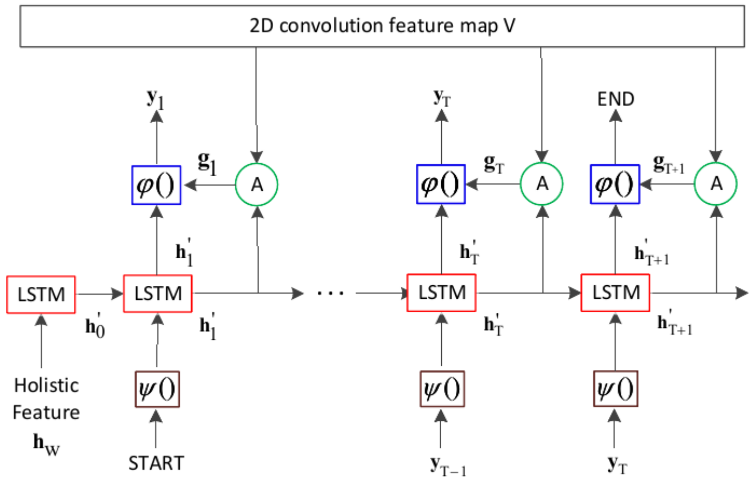


Figure 2.5: Structure of the LSTM decoder used in SAR. The last hidden state from the LSTM encoder, $\mathbf{h}_W$, is fed as initial input to the decoder, followed by the start-of-sequence token and characters from the previous time steps. If teacher forcing is not used, sampling is halted by sampling the end-of-sequence token. At each time step t, the output y_t is computed as $\phi(\mathbf{h}'_t, \mathbf{g}_t)$, where $\mathbf{h}'_t$ is the current hidden state and $\mathbf{g}_t$ is a glimpse vector obtained from the 2D attention module. Figure taken from [61].

The attention module is a modification of the standard 1D attention module for dealing with a 2D spatial layout. It takes into account neighborhood information in the 2D plane:

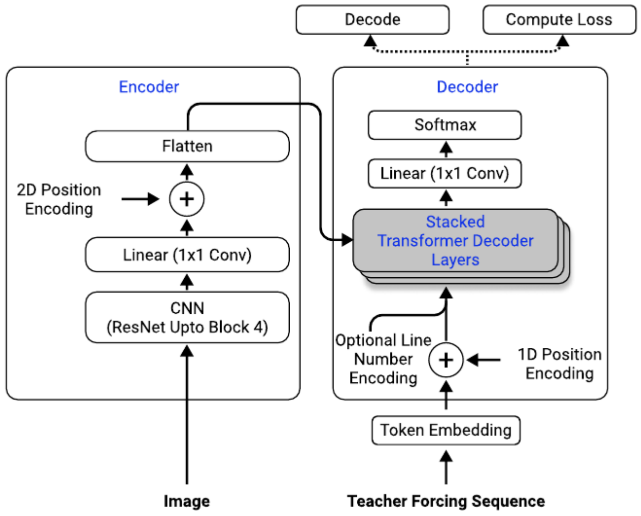
$$\begin{cases} \mathbf{e}_{ij} &= \tanh(\mathbf{W}_v \mathbf{v}_{ij} + \sum_{p,q \in \mathcal{N}_{ij}} \tilde{\mathbf{W}}_{p-i,q-j} \cdot \mathbf{v}_{pq} + \mathbf{W}_h \mathbf{h}'_t) \\ \alpha_{ij} &= \text{softmax}(\mathbf{w}_e^T \cdot \mathbf{e}_{ij}) \\ \mathbf{g}_t &= \sum_{i,j} \alpha_{ij} \mathbf{v}_{ij}, \quad i = 1, \dots, H, \quad j = 1, \dots, W \end{cases}$$

Explanation of the symbols: $\mathbf{v}_{ij}$ is the local feature vector at position (i, j) in $\mathbf{V}$; $\mathcal{N}_{ij}$ is the eight-neighborhood around this position; $\mathbf{W}_v, \mathbf{W}_h, \tilde{\mathbf{W}}$ are learned linear transformations; α_{ij} is the attention weight at location (i, j); and $\mathbf{g}_t$ is the weighted sum of local features, also known as a *glimpse*. The difference with a traditional attention module is the addition of the $\sum_{p,q \in \mathcal{N}_{ij}} \tilde{\mathbf{W}}_{p-i,q-j} \cdot \mathbf{v}_{pq}$ term when weighing $\mathbf{v}_{ij}$.

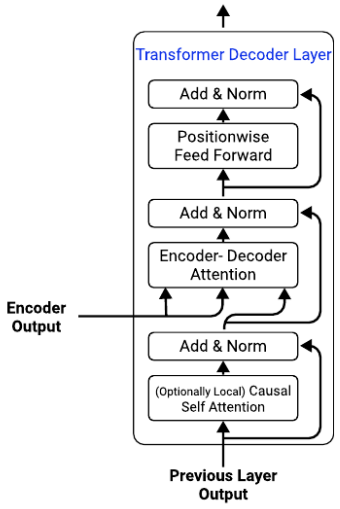
2.2.2 Transformer-based model

With the success of the Transformer architecture [103] in the field of Natural Language Processing, there has been a trend in recent years [51, 93, 119, 62] to replace the attention-based LSTM [42] with the Transformer architecture, which allows for efficient parallelization. This is especially relevant for modeling longer sequences, e.g. lines or paragraphs, where the recurrence and lack of parellization of the LSTM architecture can become a bottleneck.

We use a recent architecture proposed in [93], displayed in Fig. 2.6. It consists of a CNN backbone combined with a standard Transformer for decoding the visual feature map into a



(a) Model Outline


Transformer Decoder Layer
Add & Norm
Positionwise Feed Forward
Add & Norm
Encoder- Decoder Attention
Encoder Output
Add & Norm
(Optionally Local) Causal Self Attention
Previous Layer Output

(b) Decoder Transformer Layer

Figure 2.6: Schematic of the Transformer-based model used in [93]. It combines a CNN backbone (referred to in the figure as an encoder) with a Transformer sequence decoder.

character sequence. The architecture was originally proposed for full-document HTR, but due to its generic nature, it can easily be applied to both word and line images without any real modifications. We will refer to this model as FPHTR, with a suffix indicating the number of layers for the ResNet encoder that is used, e.g. FPHTR-18 indicates that a ResNet with 18 layers is used.

CNN backbone

Just like in the LSTM-based model, the CNN backbone consists of a ResNet architecture with the final average-pooling and linear projection layer removed. We use the same ResNet architecture as for SAR, with 18 or 31 layers, consisting of 11.3M and 46.0M parameters, respectively (see Table B.1). The CNN takes as input an image and produces as output a 2D feature map with hidden size d_{model}. A 2D position encoding based on sinusoidal functions is added (Equation 2.13) and the feature map is flattened into a 1D sequence of feature vectors – each representing a position in the image –, that can be processed by the Transformer decoder.

$$\begin{aligned}
PE(y, 2i) &= \sin(y/10000^{2i/d_{model}}) \\
PE(y, 2i+1) &= \cos(y/10000^{2i/d_{model}}) \\
PE(x, d_{model}/2 + 2i) &= \sin(x/10000^{2i/d_{model}}) \\
PE(x, d_{model}/2 + 2i + 1) &= \cos(x/10000^{2i/d_{model}}) \\
i &\in [0, d_{model}/4)
\end{aligned} \tag{2.13}$$

Transformer decoder

The decoder is a standard Transformer architecture [103] with non-causal attention to the encoder output (it can attend to the entire output of the encoder), and causal self-attention (it can only attend to past positions of its character input).

Input vectors are enhanced with 1D position encodings, as in [103]. The "optional line number encoding" displayed in Fig. 2.6 is omitted because within the current context the focus is on word images – not line images.

Sampling is done autoregressively by feeding in all previously sampled tokens as input and taking the argmax of the final layer output as the next sampled token.

2.3 Meta-learning

In this section, we provide relevant knowledge concerning the topic of meta-learning. Meta-learning is broadly defined as the process of improving a learning algorithm over multiple learning episodes [44]. It serves as one of the main inspirations for the work proposed in this research.

This section is structured as follows. First, we discuss the underlying framework used for meta-learning and how it relates to the HTR task in Section 2.3.1. Then, in Section 2.3.2, we turn to the meta-learning algorithm used in the current research known as MAML. Lastly, in Section 2.3.3, we explain a recent approach called MetaHTR that applies MAML to the task of writer-adaptive HTR.

2.3.1 Episodic learning

Neural network-based meta-learning makes use of an alternate framework for training and evaluating models, sometimes referred to as *episodic learning* [44]. One of the distinguishing characteristics of this framework is that it requires a labeled batch of task-specific data at test time that can be used to update an existing base model. For example, the MAML algorithm, which will discussed in Section 2.3.2, trains a model that – at least in most cases – requires adaptation on a batch of labeled task-specific inputs to function properly.

Note that this approach to learning is a different kind of conditioning than what has been discussed so far. Rather than conditioning a model output on a specific variable such as the writer identity, a model is now conditioned on a small set of images that share a common feature (such as the writer who produced the handwriting samples). For the case of few-shot learning (a common meta-learning problem), the adaptation batch contains examples of a new output class that should be learned. The neural network is then conditioned on this specific task.

In contrast to regular neural network learning, learning here occurs over multiple learning *episodes*, using a limited amount of adaptation data, combined with test data to evaluate the effectiveness of the adaptation. Within the meta-learning literature, these two sets are commonly referred to as the *support set* and the *query set*, respectively. In the K-shot learning setting, a model is adapted based on K samples sharing a piece of conditional information, which we denote as the *conditional label* $l \in \mathcal{W}$. The current setup presumes that each sample from a dataset contains a conditional label, and for simplicity it is presumed that a single sample only contains a single conditional label.

For training a model, the label set $\mathcal{W}$ is split into three subsets: a *training meta-set* $\mathcal{W}^{tr}$, *validation meta-set* $\mathcal{W}^{val}$, and a *testing meta-set* $\mathcal{W}^{test}$. Each set contains a disjoint set of labels, i.e., $\mathcal{W}^{tr} \cap \mathcal{W}^{val} \cap \mathcal{W}^{test} = \emptyset$. This serves as a way to test how well the adaption works on unseen conditional labels. During training, we sample a set of labels $L \subset W$ and use each $l \in L$ to sample from the label-specific set D^l a support set D^{tr} consisting of K examples, and a query set D^{val} containing all remaining examples, such that $D^l = D^{tr} \cup D^{eval}$. The support set D^{tr} is used for fast adaptation of the model based on the label-specific examples, and the query set D^{val} is used to estimate generalization performance on conditional label-specific examples after adaptation on D^{tr}.

During episodic learning, the parameters of a base model f_θ are adapted by applying some differentiable function, leading to conditional label-specific parameters $\theta_i^{'}$, with an update rule of the form

$$\theta_i^{'} = \mathcal{G}(\theta, D^{tr}) \tag{2.14}$$

where $\mathcal{G}$ depends on the specifics of the conditional adaptation algorithm. In essence, one of the primary goals of the current research is to explore what forms $\mathcal{G}$ can take to effectively adapt to conditional information for the specific case of handwritten text recognition.

During this research, writer identity will be used as the primary piece of conditional information used to adapt a learning algorithm. Thus, each conditional label l corresponds to a writer identity, i.e., $\mathcal{D}^l$ contains labeled data from a specific writer. Therefore, we will use the terms "conditional label" and "writer identity" interchangeably in what follows. However, it should be noted that using writer identity as the main piece of conditional information is not the only option. For example, generic writing styles could also be used for episodic learning, assuming that images can be clustered effectively by style. Using writer identity as the primary conditional information in this research serves as a proof of concept for the idea of conditional adaptation for handwriting recognition.

It is worth noting that the additional information source that the conditional label provides is not so much in the label itself, but in the fact that each support batch used for adaptation consists of samples that share the same conditional label. In contrast to regular learning where a single batch would contain randomly sampled writers, knowing that a single batch contains only a single writer provides additional information that can hopefully be utilized. The primary challenge, therefore, lies in effectively utilizing this holistic knowledge related to a batch of data to adapt a learning algorithm.

Rather than creating an adaptive model from scratch, we start from a trained neural network base model trained in a standard way for the HTR task, as mentioned in Section 2.2. Then, the base model should be modified to make adaptation possible. Ideally, the modification required for making the model adaptive is as model-agnostic as possible and does not require complete retraining of all the weights in the base model. This serves to make the

Algorithm 1 Model-agnostic meta-learning, as defined in [28].

```
Require: $p(\mathcal{T})$: distribution over tasks
Require: $\alpha, \beta$: step size hyperparameters
1: Randomly initialize $\theta$
2: while not done do
3:     Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4:     for all $\mathcal{T}_i$ do
5:         Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6:         Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7:     end for
8:     Update $\theta = \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: end while
```

adaptive approach as flexible and non-intrusive as possible; ideally allowing it to be applied to existing HTR models without too much additional effort in retraining or redesigning the architecture.

During *base learning* of a learning algorithm f_θ, the learning algorithm is trained to perform handwritten text recognition, i.e., transcribing handwritten text from images. This stage of training does not include any conditional information. Recall from Eq. 2.9 that an HTR model is trained to maximize the probability of the correct transcription given the input image. During the *meta-learning* stage, episodic learning is used to find the optimal model parameters:

$$\theta^* = \arg\max_\theta \mathbb{E}_{l \in \mathcal{W}} \left[\mathbb{E}_{D^{tr} \subset D^l} \left[\sum_{(\mathcal{I}, Y) \in D^{val} = D^l \setminus D^{tr}} p(Y | \mathcal{I}; \theta_i^{'}) \right] \right] \tag{2.15}$$

Given this background on episodic learning, we now turn to the primary meta-learning algorithm used in this research: model-agnostic meta-learning.

2.3.2 Model-agnostic meta-learning

Model-agnostic meta-learning (MAML) [28] is an approach to meta-learning aimed at finding initial parameters θ that facilitate rapid adaptation. A model is adapted to varying tasks, and the meta-learner aims to find initial model parameters that are sensitive to various tasks. The purpose of MAML is not to directly optimize for any particular task, but rather to optimize for rapid adaptability, where adaptation is done by performing a small number of gradient updates on a support batch. Thus, whenever a new task arrives, a MAML-adapted model should be able to rapidly adapt to that particular task.

As the name suggests, MAML is *model-agnostic*, which means that no assumptions are made about the form of the model, other than to assume that it is parameterized by some parameter vector θ and that the loss function is smooth enough in θ that gradient-based learning can be employed. We consider a distribution over tasks $p(\mathcal{T})$ to which a model should be able to adapt. For example, for image classification, these tasks could represent novel image classes, and samples from the task would represent images labeled with the task-specific image class. During meta-training, a batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$ is sampled, where samples from each task are split up in a support set D^{tr} of size K for adaptation (where typically K is relatively small, e.g., $K \leq 16$), and a query set D^{val} for testing the task-specific performance after adaptation.

Training is done using stochastic gradient descent (SGD), where the model parameters are adapted to a task as follows:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}^{inner}(D_i^{tr}; \theta) \tag{2.16}$$

This is also referred to as the *inner loop.* The inner loop can consist of a single gradient update, but can also possible be repeated several times using the updated parameters. After inner loop adaptation, the adapted parameters $\theta_i^{'}$ are evaluated on the query set, and the original parameters are updated by aggregating the loss over the sampled tasks:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}^{outer}(D_i^{val}; \theta_i') \tag{2.17}$$

This is known as the *outer loop.* It is important to note the subtle difference in the use of the parameter vector θ in (2.16) and (2.17). The result of the inner loop in (2.16) is the updated parameter vector θ_i', which is subsequently used to calculate the loss on the query set D_i^{val} in the outer loop (2.17). The final gradient update however (2.17), is performed with respect to the original parameters, and not with respect to the same parameters used for calculating the outer loss. This also implies that the outer loop gradient update requires backpropagating through multiple computational graphs, calculating second-order derivatives.

Whereas the inner loop optimizes for task-specific performance, the outer loop optimizes for a parameter set θ so that the task-specific training is more efficient. This is referred to as the *meta-objective.* In this way, the goal is to achieve a good generalization across a variety of tasks. The full general case MAML algorithm using the notation from the original paper is shown in Algorithm 1. Fig. 2.7 provides a visual representation of the MAML update.

Note that MAML requires defining two learning rates, α and β. However, since α is part of an inner objective function optimized in the outer loop, it can be learned along with the model parameters. This was originally proposed in [65], where the authors propose using a single learned learning rate for each model parameter. In this way, we optimize not just for good initial parameters, but also for the learning rate and update direction used for adaptation (the learning rate can be negative and thus change the direction of the update).

Although Algorithm 1 suggests starting from random parameters, depending on the nature of the tasks, it is also an option to start with pre-trained parameters. This means that in some cases, MAML can be applied to already-trained models, which shows the flexibility of this approach for adaptive learning.

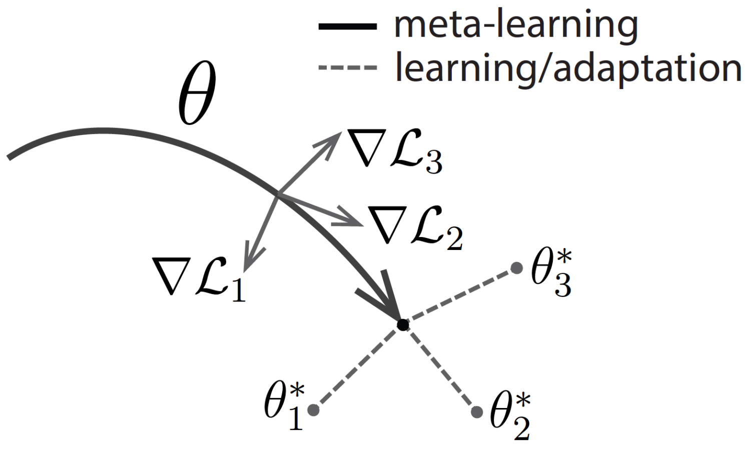


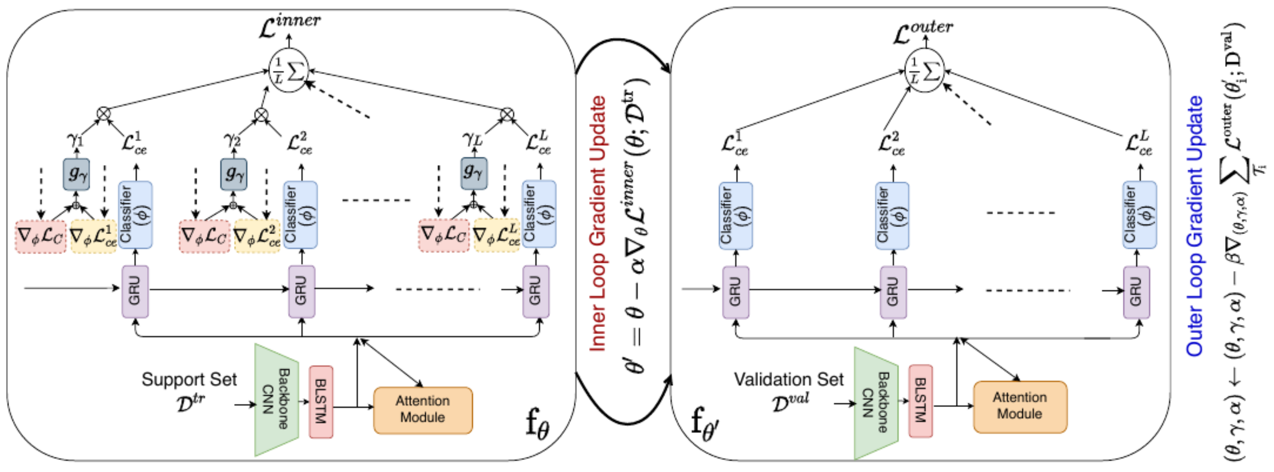
Figure 2.7: Diagram of the MAML approach, taken from [28]. MAML optimizes for a representation θ that can rapidly adapt to new tasks. After adapting the original parameters to several task-specific parameter vectors θ_i^*, the original model parameters are updated using the aggregate of task-specific gradients with respect to θ.

2.3.3 MetaHTR

Given the backdrop of the MAML algorithm as described in the previous section and its flexibility in handling changing data distributions, the question rises how this algorithm could potentially be applied as an adaptive approach to handwritten text recognition. Since MAML allows – at least in principle – absorbing information from related tasks to generalize to unseen ones during testing, it could perhaps be applied in the same way to HTR, where the idea is to generalize to arbitrary writing styles. To this end, Bhunia et al. [13] propose MetaHTR, a writer-adaptive approach to HTR grounded in the MAML framework. Compared to several baseline HTR models, MetaHTR shows increased performance on benchmark HTR datasets. This framework is shown in Fig. 2.8. It is worth noting that the HTR model architecture specified in this figure (CNN + BiLSTM + GRU + attention module) is not specific to the MetaHTR framework itself, and can be replaced by other HTR architectures.

Within the MetaHTR framework, each task instance $\mathcal{T}_i$ corresponds to a different writer. The classification target however remains the same – the model still needs to produce a correct transcription of the text displayed in an image. This distinguishes it from the usual MAML setup for few-shot classification, where the classification targets themselves change as the tasks change. For MetaHTR, the task merely serves as a piece of meta-data that can be used for improving performance on a fixed model objective, namely transcribing text in an image. This matches closely our problem definition where we would like to include conditional label information to improve text recognition performance.

The MetaHTR approach aims to adapt general-purpose HTR models to specific writers based on a limited amount of adaptation data. The underlying assumption, according to the authors, is that "there is always a new writing style that is unobserved, and is drastically different to the already captured". By modeling knowledge across styles, the aim is to learn



$\mathcal{L}^{inner}$
$\frac{1}{L}\sum$
γ_1
γ_2
γ_L
$\mathcal{L}_{ce}^1$
$\mathcal{L}_{ce}^2$
$\mathcal{L}_{ce}^L$
g_γ
$\nabla_\phi \mathcal{L}_C$
$\nabla_\phi \mathcal{L}_{ce}^1$
$\nabla_\phi \mathcal{L}_{ce}^2$
$\nabla_\phi \mathcal{L}_{ce}^L$
Classifier (ϕ)
GRU
Support Set $\mathcal{D}^{tr}$
Backbone CNN
BLSTM
Attention Module
f_θ
Inner Loop Gradient Update
$\theta' = \theta - \alpha \nabla_\theta \mathcal{L}^{inner}(\theta; \mathcal{D}^{tr})$
$\mathcal{L}^{outer}$
Validation Set $\mathcal{D}^{val}$
$f_{\theta'}$
Outer Loop Gradient Update
$(\theta, \gamma, \alpha) \leftarrow (\theta, \gamma, \alpha) - \beta \nabla_{(\theta,\gamma,\alpha)} \sum_{\mathcal{T}_i} \mathcal{L}^{outer}(\theta'_i; \mathrm{D}^{val}_i)$

Figure 2.8: Schematic of the MetaHTR approach as proposed in [13]. The optimization process sequentially involves an inner loop (left) and outer loop (right). The inner loop trains pseudo-updated model parameters θ' on the support set, by means of the inner loss $\mathcal{L}^{inner}$ weighted by learnable character instance-specific weights γ. This also includes a learnable layer-wise learning rate, α. The pseudo-updated parameters are evaluated in the outer loop by means of the outer loss $\mathcal{L}^{outer}$ on the validation set, after which the meta-parameters (θ, ψ, α) are updated.

the general rules of handwritten text more effectively than could be done using regular HTR training, where the model has no knowledge of commonalities in writing style among samples.

The full training process is summarized in Algorithm 2. It is good to emphasize once more that after training MetaHTR, the model is now ready for adaptation using task-specific data (which also holds for MAML more generally). In other words, the inference process requires adapting the model to a batch of writer examples, after which the model can make predictions on another set from the same writer. This process is summarized in Algorithm 3.

We now discuss the main modifications made to MAML to give rise to MetaHTR. The authors propose two modifications to the MAML algorithm to make it more effective for HTR: character instance-specific weights, and learnable layer-wise learning rates.

Character instance-specific weights

The authors conjecture that task adaptation could be made more efficient by additionally learning weight values for each character instance-specific loss, such that the model adapts better with respect to those characters having a high discrepancy. This leads to a modification of the inner loop loss using instance weights. Given a ground truth character sequence $Y = \{y_1, y_2, \ldots, y_L\}$ and an image $\mathcal{I}$, the cross-entropy loss used in the inner loop now uses an additional weight value γ_t for each timestep t. This leads to an inner and outer loss of the form

$$\mathcal{L}^{inner} = -\frac{1}{L}\sum_{t=1}^{L} \gamma_t \log p(y_t|\mathcal{I};\theta) \tag{2.18}$$

$$\mathcal{L}^{outer} = -\frac{1}{L}\sum_{t=1}^{L} \log p(y_t|\mathcal{I};\theta) \tag{2.19}$$

where (2.19) is the same as (2.11), and (2.18) additionaly includes γ_t values inside the summation, denoted as *character instance-specific weights.* The inner and outer loop losses are also displayed in Fig. 2.8. To ensure that the the loss instances – i.e., the summands in (2.18) – correspond one-to-one to ground truth character instances, we always use teacher forcing in the inner loop (see Section 3.3). This presumes the ground truth character sequence is available, which is always the case in the inner loop.

In order to calculate γ_t, gradient information from the final classification layer is used. The idea of using gradient information directly as input to another learned module originates from earlier work that suggests the inner loop gradient contains relevant information related to disagreement with respect to the model's initialization parameters [9]. Specifically, let the weights of the final classification be denoted as ϕ. The gradients of the t'th instance loss with respect to the weights of the final classification layer are used, denoted as $\nabla_\phi \mathcal{L}^t$, in combination with the gradients of the mean loss (Eq. 2.11), denoted as $\nabla_\phi \mathcal{L}$. Both inputs are concatenated and fed as input to a network g_ψ,

$$\gamma_t = g_\psi([\nabla_\phi \mathcal{L}^t; \nabla_\phi \mathcal{L}]) \tag{2.20}$$

where g_ψ takes the form of a 3-layer MLP with parameters ψ, followed by a sigmoid layer to produce a scalar output value in the range [0, 1].

Learnable layer-wise learning rates

As mentioned in Section 2.3.2, the inner loop learning rate used in MAML can be replaced by a learnable one, since it resides inside the outer loop objective function and can thus be differentiated as part of the outer loop gradient. Specifying a learnable learning rate for every model parameter is more expressive since it allows the model to express natural differences between what parameters should be updated more or less. However, using a learning rate for every parameter also implies a doubling in the parameter count. Therefore, a compromise is used, namely *layer-wise* learnable learning rates. Every layer in the model receives a individual learning rate, which is trained along with all the other parameters. This is also shown in Algorithm 2. In other words, in the inner loop update rule specified in (2.16), α is now a vector of size equal to the number of layers in the model.

Algorithm 2 Training for MetaHTR, adapted from [13].

```
Require: Training dataset $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{|\mathcal{W}^{tr}|}\}$
Require: $\beta$: learning rate
 1: Initialize $\theta, \psi, \alpha$
 2: while not done do
 3:     Sample writer-specific $\mathcal{T}_i = \{D_i^{tr}, D_i^{val}\} \sim p(\mathcal{T})$
 4:     for all $\mathcal{T}_i$ do
 5:         Evaluate inner objective: $\mathcal{L}^{inner}(\theta; D_i^{tr})$
 6:         Adapt: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}^{inner}(\theta; D_i^{tr})$
 7:         Compute outer objective: $\mathcal{L}^{outer}(\theta_i'; D_i^{val})$
 8:     end for
 9:     Update meta-parameters: $(\theta, \psi, \alpha) \leftarrow (\theta, \psi, \alpha) - \beta \nabla_{(\theta,\psi,\alpha)} \sum_{\mathcal{T}_i} \mathcal{L}^{outer}(\theta_i'; D^{val})$
10: end while
```

Algorithm 3 Inference for MetaHTR, adapted from [13].

```
Require: Testing dataset $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{|\mathcal{W}^{test}|}\}$
Require: Meta-learned model parameters $\{\theta, \psi, \alpha\}$
Require: A given writer $j$
 1: Evaluate inner objective: $\mathcal{L}^{inner}(\theta; D_j^{tr})$
 2: Adapt: $\theta_j' = \theta - \alpha \nabla_\theta \mathcal{L}^{inner}(\theta; D_j^{tr})$
return Writer-specialized HTR model parameters $\theta_j'$
```

2.4 Related work

In this section, we provide an overview of related works that connect to the topic of the current research. A recent trend in deep learning is the gradual convergence to architectures that are largely domain-agnostic, minimizing domain and task assumptions [50]. Because of this, we approach the problem of writer-adaptive HTR broadly, by looking not only at the literature on handwriting recognition but also at various distinct subfields of deep learning that may provide relevant insights.

2.4.1 Handwritten text recognition

Early approaches to HTR often employed Hidden Markov Models [14] (HMM). More recently, the field of HTR has progressed from HMM-based methods to end-to-end trainable neural networks with many layers. Recurrent neural networks (RNN), and in particular Multidimensional Long Short-Term Memory (MDLSTM), networks [34] have been commonly used sequence modeling architectures for HTR models [76]. The MDLSTM architecture in combination with the Connectionist Temporal Classification [33] loss (CTC) served as a replacement for Hidden Markov Model-based methods [35]. Whereas standard RNN architectures process data along a one-dimensional axis – e.g., a time axis –, the MDLSTM architecture allows recurrence across multi-dimensional sequences, such as images.

In more recent years, it has been observed that the expensive recurrence of the MDLSTM could be replaced by a CNN + bidirectional LSTM architecture [91, 76]. The CNN-RNN hybrid + CTC architecture has been a commonly used architecture in recent years (e.g., [27, 96, 107]). For example, in [27], a spatial transformer network, residual convolutional blocks (ResNet-18), stacked BiLSTMs and a CTC layer are used.

Although CTC has been a common decoding method, some of its downsides – such as the inability to consider linguistic dependency across tokens – have led to architectures that replace CTC in favor of attention modules [8]. Attention-based encoder-decoder architectures have reached state-of-the-art performance in recent years [71]. Attention alleviates constraints on input image sizes and the need for segmentation or image rectification [49] for irregular images. This thus allows for simplification in the design of HTR architectures. In [61], a ResNet-31 is combined with an LSTM-based encoder-decoder along with a 2-dimensional attention module for irregular text recognition in natural scene images.

A trend in recent years has been to replace the linear recurrence of RNNs with the more parallelizable Transformer architecture and attention-based approaches more broadly. In a recent work [24], various architectures for universal text line recognition are studied, using various encoder and decoder families. The authors find that a CNN backbone for extracting visual features, coupled with a Transformer encoder, a CTC decoder, and an explicit language model is the most effective approach for recognition of line strips.

Building on top of the recent trend [25] of using Transformer-only architectures for vision tasks, [62] explore an end-to-end Transformer encoder-decoder architecture for text recognition, initialized with a pretrained vision Transformer for extracting visual features and a pretrained RoBERTa [67] Transformer for sequence decoding. After initialization, the model is pretrained on large-scale synthetic handwritten images and finetuned on a human-labeled dataset.

2.4.2 Writer identification and verification

Let us now consider related tasks in the field of Handwriting Recognition. In the fields of writer identification and verification [88], writer identity is the primary target. These tasks are potentially relevant since the incorporation and representation of writer-based conditional information is one of the fundamental questions present in the current research.

Numerous approaches have been proposed for writer identification based on handcrafted features, where, roughly speaking, the characteristics of writer individuality can be based on textural or character-shape features ("allographs"). The Hinge feature [16] attempts to characterize writer individuality independently of the textual content of the written samples. It uses a probability distribution of the angle combination of two hinged edge fragments to characterize writer individuality.

Connected-Component Contours [90] uses a codebook of prototypical character shapes as a reference for describing new samples of handwriting, using connected components as the basis for constructing the codebook. The histogram of normalized distances to the prototypes is then used as the feature vector for new handwriting samples.

Approaches also exist based on learned features. In [40], multi-task learning with soft parameter sharing is employed to enforce the emergence of reusable features for writer identification, using a single auxiliary task related to explicit attributes of handwritten word images. The authors try out thee auxiliary tasks: word recognition, word length estimation and character attribute recognition, with word recognition and word length estimation performing best.

2.4.3 Adaptation for speech recognition

A closely related subject to writer-based adaptation for HTR is that of adaptation for automatic speech recognition [12] (ASR). Here too, there is a notion of speaker or writer identity (a particular speaker producing the speech signal), as well as style (e.g., spoken language dialects). Adaptation methods are commonly known under terms like *speaker adaptation* and *accent adaptation*, and sometimes under the broader term of *domain adaptation*. In a similar vein, *speaker enrollment* denotes using speaker-specific data to finetune a network to a particular speaker. Work on adaptive models has mostly been done in speaker adaptation – the adaptation of a model to a target speaker [12]. This method dates back to the 1990s [109], with numerous examples (e.g., [66, 1, 12]).

When it comes to creating compact representations of speaker style, the method proposed in [1] is commonly referenced. This method involves learning *speaker codes*, meant to normalize speaker variation in feature space. The speaker codes are integrated into a pretrained model using a separately trained adaptation network, where the speaker codes and adaptation network are jointly learned. At test time, speaker codes for unknown speakers are created by randomly initializing a new code, followed by backpropagation on a small batch of speaker-specific examples to update the code. Several follow-up papers have been introduced, e.g., adapting the method to CNN architectures [2], and feeding the speaker code into multiple stages of the network [112]. The notion of speaker codes has also successfully been used for speech synthesis [68, 43], i.e., producing acoustic signals based on textual input. The speaker code modulates the output signal based on the specifics of the speaker code.

So-called identity vectors (or i-vectors), estimated using means from Gaussian Mixture Models (GMM) trained on acoustic features [12], have also been used for speaker adaptation (e.g., [84]) and speech synthesis (e.g., [111]). Such a code can be fed in as part of the model input. For example, as auxiliary input in parallel with the regular acoustic featuress [85]. The usage of other contextual information, such as gender and age, has also been attempted for speaker adaptation [68].

2.4.4 Conditionality

The idea of including conditional variables into deep neural networks is commonly seen in the case of conditional generative models [72], where they steer the generating function towards a particular class of outputs. The conditioning variables can be discrete, e.g., generating images conditioned on a particular digit [72], or continuous, e.g., conditioning on a style representation [29, 54] or an image [48].

Another field where conditionality plays a major role is in the field of style transfer. Style transfer refers to the generative process of extracting style information from an image and applying it to the content of another image. Previous work in this field suggests that in order to model a style, it can be sufficient to specialize scaling and shifting parameters after normalization layers, which vary according to the style of the input [26]. Several works have applied this approach to various normalization layers. For example, [54] and [26] both make use of instance normalization layers [100], which are commonly used for normalizing image contrast. This approach normalizes each channel of each image in a minibatch independently, in order to normalize the contrast present in an image. A similar approach applied to the popular batch normalization layer [47] is proposed in [20], called conditional batch normalization. In contrast to instance normalization, batch normalization calculates statistics over an entire batch of images (see Fig 2.2). The conditional batch normalization method was originally proposed for visual question answering, where it is used to modulate high-level image features from a pretrained ResNet by means of a language embedding. In this case, the batch normalization parameters are conditioned on linguistic input, i.e., a vector representation of a question in the visual question answering task.

2.4.5 Transfer learning and domain adaptation

Transfer learning and domain adaptation refer to the situation where what has been learned in one setting is exploited to improve generalization in another setting [31]. Domain adaptation refers to the situation where the input distribution changes slightly (along with the optimal input-output mapping), although the task remains the same. Transfer learning involves learning a different kind of target while the type of input remains the same, by reusing previously learned features for a previous task. It is generally done by pretraining a large neural network on large amounts of training data, and consequently reusing the model for various related tasks, often by replacing the final layers of the network or finetuning parts of the network. The assumption here is that the features learned by pretraining on a large corpus are reusable for other tasks. For vision tasks, pretraining often involves training a CNN or Transformer architecture on an image classification task on the ImageNet database [21, 81]. For Natural Language Processing (NLP), an LSTM [45] or transformer-based model [22, 77] can be trained on large amounts of (unlabeled) natural language.

In the case of NLP, training one model for several languages has shown to lead to useful features across languages, even when training data in the target language is limited ([60, 19]), recently outperforming specialized models for single languages [19] or specific language pairs in the case of machine translation [99]. This means that well-resourced languages can be used to train models that lay the foundation for representations generalizing across languages. What's more, such representations can become more robust and effective by considering data across domains. However, it should be noted that such impressive results often build on a combination of increasingly large models and datasets.

2.4.6 Meta-learning

Meta-learning, or learning-to-learn, is an alternative paradigm to traditional neural network training, which aims to improve the learning algorithm itself [44]. By learning shared knowledge across various tasks over multiple learning episodes, the aim is to improve future learning performance.

The main meta-learning method we focus on here is Model-Agnostic Meta-Learning [28] (MAML). MAML aims to find a parameter initialization such that a small number of gradient updates using a handful of labeled samples produces a classifier that works well on validation data. MAML is related to transfer learning, in the sense that finding good initialization parameters for a model to facilitate adaptation to various tasks plays a central role. Due to its model-agnostic nature, MAML can be applied to various application domains without significant modifications.

Due to the inner/outer-loop optimization process (see Eq. 2.16 and 2.17), MAML has great flexibility in terms of the kinds of parameters that can be learned in the inner loop, e.g., parameterized loss functions [11], learning rates [65], and attenuation weights [9]. Meta-learning has been applied to various areas such as reinforcement learning and few-shot classification, but, notably, also to speech recognition, in the form of accent adaptation [108] and speaker adaptation [57].

MetaSGD [65] is a modification of MAML and involves learning the update direction and learning rate along with the parameter initialization. MAML++ [4] addresses the training instability of MAML that is commonly observed. MAML has also been used in combination with other types of meta-learning. For example, in [82], the authors combine MAML with model-based meta-learning, using a latent generative representation of model parameters and applying MAML in this lower-dimensional latent space.

2.4.7 Writer adaptation

Lastly, we turn to writer-based adaptation in the field of handwritten text recognition. Adaptation to individual writers can be referred to as *writer adaptation*. Many early approaches for writer adaptation are proposed for HMMs using Gaussian Mixture Models. For example, [104] use linear transformations between original parameters and re-estimated parameters for adjusting GMM parameters using maximum likelihood linear regression. More recently, there have been several attempts at adaptation in the space of HTR using neural networks. We list the most relevant work here.

In [73], the authors perform simple finetuning on a new handwriting collection, showing that this can lead to efficient transfer between datasets using a limited amount of finetuning data. In [98], the authors cluster writers by style and train a classifier for each cluster, using a mixture-of-experts setup for choosing the best combination of classifiers. For a new writer, the combination of classifiers is based on classification confidence for that writer. In [116], the authors learn a linear writer-specific feature transformation in order to create a style-invariant classifier, which they call Style Transfer Mapping (STM). Whereas the original approach was not used in the context of neural networks, a later approach [115] has successfully used STM for neural networks in the context of Chinese character recognition. In [106], the authors employ writer codes for writer-specific Chinese handwritten text recognition using a CNN-HMM hybrid model. They feed a writer code into adaptation layers tied to individual convolution layers. The result is added element-wise to the intermediate CNN feature maps. At train time, writer codes are jointly learned with the adaptation layers. At test time, codes for new writers are randomly initialized and optimized using one to three gradient steps. Recently, [105] use a style extractor network trained on a writer identification task to extract a writer code, used to adapt a writer-independent recognizer. Specifically, the writer code is added to the convolutional layer output after being fed through a fully-connected layer.

The writer adaptation problem has also been formulated as a domain adaptation problem [117, 52, 113]. In [117], a gated attention similarity unit is used to find character-level writer-invariant features. In [52], the authors employ an adversarial learning approach using synthetic data. A generic HTR model is initially trained using synthetic data and adapted to new writers using a domain discriminator network.

Chapter 3

Methods

In this chapter, we propose various methods for writer-adaptive HTR and formulate experiments to test their efficacy. Based on extensive experimentation, we aim to answer the research questions posed in Chapter 1, pertaining to three main aspects: 1) how to represent writer information, 2) how to include writer information into an HTR model, and 3) the role of different architectures when it comes to adaptability. By approaching these problems from various angles, we attempt to answer the primary research question of this research: Can state-of-the-art deep learning-based HTR models benefit from writer identity as a conditioning variable?.

The chapter is structured as follows. In Section 3.1, we provide information about the used dataset, along with evaluation metrics in Section 3.2. Details about the training procedure for the base models are presented in Section 3.3. The remaining three chapters propose concrete methods for writer-adaptive HTR. An approach based on writer codes is presented in Section 3.4. Then, an approach based on meta-learning is presented in Section 3.5. Finally, in Section 3.6, we explore an approach based on a method used in domain adaptation.

3.1 Dataset

We now describe the dataset used to train and evaluate HTR models. First and foremost, we focus on recognition of isolated word images. Alternatively, recognition could be done on line strips or full documents. However, both alternatives bring with them additional challenges, such as increased memory requirements and often a lack of data. Therefore, we focus on word images throughout this research.

An important condition for choosing an appropriate dataset is that it should contain the writer identity along with the handwriting images. We therefore make use of the commonly used IAM dataset [69], which contains (anonymized) writer identity information for all its images. The dataset statistics are summarized in Table 3.1.

Table 3.1: Statistics of the IAM dataset, showing the number of images and writers per set.

Set	**# Images**	**# Writers**
Training	47,841	283
Validation	7,520	56
Testing	20,115	161

3.1.1 IAM

The IAM dataset [69] consists of English handwritten texts contributed by 657 writers, making a total of 1,539 handwritten pages consisting of 115,320 segmented words. The data is labeled at the sentence, line, and word level. Examples of word images are shown in Fig. 3.1.

For splitting the data into a training, validation, and test set, we use the widely used Aachen splits [94]. An important property of these splits is that the writer sets are disjoint, i.e., writers seen during training are not seen during testing. The Aachen splits contain 500 writers making up a total of 75,476 images.

Something to note is that the IAM dataset contains additional metadata where images are tagged with a "segmentation" attribute, which indicates whether an error occurred during the original segmentation of the word bounding box from a line strip. In some cases, when this attribute indicates an error, the image does not consist of a word image, but a complete line of text (supposedly since the segmentation failed). In other cases, the segmentation appears fine even when the "segmentation" attribute says otherwise. After running experiments with and without images indicated as badly segmented, we found that excluding the bad segmentation images led to higher performance on the validation set. Therefore, we discard the images that are indicated to have a segmentation error, since they adversely impact recognition performance of the base models. It should be noted that the image count in Table 3.1 therefore does not include the badly segmented images. Also worth noting is that most papers do not report on this detail concerning the IAM dataset, even though there are multiple thousands of images that are tagged with the bad segmentation attribute.

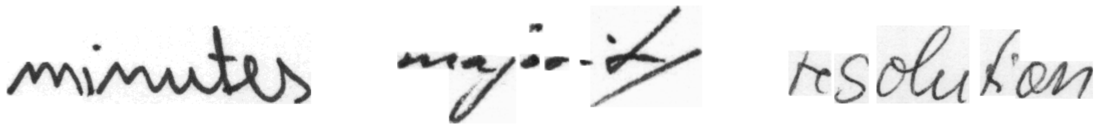


Figure 3.1: Examples of word images from the IAM dataset.

3.1.2 Data augmentation

In order to increase the effective size of the training dataset and the visual diversity of the images, we make use of image augmentations. Augmentations such as rotation, translation, and scaling, are commonly used to improve recognition performance for HTR models (e.g. [76, 27, 92]). For all augmentation modules, we randomly sample their hyperparameters every time they are used, to increase diversity. For a given training image, we use the following image transformations sequentially:

1. Random scaling (scale range: (0.9, 1.1))
2. Rotation (max. angle: 10°)
3. Random brightness and contrast adjustment (max. factor: 0.2)
4. Gaussian noise ($\mu = 0$, $\sigma^2 \in [10, 50]$)

Each transformation has a 50% probability of being applied. Hyperparameter ranges for the transformations are set based on empirical evaluation, and the used hyperparameters are sampled from a uniform distribution every time an augmentation is applied. Furthermore, all images in the dataset (for all splits) are reduced to 50% resolution to reduce memory footprint. This is an acceptable reduction in resolution that still keeps the text legible.

The augmentation stack is kept relatively simple. Other augmentations such as elastic distortions [92] or dilation/erosion are also options that can be part of an augmentation pipeline. The use of synthetic data is also a common practice [27, 51], but not used in the current research.

3.2 Evaluation

For evaluation, character error rate (CER) and word error rate (WER) are used. Given a ground truth label sequence and a predicted label sequence, CER is defined as the character-level Levenshtein distance between the two strings divided by the total number of characters in the ground truth (N_c). The Levenshtein distance is defined as the sum of the substitutions (S_c), insertions (I_c), and deletions (D_c) to transform one string into the other. Thus, we have:

$$\text{CER} = \frac{S_c + I_c + D_c}{N_c} \tag{3.1}$$

Similarly, WER is calculated as the word-level Levenshtein distance divided by the number of words in the ground truth label sequence:

$$\text{WER} = \frac{S_w + I_w + D_w}{N_w} \tag{3.2}$$

Note that we use a different subscript in this case, to indicate that the substitutions, insertions, deletions, and the token count occur at the word level.

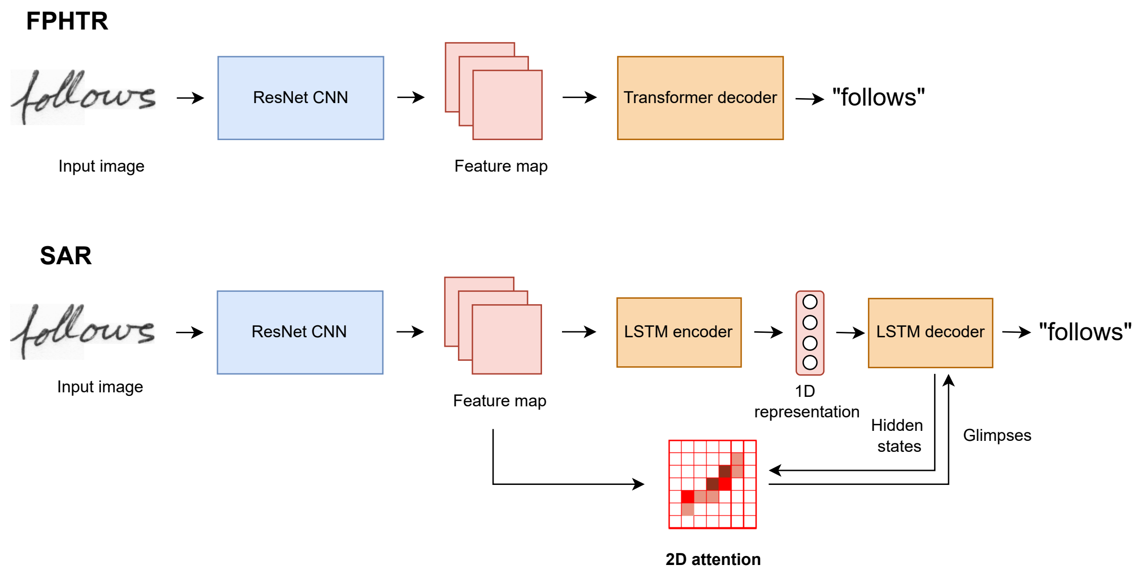


Figure 3.2: Schematic overview of the two base models: FPHTR and SAR.

3.3 Base models

We make use of two baseline models: FPHTR [93] and SAR [61], as explained in Section 2.2.2 and 2.2.1. As a reminder: FPHTR builds on the Transformer architecture and SAR on the LSTM architecture. In Fig. 3.2, we show a schematic overview of both models to highlight their overall structure and similarity. For both models, we use two versions: a smaller version using an 18-layer ResNet and a larger version with a 31-layer ResNet (see Appendix B.1 for parameter counts).

The base models are standard HTR models that do not make use of explicit writer information, chosen based on their competitive performance on common benchmarks. Their performance serves as a baseline for "writer-unaware" HTR models. In the following sections, various methods will be proposed that make these models writer-aware, in order to verify whether this can lead to improved recognition accuracy. In the following section, we explain how these models are trained.

3.3.1 Training procedure

We do not use a word lexicon, in order to provide maximum flexibility in decoding and preventing the issue of "out of lexicon" words. Instead, we use a character-level vocabulary, converting all characters to lower case. This results in a vocabulary of size 56, containing ASCII-printable characters as well as special tokens to demarcate beginning and end of sequence and padding. For sampling characters, we use greedy decoding, i.e., choosing the character with the highest probability at each time step.

Images within a batch are padded with 0 values to the size of the maximum image width and length in the batch. Ground truth character sequences are padded to the maximum sequence length in the batch.

We do not use any kind of linguistic post-processing on word predictions, such as using a statistical language model for better accuracy.

For training, the Adam optimizer is used [56], with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ (default configuration). We use gradient clipping to avoid exploding gradients, based on the L2-norm of the gradient vector.

All models are implemented using PyTorch [75], using a single Nvidia V100 GPU with 32GB of memory.

Best models are chosen based on the validation word error rate (WER). The performance for a single run is recorded as the epoch with the highest performance. See appendix A.1 for full details about hyperparameter settings.

3.4 Writer codes

Our first attempt to include writer information into the base HTR models is based on the idea of representing style or writer information as a compact feature vector. In speech recognition, such a code is known as a *speaker code* [1]. We take a similar approach by trying to model writers or styles using a small feature vector, which is used to adapt the weights of an existing HTR model. We will refer to such vectors as *style codes* or *writer codes*.

In its simplest form, the writer identity for the k-th writer could be expressed as a *one-hot vector* $\mathbf{x}^k \in \mathbb{R}^N$,

$$\mathbf{x}_i^k = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

where N is the total number of writers. This approach however is inherently limiting because it assumes a predetermined set of writers, and the writer codes have little representational capacity. Therefore, we focus on *dense* representations, also sometimes known as *distributed* representations [31]. In this case, the writer identity is represented by many features in the form of a densely populated feature vector $\mathbf{x}^k \in \mathbb{R}^M$, where M is set based on the desired representational capacity.

We experiment with style codes that have varying levels of granularity. At the lowest level of granularity, this means supplying information concerning the high-level writing style, e.g., if a writer writes cursively or not. At a more granular level, we would provide information that is highly writer-specific. As mentioned in Chapter 1, this is not so much a question about what information is useful for transcribing handwritten text *in general*, but rather a question about what information is not yet learned effectively by a deep learning-based HTR model.

A relevant property of writer codes is that it should be able to obtain them even for writers that are not part of the training set (note that this is effectively forced by our dataset, since the writers in IAM are mutually exclusive in the train, validation and test sets).

Writer codes have various properties that make them appealing as a method for writer-adaptive HTR. They can be studied and compared among writers since they are compact in size. This potentially allows studying the factors of variation that are distinctly the result of writer individuality. Moreover, most methods that we will discuss using writer codes are computationally efficient and require relatively little modification of the original base architectures.

This section is structured as follows. In Section 3.4.1, we first discuss suitable ways to insert a writer code into the base models. Then, we turn to the problem of creating the writer codes. In Section 3.4.2, we propose a method for learning writer codes jointly with other parameters. Next, we turn to a more traditional approach for representing writer information in Section 3.4.3, using handcrafted features for writer identification. If writers can be clustered in some sort of representative feature space it may also be possible to find generic style codes, which we explore in Section 3.4.4.

3.4.1 Code insertion

First, we attempt to answer the question how the codes should be inserted into the base model for effective adaptation. This is non-trivial, since adaptation based on such codes requires updating the weights of the network without catastrophic forgetting of previously learned representations. Where to insert the writer code into the network is indirectly a question about what features are most suitable for adaptation, and what features are effectively shared among writers. For example, the sequence decoding stage of the model may not benefit much from writer adaptation, since language representations may not change considerably for different writers. Based on various experiments, we found that naive insertion of style codes into the base models could easily deteriorate performance. Therefore, this requires a more delicate approach to preserve the original representations.

One possible approach is proposed in [106], where the authors generate an additional bias vector generated from a writer code, which is added to the convolutional layer output right before the batch normalization layer. The bias vector is generated by a linear transform of the writer code. However, due to the immediate normalization following the addition of the writer-dependent bias vector, the expressiveness of the writer code seems to be fairly limited, which was confirmed by our experiments with this approach.

When it comes to deciding what parts of the models should be adapted based on writer information, the ResNet backbone seems like a suitable location, since it pertains to the visual features that can differ among writers, e.g., individual letters written in an idiosyncratic way. We can apply adaptation to the output of the ResNet backbone, which contains high-level image features over a 2D spatial grid. Modulating these high-level features based on writer information seems promising since it does not affect the low-level feature representations, and only applies adaptive finetuning of the visual features at their final stage. First, note that all base models make use of a ResNet image encoder that produces a 2-dimensional feature map (see Fig. 3.2), which means that this approach can be applied to both models. Let the extracted feature map be $Q \in \mathbb{R}^{C \times h' \times w'}$, with h' and w' representing height and width of the feature map, and C the number of channels. By flattening the feature map in the temporal dimension to create $R \in \mathbb{R}^{C \times (h' \cdot w')}$, we can represent the feature map as an array of feature vectors $H = [f_1, f_2, \ldots, f_{h' \cdot w'}]$ with $f_i \in \mathbb{R}^C$. Given this array of feature vectors H containing high-level image representations and a writer code w, we can adapt each vector f_i based on the writer code by concatenating the feature vector with the conditional code and passing it through an MLP:

$$f'_i = \phi([f_i; w]) \tag{3.4}$$

where f'_i is the adapted feature vector and ϕ is an MLP. The new feature map $H' = [f'_1, f'_2, \ldots, f'_{h' \cdot w'}]$ then contains all adapted feature vectors f'_i.

Although this approach is appealing in its conceptual simplicity, it limits the expressivity of the adaptation, i.e., its ability to alter the previously learned representations, since it occurs only in a single location. Therefore, we additionally experimented with approaches where codes are inserted in multiple stages of the network. However, this can quickly lead to catastrophic forgetting if the original weights are jointly finetuned with the new adaptation weights, or suboptimal interaction between the code features and the original base model weights in the case when the base model weights are frozen. For example, one of the major pain points is the presence of batch normalization in the ResNet encoder (see Section 2.1.3). Retraining the γ and β parameters or reinitializing running batch statistics tends to have a drastic impact on performance and can quickly deteriorate previously learned representations.

In order to find a suitable method for inserting style information into neural networks, we take inspiration from works on generative models using style information, such as conditional GANs [54, 118] and methods for style transfer [26, 100]. In these research areas, style information plays an important role in modulating image features along a visual processing pipeline. As mentioned in Section 2.4.4, previous work in the field of style transfer suggests that in order to adapt features to a particular style, it can be sufficient to specialize scaling and shifting parameters after normalization layers, conditioned on style information [26]. However, such style transfer architectures often make use of instance normalization layers rather than batch normalization, which is the normalization layer used in our base models. In contrast to batch normalization, instance normalization normalizes each channel of each image in a minibatch independently, to normalize image contrast. Batch normalization, on the other hand, calculates statistics over an entire batch of images (see Fig. 2.2).

We therefore experiment with an alternative method designed for batch normalization. Conditional batch normalization [20] was originally proposed for visual question answering, in order to modulate high-level image features from a ImageNet pretrained ResNet using a language embedding (see Fig. 3.3). In the original application, the batch normalization parameters are conditioned on linguistic input in the form of a question from the visual question answering task. A regular batch normalization layer is of the form shown in Eq. 2.6, with trainable β and γ parameters of size C. In the typical case of batch normalization following a convolutional layer, C equals the number of kernels in the convolutional layer. The idea behind conditional batch normalization is to modify the β and γ parameters based on an input code. Given pretrained parameters β_c and γ_c, changes in these parameters are

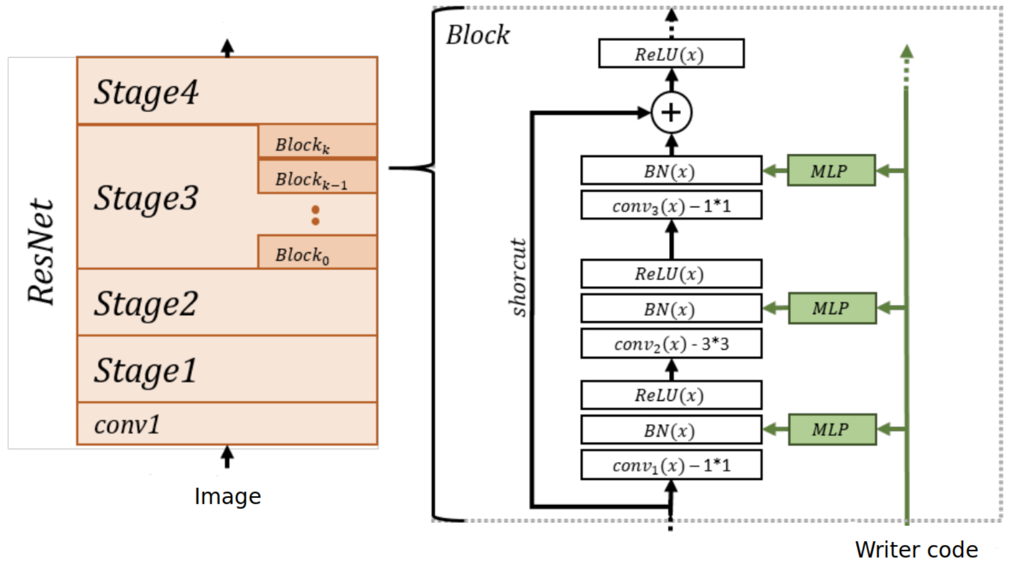


Figure 3.3: Schematic of conditional batch normalization as applied to the ResNet architecture. The parameters for each Batch Norm layer are modulated by a writer code, using a two-layer MLP. Image modified from [20].

predicted based on an input code e and a two-layer MLP:

$$\Delta\beta = \phi_1(e) \qquad \Delta\gamma = \phi_2(e) \tag{3.5}$$

where ϕ_1 and ϕ_2 are MLPs. The predicted deltas are then added to the original β_c and γ_c parameters:

$$\hat{\beta}_c = \beta_c + \Delta\beta_c \qquad \hat{\gamma}_c = \gamma_c + \Delta\gamma_c \tag{3.6}$$

where $\hat{\beta}_c$ and $\hat{\gamma}_c$ replace the Batch Norm parameters for the current forward pass. All other parameters are frozen during training, including β and γ. Note that because of this, during training, only the MLPs associated with the Batch Norm layers are trained, while all other parameters are kept frozen.

By changing the γ and β affine parameters that follow normalization, there is great flexibility in changing the intermediate feature maps according to the specifics of a particular code. Feature maps produced by convolutional layers can be modified by scaling them up or down, negating them, shutting them off, etc. By freezing the original batch normalization parameters and predicting changes to them, it is straightforward to preserve the original parameters by producing a zero output. Because of this, the risk of catastrophic forgetting is mitigated.

Based on experiments, the conditional batch normalization approach proved more effective as well as more stable than the single insertion approach presented in Eq. 3.4. Therefore, we use this approach going forward.

3.4.2 Learned codes

Now that we have laid out a method for inserting writer codes into a HTR model, we turn to the question of how to actually *create* writer codes.

One way to obtain learned writer codes is to extract them from a bottleneck layer in a neural network trained to distinguish between writers [105]. However, such a network would be trained under a closed writer set assumption: Given N possible writers, the model only has to learn representations that distinguish those N writers. This is a fundamentally different task than that of finding representations that encode relevant writer information for all possible writers.

Another, more promising approach is to learn writer codes in the same way as the weights of the network, i.e., using gradient descent. A similar idea is commonly seen in NLP (e.g., [22]), where for each token in a predefined vocabulary, an associated vector representation is learned (often called an "embedding") that is more expressive than, for example, a one-hot vector indicating the identity of the token.

For creating learned codes, we therefore build on this idea. Specifically, we take inspiration from an approach originating from automatic speech recognition [1]. This approach was originally applied in the case of a hybrid NN-HMM model, where the neural network (NN) acts as a feature extractor. In the original approach, the principal idea is to insert a speaker code into a pretrained network using an adaptation MLP at the start of the network (as is done in Eq. 3.4). This is shown visually in Fig. 3.4.

The main aspect we are interested in here is the creation of the code itself, which we do by applying the approach presented in the paper by Abdel-Hamid et al. [1]. In [1], codes are trained in the same way as the weights of the adaptation MLP, namely by taking the derivative of the loss with respect to the speaker code values and updating them using gradient descent. Note that this implies a fixed set of writer codes initialized at the start of training – one for each speaker in the training set. In the case when a new writer is presented that is unseen during training, the solution proposed in [1] is to randomly initialize a new writer code, followed by one or several gradient steps on the newly initialized code, using a small batch of labeled speaker-specific data. In other words, a separate batch of labeled data is required to initialize the speaker code for a new speaker. Note that this is similar to the episodic learning scheme used for MAML (see Section 2.3.1), where a task-specific batch is used to adapt the model parameters to the task, after which inference can be done on that new task using the task-adapted model.

3.4.3 Hinge codes

When it comes to capturing writer individuality, there exists a rich literature on this topic in the field of writer identification [88]. In contrast to the learned features discussed in the previous section, features for writer identification are often handcrafted or statistical in nature.

One of the more successful features for writer identification is the Hinge feature [16]. As mentioned in Section 2.4.2, the Hinge feature uses a probability distribution of the angle combination of two hinged edge fragments to characterize writer individuality. If these features

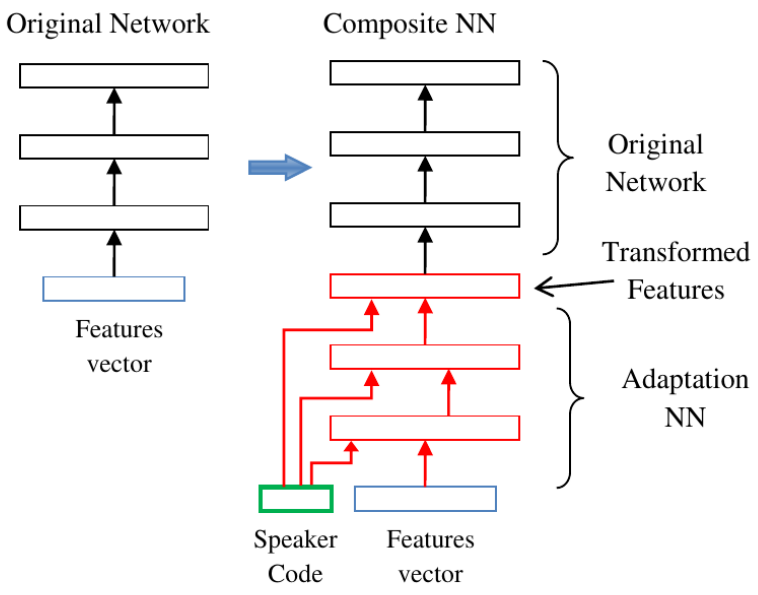


Figure 3.4: Schematic of the approach presented in [1], originally applied to automatic speech recognition. On the left side is a network without speaker adaptation, whereas the network on the right side does include speaker adaptation. A speaker code is inserted into a dedicated MLP (adaptation NN), which takes both the original features and the speaker code as input, and outputs transformed features.

can lead to a meaningful clustering of writers based on their style differences, they can potentially serve as meaningful writer codes. These writer codes are attractive because they are 1) easy to calculate, and 2) do not require additional adaptation data at test time, as is the case for the learned codes discussed in Section 3.4.2. We obtain Hinge features for each writer by concatenating all images for that particular writer, after which we apply Hinge feature extraction to it.

3.4.4 Style codes

Given the backdrop of the Hinge features discussed in the previous section, there is an open question of how writer-specific the codes should be. Potentially, it could be more meaningful to focus on generic style clusters in feature space, rather than features that are highly writer-specific. For example, style clusters could point to high-level writing styles such as cursive or mixed-cursive. If Hinge provides us with meaningful style clustering, it can be used to initialize generic style codes.

We perform k-means clustering on Hinge codes to obtain generic style clusters. For each style cluster, we train a writer code using backpropagation. Thus, given an image input, we find the closest style cluster based on the Hinge features and map the style cluster identity to a learned writer code that is updated using gradient descent. One of the more appealing properties of this approach is that it does not require initialization of new codes for unseen writers, as is the case for the approach presented in Section 3.4.2.

3.4.5 Training procedure

For all writer codes, we insert them using the conditional batch normalization method presented in Section 3.4.1. Hyperparameters are chosen based on validation set performance.

For the learned writer codes discussed in Section 3.4.2, we require adaptation data at test time to initialize codes for novel writers. Given the j'th writer with N_j total examples, we randomly split the data into an adaptation batch of size K, and use the remaining $N_j - K$ examples for evaluation using the learned writer code. To reduce the effect of randomness, we repeat this procedure 10 times for every test writer and take the mean performance. During training, the weights of the trained HTR model are frozen, and only the writer code values and the parameters of the Conditional Batch Norm MLPs are updated. We use a code size of 64 and an adaptation batch of size 16.

Training and testing for Hinge and style codes is done in the standard, non-episodic way. For style codes, we use k-means clustering with $k = 3$. Complete hyperparameters are shown in Appendix A.2.

Full hyperparameter settings for writer code models are shown in Appendix A.2).

3.5 Meta-learning

Our second attempt to make HTR models writer-adaptive uses meta-learning, a research area that has gathered much interest in recent years [44]. In this case, adaptation occurs by providing the model with labeled examples of a writer that the model should adapt to, which is done by updating all the weights in the model. Thus, the approach requires a handful of labeled data at test time. This can be seen as an efficient form of transfer learning: Given a small amount of writer-specific labeled data not seen during training, this data should be used to improve writer-specific performance in a way that is maximally efficient in making use of previously learned representations.

We build on top of the MAML algorithm (see Section 2.3.2) and use the modifications as proposed in [13]. One downside of the MAML approach and MetaHTR in particular is that it leads to a notable increase in memory and computational requirements. We therefore analyze variations of the MAML-based approach to investigate to what degree it can be simplified. Concretely, we experiment with three different models: MAML, MAML + llr, and MetaHTR.

1. **MAML**. The original MAML algorithm as proposed in [28], using the sequence-based cross-entropy loss function shown in 2.11.

2. **MAML + llr**. The MAML algorithm is complemented with learnable inner loop learning rates, as explained in Section 2.3.3. This alleviates the need to manually set the inner loop learning rate, at the cost of only a few hundred additional parameters (see Appendix B.3)

3. **MetaHTR**. The full MetaHTR model as explained in Section 2.3.3. A downside of the MetaHTR approach is the additional complexity that it introduces. Next to the calculation of higher-order derivatives as part of the MAML algorithm, MetaHTR also requires an additional backward pass in order to calculate the instance-specific weights. This makes the approach expensive both in terms of computation and in terms of memory usage, therefore making it challenging to scale to larger contexts such as sentence-level HTR.

3.5.1 Training procedure

In the K-shot N-way meta-learning formulation, we use $K = 16$ and $N = 8$, as is done in [13]. This means that during adaptation, a batch of $K = 16$ writer-specific examples are used to adapt the model to a specific writer, and outer loop gradients are averaged over $N = 8$ writers (see Eq. 2.17). For adaptation, we use a single inner loop optimization step, as additional steps did not seem to improve performance further.

During training, we randomly sample writer-specific batches of size $2K$, split into a support and query set of size K. At test time, we use all examples for a given writer: Given the j'th writer with N_j total examples, we randomly split the data into a support batch (adaptation batch) of size K, and use the remaining $N_j - K$ examples for evaluation of the adapted model. To reduce the effect of randomness, we repeat this procedure 10 times for every test writer and take the mean performance. Furthermore, we ensure that the used splits remain constant with respect to each writer.

For all models, we use dropout in the outer loop. Batch Norm statistics are fixed to their running values and not updated during training, as this led to more stable performance (see Appendix C for a more extensive discussion concerning the particulars of using Batch Norm in combination with MAML). We use the learn2learn library [6] for implementing all meta-learning methods. Full hyperparameter settings are shown in Appendix A.3.

3.6 Domain adaptation

Our final attempt for writer-adaptive HTR is based on an approach employed in domain adaptation. We established in Section 3.4.1 that batch normalization parameters are some of the most effective parameters to tune for modulating the convolutional layers of a ResNet architecture based on a writer code. Whereas conditional batch normalization focuses on modulating the affine β and γ parameters, the accuracy of the mean and variance used for normalization also forms a potentially important part of the effectiveness of batch normalization [47]. Therefore, we explore whether adaptation could be done by modifying the running statistics of the batch normalization layers based on the writer-specific mean and variance. As a reminder: during training time, a running average of the activation statistics is collected for each batch normalization layer. At test time, batch statistics are replaced with saved running statistics.

This approach has been used before in domain adaptation [58]. Within the field of domain adaptation, the aim is to transfer learned model features from a source domain to a new, target domain. It has been proposed that this could be done by modulating batch normalization statistics from the source domain to the target domain in all batch normalization layers across a model [64, 63, 18]. Furthermore, this approach has also been applied to improve robustness against image corruptions [86]. If we view a novel writer analogously to a domain shift, i.e., a distribution shift, we can apply the same idea here, applying writer-specific normalization statistics at each Batch Norm layer. As shown in Fig. 3.5, the difference between writer-specific activation statistics is substantial enough to make this plausible.

We use a variation of the approach proposed in [64], named adaptive batch normalization (AdaBN). This approach involves a complete replacement of source statistics with target statistics. However, completely replacing the writer-independent Batch Norm statistics can be harmful to model performance. Therefore, we calculate the new statistics by taking a weighted combination of writer-independent and writer-dependent statistics, as done in [86]. For a given writer w and activation neuron j, we have:

$$\begin{aligned} \bar{\mu}_j^w &= (1-\alpha)\mu_j^w + \alpha\mu_j \\ (\bar{\sigma}_j^w)^2 &= (1-\alpha)(\sigma_j^w)^2 + \alpha(\sigma_j)^2 \end{aligned} \tag{3.7}$$

where $\alpha \in [0, 1]$, μ_j^w and $(\sigma_j^w)^2$ are writer-dependent statistics, and μ_j and $(\sigma_j)^2$ are writer-independent statistics. The hyperparameter α controls the trade-off between writer-dependent and writer-independent statistics.

This approach is straightforward and simple: It does not require additional training of model parameters and can be calculated efficiently using existing implementations for batch normalization [1]. Note that setting $\alpha = 1$ is the same as not using writer-specific statistics, i.e., using the default statistics for the base models. We select the α hyperparameter by picking the best performing value on the validation set. For all settings, we use a batch size of 256.

[1]We make use of PyTorch's `torch.nn.BatchNorm2d` module for efficiently calculating both writer-dependent and writer-independent statistics.

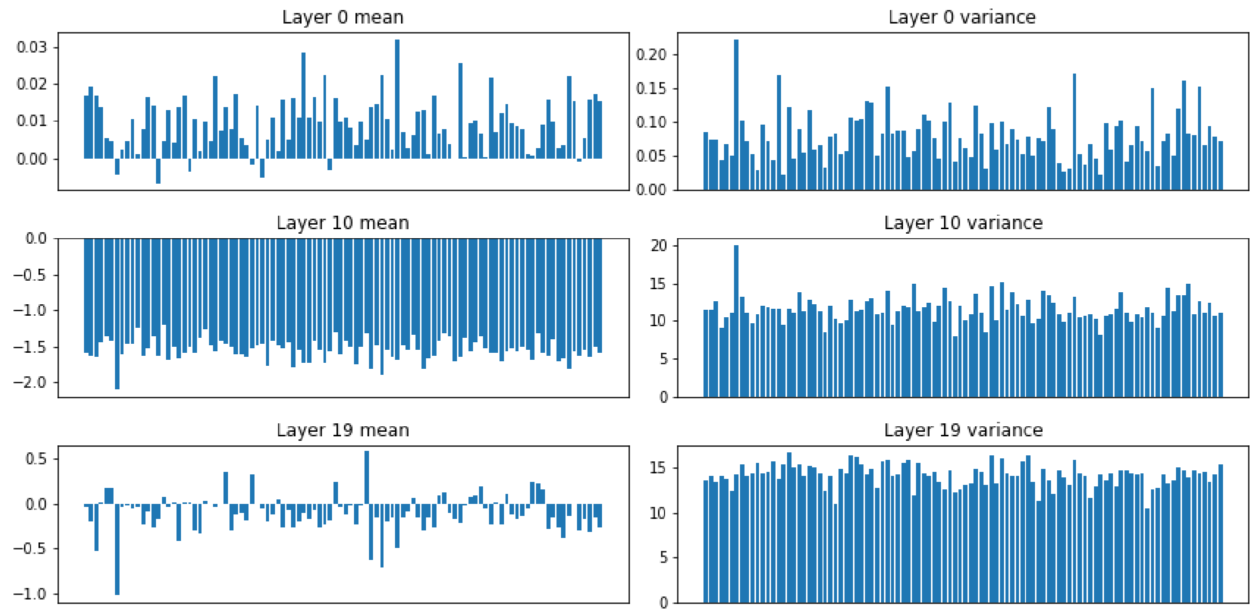


Figure 3.5: ResNet-18 layer activation statistics for 100 randomly selected writers from IAM. The x-axis indicates writer identity. Activations are recorded right before batch normalization occurs.

Chapter 4

Results

In this chapter, we report results for all models and experiments proposed in Chapter 3. A direct comparison between models is not always possible, as some models (such as MetaHTR) use a different learning scheme that relies on additional adaptation data at test time. Therefore, we indicate the closest baseline on an individual basis.

For all results, we report average performance over 5 different random seeds, along with standard deviations. For the purpose of rapid experimentation, we mainly perform experiments using the smaller SAR-18 and FPHTR-18 models. Whenever it is deemed relevant we also include results for the larger 31-layer variants of the base models. We select the best performing models based on lowest word error rate (WER).

It is worth noting that due to the way image padding is employed in the pipeline (all images are padded to the maximum size in the batch), the batch size influences the degree of padding, which can have an impact on performance. Therefore, we show base model results that use the same amount of padding as the models to which they are compared.

4.1 Base models

The results for the base models on the IAM validation and test set are shown in Table 4.1 and visually in Fig. 4.1. We report average performance as well as performance of the best run.

From the results in Table 4.1 and Fig 4.1, we can see that the Transformer-based model (FPHTR) outperforms the LSTM-based model (SAR) on validation and test, both for the smaller 18-layer case (15-18M weights) and the larger 31-layer case (52-58M weights). This difference is significant in the case of the larger 31-layer models, with FPHTR outperforming SAR on test with a difference of 4.1 WER and 4.8 CER. For the smaller 18-layer models, FPHTR outperforms SAR by a difference of 0.5 WER and 0.7 CER.

Table 4.1: Results of the base models on the IAM val and test set.

	Val				**Test**			
	WER		**CER**		**WER**		**CER**	
	Avg.	Best	Avg.	Best	Avg.	Best	Avg.	Best
SAR-18	16.3 ± 0.6	15.5	13.5 ± 1.0	12.2	20.7 ± 0.8	19.7	17.3 ± 0.8	15.8
FPHTR-18	$\mathbf{16.0 \pm 0.4}$	15.3	$\mathbf{12.6 \pm 0.4}$	12.1	$\mathbf{20.2 \pm 0.2}$	19.9	$\mathbf{16.6 \pm 0.3}$	16.4
SAR-31	14.9 ± 0.2	14.7	11.3 ± 0.5	10.6	19.7 ± 0.7	18.8	15.7 ± 1.0	14.5
FPHTR-31	$\mathbf{11.6 \pm 0.3}$	11.1	$\mathbf{7.9 \pm 0.4}$	7.5	$\mathbf{15.6 \pm 0.8}$	14.6	$\mathbf{10.9 \pm 0.7}$	10.0

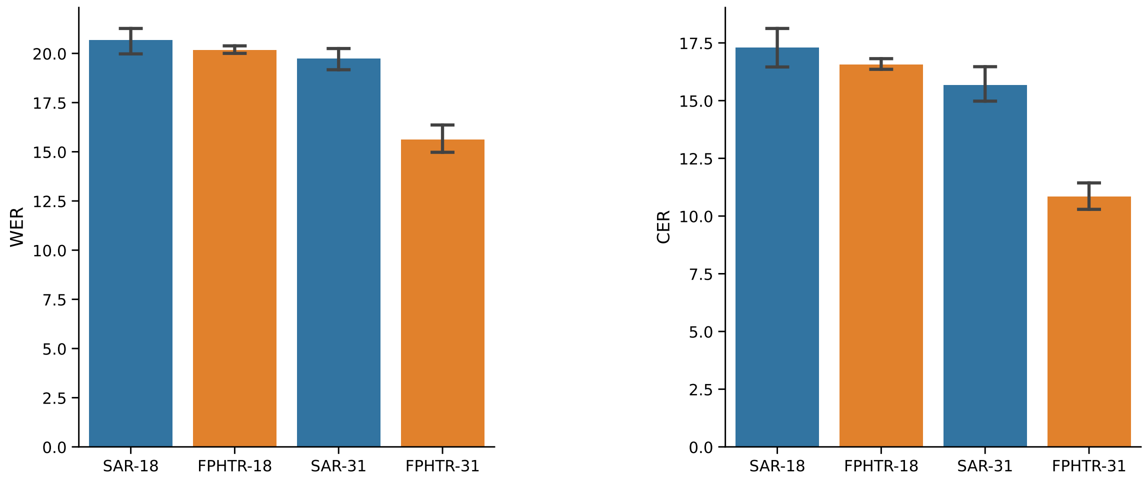


Figure 4.1: Base model results on the IAM test set, measured in WER (left) and CER (right). Confidence intervals are obtained by bootstrapping.

4.2 Writer codes

We now consider the results of the writer code approach as discussed in Section 3.4. The idea of writer codes is attractive due to its relative simplicity and limited additional computational costs. We show results for all writer codes in Table 4.2.

From the table, it can be seen that the learned codes created using backpropagation do not improve upon the performance of the baseline. The fact that writer codes at test time are created by random initialization followed by a small number of gradient steps is a potential factor here, since the writer codes obtained during training have been updated for many more gradient steps. Potentially, the writer codes used during training may contain more relevant information as they have been finetuned over several epochs. However, training error alone is not a good metric to test the efficacy of the writer codes obtained during training, as a low training error may simply be a result of overfitting. In order to gain more insight into the learned writer codes during training, we plot the codes for a single run of the FPHTR-18 model (Fig. 4.2). Although it is difficult to draw hard conclusions from such a plot, the lack of meaningful style clusters seems to suggest that the writer codes are not used effectively by the model.

Next, we consider Hinge and style codes. Both methods outperform the baseline. For the Hinge code, this is a difference of 1.7 and 1.6 WER for FPHTR and SAR, respectively. A similar performance improvement can be seen for the style code, obtained by clustering Hinge codes with a single learned code per style cluster. In this case, the difference is 1.8 and 1.7 WER for FPHTR and SAR, respectively.

Although these results show improvement compared to the baselines, these results alone do not provide adequate insight into the efficacy of the codes themselves. Recall from Section 3.4.1 that Conditional Batch Norm uses a 3-layer MLP with the writer codes as input to predict changes to the original Batch Norm weights. In theory, it is possible that the MLP can learn effective bias vectors that improve performance regardless of the writer code input, i.e., the writer code could simply be ignored (e.g., assigned zero weights). In order to test this, we replace the writer codes with a single code that contains no writer information whatsoever: a zero code, i.e., a vector with only zero values. As can be seen from Table 4.2, this leads to almost identical performance compared to both the Hinge and style code. This is a strong indication that writer information is not the direct cause of the increase in performance, but rather, Conditional Batch Norm simply seems to be an effective way to finetune the Batch Norm weights, even without the presence of conditional information. Although this is an indication that tuning the Batch Norm weights is an effective way to finetune a trained HTR model, this result does not rely on writer information to make it possible.

Table 4.2: Writer code results on the IAM test set, measured in WER.

	FPHTR-18	SAR-18
Baseline	20.2 ± 0.2	20.7 ± 0.8
Learned code	24.5 ± 0.3	23.7 ± 0.4
Hinge code	$\mathbf{18.5 \pm 0.2}$	$\mathbf{19.1 \pm 0.6}$
Style code	$\mathbf{18.4 \pm 0.2}$	$\mathbf{19.0 \pm 0.6}$
Zero code	$\mathbf{18.5 \pm 0.3}$	$\mathbf{19.0 \pm 0.5}$

4.3 Meta-learning

We now turn to the models based on MAML [28], discussed in Section 3.5. These models are the most complex and require more computation to run.

It should be noted that since all models presented here make use of additional adaptation data at test time, a direct comparison between the base models in Table 4.1 is not directly meaningful. In other words, the MAML-based models have access to parts of the test data as

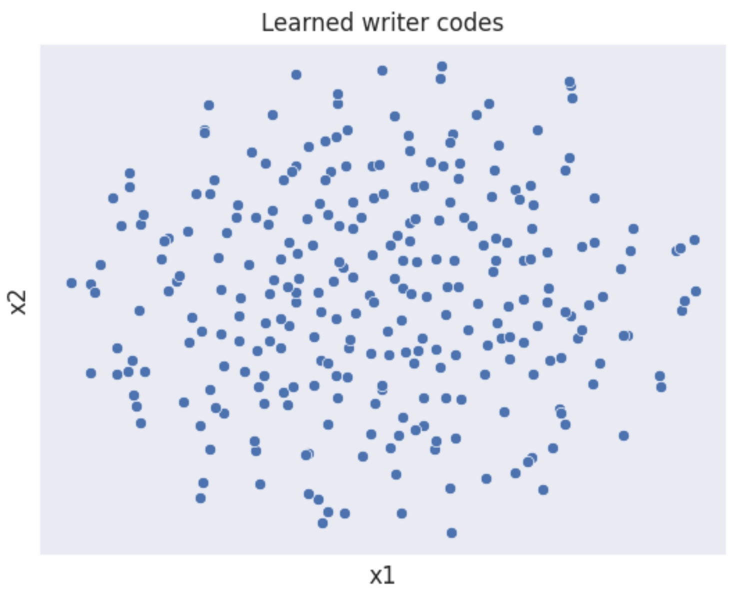


Figure 4.2: Writer codes learned through backpropagation for FPHTR-18, reduced to two dimensions using t-SNE [101]. Although it is difficult to draw hard conclusions from such a plot, the lack of meaningful style clusters seems to suggest that the writer codes are not used effectively by the model.

part of their adaptation procedure. Therefore, we devise a different baseline, by evaluating the base models after performing finetuning on the same adaptation data that is made available to the MAML-based models. Specifically, we finetune the final classification layer of a base model using the adaptation data. We use the Adam [56] optimizer with a learning rate of 1e-3 for 3 optimization steps. Results including these baselines are shown in Table 4.3. Due to persistent out-of-memory errors for the SAR-31 MetaHTR model, we only include FPHTR-31 in addition to the smaller 18-layer variants. From these results, we can see that MetaHTR performs best, improving upon the baseline by 1.4, 2.0, and 1.7 WER for FPHTR-18, SAR-18, and FPHTR-31, respectively.

We can plot the learned inner loop learning rates to get an idea of the relative weight assigned to each layer in the adaptation process (Fig. 4.3). We show learned inner loop learning rates for two randomly chosen runs of the FPHTR-18 and FPHTR-31 models using MAML + llr. Looking at these plots, we can observe an increasing trend in the Transformer learning rates across layers. This is an indication that the lower layers of the Transformer network require relatively fewer adaptation than layers closer to the output, with the final classification layer requiring the most adaptation. Furthermore, we can see that some of the layers in the Transformer sequence decoder are adapted with approximately the same magnitude as many of the ResNet layers. In some sense, this can be interpreted as counter-evidence for the hypothesis that the ResNet backbone is the most important part of the base models for adaptation, as suggested in Section 3.4.1.

It is worth noting that MetaHTR requires calculation of instance-specific gradients, which is at the time of writing something that is not supported in batch form in the PyTorch library. Therefore, this required a manual calculation of instance-specific gradients using a for-loop, which made the MetaHTR training procedure considerably slower than MAML. This problem is something that can be fixed using additional software, but the additional complexity due to the extra backward pass remains.

It is worth noting here that the performance gains for MetaHTR (an improvement between 1.4 to 2.0 WER compared to the baseline) are much smaller than reported in the original paper [13], where MetaHTR improved upon the SAR baseline by a difference of 7.1 WER, and 6.8 after naive finetuning on the adaptation data. In email correspondence with the authors of the MetaHTR paper, we were not able to resolve the cause of this discrepancy. This ambiguity is exacerbated by the absence of published code for MetaHTR.

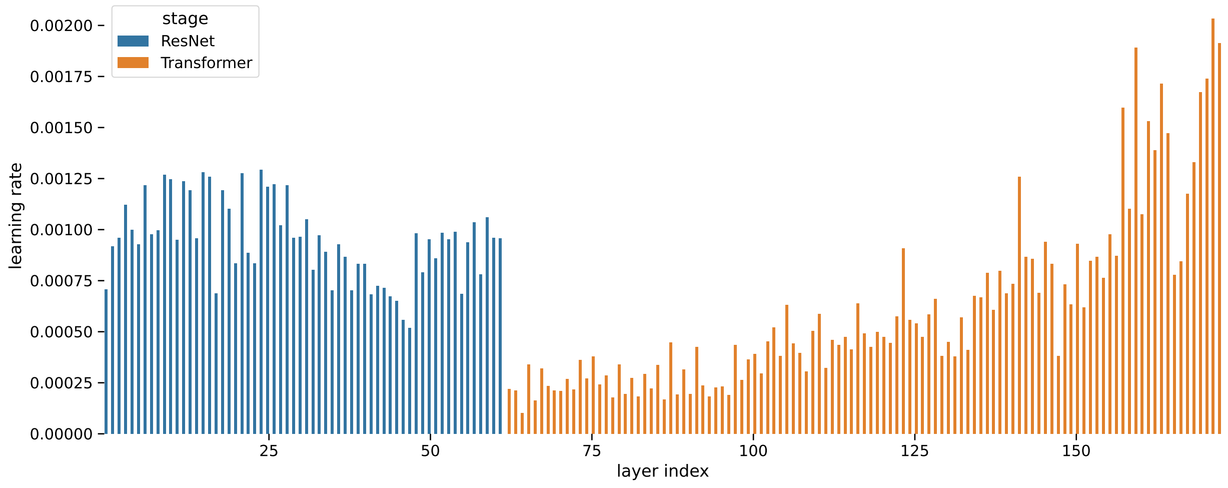
Table 4.3: Meta-learning results on the IAM test set, measured in WER.

	FPHTR-18	SAR-18	FPHTR-31
Baseline	20.0 ± 0.2	20.6 ± 0.6	15.3 ± 0.7
MAML	19.1 ± 0.3	19.5 ± 0.7	14.3 ± 0.3
MAML + llr	19.3 ± 0.5	19.3 ± 0.7	14.3 ± 0.2
MetaHTR	$\mathbf{18.6 \pm 0.4}$	$\mathbf{18.6 \pm 0.5}$	$\mathbf{13.5 \pm 0.2}$

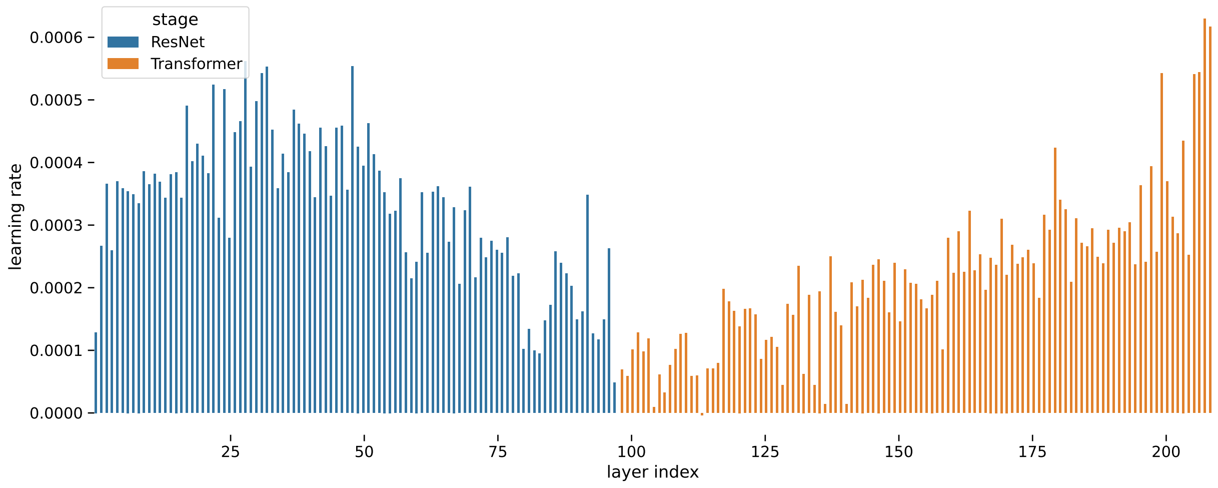
4.3.1 Testing the adaptation premise of MetaHTR

An important question concerning the efficacy of MetaHTR is to what degree it truly *adapts* based on a set of writer-specific images at test time. This is an important premise, since the additional computational overhead of MetaHTR as well as the increased complexity compared to regular neural network training is supposedly warranted by a clear goal: An ability to adapt in a flexible way to various writers leading to a concrete performance improvement compared to a writer-unaware model. In the words of the authors, the goal of MetaHTR is to offer a "adapt to my writing button" [13], where one is asked to write a specific sentence in order to make recognition performance of that handwriting more accurate.

It has been suggested that feature reuse is an important part of the success of MAML, more so than rapid adaptation [78]. According to [78], in the case of few-shot adaptation, the inner loop adaptation could be skipped for most of the layers of the network. The head of the network, however, requires inner loop adaptation to enable task specificity. Note that this is a result of the nature of the few-shot classification task. In order to adapt a base model to a new few-shot classification task, it is necessary to calibrate the weights of the final output layer, as the output neurons may correspond to different classes, depending on the task. However, in the current context, this is not strictly necessary, as the output neurons (corresponding to



(a)



(b)

Figure 4.3: Learned per-layer learning rates for the MAML + llr model, for (a) FPHTR-18 and (b) FPHTR-31.

ASCII characters) stay the same across tasks. Therefore, it is not obvious whether the inner loop adaptation plays an important role in the performance of MetaHTR.

In order to test this, we perform an additional experiment where we leave out the inner loop adaptation at test time. More concretely, we train MetaHTR the same way as done before, but evaluate it without performing inner loop adaptation on a support batch of K images. Results are shown in Table 4.4. The additional benefit of adaptation is 0.2 WER for FPHTR-18, 0.7 WER for SAR-18, and 0.7 WER for FPHTR-31. Since the sample mean of the performance on the test set across different random seeds will approximately follow a normal distribution (according to the central limit theorem), we can use a two-sample t-test to measure the statistical significance of the difference in results. Using a significance level $\alpha = 0.05$, we observe that the difference in results is not significant for FPHTR-18 ($p = 0.4143$) and SAR-18 ($p = 0.0832$), but *is* significant for FPHTR-31 ($p = 0.0001$). In other words, adaptation only shows a significant effect for the larger FPHTR-31 model, but not for the smaller 18-layer variants.

Table 4.4: MetaHTR performance with and without writer adaptation, measured in WER.

	FPHTR-18	SAR-18	FPHTR-31
w/ adaptation	18.6 ± 0.4	18.6 ± 0.5	13.5 ± 0.2
w/o adaptation	18.8 ± 0.4	19.3 ± 0.5	14.2 ± 0.2

4.4 Domain adaptation

Lastly, we turn our attention to the domain adaptation approach as presented in Section 3.6. The results for AdaBN are shown in Table 4.5. From these results it is clear that AdaBN does not produce a tangible performance improvement, performing the same or slightly worse than the baseline on the test set. Most of the time (70% of all results), the best results were obtained by setting the α weighing parameter equal to 1, meaning that it was most effective to use the writer-independent statistics and to ignore the writer-specific information.

Table 4.5: Results for the AdaBN approach on the IAM test set, measured in WER.

	FPHTR-18	SAR-18
Baseline	20.2 ± 0.3	20.5 ± 0.8
AdaBN	20.2 ± 0.3	20.6 ± 0.7

Chapter 5

Discussion and conclusion

In this section, we discuss the findings from Chapter 4 in more depth. Then, in the conclusion, we use our findings to answer the research questions posed in Chapter 1.

5.1 Discussion

5.1.1 Writer codes

The results in Table 4.2 show the limited effectiveness of the writer code idea. It is difficult to isolate the usefulness of the writer codes themselves since the performance is determined by both 1) the writer code itself and 2) the way the writer code is inserted into the model. We chose conditional batch normalization as an approach for inserting writer codes, based on credible literature in related fields indicating that adaptation of batch normalization parameters lends itself well to vector-based adaptation. Furthermore, this approach mitigates the risk of catastrophic forgetting. However, at the same time, it does not provide any kind of adaptation for either the Transformer or LSTM sequence decoder following the ResNet backbone (see Fig. 3.2), since these architectures do not use batch normalization. In a more ideal case, the writer code would be inserted into all stages of the full architecture, not just in the CNN backbone. Therefore, it is possible that there may be a more refined way to effectively insert the writer codes. As long as the insertion scheme is not optimal, it is difficult to isolate the usefulness of the writer codes themselves. Effectively, this is a chicken-and-egg problem: Measuring the effectiveness of the writer codes requires knowing the effectiveness of the insertion scheme, and vice versa.

Compare this to the MAML approach, where all the weights in a model can be adapted. This allows for more flexibility in deciding how the model representations should change to allow for effective adaptation. It could be argued that this approach better fits the overarching trend of deep learning, moving away from handcrafted solutions to solutions that can be learned from data. Some even argue that meta-learning, or in this case more fittingly, learning-to-learn, is the next step of integrating joint feature, model, and *algorithm* learning [44].

Then, there is the question of how writer information should be represented in the first

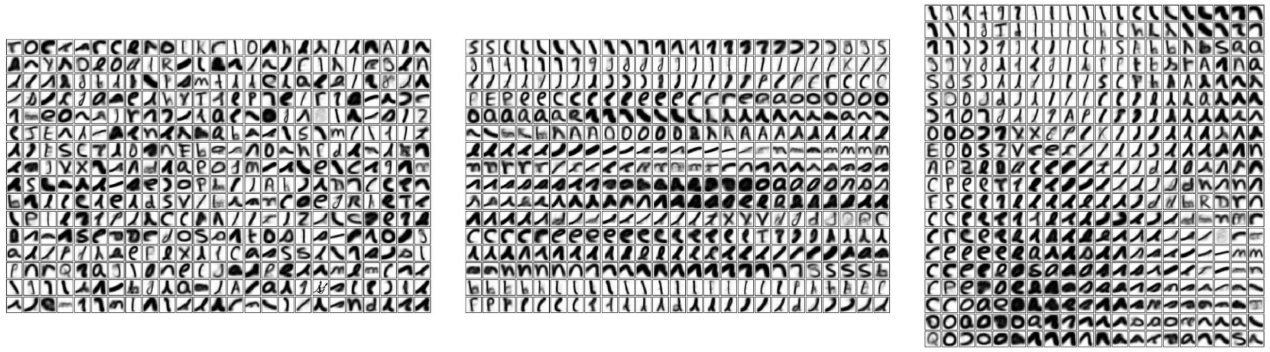


Figure 5.1: Examples of codebooks that capture shape information based on clustering of character shapes. The codebook entries act as prototypes representative of the types of shapes commonly seen in handwriting. Figure taken from [16].

place. We showed that statistical features for characterizing writer identity do not appear to show a benefit over a constant zero vector. The fact that the Hinge feature is designed to be independent of the textual context of the handwriting samples may play a role here [16]. An option for future work would be to explore features that lend themselves better to characterize the most relevant writer characteristics, such as idiosyncratic letter shapes that are difficult to classify. For example, a Fraglet approach based on shape codebooks [16] may capture the individual shape features of a particular handwriting more appropriately (see Fig. 5.1). By matching codebook prototypes with the character shapes observed for an individual writer, a histogram can be compiled counting the matched codebook entries. The normalized histogram can subsequently be used as a vector representation. Still, this leaves open the problem of applying the writer-specific vector in an appropriate way to the previously learned model weights.

Despite the limited success of the conditional context vector for the HTR models, the use of context vectors is highly successful for generative deep learning models. Why is this the case? This can perhaps best be understood by looking at the difference in the role of the context vector in generative and discriminative models. In generative models, the context vector provides a way to constrain the output of a stochastic process. Put another way, it reflects a *decision* by the user that constrains an otherwise random process, e.g., the decision of the ethnicity of a person when generating images of human faces. Compare this to the case of discriminative models such as the HTR models presented in this research. The model output is deterministic, in the sense that any particular input will always produce the same output (given that the training process is completed, of course). There is no notion of stochasticity because there should only be a single correct output, e.g., the correct transcription for an input image containing handwritten text. In this case, the context vector only serves to make the implicit more explicit, rather than providing a signal source that could not be learned by the model itself since it is non-deterministic.

It is interesting to draw parallels between the field of HTR and that of automatic speech recognition, where context vectors for, e.g., speaker adaptation, seem to be more common.

Just like most areas of deep learning, research in speech recognition shows how large-scale data is important for obtaining features that generalize well [55], e.g., for transferability to other languages. This is relevant because one facet in which speech and text recognition diverge is in the availability of large-scale labeled datasets. Whereas collecting and labeling handwriting samples can be cumbersome and labor-intensive, speech transcriptions are generally easier to obtain. Thus, if data volume is the critical bottleneck for learning robust representations that lend themselves well to adaptation, methods used in speech recognition relying on large-scale datasets may not transfer as well to HTR.

5.1.2 Meta-learning

Table 4.3 shows that meta-learning improves upon the performance of the baseline models. Several aspects make this approach appealing. First, the MAML framework allows for adaptation that does not rely on the insight of the human practitioner. Most notably, by adding a small number of learnable parameters such as the learnable layer-wise learning rates explained in Section 2.3.3, there is a great deal of flexibility in the way the model can adapt to a writer. This is shown in Fig. 4.3, which shows how such learning rates take shape. Considering that deep learning models build upon low-level concepts to create more complex ones in a hierarchical way, it makes sense to adjust parameters by taking into account their layer-wise position in the network (as opposed to using a single learning rate, as is done for standard SGD). MAML allows learning such parameters *from the data*, without relying on human insight.

Learning (hyper)parameters directly from training data is an appealing property that is often mentioned within a Bayesian context [15]. In the Bayesian framework, prior distributions can be set for various hyperparameters, followed by marginalization with respect to these variables. However, practical implementations of this method often revert to approximations. This is due to the computational complexity of marginalizing over continuous probability distributions, scaling exponentially with the number of parameters and quickly becoming intractable for high-dimensional cases.

Nevertheless, the added benefit of writer-adaptation using MetaHTR is not always obvious, as shown in Section 4.3.1. Even without using any adaptation data at test time, the MetaHTR model still improves upon the baseline performance. This is an indication that more effective feature reuse plays a role in the additional performance gains, rather than rapid adaptability of the model parameters – a phenomenon that has been observed before in the literature on meta-learning [78].

A downside of the MetaHTR approach is the additional complexity that it introduces. Next to the calculation of higher-order derivatives as part of the MAML algorithm, MetaHTR also requires an additional backward pass to calculate the instance-specific weights (as explained in Section 2.3.3). This makes the approach expensive both in terms of computation and memory usage and makes it challenging to scale to larger contexts such as sentence-level HTR. This is exemplified by the fact that we were not able to train MetaHTR in combination with the SAR-31 base model due to persistent out-of-memory errors.

The principle of Occam's razor may perhaps be applied here. Common benchmarks can be deceptive, and a simpler solution should in principle be preferred over one that is more complex. Along with the additional complexity of MetaHTR, training of MetaHTR requires a good deal of finetuning of various settings and hyperparameters to make it work well, which is something that has also been observed for MAML more broadly [4]. We showed in Table 4.2 that a relatively simple procedure like finetuning of Batch Norm parameters can lead to performance gains competitive with the meta-learning approach. Of course, it cannot be excluded that this improvement is orthogonal to the benefits of the MAML approach – they are not necessarily mutually exclusive. However, the additional complexity introduced by the MetaHTR approach is something to be taken into account and should be weighed with its potential benefit.

The results in Table 4.4 seem to indicate that a deeper model lends itself better to adaptation using MetaHTR than a shallower one. A similar conclusion has also been found in a recent meta-learning study [5], which showed that MAML adapts better with deep architectures, even if the tasks need only a shallow one. However, as mentioned before, scaling up MAML-based approaches to deeper HTR models brings additional challenges in terms of memory requirements, as well as increased computation time.

Something that was not studied in the current research is how varying the size of the adaptation batch impacts performance, although this is mentioned in the original MetaHTR paper [13]. One difficulty in using small sample sizes for adaptation is that the most relevant information for adaptation may not always be present. For example, knowing that a writer writes a particular letter in an idiosyncratic way makes it more relevant to see examples of that particular letter, whereas other, more generically written letters (such as the letter "o"), are not as rich in information. As part of future work, a more extensive analysis of the relationship between the adaptation sample size and the performance using adaptation should shed more light on the efficacy and potential benefit of the MetaHTR approach.

An interesting question is how well MetaHTR can adapt to more radical changes in handwriting, such as handwriting seen in historical collections. Digitization efforts could benefit from more flexible HTR solutions that can adapt based on changes in writer, but also changes in other attributes such as script and background conditions. This could lend itself to a lifelong learning setup [89], whereby new models would not have to be trained for every new collection of handwritten documents. A change in script poses an additional challenge, since the output classes, pertaining to individual characters, change in this case. Nevertheless, MAML-based learning could potentially be applied in this case as well. Recall that the traditional few-shot classification problem setting using MAML generally requires replacing the existing classification head of a network with a newly initialized one, which can be trained in a handful of gradient updates using the support batch. In a similar vein, a new classification head could be trained using MAML that reflects the dictionary set of a novel script. On the other hand, using only a small adaptation batch will most likely not cover all characters in a script, which means that some of the weights of the new classification head would not be trained at all. This would then require some sort of feature reuse of the initial classification head to make this feasible. In the case of the addition of a small number of additional characters, however, this approach seems more feasible (e.g., the addition of the ß character when switching from

English to German). This is an interesting possibility for future work. It is good to note in this case that switching to a different language will most likely require adaptation of the implicit language model, which may be more difficult to perform using only a single batch of data.

5.1.3 Domain adaptation

It can be seen from Table 4.5 that the domain adaptation approach using AdaBN does not provide a meaningful benefit. The AdaBN layer was originally proposed to alleviate the problem of domain shift, i.e., how layer activation statistics diverge for different domains. Although we showed in Fig 3.5 that writer-specific activation statistics tend to show a good deal of variation, it may be the case that the impact of a new writer compared to the impact of a completely new domain such as a different dataset is not comparable, which makes the method less effective, and sometimes even harmful to performance. Even though handwriting from unseen writers may be difficult to transcribe accurately if the handwriting is sufficiently novel, the performance decrease is generally not as steep as in the case of transfer from one domain to another. In such cases, it may be relatively easier to improve upon an unadapted baseline, as naive transfer of a deep learning model from one dataset to another can lead to serious performance degradation [73].

Furthermore, the question can be asked what the added benefit is of using writer-specific normalization statistics. After all, the claim that limiting distributional shift plays a major role in the effectiveness of Batch Norm has been disputed [83]. Replacing the writer-independent statistics with statistics for specific writers will lead to activations that are closer to having zero mean and unit variance, but at the same time, it is clear from the results that placing large emphasis on the writer-specific statistics also interferes with the learned representations in a way that is detrimental to performance.

5.2 Conclusion

In this thesis, we proposed various methods for integrating writer information into deep learning-based models for offline handwritten text recognition. To conclude, we now refer back to the research questions posed in Chapter 1 and try to answer them. First, we go through the secondary research questions, after which we attempt to answer our primary research question.

RQ1: How can writer information be represented in a way that is effective for facilitating adaptation and improving recognition performance?

We found that the most effective way to represent writer information is in the form of labeled data examples that can be used to directly update the weights of an HTR model, as is done in MAML-based approaches such as MetaHTR. Although compact representations of writers in the form of writer-specific vectors are conceptually appealing, these vectors are difficult to create effectively.

RQ2: What is a suitable method to include writer information in a state-of-the-art deep learning-based HTR architecture?

Out of all the methods for writer-based adaptation covered in this research, the MAML-based models showed the most promising results. Effectively, this implies that letting the model *learn* how to adapt its weights using methods such as learnable learning rates is a promising direction. This is more flexible than manually deciding which parts of the model architecture should be adapted and which parts should be left unchanged.

RQ3: Does architectural choice play a meaningful factor in facilitating effective adaptation?

For the MAML-based models, the depth of the base model is potentially an important factor to allow for effective adaptation. Therefore, when applying MAML-based methods to HTR, it seems reasonable to assume that architectures with more layers should be preferred over more shallow architectures.

Given these conclusions, we can attempt to answer our main research question: Can state-of-the-art deep learning-based HTR models benefit from writer identity as a conditioning variable? Overall, there seems to be a small benefit to writer-adaptation using methods based on meta-learning. Nevertheless, these improvements are not substantial, and it remains to be seen whether MAML could be used to handle more radical domain shifts such as historical handwriting. In a practical setting, the additional effort needed to implement such a method, both in terms of computational cost and other factors such as collecting sufficiently many adaptation samples, may not weigh up to the potential gains that these methods offer. However, in cases where high-precision recognition is crucial, writer-specific adaptation using MetaHTR may still provide a meaningful performance boost.

Methods based on writer codes did not show promising results. Even though this method has been used in more shallow neural networks using HMMs, it's possible that the deep hierarchical representations learned by state-of-the-art neural networks make it difficult to meaningfully add information that is not already learned by the model itself.

One (perhaps obvious) conclusion that can be made is that the problem of effectively integrating writer information into deep learning-based HTR models is a non-trivial one. This is not to say that integrating writer information – or conditional variables more broadly – into HTR models is not useful, but there are many ways in which tinkering with previously learned representations can prove problematic. Therefore, it seems reasonable to conclude that interjections based on human insight about how an HTR model "should" adapt are to be seen with a healthy dose of skepticism. If there is one thing that the success of deep learning has shown, it is perhaps that learning from data is best done by minimizing human judgments, while maximizing the capability of a model to learn autonomously.

Bibliography

[1] Ossama Abdel-Hamid and Hui Jiang. Fast speaker adaptation of hybrid nn/hmm model for speech recognition based on discriminative learning of speaker code. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7942–7946. IEEE, 2013.

[2] Ossama Abdel-Hamid and Hui Jiang. Rapid and effective speaker adaptation of convolutional neural network based models for speech recognition. In *INTERSPEECH*, pages 1248–1252, 2013.

[3] Mahya Ameryan and Lambert Schomaker. A limited-size ensemble of homogeneous cnn/lstms for high-performance word classification. *Neural Computing and Applications*, 33(14):8615–8634, 2021.

[4] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. In *International Conference on Learning Representations*, 2018.

[5] Sébastien Arnold, Shariq Iqbal, and Fei Sha. When maml can adapt fast and how to assist when it cannot. In *International Conference on Artificial Intelligence and Statistics*, pages 244–252. PMLR, 2021.

[6] Sébastien M R Arnold, Praateek Mahajan, Debajyoti Datta, Ian Bunner, and Konstantinos Saitas Zarkias. learn2learn: A library for meta-learning research. August 2020.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[9] Sungyong Baik, Seokil Hong, and Kyoung Mu Lee. Learning to forget for meta-learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2379–2387, 2020.

[10] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.

[11] Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

[12] Peter Bell, Joachim Fainberg, Ondrej Klejch, Jinyu Li, Steve Renals, and Pawel Swietojanski. Adaptation algorithms for neural network-based speech recognition: An overview. *IEEE Open Journal of Signal Processing*, 2:33–66, 2021.

[13] Ayan Kumar Bhunia, Shuvozit Ghose, Amandeep Kumar, Pinaki Nath Chowdhury, Aneeshan Sain, and Yi-Zhe Song. Metahtr: Towards writer-adaptive handwritten text recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15830–15839, 2021.

[14] Anne-Laure Bianne-Bernard, Fares Menasri, Rami Al-Hajj Mohamad, Chafic Mokbel, Christopher Kermorvant, and Laurence Likforman-Sulem. Dynamic and contextual information in hmm modeling for handwritten word recognition. *IEEE transactions on pattern analysis and machine intelligence*, 33(10):2066–2080, 2011.

[15] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[16] Marius Bulacu and Lambert Schomaker. Text-independent writer identification and verification using textural and allographic features. *IEEE transactions on pattern analysis and machine intelligence*, 29(4):701–717, 2007.

[17] Sukalpa Chanda, Jochem Baas, Daniel Haitink, Sébastien Hamel, Dominique Stutzmann, and Lambert Schomaker. Zero-shot learning based approach for medieval word recognition using deep-learned features. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 345–350. IEEE, 2018.

[18] Woong-Gi Chang, Tackgeun You, Seonguk Seo, Suha Kwak, and Bohyung Han. Domain-specific batch normalization for unsupervised domain adaptation. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition*, pages 7354–7362, 2019.

[19] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. *arXiv preprint arXiv:1911.02116*, 2019.

[20] Harm De Vries, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron C Courville. Modulating early visual processing by language. *Advances in Neural Information Processing Systems*, 30, 2017.

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Maruf A Dhali, Jan Willem de Wit, and Lambert Schomaker. Binet: Degraded-manuscript binarization in diverse document textures and layouts using deep encoder-decoder networks. *arXiv preprint arXiv:1911.07930*, 2019.

[24] Daniel Hernandez Diaz, Siyang Qin, Reeve Ingle, Yasuhisa Fujii, and Alessandro Bissacco. Rethinking text line recognition models. *arXiv preprint arXiv:2104.07787*, 2021.

[25] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[26] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. *arXiv preprint arXiv:1610.07629*, 2016.

[27] Kartik Dutta, Praveen Krishnan, Minesh Mathew, and C.V. Jawahar. Improving cnn-rnn hybrid networks for handwriting recognition. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 80–85, 2018.

[28] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[29] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[30] 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*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[32] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[33] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[34] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Multi-dimensional recurrent neural networks. In *International conference on artificial neural networks*, pages 549–558. Springer, 2007.

[35] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. *Advances in neural information processing systems*, 21:545–552, 2008.

[36] DR Guérillot and J Bruyelle. Uncertainty assessment in production forecast with an optimal artificial neural network. In *SPE Middle East oil & gas show and conference*. OnePetro, 2017.

[37] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[40] Sheng He and Lambert Schomaker. Deep adaptive learning for writer identification based on single handwritten word images. *Pattern Recognition*, 88:64–74, 2019.

[41] Sheng He and Lambert Schomaker. Fragnet: Writer identification using deep fragment networks. *IEEE Transactions on Information Forensics and Security*, 15:3013–3022, 2020.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[43] Nobukatsu Hojo, Yusuke Ijima, and Hideyuki Mizuno. An investigation of dnn-based speech synthesis using speaker codes. In *INTERSPEECH*, pages 2278–2282, 2016.

[44] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

[45] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[46] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems*, 30, 2017.

[47] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[48] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[49] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. *Advances in neural information processing systems*, 28, 2015.

[50] Andrew Jaegle, Sebastian Borgeaud, Jean-Baptiste Alayrac, Carl Doersch, Catalin Ionescu, David Ding, Skanda Koppula, Daniel Zoran, Andrew Brock, Evan Shelhamer, Olivier J Henaff, Matthew Botvinick, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. Perceiver IO: A general architecture for structured inputs & outputs. In *International Conference on Learning Representations*, 2022.

[51] Lei Kang, Pau Riba, Marçal Rusiñol, Alicia Fornés, and Mauricio Villegas. Pay attention to what you read: Non-recurrent handwritten text-line recognition. *arXiv preprint arXiv:2005.13044*, 2020.

[52] Lei Kang, Marçal Rusinol, Alicia Fornés, Pau Riba, and Mauricio Villegas. Unsupervised writer adaptation for synthetic-to-real handwritten word recognition. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3502–3511, 2020.

[53] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[54] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[55] Kazuya Kawakami, Luyu Wang, Chris Dyer, Phil Blunsom, and Aaron van den Oord. Learning robust and multilingual speech representations. *arXiv preprint arXiv:2001.11128*, 2020.

[56] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[57] Ondřej Klejch, Joachim Fainberg, and Peter Bell. Learning to adapt: a meta-learning approach for speaker adaptation. *arXiv preprint arXiv:1808.10239*, 2018.

[58] Wouter M Kouw and Marco Loog. A review of domain adaptation without target labels. *IEEE transactions on pattern analysis and machine intelligence*, 43(3):766–785, 2019.

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[60] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[61] Hui Li, Peng Wang, Chunhua Shen, and Guyu Zhang. Show, attend and read: A simple and strong baseline for irregular text recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8610–8617, 2019.

[62] Minghao Li, Tengchao Lv, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. Trocr: Transformer-based optical character recognition with pre-trained models. *arXiv preprint arXiv:2109.10282*, 2021.

[63] Yanghao Li, Naiyan Wang, Jianping Shi, Xiaodi Hou, and Jiaying Liu. Adaptive batch normalization for practical domain adaptation. *Pattern Recognition*, 80:109–117, 2018.

[64] Yanghao Li, Naiyan Wang, Jianping Shi, Jiaying Liu, and Xiaodi Hou. Revisiting batch normalization for practical domain adaptation. *arXiv preprint arXiv:1603.04779*, 2016.

[65] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[66] Hank Liao. Speaker adaptation of context dependent deep neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7947–7951. IEEE, 2013.

[67] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[68] Hieu-Thi Luong, Shinji Takaki, Gustav Eje Henter, and Junichi Yamagishi. Adapting and controlling dnn-based speech synthesis using input codes. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4905–4909. IEEE, 2017.

[69] U-V Marti and Horst Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5(1):39–46, 2002.

[70] Paul Mermelstein and Murray Eyden. A system for automatic recognition of handwritten words. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, pages 333–342, 1964.

[71] Johannes Michael, Roger Labahn, Tobias Grüning, and Jochen Zöllner. Evaluating sequence-to-sequence models for handwritten text recognition. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1286–1293. IEEE, 2019.

[72] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[73] Rathin Radhakrishnan Nair, Nishant Sankaran, Bharagava Urala Kota, Sergey Tulyakov, Srirangaraj Setlur, and Venu Govindaraju. Knowledge transfer using neural network based approach for handwritten text recognition. In *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*, pages 441–446. IEEE, 2018.

[74] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[75] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[76] Joan Puigcerver. Are multidimensional recurrent layers really necessary for handwritten text recognition? In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 01, pages 67–72, 2017.

[77] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

[78] Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of maml. In *International Conference on Learning Representations*, 2020.

[79] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[80] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[81] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[82] Andrei A Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. In *International Conference on Learning Representations*, 2018.

[83] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *Advances in neural information processing systems*, 31, 2018.

[84] George Saon, Hagen Soltau, David Nahamoo, and Michael Picheny. Speaker adaptation of neural network acoustic models using i-vectors. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 55–59. IEEE, 2013.

[85] George Saon, Hagen Soltau, David Nahamoo, and Michael Picheny. Speaker adaptation of neural network acoustic models using i-vectors. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 55–59. IEEE, 2013.

[86] Steffen Schneider, Evgenia Rusak, Luisa Eck, Oliver Bringmann, Wieland Brendel, and Matthias Bethge. Improving robustness against common corruptions by covariate shift adaptation. *Advances in Neural Information Processing Systems*, 33:11539–11551, 2020.

[87] Lambert Schomaker. *Patronen en symbolen: een wereld door het oog van de machine.* s.n., 2002.

[88] Lambert Schomaker. Advances in writer identification and verification. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 1268–1273. IEEE, 2007.

[89] Lambert Schomaker. *Lifelong learning for text retrieval and recognition in historical handwritten document collections.* World Scientific Publishing, 2021.

[90] Lambert Schomaker and Marius Bulacu. Automatic writer identification using connected-component contours and edge-based features of uppercase western script. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(6):787–798, 2004.

[91] Baoguang Shi, Xiang Bai, and Cong Yao. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence*, 39(11):2298–2304, 2016.

[92] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[93] Sumeet S Singh and Sergey Karayev. Full page handwriting recognition via image to sequence extraction. In *International Conference on Document Analysis and Recognition*, pages 55–69. Springer, 2021.

[94] Open SLR. Aachen data splits (train, test, val) for the iam dataset. https://www.openslr.org/56/.

[95] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[96] Jorge Sueiras, Victoria Ruiz, Angel Sanchez, and Jose F Velez. Offline continuous handwriting recognition using sequence to sequence neural networks. *Neurocomputing*, 289:119–128, 2018.

[97] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[98] Martin Szummer and Christopher M Bishop. Discriminative writer adaptation. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[99] Chau Tran, Shruti Bhosale, James Cross, Philipp Koehn, Sergey Edunov, and Angela Fan. Facebook ai wmt21 news translation task submission. *arXiv preprint arXiv:2108.03265*, 2021.

[100] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[101] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[102] Tijn Van der Zant, Lambert Schomaker, and Koen Haak. Handwritten-word spotting using biologically inspired features. *Ieee transactions on pattern analysis and machine intelligence*, 30(11):1945–1957, 2008.

[103] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[104] Alessandro Vinciarelli and Samy Bengio. Writer adaptation techniques in hmm based off-line cursive script recognition. *Pattern Recognition Letters*, 23(8):905–916, 2002.

[105] Zi-Rui Wang and Jun Du. Fast writer adaptation with style extractor network for handwritten text recognition. *Neural Networks*, 147:42–52, 2022.

[106] Zi-Rui Wang, Jun Du, and Jia-Ming Wang. Writer-aware cnn for parsimonious hmm-based offline handwritten chinese text recognition. *Pattern Recognition*, 100:107102, 2020.

[107] Curtis Wigington, Seth Stewart, Brian Davis, Bill Barrett, Brian Price, and Scott Cohen. Data augmentation for recognition of handwritten words and lines using a cnn-lstm network. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 639–645. IEEE, 2017.

[108] Genta Indra Winata, Samuel Cahyawijaya, Zihan Liu, Zhaojiang Lin, Andrea Madotto, Peng Xu, and Pascale Fung. Learning fast adaptation on cross-accented speech recognition. *arXiv preprint arXiv:2003.01901*, 2020.

[109] Phil C Woodland. Speaker adaptation for continuous density hmms: A review. In *ISCA Tutorial and Research Workshop (ITRW) on Adaptation Methods for Speech Recognition*, 2001.

[110] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[111] Zhizheng Wu, Pawel Swietojanski, Christophe Veaux, Steve Renals, and Simon King. A study of speaker adaptation for dnn-based speech synthesis. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[112] Shaofei Xue, Ossama Abdel-Hamid, Hui Jiang, and Lirong Dai. Direct adaptation of hybrid dnn/hmm model for fast speaker adaptation in lvcsr based on speaker code. In *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 6339–6343. IEEE, 2014.

[113] Hong-Ming Yang, Xu-Yao Zhang, Fei Yin, Jun Sun, and Cheng-Lin Liu. Deep transfer mapping for unsupervised writer adaptation. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 151–156. IEEE, 2018.

[114] Weixin Yang, Lianwen Jin, and Manfei Liu. Deepwriterid: An end-to-end online text-independent writer identification system. *IEEE Intelligent Systems*, 31(2):45–53, 2016.

[115] Xu-Yao Zhang, Yoshua Bengio, and Cheng-Lin Liu. Online and offline handwritten chinese character recognition: A comprehensive study and new benchmark. *Pattern Recognition*, 61:348–360, 2017.

[116] Xu-Yao Zhang and Cheng-Lin Liu. Writer adaptation with style transfer mapping. *IEEE transactions on pattern analysis and machine intelligence*, 35(7):1773–1787, 2012.

[117] Yaping Zhang, Shuai Nie, Wenju Liu, Xing Xu, Dongxiang Zhang, and Heng Tao Shen. Sequence-to-sequence domain adaptation network for robust text image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2740–2749, 2019.

[118] Zhenxing Zhang and Lambert Schomaker. Divergan: An efficient and effective single-stage framework for diverse text-to-image generation. *Neurocomputing*, 473:182–198, 2022.

[119] Yiwei Zhu, Shilin Wang, Zheng Huang, and Kai Chen. Text recognition in images based on transformer with hierarchical attention. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1945–1949, 2019.

Appendix A

Hyperparameters

In this section, we include all relevant hyperparameters used to train the models in Chapter 3. We show hyperparameters for the base models in Table A.1, hyperparameters for writer code models in Table A.2, and meta-learning hyperparameters in Table A.3.

Table A.1: Hyperparameters for the base HTR models.

	FPHTR-{18,31}	**SAR-18**	**SAR-31**
Batch size	32	32	32
Learning rate	1e-4	1e-3	1e-3
Adam β_1	0.9	0.9	0.9
Adam β_2	0.999	0.999	0.999
d_model	260	-	-
Feedforward hidden size	1024	-	-
Hidden size LSTM encoder	-	256	512
Hidden size LSTM decoder	-	256	512
Attention module dim.	-	256	512
Dropout encoder	0.1	0.1	0.1
Dropout decoder	0.1	0.0	0.0
Transformer heads	4	-	-
Transformer layers	6	-	-
LSTM encoder layers	-	2	2
LSTM decoder layers	-	2	2
Max sequence length	55	55	55
Max gradient L2-norm	-	5.0	5.0

Table A.2: Hyperparameters for the writer code approach.

	Learned code	Hinge code	Style code	Zero code
Learning rate	1e-3	1e-3	1e-3	1e-3
Learning rate codes	1e-3	-	1e-3	-
AdaBN MLP hidden units	128	128	128	128
Batch size	128	64	64	64
Code size	64	465	64	465
Shots (K)	16	-	-	-
Num. clusters (k)	-	-	3	-

Table A.3: Hyperparameters for the meta-learning approach.

	MAML / MAML + llr			MetaHTR		
	FPHTR-18	SAR-18	FPHTR-31	FPHTR-18	SAR-18	FPHTR-31
Learning rate (β)	3e-5	1e-4	3e-5	8e-6	1e-4	8e-6
Inner learning rate (α)	1e-4	1e-3	1e-4	-	-	-
MLP (g_ψ) hidden units	-	-	-	128	128	128
Shots (K)	16	16	16	16	16	16
Ways (N)	8	8	8	8	8	8
Num. inner steps	1	1	1	1	1	1
Max gradient L2-norm	5.0	5.0	5.0	5.0	5.0	5.0

Appendix B

Number of parameters per model

We indicate learable parameter counts for all models below. Base model parameters are shown in Table B.1, whereas additional parameters required for each approach in Chapter 3 are shown in Tables B.2 and B.3.

Table B.1: Total number of trainable parameters per base model. For each model, the total parameter count is decomposed into the constituent submodules.

	# parameters
FPHTR-18	**17.8M**
ResNet	11.3M
Transformer decoder	6.5M
SAR-18	**14.9M**
ResNet	11.1M
LSTM encoder	1.4M
LSTM decoder	2.4M
FPHTR-31	**52.6M**
ResNet	46.1M
Transformer decoder	6.5M
SAR-31	**57.4M**
ResNet	45.8M
LSTM encoder	4.5M
LSTM decoder	6.9M

Table B.2: Additional number of learnable parameters per writer code variant.

	FPHTR	SAR
Learned code	1.4M	1.4M
Hinge code	2.4M	2.4M
Style code	1.6M	1.6M
Zero code	2.4M	1.6M

Table B.3: Additional number of learnable parameters per meta-learning variant.

	FPHTR-18	SAR-18	FPHTR-31
MAML	0	0	0
MAML + llr	173	87	209
MetaHTR	3.7M	14.7M	3.7M

Appendix C

Batch normalization in MAML

In this section, we discuss the role of batch normalization in the MAML-based models. For MAML and MetaHTR models, using batch normalization [47] (see Section 2.1.3) in the right way was generally crucial to obtain good performance, and would often determine whether a model could work at all. However, this is something that is not mentioned in the MetaHTR paper [13]. Although the current discussion is not directly relevant to the main narrative of the thesis, we include it here for the sake of completeness.

It has been reported in [4] that the implementation from the original MAML paper [28] makes use of batch statistics to normalize the activations in batch normalization layers, and that [4] discovered through experimentation that standard batch normalization using stored statistics does not work well. One can imagine why batch normalization could be problematic when training on radically different tasks. During normal neural network training, batches of data are randomly sampled, which, if large enough, have statistics that are close to the dataset statistics. This implies that the batch statistics will remain relatively stable during training. However, introducing task-specific batches of data can potentially lead to large shifts in activation statistics during training, since batches of data are now *task-specific*, i.e., one batch contains a single task. Especially as the number of inner loop optimization steps is increased, the deviation from the global mean and variance will tend to grow.

Nevertheless, based on our experiments, we found the opposite to hold true for our HTR models. Using batch statistics degraded performance, and depending on the base model, it would lead to consistently inferior performance. Numerous setups have been tried out in this regard, based on what was proposed in [4], e.g., fixing the γ parameter in the batch normalization layers, or only using batch statistics in the inner loop, but none of these setups yielded good results.

The explanation for this discrepancy may lie in the nature of the tasks used in MAML. In traditional MAML setups such as few-shot image classification, introducing a new task implies introducing one or several new image classes. The image distribution may therefore change radically, along with the distribution of the intermediate layer activations, and the previously stored statistics may not work well anymore. By contrast, in the HTR setting, different handwriting styles may be similar enough such that shared statistics can still be used for normalization.

Interestingly, the effect of batch normalization was much stronger for the LSTM-based model (SAR). For the SAR base model, using batch statistics for normalization would lead to a significant drop in performance to about 40% WER. For the FPHTR model, performance was generally worse than with stored statistics, but only by a margin of a few percentage points.

Note that the only place where batch normalization takes place is in the ResNet backbone (which FPHTR and SAR both use). Therefore, the LSTM model seems to be more sensitive to the changes in normalization statistics expressed in the ResNet output. Recall that the structure of SAR is such that the output of the ResNet encoder is passed through an initial encoder LSTM processing image strips, followed by a decoder LSTM for language decoding using 2D attention (Fig. 2.3). One difference between the FPHTR and SAR models is that FPHTR uses layer normalization [7] following the multi-head attention modules. By contrast, SAR uses no normalization layers after the ResNet encoder. Possibly, this could result in a larger sensitivity to changes in the ResNet output distribution, since the additional variability does not get normalized along the way.