



university of
 groningen

faculty of science
 and engineering

Context is All You Need:
**Comparing Transformer-based to
RNN-based Models in a Handwritten Text
Recognition Task**

Maximilian Velich



**university of
 groningen**

**faculty of science
 and engineering**

University of Groningen

**Context is All You Need: Comparing Transformer-based to RNN-based models in a
 Handwritten Text Recognition task**

Master's Thesis

To fulfill the requirements for the degree of
 Master of Science in Artificial Intelligence
 at University of Groningen under the supervision of
 Prof. dr. L.R.B. Schomaker (Artificial Intelligence, University of Groningen)
 MSc M. Ameryan (PhD student, Artificial Intelligence, University of Groningen)

Maximilian Velich (s2762927)

July 12, 2022

Contents

	Page
Abstract	5
Abbreviations	6
1 Introduction	7
1.1 Research Questions and Operationalization	9
1.2 Outline of the Thesis	10
2 Theoretical Background	11
2.1 Neural Networks and Deep Learning	11
2.1.1 A brief Introduction to Deep Learning	11
2.1.2 From Recurrent Neural Networks to BiLSTMs	13
2.1.3 Convolutional Neural Networks	16
2.2 Natural Language Processing	18
2.2.1 The significant Problem of Context and Heuristics	18
2.2.2 Attention Mechanism	19
2.2.3 The Self-Attention mechanism	19
2.2.4 The Transformer	21
2.3 Handwritten Text Recognition	24
2.3.1 The significant Problem of Context in Handwritten Text Recognition	24
2.3.2 The History of Handwritten Text Recognition in a nutshell	25
2.3.3 Connectionist Temporal Classification framework for Loss computation	26
3 Methods	29
3.1 Dataset	29
3.1.1 Data Split	30
3.2 Preprocessing of Images	30
3.2.1 Augmentation	30
3.2.2 Image Preparation	33
3.3 Preprocessing of Labels	34
3.4 Gated-Convolutional Visual Frontend	36
3.4.1 Gated-Convolutional Layer	38
3.5 LSTM-based Model	38
3.6 Transformer-based Model	39
3.6.1 Encoder Layer	40
3.6.2 Decoder Layer	41
3.6.3 Multi-head Attention Module	41
3.6.4 Position-wise Feedforward	42
3.7 Implementation Details	43
3.7.1 Training versus Inference	43
3.7.2 Loss Calculation and Post-processing	43
3.8 Evaluation	45
3.8.1 Error Rates	45
3.8.2 Data aggregation and Plotting	45

4	Experiment I: RNN vs. Transformer	46
4.1	Setup	46
4.2	Results	46
4.3	Discussion	48
5	Experiment II: CTC loss at the encoder	51
5.1	Setup	51
5.2	Results	51
5.3	Discussion	53
6	Experiment III: Weighted composite loss	54
6.1	Setup	54
6.2	Results	54
6.3	Discussion	54
7	Experiment IV: Curriculum Weighing	57
7.1	Setup	57
7.2	Results	57
7.3	Discussion	59
8	General Discussion	60
8.1	Summary & Conclusions	60
8.2	General Discussion	62
	Bibliography	63

Abstract

In this thesis, two Deep Learning architectures are compared in a Handwritten Text Recognition (HTR) task. The Transformer-based model constitutes a modern method to sequence-to-sequence learning tasks, especially potent in Natural Language Processing tasks. The RNN-based model represents the state-of-the-art; many impressive systems were built with RNNs at their core. The complete absence of recurrence of the Transformer represents a paradigm shift from recurrence to attention mechanisms at the core of sequence-to-sequence models. Thus, it is imperative to examine what role Transformers could play in HTR. The focus is on maximizing the available lexical contextual information extraction. For that purpose, two comparable systems were built — parameter counts were kept low, no language models were used, and both systems feature the same visual CNN-based frontend. A series of experiments was conducted, the results are based on evaluation and test curves, utilizing Character Error Rates (CER) and Word Error Rates (WER). The results were evaluated on the IAM data set. The Transformer-based model showed lower error rates (CER: 12.8%, WER: 31.6%) than the RNN-based model (CER: 18.6%, WER: 49.7%). The difference between observed WER and expected WER was larger in the case of the Transformer (27.1% lower) than the RNN-based model (13.9% lower). Thus, the results include direct evidence that the Transformer-based model exploits the available context of the given line-strip images more than the RNN-based model. Furthermore, it is shown that the Connectionist Temporal Classification framework can be utilized in a composite loss function to decrease error rates quicker. Overall error rates can be decreased by introducing weighing to the loss function following a curriculum function that shifts training emphasis from the encoder to the decoder. This also shows smooth training curves. Mean CER decreased to 9.1% and mean WER decreased to 25.9%. In summary, within the perimeters of the experiments, the Transformer-based model achieves lower error rates, and shows better ability to make use of the context available. Thus, contending with the Transformer to further the field of HTR, is imperative.

Abbreviations

HTR Handwritten Text Recognition

AI Artificial Intelligence

NLP Natural Language Processing

CTC Connectionist Temporal Classification

RNN Recurrent Neural Network

CNN Convolutional Neural Network

CE Cross Entropy

LS Label Smoothing

CER Character Error Rate

WER Word Error Rate

LSTM Long Short-Term Memory

BiLSTM Bidirectional Long Short-Term Memory

MDLSTM Multi-dimensional Long Short-Term Memory

SDPA Scaled Dot-Product Attention

MHA Multi-Head Attention

1 Introduction

Handwritten Text Recognition (HTR) is the field of translating images of handwritten text, see Figure 1, into digital text. Handwritten documents need to be scanned and ingested into a digital system. Not only image quality matters, but also the scope of the image, whether the image contains a single character, a line, or an entire page. Often, images also need to be preprocessed – documents might need to be cleaned, corrected, segmented, et cetera. Then, the data needs to be actually translated into digital text. All of these steps, especially the latter, pose significant challenges. Inherent human variability coupled with ambiguities and vagueness of natural language leads to an enormous search-space.

Manually constructing a rule-based digital system that can handle all possible combinations of handwritten strokes, language rules, and the disambiguation of meaning, is a tedious and eventually futile effort. For that reason, HTR tasks are tackled as an Artificial Intelligence (AI) problem — systems are trained on large data sets to automatically extract information without human involvement¹. In recent years, HTR has been framed as a Deep Learning task featuring advancements from Natural Language Processing (NLP) and Computer Vision [1, 2, 3]. Complex systems with various extraction and classification layers, and many hundreds of thousands of adjustable parameters are trained on large numbers of labelled handwritten documents. From the data, Deep Learning models appear to *learn* patterns and knowledge by extracting generalizations². Thus, a trained system can be employed to process previously unseen data. However, modern systems are not perfect. Generally, these are trained for a niche domain, e.g. a specific language, symbol dictionary, or a set of historic documents. Furthermore, although models perform better every year, results still leave much to be desired. AI is advancing rapidly. Consequently, new technologies and models are developed at a fast pace. The expectation is that those might be applicable to HTR systems as well. This thesis is about one such technology, the *Transformer*, and its potential contribution to HTR.

AI and HTR relate interestingly to one another. For one, HTR utilizes AI methods. However, HTR reveals an open wound of the field of AI — the question of how to exploit available contextual information maximally. Humans heavily dependent on context when making inferences about the natural world. Understanding context appears imperative for any intelligent agent, whether digital or not. Especially considering natural languages, underutilization of context poses problems at many levels. A sloppily written letter closely resembling another will not result in a misclassification of the entire word — humans can handle such mistakes with ease using the context of the word, the sentence, and more. Words may encompass multiple meanings, but humans effortlessly disambiguate them. Also, sentences appear in paragraphs and sections, written by writers with unique intentions, maybe during times of historic or cultural significance. Thus, even more layers of context might be relevant. While reading handwritten text, humans bind together contextual information and funnel the data through this intricate hierarchy of context. In HTR, this implicit context hierarchy becomes prevalent at every step of the pipeline — from processing pixels all the way up to understanding the meaning of sentences. For true intelligence, the field demands simultaneously addressing the problem at all levels, utilizing the available data maximally.

Within AI, and by extension also HTR, one might observe a tendency towards optimizing only a few performance metrics, and there are obvious, good reasons for that. However, it is imperative to be aware that addressing the big questions in AI will require focusing on what truly constitutes an intelligent system — dispensing of quick heuristics, potentially sacrificing performance for the sake of revealing steps required for intelligence³. In this thesis, an attempt is made to strike a balance between performance metrics and finding ways to tackle the problem of the context hierarchy. Specifically, the focus is on lexical context extraction, translating line-strip images into digital text. Thus, two Deep Learning architectures are compared — one based on well-established Recurrent Neural Networks, and one based on the more novel Transformer architecture. The goal is to determine which is more suitable for an HTR task.

¹ People still have to build the system, however, the details of the information extraction from the data is left to the AI model.

² This is a simplified and rhetorical statement omitting what *learning* really entails here. See Section 2.1 for more information.

³ This does not suggest that humans do not use heuristics — we do — it merely suggests that some heuristics implemented by system engineers depend on external knowledge to an extent that renders any argument of true intelligence untenable.

The image shows a line of handwritten text in black ink on a white background. The text reads "He has also had talks with". Each word is enclosed in a light gray rectangular bounding box. The handwriting is somewhat cursive and slightly slanted.

Figure 1: An example of an image of handwritten text. The correct translation would be: He has also had talks with. The example is taken from Marti and Bunke [4].

HTR was initially tackled by Hidden Markov Models, simple Neural Networks, and combinations of the two. HTR was framed as a sequence matching problem. Quickly, that turned out to be insufficient given the inappropriateness of the Markovian assumption for the task — each observation depends on only the current state [5]. Moreover, Neural Networks were not yet powerful and specialized enough at the time. With the advent of Recurrent Neural Networks (RNN), many advancements were made. RNNs could extract patterns across character sequences, which enabled the encoding of contextual information to a degree. Similarly, Convolutional Neural Networks (CNN) were used as a first guard to extract high quality features from the messy handwritten data [6]. Thus, systems made of CNNs followed by RNNs achieved impressive results for that time.

Using RNNs, HTR tasks were reframed as sequence-to-sequence problems — a sequence of images, or segments thereof, needs to be mapped to a sequence of characters, for instance see [7, 8]. However, training such a network proved to be difficult due to an alignment problem: given an image of a sentence, and its known translation, how does one automatically align the correct characters over the input image? The fact that the letter `A` appears in the image is not enough, the location is also relevant. Otherwise, the system has little chance of learning the token properly. A solution to this problem was provided by the Connectionist Temporal Classification (CTC) framework initially proposed by Graves et al. [9]. The CTC framework provides a way to find all possible alignments of the label across the input image, and then uses the system’s output probabilities to formulate a probability distribution across the alignments. This is then used to calculate the mismatch for each example and hence provides a loss function to train the network. CTC has shown to be vital in achieving good results in HTR tasks, see [10, 11] for examples. The modern HTR landscape is mostly dominated by systems made of a CNN frontend and an RNN-based model trained using a CTC stage, for instance see [12].

The Transformer architecture was developed by Vaswani et al. [13]. Initially, the Transformer was created to solve NLP tasks such as Machine Translation, but it also spread to other fields. The Transformer achieved remarkable results in tasks related to language comprehension and language generation, such as BERT [14] and the GPT models [15, 16].

Previous models using RNNs showed improved results in combination with attention mechanism [2]. Such mechanisms pose an alternative to modeling sequence data by the recurrent property of RNNs. Transformers mark the end of a paradigm shift *from recurrence to attention* since they rely solely on attention mechanisms in conjunction with simple feed-forward neural network layers. Recurrence essentially operates on a proximity-based function given its sequential nature. Whereas attention, in particular Self-Attention⁴, models relations between tokens in a provided sequence disregarding the order they appear in⁵. Hence, those two mechanisms will be under scrutiny when comparing RNN-based systems with Transformer-based ones. As part of this thesis, these mechanisms will be compared in relation to their respective capacity to extract contextual information.

⁴ Self-Attention is the specific type of attention mechanism utilized in Transformers [13].

⁵ How these mechanics work will be subject to deeper examination in Section 2.1.2 and Section 2.2.3.

1.1 Research Questions and Operationalization

Attention mechanisms allow for specific relational modelling between words or characters. Conceptually, this appears to result in more dynamic modelling as opposed to the more rigid proximity-based modelling of the RNNs. Hence, attention mechanisms *seem* to be useful for capturing context. However, the proximity-based function of the RNNs were shown to be effective already. While Transformers are in principle capable of simply emulating the proximity-based behavior, they will inherently prefer a more complex web of relations between tokens. The question is whether this dynamic modelling property actually results in better performance in an HTR task. And, furthermore, whether attention mechanisms also result in better capturing of the provided context. Even if the Transformer is conceptually better suited, this might not necessarily translate into better results. Consequently, testing is required to shed light on this topic. This *progressive case* is encapsulated by Research Question 1, see **RQ1** below.

The *conservative case*, captured by **RQ2**, is a defensive case: Although Transformers and their sole reliance on attention mechanisms pose a potent method, immediately declaring them the new default seems shallow, even if they exhibit better performance. Instead, it seems more appropriate to focus on salvaging useful methods from what was once the default. This holds especially true given that the success story of RNNs was not only the one of RNNs, but also the success story of CNN frontends and the CTC framework. Therefore, the second part of this thesis is dedicated to integrating a CTC stage into the Transformer-based system to test its effect in an HTR task. Again, the evaluation will touch on two main points: first, whether there are any performance increases from the addition of the CTC framework, and if so, whether those show better utilization of the available contextual information.

After questions **RQ1** and **RQ2** are answered, the focus will shift to experimenting with the established results and systems to gain insight into how to further improve the model. The focus is on how to improve the quality of the classification, for instance, by reducing variability. This case is captured in **RQ3**.

RQ1 In comparison to RNN-based models, are Transformer-based models better equipped to both recognize handwritten text and to exploit the available contextual information maximally? This question is tackled in Experiment I (Section 4).

RQ2 Can the CTC framework be leveraged in conjunction with a Transformer-based model to enhance performance and contextual information extraction? See Experiment II for answers (Section 5).

RQ3 What can be done to refine the results further? And, how can the quality of the classification be enhanced? This question is tackled in Experiment III and IV (Section 6 and Section 7).

To answer these research questions, two HTR models will be built; one is RNN-based and one is Transformer-based. A focus point of the experiments is comparability, hence the models in question will be kept as similar as possible. They both share the same CNN-based visual-frontend to extract high quality features from the provided images. The dataset used to train them and the data preparation phase will also be kept identical. The size as given by trainable parameter count is approximately matched. Experiments will be conducted that share the same evaluation methodology across the architecture types.

By answering these questions, we contribute to the field of HTR and outline promising future steps. Specifically, a contribution regarding the difference of contextual information extraction between Recurrent Neural Networks and Transformers. This should further educate the field whether attention mechanisms are also a promising approach to tackle HTR tasks. This thesis also attempts to marry older established frameworks such as CTC with newer modern solutions such as Transformers.

1.2 Outline of the Thesis

In Section 2, the theoretical background is explained, consisting of three parts:

Neural Networks and Deep Learning in Section 2.1

This section includes a brief introduction to the field in Section 2.1.1 which is followed by an explanation of Recurrent Neural Networks and their current common architectures in 2.1.2. Finally, the topic is concluded with a section on Convolutional Neural Networks in Section 2.1.3.

Natural Language Processing in Section 2.2

This section first elaborates on the aforementioned context problems in natural language in 2.2.1. Then, the focus shifts to attention mechanisms in Section 2.2.2 and an additional section on Self-Attention in Section 2.2.3. Lastly, the Transformer architecture is dissected and explained in Section 2.2.4.

Handwritten Text Recognition in Section 2.3

First, the context discussion is revisited through the lens of HTR in Section 2.3.1. What follows is a brief history of the field in Section 2.3.2. The topic is concluded with an explanation of how the CTC framework is used to derive a loss function for training a sequence-to-sequence model in Section 2.3.3.

In Section 3 the methods for the experiments are explained. The dataset and its preprocessing steps are described from Section 3.1 to Section 3.3. Then the details of the proposed systems follow from Section 3.4 to Section 3.7. This general section is concluded with a description of the evaluation in Section 3.8.

The experiment sections follow: Experiment I in Section 4, Experiment II in Section 5, Experiment III in Section 6 and Experiment IV in Section 7. Each of these is structured into a Setup section, a Results section and a Discussion section. A final discussion and conclusion is provided in Section 8.

2 Theoretical Background

To answer the research questions, Deep Learning models will be constructed and trained to tackle a Handwritten Text Recognition (HTR) task. In this section, Deep Learning is explained followed by a subsection on Natural Language Processing, and lastly, the focus shifts to the field of HTR. Note that all three topics covered here are vast in their scopes — this section only deals with explanations relevant to the methodology of the models and the task at hand.

2.1 Neural Networks and Deep Learning

This section provides a brief overview of Neural Networks and information regarding Deep Learning. This includes basic concepts such as how supervised learning works. Special types of neural networks can be connected to form specific types of networks. The ones explained here are Recurrent Neural Networks (see Section 2.1.2) and Convolutional Neural Networks (see 2.1.3).

2.1.1 A brief Introduction to Deep Learning

In its basic form Machine Learning constitutes a method for a program to *learn* from data. Contrary to that stands a traditional computer program that strictly follows rules provided by the programmer. In a Machine Learning system, these rules are learned by the program itself. Usually, the programmer sets up a model which is adjusted by an optimization algorithm that uses example data. This process is called *training*. During training the algorithm uses the data to *teach* the model to *learn* patterns in the data. Over time, a proper model can then extract overarching patterns from the example data which allows for generalization. Once the model is trained it can be used in conjunction with previously unseen data to compute a classification. This is called *inference* [6].

Many Machine Learning methods exist for various tasks. A popular one, and the one utilized in this thesis, is a Neural Network. The building blocks of such are Multilayer Perceptrons which pose a simple mathematical model that maps an input to an output. Many perceptrons can be used to form a network of nodes and connections between them. The nodes receive input signals via their input connections. Those connections have associated values, called *weights*. The nodes multiply the incoming signals with their connections' associated weights and compute the sum of the result. They then compute the output via an *activate function* over the sum. There are many such functions. There are mainly two purposes to those. First, they introduce non-linearity into the model. For instance, the *sigmoid* activation function transforms the incoming linear signal into a non-linear output following a sigmoid curve. Second, the function squeezes the output into a certain range. For instance, the *hyperbolic tangent* function's output is a number between -1 and 1 . The output of such a node can then be sent as input to another node [6].

The aforementioned *weights* serve as the trainable parameters. They are optimized by a training algorithm. This optimization process can take many forms. In a supervised learning process, the model is exposed to example data and their correct labels. For instance, a Machine Learning classifier could be used to learn how to identify types of dogs in pictures. During optimization, the model would be exposed to many images of various dogs labelled with the correct type. Whenever the model predicts an incorrect type the training algorithm adjusts the weights of the model such that the output results in a classification that is closer to the true one. This is repeated for many examples, sometimes millions. After many training epochs, the model's weights become tuned enough to serve as an accurate classifier [6].

An untrained model will err in its classification. Some classifications will be farther from the target than others. Thus, some mismatches need more correction than others. A heuristic for this magnitude of required correction is given by loss functions. They produce a numerical representation, called loss, of the mismatch between classification and correct *label*. The loss is then used in a process called Back-propagation which adjusts the weights of the network accordingly. The optimization task of the network during training is therefore to minimize the loss of the network.

An example of a loss function is Cross Entropy loss (CE). Let x be the probability distribution of the output, i.e. the classification, and let y be the label, i.e. the ground truth. The CE first calculates the logarithm of each entry in x , then multiplies that by the corresponding entry of y . The sum of all products are taken. Finally, the sum is multiplied by -1 ; loss functions are typically minimized, the Maximum Likelihood Estimation which CE loss is based on, is negated. Thus, this function is also called Negative Log Likelihood. The CE formula can be seen in Equation 1. The result is a single numerical value. The lower this value, the smaller loss, i.e. the higher the accuracy of x , the classification.

$$\text{CE}(y, x) = L(y, x) = - \sum_i y_i \cdot \log(x_i) \quad (1)$$

The quality of the training outcome depends on the input data. The better the task that should be learned is represented by the data, the easier it will be for the training algorithm to set the weights correctly. Whilst data can be manually prepared to be a good representation, it is often better to let the model also learn that representation by itself [6]. This means one can construct a model containing hierarchies of sub-models — each sub-model taking the previous representation and formulate a new simplified representation. Such a chained Representation Learning is particularly often used in an end-to-end training context which means the model is trained as a whole, from one end to the other. This approach often results in large, *deep* chains of many intermediate representations, hence the name Deep Learning.

When training a Deep Learning system the hope is that it performs well on data that was not in the training set. During training, a number of examples are selected to let the model learn certain patterns and structures. *Learning* really means generalization. If the model generalizes well, previously unseen examples will be classified correctly. During training, an optimization problem is solved using the loss function. This however only affects the training performance, i.e. the error during training will go down over time. How well the model generalizes is not necessarily demonstrated by the train error. Therefore, a second error, the test set error should be minimized as well. The test set error is measured on previously unseen set of test data. A model that learns well shows a decrease in both train and test set error.

A model's *capacity* describes how many functions it can model. If a model has low capacity, it might not be enough to enable generalization — the model is under-fitted. Generally, the solution to that would be to increase capacity by extending the model. In contrast, if a model keeps showing lower train errors during training, but the test set errors increase, the model exhibits over-fitting. Here, the model's capacity is high enough so that it can store the given examples perfectly, which however, inhibits generalization. Roughly speaking, this means that a model's capacity needs to be small enough so that during training the model is forced to find overarching patterns rather than specific details pertaining to individual examples. This enables true learning — generalizing examples to patterns and formulating rules. Strategies that lead to lowering test set errors while potentially neglecting train errors are called Regularization.

Two examples of Regularization are Dropout and Dataset Augmentation, both are used in later experiments. Dropout randomly disables connections within a network, but only during training. This forces the model to train sub-versions of its full network. These subnetworks change with each epoch. This makes the network more effective since the model is forced to adapt a certain robustness for the given examples. Dropout also reduces the opportunity to over-fit on specific examples as connections necessary to memorize details might simply be missing due to dropout.

Similarly, Dataset Augmentation seeks to reduce the test set error by augmenting the training dataset with examples that are based on the given examples but modified in certain ways. In an image dataset for instance, images can be rotated, translated or even distorted to the point where the subject of the image does not change — an upside-down cat is still a cat⁶. This increase in examples then inhibits over-fitting on specific examples, since there might exist many versions of the same image each modified in some way, which makes learning details pertaining to only one example much more difficult.

⁶ Although true in this case, this assumption does not necessarily hold in the context of an HTR task. For instance, an upside-down p is a b. This detail on augmentation is later revisited in Section 3.2.1.

2.1.2 From Recurrent Neural Networks to BiLSTMs

For many tasks, a regular multi-layer neural network lacks specialization. One such specialization is processing sequence data. Many tasks involve sequential data. For instance, audio data like recorded speech and music, text which is a sequence of letters and words, stock market price charts or any time series data. Recurrent Neural Networks (RNNs) are specialized for such data. They are based on the idea of sharing a network state with another part of the network. For a time series, that could mean the output of an artificial neuron at time $t = 1$ can be an input to another one at $t = 2$, although that concept generalizes to all sorts of sequence data. Such preservation of states over time enables learning across longer sequences by introducing the concept of the past – effectively, a sequential memory. For the remainder of this section, the focus is on time series data.

At each time step an artificial neuron in an RNN receives input from an example and *contextual input* from the previous time step. Those two inputs are used to compute the output which serves also as input to the next recurrent unit. A schematic of that can be seen in Figure 2. This way, network states are propagated through the network. Via this mechanic patterns across time can be incorporated into the learning process. However, this comes with an issue. The recurrent data from step $t - 1$ combines all previous states into a single one to be used at time t . That means that all previous states are represented with a discount proportional to the time that has past. The state from $t - 1$ is more emphasized than from $t - 2$; the state from $t - 2$ is more prominent than $t - 3$; et cetera. This so-called Short-Term Memory problem results in more recent recurrent steps being better remembered than steps farther in the past.

The Short-Term Memory problem is also related to the Vanishing or Exploding Gradient Problem. When training such a network one uses a modified version of back-propagation called Back-Propagation-Through-Time. At each time step the loss is computed, and the gradients are used to update the weights. However, the farther back in time the gradients go, the higher the chance that they become vanishingly small, or very large. That is because of the many multiplications that happen with the same values as part of the gradient propagation process — small values continue to shrink, large values continue to grow. Large values will skew the gradients dramatically, and small values will effectively halt learning all together. Using activation functions that force output values into specific ranges alleviates this issue to a degree — especially the exploding gradients. However, the vanishing gradient problem still persists.

A remedy is provided by gated RNNs, such as Gated Recurrent Units [17], or Long Short-Term Memory (LSTM) units [18]. Both units use the concept of a gate which enables controlling the flow of tensors⁷. Gates come with their own set of trainable weights, hence they are an integral part of a Deep Learning system. In the context of HTR and the task at hand the rest will focus on LSTMs specifically.

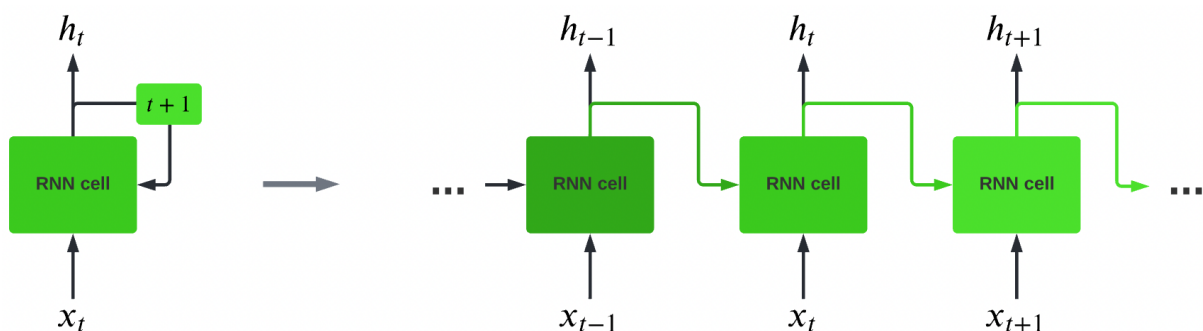


Figure 2: A schematic example of how an RNN cell is unfolded over time. Modified from [6, p. 376].

⁷ Within the context of a Deep Learning system, a *Tensor* is a multidimensional array or vector. The term is mostly used in a practical setting, i.e. during programming, when denoting data structures.

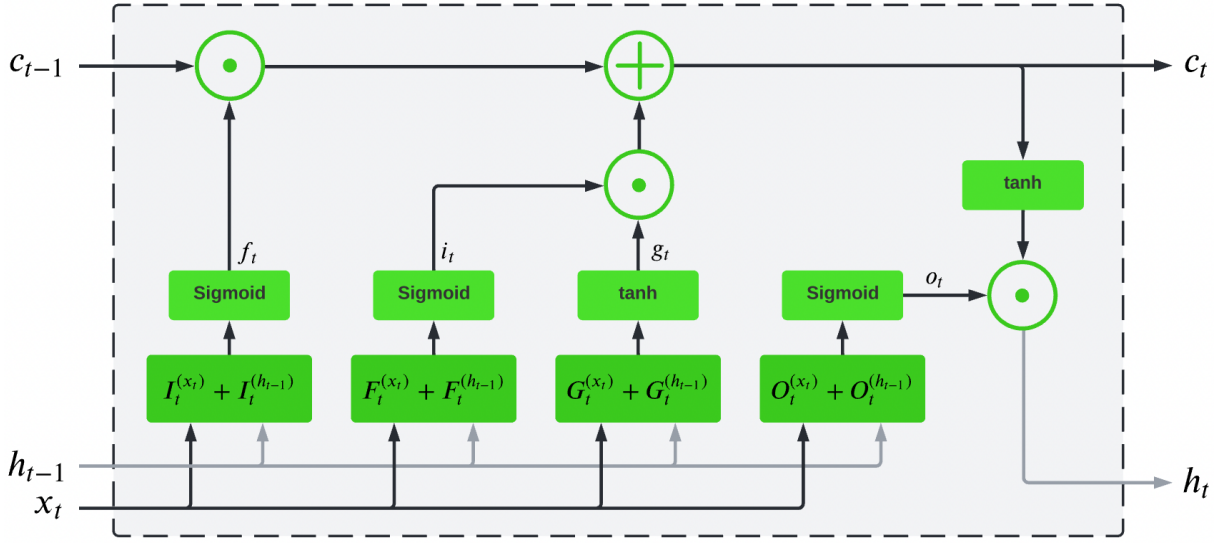


Figure 3: An overview of how the gates within an LSTM cell are connected and calculated. The input is denoted as x_t and the output as h_t . The context signal c_t is provided from one cell to the next similarly to the hidden state h_t . There are three gates, the forget gate between f_t and c_{t-1} , the candidate gate between i_t and g_t , and finally the output gate between o_t and the candidate gate. Modified from [6, p. 409].

LSTMs are a specialized form of RNNs. They were conceived with the focus on solving the short-term memory issue whilst also addressing the Vanishing Gradient Problem. An overview of an LSTM cell can be seen in Figure 3. The cell has an input x_t and an output state h_t , which also serves as input to the next cell as h_{t+1} . This constitutes the recurrent property as seen in the general RNN formula in Figure 2. In addition to that, the LSTM cell also features a cell state input (c_{t-1}) and output (c_t). From the figure, one can see that this cell state runs through the cells for each time step. Each cell is then comprised of three gates that can modify this cell state.

The first gate, in line from left to right, is called the *forget gate*, regulated by $f(t)$. Both input tensors, x_t and h_{t-1} are passed through two linear layers — $F_t^{(x_t)}$ and $F_t^{(h_{t-1})}$ — their outputs are added, and finally passed through a sigmoid activation function. This results in a value between 0 and 1, which is then used to regulate the cell state using the Hadamard product, \odot . The closer the value is to 0, the more should be *forgotten*; the closer the value is to 1, the more should be *remembered*. See Equations 2 for the exact math. Notice how W_{if} and W_{hf} constitute two trainable set of weights together with their associated biases b_{if} and b_{hf} , i.e. subnetwork layers.

$$F_t^{(x_t)} = W_{if} x_t + b_{if} \quad F_t^{(h_{t-1})} = W_{hf} h_{t-1} + b_{hf} \quad f_t = \sigma \left(F_t^{(x_t)} + F_t^{(h_{t-1})} \right) \quad (2)$$

The next gates control what new information is added to the cell state. It comprises two signals, i_t and g_t . Notice how both mirror f_t . However, g_t uses a *tanh* activation function which results in an output range of -1 to 1 . The input signal i_t is used to scale the candidate signal's output g_t using the Hadamard product. This forms the second gate, the *candidate gate*. After that, it is merged with cell state c_t via a simple addition operation. This results in the updated values for c_t , see Equation 5. Notice how Equations 3 and Equations 4 are describing the input signal and the candidate signal, respectively. Also, the weight vectors $W_{ii}, W_{hi}, W_{ig}, W_{hg}$ constitute trainable parameters of the cell, together with the associated biases.

$$I_t^{(x_t)} = W_{ii} x_t + b_{ii} \quad I_t^{(h_{t-1})} = W_{hi} h_{t-1} + b_{hi} \quad i_t = \sigma \left(I_t^{(x_t)} + I_t^{(h_{t-1})} \right) \quad (3)$$

$$G_t^{(x_t)} = W_{ig} x_t + b_{ig} \quad G_t^{(h_{t-1})} = W_{hg} h_{t-1} + b_{hg} \quad g_t = \tanh \left(G_t^{(x_t)} + G_t^{(h_{t-1})} \right) \quad (4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (5)$$

Finally, the *output gate*, regulated by o_t , is used to compute the cell output h_t . Once again o_t is formed similarly to the previous gates. The math is outlined in Equations 6. The result is then used in a Hadamard operation together with the updated cell state c_t from the previous gates. See the mathematical details in Equation 7. The product constitutes h_t and therefore the output of a cell. Note how h_t is therefore a combination of all inputs and gates.

$$O_t^{(x_t)} = W_{io} x_t + b_{io} \quad O_t^{(h_{t-1})} = W_{ho} h_{t-1} + b_{ho} \quad o_t = \sigma \left(O_t^{(x_t)} + O_t^{(h_{t-1})} \right) \quad (6)$$

$$h_t = o_t \odot \tanh(c_t) \quad (7)$$

This concludes the inner workings of the LSTM cell. Otherwise, the cell follows largely the same methodology as normal RNN cells. However, with a remedy for the vanishing and exploding gradient problem, LSTM-based systems found wide applicability and proved useful for many tasks in many areas.

LSTMs can also be used in a bidirectional configuration (BiLSTM). A second layer of LSTM cells is arranged which is trained on the input data in reverse. This is depicted in Figure 4. This results in effectively two networks, trained on the same text, but from both directions. During inference, both networks are consulted, and both networks produce a classification. For instance, the classification at time t with input x_t are h_t^B and h_t^F . The final classification can then be decided on using methods such as averaging [19]. In modern BiLSTM-based systems however, this may be handled by another network layer that compresses the double-sized output back to the original single-sized one, i.e. mapping both h_t^B and h_t^F to h_t for all time steps t . The specifics on how to *vote* on the best classification is therefore also part of the learning process.

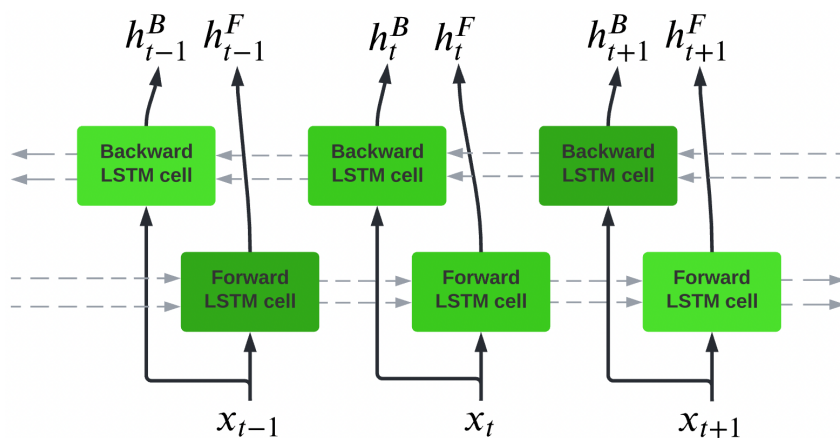


Figure 4: A schematic overview of an unfolded bidirectional LSTM network. Note how there are two directions, the forward line from left to right, and the backwards line from right to left. Each cell has a unique output that later needs to be combined. Modified from [6, p. 394].

2.1.3 Convolutional Neural Networks

Another special type of neural network is a Convolution Neural Network (CNN). They find applicability for data that can be meaningfully represented as a grid. Amongst others, the prime example of that are images, which certainly constitute the input to a large portion of CNN-based systems. An image generally takes the form of a two-dimensional grid of pixel values — one per channel, usually colors.

Generally, images are made up of hierarchies of depicted concepts. Imagine a photo of a cat. The cat has a body with legs and a head. The head is made up of eyes, a snout and ears. Each of these concepts is further made up of pixels and their arrangements. To identify an eye of that cat a specific pixel cluster is highly relevant. The pixels within that cluster stand in a specific formation that forms the eye. However, pixels pertaining to the legs of the cat are irrelevant to the eye cluster. Roughly speaking, pixel clusters show high dependency between the pixel within the cluster, but there are rarely any dependencies between clusters. Thus, images have this property of local relevance.

CNNs utilize the *convolution* operation. Convolution is a mathematical function that takes two other functions and returns a function which serves as a means to explain how one of the input functions is altered by the other one. The convolution $(K * I)(i, j)$ calculates the convolution between the kernel K and the input image I at a position in I given by coordinates on the two-dimensional pixel grid. It is defined in Equation 8 [6]. Note, the convolution operation is denoted by $*$.

$$(K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n) \quad (8)$$

Constructing a neural network to find concepts in a picture comes with some problems. In a fully connected layer, every input is connected to every output. As just explained however, especially in images, not every output needs to know about the entire image. CNNs solve this via their sparse connectivity. In a CNN layer, a matrix of trainable values, called a kernel, slides over the input image. At each step, the kernel is convolved with the pixel values within the cutout of the input. The output values of the convolutions are stored in a feature map [6]. See Figure 5 for a simplified example. The distance the kernel is moved over the input is called the *stride*, e.g. at each step the kernel is moved by 2 pixels. Padding can be used at the margins to avoid feature maps that are slightly smaller than the input [6].

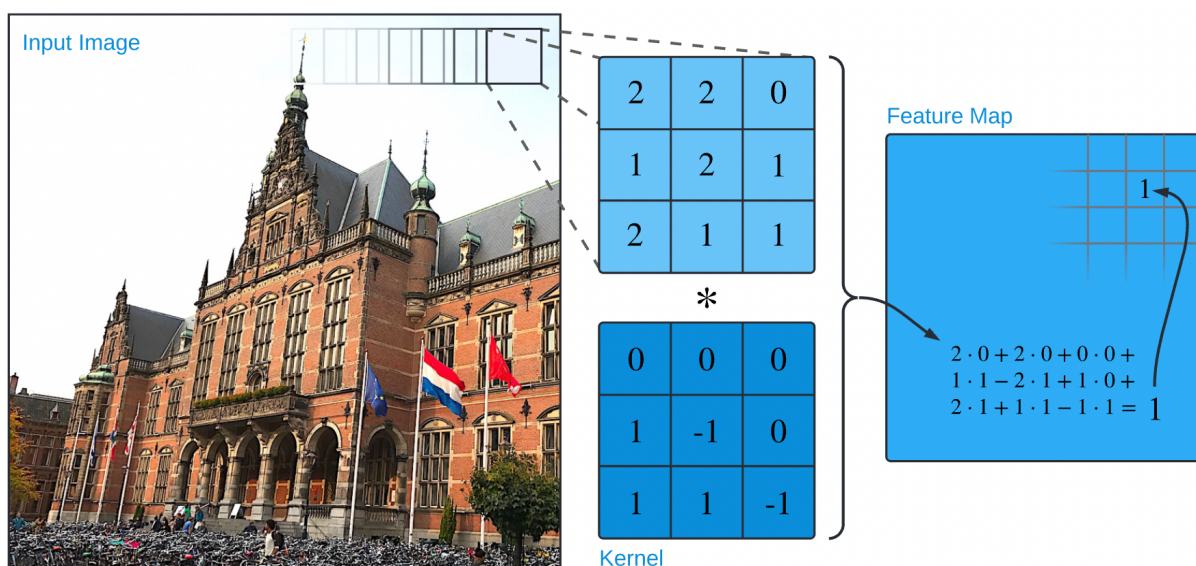


Figure 5: An illustration of how convolution works on an input image. A mask slides over the input image. At every step the pixels of the image that fall into the mask region are convolved with the *Kernel* and the result is added to the *Feature Map*. The convolution operation is denoted by $*$.

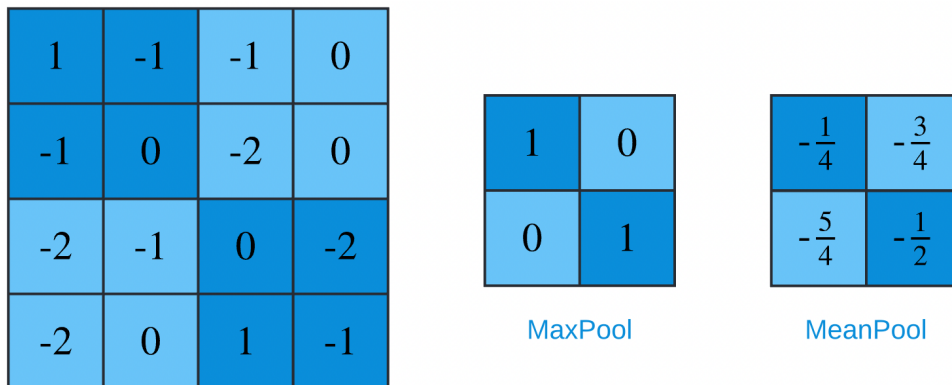


Figure 6: This figure shows two pooling examples. In case of the *MaxPool* function, the maximum of the four values is selected, in case of the *MeanPool* function, the mean of the four numbers is selected.

Each value in the feature map is therefore only dependent on a cluster of pixels. This also means that the values of the kernel are used many times on the same input. This *parameter sharing* feature reduces the CNN's storage requirements drastically when compared to a plain neural network which would need to store unique individual connections for each pixel [6].

Kernels are usually substantially smaller than the input dimensions. During training, this forces the kernel to be focused on specific features only, since there is simply not much freedom in such a limited space. For that reason, usually multiple kernels are used per CNN layer. Each kernel then specializes on a specific feature — recall the cat example, one kernel could be for eyes, one for legs, et cetera.

This learning of local features also enables constructing hierarchies of concepts when employing multiple CNN layers. For instance, a first layer might find features such as edges, curves or corners. A second layer finds circles, rectangles and other shapes. Another layer would find combinations of shapes that constitute more complex concepts, and so on. This extraction process can be powerful, especially when combined with other network types. Using CNN layers, one could first extract features. Then, a simple fully connected layer could be trained on those features rather than the raw input. This is also useful in situations where the input data is of low quality — for instance, in an HTR task, see Section 2.3.

A key mechanism to feature extraction are pooling operations. Assuming multiple layers of CNN, one usually wants to extract higher and more abstract features farther down the pipeline. Between the CNN layers it is useful to pool together parts of the intermediate feature maps to shrink down the representations. There are many ways to achieve that. In Figure 6 two example methods are depicted: *MaxPool* and *MeanPool*. The former takes the maximum of the selected area (a 2x2 grid); the latter takes the mean of the selected area. The result is a pooled feature map shrunken down to a size four times less than that.

2.2 Natural Language Processing

In the context of Natural Language Processing (NLP) a natural language is an existing human language such as English. NLP therefore refers to digitally processing language. The field of NLP involves computationally dealing with language data, mostly large amounts of data. It includes tasks such as Language Translation, Speech Recognition and, of course, HTR. Tasks like those involve disciplines such as Linguistics, Computer Science, and AI. Language data occurs as a sequence of letters, of words, of sentences, et cetera, This means tasks involve sequence-modeling or sequence-to-sequence processing models. Such models come with their own set of difficulties, however. In this section, the focus is on outlining those difficulties, especially the ones regarding context. Then, the focus will shift to solutions to NLP problems leading up to the Transformer architecture.

2.2.1 The significant Problem of Context and Heuristics

Traditional software consists of rules and facts provided by the programmer. For holistic and accurate processing, the programmer thus needs to fully grasp the rules and facts that constitute the language. This issue is at the heart of what makes NLP tasks challenging. Language is inherently messy, complex and complicated. Syntax follows broad rules but comes with many exceptions; words often encompass multiple meanings that give rise to ambiguity and vagueness; pragmatics and prosody add additional layers of complexity that go beyond the written. Many of these issues can be alleviated by understanding the context the language is embedded in. Humans infer the context seemingly effortlessly, while employing traditional computational methods of processing and *understanding* language would result in sheer infinite computational demands. What adds to the issue is that although humans have little difficulty understanding natural language, articulation of the rules and concepts is demonstrably harder for us. It is for that reason that Machine Learning methods became popular – systems learning and inferring the context from large corpora by themselves would seem to solve this issue. Yet, it is still far from trivial.

In the case of language, context refers to the substrate that a piece of language is embedded in. Context is therefore a hierarchy of many layers encapsulating implicit rules. Words are made of characters; they follow morphological rules which may exist due to phonetic reasons. Words also appear in sentences following syntax and their exceptions. Words and sentences convey meaning following semantics. Sentences may be organized in paragraphs and sections. Those may appear in books, speech, et cetera. Language may be embedded in a cultural context, it might carry intention of the writer, and the historic baggage of when it was written. To fully understand the meaning of a piece of language, all these layers of this implicit contextual hierarchy need to be addressed holistically — top-down and bottom-up rules need to be integrated simultaneously. If this does not happen, predicting the true meaning might fail drastically. Language mistakes certainly do not follow a linear scale — sometimes misclassifying many words does not change the meaning much, sometimes getting one letter wrong results in the opposite meaning.

When considering a complex topic such as language in the scope of Artificial Intelligence and the research thereof, it is important to realize when shortcuts are being taken. Shortcuts are often employed as a means to achieve higher performance quickly. While such goals are certainly desirable in specific circumstances, it also distracts from a more general goal in AI — building systems that exhibit intelligence. For example, in Section 2.1.2 a bidirectional configuration of LSTMs was discussed. Generally, performance rises when employing both forward and backward passes on text. However, if the LSTM is the answer to gaining human-like language understanding, then why is the backwards pass even necessary? Humans certainly do not learn language like that, a forward pass is all we need⁸. Another example is employing a pre-existing vocabulary to mold an imprecise classification into words that were not learned by the system itself. This way, classifications can only ever be existing words. This omits the

⁸ To clarify, humans *do* use heuristics; human cognition almost certainly does not just parse words one by one from left to right. This example only points to the weakness of using the backwards pass as a heuristic analogue to human cognition — the addition of parsing words from right to left would be ridiculous as an explanation for human language learning.

possibility for predicting words that were not in the provided vocabulary. Again, humans do not need that. In fact, language and its constant evolution would not occur with such shortcuts in place. Heuristics choices like these are everywhere in NLP systems. Heuristics are either a strength or a weakness.

2.2.2 Attention Mechanism

Around 2015, most of the state-of-the-art in Machine Translation were RNN-based Encoder-Decoder models. Such models take a source sentence and encode it into a context vector which is then fed into a decoder. The decoder then produces the translated sentence word by word using the context vector as input. Such systems were then mainly trained in an end-to-end fashion, i.e. as a whole, which necessitates fixed tensor sizes. However, limiting the context vector to a fixed size results in a bottleneck. If the source sentence gets too long, the encoder is forced to cramp all the information into the same space as it would need for a shorter sentence. Results of such systems were thus as expected: performance was good up until a certain sentence length, and then performance starts to decrease as the bottleneck limits throughput.

A remedy was provided by Bahdanau et al. [20] and their attention mechanisms. The idea of attention is to focus on relevant parts of data while ignoring parts that are irrelevant. Imagine being required to translate a very long sentence into a different language. If you were asked to memorize the entire sentence first, you would struggle to produce an accurate translation similar to the aforementioned. However, if you had the freedom to translate as you wished, you would translate the sentence bit by bit. You would decide on what to pay attention to during translation and ignore the rest. Similarly, attention mechanisms allow the decoder to selectively pay attention to parts of the encoded input. This alleviates the *pressure* on the encoder to compress all needed information into one context vector.

This was achieved by encoding the source sentence into a vector sequence rather than a single vector. The decoder can then choose a subset of this sequence to generate the translation. This choosing is itself part of the training process — attention is learned as part of the complete system. In RNNs, generally speaking, information is propagated from cell to cell. This means that there is an implicit sequential dependency between neighboring cells — the closer two cells are the more correlated their output will be. Depending on the use case this can be an advantage or a disadvantage. In the case of NLP, it appears to be both, depending on the situation — words immediately next to one another are not necessarily dependent on one another, though in many cases they are. In contrast, the attention mechanism allows for more precise modeling of such dependencies in the input sequence.

Attention mechanisms found applicability in tasks involving natural language, but also other fields such as computer vision [21]. Different methods for the attention mechanisms were tested as well as different ways to calculate the attention scores. Attention comes roughly in three categories: Global and Local Attention [22], Soft and Hard Attention [21], and the Self-Attention mechanism [23]. The focus moving forward will be on Self-Attention since it is critical to the Transformer architecture.

2.2.3 The Self-Attention mechanism

In NLP tasks, tokens⁹ are generally mapped into an embedding space. Such embeddings represent the tokens as vectors of numerical values. This enables computational processing which requires numerical values. It also enables capturing meaning of the words such that distance measures between embeddings can be interpreted. For instance, the words *King* and *Queen* are very different in their morphological make-up, but are similar in their meaning. In fact, they only really differ along an *implicit* gender dimension. Embeddings seek to capture such implicit dimensions such that they become useful in a mathematical sense — for instance, the Levenshtein distance between those two words is high; but, a distance measure between their high-dimensional embedding representation should be low. In other words, they should cluster together in their embedding space.

⁹ Tokens refer to the individual bits that make up a sequence. Thus, what a token exactly is might change depending on the context of the sequence in question. In this thesis, tokens refer to either characters or words.

Self-attention operates only within one given sequence, hence *self* [23]. Its broad goal is to encapsulate contextual information that is not guided by proximity. Consider the following sentence:

Joseph can be annoying, but he is a great friend.

Following only proximity, the word *friend* will have the largest relation to its neighboring word *great* which is reasonable because, as an adjective, it modulates *friend*. However, the context behind who *friend* refers to, i.e. *Joseph*, would entail a strong relation to the other end of the sentence. Thus, in this example, proximity as a guide to encapsulate context fails dramatically. Consider another example:

Draft of the thesis

The word *draft* is loaded with meaning: a current of air, obligatory military participation, or a document. However, when actually using the word, this collection of meanings necessarily needs to collapse into a single one¹⁰. And clearly, people immediately understand the specific type of *draft* that is meant here. The context allows us to collapse *draft* into a precise definition modulated by the word *thesis*. Self-attention enables that by using the dot-product as a measure of meaningful similarity.

The dot-product is defined as the sum of the point-wise multiplications of two vectors' entries. The result is a single number that can be interpreted as follows: If two vectors are pointing in the same direction, the dot-product is 1. If they are perpendicular to one another, the dot-product is 0. In other words, the closer the two vectors are in their direction, the closer the value is to 1. Considering words mapped to an embedding space, the dot-product can then be used to find related words. For instance, *King* and *Queen* should roughly be in the same high-dimensional neighborhood in the embedding space. Taking the dot-product of the two should therefore result in a value close to 1. Thus, the dot-product can serve as a means to determine relatedness of two words.

Consider the *Draft* example once more. The goal is to take the original loaded vector and filter it down to a more light version relevant for this specific phrase. Self-attention weighs each vector in the phrase with the word *Draft* and then uses those weights to create a new vector. Let the four words be labelled v_1 to v_4 , respectively. The four weights, one for each word, are calculated using the dot-product, see Equation 9. These weights are also referred to as *attention scores*.

$$w_{11} = v_1 \cdot v_1 \quad w_{12} = v_1 \cdot v_2 \quad w_{13} = v_1 \cdot v_3 \quad w_{14} = v_1 \cdot v_4 \quad (9)$$

The new vector y_1 for v_1 is the sum of the weights times the original word vectors, see in Equation 10.

$$y_1 = w_{11}v_1 + w_{12}v_2 + w_{13}v_3 + w_{14}v_4 \quad (10)$$

Given that w_{12} and w_{13} will be relatively small, y_1 will be mostly a combination of v_1 , the original vector, and v_4 from *thesis*. This does not mute the other meanings of *Draft*, but it emphasizes the relevant one. The outcome, y_1 , is therefore a specific version of v_1 , which is relevant to the given context¹¹. This process is repeated for each word in the given sequence until all original values v are converted into y .

Self-attention comes with interesting properties. First, proximity as a potential context marker is removed — the dot-product instead uses relatedness from the embedding space. Second, the order of the given tokens in the phrase is not respected due to the commutative property of the sum. Third, it is length independent. No matter the length of the input is, the output length will be the same — the math does not change. Recall the aforementioned example of *Joseph*. Regardless of how long the input sentence is, the Self-Attention process should ensure an emphasis on the relation between *Joseph* and *friend*, regardless of proximity or order. Recall the bottleneck issue resulting from a fixed size vector requirement

¹⁰ Proverbial: *If a word can mean anything, it means nothing.*

¹¹ Note that generally, the weights w would really be the normalized attention scores, so they sum to 1. The result is similar to the weighted average.

— it is not an issue when using Self-Attention. And fourth, relevant for Machine Learning, there are no trainable parameters. The weights here refer to the dot-product and not a set of parameters adjusted by an optimization algorithm. This begs the question of how one can use it in a Deep Learning system.

2.2.4 The Transformer

The Transformer is an influential Deep Learning architecture introduced by Vaswani et al. [13]. It established a significant milestone in sequence-to-sequence learning mainly due to the omission of recurrence. The main building blocks consist of Multi-Head Self-Attention modules which come with independent heads for the input signals. This allows for parallelization of the training process, and therefore it leads to faster training. However, conceptually, the point of interest is that it solely relies on Self-Attention in conjunction with some regular feed-forward Neural Network layers. Prior to the introduction of the Transformer the state-of-the-art in NLP tasks was primarily in RNNs combined with attention mechanisms. It therefore marks the end of a paradigm shift from recurrence to attention.

The building blocks of the Transformer are the Scaled Dot-Product Attention (SDPA) module and the Multi-Head Attention (MHA) module. Their schematics can be seen in Figure 7a and Figure 7b.

The SPDA module is conceptually close to the logic outlined in Section 2.2.3. There are some differences, however. First, the Transformer borrows database keywords to name the inputs. The attention scores are computed by multiplying a set of *queries* Q , and a set of *keys* K . Queries refer to what should be attended to and keys refer to what the current sequence is — in the metaphorical database. Later, once queries and keys are combined, the *values* V , are needed to compute the contextualized output¹². Second, Transformers are usually large and come with many layers of attention, which means the exploding and vanishing gradient problem might occur. For that reason, vectors are scaled down by the square root of the vector length. Third, the *SoftMax* function is applied to make the tensor values sum to 1. Lastly, masking may be applied to the combined queries and keys. This step is optional and further explained later.

The MHA module introduces two attributes. First, it wraps the SDPA into a Deep Learning module¹³. The MHA module first feeds the incoming values, keys and query streams into fully-connected neural network layers. The output vectors of these are fed into the SDPA. At the end, the result is also again passed through a fourth linear layer. Second, it introduces the notion of multi-heads. The assumption here is that multiple layers of attention learning might be beneficial given the vast network of relations even simple sentences can have between their words. Thus, the input is split into multiple heads h , the SDPA is performed, and later, the heads are concatenated back together¹⁴. This step also enables parallelization between the heads, since their computation is independent of one another. The MHA module is the core building block for the Transformer architecture with the Self-Attention mechanism at its heart.

The entire Transformer architecture is depicted in Figure 8. It consists of an encoder and a decoder. Both parts consist of multiple stacked submodules. Note how the output of the entire chain of encoder modules serves as input to all decoder modules, not only the first one.

After the embedding step, the input is enriched with positional clues. In Section 2.2.3 it was explained that Self-Attention lacks the encoding of order of the input series. While this can be an advantage — dispensing of proximity as main heuristic — order within a natural sentence still carries meaning in certain circumstances. The positional encoding works as follows. Imagine a token has been mapped to the embedding vector $[a, b, c]$. The positional encoding module now adds positional clues to this vector depending on where the token is located in the input sequence. This results in $[a+p1, b+p2, c+p3]$. During training the network learns what parts of the vector constitute the actual token, and what part signifies the position. This happens due to the many repetitions of the network being exposed to the same tokens in different context. Thus, it learns to separate $[a, b, c]$ from $[p1, p2, p3]$ ¹⁵.

¹² This is an analogy and serves only to refer to the input signals without confusion.

¹³ The SDPA has no trainable parameters on its own.

¹⁴ The number of heads h is a hyperparameter of the model's training process.

¹⁵ The positional encoding values are not part of the training process, they are static values. For details see [13].

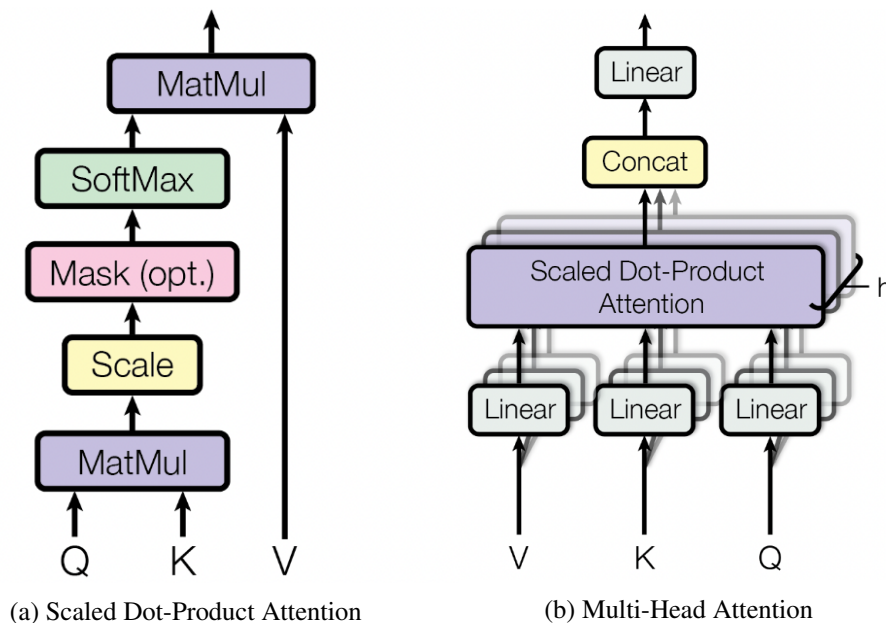


Figure 7: Depicted are the two building modules of the Transformer. The attention scores are computed by multiplying a set of *queries* Q , and a set of *keys* K . Queries refer to what should be attended to and keys refer to what the current sequence is. Once queries and keys are combined, the *values* V , are used to compute the output. The variable h refers to the number of attention heads. Images are taken from [13].

The rest of the architecture is straightforward. Enriched embedded inputs are routed through the MHA module, the output is summed with the original input¹⁶ and then normalized. This bifurcation of the signal is done to avoid vanishing and exploding gradients which may exist due to the architecture often consisting of many layers of encoder or decoder modules. In the case of the decoder module, a second MHA layer follows with an *Add & Normalize* step. Note that this second MHA module takes the values and keys from the Encoder output. Thus, here is where the encoded input is decoded. Lastly, both modules end in a position-wise Feed-Forward network followed by an *Add & Normalize* step. Details on this feed-forward layer can be found in [13] or in Section 3.6.4; it means two linear transformations are applied with an activation function in between. Note how the only real neural network modules used in the entire architecture are simple feed-forward layers. The complexity comes from the interplay between the various subcomponents and the attention mechanisms rather than the underlying trainable parameters.

In the first MHA module in the decoder module the input is masked. Masking is employed to prohibit looking at *future* tokens during training. For instance, at position $t = 4$ the decoder should only be able to access the previous tokens in the sequence $t = 0, 1, 2, 3$ together with the encoder states. Thus, tokens that occur at $t > 4$ are masked, i.e. set to $-\infty$. This forces the network to only consider past classifications. During inference, the model is therefore *conditioned* to only working with what was previously established. This also works in tandem with the fact that output embeddings are fed into the model with an offset of 1. Roughly speaking, this means that at any given moment, the decoder is only allowed to look at the past, then predict a new token, append that token to the list of classifications¹⁷, and continue until it is done.

Since their introduction, the Transformer architecture is used for various tasks. They find most application indeed in NLP tasks, but also Computer Vision tasks. They are also set to replace many RNN-based systems. Especially as pretrained models Transformers seem to shine for many tasks in fields like Machine Translation, Time Series classification, Name Entity Recognition, and more. Popular pretrained models BERT [14], the GPT models [15, 16], and others.

¹⁶ Refer to Figure 8 for the exact data flow; a written description may be more confusing.

¹⁷ Once appended, tokens in that list are immutable, i.e. later classifications cannot alter the already predicted tokens.

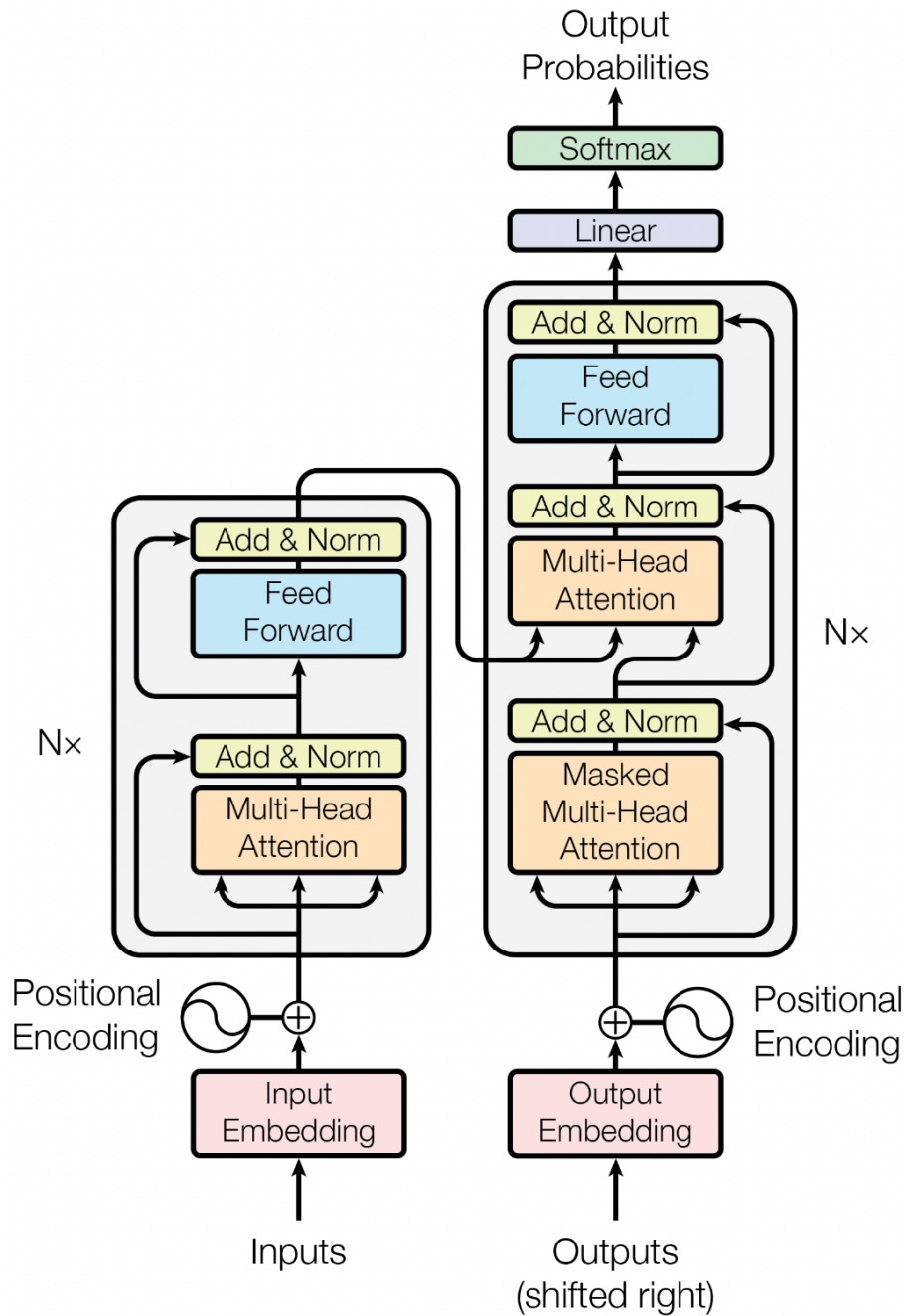


Figure 8: A sketch of the Transformer architecture as originally given by Vaswani et al. [13].

2.3 Handwritten Text Recognition

The goal of Handwritten Text Recognition (HTR) is to transcribe documents containing handwritten text into digital text. This can happen online, while writing, or offline, as a finished document [24]. HTR finds wide application in areas such as historic document transcription, medical form transcription and transcription of modern documents. Usually, the problem is approached by a combination of methods from fields such as Natural Language Processing, Computer Vision and Language Modeling [25]. This section will explain how the issue of context is amplified by handwriting. Then, a summary of the history of HTR is provided. Finally, the Connectionist Temporal Classification (CTC) framework is explained.

2.3.1 The significant Problem of Context in Handwritten Text Recognition

In Section 2.2 the challenges of NLP tasks were explained referring to context as a multi-layer problem that needs to be addressed holistically. Handwritten text poses additional challenges to this already significant issue. When processing digital text, the same characters are always the identical. The letter A used in one word is indubitably the same letter as used in another word. However, in handwritten text, this becomes more problematic. Firstly, there is inherent variability in writing styles between writers influenced by their cultural, educational and even artistic background. Letters come in all shapes and forms: thick and thin lines, slanted characters, additional ornaments like loops and circles, et cetera. See Figure 12 for some mild examples. The handwritten letter A can take on many shapes and forms, so even if it means the same, a system cannot rely on the letter always being the same.

Secondly, characters have no clear boundaries. They can overlap and frequently do so. Thirdly, humans are not perfect, their characters might simply resemble another character closer than the intended one. The human reader can use the context, the environment, of the letter to infer its intended meaning. For an example, see Figure 9. The letter in the middle might be an H, or it might be an A. Reading the vertical word THE a human reader will *see* the character resemble more the former, while reading the horizontal word CAT the letter now morphs into the latter.

Handwritten characters and their boundaries can be hard to identify, but it does not stop there. Words and sentences are not guaranteed to be written in a straight horizontal lines. Sentence can stretch over multiple lines, and even pages. The same holds for paragraphs and sections. Especially when considering historic documents, even more issue become apparent. Documents might have missing or destroyed parts. The writers might use unknown styles, fonts and languages. These additional challenges exemplify the need to incorporate context at all levels – only then can the more challenging HTR tasks be achieved.



Figure 9: This is an example of how context can help a reader to disambiguate a sloppily written character. The middle letter may be an A or an H. Depending on which word to read, either THE or CAT, a human reader can effortlessly disambiguate the middle letter.

2.3.2 The History of Handwritten Text Recognition in a nutshell

HTR was originally posed as a sequence matching problem. The input sequence resulting from a feature extraction process is aligned with the output text characters. Hidden Markov Models were employed, also as neural network hybrids [26, 27]. However, given the Markovian assumption that each observation depends on only the current state, they were deemed to miss contextual information [5]. Modern HTR methods involve either one or a combination of systems based on Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), nearest neighbor search or attention mechanisms [28].

RNN-based systems can partly extract linguistic context, though language modeling methods can improve performance by e.g. constraining the output probabilities [29]. Word embeddings can be used to map words into a semantic subspace and apply nearest neighbor search or other similar methods. Kang et al. [28] note that as an important disadvantage many language models rely on predefined lexicons which renders the recognition of out-of-vocabulary words impossible. This again shows the heuristic choices that are taken at many steps — to achieve high performance over authentic intelligent behavior.

Advancements in Deep Learning increased performance [28]. The sequential property of text and language has been addressed by RNNs [30], especially BiLSTMs. See Section 2.1.2 for details. The inherent constraints of languages can be leveraged to boost the optical recognition of characters and words via statistical language models and neural networks [29, 31]. Attention mechanisms were introduced to alleviate alignment of characters and words by learning what to pay attention to [32]. See Section 2.2.2 for detail on attention mechanisms.

When predicting characters over (pseudo-)time, the output represents a grid of character hypotheses. The best path across this grid needs to be found, e.g. via linguistic resources such as dictionaries [33]. For RNN-based systems the Connectionist Temporal Classification (CTC) framework is often used [10]. CTC-based systems form a conditional probability distribution over the target sequence. The most probable output can then be selected from that distribution. Thus, precise alignment is unnecessary [9]. Decoding strategies, such as beam search variations and language models can boost performance. CTC-based systems are limited in that the alignments of the input and output pairs are strictly monotonic, and the output length is bound to the length of the input [34]. Details on CTC follow in Section 2.3.3.

Initially, CTC methods were applied to the pixel grid directly. This results in all sorts of issues since pixel values are not representative of the shape they might make up. In fact, even the more basic difficulties of HTR, such as slanted characters, posed a challenge. In addition, manual selection of features from the input image was sometimes done, but that still limited RNN-CTC models [5]. The solution was provided by Convolutional Neural Networks (CNN). They were used for the visual feature extraction while RNNs enable sequence prediction. This improved end-to-end HTR drastically [7, 24]. Usually provided in the form of encoder-decoder architectures, such systems pose an alternative to the previous. The encoder takes the role of feature extraction of the input images, while the decoder is responsible for handling the output sequence trained on text corpora, token by token. By decoupling the feature extraction from the sequence decoder these models feature more flexibility, unbounded output sizes, and still suit the temporal properties of text [34, 35]. The decoupling also enables including attention mechanisms between encoder and decoder [20]. Via these mechanisms the model needs not to rely on mapping images to output tokens, but can learn language relations between the parts of the architecture [34]. Thus, the network can learn more specific patterns.

It was therefore clear that the RNN-based models need high quality input to achieve good performance. Many successes in HTR were achieved in conjunction with CTC as a good framework to tackle the sequential alignment issue for training. Especially, Multi-Dimensional Long Short-term Memory networks (MDLSTM) as sequence decoders showed impressive results [36]. However, they come with a high computational cost which led to the use of BiLSTM that are often applied to language modeling. BiLSTM models, like CNN-BiLSTM, perform similarly to MDLSTMs, albeit slightly less well [37, 24].

Bluche et al. [2] built an end-to-end attention-based system that does not rely on text paragraph segmentation. Sueiras et al. [8] built an HTR attention-based sequence-to-sequence model which requires

tuning of their sliding windows over the input images. Without the use of any language model, Michael et al. [34] achieved competitive results. Sueiras et al. [8] and Kang et al. [28] built encoder-decoder networks to transcribe words together with a lexicon containing the words in the training set. Similarly, but trained on character sequences, Poulos and Valle [38] experimented with the attention mechanism’s activation functions to find differences in the aligning between images and labels. These examples show remarkable results for HTR using attention-based encoder-decoder systems. Some researchers believe it will become the new state-of-the-art [39].

The mentioned RNN-based methods inherently lend themselves to modeling language and sequence data. However, recognition is still improved by the addition of post-processing steps integrating language models. This suggests that although conceptually appropriate, architectures involving RNNs are limited regarding their language modeling capabilities. Motivated by that and the high complexity of recurrent systems, the inherent need for RNNs has been questioned. Replacing the recurrent layers partly by convolutional layers results in easier-to-train models. The performance of these approaches suffers compared to recurrence-only layers, yet results of such recurrence-free methods are still noteworthy [30, 40, 41]. Further development led to systems based on CNNs or attention devices only [38]. Although not without its criticism [42] such methods are competitive to encoder-decoder networks [40, 41].

Vaswani et al. [13] proposed the Transformer with a complete reliance on attention mechanisms. See Section 2.2.4 for more details. Naturally, systems are proposed to include Transformers in HTR, specifically to model structures of language from the training data. Kang et al. [25] constructed an HTR system — for the first time, they claim — inspired by Transformers which also achieved remarkable results. For both optical and textual stages they adopted multi-head attention layers in order to recognize characters at the right step and to model language structures in the decoder. Synthetic data is used for pre-training. They achieve state-of-the-art performance, especially in few-shot learning situations after pre-training. Transformer-based architectures therefore constitute the latest in the series of HTR methods.

2.3.3 Connectionist Temporal Classification framework for Loss computation

HTR is a sequence-to-sequence problem. Fundamentally, that means that an input sequence, i.e. many columns of vertical pixel vectors forming an image, need to be translated into a sequence of ASCII characters. Clearly, there will be more pixel columns than translated characters — the input sequence length will be smaller than the output length. Observe Figure 10; ideally, a system would output the character sequence `Life` given the input image. However, since it is unknown how many characters there are in the image, a fixed length needs to be applied to accommodate a reasonable number of them. Thus, a working system would output a fixed-length sequence containing the most probable tokens for a slice of the input image. See the figure’s bottom row for an example of that.



Figure 10: An example of an output that correctly matches the classifications to the location of the input characters. At each output position a token is predicted. Note how that also entails predicting parts of a character that visually do not resemble the real token — e.g. at the end of the `e` character.

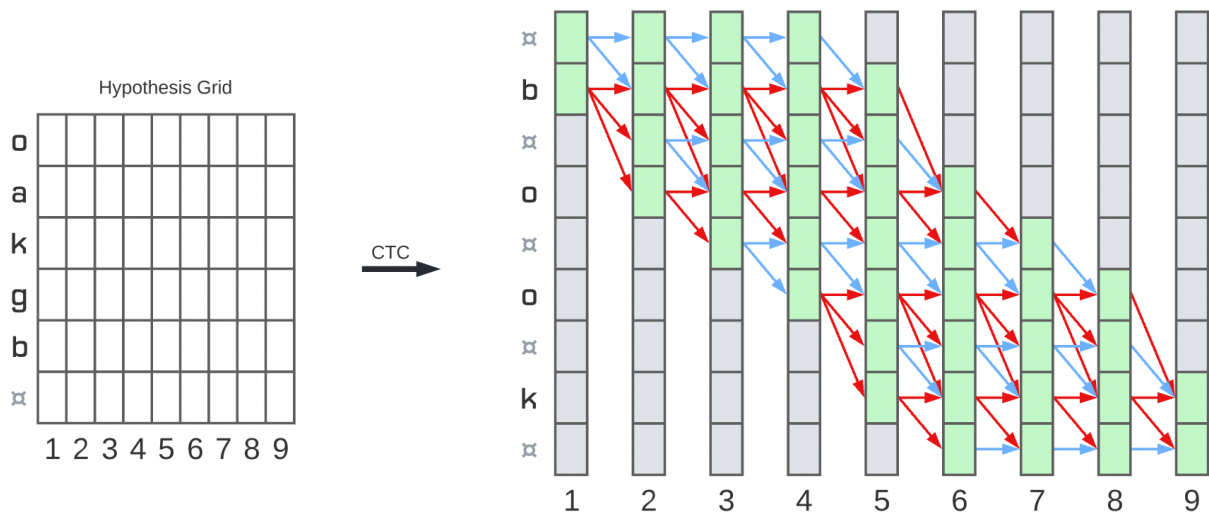


Figure 11: An example of how the CTC framework is used to reframe an output grid (left), to a more constraint graph (right) that allows finding all possible alignments over the output grid. Paths can be formed starting at the top left, and following the arrows. Those arrows are constraint in their direction.

The question arises how to train such a model. This poses a non-trivial issue. Although, assuming a supervised learning setup, the label of the image is known during training, the location of the individual tokens is certainly unknown. Moreover, it cannot be generalized given the variability of handwritten text. The Connectionist Temporal Classification (CTC) framework offers a solution to this problem.

Assume the output of an RNN-based system is a two-dimensional grid — probability distributions over the vocabulary across (pseudo-)time. Thus, at each location the system predicts the likelihood of each possible character to occur. An alignment is an expansion of the label that stretches across this hypothesis grid of the network’s output, see Figure 10 for the alignment `LLiiffeeeeee`. Given such an alignment, a system can be trained, e.g. via Cross Entropy Loss for each alignment position.

CTC enables finding all possible alignments over this hypothesis grid by reframing the grid. See Figure 11. On the left, a simple hypothesis grid is presented. It stretches across nine positions, each with a probability per character in the alphabet. First, rows of the original grid pertaining to characters not in the label are discarded. Second, in HTR an assumption is that characters strictly preserve the order of characters from input to output¹⁸. That means, the grid can be re-ordered such that the grid characters follow the label characters. Third, alignments pose a problem when dealing with intended double characters. As per this example, consider the label `book`. When finding alignments, repeated characters are expected as part of the process. Hence, when actually collapsing the output in order to obtain a usable classification, intended double characters such as the `oo` are lost. CTC solves this by introducing a new *blank* character to the alphabet, denoted `␣`. The character’s sole purpose is to deal with intended double characters by being inserted between them, thus `o␣o` in this case. Given that the network should learn that it denotes a separation between characters, it also needs to be possible for the network to predict this token between all characters in the output sequence¹⁹. Applying these three restrictions and assumptions, the original grid on the left becomes the one on the right.

Equipped with this reframed grid, all possible alignments can be found. Observe Figure 11 once again. First, note that within this grid the sequence flow is strictly monotonically going from the top left to the bottom right. The flow can never go up due to the aforementioned assumption of order²⁰. For the blank token, top left at position 1, only another blank token or the token `b` may follow. At the same position, `b`

¹⁸ In other sequence-to-sequence problems this might not be the case.

¹⁹ There are other methods of dealing with this issue such as introducing special characters per double-token occurrence.

²⁰ This assumption also restricts for what tasks CTC can be useful. Thus, it is both an advantage and a disadvantage.

can only be followed up by another b , a blank, or \circ , but nothing else. Filling out this grid with all possible sequences shows therefore all possible alignments.

Given the output probabilities, some of these alignments will be more likely to occur in the grid. This means, the alignments themselves form a probability distribution. To arrive at a loss function, CTC uses this whole distribution of all possible alignments to calculate how likely each character classification at each position is to occur, given the provided label and its possible alignments. These probabilities for each alignment are computed using the so-called Forward-Backward algorithm which is outlined by Graves et al. [9]. The result of that is then used to calculate the gradients to train the network in question.

In this thesis, the CTC framework is used to train the encoder in a Transformer-based system for an HTR task. In later experiments the CTC loss is part of a composite loss function. The results and discussions of how well CTC can be leveraged in this context is explained in Section 5 and later sections.

3 Methods

To answer the research questions, a series of experiments were conducted. The general methodology of those is described in this section. Note that the specifics pertaining to individual experiments are described in Sections 4 to 7. Concepts and technologies explained in Section 2 are referenced here.

The goal of this project is to compare two different Deep Learning architectures in an HTR task. Details on Deep Learning can be found in Section 2.1. The task is to accurately transcribe handwritten text from line-strip images into ASCII text, see details in Section 2.3. A Deep Learning pipeline was constructed which enables various configurations including the choice of the underlying model. These model are trained in a supervised fashion and in an end-to-end fashion. Most of the pipeline remains unchanged throughout the later experiments; only specific variables are changed. Thus, potential differences in performance can be attributed to those variables which allows for evaluation of the impact of the variables.

The dataset is described first in Section 3.1 to explain the line-strip images and their labels. In Section 3.2 and 3.3 the preprocessing steps are explained. The following sections (3.4 to 3.7) deal with the used models and their implementation details. Lastly, the evaluation methods are described in Section 3.8.

The implementation of the methodology was written in the programming language Python using various modules and packages, most notably the OpenCV library [43] and the PyTorch framework [44].

3.1 Dataset

The dataset that was used is IAM [4] which contains over 13,000 line-strips of unconstrained handwritten English text from about 400 writers. The text is written in full sentences, though each image only contains one line of handwriting — sentences are cut. A difficulty of this dataset lies in its variety of writers of the handwritten content. Each writer has a different style of writing which results in vast difference in written letters and words, hence increasing the difficulty of the recognition task, see Section 2.3 for details. It is worth pointing out that the dataset also includes erroneously labelled line-strips that are marked as such. These examples were excluded, so all further numbers are regarding the correctly labelled data only.

According to Marti and Bunke [4] this dataset lends itself well to tasks that make use of linguistic understanding that surpasses the lexicon level. This is because the knowledge could be inferred from the corpus that the dataset is based on. The labels that the images are paired with are stored in a file. Each line-strip image is paired with one a string of characters representing the translation of that image. The string contains the letters in ASCII encoding as conventionally used, but space characters are replaced by the ASCII pipe symbol: |. Three examples of line-strip images and their labels can be seen in Figure 12.

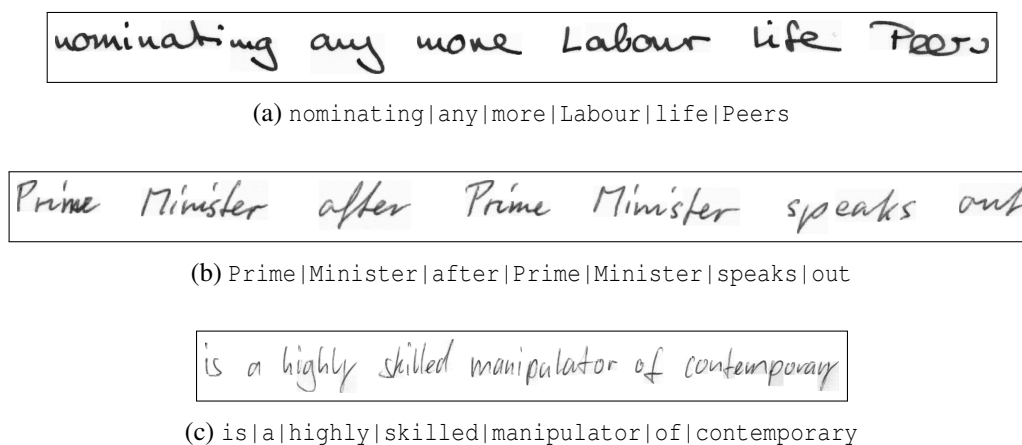


Figure 12: Three examples from the IAM database. The labels are provided in the captions. Note how | denotes a space. Also note how length of label does not translate to length of line-strip image.

3.1.1 Data Split

There are a total of 11,340 examples in the IAM dataset. For later validation and test purposes, the dataset was split into three distinct sets: a training set, an evaluation set, and a test set. The training set is used to train the model. The test set contains examples that were not in the training set to assess the performance on previously unseen examples. The evaluation set is a small subset of the test set to assess how well the model is performing after each epoch. Additionally, to validate the model on all available data, k -fold validation was used with a $k = 5$. That number was chosen for fairly arbitrary reasons: it is a prime factor of 11,340, not too large, and not too small; it falls within the normal range of k 's chosen in k -fold cross validation steps. This means that the entire dataset is split into 5 partitions. In 5 different runs, each with an independently trained model, a different partition is declared the test and evaluation set. That means each partition contained $11340/5 = 2268$ examples.

For training purposes a mini batch size had to be chosen. Through testing and, again, arbitrary choosing, a mini batch size of 42 was chosen. Thus, the training set consists of 9,072 examples which are split into 216 mini batches of size 42. The evaluation set contains one mini batch, so 42 examples. The test set contains 2226 examples in 53 mini batches.

3.2 Preprocessing of Images

Before the examples and their labels were processed by the Machine Learning systems, they were processed in a few ways. This preprocessing is explained in this subsection. Before any other preprocessing happened, all images were resized²¹ to a height of 64 pixels keeping the aspect ratio and therefore a variable width. Afterwards, the dataset was augmented using various methods to generate variations of the input images. An explanation of that follows now.

3.2.1 Augmentation

For comparison reasons, the sizes of the Deep Learning models were approximately matched. However, given the overall complexity of BiLSTMs, Transformers and a Gated-Convolutional frontend, the models were still fairly large — about 400,000 trainable weights. That means that with only 9,000 training examples the risk of over-fitting is high. A possible regularization method is dataset augmentation.

During training, each example and each variation thereof was used once per epoch. For each example one out of seven random augmentations was chosen. Most of these augmentations had their own element of randomness which further increases training set. What follows is a breakdown of the methods.

1. Image Morphing

Image Morphing was adopted from Bulacu et al. [45]. It works by displacing the pixels of an image. This displacement is done randomly and later smoothed by a Gaussian convolutional kernel. The image pixels are rescaled using an average amplitude. The authors pose an analogy to drawing an image on a rubbersheet and then stretching and contracting it at various places. This results in a variety of transforms for each image, and thus constituting a potent augmentation method. An example can be seen in Figure 13b.

Image morphing is characterized by the average pixel displacement A and the smoothing radius σ . After testing, it was found that leaving A at a constant value of 1, and varying σ achieved the best results for this project. The displacement radius σ was sampled uniformly from the set $\{6, 7, 8, 9\}$ for each example. Those values were found to produce the best results as they are not distorting the original image too much, but still provide a noticeable change.

It produces 4 different augmentations, one for each value of σ . Note, that the true number of unique outputs is larger given the randomness involved in creating the displacement field. However,

²¹ Resizing was performed using bilinear interpolation from Bradski [43].

this should only account for slight variations, and it constitutes a net positive for the sake of regularization in any case, thus details on the precise count are omitted going forwards.

2., 3. Left and Right Shearing

These augmentation options refer to horizontal shearing operations. Shearing an image means gradually increasing the horizontal pixel offset when moving vertically over the image, starting at the bottom. A rough analogy is given by italic fonts, which can be compared to a *right shear*. Both left and right shears were used. Examples can be seen in Figure 13c and Figure 13d.

The method was implemented using wrap perspective functions via a transformation matrix M using tools provided by Bradski [43]. This matrix M , as seen in Equation 11, is simply the identity matrix with the added shear variable s . This variable is either positive, for left shears, or negative, for right shears. Further, the amplitude of this variable determines the magnitude of the shear. The amplitude was uniformly sampled from the set $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. These two methods combined produce 10 different augmentations.

$$M = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (11)$$

4., 5. Shearing and Image Morphing

These two augmentation options are compound operations. First, a shear is selected, either left or right, then, the sheared image is transformed by the aforementioned Image Morphing operation. The methods are the same — including variable sampling — as described above. Examples can be seen in Figure 13e and Figure 13f. These operations combined produce 40 different augmentations, which come from 10 different shears and 4 different image morph transforms per sheared image.

6 Thickening

This method results in images with lines that are thicker than the original. This operation can be roughly compared to **bold font**. An example can be seen in Figure 13g.

The method was implemented using an erosion function. Erosion iteratively 'removes' pixel from a shape in an image. Thus, the object in the foreground will slowly shrink in size. The foreground in such a case is defined by the color white. Therefore, in the case of the line-strip images, the white parts will be eroded away, leaving the black parts — the handwritten text — more pronounced. Thus, seemingly paradoxically at first, erosion is used to thicken the lines.

For the purposes of the erosion function a 2x2 kernel, consisting of only ones, is applied in iterations. The magnitude of the operation is defined by the number of iterations, i.e. how often erosion is applied. The number of iterations is uniformly sampled from the set $\{1, 2, 3\}$. The thickening augmentation produces 3 different transformations.

7 Thinning

This method works identically as the Thickening operation, with the one exception that instead of erosion, dilation is used. Dilation is the reverse process of erosion — pixels are iteratively added to the foreground, the white shapes. Thus, the black handwritten text shrinks in shape. An example can be seen in Figure 13h.

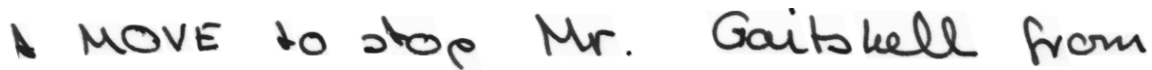
In this case however, only one augmentation is created as anything thinner than the output after one iteration becomes quickly too thin. Some text in the IAM dataset was written by writers with an already thin line width. Making those lines even thinner results in various issues such as line breaks, vanishing features, etc.



(a) The original line-strip image.



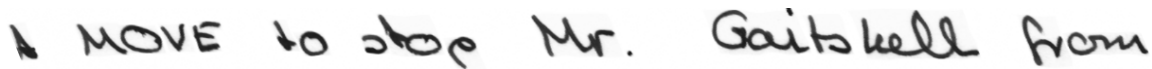
(b) **Image morphing** with average pixel displacement $A = 1$ and a smoothing radius $\sigma = 8$.



(c) An example of a **left shear** augmentation.



(d) An example of a **right shear** augmentation.



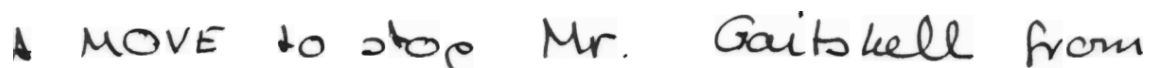
(e) An example of a **left shear** in combination with **image morphing**.



(f) An example of a **right shear** in combination with **image morphing**.



(g) An example of a **thickening** augmentation.



(h) An example of a **thinning** augmentation.

Figure 13: An overview of the various augmentation methods used to increase the example count of the IAM dataset as part of a regularization technique.

Overall, this means there are $4 + 10 + 40 + 3 + 1 = 58$ different augmentations possible. This in turn means that during the training phase, the model could get exposed to up to $9,072 \cdot 58 = 526,176$ (quasi-)unique images²². Note, augmentations were uniformly sampled out of the seven possible methods, not out of the 58 possible augmentations. Thus, each method had an equal chance to be chosen each epoch per example. The random variables within each method were again uniformly sampled from, but given their unequal numbers of variables per method, the total sampling space is not uniform.

The augmentations and their parameters were chosen after extensive testing. One can see that combinations of methods, like Shearing and Image Morphing, were also tested. However, some combinations, e.g. thickening and shearing did not result in better performance overall. Many of those combinations, and also parameter values, eventually were left out. The current set of augmentations prevailed as a good balance between enough images and not too complicated transforms. Note however, this is certainly not an exact science and should be regarded as bias in this project's setup.

Lastly, note that some augmentations that are classically associated with images, especially in a convolutional frontend context, were completely omitted as they are incompatible with the *natural task* of HTR. Those include rotation, mirroring, extreme translations, color transforms and other distortions. To illustrate this, imagine a video of car driving to the left, turning, then driving to the right. Analyzing this video frame by frame, the car never stops being a car. A car detection task can therefore make use of mirroring and rotation augmentations as the concept of the car is contained within the shape of the car itself. That does not hold for letters and words. Rotating and mirroring letters can change their meaning, or make them meaningless. Take the letter *P* for instance. Mirrored horizontally it becomes a lower-case *b*, mirror vertically it becomes meaningless; and, if both transforms would be applied, it becomes the letter *d*. The meaning of the letter is not contained by only the shape of the letter, but also by its orientation.

3.2.2 Image Preparation

All images had to be prepared to match the input requirements of the systems. Thus, after an augmentation was performed, the images were resized, the values were adjusted, and padding was added. Those steps and their details are described now.

– Resizing

The system accepts a specific input shape. The example images however come in variable heights and widths. Therefore, they need to be resized. Resizing needs to happen without distorting the images — the overall shape of the lines should be conserved, no squeezing or stretching should occur. The best way to achieve that is by resizing all images to a fixed height and proportionally scaling²³ the width with it. This preserves the aspect ratio of the images. Resizing all images to a fixed height of 64 pixels yields a mean width of 955 pixels. The mean however is not relevant since an input shape needed to be chosen that allows for all images. Hence, the maximum adjusted width of 2235 pixels is used.

– Pixel Value Adjustment

Each pixel of a given line-strip image comprises three values, one for each color channel²⁴. Each of those values falls within the range of 0 (absence) and 255 (presence). Although such values are possible to process by the Machine Learning systems, the values here are redefined to fit on a range from -1 and 1. This redefinition enables a clearer distinction between the absence of information (0) and emphasizes the contrast between blank background (-1) and dark handwritten lines (1).

²² Although augmentations certainly help, this set of (quasi-)unique *images* is not the same as an equivalently sized set of unique *examples* — augmentation reduces the risk of overfitting, but it does not solve it.

²³ Again, this is performed via bilinear interpolation using the implementation of Bradski [43].

²⁴ Although the images include three color channels they appear to be grayscale only. Thus, an extra preprocessing step to summarize those three channels into one is possible, however, for the sake of compatibility in this project the extraction of relevant information such as this is left to the visual frontend of the system.

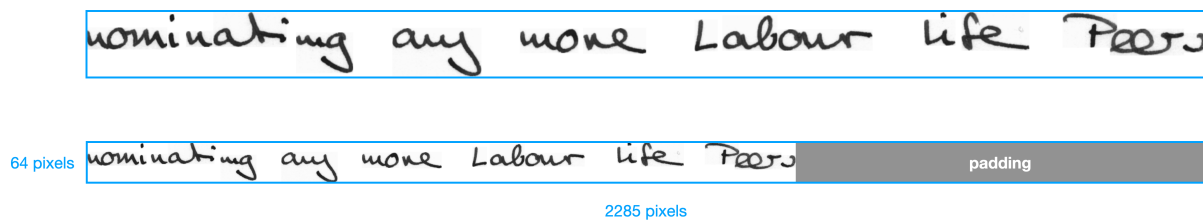


Figure 14: An example of how an input image is resized and padded.

– Padding and Extra Padding

Images were padded to fit within the newly selected shape 64x2235. For instance, if an image is of size 64x1000 after the resizing step, the remaining 1235 width pixels are artificially added to image in the form of zero values for each color channel. The padding is added to the right margin of the resized image. See Figure 14 for an example.

After adjusting the pixel values in the previous step, the value 0 does not appear frequently anymore in the image data, especially not in large chunks. When padding the images with 0 it carries with it a rather explicit encoding of the end of a line.

This explicit association with the end of a line creates the issue that some images only have little or no padding. In such cases, the systems will not stop producing characters as now this wall of padding is expected. To circumvent that, extra padding is added to extend this explicit coding to all images. An extra 50 pixels of 0 values are appended to each image. This results in a final shape of 64x2285 pixels.

An example of how the resizing and padding looks like can be seen in Figure 14. The final shape of the input tensor for the image examples is therefore (42, 3, 64, 2285). In words: one mini batch includes 42 images, each with 3 color channels, 64 height pixels and 2285 width pixels.

3.3 Preprocessing of Labels

As shown in Section 3.1, the labels included in the dataset are strings of text which represent the correct translation of the line-strip images of handwritten text. The labels do not include whitespace characters but rather denote the space between words with the pipe symbol |.

To prepare label data for the Deep Learning system, characters and symbols needed to be replaced by a numerical representation since the system cannot deal with anything else. For that purpose, each symbol needs to be replaced by a number. This can be done by encoding the text using a dictionary that contains each symbol and its numerical value. Before that, the role of special characters needed to be examined. The space character is already a special character provided by the dataset that explicit encodes white-spaces and therefore the boundary between words. For the purposes of this project three additional special characters will be included.

- A **start-of-line** token is inserted at the start of the string. This token should make explicit the beginning of a new line-strip. If correctly learned, the systems will associate this symbol with the start of a line. During inference simply giving this symbol to the system explicitly denotes the need for a translation to start. This is important for the Transformer-based model as it predicts tokens one-by-one using the start-of-line token as a signal to start. This symbol is €.
- Similarly, the **end-of-line** is also explicitly encoded by a symbol — \$. This provides a marker for when to stop inferring new symbols. The system should learn that this symbol is associated with the trailing padding. Thus, the start-of-line and end-of-line token make explicit the boundaries of the text.

- During inference, the system might output repeated characters as it recognizes the same token across multiple time steps due to the sequence-to-sequence alignment problem. See Section 2.3.3 for details. During a post-processing stage repeated characters are thus removed, see Section 3.7.2 for specifics. However, that leaves the issue of how to deal with actually repeated characters, e.g. in the word `book`. Following the explanation of the CTC framework, see Section 2.3.3, this is handled by a **blank** token, `␣`. It is inserted between repeated characters to make explicit that the repetition is intentional. For example, the word `book` would be encoded as `bo␣ok`²⁵.

The dictionary was constructed using all symbols that appeared in the dataset plus the aforementioned special characters – 82 symbols in total. Lower-case letters, 26 symbols, and upper case letters, another 26 symbols, were added separately, as well as all digits, 10 symbols. The remaining symbols were special characters including the ones above, and others provided in the labels, e.g. `+`, `&` and `?`.

Each symbol in the dictionary was assigned a number from 0 to 81. Each label, after adding the aforementioned special characters, was encoded into a series of numbers using this dictionary. In the post-processing step, this same dictionary was used to decode the output back into ASCII text.

Lastly, the encoded labels had to be padded. Labels come in different lengths, for obvious reasons. However, within the Deep Learning system, one gains several advantages when data structures are identical in length. These reasons are pure implementation details, however. It would not affect the system’s outcome as a whole if this is handled differently. Thus, this step is included for completeness’ sake. The labels were padded by appending zeros²⁶ to the end of the encoded series of numbers. They were appended until the total length of the label was 100. This obviously mirrors the shape of the output — the system is allowed to output up to 100 characters per given line-strip.

Examples of those three steps are provided now. Notice how the example label in (1) becomes (2) after adding the special characters. It then gets encoded using the dictionary and becomes (3). Lastly, padding is added so the final encoded label is in (4). The final shape of the input tensor for the example labels is therefore (42, 100). In words: one minibatch includes 42 labels, each with 100 encoded characters.

(1) `nominating|any|more|Labour|life|Peers`

(2) `€nominating|any|more|Labour|life|Pe€ers$`

(3) 1, 17, 18, 16, 12, 17, 4, 23, 12, 17, 10, 3, 4, 17, 28, 3, 16, 18, 21, 8, 3,
41, 4, 5, 18, 24, 21, 3, 15, 12, 9, 8, 3, 45, 8, 0, 8, 21, 22, 2

(4) 1, 17, 18, 16, 12, 17, 4, 23, 12, 17, 10, 3, 4, 17, 28, 3, 16, 18, 21, 8, 3,
41, 4, 5, 18, 24, 21, 3, 15, 12, 9, 8, 3, 45, 8, 0, 8, 21, 22, 2, 0, 0, 0,
0,
0,
0, 0, 0, 0, 0, 0, 0

²⁵ Note that the CTC method also inserts blank tokens between each token for the sake of an intermediate representation to find all alignments. This is handled by the CTC loss computation as provided by Paszke et al. [44], however. See details in Section 2.3.3

²⁶ One might wonder if the value of 0 matters here since it really stands for the blank token `␣`. During the CTC loss calculation however, the true lengths of the labels are provided, and the system only uses the label contents up to the original length. The remaining tokens, i.e. zeroes, are simply omitted. In the case of the Label Smoothing loss, the system will simply learn that the blank token not only denotes spaces between tokens, but white space in a broader sense. Regardless, testing revealed that there are no disadvantages using zeroes as padding.

3.4 Gated-Convolutional Visual Frontend

This section and the following lay out the details of the Deep Learning systems used in the later experiments. Following the focus on comparability, the systems have the same input examples. Therefore, both systems need a visual front-end that can encode the image data of the line-strips. This visual front-end is identical in both cases and is therefore described separately first. The resulting latent space from this visual frontend serves as input for both systems later.

The frontend encodes the images into a compressed representation. During training, it learns features of those and represents those in a latent space. Both models feature the same visual frontend²⁷ in their architecture to eliminate the influence of the CNN from what might account for later differences.

The frontend was inspired by Bluche and Messina [37]. Fundamentally, the frontend is made up of a number of two-dimensional convolutional layers, see Section 2.1.3 for an explanation of Convolutional Neural Networks. They also appear in a gated version in the architecture. A comprehensive architecture overview can be seen in Figure 15. Here, blocks denote either a function or a network layer. Arrows depict the flow of the tensors. The details of the layers and functions are described below, following the flow of Figure 15. As explained in Section 3.2.2, the images are represented as four-dimensional tensors of shape $(42, 3, 64, 2285)$ and thus serve as input to the front-end.

Convolutional Layer 1 This layer has 3 input channels and 4 output channels. The kernel size is $(2, 2)$ and with a stride of $(1, 1)$. No padding is applied. The output is passed through a hyperbolic tangent activation function, *tanh*. Those are also used later, but omitted in the description. The resulting output tensor is of shape $(42, 4, 63, 2284)$.

Convolutional Layer 2 This layer has 4 input channels and 8 output channels. The kernel size is $(3, 3)$, the stride is $(1, 1)$. No padding is applied. The output shape is $(42, 8, 61, 2282)$.

Convolutional Layer 3 This layer has 8 input channels and 16 output channels. The kernel size is $(2, 3)$ and with a stride of $(2, 3)$. The padding is 1. The output shape is $(42, 16, 31, 761)$.

Gated-Convolutional Layer 1 This layer has 16 input channels and 16 output channels. The kernel size is $(3, 3)$ and the stride is $(1, 1)$. The padding is 1. The output shape is $(42, 16, 31, 761)$.

Convolutional Layer 4 This layer has 16 input channels and 32 output channels. The kernel size is $(3, 3)$ and the stride is $(1, 1)$. The padding is 1. The output shape is $(42, 32, 31, 761)$.

Gated-Convolutional Layer 2 This layer has 32 input channels and 32 output channels. The kernel size is $(3, 3)$ and the stride is $(1, 1)$. The padding is 1. The output shape is $(42, 32, 31, 761)$.

Convolutional Layer 5 This layer has 32 input channels and 64 output channels. The kernel size is $(2, 3)$ and the stride is $(2, 3)$. The padding is 1. The output shape is $(42, 64, 16, 254)$.

Gated-Convolutional Layer 3 This layer has 64 input channels and 64 output channels. The kernel size is $(3, 3)$ and the stride is $(1, 1)$. The padding is 1. The output shape is $(42, 64, 16, 254)$.

Convolutional Layer 6 This layer has 64 input channels and 64 output channels. The kernel size is $(3, 3)$ and the stride is $(1, 1)$. The padding is 1. The output shape is $(42, 64, 16, 254)$.

MaxPool & Flatten This pool layer has a kernel size of $(16, 1)$ and a stride of $(16, 1)$. This reduces the tensor to $(42, 64, 1, 254)$. After flatten in the third dimension, the tensor becomes $(42, 64, 254)$.

Linear The tensor passes through a translation layer that compresses the third dimension from 254 into 100. The last two dimensions are then swapped, resulting in a final output tensor shape of $(42, 100, 64)$.

²⁷ Although both systems have an identically constructed frontend, the systems are trained independently in an end-to-end fashion. Thus, the weights of the visual front-end may indeed be very different in both cases.

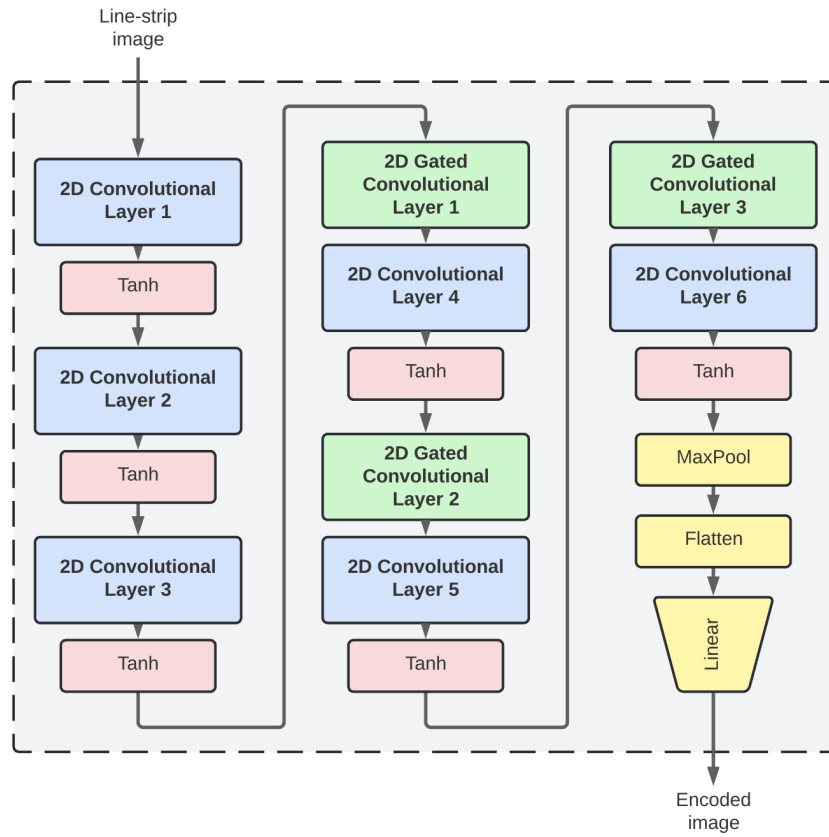


Figure 15: A sketch of the Visual Frontend architecture.

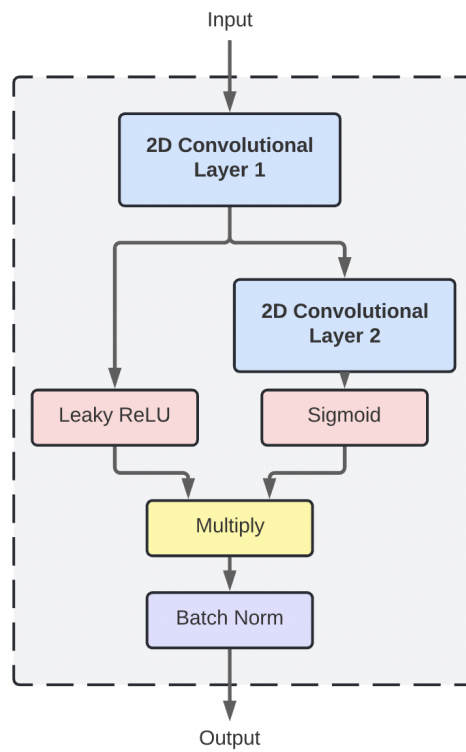


Figure 16: A sketch of the 2D Gated-Convolutional Layer architecture.

3.4.1 Gated-Convolutional Layer

The inner architecture of the 2D Gated-Convolutional layer are depicted in Figure 16. Similarly to before, the flow is denoted by arrows. A description of the individual layers and functions follows now. Note that each gated layer comes with different settings outlined in the above overall details.

Convolutional Layer 1 This layer copies the given settings. The output shape does not change.

Convolutional Layer 2 This layer copies the given settings. The output shape does not change.

Leaky ReLU The output of Convolutional Layer 1 is split and passed through a Leaky Rectified Linear Unit activation function, *Leaky ReLU*.

Sigmoid Activation The output of Convolutional Layer 2 is passed through Sigmoid activation.

Multiply & Batch Norm The output of the Leaky ReLU and the output of the sigmoid are multiplied. The output is passed through a 2D Batch Normalization function.

3.5 LSTM-based Model

The BiLSTM-based model poses the first of the two models that are being compared. This model is inspired by Bluche and Messina [37] and others. The most relevant part of this model is constituted by bidirectional LSTM cells which are described in more detail in Section 2.1.2. The entire architecture of this model is sketched out in Figure 17. Arrows denote the direction of the tensor flow; shapes represent network layers or specific functions. What follows is a detailed description of those components.

Visual Front-End This part is the previously outlined Visual Frontend with its convolutional layers.

BiLSTM 1 The input shape to this layer is $(42, 100, 64)$. The layer's input feature size is therefore set to 64. The hidden size, which is the output size of this layer is set to 82 which is the size of the dictionary used for encoding the example labels. See Section 3.3 for details. Since this LSTM layer is bidirectional the actual output size will be $82 \cdot 2 = 164$. The output tensor shape is therefore $(64, 100, 164)$.

Linear 1 This layer has an input size of 164 and compresses it to 82. The output shape is $(64, 100, 82)$. The output is passed through a *tanh* activation function.

BiLSTM 2 This layer's input and output size is 82. The output shape is $(64, 100, 164)$.

Linear 2 This layer takes the input size 164 and compresses it to the dictionary size, 82. The output is passed through a *Log Softmax* activation function. The output shape is therefore $(64, 100, 82)$.

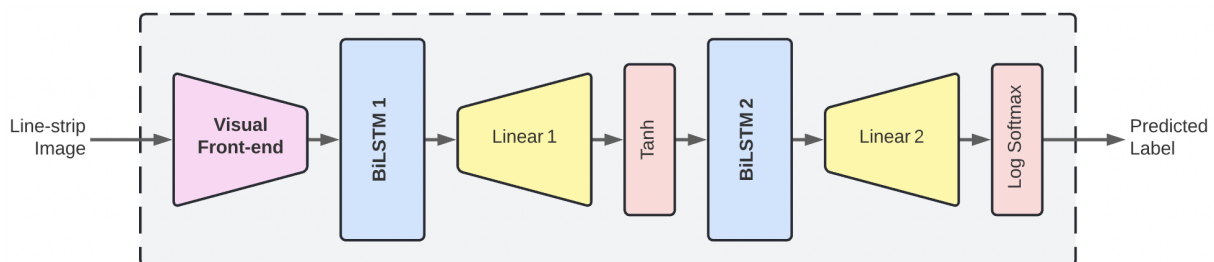


Figure 17: A sketch of the BiLSTM-based model.

Note that there is basically no difference in architecture and tensor flow between training and inference. During inference, images serve as input as described, the tensors are pushed through the model, and the output at the other end is a grid of character hypotheses which can be used to compute a classification. During training, this classification is judged against the correct label. The resulting loss is used to adjust the model's weights during back-propagation. Thus, the only difference between an inference step and a training step is the loss calculation and the back-propagation²⁸.

3.6 Transformer-based Model

The Transformer-based model poses the second of the two models that are being compared. The model is inspired by Kang et al. [25], see Section 2.3.2 for some more details. The model includes an encoder part and a decoder part. The images serve as input to the encoder, while the preprocessed labels are the input to the decoder. Note that part of the encoder is used as input to the decoder. The overall architecture can be seen in Figure 18. Arrows denote tensor flow direction; shapes stand for network layers and functions. The following is an explanation of first, the encoder, and second, the decoder.

Visual Front-End This part is the previously outlined Visual Frontend with its convolutional parts.

Layer Norm This layer normalizes the tensor over the mini batches. It is implemented using the module provided by Paszke et al. [44] which follows the methodology of Ba et al. [46]. All subsequent mentions of Layer Norm follow the same implementation.

Positional Encoding This module takes the input tensor and enriches it with positional clues which are pre-computed value mixtures of sinus and cosine functions²⁹. Essentially, this provides clues about where in the current context a token is located. For more details see Section 2.2.4.

Dropout This layer randomly sets some input nodes in the network to 0 during training. This serves as regularization technique. See Section 2.1 for details.

Encoder Layers This component is made up of two layers. The input tensor is the input to the first layer, and the output of the first layer is input to the second. The output of the second is the output of the entire encoder layers (green box in Figure 18). The details of those layers are explained in Section 3.6.1.

Dropout This layer has the same function as the previous one.

Linear During the previous steps the shape of the tensor does not change being (42, 100, 64) from the Visual Front-End. This layer serves as translation from 64 features to 82, which is the dictionary size. The resulting shape is therefore (42, 100, 82). The output is finally passed through a Log Softmax layer. This constitutes the output of the encoder.

The encoder effectively has two outputs, first the output after all components have been applied up to the last *Log Softmax*, and, a second output after the Encoder Layers (green box) are finished. This latter output serves as input to the decoder. The decoder is now explained.

Target Embedding The preprocessed labels serve as input to the decoder. Thus, they are first converted into vectors of 64 dimensions. This is done using an embedding layer. See Section 2.2.3 for more details on embeddings. Note that the input here is not a list of words as would be usually the case with embedding layer, rather it is a list of characters. The embedding therefore learns to represent the characters as vectors following their contextual environment. Following the original paper [13], the tensor values of the output of this layer are scaled by a factor of $\sqrt{64} = 8$.

²⁸ The Transformer-based model's differences between inference and training are more significant, thus it is worth pointing out that the BiLSTM-based model does not feature such differences.

²⁹ This is not part of the training process, those values are static.

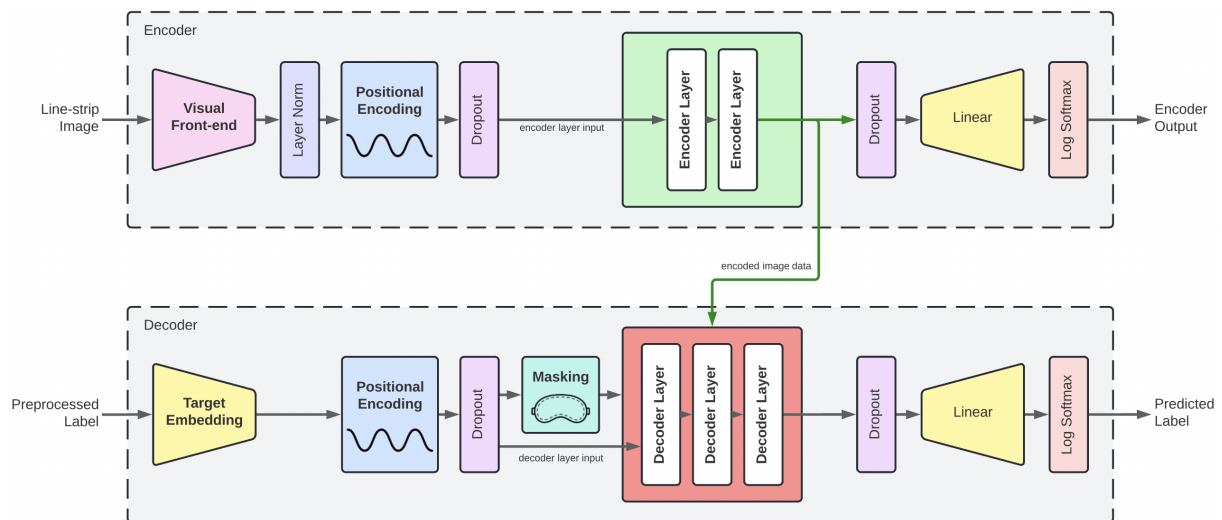


Figure 18: An overview of the complete transformer-based architecture.

Positional Encoding This module works identically to the one in the encoder.

Dropout This module works identically to the ones in the encoder.

Decoder Layers The output of the previous component is masked, see Section 2.2.4 for why and how. The masked tensor together with the original one serve as part of the input to the decoder layers. The other input comes from the encoder, specifically the output of the Encoder Layers. There are three Decoder layers. The original tensor is the input to the first layer, and the output of the first layer is input to the second, and again, the second layers output is input for the third layer. The masked tensor and the original encoder layer input however serve as identical input to all layers. The details of those layers are explained in Section 3.6.2.

Dropout This layer has the same function as the previous one.

Linear During the previous steps the shape of the tensor does not change being $(42, 100, 64)$ coming from the embedding layer. This linear layer serves as translation from 64 features to 82, i.e. the dictionary size. The resulting shape is therefore $(42, 100, 82)$. The output is passed through a *Log Softmax* function. This constitutes the output of the decoder and therefore the entire system.

3.6.1 Encoder Layer

The encoder layer represents the core module of the encoder. See Figure 19 for a depiction of its modules and tensor flows. Note that this layer is used twice, one directly after the other. The Encoder Layer output is used as Encoder Layer input to the next layer. Thus, the input of the second layer is not the same as the input of the first layer.

Layer Norm First, the input is normalized using the Layer Norm function. For details see the above Layer Norm description.

Multi-Head Attention The normalized tensors then serve as input for the Multi-Head Attention module, see Section 3.6.3 for details. Note that all three inputs, for keys, values, and queries are identical. The output of the attention module is added together with the original encoder layer input.

Layer Norm Once more, the output is normalized using the Layer Norm.

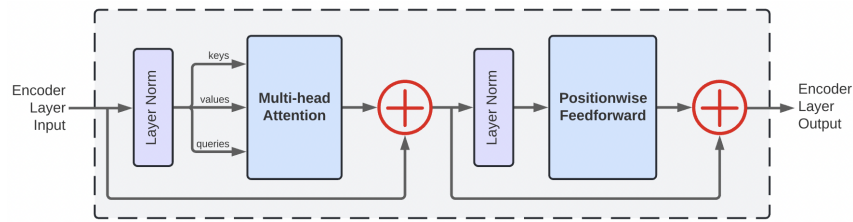


Figure 19: An overview of the Encoder Layer architecture.

Position-wise Feedforward The normalized tensors go through the Position-wise Feedforward module. See Subsection 3.6.4 for details. Lastly, the output is added to the original input before the second Layer Norm. The result is the output tensor of this layer.

3.6.2 Decoder Layer

The Decoder Layer module is structured similarly to the Encoder Layer module, a detailed description is therefore omitted. The main difference is that it chains two Multi-Head Attention modules in a row before the Position-wise Feed-forward module. The second attention module takes the encoded image data. The precise flow can be seen in Figure 20. Note that this layer is used thrice, chained one after the other. The Decoder Layer Output is used as Decoder Layer Input to the next layer. However, the masked tensor as well as the encoded image data, are copied to each individual layer, i.e. they are identical in values.

3.6.3 Multi-head Attention Module

This module encapsulates the attention mechanism of the Transformer architecture. It is based on the Vaswani et al. [13] which is described in more detail in Section 2.2.4. The architecture overview can be seen in Figure 21.

Linear Layers The three inputs all go through their own linear layers. Since the inputs are the same, the layers are also identical. The output shape of each layer is $(42, 100, 64)$.

Head Splitting The tensor is split into multiple heads. The number of heads is 2. That means the tensors are reshaped from the previous dimension into $(42, 2, 100, 32)$, i.e. the last dimension is split into up into an extra dimension.

Dot-Product Attention The split tensors are then directed into the Dot-Product Attention module together with the applied optional mask. The keys are transposed, from $(42, 2, 100, 32)$ to $(42, 2, 32, 100)$.

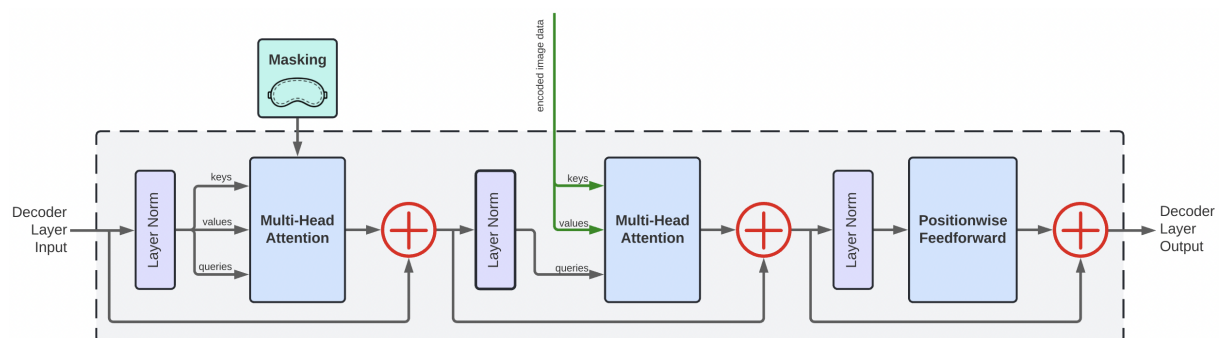


Figure 20: An overview of the Decoder Layer architecture.

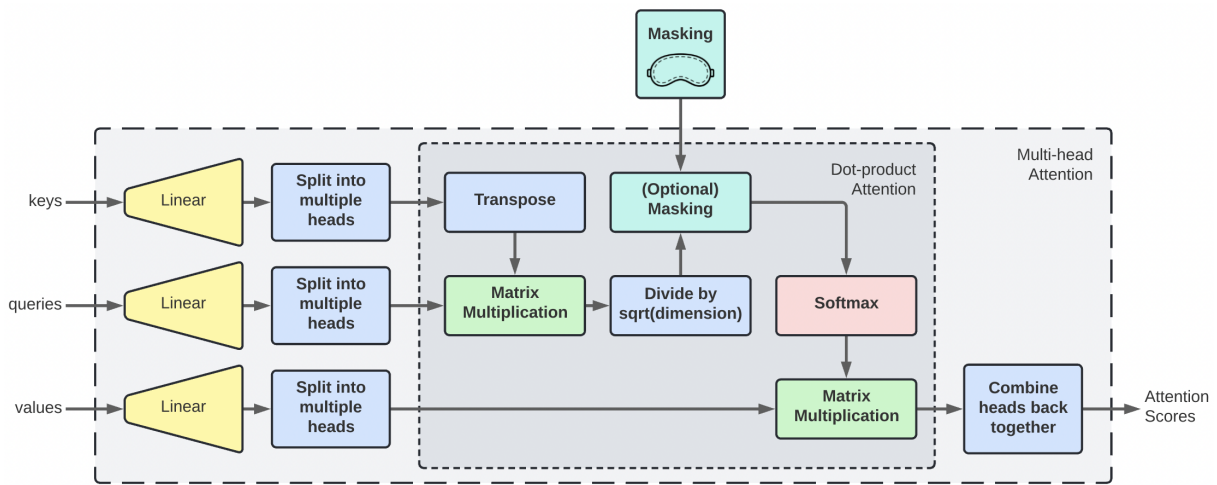


Figure 21: An overview of the Multi-Head Attention architecture.

Then, they are multiplied with the queries which results in a tensor of shape $(42, 2, 100, 100)$. In addition, the output is scaled by dividing the tensor's values by the square root of 64, so 8. The scaled result is masked if a mask is provided. If not, the masking step is simply skipped. The potentially masked tensor is then pushed through a *Softmax* layer. Finally, the result is multiplied with the values. This output shape is $(42, 2, 100, 32)$. This constitutes the output of the Dot-product attention module.

Head Combination The two heads are combined again in a reversed process of the former splitting. The output shape is $(42, 100, 64)$ once again and thus also the final output of the Multi-Head Attention module.

3.6.4 Position-wise Feedforward

This module is composed of two linear layers, two dropout layers and one Rectified Linear Unit. Figure 22 illustrates how those parts interact. This module is based on Vaswani et al. [13]. The linear layers expand the input, then apply the *ReLU* activation function, and finally compress it down again. This is called Forward Expansion.

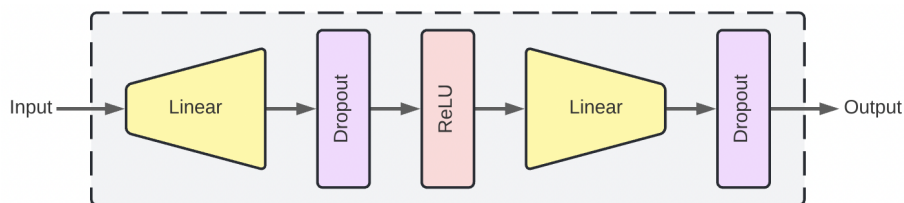


Figure 22: An overview of the Position-wise Feedforward module.

3.7 Implementation Details

Here, the implementation details pertaining to both architectures are explained. For comparison reasons the two models have been made as similar as possible. Obviously, given the drastic differences conceptually between the two architectures this was only possible to a certain degree. A list of similarities follows.

- Both models were trained from scratch. No language models were used. Both models received the same examples and the same labels.
- Roughly speaking, the more trainable parameters a model has, the larger the capacity for learning³⁰. Therefore, the parameter count was approximately matched for the two models. The BiLSTM-based model had 410,812 parameters. The Transformer-based model had 375,528 parameters. Both of these numbers include the same Gated-Convolutional Visual Frontend which alone features 177,768 parameters. The frontend is trained as part of the end-to-end process of the whole. The actual values of the weights per model therefore differ substantially.
- Both models had a dropout probability of 10% for their Dropout layers. This served as regularization method, see Section 2.1 for more details.
- During training both models had the same learning rate and learning rate schedule. The learning rate started at 0.001, and decayed every 4 epochs according to $0.001 \cdot 0.98^s$ where s is the current learning rate slate that changes every 4 epochs. For instance, the first four epochs will have a learning rate of 0.001, the next four a learning rate of $0.001 \cdot 0.98^1 = 0.00098$, the next four $0.001 \cdot 0.98^2 = 0.0009604$, et cetera.

3.7.1 Training versus Inference

As previously noted in Section 3.5, the BiLSTM-based model structure does not change when switching from training to inference. In fact, during training an inference step is performed followed by a training step. During regular inference, the training step is simply omitted, i.e. no adjusting of the model takes place. That is not the case for the Transformer-based model. During training the example line-strips are fed to the encoder input, and the corresponding labels are fed to the input of the decoder. The output of the encoder and decoder are then subject to the loss function. Note how therefore the labels are not only used to compute the loss but also serve as input.

During inference, obviously no label is provided. Thus, the input of the decoder needs to be computed first, token by token. The start-of-line token is used to signal to the decoder that a new classification will be made. It is then combined with the encoded line-strip image from the encoder. The model then predicts the next token. This token is combined with the start-of-line token, and those two then serve as input for the decoder. This repeats until the maximum token count, 100, is reached, and the next image can be translated.

This also means that the model needs to be trained in such a way that the next token should be predicted. Thus, during training the label input to the decoders are shifted to the right by one token. Furthermore, the use of the mask on the labels prevents the model take into account the *future* of the label — only the past should matter when predicting a label.

3.7.2 Loss Calculation and Post-processing

The output of both models is a three-dimensional tensor of shape (42, 100, 82). The first dimension denotes the mini batch size. The second one is the maximum length of predicted tokens. The third gives the likelihood of each token. That means that per line-strip, the output is defined as a two-dimensional hypothesis grid: at each position a probability distribution of the potential tokens is given.

³⁰ This also increases the risk of over-fitting.

The actual classification is computed by simply taking the *best path* of the hypothesis grid. This means, at every location along the 100 possible tokens, take the one that is most likely. This compresses the tensor from $(100, 82)$ to (100) , i.e. 100 tokens.

In a process resembling the reverse of the one outlined in Section 3.3, the tokens are converted into ASCII text. First, the padding is removed, then, the numbers are translated into characters using the same previously defined dictionary. Lastly, the special characters are removed, except for the `|` character as it was also used in the original dataset.

The two loss functions used in the following experiments are: Label Smoothing (LS) Loss and CTC Loss. They both take this hypothesis grid to compute the loss. An explanation of CTC loss can be found in Section 2.3.3; the implementation is provided by Paszke et al. [44], a technical description is omitted here. An explanation of the implementation of LS loss follows. It was based on [47] and [48], though it might vary in its details.

Label Smoothing loss is a combination of two parts: Cross Entropy Loss (CE) as given in Section 2.1 in Equation 1 and an additional smoothed term. This term is given by the average of the sum of likelihoods in y multiplied by the smoothing factor s . In Equation 12 one can see the details. Note that y denotes the output from the model, and l the label. The constant G is the grid dimension, i.e. 100, since there are 100 potential tokens to predict. The constant T is the number of tokens that could be predicted, i.e. 82. The term y_{gt} therefore iterates over all position in the tensor per given example in the mini batch. Throughout all experiments in this thesis s was set to 0.1.

$$\text{LS} = (1 - s) \text{CE}(l, y) + s \cdot \frac{1}{G} \sum_{g=0}^G \sum_{t=0}^T y_{gt} \quad (12)$$

Label Smoothing results in a loss that is less punishing of mistakes. This is achieved by slightly de-emphasizing the CE loss, and instead adding the mean sum of all likelihoods. This means that if the output would become confident, i.e. less likely for all tokens except one per grid position, the additional term would inhibit a strong rewarding of the training algorithm. This is useful here, since some handwritten characters are not always clear-cut classification cases. For instance, the lowercase letter *d* has an *o*-shape within it. If the model predicts an *o*, punishing it too harsh seems inadequate as it is not too far off. Label Smoothing thus poses a Regularization method, see 2.1, by toning down the Cross Entropy Loss.

3.8 Evaluation

For every output, the loss is calculated. Then, the hypothesis grid is decoded into ASCII text. Using that predicted text and the label, the Character Error Rate and the Word Error Rate are calculated. These three values serve as evaluation basis.

3.8.1 Error Rates

The Character Error Rate (CER) is based on the *Levenshtein distance*, or *edit distance*. This distance is used when needing to calculate the difference between two given strings of text. It is defined by the minimum number of edits which can be insertions, substitutions or deletions. For instance, let a label be `a|cat`, and let the model's classification be `a|ct`. The distance between those is 1, because inserting one `a` into the second string results in the first string. The resulting distance is then divided by the total amount of characters. In this example, there are 5 characters, and the distance is 1, hence the CER is $1/5 = 0.2$. Thus, this can be interpreted as: 20% of the classification is erroneous.

The Word Error Rate (WER) follows the same principle but instead of using characters it is calculated at the word level. For instance, let the label be `a|cat|in|a|tree` and let the predicted text be `a|cat|n|tree`. There are 5 words in the label, but only 4 words in the classification. Also, one of the words (`n`) is incorrect. Thus, the edit distance at the word level is 2. Similarly to before, the length of the label is 5 (5 words), so the WER is $2/5 = 0.4$. Or, interpreted: 40% of the classification is erroneous³¹.

3.8.2 Data aggregation and Plotting

For each classification, the loss, the CER, and the WER, are calculated. These three values are then aggregated per mini batch and per epoch. The mean is used for aggregation. Note that k -fold validation with $k = 5$ was used, thus, the mean CER, the mean WER, and the mean loss were recorded for each fold, for each epoch, for each mini batch.

Across folds and across batches, the means of the mean CER, the mean WER, and the mean loss were calculated. To also measure the spread, the minimum and maximum values of each aggregation were recorded. Thus, per epoch, a mean CER, a mean WER, a mean loss, and their respective minima and maxima were recorded.

In the experimental process there are three phases. First, the model is trained epoch by epoch using a large portion of the dataset. Second, the model is evaluated after each epoch with a small subset of the dataset that was not in the training set³². Third, for the last 10 epochs of training, the model is tested on a larger set of examples, which it also has never seen before. From those last 10 epochs, the best performing result — according to the lowest mean — is recorded. Thus, it is important to note that the test results are the most relevant when evaluating the model's performance — it is done on a never-seen-before large dataset, which mimic real-world use. The evaluation data is useful to visualize the progress during training. Thus, in the upcoming results, the evaluation plots and the test results are depicted.

The plots for the training and the evaluation are line plots showing their progress over time. The shaded area shows the spread, from minimum to maximum. The plots showing the test results are bar plots as they only need to depict single values per run. These plots are used in the following experimental sessions.

³¹ The WER is more punishing, because a single error in a word renders the whole word erroneous. One can therefore expect the WER to be always at least as high as the CER.

³² The dataset split details including set sizes is provided in Section 3.1.1

4 Experiment I: RNN vs. Transformer

This experiment attempts to answer **RQ1** as defined in Section 1.1: In comparison to RNN-based models, are Transformer-based models better equipped to both recognize handwritten text and to exploit the available contextual information maximally?

4.1 Setup

To answer the question, two systems were built according to the specifications in Section 3. The question demands an experimental setup in which the only variable is the type of model: RNN or Transformer. All other variables should be equalized as much as possible. Most of that is already handled in the methodology, however the choice of loss function is still open. Important to note here: the Transformer-based architecture in Figure 18 includes layers for the Encoder Output. However, this output is utilized in later experiments only since they are only relevant for the inclusion of an encoder loss function. Therefore, layers right before that are not part of this current experiment; they are simply omitted.

BiLSTMs are commonly trained with CTC loss, while Transformers are commonly trained with Label Smoothing (LS) loss. One of those loss functions, however, might just be better suited for an HTR task. Thus, in principle, four experimental runs should be conducted following a 2x2 experimental design: 2 levels for the loss, 2 levels for the architecture. This enables controlling for the effect of the type of loss function on the architecture as well.

This setup quickly turns untenable, however. The RNN-based system coupled with LS loss learns at a pace many times slower than with CTC loss. There is a reason why the CTC framework is considered a leap forward. In contrast, the Transformer-based system coupled with CTC loss does not generalize at all; the way the Transformer needs to be trained³³ does not suit the alignment distribution calculation of CTC loss. Therefore, only the results of the conditions *RNN+CTC* and *Transformer+LS* are examined.

4.2 Results

During training, the evaluation curves were recorded. The results can be seen in Figure 23. Both curves decline along increasing epochs. This shows the learning process. They start off with a high error rate, and then they start to generalize. Up until approximately epoch 100 the mean error rates of both models are similar, then they start to diverge. In this period, they also both show rapid improvements as the lines both quickly fall from a height of over 0.7 to approximately 0.25. Over the next 200 epochs, the *Transformer+LS* line finds a lower low than the *RNN+CTC* line.

The variability of the *RNN+CTC* is lower than the other, however. The line itself also seems to become flatter quicker. Up until epoch 200, both systems show high spikes upwards in their shaded areas³⁴. They also correspondingly push the mean upwards. These evaluation curves are made using one mini batch of examples only (42 examples), thus it is expected that a singular value in the set can skew the results. Towards the end, the spikes subside. The shaded areas of the lines overlap the entire run.

The Word Error Rate (WER) shows similar patterns. They are depicted in the right plot in Figure 23. There are some immediately visible differences, however. First, the lines do not overlap in the visible part of the chart. They rapidly separate. The spikes are also visible here and correspond to the spikes in the CER plot — albeit, here more pronounced. The *Transformer+LC* lines finds a lower low than the *RNN+CTC* line. In fact, the difference between the lines seems amplified.

Regarding variability, the *Transformer+LS* data still show a larger spread between the minima and maxima overall. However, the difference in mean WER is large enough so that after approximately epoch

³³ The Transformer is trained with a shifted decoder input signal, so that at inference it can predict one token at a time whilst only looking at the past.

³⁴ Note that no smoothing was applied to the curves. This decision was made to not hide qualitative difference between models and their training.

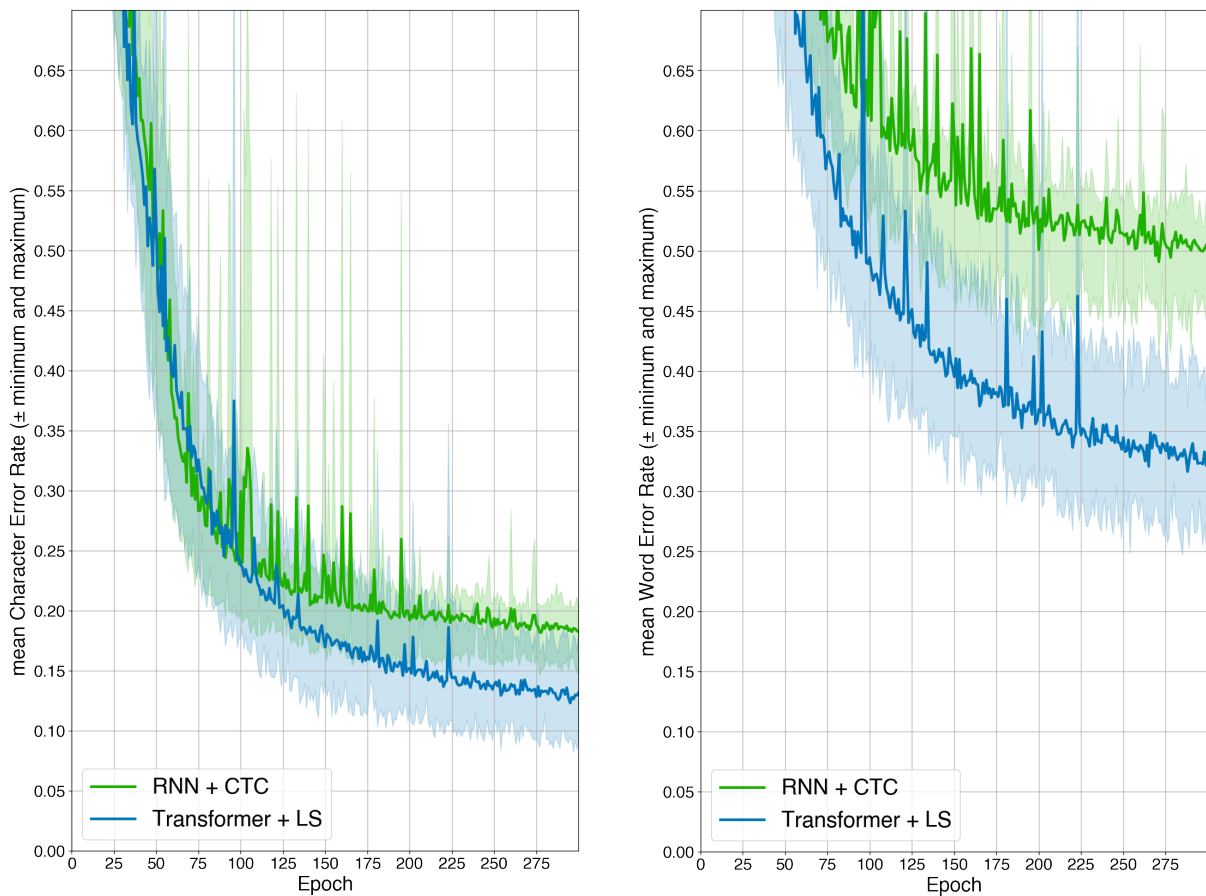


Figure 23: The **evaluation** results of Experiment I: the Character Error Rates are on the left; the Word Error Rates are on the right. The lines represent the mean error rate of the evaluation dataset per epoch. The shaded area shows the range defined by minima and maxima of the same data.

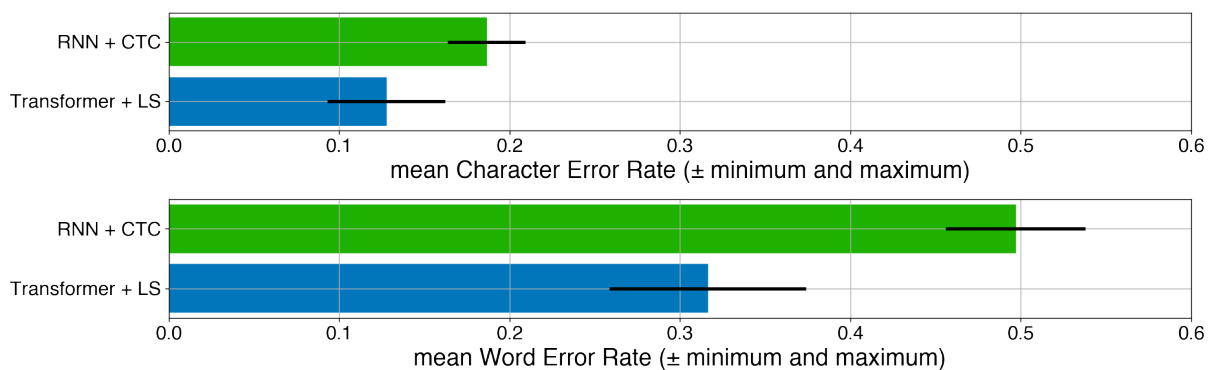


Figure 24: The **test** results of Experiment I: the Character Error Rates are on top; the Word Error Rates are at the bottom. Each bar represents the mean across all mean error rates of each mini batch in the test dataset. The error bars show the range defined by minima and maxima of the same data. The test performance was evaluated for the last 10 epochs for each condition, the best of those 10 is depicted.

175, even the two shaded areas stop overlapping. Towards the end of the training, one can see a visible gap between the two shaded areas. Note that this is not visible in the CER plot.

The test results are depicted in Figure 24. They mirror the evaluation results of Figure 23. In the left graph, one can see that the mean CER of the *Transformer+LS* condition is lower than of the *RNN+CTC* one. This time however, the variability bars are also not overlapping. Still, the *RNN+CTC* bar shows less variability than the other one.

Finally, turning to the WER bars at the bottom of Figure 24, one can see that here too the results mimic the ones of the right plot in Figure 23. The *Transformer+CTC* mean WER is lower than the other one. The variability bars are not overlapping, and, it seems that the cleft between the two bars is larger than the one of the evaluation plot.

4.3 Discussion

From the results, it becomes immediately clear that the Transformer-based model achieves better values across the board. Especially, the test results show a clear difference between the two models. Given that the error bars are not even overlapping, the differences in mean can be called significant.

To counter this conclusion, a valid question to ask is whether the RNN-based model was not hampered by the restrictive design choices. One can see that the curve becomes flat after only being halfway through the training. This suggests that the RNN-based model simply had no more capacity to learn. This is certainly true. Increasing the number of BiLSTM layers, or node count, might result in better performance. However, the objective of this thesis is not to find the best performing model. It is to gather understanding of which architecture might be better suited for the task. Recall Section 3.7. The number of parameters is larger for the RNN-based model. Indeed, the performance of the RNN-based model can become better by increasing the number of parameters, but that also holds for the Transformer-based model. Thus, in terms of parameter count the comparison is fair, if not even slightly in favor of the RNN-based model.

Similarly, one can argue that it is the specific arrangement of the layers inside the RNN-based model that leads to worse performance — if those same layers would be rearranged one would see the performance increasing. While this might be true, during the course of the experiments many constellations of layers were tried. The choice was taken in accordance with those findings, and arrangements that are typically found in the literature for high-performing systems. Though indeed, there might be a specific constellation that results in better performance — it is impossible to test all combinations, such is the nature of a complex system. Although, a too specific constellation might classify as over-fitting if it is matched too well to the given dataset. The approach taken seems like a fair balance.

While the CER provides a measure of accuracy at the character-level, the WER is applied at the word-level. This means, in terms of a context hierarchy, the WER operates at a higher level. To achieve high performance here, a system needs to extract more contextual information compared to the character level. From the results it is clear that the WER numbers are indeed higher than the CER numbers. Logically, this makes sense as explained in Section 3.8.1 — one character mistake in a word renders the entire word as erroneous. A higher WER as compared to CER is expected. However, if the WER is lower than expected this would mean the system is able to extract additional information in order to compensate for the mere character mistakes. Thus, it means it exploits information beyond the simple character level. Whilst the difference between the two CER mean values is fairly small, the difference between the two WER mean values is large. In order to demonstrate what that means, consider the following.

To gain understanding of how well the systems make use of the contextual information available, the CER can be used as basis to estimate the WER assuming all absence of deeper context learning. Assume words are simply sequences of characters where each character is independent of the others.³⁵ Given that, one can *naively* derive the WER from the CER by multiplying the expected probabilities. For instance,

³⁵It should be emphasized that this assumption certainly does not hold for *real* words. Humans derive a great deal of meaning from the context and use that to correct for mistakes. See Section 2.2.1 and Section 2.3.1 for a broad discussion on context.

a system with a CER of 0.1, is expected to have a WER of approximately $1 - (1 - 0.1)^4 = 0.344$ for a word 4 characters in length.³⁶ Rather than estimating 4 as the word length, the mean word length in the IAM dataset is used — 4.17 words. Equipped with this logic the results can be analyzed as follows.

The test mean CER of the RNN-based system is 0.1864. Thus, the expected mean WER is about

$$1 - (1 - 0.1864)^{4.17} = 0.5769.$$

The actual observed mean WER from the trained RNN-based system is 0.4968 — or, 13.9% lower³⁷. Thus, the RNN-based system is able to utilize the provided information beyond the character level as shown by the drop in observed WER as compared to expected WER by 13.9%. In contrast, the test mean CER of the Transformer-based system is 0.1276, so, the expected WER is approximately

$$1 - (1 - 0.1276)^{4.17} = 0.4340.$$

The actual observed WER is however 0.3162 — or, a 27.1% drop³⁸. Roughly speaking, these drops can be used as a measure of how well the system learns the context of the words. At face value it becomes immediately clear that the difference between expected WER and observed WER is significantly higher, almost by a factor of 2, for the Transformer-based system when comparing it to the RNN-based system.

The quality of the results, as a measure of variability, appears to be better in case of the RNN-based system. Both the shaded areas in the evaluation plots in Figure 23 and the error bars in Figure 24 appear to be smaller. There is more spikiness in the curves up until epoch 200 for the RNN-based results, however. Disregarding that however, one can conclude that in this experiment, the RNN-based model produced more consistent results compared to the Transformer-based one. It will therefore be part of the objective moving forward to focus on improving the consistency of the Transformer-based model's classifications.

In conclusion: The Transformer-based model appears to be better equipped for this HTR task. It produces lower mean error rates while featuring a similar setup and parameter count as the RNN counterpart. In addition, it exhibits an improved ability to exploit the available contextual information at the word-level when compared to the RNN-based model.

This conclusion also sheds light on the fundamental difference between the two models. RNNs model the data as a function of proximity. Data points closer together along the temporal axis are more impactful to one another. They certainly have the ability to extract patterns across longer time frames, especially in the case of LSTMs, but the fundamental proclivity to prioritize proximity might not be appropriate enough for tasks involving natural language. As outline in Section 2.2 proximity can be a powerful heuristic in language. For instance, a sentence is for the most part a collection of related concepts. However, it appears that careful and dynamic modelling of relations between words and concepts is more appropriate.

Another conceptual angle to this could be the following. RNNs model relations between tokens based on proximity — they can learn patterns over time. This pattern extraction is however the only mechanism they feature. In contrast, Transformers perform two actions that are mostly independent. First, they use the dot-product as a means to find similarities between word embeddings. Depending on the quality of the embeddings this poses a promising way to solve the similarity problem of text³⁹. This is then further utilized to disambiguate overloaded embeddings: Recall the example of the word `Draft` in Section 2.2.3. RNNs have no clear path to limit the scope of the embedding in relation to the surrounding context. Second, using this more specific embedding, the Self-Attention mechanism allows for dynamic modeling of the relations between words. It is important to note, that this might *also* include proximity, but it does not have to. Thus, Transformers appear to have a more appropriate range of tools at their disposal.

³⁶Note how error rate need to be converted into accuracy, i.e. accuracy = 1 - ER, and then back to the error rate.

³⁷Calculated as $1 - 0.4968/0.5769 = 0.139$

³⁸Calculated as $1 - 0.3162/0.4340 = 0.271$

³⁹I.e. how to numerically represent differences between concepts captured by text

Following the rationale outlined in Section 1, simply declaring Transformers to be superior to all previous models seems too shallow. It was shown they indeed perform better under the presented circumstances. However, previous systems' successes were not only due to RNNs but also other methods, in particular the Connectionist Temporal Classification (CTC) framework. Utilizing the alignment calculation appears to be a natural way forward, even if RNNs will fade in favor of Transformers. The following experiment will therefore focus on the incorporation of CTC into the current pipeline. Recall that simply using CTC loss for the Transformer-based system is not an option. Thus, an approach will be taken inspired by Karita et al. [49] who utilized the CTC framework for the encoder part of their architecture in a Speech Recognition task.

5 Experiment II: CTC loss at the encoder

This experiment attempts to answer **RQ2** as defined in Section 1.1: Can the CTC framework be leveraged in conjunction with a Transformer-based model to enhance performance and contextual information extraction?

5.1 Setup

This experiment is similar to the previous one, with two exceptions. First, the RNN-based model is left out, the focus is on only the Transformer-based one. Second, a composite loss function is created that combines the output of both encoder and decoder. Thus, the layers responsible for mapping the encoder layer outputs to the encoder output, see Figure 18, are included in this experiment.

This experiment serves to test whether the CTC framework can be incorporated into the Transformer-based model in this HTR task. As noted in Section 4, CTC loss cannot be used at the decoder output. Thus, that leaves three conditions for this experiment: using CTC loss on the encoder output in conjunction with LS loss at the decoder (*CTC+LS*), using LS loss on both outputs (*LS+LS*), and only using LS loss at the decoder which simply is the result of the previous experiment (*Baseline I*). Analyzing these three conditions should provide enough insight to see whether CTC has any influence.

The new loss L is calculated as follows. The two losses of the encoder output and the decoder output are simply added. See Equation 13. Note that L_{decoder} naturally refers to the output of the entire system. Thus, calculating the loss and using it during back-propagation *also* trains the encoder part. The addition of L_{encoder} merely emphasizes the training of the encoder, but does not fully encapsulate it.

$$L = L_{\text{encoder}} + L_{\text{decoder}} \quad (13)$$

5.2 Results

The evaluation results of the three conditions are in Figure 25. On the left, one can see the mean Character Error Rates (CER) over the epochs. On the right, one can see the mean Word Error Rates (WER).

In the left plot in Figure 25, all three lines decrease with increasing epoch count. This shows that all three models train over time. The *CTC+LS* condition is lower than the others during the entire training. The condition *LS+LS* starts off being lower than *Baseline I* but is overtaken by it at approximately epoch 100. The line of *Baseline I* then continues to decline and flatten, almost touching the blue *CTC+LS* one. Both red and blue lines also flatten quickly after about 100 epochs. These properties appear to be mirrored for the mean WER curves in Figure 25. The gaps are a bit more pronounced between the conditions.

Regarding variability, from the evaluation, no noteworthy characteristics are observed. The shaded areas depicting the spread between examples. They appear to be about equal in size. The high spikes subside for the conditions *CTC+LS* and *Baseline I* after about 100 epochs. However, condition *LS+LS* shows high spikes approximately up until 250 epochs. These spikes are mirror in the mean WER plot.

The test results for both mean CER and mean WER are shown in Figure 26. At the top, the mean CER values are depicted. The lowest values are from the condition *CTC+LS*, closely followed by the condition *Baseline I*. Lastly, condition *LS+LS* features the highest error rates. Like for the evaluation plots, the test WER plots mirror these results as well. Observe in the bottom plot how the values are proportional to the ones in the CER plot.

However, the variability of the condition *LS+LS* seems notably lower than the other two, in both CER and WER plots. The variability of condition *CTC+LS* also appears slightly smaller than condition *Baseline I* which features the largest spread.

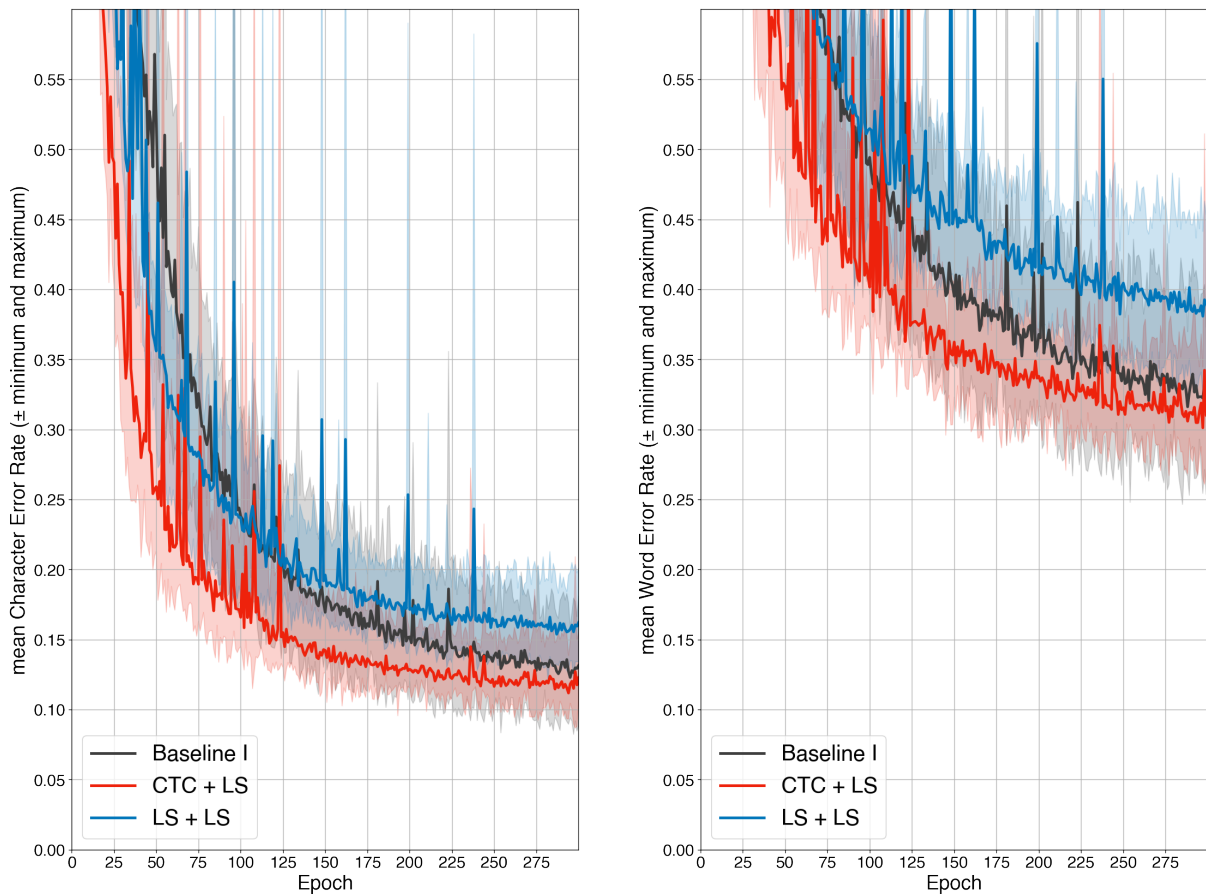


Figure 25: The **evaluation** results of Experiment II: the Character Error Rates are on the left; the Word Error Rates are on the right. The lines represent the mean error rate of the evaluation dataset per epoch. The shaded area shows the range defined by minima and maxima of the same data.

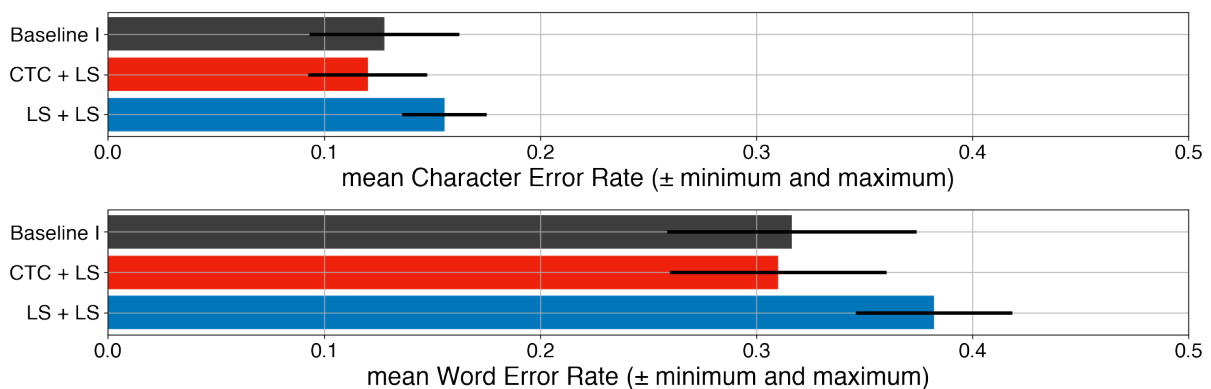


Figure 26: The **test** results of Experiment II: the Character Error Rates are on top; the Word Error Rates are at the bottom. Each bar represents the mean across all mean error rates of each mini batch in the test dataset. The error bars show the range defined by minima and maxima of the same data. The test performance was evaluated for the last 10 epochs for each condition, the best of those 10 is depicted.

5.3 Discussion

Following the results, it is immediately clear that the inclusion of the CTC framework has a positive effect. The values in the CTC condition are clearly lower than the other ones. This is visible both during training and during the final test. Also, during training the slope at which the CTC condition decrease in error rates is remarkably faster than the other two. This suggests that CTC does not just result in lower error rates, but it also speeds up the training process.

The condition $LS+LS$ shows that CTC actually provides an advantage that really stems from CTC itself. Due to the way CTC is added to the overall loss, one could make the argument that the improved results come from the additional training exposure — the encoder is simply trained twice, hence the rate of learning is implicitly adjusted. While this is true to an extent, the condition $LS+LS$ offers the control case: In this condition, the encoder is also trained twice, and it results in worse error rates, in fact, it shows a consistently worse result during training.

Although CTC adds value, the question is how relevant it is. Looking at the CER test results in Figure 26, it is not obvious that CTC provides a significant advantage. The mean value is slightly lower, and the variability too. However, consider the spikes during training. The line of the CTC condition appears to be smoother overall when compared to the other two, albeit less prominent in the baseline case. It is the combination between the slightly lower values, the pace at which it speeds up learning and smoother training curve that appears to make this condition superior to the others for this specific experiment.

Whether CTC also enables better contextual information extraction is more difficult to answer. The expected mean WER values fall in line with the observed ones. Although results are better, there seems to be no other way of judging if the model predicts whole words better than without the CTC framework.

In conclusion: The use of a composite loss function that incorporates CTC loss at the encoder of the Transformer-based model demonstrates lower error rates and smooth evaluation curves. Apart from those gains, it is however not obvious the extraction of contextual information is enhanced as well.

Using CTC's alignment calculation at the encoder side is beneficial to the system as a whole. This suggests that the alignment distribution helps this Transformer-based model similarly to the RNN-based one. The issue of aligning a sequence of tokens with the output of a neural network with a sequence as input seems to be not solved by the Transformer architecture either. This suggests, that although RNN-based systems in HTR tasks might become obsolete, the CTC stage still provides value since the alignment problem is otherwise not solved. However, this begs future research.

The coming experiments will focus on improving the results of the Transformer-based model using CTC at the encoder side. There are many options to go for. The focus will first be on weighing the losses, i.e. emphasizing one over the other. The reason is that before, in this current experiment the two losses were almost arbitrarily added together. This might result in suboptimal proportions during training. Introducing a weight to the composite loss will proportionally scale one down and one up, and vice versa. This allows for a scanning of the spectrum to see at which proportions the model performs best. Thus, the hope is that performance will see an improvement once again.

6 Experiment III: Weighted composite loss

This experiment attempts to answer **RQ3** as defined in Section 1.1: What can be done to refine the results further? And, how can the quality of the classifications be enhanced?

6.1 Setup

Following the promising results of Experiment II, further capacity for improvement will be tested. The previous loss calculation is augmented by a coefficient α that weighs the two individual losses proportionally. This effectively introduces a bias. With large values of α the decoder loss is emphasized while the encoder loss is de-emphasized; and, vice versa. This enables more tuning, and therefore more capacity for improvement. See Equation 14 for the new composite loss function.

$$L = (1 - \alpha) L_{\text{encoder}} + \alpha L_{\text{decoder}} \quad (14)$$

To test whether weighing of the losses impacts performance, a new experiment is devised. Once again, the setup of the system is similar to the ones before, with exception to the aforementioned loss function. The experimental conditions are given by a range of static α values. This shows whether the introduction of α has any importance to begin with, i.e. if the difference between the evaluation and test outcomes are noteworthy enough. The values of α to be tested are $[0.1, 0.25, 0.5, 0.75, 0.9]$. Those values represent a stepwise shift from emphasizing the encoder and de-emphasized the decoder, to the other way around. There is no specific reason behind choosing these exact values other than that they represent the spectrum sufficiently⁴⁰. Technically, the run for $\alpha = 0.5$ is unnecessary since it mimics Experiment II by cutting the losses of both encoder and decoder equally. For completeness's sake, it is run again and included here.

6.2 Results

Figure 27 depicts both mean CER and mean WER during training. On the left, the CER is depicted. From looking at the lines, one can see that with increasing values of α , the mean CER curves are shifted downwards. At least that holds for $\alpha \leq 0.75$. The last two runs for $\alpha = 0.75$ and $\alpha = 0.9$ seem to overlap after about 100 epochs.

On the right, the WER is depicted. Note, that here it appears that the line for $\alpha = 0.9$ is slightly lower than the one for $\alpha = 0.75$. Otherwise, the chart follows the trends of the CER plot.

The variability is higher for small values of α , and lower for high values of α . However, there are more spikes the higher α , and consequently the lower the variability — with the exception for $\alpha = 0.9$ which is smoother than the previous $\alpha = 0.75$. This is also mirrored in the mean WER plot to the right of Figure 27. Here, the spikes are even more pronounced.

The test plots are in Figure 28. Notice how the picture does not change much. The error rates, both for CER and WER, are decreasing with higher levels of α until they seem to plateau after $\alpha = 0.75$. The variability also decreases with higher numbers of α until $\alpha = 0.75$. In terms of mean test CER, $\alpha = 0.75$ has both the lowest number, and the smallest spread. However, the results for $\alpha = 0.9$ show a slightly higher CER, but also slightly lower WER. Also, it has lower lows and higher highs compared to $\alpha = 0.75$.

6.3 Discussion

From the results, it appears obvious that the model with $\alpha = [0.1, 0.25, 0.5]$ are undesirable. Their error rates are too high, and their spread is too wide. The choice within the selected values of α therefore falls on either $\alpha = 0.75$ or $\alpha = 0.9$. This is not an easy decision, however.

⁴⁰ Extreme values $\alpha = 0$ and $\alpha = 1$ were omitted, since, in the first case, it would simply reproduce Experiment I (no encoder training), and, in the second case, it would simply result in no (usable) output (no decoder training).

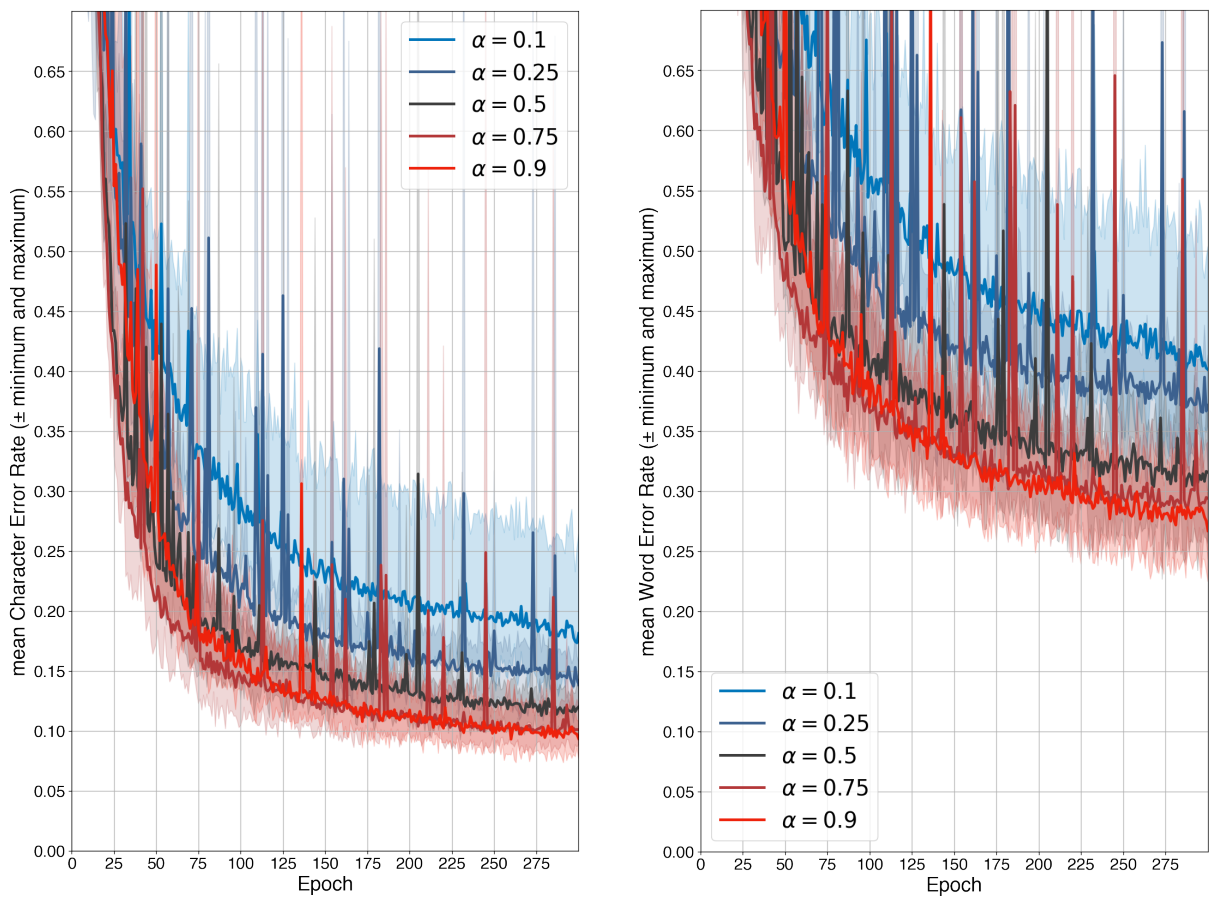


Figure 27: The **evaluation** results of Experiment III: the Character Error Rates are on the left; the Word Error Rates are on the right. The lines represent the mean error rate of the evaluation dataset per epoch. The shaded area shows the range defined by minima and maxima of the same data.

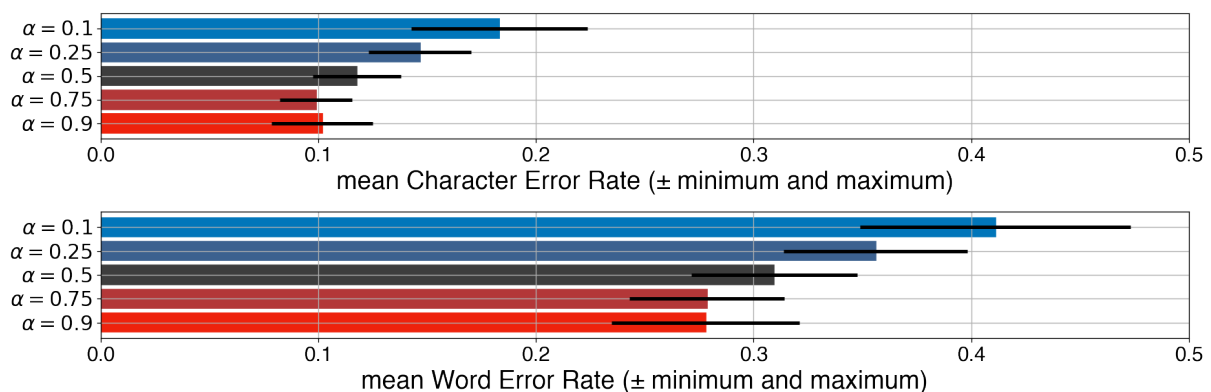


Figure 28: The **test** results of Experiment III: the Character Error Rates are on top; the Word Error Rates are at the bottom. Each bar represents the mean across all mean error rates of each mini batch in the test dataset. The error bars show the range defined by minima and maxima of the same data. The test performance was evaluated for the last 10 epochs for each condition, the best of those 10 is depicted.

At first glance, one can see that $\alpha = 0.75$ appears to be the most consistent run with the lowest mean CER. In contrast, the model with $\alpha = 0.9$ has a wider spread, and a slightly higher mean CER. Upon further examination however, one will notice that although the test spread is higher, during training there are fewer spikes compared to $\alpha = 0.75$. Also, the CER is higher indeed, but the WER is lower. This suggests that $\alpha = 0.9$ is slightly better able to extract contextual information than the $\alpha = 0.75$ model. However, when observing the curve during training, one can also see that the $\alpha = 0.75$ model seems to learn quicker until about epoch 100 at which point the curve seems to merge with the $\alpha = 0.9$ model. Thus, it appears there is no clear winner.

In conclusion: In an HTR task the Transformer-based model achieves better results for higher α values, specifically for $\alpha \geq 0.75$. If one needs to prioritize consistency of the classifications $\alpha = 0.75$ should be chosen. If one wants to optimize for a smooth training curve and a lower WER, $\alpha = 0.9$ appears to be more appropriate.

Given the unclear choice between $\alpha = 0.75$ and $\alpha = 0.9$ the question arises if there is a way to combine the best of both. A simple, yet effective, way to include multiple stages for a hyperparameter like α is to include it as a dynamic variable instead of a static one. Clearly, CTC has a positive effect overall on the Transformer-based model. However, it might be more effective during early stages of the training phase due to first rapidly constructing internal representations of the alignments in the encoder. The later stages during training could then be shifting more towards the decoder side to fine tune the results. Roughly speaking, this could mimic first focusing on properly extracting high quality features and learning their possible alignments, and then using those learned patterns to fine tune the remaining system.

Starting with a high value for α and gradually stepping it down towards a low value might result in better, more consistent values. The results show however that there is a general tendency towards higher values of α , thus still operating the majority of the training time under an $\alpha \geq 0.75$ schedule seems appropriate. This hypothesis will be tested in a last experiment.

7 Experiment IV: Curriculum Weighing

This experiment attempts to continue to provide answers to **RQ3** as defined in Section 1.1: What can be done to refine the results further? And, how can the quality of the classifications be enhanced?

7.1 Setup

To further investigate how to improve performance and quality of the Transformer-based model, a hypothesis was posed in Section 6.3: *Starting with a high value for α and gradually stepping it down towards a low value results in better performance, more consistent values.* This hypothesis is tested here.

The value of α can be regulated by a curriculum: α decreases by 0.1 every specific number of epochs — the step size. This effectively converts α into a dynamic value. The curriculum gradually shifts values of α over the course of the training period. Note that α always starts off at 1.0 and then decreases over time eventually bottoming at 0.1. This results in a warm-up phase for the encoder where the decoder loss is fully disabled, then, gradually, the balance shifts towards the decoder loss. Depending on the step size, α can decline quicker or slower.

This experiment tests three conditions to see how a curriculum affects the results. The conditions are different values for the step size. The values chosen are 5, 10, and 15. The previous experiment showed that there should be a tendency towards lower values of α , because the overall training emphasis should be on the decoder side. Thus, the majority of the training time, α will be set to 0.1. The curriculum for each condition can be seen in Figure 29. Note how, depending on the step size, the CTC-warm-up phase is regulated. For a step size of 10, the warm-up phase lasts for 90 epochs, and the refinement phase ($\alpha = 0.1$) lasts for 210 epochs.

7.2 Results

The evaluation results are depicted in Figure 30. The mean CER values are on the left. All three conditions exhibit roughly similar curves, especially towards the end of the epochs. However, the condition 10 results in a line that is lower than the others, while 5 and 15 seems to overlap after about 150 epochs. Condition 5 initially declines faster than the others, but is overtaken by condition 10 after about 90 epochs.

The mean WER curves on the right of Figure 30 show similar results. Condition 10 has the lowest mean curve, followed by condition 15 and 5. In contrast to the CER plot, condition 5 and 15 are not overlapping — 15 gets closer to 10 instead.

Condition 10 shows the most consistent results, its variability is lowest amongst all three. Both condition 10 and 5 exhibit a smooth learning curve, almost no spikes are visible. In contrast, condition 15 shows many more spikes during the first half, then spikes subside. It is to note that condition 15 shows the lowest lows in the second half by a fairly large margin which is especially visible in the WER plot.

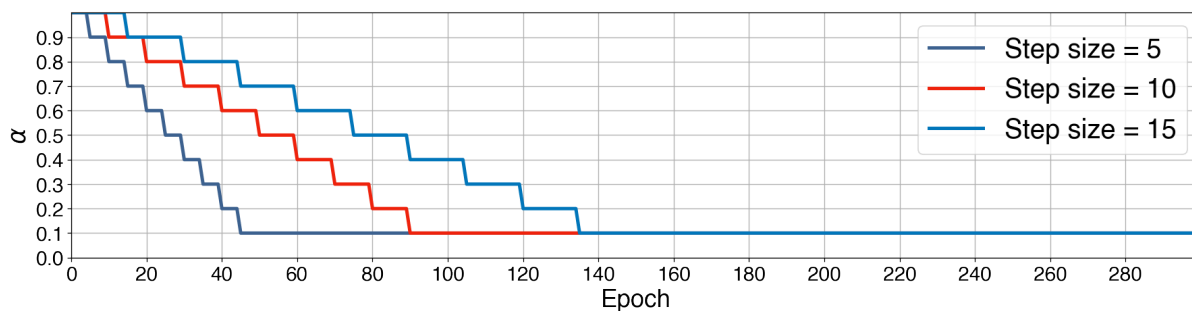


Figure 29: A graph showing the curriculum for α depending on the chosen step size.

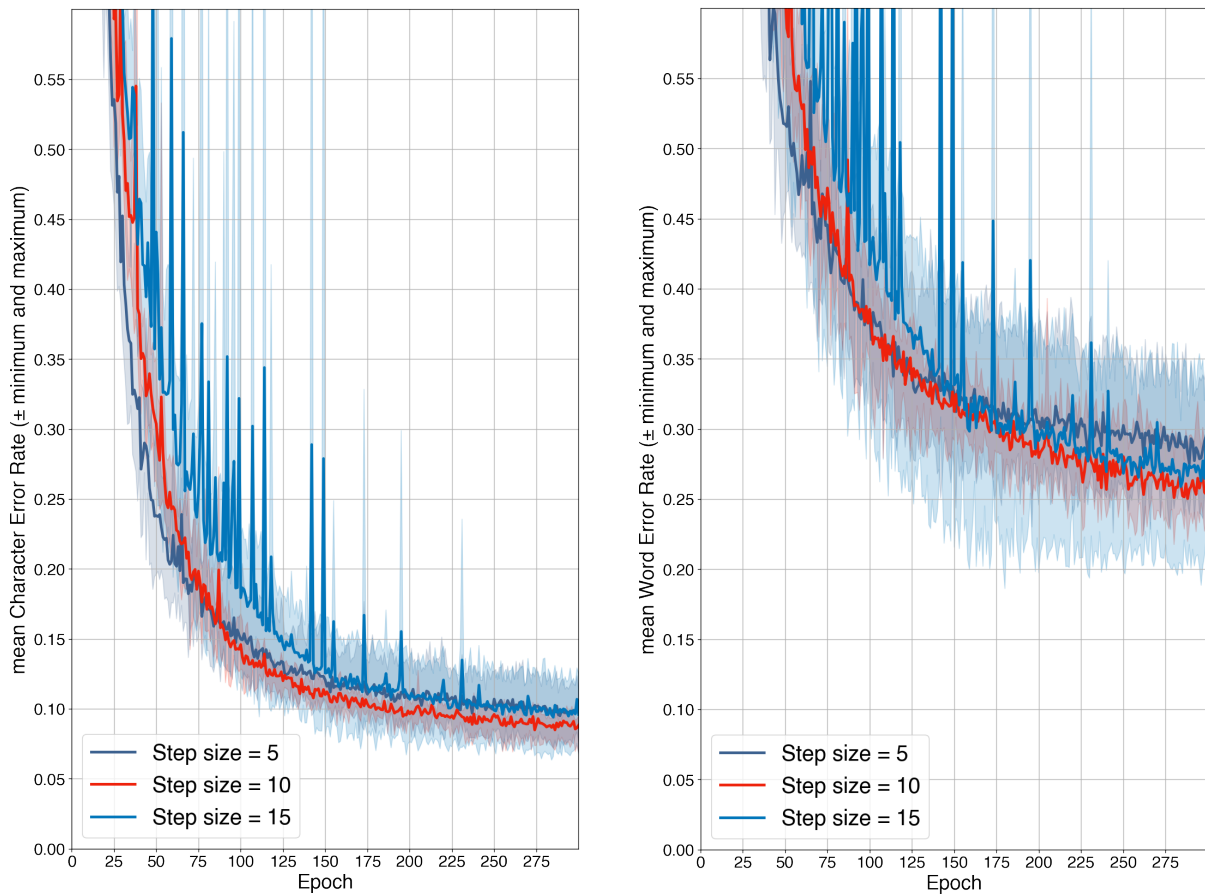


Figure 30: The **evaluation** results of Experiment IV: the Character Error Rates are on the left; the Word Error Rates are on the right. The lines represent the mean error rate of the evaluation dataset per epoch. The shaded area shows the range defined by minima and maxima of the same data.

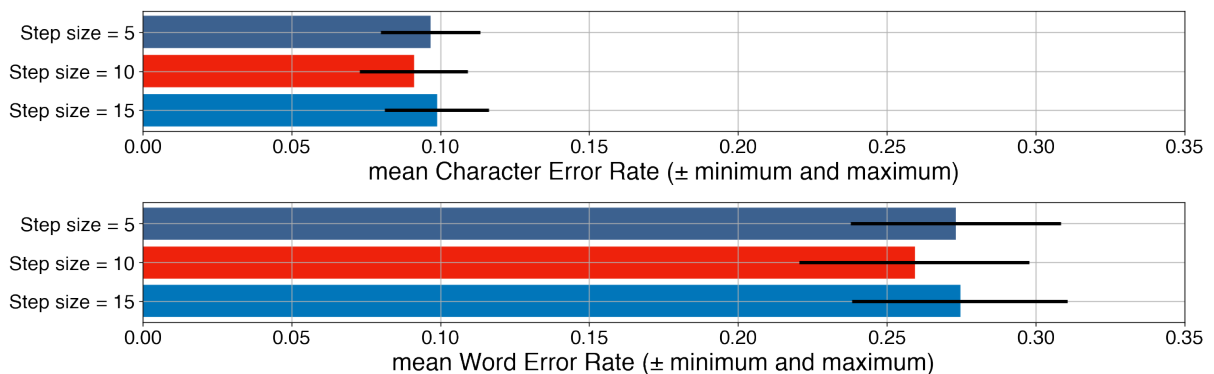


Figure 31: The **test** results of Experiment IV: the Character Error Rates are on top; the Word Error Rates are at the bottom. Each bar represents the mean across all mean error rates of each mini batch in the test dataset. The error bars show the range defined by minima and maxima of the same data. The test performance was evaluated for the last 10 epochs for each condition, the best of those 10 is depicted.

The test results, see Figure 31, are straightforward. Condition 10 shows the lowest mean for both CER and WER. Conditions 5 and 15 appear roughly similar, with condition 5 showing a slightly lower CER. All three conditions across both CER and WER show roughly the same variability. The differences that were visible during training do not appear here.

7.3 Discussion

From the results it is clear that introducing a curriculum for α affects performance and quality of the results during training. Especially visible for a step size of 10, the training curve is remarkably smooth and shows good test performance. Thus, depending on the step size, the hypothesis appears to hold some truth: Starting with a high value for α and gradually stepping it down towards a low value results in lower error rates and more consistent values compared to the previous experiments.

The curriculum for α also appears to be a good solution for the spikes. Warming up the encoder, i.e. emphasizing the CTC loss initially results in less variability. This could be due to forcing the model to first solve the alignment problem across the input images, and only later refining the decoding stage.

In conclusion: In an HTR task the Transformer-based model shows better, more consistent results if α is dynamically regulated by a curriculum that slowly shifts emphasis from the CTC stage at the encoder to the LS stage at the decoder. Specifically, a step size of 10 showed good and smooth results both during training and testing. Thus, it appears that dynamically training a Transformer-based HTR system using CTC loss reduces variability while also showing lower error rates compared to non-dynamic training.

The question arises, why a more dynamic training approach such as the current one would result in better performance as well as smoother training curves. It seems that starting with a slightly pre-trained encoder benefits performance, as compared to simply training it as part of the whole. The only real disadvantage appears to be a slower training progression during the first 100 epochs, roughly speaking. Still, given that the number of epochs was held constant across all experiments, it seems remarkable that although the decoder loss was inhibited compared to previous runs, it still eventually decreases error rates. An explanation might be that the Transformer really benefits from high-quality encoded input. Thus, focusing all training efforts to first extract information out of the pixel space, and also storing some initial alignments between extracted letters, appears beneficial. Daringly, one could draw an analogy to human language acquisition where children first learn syllables, then practice those before forming words.

8 General Discussion

Here, the results are summarized and compared across experiments. Then, the conclusions of those are repeated. Finally, a discussion is provided including potential flaws and future research suggestions.

8.1 Summary & Conclusions

The goal of this thesis was to examine how well Transformer-based models perform in an HTR task, compared to traditional RNN-based models. The Transformer poses a potentially potent new technology, hence experimenting with it is necessary. For that purpose, two systems, each based on the respective architecture, were developed and then tested in a series of experiments. First, the Transformer-based model was compared to a comparable RNN-based version. Later experiments then utilized the well-established CTC framework to further optimize recognition error rates. The focus was on how well the models could exploit the lexical contextual information, which is deemed a key factor to more intelligent behavior. Therefore, a contribution to the field of Artificial Intelligence is provided as well.

The best results of each of the four experiments of the Transformer-based model are depicted in Figure 32 and Figure 33. They show the progress made across experiments. In the evaluation plots, one can see that each iteration notably improved recognition. Especially the last experiment's WER results seem to make a push downwards, hinting at better contextual information extraction. The progress in quality of the results, as defined by smoothness of the curves or variability during training, also progresses remarkably across the experiments — Experiment IV results show a smooth evaluation curve only lacking in initial speed compared to the others. The steady decline of the error rates is also clearly visible in the test plots. The major findings relating to the research questions (see Section 1) follows:

RQ1 In comparison to RNN-based models, are Transformer-based models better equipped to both recognize handwritten text and to exploit the available contextual information maximally?

The Transformer-based model appears to be better equipped for this HTR task. It produces lower mean error rates under similar conditions as the RNN counterpart. In addition, it exhibits better ability to exploit the available contextual information at the word-level, compared to the other.

RQ2 Can the CTC framework be leveraged in conjunction with a Transformer-based model to enhance performance and contextual information extraction?

The use of a composite loss function that incorporates CTC loss at the encoder of the Transformer-based model demonstrates lower error rates and smooth evaluation curves. Apart from those gains, it is however not obvious the extraction of contextual information is enhanced as well.

RQ3 What can be done to refine the results further? And, how can the quality of the classifications be enhanced?

Introducing weighing to the composite loss function shows improvements if the weight leans towards the decoder loss. Results get better and more consistent if this weight is dynamically regulated such that it slowly shifts emphasis from the encoder to the decoder during training.

Within the perimeter of the experiments, it indeed appears that a Transformer-based models offer potential. The experiments have shown that such models can achieve better results than a comparable RNN-based one. They have also shown that established methods like the CTC framework can still be salvaged, and even lead to performance increases. Furthermore, they revealed that there are more options for further improvements. Thus, it appears that the Transformer's Self-Attention mechanism might indeed be potentially superior to a recurrence-based approach when it comes to exploiting available lexical contextual information. Obviously, the results presented here only hold for the given setup. Potential issues with that are discussed in the next section. However, if the results also generalize to more systems, the field of HTR could see a shift towards Transformers over RNNs in the near future.

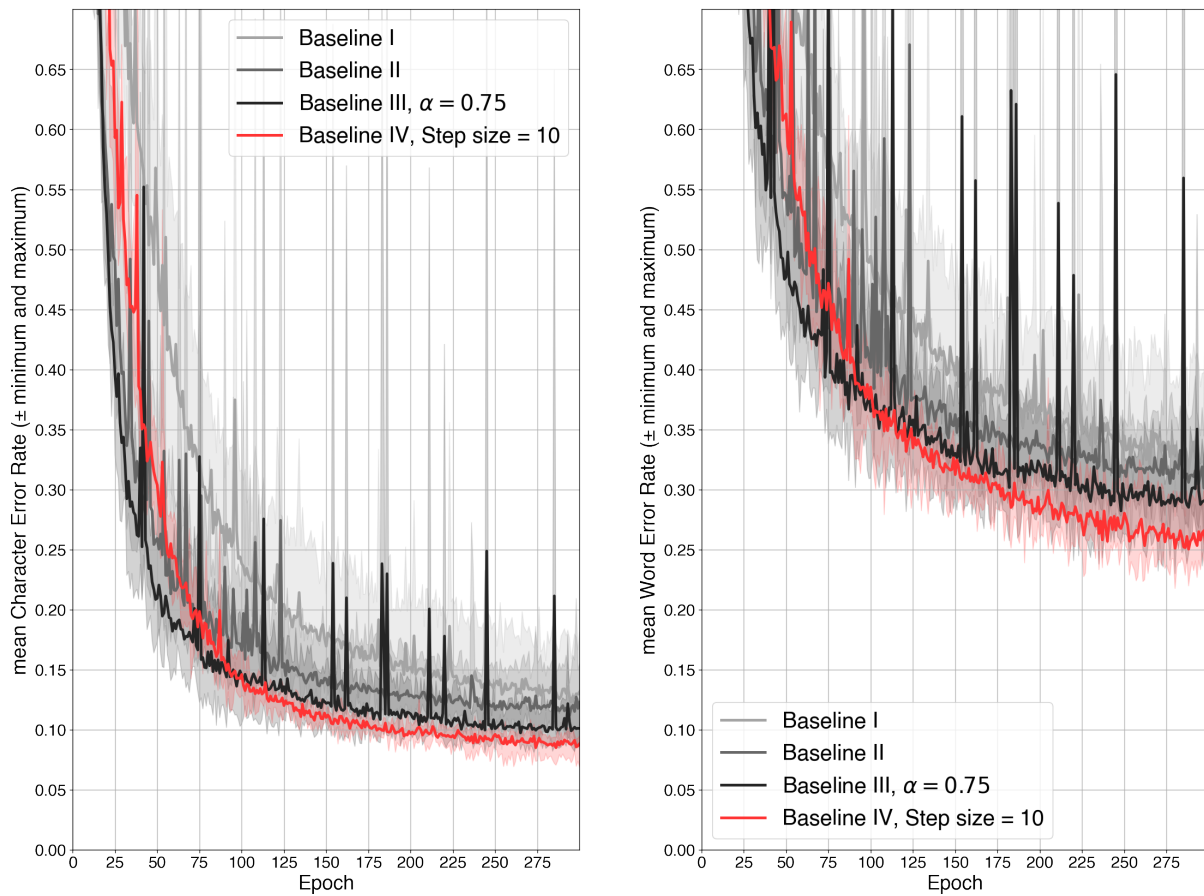


Figure 32: The main **evaluation** results of all experiments: the Character Error Rates are on the left; the Word Error Rates are on the right. The lines represent the mean error rate of the evaluation dataset per epoch. The shaded area shows the range defined by minima and maxima of the same data.

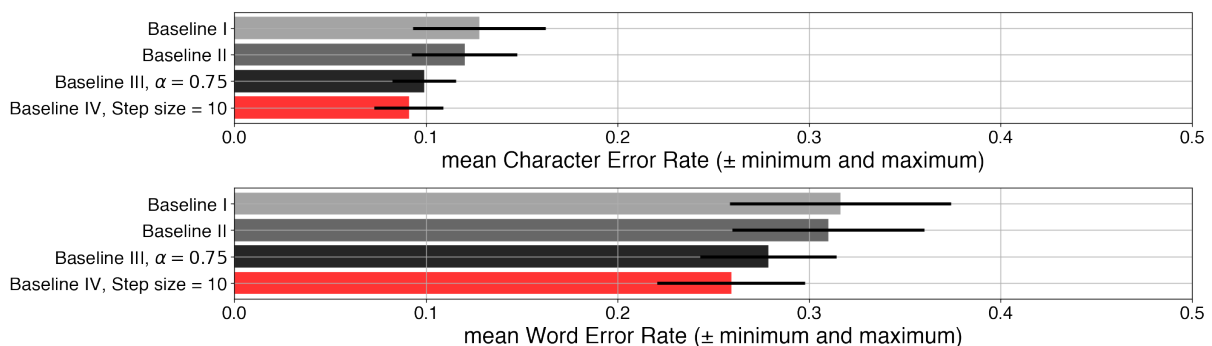


Figure 33: The main **test** results of all experiments: the Character Error Rates are on top; the Word Error Rates are at the bottom. Each bar represents the mean across all mean error rates of each mini batch in the test dataset. The error bars show the range defined by minima and maxima of the same data. The test performance was evaluated for the last 10 epochs for each condition, the best of those 10 is depicted.

8.2 General Discussion

The presented setup came with shortcomings. First is the potential for over-fitting by the experimental progression itself. Over the course of the experiments, different variations have been tried that subsequently improved the results. However, given the specifics of the IAM dataset, those improvements might not generalize to other datasets as well. Granted, the test methodology in conjunction with k -fold validation does indicate that the models are able to generalize fairly well. Still, the dataset used is small compared to what usually is used — a collection of different databases, such as IAM. To mitigate over-fitting, three Regularization methods have been employed: lowering the models' capacity via reducing parameter count, dropout during training, and dataset augmentation. Given those, and the promising test results, there seems to be enough potential to go forward with larger models to see if the results continue to hold.

Second, the last two experiments introduced a new hyperparameter α and a curriculum for it. That increase in human involvement is not exactly in line with the general goal of Artificial Intelligence — creating intelligent systems. The variables governing the curriculum in conjunction with α itself needed a lot of testing and fine-tuning in order to produce better results. Even though they eventually did, such measures should better be part of the learning process itself for an intelligent system. This can be mitigated by reducing the number of hyperparameters necessary to constitute the curriculum, for instance, an exponential curve instead of a step function could reduce the number of variables.

Third, when comparing the results to the state-of-the-art one will quickly find lower error rates in other studies. However, the setup presented here is also remarkably special in its own rights: the number of parameters is *substantially* lower than other state-of-the-art models. For instance, the Transformer-based model by Kang et al. [25] which was used as inspiration for our model comes with 100 million weights whereas our model has only roughly 400,000 weights. Also, no explicit heuristics were used whereas many other systems make extensive use of shortcuts, for instance language models with predefined dictionaries to restrict classification outcomes to only existing words. Thus, although error rates presented here are not competitive with the state-of-the-art, there is a lot of room to grow — increasing capacity, using more training data, and utilizing linguistic heuristics. In Section 1, an argument was made in favor of balance between low error rates and true intelligent behavior. We believe the findings regarding the contextual extraction benefits of Transformers over RNNs is worth the higher error rates in this setup.

This thesis offers a basis for many further studies. It was shown that Transformer-based HTR systems are certainly potent enough to seriously consider for HTR applications. Especially in combination with the CTC framework, the models shine. Here, a few promising options for further research are proposed.

First, the step function is a rather crude way of shifting the balance within the composite loss. Instead, different methods could be employed such as smooth exponential curves that gradually shift α . In contrast, one could also experiment with even less linear curriculums that for instance, emphasize the encoder warm-up phase longer, before rapidly shifting to the decoder. If the warm-up of the encoder is imperative, the intermediate values of α might be superfluous, and simply pre-training the encoder might also work. How exactly the CTC stage can be leveraged at the encoder is therefore worth investigating.

The Self-Attention mechanisms inside the Transformer-based model use the dot-product to calculate attention scores. There are however alternatives to derive those. The dot-product has been shown to be one of the best [13], but this might not hold in an HTR context. Thus, further tuning the inner workings of the attention mechanisms might result in better results. From the above critique, one can also find other suggestions for improvement, and future research. How do the presented models fare with more training data, especially if that data introduces substantially different examples? Given that language models are a prominent tool in the literature, how would an inclusion of those affect the performance? Also, how could language models with utilized during the CTC stage which is now at the encoder instead of the decoder?

In this thesis, a contribution to the field of HTR was made in the form of merging novel technologies with the well-established. The focus on the balance between performance and intelligence enabled observing direct evidence of contextual extraction advantages using attention-based methods over the former recurrence-based state-of-the-art — *something to pay attention to*.

Bibliography

- [1] P. Voigtlaender, P. Doetsch, and H. Ney, “Handwriting Recognition with Large Multidimensional Long Short-Term Memory Recurrent Neural Networks,” in *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. Shenzhen, China: IEEE, Oct. 2016, pp. 228–233. [Online]. Available: <http://ieeexplore.ieee.org/document/7814068/>
- [2] T. Bluche, J. Louradour, and R. Messina, “Scan, Attend and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention,” *arXiv:1604.03286 [cs]*, Aug. 2016, arXiv: 1604.03286. [Online]. Available: <http://arxiv.org/abs/1604.03286>
- [3] K. Dutta, P. Krishnan, M. Mathew, and C. Jawahar, “Improving CNN-RNN Hybrid Networks for Handwriting Recognition,” in *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. Niagara Falls, NY, USA: IEEE, Aug. 2018, pp. 80–85. [Online]. Available: <https://ieeexplore.ieee.org/document/8563230/>
- [4] U.-V. Marti and H. Bunke, “The IAM-database: an English sentence database for offline handwriting recognition,” *IJDAR*, vol. 5, no. 39-46, 2002.
- [5] A. Chowdhury and L. Vig, “An Efficient End-to-End Neural Model for Handwritten Text Recognition,” *arXiv:1807.07965 [cs]*, Jul. 2018, arXiv: 1807.07965. [Online]. Available: <http://arxiv.org/abs/1807.07965>
- [6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, Nov. 2016, google-Books-ID: omivDQAAQBAJ.
- [7] B. Shi, X. Bai, and C. Yao, “An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition,” *arXiv:1507.05717 [cs]*, Jul. 2015, arXiv: 1507.05717. [Online]. Available: <http://arxiv.org/abs/1507.05717>
- [8] J. Sueiras, V. Ruiz, A. Sanchez, and J. F. Velez, “Offline continuous handwriting recognition using sequence to sequence neural networks,” *Neurocomputing*, vol. 289, pp. 119–128, May 2018. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0925231218301371>
- [9] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber, “Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks,” in *Proceedings of the 23rd international conference on Machine learning*. Pittsburgh, PA: International Conference on Machine Learning, 2006, p. 8.
- [10] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber, “A Novel Connectionist System for Unconstrained Handwriting Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, May 2009. [Online]. Available: <http://ieeexplore.ieee.org/document/4531750/>
- [11] M. Ameryan and L. Schomaker, “Improving the robustness of LSTMs for word classification using stressed word endings in dual-state word-beam search,” Sep. 2020, pp. 13–18.
- [12] D. Nurseitov, K. Bostanbekov, M. Kanatov, A. Alimova, A. Abdallah, and G. Abdimanap, “Classification of Handwritten Names of Cities and Handwritten Text Recognition using Various Deep Learning Models,” *Advances in Science, Technology and Engineering Systems Journal*, vol. 5, no. 5, pp. 934–943, 2020, arXiv: 2102.04816. [Online]. Available: <http://arxiv.org/abs/2102.04816>
- [13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” *arXiv:1706.03762 [cs]*, Dec. 2017, arXiv: 1706.03762. [Online]. Available: <http://arxiv.org/abs/1706.03762>

- [14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805. [Online]. Available: <http://arxiv.org/abs/1810.04805>
- [15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” p. 24.
- [16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language Models are Few-Shot Learners,” *arXiv:2005.14165 [cs]*, Jul. 2020, arXiv: 2005.14165. [Online]. Available: <http://arxiv.org/abs/2005.14165>
- [17] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,” *arXiv:1412.3555 [cs]*, Dec. 2014, arXiv: 1412.3555. [Online]. Available: <http://arxiv.org/abs/1412.3555>
- [18] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, conference Name: Neural Computation.
- [19] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” *Signal Processing, IEEE Transactions on*, vol. 45, pp. 2673–2681, Dec. 1997.
- [20] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *arXiv:1409.0473 [cs, stat]*, May 2016, arXiv: 1409.0473. [Online]. Available: <http://arxiv.org/abs/1409.0473>
- [21] K. Xu, J. Lei, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,” p. 10.
- [22] M.-T. Luong, H. Pham, and C. D. Manning, “Effective Approaches to Attention-based Neural Machine Translation,” *arXiv:1508.04025 [cs]*, Sep. 2015, arXiv: 1508.04025. [Online]. Available: <http://arxiv.org/abs/1508.04025>
- [23] J. Cheng, L. Dong, and M. Lapata, “Long Short-Term Memory-Networks for Machine Reading,” *arXiv:1601.06733 [cs]*, Sep. 2016, arXiv: 1601.06733. [Online]. Available: <http://arxiv.org/abs/1601.06733>
- [24] A. F. de Sousa Neto, B. L. D. Bezerra, A. H. Toselli, and E. B. Lima, “HTR-Flor: A Deep Learning System for Offline Handwritten Text Recognition,” in *2020 33rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. Recife/Porto de Galinhas, Brazil: IEEE, Nov. 2020, pp. 54–61. [Online]. Available: <https://ieeexplore.ieee.org/document/9266005/>
- [25] L. Kang, P. Riba, M. Rusiñol, A. Fornés, and M. Villegas, “Pay Attention to What You Read: Non-recurrent Handwritten Text-Line Recognition,” *arXiv:2005.13044 [cs]*, May 2020, arXiv: 2005.13044. [Online]. Available: <http://arxiv.org/abs/2005.13044>
- [26] H. Bunke, M. Roth, and E. G. Schukat-Talamazzini, “Off-line Cursive Handwriting Recognition using Hidden Markov Models,” *Elsevier*, 1995.
- [27] T. Bluche, H. Ney, and C. Kermorvant, “Tandem HMM with convolutional neural network for handwritten word recognition,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 2390–2394, iSSN: 2379-190X.

- [28] L. Kang, J. I. Toledo, P. Riba, M. Villegas, A. Fornés, and M. Rusiñol, “Convolve, Attend and Spell: An Attention-based Sequence-to-Sequence Model for Handwritten Word Recognition,” in *Pattern Recognition*, T. Brox, A. Bruhn, and M. Fritz, Eds. Cham: Springer International Publishing, 2019, vol. 11269, pp. 459–472, series Title: Lecture Notes in Computer Science. [Online]. Available: <http://link.springer.com/10.1007/978-3-030-12939-2>
- [29] W. Swaileh, “Language Modelling for Handwriting Recognition,” *Modeling and Simulation*, p. 178, 2017.
- [30] J. Puigcerver, “Are Multidimensional Recurrent Layers Really Necessary for Handwritten Text Recognition?” in *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*. Kyoto: IEEE, Nov. 2017, pp. 67–72. [Online]. Available: <http://ieeexplore.ieee.org/document/8269951/>
- [31] S. Dutta, N. Sankaran, K. P. Sankar, and C. V. Jawahar, “Robust Recognition of Degraded Documents Using Character N-Grams,” in *2012 10th IAPR International Workshop on Document Analysis Systems*, 2012, pp. 130–134.
- [32] B. Shi, X. Wang, P. Lyu, C. Yao, and X. Bai, “Robust Scene Text Recognition with Automatic Rectification,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 4168–4176. [Online]. Available: <http://ieeexplore.ieee.org/document/7780821/>
- [33] M. Ameryan and L. Schomaker, “A limited-size ensemble of homogeneous CNN/LSTMs for high-performance word classification,” *Neural Computing and Applications*, Feb. 2021. [Online]. Available: <https://doi.org/10.1007/s00521-020-05612-0>
- [34] J. Michael, R. Labahn, T. Grüning, and J. Zöllner, “Evaluating Sequence-to-Sequence Models for Handwritten Text Recognition,” in *2019 International Conference on Document Analysis and Recognition (ICDAR)*, Sep. 2019, pp. 1286–1293, iSSN: 2379-2140.
- [35] L. Kang, P. Riba, M. Villegas, A. Fornés, and M. Rusiñol, “Candidate Fusion: Integrating Language Modelling into a Sequence-to-Sequence Handwritten Word Recognition Architecture,” *arXiv:1912.10308 [cs]*, Dec. 2019, arXiv: 1912.10308. [Online]. Available: <http://arxiv.org/abs/1912.10308>
- [36] J. A. Sánchez, V. Romero, A. H. Toselli, M. Villegas, and E. Vidal, “A set of benchmarks for Handwritten Text Recognition on historical documents,” *Pattern Recognition*, vol. 94, pp. 122–134, Oct. 2019. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0031320319302006>
- [37] T. Bluche and R. Messina, “Gated Convolutional Recurrent Neural Networks for Multilingual Handwriting Recognition,” in *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*. Kyoto: IEEE, Nov. 2017, pp. 646–651. [Online]. Available: <http://ieeexplore.ieee.org/document/8270042/>
- [38] J. Poulos and R. Valle, “Character-Based Handwritten Text Transcription with Attention Networks,” *arXiv:1712.04046 [cs, stat]*, Feb. 2021, arXiv: 1712.04046. [Online]. Available: <http://arxiv.org/abs/1712.04046>
- [39] A. Abdallah, M. Hamada, and D. Nurseitov, “Attention-Based Fully Gated CNN-BGRU for Russian Handwritten Text,” *Journal of Imaging*, vol. 6, no. 12, p. 141, Dec. 2020, number: 12 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2313-433X/6/12/141>

- [40] M. Yousef, K. F. Hussain, and U. S. Mohammed, “Accurate, Data-Efficient, Unconstrained Text Recognition with Convolutional Neural Networks,” *arXiv:1812.11894 [cs]*, Dec. 2018, arXiv: 1812.11894. [Online]. Available: <http://arxiv.org/abs/1812.11894>
- [41] F. P. Such, D. Peri, F. Brockler, P. Hutkowski, and R. Ptucha, “Fully Convolutional Networks for Handwriting Recognition,” *arXiv:1907.04888 [cs]*, Jul. 2019, arXiv: 1907.04888. [Online]. Available: <http://arxiv.org/abs/1907.04888>
- [42] D. Coquenat, Y. Soullard, C. Chatelain, and T. Paquet, “Have convolutions already made recurrence obsolete for unconstrained handwritten text recognition ?” *2019 International Conference on Document Analysis and Recognition Workshops (ICDARW)*, pp. 65–70, Sep. 2019, arXiv: 2012.04954. [Online]. Available: <http://arxiv.org/abs/2012.04954>
- [43] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.
- [44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [45] M. Bulacu, A. Brink, T. v. d. Zant, and L. Schomaker, “Recognition of Handwritten Numerical Fields in a Large Single-Writer Historical Collection,” in *2009 10th International Conference on Document Analysis and Recognition*, Jul. 2009, pp. 808–812, iSSN: 2379-2140.
- [46] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer Normalization,” *arXiv:1607.06450 [cs, stat]*, Jul. 2016, arXiv: 1607.06450. [Online]. Available: <http://arxiv.org/abs/1607.06450>
- [47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 2818–2826. [Online]. Available: <http://ieeexplore.ieee.org/document/7780677/>
- [48] C.-B. Zhang, P.-T. Jiang, Q. Hou, Y. Wei, Q. Han, Z. Li, and M.-M. Cheng, “Delving Deep into Label Smoothing,” *IEEE Transactions on Image Processing*, vol. 30, pp. 5984–5996, 2021, arXiv: 2011.12562. [Online]. Available: <http://arxiv.org/abs/2011.12562>
- [49] S. Karita, N. E. Y. Soplin, S. Watanabe, M. Delcroix, A. Ogawa, and T. Nakatani, “Improving Transformer-Based End-to-End Speech Recognition with Connectionist Temporal Classification and Language Model Integration,” p. 5.