Dissertation Type: Research

DEPARTMENT OF COMPUTER SCIENCE

SANscript

Self-Attention Networks for Handwritten Text Recognition

Rafael J. C. d'Arce

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Tuesday 13th July, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.
As agreed to by Dr Sion Hannuna, this dissertation did not require ethical review.

Rafael J. C. d'Arce, Friday 21st May, 2021

Contents

List of Figures

List of Tables

Executive Summary

Despite the increasing digitisation of human life, the inability to automatically and reliably recognise handwriting remains a major barrier in the wide-spread access of information. It can still be challenging for the visually or cognitively impaired to read vital documents in day-to-day life; whether it's educational material, medical prescriptions, or invitations to receive a vaccine. The manual transcription of written text — such as from cheques, mail or forms — is a bottleneck for many businesses, preventing them from fully automating the flow of their data. Countless historical documents and manuscripts still remain undigitised. The amount of knowledge and cultural heritage contained within them in libraries and archives around the world is immeasurable, but inaccessible to most people.

Automatic, human-level *handwritten text recognition* (HTR) is still an unsolved problem in the fields of machine learning and pattern recognition. HTR is especially challenging due to the great variability between different styles and scripts, and the costs of acquiring sufficient training data. Nevertheless, this technology has improved significantly in the last decade due, in part, to advancements in artificial neural networks. Specifically, the current state-of-the-art HTR models rely on *recurrent neural networks* (RNN). Unfortunately, due to their recurrent nature, RNNs are incredibly slow to train. This is an issue for applications of HTR; usually, an individual model has to be trained for each script of interest.

In 2017, the *Transformer*, a novel neural architecture, was proposed as an alternative to RNNs. It is entirely nonrecurrent and, instead, relies on *self-attention networks* (SAN) to process its inputs. Compared to RNNs, SANs are much faster to train and can achieve the same — if not better — performance. Since their introduction, SAN-based architectures have quickly become the dominant methods for natural language processing (NLP), replacing the recurrent networks that were used previously and leading to significant advancements in recognition. This begs the question: can the same be done for HTR?

We hypothesised that self-attention networks could replace the recurrent layers in existing, state-of-the-art HTR models; achieving comparable levels of recognition whilst reducing the time and resources required to train. To this end, we have:

- identified and successfully replicated some of the state-of-the-art architectures from the literature;
- proposed *SANscript*, a novel and nonrecurrent optical model architecture based on SANs;
- performed a rigorous ablation study into the design of SANscript;
- thoroughly benchmarked SANscript against the state-of-the-art on multiple public datasets;
- created a large, synthetic dataset and demonstrated that pre-training models on it can greatly improve performance;
- demonstrated that SANscript can closely approximate state-of-the-art recognition, whilst indeed requiring significantly less time to train than recurrent models; and
- made open-source contributions to a public HTR framework.

The idea for this project was based on a gap found in the handwriting recognition literature, and links concepts from various research communities. All design decisions are well motivated, and a detailed technical background is provided. Our work is heavily illustrated, containing over 50 custom-made visualisations. The few figures that were taken from other works have been cited accordingly.

We hope that our work is insightful to the reader; and that our results can inspire the use of SANs in the wider HTR community, as well as for other pattern recognition tasks.

Supporting Technologies

The following third-party resources have been used during this project:

- the Tensorflow[1] 2.3.0 deep learning library for the Python[2] 3.7.3 programming language;
- A. Flor's `handwritten-text-recognition` framework [47];
- E. Belval's `TextRecogntionDataGenerator` tool [12];
- M. Ernestus' implementation of the elastic deformation [39];
- the University of Bristol's BlueCrystal Phase 4 HPC machine [2];
- the Bentham Dataset [52, 144];
- the IAM Handwriting Database [105];
- the Saint Gall Database [45];
- the Washington Database [46]; and
- S. Kobayashi's Homemade Bookcorpus [86].

[1] https://www.tensorflow.org/
[2] https://www.python.org/

Acronyms and Notation

ANN	:	Artificial Neural Network
ASR	:	Automatic Speech Recognition
CNN	:	Convolutional Neural Network
CTC	:	Connectionist Temporal Classification
FFN	:	Feed-Forward Network
GCNN	:	Gated Convolutional Neural Network
GRU	:	Gated Recurrent Unit
HMM	:	Hidden Markov Model
HTR	:	Handwritten Text Recognition
MHA	:	Multi-Head Attention
NLP	:	Natural Language Processing
LSTM	:	Long Short-Term Memory
OCR	:	Optical Character Recognition
PE	:	Positional Encoding
RNN	:	Recurrent Neural Network
SAN	:	Self-Attention Network
SDPA	:	Scaled Dot-Product Attention
Seq2Seq	:	Sequence-to-Sequence
	:	
x	:	a scalar
$\boldsymbol{x}$	:	a vector
$\boldsymbol{x}_i$	:	the i-th value of $\boldsymbol{x}$
$\boldsymbol{x}^{<t>}$	:	$\boldsymbol{x}$ at time step t
$\boldsymbol{X}$	:	a matrix or, equivalently, a sequence of vectors
$\boldsymbol{X}^{(i)}$	:	the i-th vector of $\boldsymbol{X}$

Acknowledgements

I would like to thank Dr. Sion Hannuna and Dr. Nello Cristianini for their excellent supervision both before and during this project; especially given the unprecedented circumstances of the COVID-19 pandemic. I also thank Dr. Tom Todd, Dr. Terry Norton, Sam Tozer and Alessio Zakaria for their enlightening technical advice.

Chapter 1

Contextual Background

1.1 Handwritten Text Recognition

Handwritten text recognition (HTR) is the task of automatically transcribing a representation of handwriting into a digital text format. With its origins in the mid 20th Century [16, 65], HTR has been a long sought-after problem in the field of machine learning, closely tied with some of its most significant advancements. For example, some of the first applications of artificial neural networks were for recognising handwriting [92, 93, 94] and the popular *MNIST* handwritten digit dataset [95] is now one of the standard research benchmarks [31]. HTR is still an open area of research and its advancements are catalogued in conferences such as the *International Conference on Frontiers in Handwriting Recognition* (ICFHR) and *International Conference on Document Analysis and Recognition* (ICDAR).

HTR is differentiated from the task of *optical character recognition* (OCR), which involves transcribing printed text. HTR is generally considered a harder task than OCR, for example, training a HTR system can require up to 4× the amount of data than one for OCR [112]. This is often attributed to the great variability found between different handwriting styles (Figure 1.1). There is even much variability between samples of text written in the same style or by the same hand (Figure 1.2).

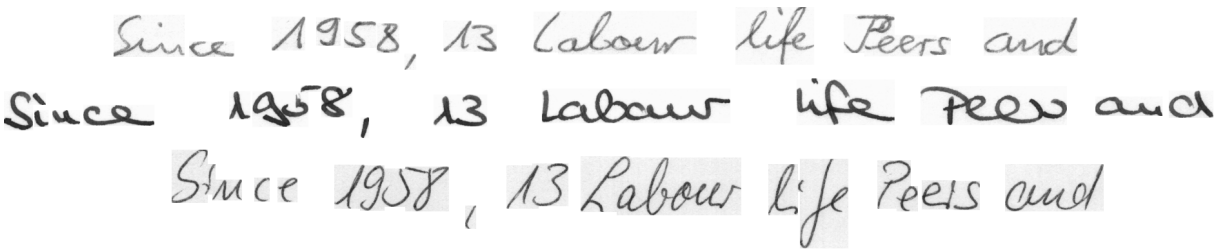


Figure 1.1: Examples from the IAM dataset of the same line of text written by different hands.

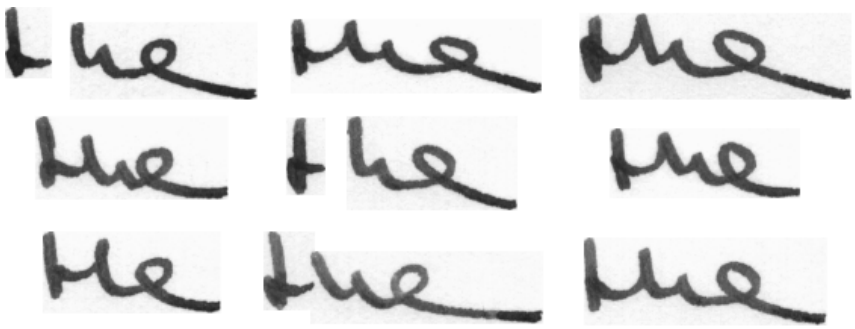


Figure 1.2: Examples from the IAM dataset of the same word written by the same hand.

HTR technology has, so far, been applied mostly in a commercial setting; such as for the recognition of bank cheques [48, 53, 142] and of postal addresses [117, 131, 132]. However, coinciding with mass digitisation initiatives at several archives and libraries around the world, there has been a growing demand

for the recognition of historical documents and manuscripts [116]. Initially, these applications of HTR were problematic: existing models were not trained on historical styles and many manuscripts lacked sufficient (if any) human transcriptions for these models to learn from [38]. However, recently, this has changed thanks to notable research efforts, such as the *tranScriptorium* (2013–2015) and *Recognition and Enrichment of Archival Documents* (READ; 2016–2019) projects funded by the European Union [40, 43]. These projects made several scientific breakthroughs in the field of HTR, such as the reduction of transcription errors on historical documents by 30 to 50% [111]. They also saw the development of a Virtual Research Environment for HTR called *Transkribus*. Now with over 25,000 registered users [41], it functions as an unified platform for all those interested in handwriting recognition: archivists, humanities scholars, computer scientists and even the general public [42].

This dissertation is concerned specifically with the *offline, line-level and segmentation-free* HTR task. Offline HTR converts text from static images of handwriting. This is opposed to online HTR, which dynamically transcribes text as it is being written on a medium such as a touchscreen. Online systems can leverage the physical properties of writing over time — including the position, trajectory and pressure of the pen [82] — whereas an offline system only works with the final image. For this reason, the offline HTR task is generally considered more challenging than the online one [123]. Some offline HTR systems are also required to first segment input images (say, of whole documents) into regions of interest that contain text [83, 130]. Meanwhile, segmentation-free HTR systems work directly with pre-segmented images. We consider segmentation-free HTR at the line-level, where the input images have already been segmented into individual text lines (which are comparable to sentences). It is also possible to perform the task on the level of individual characters [10, 11], words [10, 29], or paragraphs [18, 127].

In common practice, offline HTR systems make use of an *optical model* and a *language model* [146, 50, 127] (Figure 1.3). The role of the optical model is to generate the initial transcription from an input image, while the language model is used as a form of post-processing correction. Language models can be used to constrain transcriptions to only include words from a predefined vocabulary [87], or to refine them statistically based on what characters or words are generally most likely to follow one another [23]. We assume that the effects of using such a language model is independent of the architecture used for the optical model. Therefore, we do not apply any language models in our experiments, and only explore the design and use of the optical models for HTR.

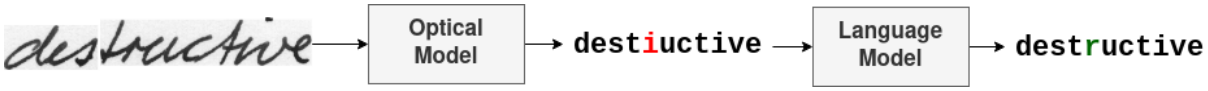


Figure 1.3: An offline HTR system that uses both optical and language models.

1.2 Optical Models for HTR

At its core, offline HTR is both an image processing and a sequence transduction task. The inputs to such systems are images, which are provided in the form of two-dimensional matrices of pixels. These matrices can be interpreted as sequences of vectors, where each vector corresponds to a column of pixels in the original image (Figure 1.4). The goal for the optical model is both to extract the visual features[1] from such an input sequence and to generate an output sequence of characters from these features.

Figure 1.4: Offline HTR as a sequence transduction task.

In the last decade, the literature for optical models has shifted from using *hidden markov models* (HMMs) for sequence transduction [21, 104, 106, 135] to using *recurrent neural networks* (RNN) [63, 64, 121]. HMMs are unable to model long-term sequence dependencies due to their Markovian assumption — the next character in the output depends only on the current state of the system, and not on any

[1] Previously, when hand-crafted features were used, feature extraction was performed as a pre-processing step and the optical model was only required to generate the transcription [94]. However, now that learned features are used, optical models are more commonly trained to jointly perform both tasks.

previous or subsequent ones. HMMs are, therefore, memoryless. Meanwhile, RNNs are able to model longer dependencies by storing an internal context of the previous sequence elements. There has also been a shift from using hand-crafted features [1, 90, 126] to those automatically learnt by *convolutional neural networks* (CNN) [13, 20, 121]. This has coincided with the breakthroughs achieved with CNNs in the field of computer vision [89].

The current state-of-the-art optical models for HTR use a combination of CNNs and RNNs in an *encoder-decoder* design [19, 50, 127]. Such architectures are composed of a block of convolutional layers for encoding a sequence of pixels into a sequence of feature vectors, succeeded by a block of recurrent layers for decoding the features into an output sequence (Figure 1.5). Despite operating on the line-level in our case, optical models actually generate outputs on the character-level. More specifically, given an input sequence of pixels, they predict a sequence of probability distributions over an predefined alphabet of characters.

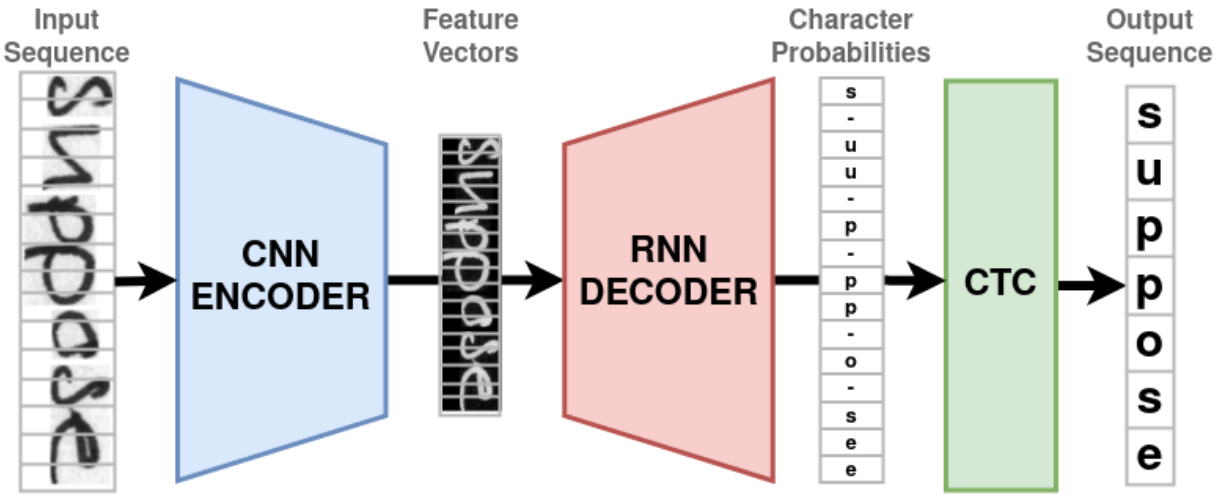


Figure 1.5: The general architecture of a CNN-RNN encoder-decoder optical model under CTC.

These character probabilities need to be decoded somehow in order to produce a concrete output sequence. However, the target sequence of characters is often shorter than the input sequence of pixels; meaning that one character in the transcription may actually span over multiple columns in the image. This can lead to incorrectly duplicated characters after decoding (Figure 1.6). Unfortunately, the true alignment between characters and pixel columns is usually unknown. Manually annotating the alignments for a significantly sized dataset is prohibitively time-consuming.

Figure 1.6: An example of unaligned HTR leading to incorrectly repeated characters after decoding.

These issues can be overcome by using the *connectionist temporal classification* (CTC) method. First proposed by A. Graves et al [59], CTC is able to model all possible alignments by introducing a *blank* character. The optical model can insert blanks anywhere, but must include them between any intentionally repeated characters. For the final output, CTC collapses repeated characters without a blank symbol into one, and removes all blank symbols (Figure 1.5). CTC can be used both as a loss function during training and as a decoding method at inference time [59]. It is widely used by the state-of-the-art CNN-RNN optical models for HTR [19, 50, 59, 97, 127]. It is also popular for other alignment-free sequence transduction tasks, such as speech recognition [62, 75, 152, 163].

1.3 Recent Advancements in Sequence Processing

1.3.1 The Limitations of RNNs

Despite their successes, RNNs suffer from some intrinsic limitations. Firstly, there is the *vanishing gradient problem* [14, 119]. As an RNN iterates, it is continually overwriting its memory of past elements; therefore, the information it stores of increasingly distant elements becomes increasingly attenuated. During training, this means that RNNs cannot effectively learn very long-term dependencies. Secondly, due to their recurrent nature, RNNs cannot be effectively parallelized[2] so they generally suffer from notoriously long training times. The *long short term memory* (LSTM) unit [72] is a greatly improved variant of the traditional RNN, that alleviates much of the vanishing gradient problem through the use of additional memory cells. In fact, this is the type of recurrent layer that much of the state-of-the-art optical models use [19, 63, 127], as well as what is used in industry by the Transkribus platform [108, 153] (Section 1.1). Despite this, LSTMs still cannot be parallelized well due to their recurrency, and so suffer from inefficient training. Their use in Transkribus was identified as a major bottleneck and that future developments of its HTR services will involve finding "*a faster, and maybe better, alternative to the LSTM*" [108].

1.3.2 Sequence-to-Sequence with Attention

In the wider literature for sequence transduction, there have been many recent attempts to improve on the limitations of recurrent networks. One such attempt was the so-called *sequence-to-sequence* architecture [8, 143]. The seq2seq design (Figure 1.7) is based on that of the encoder-decoder[3] (Section 1.2), however, there are some major differences. Firstly, recurrent layers are used in the encoder block - where previously they were only used in the decoder - in order to further process the sequence of feature vectors. Secondly, the decoder of a seq2seq model is autoregressive: it generates the output one item at a time, consuming previously generated items as an additional input to the feature vectors. For comparison, the decoder originally only uses the features. Lastly, but most importantly, attention is employed between the encoder and decoder.

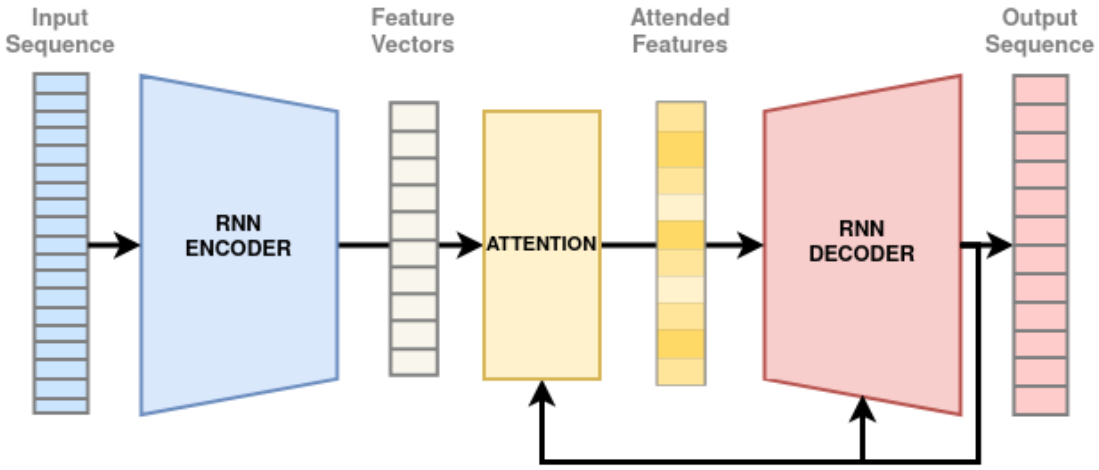


Figure 1.7: A generalised seq2seq architecture with attention. Note its autoregressive nature.

Attention mechanisms loosely mimic biological cognitive attention, in that they enhance the parts of an input that are the most relevant for the task at hand [8]. This is determined by a similarity function, which measures the interdependence of the elements of two sequences. In the case of seq2seq models, attention is applied between the encoder's features and the previous output of the decoder (Figure 1.7). This means that the decoder can, at each time step, selectively focus on the input features that are the most relevant for the current output element; regardless of their position in the input. In practice, this

[2] For example, an RNN cannot be applied to all elements of a sequence simultaneously — it must be done sequentially and in order.

[3] In some literature, seq2seq models are also referred to as encoder-decoders [8, 107]. However, before their rise in popularity, this name was used for the type of non-autoregressive models we describe [118, 19]. We understand that this may lead to some confusion for anyone previously acquainted with seq2seq.

allows for the flow of arbitrarily distant information (removing the vanishing gradient problem) and for the models to learn their own alignment between inputs and outputs (removing the need for CTC).

Seq2seq architectures saw adoption for sequence transduction between 2014 and 2017, achieving some impressive performance gains at tasks such as text translation [8, 143], text summarisation [113] and speech recognition [9, 162]. There have even been some applications of seq2seq for HTR [18, 81, 107], however, they do not provide the same levels of performance gain. Despite their improved recognition rates, seq2seq models are still incredibly inefficient to train due to their heavy reliance on recurrent networks. It also turns out that their improved performance is attributed to their use of attention and not of recurrency [148]. This spurred the idea that RNNs are not necessary for processing sequences and that *attention is all you need*. This would culminate in the development of the Transformer.

1.3.3 The Transformer and Self-Attention Networks

More recently, in the field of natural language processing (NLP), the *Transformer* has become the dominant architecture for sequence transduction, eclipsing the recurrent-based methods (including seq2seq) that came before it [155]. First presented in the seminal work by A. Vaswani et al [148] in 2017, this novel architecture can process sequential data without any recurrency. The Transformer architecture is based on that of seq2seq (i.e., encoder and autoregressive-decoder), however it removes all use of RNNs and replaces them with attention (Figure 1.8). More specifically, it uses stacks of *self-attention networks* (SAN) to repeatedly transform inputs. The self-attention mechanism in a SAN is almost identical to that in a seq2seq model, however, it compares an input sequence with itself rather than to a different sequence. This means that the new representation of a sequence generated by a SAN is based on how its elements all relate to one another.

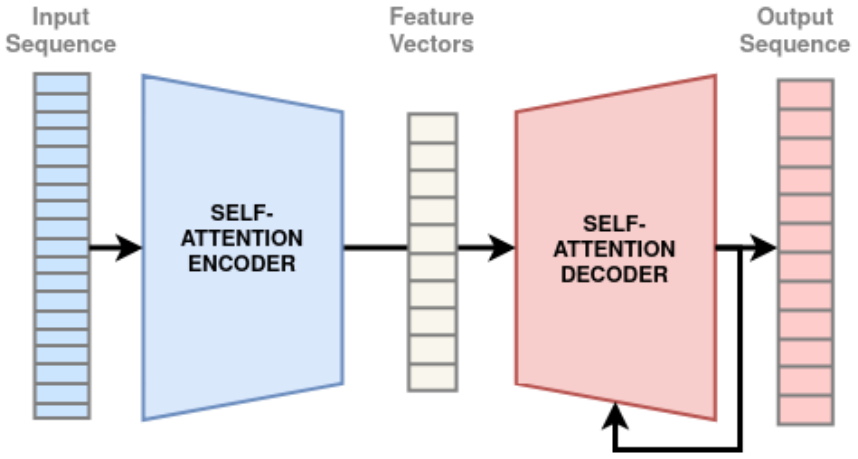


Figure 1.8: A generalised Transformer architecture. Note its similarity to the seq2seq design.

Therefore, SANs can model arbitrarily distant sequence dependencies by considering whole sequences at once. It does this very efficiently, connecting all positions in a constant number of sequential operations. On the other hand, the number of sequential operations required by a recurrent layer grows linearly with the length of the sequence. Also, recurrent layers are generally more computationally complex than self-attention [148]. This means that the Transformer can be much more effectively parallelized during training. In its original paper, the Transformer was demonstrated to achieve state-of-the-art results at machine translation whilst requiring significantly less time to train than its recurrent counterparts.

Since their introduction, the Transformer has also become the state-of-the-art for a variety of other NLP tasks; such as language understanding, text classification, text summarisation, sentiment analysis and question answering [91, 99, 100, 158]. This has been enabled by the pre-training of large Transformer architectures on very large datasets, which was previously impractical to do with RNNs due to their inefficient training. These pre-trained systems, such as *Bidirectional Encoder Representations from Transformers* (BERT) [35] and *Generative Pre-trained Transformer* (GPT) [128], are powerful and generic language models that can be adapted to downstream tasks with state-of-the-art performance using little-to-no fine-tuning. Transformer-based architectures have also started to replace recurrent networks outside of NLP, such as for speech recognition [36], object detection [25], image generation [129] and graph processing [149].

1.4 Motivation

This dissertation aims to find an improvement for the current state-of-the-art optical models using recent advancements in sequence transduction technology; this can be in terms of recognition ability or of training efficiency. There are multiple possible paths to take. For example, we could build a seq2seq model with attention — in fact, there have been several attempts at this before [18, 81, 107]. However these seq2seq models only provide minimal gains in performance over encoder-decoder models, whilst requiring more time to train. This is in contrast to the applications of seq2seq for machine translation, which provided significant enough performance gains to justify the increase in training times [8, 143]. There are some possible explanations for this discrepancy. For example, encoder-decoder models under the CTC framework can only output sequences that are less than or equal to in length to the input, whereas the autoregressive nature of a seq2seq model allows it to output sequences of any length. In the machine translation task, this is incredibly useful as a phrase in one language may use more words when translated into another language. However in HTR, there are generally fewer characters in the output than there are columns in the input image; so this is not an issue. Moreover, CTC enforces a monotonic alignment of inputs and outputs: strictly left-to-right and in increasing order. Whereas, the use of attention allows seq2seq models to learn alignments between elements in any position. This is ideal for translation, where corresponding words in two languages may be used at different positions, but for recognising handwriting this is usually not necessary - especially at the line level.

Another approach could be to build a Transformer-based optical model. To the best of our knowledge, there has only been one attempt at this so far. In the work by L. Kang et al [79], the original Transformer architecture (Section 1.3.3) is extended to use a CNN in its encoder to extract visual features. This nonrecurrent system is claimed to outperform the recurrent seq2seq models it was compared to, requiring $0.6\times$ the time to train despite having almost $3\times$ the number of parameters. This is both proof that SANs are viable for modelling the dependencies in handwritten text, and that they can be trained to do so more efficiently than RNNs. It was also shown that a Transformer optical model for HTR, pre-trained on generic handwriting styles, is able to adapt to a new and specific style using significantly less data than a seq2seq model. It turns out that the use of self-attention in the encoder was not necessary, and that the Transformer can achieve the same performance with a purely convolutional encoder; however, the use of SANs in the decoder is essential [79]. This, plus the fact that an autoregressive decoder is not required for line-level HTR, motivates the idea that SANs could also replace RNNs in the decoders of CTC-based models (Figure 1.9).

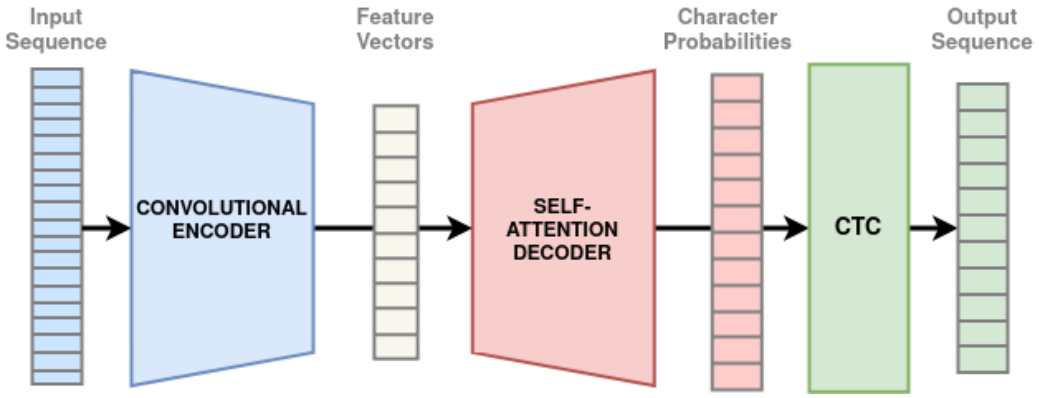


Figure 1.9: The general architecture of SANscript: an encoder-decoder optical model under the CTC framework, that has its RNNs replaced with SANs.

This is where we find a gap in the HTR literature. So far, there have been no attempts to incorporate self-attention layers into an encoder-decoder CTC optical model. However, there has been such an attempt for *automatic speech recognition* (ASR). ASR is a sequence transduction task closely related to handwriting recognition. This is both in terms of the linguistic nature and wide-variability of their data, as well as the model architectures that are used [74]. J. Salazar et al [133] proposed an entirely nonrecurrent *acoustic model*, which only uses SANs to model temporal dependencies. It is claimed to match the current state-of-the-art performance at speech recognition, whilst requiring up to $0.13\times$ the time to train. This proves that training a SAN-based model under CTC is tractable. While this acoustic model does not use convolutions in its encoder, the authors have suggested adding CNNs as a future development. This is further motivation for researching a CNN-SAN optical model.

1.5 Aims

The overall aim for this dissertation is to evaluate if SANs can be used as drop-in replacements for RNNs in existing encoder-decoder optical models under CTC. That is, to see if their use leads to the same level of performance as recurrent-based methods, whilst requiring less time to train and no more parameters. We facetiously refer to such an architecture as *SANscript*, and it is novel in the HTR literature. There are several challenges in doing this. Firstly, SAN-based networks are known to be unstable during early training steps; this could be resolved with careful learning rate scheduling [148]. Secondly, deep neural networks (whether recurrent or self-attentional) generally need large amounts of training data in order to generalise well, unfortunately, there is a scarcity of public datasets for HTR [79]. Two popular approaches to alleviate the cost of collecting sufficient annotated data are to randomly augment existing data samples [50, 127, 154], and to generate new samples synthetically [15, 79, 88].

In brief, the individual aims for this project are to:

- identify and replicate the current state-of-the-art performance for offline HTR;
- perform an ablation study into the design of the SANscript architecture;
- develop a large and synthetic HTR dataset;
- explore the benefits of synthetic pre-training; and
- evaluate if SANscript can approximate the state-of-the-art recognition and reduce the time required to train.

Chapter 2

Technical Background

This chapter aims to provide a complete technical background for this dissertation. This includes formalising relevant neural network components (Sections 2.1, 2.2 and 2.3) and the current state-of-the-art architectures for HTR optical models (Section 2.5). It also introduces attention mechanisms (Section 2.4) and our proposed architecture (Section 2.6). Note, the content of this chapter is considerably complex and low-level. We have reserved all necessary technical detail to this chapter, so that later chapters can discuss the relevant topics on a higher-level.

2.1 Neural Networks

It is assumed that the reader already understands the fundamentals of artificial neural networks, however for completeness, we will briefly discuss and formalise some key concepts in this section.

2.1.1 General Structure

Artificial neural networks (ANN; Figure 2.1) are nonlinear function approximators, loosely based on the biological neural networks found in the brain. Such systems consist of small units of computation (known as *neurons*) arranged into interconnected layers. Each connection has an associated *weight* that determines the strength of that connection. The value (or *activation*) that each neuron holds is calculated as the sum of the activations of the neurons connected to it, weighted by the strength of their connections. A *bias* term is also usually added after such a sum, in order to shift activations by some constant value. Also, the nonlinearity arises from the use of a nonlinear *activation function* at certain neurons. ANNs have an input layer and output layer, with zero or more *hidden* layers in between them — computation is performed by propagating the activations through each layer in order. The more hidden layers an ANN has, the *deeper* it is considered.

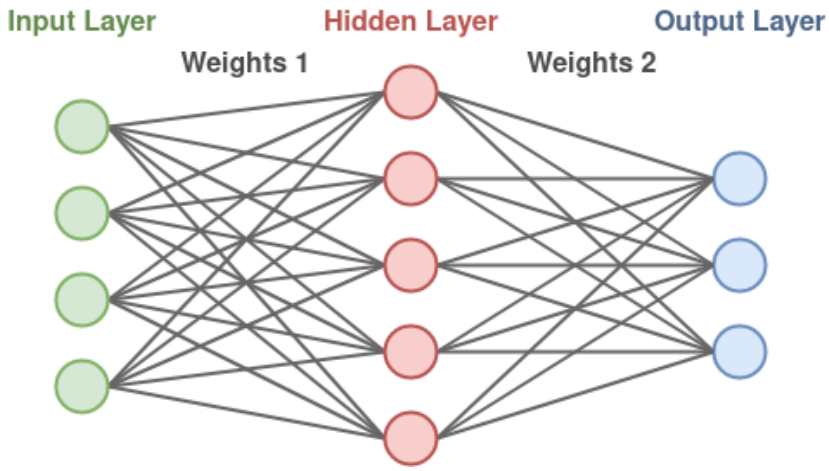


Figure 2.1: An artificial neural network with a single hidden layer (also known as a feed-forward network) as defined in Equations 2.2 and 2.3. Note, the biases and activation functions have been omitted for diagrammatic simplicity.

2.1.2 Formalisation

The most basic ANN, which we refer to as a *dense* network[1], is formalised as follows. It consists of two layers, input vector $\boldsymbol{x} \in \mathbb{R}^{D_x}$ and output vector $\hat{\boldsymbol{y}} \in \mathbb{R}^{D_y}$, that are connected with weight matrix $\boldsymbol{W} \in \mathbb{R}^{D_y \times D_x}$. Each input neuron $x_i \in \boldsymbol{x}$ is connected to each output neuron $y_j \in \hat{\boldsymbol{y}}$ with weight $\boldsymbol{W}_{ji}$. The Dense network also has an element-wise activation function $g : \mathbb{R}^{D_y} \to \mathbb{R}^{D_y}$ and bias vector $\boldsymbol{b} \in \mathbb{R}^{D_y}$. The activations of the output neurons are computed by multiplying[2] of the weights and the inputs, adding the bias and then applying the activation function (Equation 2.1).

$$\hat{\boldsymbol{y}} = g(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.1}$$

The next level up is the *feed-forward network*[3] (FNN) which introduces a hidden layer, $\boldsymbol{h} \in \mathbb{R}^{D_h}$, in between the inputs and outputs of the dense network. There are now two sets of weights and biases, $(\boldsymbol{W_1} \in \mathbb{R}^{D_h \times D_x}, \boldsymbol{b_1} \in \mathbb{R}^{D_h})$ and $(\boldsymbol{W_2} \in \mathbb{R}^{D_y \times D_h}, \boldsymbol{b_2} \in \mathbb{R}^{D_y})$; as well as two activation functions, $g_1 : \mathbb{R}^{D_h} \to \mathbb{R}^{D_h}$ and $g_2 : \mathbb{R}^{D_y} \to \mathbb{R}^{D_y}$. D_x, D_h and D_y are known as the input, hidden and output dimensions, respectively. This network operates by first propagating the input activations to the hidden layer (Equation 2.2), producing an intermediate representation that is then propagated to the output layer (Equation 2.3). This intermediate representation (also known as the *hidden state*) can be incredibly useful, as will be demonstrated later, because it encodes the latent features that the network has learnt to extract.

$$\boldsymbol{h} = g_1(\boldsymbol{W_1}\boldsymbol{x} + \boldsymbol{b_1}) \tag{2.2}$$

$$\hat{\boldsymbol{y}} = g_2(\boldsymbol{W_2}\boldsymbol{h} + \boldsymbol{b_2}) \tag{2.3}$$

2.1.3 Activation Functions

The weighted sums performed when propagating activations are linear transformations of their inputs, therefore in order for an ANN to model nonlinearities, a nonlinear *activation function* is required. For some layer $\boldsymbol{a}$, the activation function $g : \mathbb{R} \to \mathbb{R}$ is applied to each of its neurons a_i individually. Some popular activation functions (Figure 2.2) include the *hyperbolic tangent* (tanh; Equation 2.4), the *logistic sigmoid* (σ; Equation 2.5) and the *rectified linear unit* (ReLU; Equation 2.6). Another common activation, usually only applied on the output, is *softmax* (Equation 2.8). It ensures that a layer's activations are in the form of a probability distribution over a discrete variable with finite possible values [56]. This is ideal for classification tasks, where the model has to predict one of many classes; such as in HTR optical models (Section 1.2). It is also worth mentioning the *leaky ReLU* (LReLU; Equation 2.7) activation. LReLU is almost identical to the traditional ReLU; however, inputs $a_i \leq 0$ are mapped to small negative values αa_i (where hyperparameter $\alpha \ll 1$). Using either activation function generally leads to the same performance in the final network, but LReLU can result in faster training convergence [102].

$$tanh(a_i) = \frac{e^{2a_i} - 1}{e^{2a_i} + 1} \tag{2.4}$$

$$\sigma(a_i) = \frac{1}{1 + e^{-a_i}} \tag{2.5}$$

$$ReLU(a_i) = \begin{cases} 0, & \text{if } a_i \leq 0, \\ a_i, & \text{otherwise.} \end{cases} \tag{2.6}$$

$$LReLU(a_i) = \begin{cases} \alpha a_i, & \text{if } a_i \leq 0, \\ a_i, & \text{otherwise.} \end{cases} \tag{2.7}$$

$$softmax(a_i) = \frac{e^{a_i}}{\sum\limits_{a_j \in \boldsymbol{a}} e^{a_j}} \tag{2.8}$$

[1] We take this name from the Tensorflow implementation (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense). It is also known as a *fully-connected layer*.

[2] This is equivalent to calculating $\sum_{i=1}^{|\boldsymbol{x}|} \sum_{j=1}^{D_2} \boldsymbol{W_{ji}}\boldsymbol{x}_i$

[3] We take this name from [148]. It can also be used to refer to networks with more than one hidden layer.

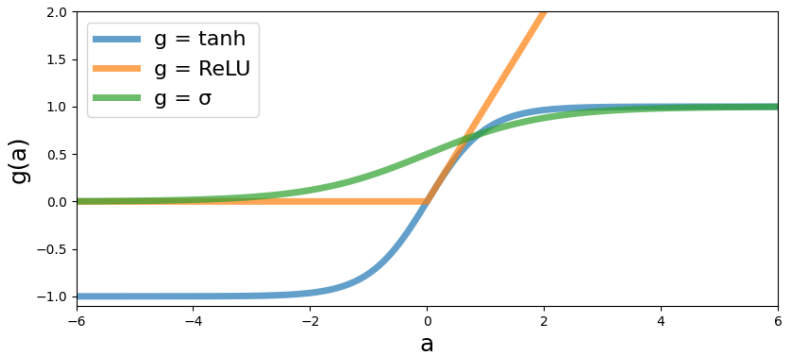


Figure 2.2: The value of different activation functions g over a range of input activations a.

2.1.4 Training

An ANN is optimised (or *trained*) to perform a task by adjusting its weights and biases (or *parameters*) in order to minimise a loss function. For a given input, the loss function $\mathfrak{L}$ measures how dissimilar the network's output, as currently parameterised, is from the correct output that has been pre-determined by a human (the *ground-truth*). Training is usually done using a *gradient descent* algorithm, which iteratively updates the network parameters' based on how they contribute to the loss.

More specifically, each iteration involves two steps. Firstly, the network is given an input $\boldsymbol{x}$, and the activations of its output $\hat{\boldsymbol{y}}$ and intermediate layers $\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_n$ are stored. This input has a corresponding ground-truth output $\boldsymbol{y}$. Secondly, the derivative (or *gradient*) of the loss function $\mathfrak{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is calculated with respect to each layer's set of parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, ..., \boldsymbol{\theta}_n$. These parameters are updated proportionally to the negative of this gradient (Equation 2.9, where η is the proportionality). η is also known as the *learning rate*. The *back-propagation* algorithm can efficiently calculate these gradients via the chain rule and dynamic programming; propagating the loss signal backwards through the network layers (Equation 2.10). In standard gradient descent, all training samples are used at each iteration. With *stochastic gradient descent*, a fixed-size subset (*or batch*) of inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_B$ is used at each training iteration. However, the batch size B is limited by the amount of computer memory available during training.

$$\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i - \eta \frac{\partial \mathfrak{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{\theta}_i} \tag{2.9}$$

$$\frac{\partial \mathfrak{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{\theta}_i} = \frac{\partial \mathfrak{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h_n}} \frac{\partial \boldsymbol{h_n}}{\partial \boldsymbol{h_{n-1}}} ... \frac{\partial \boldsymbol{h_{i+1}}}{\partial \boldsymbol{h_i}} \frac{\partial \boldsymbol{h_i}}{\partial \boldsymbol{\theta}_i} \tag{2.10}$$

The training of ANNs is often rife with issues, however, there exists some general techniques that help improve the process. For example there is a phenomenon called *internal covariate shift*: during training, as a layer's weights are adjusted, the distribution of its activations change [77]. This effect is amplified the more layers a network has. Learning, especially for a deep network, becomes more challenging as the layers have to adapt to continuously changing distributions. *Batch normalisation* (BN) [77] is a popular and powerful technique for alleviating covariate shift. For every neuron in the layer it is applied to, BN normalises the activation value by approximating the mean and variance across all batch samples. This encourages a fixed activation distribution for the layer, which can help greatly speed up the training process of the whole network [7, 77]. Another issue that often plagues training is *overfitting*. One way to prevent this is through the use of a validation dataset. The network is not trained on this dataset, instead it is evaluated on it at periodic intervals during training. By monitoring this performance, training can be terminated early before the model overfits (i.e. when the validation loss gets worse despite the training loss improving). *Dropout* can also help prevent overfitting: at every training iteration, each neuron and their connected weights are temporarily removed with some fixed probability rate [140]. This prevents neurons from becoming codependent, reducing the risk of overfitting.

The way that the parameters of an ANN are initialised at the start of training can have significant impact on both the time required to train it and its final performance [55, 70]. While there are many possible ways to do this, the *Glorot* [55] and *He* [70] methods are some of the most popular. For a weight matrix $\boldsymbol{W} \in \mathbb{R}^{D_y \times D_x}$, both methods randomly initialise each weight $\boldsymbol{W}_{ij} \sim U[-v, v]$ according to an uniform distribution U over the range $(-v, v)$. Glorot uses $v = \sqrt{\frac{6}{D_x + D_y}}$ and He uses $v = \sqrt{\frac{6}{D_x}}$. As a general rule of thumb, the Glorot method is applied on networks with tanh or sigmoid activations, and He on networks with ReLU activations [70].

2.2 Recurrent Neural Networks

In this section, several types of recurrent neural network are formalised and their limitations discussed. This is highly relevant, as these architectures are widely used in HTR optical models (Section 1.2) and our aim is to improve on them (Section 1.5).

2.2.1 Traditional RNN

The *recurrent neural network* (RNN) is a natural extension of the ANN that is able to process variable-length sequences of vectors, such as the features of an image (Section 1.2). As its name suggests, the RNN processes such a sequence iteratively with a recursive connection between its hidden states. At each time step, it consumes a new input element and the previous hidden state, generating a new hidden state (Figure 2.3). This recursion allows the RNN to store information of previous elements over time, enabling it to model dependencies between potentially distant elements.

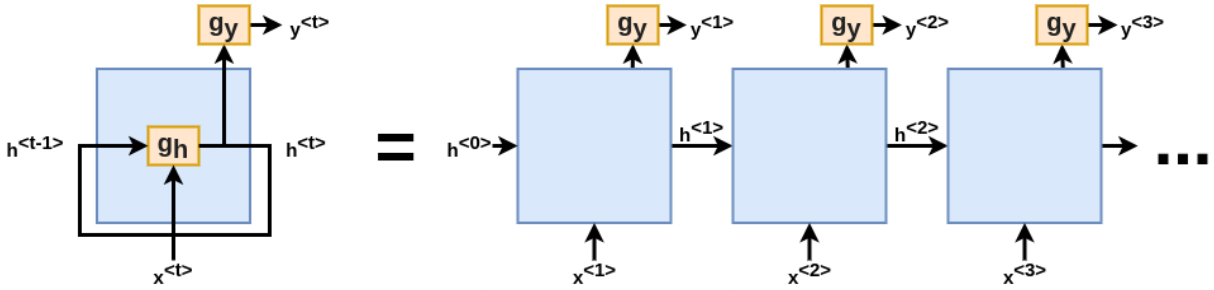


Figure 2.3: The traditional RNN, as defined in Equations 2.11 and 2.12, processing a sequence.

When a new hidden state is generated, it can be optionally transformed into an output vector using another neural network layer (Figure 2.3). In the case of HTR optical models, we would like distributions of character probabilities as output; this is usually done with a softmax dense layer [50, 127]. Another option is to collect all hidden states into a new sequence and process that using another RNN. Stacking RNNs in this way can allow for more complex dependency modelling.

The RNN, by default, processes a sequence in only one direction (usually position-increasing order). However, this means that each hidden state can only encode information of the elements prior to it and none after. It is possible to have bi-directional RNNs [134]. That is, two recurrent networks that process the same sequence from both directions, combining corresponding hidden states in the final output. Furthermore, RNNs can be extended to multiple-dimensions [61]: by introducing recurrency along another axis, they can model both temporal and spatial dependencies.

We formalise the canonical, uni-directional and one-dimensional RNN following the work of R. Pascanu et al [119]. At each time step t, an RNN takes two vectors as input: the current sequence element $\boldsymbol{x}^{<t>} \in \mathbb{R}^{D_x}$ and the previous hidden state $\boldsymbol{h}^{<t-1>} \in \mathbb{R}^{D_h}$. First, these inputs are projected using weights $\boldsymbol{W_x} \in \mathbb{R}^{D_x} \times \mathbb{R}^{D_h}$ and $\boldsymbol{W_h} \in \mathbb{R}^{D_h} \times \mathbb{R}^{D_h}$ (respectively). The projections are added together, along with a bias vector $\boldsymbol{b_h} \in \mathbb{R}^{D_h}$. Finally, an activation function $g_h : \mathbb{R}^{D_h} \to \mathbb{R}^{D_h}$ is applied. This produces a new hidden state $\boldsymbol{h}^{<t>}$ (Equation 2.11).

$$\boldsymbol{h}^{<t>} = g_h(\boldsymbol{W_{h1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{h2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_h}) \tag{2.11}$$

The new hidden state can be used to generate the output vector $\hat{\boldsymbol{y}}^{<t>} \in \mathbb{R}^{D_h}$ for this time step. This is done using a dense layer with weights $\boldsymbol{W_y} \in \mathbb{R}^{D_h} \times \mathbb{R}^{D_y}$, bias $\boldsymbol{b_y} \in \mathbb{R}^{D_y}$ and activation function $g_y : \mathbb{R}^{D_y} \to \mathbb{R}^{D_y}$ (Equation 2.12). For the first time step $t = 1$, the initial hidden state is usually set to zero $\boldsymbol{h}^0 = \{0\}^{D_h}$, but it can also be supplied by the user or learnt by the network during training.

$$\hat{\boldsymbol{y}}^{<t>} = g_y(\boldsymbol{W_y}\boldsymbol{h}^{<t>} + \boldsymbol{b_y}) \tag{2.12}$$

This formulation demonstrates why RNNs cannot be parallelized well: $\hat{\boldsymbol{y}}^{<t>}$ cannot be computed until $\{\boldsymbol{h}^{<1>}, \boldsymbol{h}^{<2>}, ..., \boldsymbol{h}^{<t-1>}\}$ have been computed first. Recurrency itself precludes parallelisation, and even the improved variants of the RNN (which will be discussed later) cannot resolve this.

As for the vanishing gradient problem, we must discuss how to train an RNN. The total error $\mathcal{E}$ for a given input-output pair $(\boldsymbol{x}, \boldsymbol{y})$ is the sum of the values of the loss function at each time step (Equation 2.13). For simplicity let us collect all of our parameters — the weights and biases — into θ. We can train an RNN using gradient descent, by computing the derivatives using *backpropagation through time*. This is defined in Equations 2.14, 2.15 and 2.16; where ∂^+ is the immediate partial derivative, *diag* a function that maps vectors to diagonal matrices and ${g_h}'$ the derivative of g_h. Each summation term in Equation 2.14 measures how much θ affects the error at step k. Meanwhile, Equation 2.16 transports the error signal from step t back to k. We reserve the derivation of these equations to the work by R. Pascanu et al [119].

$$\mathcal{E} = \sum_{t=1}^{T} \mathcal{E}^{<t>} = \sum_{t=1}^{T} \mathfrak{L}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>}) \tag{2.13}$$

$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{E}^{<t>}}{\partial \theta} \tag{2.14}$$

$$\frac{\partial \mathcal{E}^{<t>}}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial \mathcal{E}^{<t>}}{\partial \boldsymbol{h}^{<t>}} \frac{\partial \boldsymbol{h}^{<t>}}{\partial \boldsymbol{h}^{<k>}} \frac{\partial^+ \boldsymbol{h}^{<k>}}{\partial \theta} \tag{2.15}$$

$$\frac{\partial \boldsymbol{h}^{<t>}}{\partial \boldsymbol{h}^{<k>}} = \prod_{i=t}^{k} \frac{\partial \boldsymbol{h}^{<i>}}{\partial \boldsymbol{h}^{<i-1>}} = \prod_{i=t}^{k} \boldsymbol{W_{h2}}^T diag({g_h}'(\boldsymbol{h}^{<i-1>})) \tag{2.16}$$

In Equation 2.16 is where we find the vanishing gradient problem: it takes the form of the product of $t-k$ Jacobian matrices. Just as a product of $t-k$ small real numbers can shrink exponentially quickly to zero[4], so too can this product. If $\boldsymbol{W_h}$ is small, this product gets exponentially smaller as t and k get further apart [14, 119]. It is sufficient for the largest eigenvalue of $\boldsymbol{W_h}$ to be less than 1 for this to occur. When this happens during training, there is no way to update the parameters in order to capture a dependency between the elements at t and k. By the same token, there exists the *exploding gradient problem*: if the largest eigenvalue of $\boldsymbol{W_h}$ is greater than 1, it is possible for the product to explode to infinity. However, unlike vanishing gradients, we can easily remedy this problem. For example, we can clip gradients if their magnitudes $m = ||\frac{\partial \mathcal{E}}{\partial \theta}||$ exceed some threshold M (Equation 2.17). To alleviate the vanishing gradient problem, meanwhile, the RNN architecture itself has to be changed.

$$\frac{\partial \mathcal{E}}{\partial \theta} \leftarrow \frac{M}{m} \frac{\partial \mathcal{E}}{\partial \theta} \tag{2.17}$$

The key point to take away about the vanishing gradient problem is that it occurs because the hidden state of the RNN (its memory) is constantly being overwritten via nonlinear transformations (Equation 2.11). Let us consider how a computer works: it has a large amount of memory, but usually only interacts with a small portion of it during each instruction. Most of the memory remains unchanged. If we consider the state of a computer's memory as a large vector, the mapping of the state at one time step to the next is approximately the identity function. Unfortunately, nonlinear functions (such as *tanh* and σ) cannot be used to represent an identity mapping. It is, therefore, difficult for the RNN to keep its memory unchanged over many time steps [66].

[4] For example, $0.05 \times 0.05 \times 0.05 \times 0.05 = 0.00000625$. That is only 4 time steps.

2.2.2 LSTM

The Long Short-Term Memory (LSTM) network is a variant of the traditional RNN that directly addresses the issue of preserving information over many time steps. Despite being first proposed in the 1990s [72], it only started to see wide-adoption in the early 2010s [57] and now it is the de facto recurrent unit in sequence modelling [66]. The LSTM introduces a series of gating mechanisms, which enable it to accurately control which information is stored over time. It also maintains a dedicated memory cell $\boldsymbol{c}^{<t>}$ recursively, alongside the hidden state $\boldsymbol{h}^{<t>}$. This is an attempt to decouple the information that is to be stored and emitted. In the traditional RNN, there was only a recursive connection between hidden states and this had to perform both roles.

A gate is simply a dense layer (Equation 2.1) with a sigmoid activation (σ; Equation 2.5) that is used to control the flow of information. The output of a gate is a vector of values between 0 and 1, which is then multiplied position-wise to some other vector of the same length. A gating value closer to 0 blocks information at that position in the other vector, whereas a value closer to 1 allows it to flow through. In the LSTM, there are three gates at every time step t: *forget* $\boldsymbol{f}^{<t>}$, *input* $\boldsymbol{i}^{<t>}$ and *output* $\boldsymbol{o}^{<t>}$; each of which consumes the previous hidden state $\boldsymbol{h}^{<t-1>}$ and the current input element $\boldsymbol{x}^{<t>}$ (Equations 2.18, 2.19 and 2.20). The forget gate controls what information to keep from the previous hidden state. The input gate controls what new information to add to the new hidden state. The output gate controls what parts of the new hidden state are to be used in the new output.

$$\boldsymbol{f}^{<t>} = \sigma(\boldsymbol{W_{f1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{f2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_f}) \tag{2.18}$$

$$\boldsymbol{i}^{<t>} = \sigma(\boldsymbol{W_{i1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{i2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_i}) \tag{2.19}$$

$$\boldsymbol{o}^{<t>} = \sigma(\boldsymbol{W_{o1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{o2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_o}) \tag{2.20}$$

At every time step, the LSTM first extracts some new information $\tilde{\boldsymbol{c}}^{<t>}$ from the current inputs using a dense layer with a *tanh* activation (Equation 2.21). Next, it updates its memory $\boldsymbol{c}^{<t>}$ by erasing (or forgetting) parts of the previous memory state, and by adding parts of the new information selectively (Equation 2.22; where $\times$ is the element-wise product). The new hidden state emitted $\boldsymbol{h}^{<t>}$ is computed by applying a *tanh* activation and the output gate to the new memory cell state (Equation 2.23).

$$\tilde{\boldsymbol{c}}^{<t>} = tanh(\boldsymbol{W_{c1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{c2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_c}) \tag{2.21}$$

$$\boldsymbol{c}^{<t>} = \boldsymbol{f}^{<t>} \times \boldsymbol{c}^{<t-1>} + \boldsymbol{i}^{<t>} \times \tilde{\boldsymbol{c}}^{<t>} \tag{2.22}$$

$$\boldsymbol{h}^{<t>} = \boldsymbol{o}^{<t>} \times tanh(\boldsymbol{c}^{<t>}) \tag{2.23}$$

As you can in Equation 2.22, the memory cell is only ever updated with a few linear transformations. This can be also seen in Figure 2.4, where the information travels through the LSTM with minimal modification. This is the trick to ensuring that information can be stored unchanged for a long period of time, in turn alleviating the vanishing gradient problem.

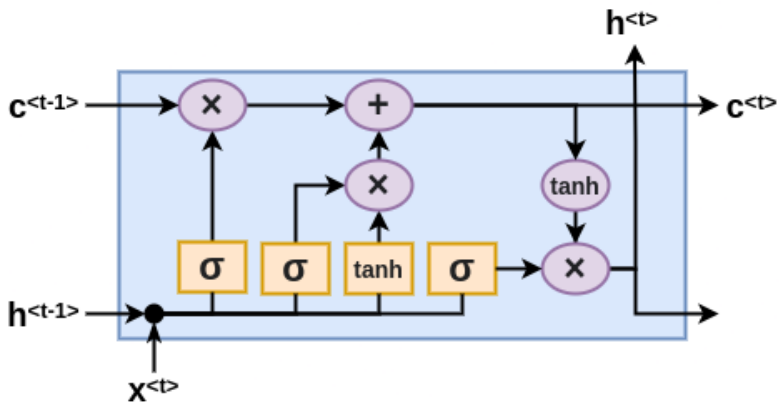


Figure 2.4: The architecture of the LSTM, based on Equations 2.18 to 2.23.

2.2.3 GRU

The *gated recurrent unit* [28] (GRU; Figure 2.5) is a simplified alternative to the LSTM, that also makes use of gating mechanisms but does away with dedicated memory cells. Another major difference is that the GRU exposes its entire hidden state at every time step, whereas the LSTM only exposed parts of it selectively via the output gate. Generally, LSTMs and GRUs share comparable sequence modelling capabilities and both equally alleviate the vanishing gradient problem. However, due to its simpler design, the GRU is usually the smaller and faster option [30].

At every time step t, the GRU takes in the previous hidden state $\boldsymbol{h}^{<t-1>}$ and current input element $\boldsymbol{x}^{<t>}$ to produce a new hidden state $\boldsymbol{h}^{<t>}$. To do so, it uses two gates: *update* $\boldsymbol{u}^{<t>}$ (Equation 2.24) and *reset* $\boldsymbol{r}^{<t>}$ (Equation 2.25). The update gate controls both what parts of the hidden state are updated and preserved, whereas the reset gate controls what parts of the previous hidden state are used in computing the new information.

$$\boldsymbol{u}^{<t>} = \sigma(\boldsymbol{W_{u1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{u2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_u}) \tag{2.24}$$

$$\boldsymbol{r}^{<t>} = \sigma(\boldsymbol{W_{r1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{r2}}\boldsymbol{h}^{<t-1>} + \boldsymbol{b_r}) \tag{2.25}$$

Using a tanh dense layer and the reset gate, some new information $\tilde{\boldsymbol{h}}^{<t>}$ is extracted from the inputs (Equation 2.26). The new hidden state is a linear interpolation of this information and the previous hidden state (Equation 2.27). Notice how the update gate controls both what is kept from the previous state and what is added from the new information — this is for efficiency. Also, the hidden state is emitted as a whole without any further gating.

$$\tilde{\boldsymbol{h}}^{<t>} = tanh(\boldsymbol{W_{h1}}\boldsymbol{x}^{<t>} + \boldsymbol{W_{h2}}(\boldsymbol{r}^{<t>} \times \boldsymbol{h}^{<t-1>}) + \boldsymbol{b_h}) \tag{2.26}$$

$$\boldsymbol{h}^{<t>} = \boldsymbol{u}^{<t>} \times \tilde{\boldsymbol{h}}^{<t>} + (1 - \boldsymbol{u}^{<t>}) \times \boldsymbol{h}^{<t-1>} \tag{2.27}$$

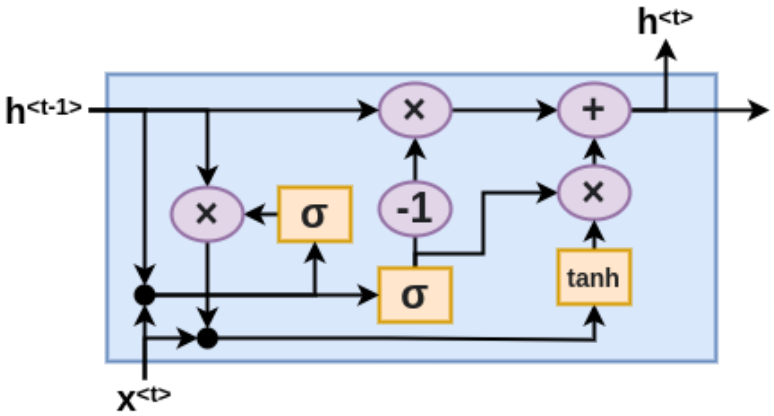


Figure 2.5: The architecture of the GRU, based on Equations 2.24 to 2.27.

2.3 Convolutional Neural Networks

In this section, convolutional neural networks are presented in brief. Despite being crucial in achieving the state-of-the-art performance in HTR (Section 1.2), this dissertation is focused more on methods for sequence modelling than those for image processing.

2.3.1 Traditional CNN

Convolutional neural networks (CNN) [96] are a class of ANN architecture used for extracting robust hierarchies of features from data with a grid-like format, such as images of handwriting. In fact, some of the earliest applications of CNNs were for recognising handwritten text [92, 93, 94]. CNNs are multi-stage systems, with each stage generally consisting of a *filter bank* layer, a nonlinearity layer and a *pooling* layer (Figure 2.6). The inputs and outputs of each stage are sets of arrays called *feature maps*. To extract features from an input map, the filters are applied to all local groups of values in a sliding-window fashion via discrete convolutions[5]. At this stage, the activation can be applied element-wise. Next, the pooling layer down-samples these intermediate feature maps in a similar sliding-window manner, usually taking the average or maximum value amongst local groups. At the end of all stages the final set of feature maps can be flattened into a sequence of vectors, to be further processed or classified by subsequent neural networks (Section 1.2).

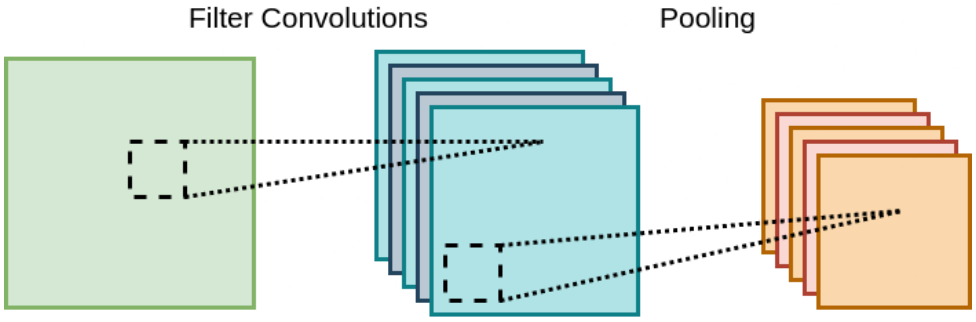


Figure 2.6: The general structure of a convolution neural network stage, consisting of filters for feature extraction and poolings for intermediate down-sampling. Note, the activation function has been omitted for diagrammatic simplicity.

In the context of an image processing task, such as HTR, these feature maps are 2-dimensional matrices. We formalise such a CNN stage following the work of J. Springenber et al [138]. The input $\boldsymbol{X} \in \mathbb{R}^{N \times W_1 \times H_1}$ is a set of N many feature maps of size $W_1 \times H_1$. Each input map is denoted as $\boldsymbol{X}^{(n)}$, and each value within it as $\boldsymbol{X}^{(n)}_{w,h}$. The final output of a CNN stage, $\boldsymbol{Y} \in \mathbb{R}^{M \times W_2 \times H_2}$ is denoted in a similar fashion. The filter bank has M many filters (or *kernels*) of size $K_1 \times K_2$, where each kernel $\boldsymbol{K}^{(m)}$ connects all input maps $\boldsymbol{X}$ to the output map $\boldsymbol{Y}^{(m)}$. Each intermediate feature map $\boldsymbol{Z}^{(m)}$ is calculated as per Equation 2.28, where g is a nonlinear activation function (usually *ReLU* [68]), $K'_1 = \lfloor K_1/2 \rfloor$, $K'_2 = \lfloor K_2/2 \rfloor$ and (S_1, S_2) is the stride of the sliding-window convolution.

$$\boldsymbol{Z}^{(m)}_{i,j} = g\left(\sum_{w=-K'_1}^{K'_1} \sum_{h=-K'_2}^{K'_2} \sum_{n=1}^{N} \boldsymbol{K}^{(m)}_{w,h} \cdot \boldsymbol{X}^{(n)}_{S_1 \cdot i+w,\ S_2 \cdot j+h} \right) \tag{2.28}$$

The output map $\boldsymbol{Y}^{(m)}$ is acquired by pooling (in this case, p-norm subsampling) the intermediate map $\boldsymbol{Z}^{(m)}$ as per Equation 2.29; where $(P_1 \times P_2)$ is the pool size, $P'_1 = \lfloor P_1/2 \rfloor$ and $P'_2 = \lfloor P_2/2 \rfloor$. As $p \to \infty$, this is the *max pool* operation, and when $p = 1$ this is the *average pool* [67].

$$\boldsymbol{Y}^{(m)}_{i,j} = \left(\sum_{w=-P'_1}^{P'_1} \sum_{h=-P'_2}^{P'_2} \left| \boldsymbol{Z}^{(m)}_{P_1 \cdot i+w,\ P_2 \cdot j+h} \right|^p \right)^{1/p} \tag{2.29}$$

[5] Unsurprisingly, this is where CNNs get their name from.

The convolutional filters exploit the fact that local neighbourhoods of pixels are highly correlated, and they extract features at all positions of the input image. This also means that the filters are *spatially equivariant*: for a given visual object, the same feature is extracted regardless of its location in the image. Meanwhile, pooling reduces the resolution of the feature maps, enabling subsequent stages to extract higher level features. This is because some region in a post-pool map represents a larger region in the pre-pool map. By having multiple stages, a CNN is able to extract hierarchies of features at increasing levels of abstraction. The earliest filters extract low-level features — such as edges and textures — which subsequent filters will combine into more complex shapes, and then into whole objects (Figure 2.7). Previous image processing methods would also combine feature extractors in such a way, however the filters would be hand-crafted [96]. CNNs alleviate this work by learning their own filters, which often outperform hand-crafted ones [89].

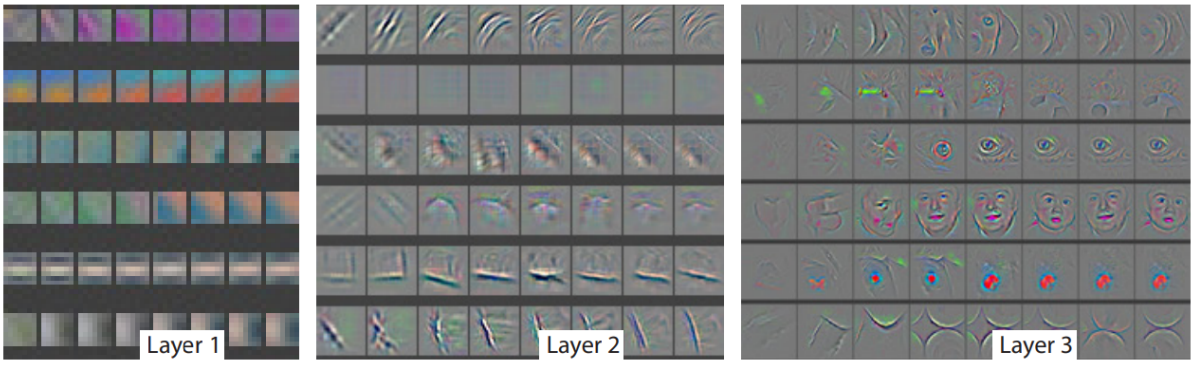


Figure 2.7: A visualisation of the features extracted at various stages of a CNN trained for object detection; taken from [161]. Note how the level of abstraction of the features increases with depth.

2.3.2 Gated CNN

Inspired by the gating mechanisms used by the LSTM (Section 2.2.2), various *gated convolution* layers have been proposed to better control the flow of features through a CNN [19, 34, 50]. In this work, we choose to follow the implementation described by T. Bluche et al [19]. Simply, a gated convolution G is a standard convolution layer with a sigmoid activation. A set of feature maps $\boldsymbol{X}$ goes through the gate, the output of which is multiplied position-wise with the original maps (Equation 2.30). The gated convolution selectively controls which features are propagated forwards based on their context. This enables features to be either excitatory or inhibitory (Figure 2.8).

$$\boldsymbol{Y} = G(\boldsymbol{X}) \times \boldsymbol{X} \tag{2.30}$$

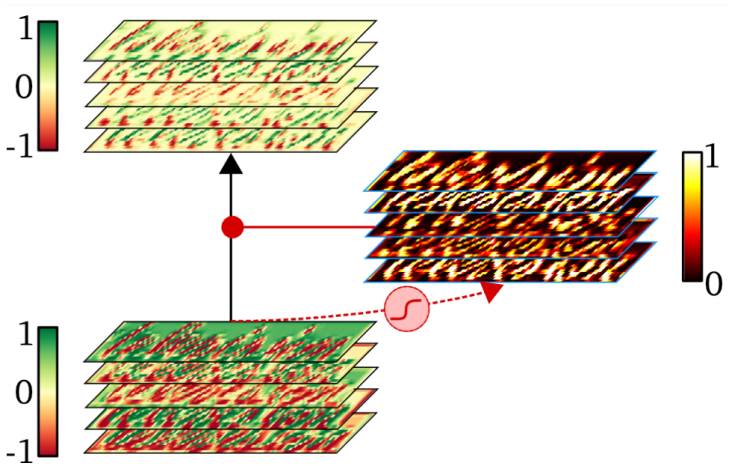


Figure 2.8: A demonstration of a gated convolution layer operating on an image of handwriting; taken from [19]. Note how the background information is inhibited, whilst the text itself is enhanced.

2.4 Neural Networks with Attention

This section discusses attention mechanisms, how they work, and their applications in a variety of neural network models. Such mechanisms are core to our proposed architecture (Sections 1.4 and 2.6).

2.4.1 Attention

Attention mechanisms in neural networks mimic cognitive attention found in animals and humans: it is the ability to pick out the most salient parts of an input for the task at hand (Figure 2.9). Attention, really, involves ignoring things; it doesn't add more information into the system, but instead removes information deemed irrelevant. All neural networks have some form of *implicit* attention [60]; that is, they learn to respond more to certain features of their inputs. However, the use of an *explicit* attention mechanism has proved to be powerful — improving the interpretability of neural network models and being used to achieve significant advancements in pattern recognition [26].

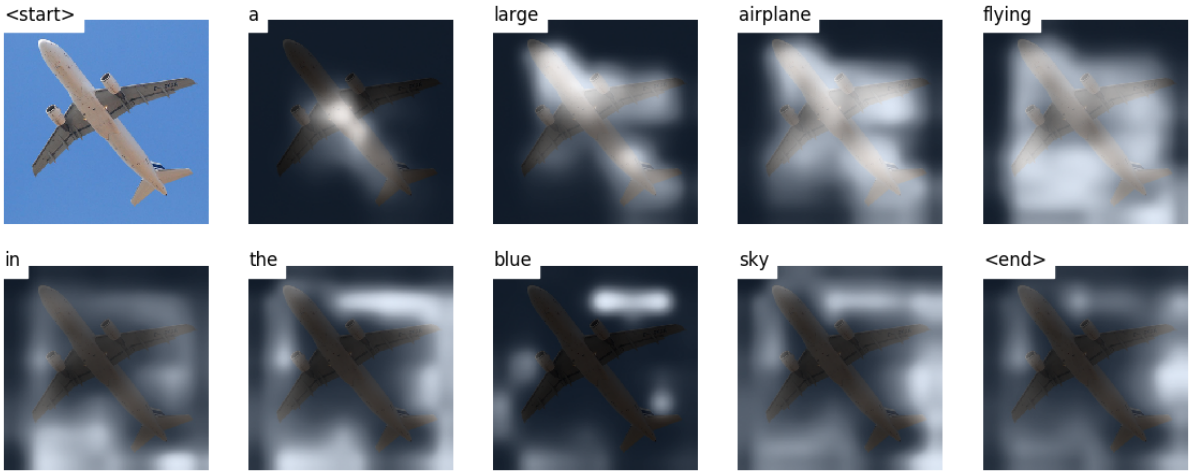


Figure 2.9: Given the input image in the top left corner, the attention weights calculated by an image captioning seq2seq model for each word in the output *"a large airplane flying in the blue sky"*; taken from [157].

The aim for an attention mechanism is to produce a vector of weights $\boldsymbol{w} \in \mathbb{R}^M$, such that each weight w_i represents the importance of the vector value $\boldsymbol{v}_i$ in a sequence of values $\boldsymbol{V} \in \mathbb{R}^{M \times D}$. A weight of 0 removes that value completely; the larger the weight, the more information is kept of that value. Ideally, this attention weight vector is a discrete probability distribution (i.e. $\sum_i w_i = 1$ and $0 \leq w_i \leq 1, \forall w_i \in \boldsymbol{w}$). These weights and values are combined in a weighted sum to produce an attended vector $\boldsymbol{v'} \in \mathbb{R}^D$ that captures all of the most important information (Equation 2.31; where $\times$ is the element-wise product).

$$\boldsymbol{v'} = \sum_{i=1}^{M} \boldsymbol{w} \times \boldsymbol{v}_i \tag{2.31}$$

Generally, an attention mechanism calculates the weights w_i as per Equation 2.32. There is a *query* vector $\boldsymbol{q} \in \mathbb{R}^D$, and a sequence of *key* vectors $\boldsymbol{K} \in \mathbb{R}^{M \times D}$; however, $\boldsymbol{K} = \boldsymbol{V}$ usually. A vector similarity function $S : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ is applied on every $\boldsymbol{k_i}$ with the query. The output of this is converted to a probability distribution via the $softmax$ function (Equation 2.8). S can be a fixed function (such as the dot-product or the cosine similarities) or be learnt via a dedicated network. $\boldsymbol{K}$, $\boldsymbol{V}$ and $\boldsymbol{q}$ are the inputs.

$$w_i = \frac{e^{S(\boldsymbol{q}, \boldsymbol{k}_i)}}{\sum_{j=1}^{T} e^{S(\boldsymbol{q}, \boldsymbol{k}_j)}} \tag{2.32}$$

Note that there are no trainable parameters within an attention mechanism; it simply takes queries, keys and values as input. In practice, these inputs are the feature vectors extracted by other components of a larger neural architecture. For features that share some semantic dependency, the implicit goal of the architecture is to extract similar vector representations for them. This similarity ensures that the attention mechanism calculates a large attention weight. The opposite is the case for unrelated features.

There are several ways to interpret the attention mechanism. One way is as a fuzzy feature look-up method: for a given query feature, it searches a set of candidate features for the most similar ones and produces an amalgamation of them. Or, as the attention weights are a probability distribution, it can be seen as taking the expectation of the values given a query. Another interpretation of attention is as a convolution with an infinitely-wide[6] kernel, whose weights dynamically change based on context. A traditional convolutional layer has a finite kernel that is only ever applied to a small region of the input at a time, with weights that are fixed outside of training.

2.4.2 Sequence-to-Sequence

As mentioned in Section 1.3.2, seq2seq models use attention. In fact, attention mechanisms were first proposed as an extension to the original seq2seq design [8]. Previously, the recurrent encoder of such architectures only produced a single feature vector — the final hidden state of its internal RNN [143]. However, this fixed-sized context was found to be a bottleneck in performance, especially for longer input sequences. With attention, meanwhile, the seq2seq encoder emits a feature vector for every input element (i.e. all RNN hidden states). During the auto-regressive decoding process, the last output element of the decoder is used as the query vector in the attention mechanism; which is applied to the set of encoder feature vectors to select the most relevant information to use in the next decode step (Figure 2.9).

By modelling the dependencies in a sequence, an attention mechanism also implicitly learns the alignments between input and output elements [8]. As previously discussed (Section 1.4), the alignments in a seq2seq model can be between elements in any position (Figure 2.10); whereas alignments under CTC are strictly monotonic. However, as already argued, this is not an issue for our task.

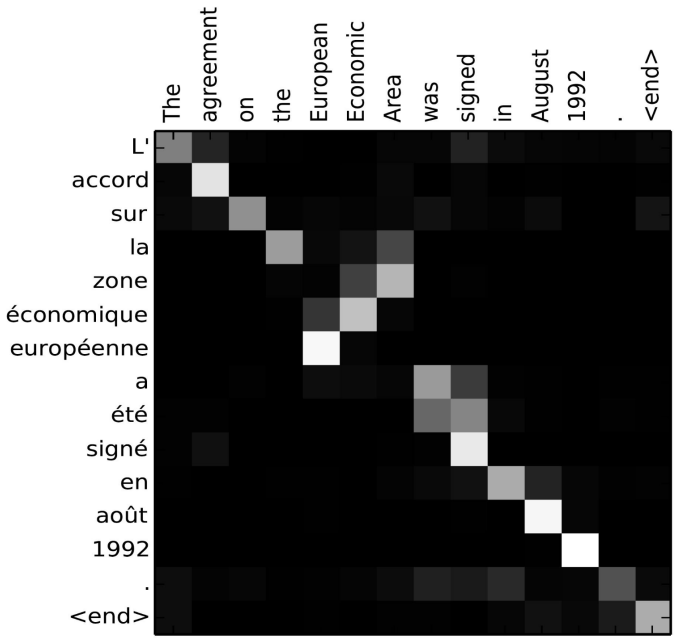


Figure 2.10: For the input `The agreement on the European Economic Area was signed in August 1992.`, a visualisation of the attention weights calculated between its words and the words in the output `L'accord sur la zone économique européenne a été signé en août 1992.` by a seq2seq model for English-French translation; taken from [8]. Note the non-monotonic alignment.

We formalise the general structure of a seq2seq model as per Vaswani et al [148]. The encoder maps an input sequence of vectors $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_M)$ to an intermediate sequence of features $\boldsymbol{Z} = (\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_M)$. Given $\boldsymbol{Z}$, the decoder autoregressively generates an output sequence $\hat{\boldsymbol{Y}} = (\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, ..., \hat{\boldsymbol{y}}_{M'})$ one vector at a time; at each time step t, consuming its last output element $\hat{\boldsymbol{y}}_{t-1}$ as an additional input. While there are many ways to incorporate attention into this structure, the original seq2seq architecture with attention [8] employs it between the encoder and decoder; using query $\boldsymbol{q} = \hat{\boldsymbol{y}}_{t-1}$, keys/values $\boldsymbol{K} = \boldsymbol{V} = \boldsymbol{Z}$ and the dot-product similarity function $S(\boldsymbol{q}, \boldsymbol{k}_i) = \boldsymbol{q} \cdot \boldsymbol{v}_i = \boldsymbol{q}^T \boldsymbol{k}_i$.

[6] Infinite in the sense that it operates on the entire input sequence at once, and that we can increase the length of the sequence as much as we want without changing the mechanism.

2.4.3 The Transformer

The *Transformer* [148] takes attention to its logical extreme: it processes a sequence without any recurrent or convolutional networks, instead by repeatedly *transforming* it through attention mechanisms. Transformers follow the same general seq2seq structure, but remove the attention mechanism in-between the encoder and decoder (Figure 1.7); the entire sequence of features is passed on. Attention is now used within the decoder and encoder themselves.

More specifically, the Transformer uses stacks of *self-attention networks* (SAN) and *mutual attention networks* (MAN) (Figure 2.11); however, due to their similarity, we will continue to refer to both types as SAN. SANs themselves consist of *multi-head attention components* (MHA; Section 2.4.3.1), *feed-forward networks* (FFN; Section 2.4.3.2), *residual connections* (Section 2.4.3.3) and *layer normalisations* (LN; Section 2.4.3.4). Also, all inputs to the encoder and decoder are first embedded with a *positional encoding* (PE; Section 2.4.3.5). These components are each formalised in the following sections.

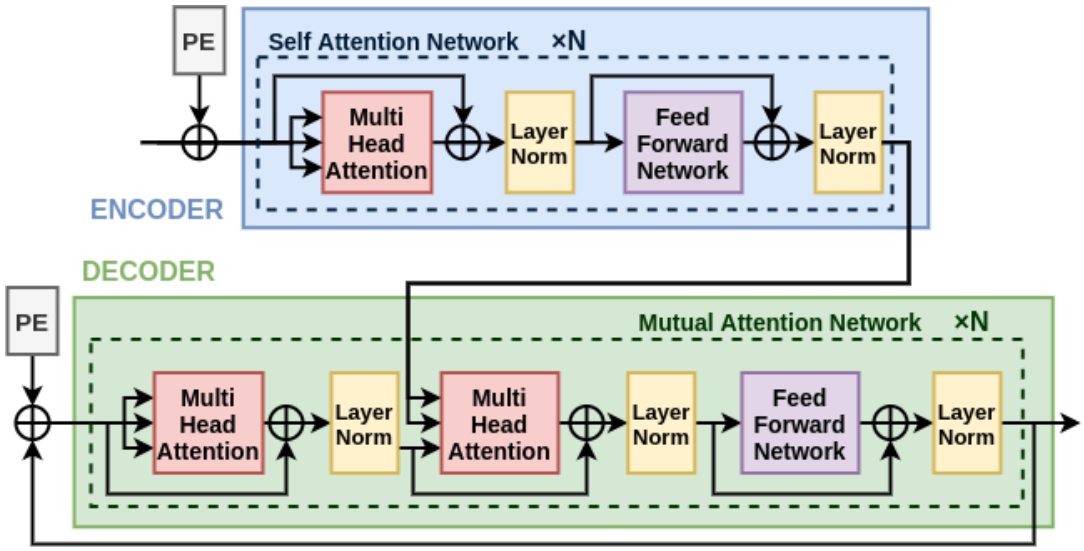


Figure 2.11: The Transformer architecture, consisting of SANs.

2.4.3.1 Multi-Head Attention

The *multi-head attention* component (MHA) is how a SAN implements attention. The particular attention mechanism it uses is called the *scaled dot-product attention* (SDPA; Equation 2.33, where T is the transpose). Like the mechanism used by the original attentional seq2seq model [8], SDPA uses the dot-product similarity function; however, it scales its attention weights based on the dimensionality of the vectors D[7]. In addition, it generalises the mechanism to use multiple queries at once: it compares each vector in sequence $\boldsymbol{Q} \in \mathbb{R}^{M \times D}$ to all keys in $\boldsymbol{K} \in \mathbb{R}^{M \times D}$, producing a $M \times M$ matrix of attention weights. Applying these weights to the values $\boldsymbol{V} \in \mathbb{R}^{M \times D}$ produces a M-length sequence of attention vectors, rather than just one. Therefore, SDPA reorganises the information in $\boldsymbol{V}$ in a way that reflects the semantic structure of $\boldsymbol{K}$ relative to $\boldsymbol{Q}$. This is how a SAN is able to model sequence dependencies without recurrency.

$$SDPA(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{D}})\boldsymbol{V} \tag{2.33}$$

[7]This is to prevent the dot product from becoming too large. Assuming that the elements of $\boldsymbol{q}$ and $\boldsymbol{k}$ are independent random variables with mean 0 and variance 1, their dot product $\boldsymbol{q} \cdot \boldsymbol{k} = \sum_{i=1}^{D} q_i k_i$ has mean 0 and variance D. Multiplying the output by $\frac{1}{\sqrt{D}}$ counteracts this.

SANs are *self-attentional* because the same input sequence is used for the queries, keys and values (i.e. $\boldsymbol{Q} = \boldsymbol{K} = \boldsymbol{V}$) in their MHA layers. MANs are almost identical to SANs, but are distinguished by an additional MHA layer which takes its queries and keys from the output of the decoder (Figure 2.11). This is how information is shared between the encoder and decoder of the Transformer, and allows each output element to be conditioned on any decoder feature. This is somewhat reminiscent of the attention layer in a seq2seq model (Section 1.3.2).

As its name suggests, MHA employs SDPA in a multi-headed fashion (Figure 2.12). That is, MHA projects its inputs into a number of different representational subspaces — attending to the information in each subspace individually — before combining it all into a single output. Thus, MHA provides multi-modal attention: its different heads each learn to model a different type of dependency. In the translation task of the original Transformer paper [148], for example, some attention heads were shown to focus on local groups of words; whereas others seemed to capture syntactic dependencies between distant words (Figure 2.13).

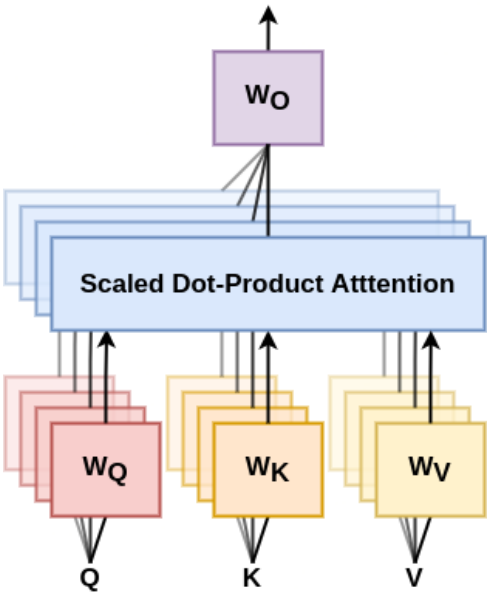


Figure 2.12: The multi-head attention component, consisting of multiple parallel heads of scaled dot-product attention.

Speaking more formally, an MHA layer has H many heads, each of dimension $D_H = \frac{D}{H}$, and provides each one the same input sequences. Each $head_i$ projects its inputs into separate subspaces using weights $\boldsymbol{W_Q}^{(i)} \in \mathbb{R}^{D \times D_H}$, $\boldsymbol{W_K}^{(i)} \in \mathbb{R}^{D \times D_H}$ and $\boldsymbol{W_V}^{(i)} \in \mathbb{R}^{D \times D_H}$; before applying SDPA[8] (Equation 2.35). MHA concatenates the output of each head $head_i$ and projects them with a set of output weights $\boldsymbol{W_o} \in \mathbb{R}^{D \times D}$ (Equation 2.34; where [...] represents concatenation).

$$MHA(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = [head_1(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}), head_2(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}), ..., head_H(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})]\boldsymbol{W_o} \tag{2.34}$$

$$head_i(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = SDPA(\boldsymbol{Q}\boldsymbol{W_Q}^{(i)}, \boldsymbol{K}\boldsymbol{W_K}^{(i)}, \boldsymbol{V}\boldsymbol{W_V}^{(i)}) \tag{2.35}$$

[8] In this case, the dimensionality of the vectors in SDPA are D_H not D.

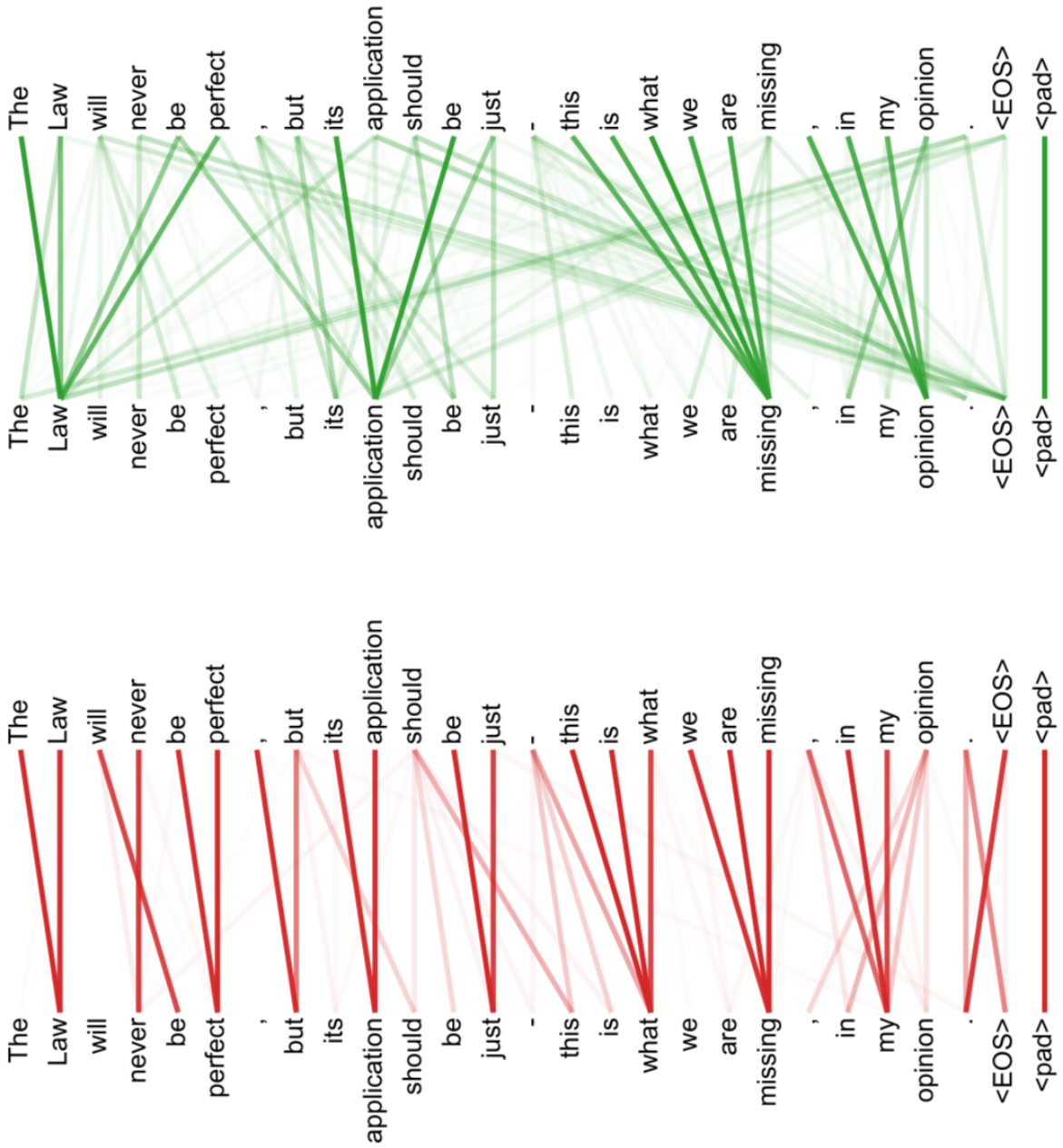


Figure 2.13: A visualisation of the self-attention weights calculated for a sequence of words by two different heads in the same SAN layer of a Transformer encoder; taken from [148]. The opacity of a line represents the weight between the two words it connects. Each head has clearly learnt to model a different type of relationship, with some seemingly capturing real syntactic dependencies.

2.4.3.2 Feed-Forward Network

While the MHA layers of a SAN reorganise information across the entire sequence, the *feed-forward networks* (FFN) consolidate the information for each element. This is where the actual nonlinear transformation happens, taking into account the dependencies of the entire sequence. An FFN is implemented as per Equations 2.2 and 2.3; with input and output sizes of D, a hidden size of F, and a $ReLU$ activation on the hidden layer. There is no activation function for the output layer (i.e. $g_2 = id$). The FFN is applied on an attended sequence position-wise; transforming each vector individually but with the same set of weights each time.

The hidden size F is considerably larger than the input and output size D; usually $4\times$ larger [148]. This is unusual, and the original authors of the Transformer do not provide the intuition behind this. Such an arrangement is somewhat reminiscent of *sparse autoencoders* [114]. These architectures also consist of FNNs with large hidden dimensions and these hidden layers learn to encode their input features sparsely[9]. It has been demonstrated that learning representations in a way that encourages sparsity helps to improve performance at classification tasks [103]. However, encouraging sparsity also requires special penalisation terms in the loss function during training [114]; the Transformer authors do not do this [148]. What is certain about these FFNs is that they transform their inputs by first projecting them to some higher representational space.

2.4.3.3 Residual Connection

When training very deep neural networks, such as the Transformer, a degradation problem has been observed. As you increase depth, the model's test performance begins to plateau and then degrades rapidly [141]. Surprisingly, this is not due to overfitting — the training performance similarly degrades. *Residual connections* have been proposed as a solution to this problem. Also known as skip connections, they simply involve adding the original input x for a series of layers L to their output $L(x)$ (Figure 2.14). Such connections improve the flow of information through networks, and their use has enabled significant advancements in computer vision [69].

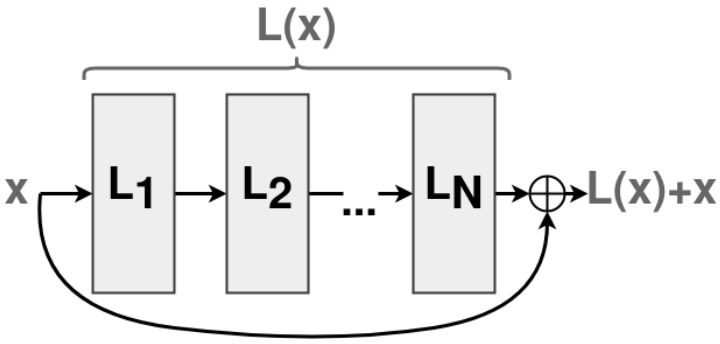


Figure 2.14: A residual connection that skips over N neural network layers.

[9] A neural network layer is considered sparse when only a small number of their neurons are ever strongly activated. The vast majority of neurons will have close to no activation at any time.

2.4.3.4 Layer Normalisation

Layer normalisation (LN) [7] is very similar to batch normalisation (BN; Section 2.1). However, instead of approximating the normalisation parameters μ_i and σ_i for each neuron over the batch, LN normalises does it for all neurons of a layer for each batch sample (Figure 2.15). This is essential for RNNs and Transformers, which process variable length inputs [7, 148]. BN is incompatible with these architectures; if samples are of different lengths, then the approximations at some time steps become inconsistent.

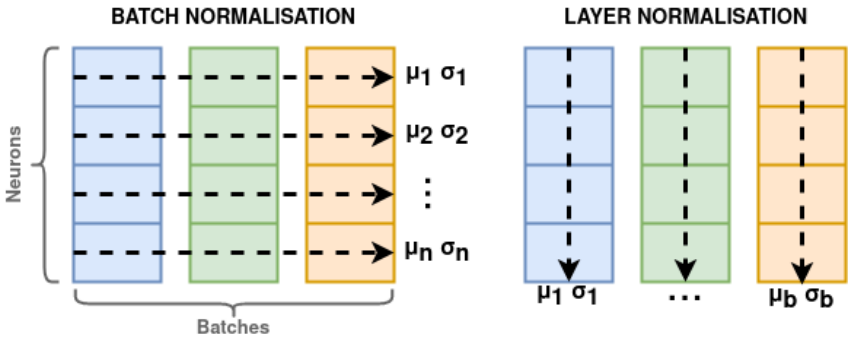


Figure 2.15: A simplified demonstration of the differences between batch normalisation and layer normalisation; where n is the size of the layer and b is the batch size.

2.4.3.5 Positional Encoding

Self-attention is inherently content-based: it compares sequence elements based on the information they hold. This is done regardless of the elements absolute or relative positions. Therefore, in order for a SAN to model positional dependencies, this information has to be embedded into its input sequences. The Transformer adds such a positional encoding $\boldsymbol{PE} \in \mathbb{R}^{M \times D}$ to the inputs of its encoder and decoder (Figure 2.11). While $\boldsymbol{PE}$ can be optimised during training as an additional set of parameters, the original authors of the Transformers define each of its values based on the sine and cosine functions (Equation 2.36 and Figure 2.16). Both methods achieve comparable results, however, the sinusoidal method was chosen because it can be easily extended for larger inputs; whereas a learned encoding has to be optimised again [148].

$$\begin{aligned} \boldsymbol{PE}_{m,2d} &= sin(m^{-1} \times 10000^{\frac{2d}{D}}) \\ \boldsymbol{PE}_{m,2d+1} &= cos(m^{-1} \times 10000^{\frac{2d}{D}}) \end{aligned} \tag{2.36}$$

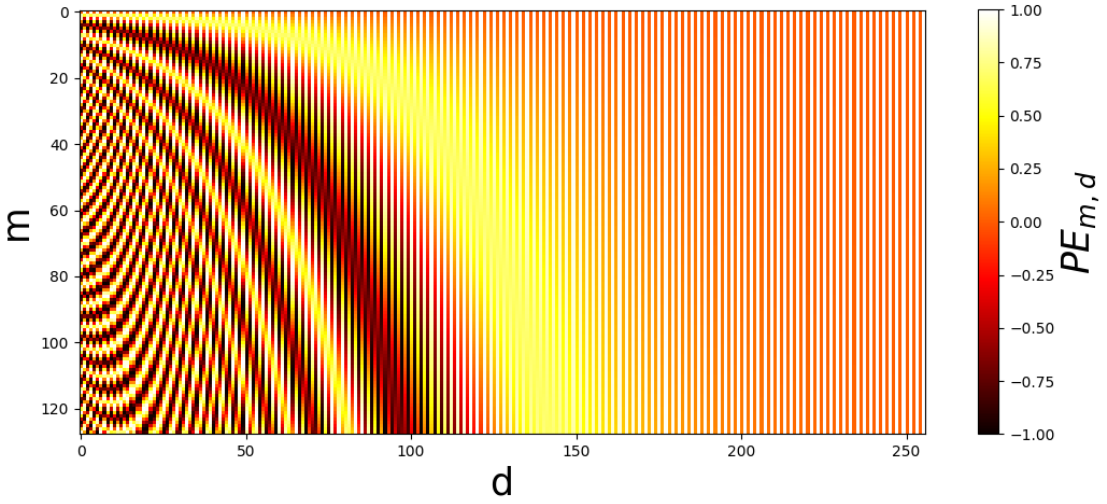


Figure 2.16: The value of the sinusoidal positional encoding $\boldsymbol{PE}_{m,d}$ (Equation 2.36) for sequence positions $1 \leq m \leq 128$ and vector dimensions $1 \leq d \leq 256$.

2.5 State-of-the-art Optical Models

As previously discussed (Section 1.2), the current state-of-the-art optical models follow an encoder-decoder design and are composed of recurrent and convolutional neural networks (Sections 2.2 and 2.3). Moreover, they are trained and decoded using the CTC method. In this section, we first present CTC as a general model framework; before identifying and formalising some state-of-the-art architectures. We refer to each architecture presented by the surname of its primary author.

2.5.1 CTC Framework

Connectionist temporal classification (CTC) [59] is a popular framework for training neural network models at sequence transduction tasks, where the alignment between inputs and outputs is unknown; such as in HTR. This lack of a predetermined alignment makes scoring a model non-trivial: there are often more input elements than output ones[10], with many possible alignments between them. Rather than manually labelling the correct alignments, using CTC efficiently alleviates this issue in a way that is independent of the underlying model architecture. We formalise CTC in the context of an encoder-decoder optical model for HTR.

Consider an input image $\boldsymbol{X} \in \mathbb{Z}^{W \times H \times C}$ with a corresponding ground-truth sequence of characters $\boldsymbol{y} = (y_1, y_2, ..., y_{M'}) \in \mathcal{A}^{M'}$ from the alphabet $\mathcal{A}$. The optical models encoder first maps the input to an M-length input sequence of D-dimensional features $\boldsymbol{Z} \in \mathbb{R}^{M \times D}$. The length of the ground-truth sequence is at most that of the features, i.e. $M' \leq M$. Given these features, the end goal of the decoder is to generate an output sequence of characters $\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_{\hat{M}}) \in \mathcal{A}^{\hat{M}}$, where $\hat{M} \leq M'$.

Under the CTC framework, we assume an intermediate alphabet $\mathcal{A}' = \mathcal{A} \cap \{\texttt{-}\}$ with an extra symbol "–" called *blank*. Now, the aim of the decoder is to generate a sequence of probability distributions[11] $\boldsymbol{\Pi} = (\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, ..., \boldsymbol{\pi}_{M'}) \in \mathbb{R}^{M' \times |\mathcal{A}'|}$ over the intermediate alphabet. The blank symbol represents the invisible gap between characters; it can be inserted arbitrarily at any position, but must always be included between any intended duplicate characters.

A *path* $\boldsymbol{y'} = (y'_1, y'_2, ..., y'_{M'}) \in \mathcal{A}'^{M'}$ is a sequence of intermediate characters and is associated with a sequence of output characters $\hat{\boldsymbol{y}}$ via a many-to-one mapping $\mathcal{B} : \mathcal{A}'^{M'} \rightarrow \mathcal{A}^{\hat{M}}$. In the mapping, blank symbols are removed and any repeated characters without a blank in between them are collapsed into one. In this way, a path can be regarded as a possible alignment for an output sequence and there are many possible paths for the same output; for example:

$$\mathcal{B}(\texttt{-h-e-l-l-o-}) = \mathcal{B}(\texttt{h-eee-l-ll-o}) = \mathcal{B}(\texttt{-h--e-l-lo--}) = \texttt{hello}$$

CTC itself models the probability of an output $\hat{\boldsymbol{y}}$ given an input $\boldsymbol{X}$ by marginalizing over all paths $\boldsymbol{y'}$ which map to that output (Equation 2.37), and by assuming that the elements of a path $\hat{y}_i$ are conditionally-independent (Equation 2.38).

$$P(\hat{\boldsymbol{y}}|\boldsymbol{X}) = \sum_{\boldsymbol{y'} \in \mathcal{B}^{-1}(\hat{\boldsymbol{y}})} P(\boldsymbol{y'}|\boldsymbol{X}) \tag{2.37}$$

$$P(\boldsymbol{y'}|\boldsymbol{X}) = \prod_{m=1}^{M'} P(y'_m, m|\boldsymbol{X}) \tag{2.38}$$

$P(y'_m, m|\boldsymbol{X})$ is approximated using the decoders output probabilities $\boldsymbol{\Pi}$, i.e. $P(y'_m, m|\boldsymbol{X}) \approx \boldsymbol{\pi}_m$. Optical models can be trained end-to-end to minimize the CTC loss function (Equation 2.39), as well as use CTC to decode their outputs (Equation 2.40).

$$\mathfrak{L}_{CTC}(\boldsymbol{y}, \boldsymbol{X}) = -\log P(\boldsymbol{y}|\boldsymbol{X}) \tag{2.39}$$

$$\hat{\boldsymbol{y}} = \underset{\hat{\boldsymbol{y}} \in \mathcal{A}^{\hat{M}}}{\text{argmax}} P(\hat{\boldsymbol{y}}|\boldsymbol{X}) \tag{2.40}$$

[10] For example, in HTR, there are usually many more columns of pixels in the input image than there are characters in its corresponding transcription

[11] Usually achieved using a softmax activation in the decoders final layer (Section 2.1.3)

2.5.2 Architectures

The current state-of-the-art optical models for HTR use an CNN-RNN encoder-decoder architecture that is trained using CTC (Sections 1.2 and 2.5.1). The role of the encoder is to extract features from images of handwriting, while the decoder models the dependencies between features in order to generate a transcription character-by-character.

More formally, matrices of pixel values $\boldsymbol{X} \in \mathbb{Z}^{W \times H \times C}$ are the inputs to the network; where W is the image width, H the height and C the number of colour channels. An input image is first propagated through an encoder block of convolutional layers to produce an M length sequence of two-dimensional features maps $\boldsymbol{H} \in \mathbb{R}^{M \times a \times b}$. These maps are flattened $\mathcal{R} : \mathbb{R}^{M \times a \times b} \rightarrow \mathbb{R}^{M \times D}$ to produce a sequence of feature vectors $\boldsymbol{Z} = \mathcal{R}(\boldsymbol{H}) \in \mathbb{R}^{M \times D}$; where $D = a \times b$.

Next, the feature sequence $\boldsymbol{Z_1}$ is processed by the recurrent layers of the decoder. These layers each have a total of D hidden units, and the final output of the recurrent block is a sequence of RNN hidden states $\boldsymbol{Z_2} \in \mathbb{R}^{M \times D}$. Finally, the hidden states are projected position-wise to probability distributions $\boldsymbol{\Pi} \in \mathbb{R}^{M \times |\mathcal{A}'|}$ over the intermediate alphabet $\mathcal{A}'$ using a dense layer of size $|\mathcal{A}'|$ and a softmax activation. These distributions are passed onto CTC to either decode a predicted transcription $\hat{\boldsymbol{y}} \in \mathcal{A}^{\leq M}$; or with the ground-truth transcription $\boldsymbol{y} \in \mathcal{A}^{\leq M}$ to calculate the loss.

2.5.2.1 Puigcerver Architecture

In 2017, J. Puigcerver proposed [127] an optical model architecture that uses traditional convolutional layers (Section 2.3) and one-dimensional, bi-directional LSTMs (BLSTM; Section 2.2.2). He claimed that such an architecture CAN achieve the same level of performance as the multi-dimensional LSTM-based architectures of previous literature [64, 97, 109]; whilst being computationally faster. We firmly believe that this *Puigcerver* architecture represents the current state-of-the-art in HTR optical models. Firstly, the highly-successful Transkribus platform (Section 1.1) uses it to provide its HTR service [108, 153]. Secondly and most importantly, Puigcerver has now become the standard baseline in HTR literature to benchmark against [6, 24, 29, 48, 49, 50, 79, 101, 136, 145]. The Puigcerver architecture is defined as follows.

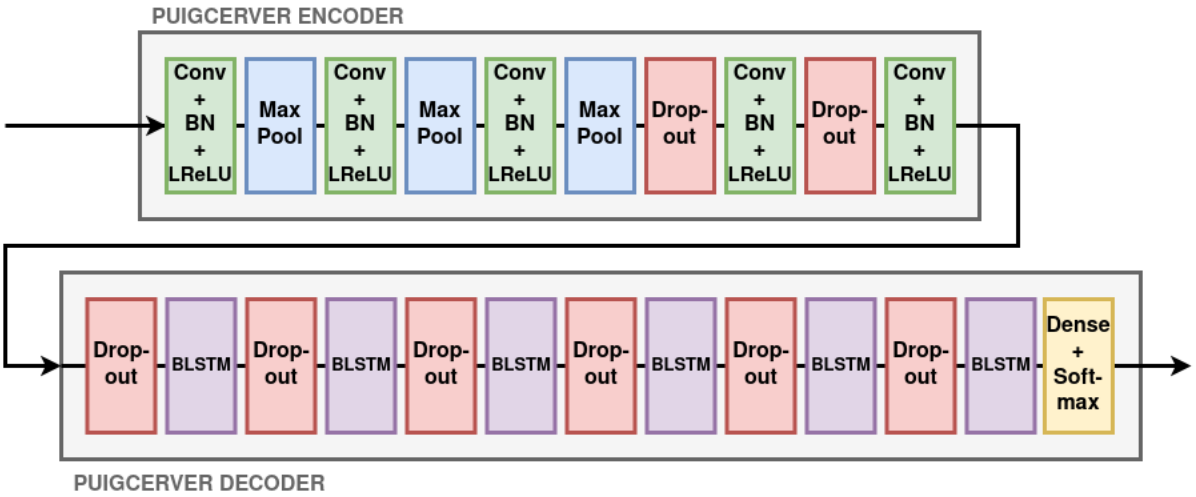


Figure 2.17: The Puigcerver architecture [127]; with a convolutional encoder and an LSTM-based decoder.

The encoder consists of five sets of convolutional (Conv), batch normalisation (BN) and leaky ReLU activation (LReLU) layers. The first three sets are each followed by a max pooling (maxpool) layer, while dropout layers precede the last two. The LReLUs use $\alpha = 0.001$, the dropouts use a rate of 0.2 and the maxpools have 2×2 kernels. The encoder Conv layers are configured as per Table 2.1.

Meanwhile the decoder has six pairs of dropout and BLSTM layers, followed by another dropout layer and a position-wise dense layer with a softmax activation. The dropouts in the decoder use a rate of 0.5, the BLSTMs have 256 hidden units in each direction and the output dense layer is of the size of the intermediate alphabet $|\mathcal{A}'|$. In total, the Puigcerver has 9.6 million parameters, which are initialized at the start of training with the Glorot method (Section 2.1.4).

Layer	Kernel	Stride	Filters
Conv 1	3×3	1×1	16
Conv 2	3×3	1×1	32
Conv 3	3×3	1×1	48
Conv 4	3×3	1×1	64
Conv 5	3×3	1×1	80

Table 2.1: The configuration of the convolutional layers in the Puigcerver encoder.

2.5.2.2 Flor Architecture

More recently, in 2020, A. Flor et al [50] proposed an architecture that builds upon the Puigcerver design. It is claimed to match the performance of Puigcerver whilst having $0.1\times$ the number of parameters. The *Flor* architecture (Figure 2.18) achieves this by, in the encoder, incorporating gated convolutional layers (Gated Conv; Section 2.3.2) and, in the decoder, replacing BLSTM layers with bi-drectional GRUs (BGRU; 2.2.3). The author justifies this change based on the fact that GRUs generally achieve the same performance as LSTMs, but require fewer parameters (Section 2.2.3). Moreover, they argue that using gated convolutions results in better encoder features, therefore fewer recurrent layers are required in the decoder; further reducing the total number of parameters [50].

Other attempts to improve on Puigcerver have included adapting it to the seq2seq paradigm [29] and extending its internal LSTMs with additional memory cells [115]. However, for the most part, these only provide minimal performance boosts while incurring high training and computational costs. Flor appears to be one of the only attempts that both matches the Puigcerver performance and reduces its training/computational complexity. Moreover, the Flor architecture is an attractive baseline choice for this dissertation because of its use of GRUs. Our aim is to evaluate if self-attention networks can replace RNNs in optical models (Section 1.5), so it is important to compare them to RNN variants other than the LSTM.

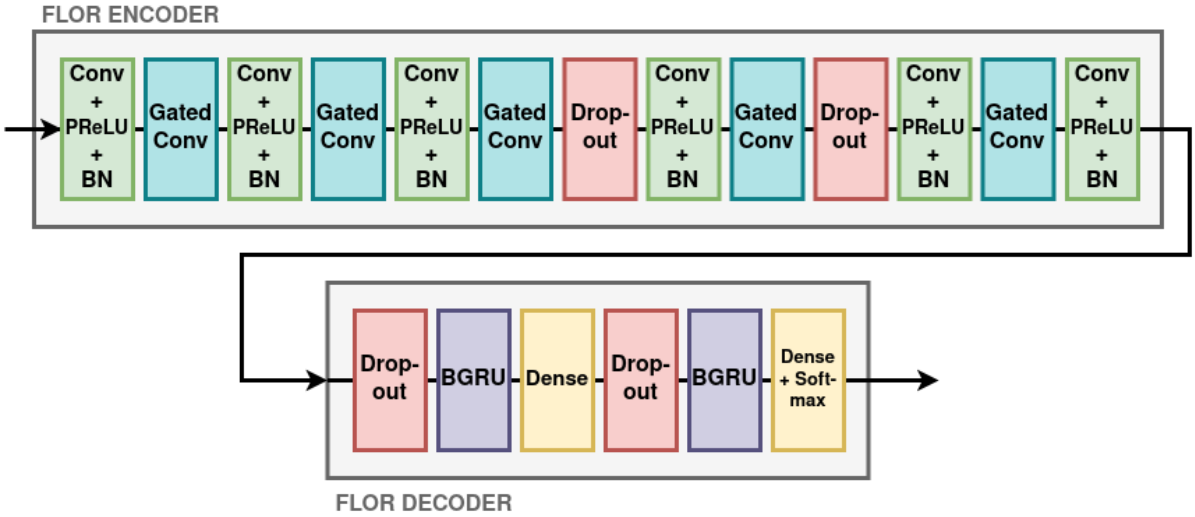


Figure 2.18: The Flor architecture [50]; with a gated convolutional encoder and a GRU-based decoder.

The Flor encoder has six Conv layers interleaved with five Gated Conv layers. Each Conv layer is applied with a *parametric* ReLU activation[12] (PReLU) and batch normalisation (BN). The fourth and fifth Conv layers are each preceded by a dropout layer. The dropout layers use a drop rate of 0.2, while the configurations of the convolutional layers are listed in Table 2.2. Whereas, the decoder consists of two sets of dropout, BGRU and dense layers. The dropouts use a rate of 0.2, while the BGRUs have 128 hidden units in each direction. The first dense layer is of size 256 and has no activation function; the second is of size $|\mathcal{A}'|$ and has a softmax activation. In total, Flor has 920 thousand parameters and its weights are initialized according to the He method (Section 2.1.4).

[12]Parametric ReLU [70] is identical to Leaky ReLU (Section 2.1.3), however the hyperparameter α is now a trainable parameter.

Layer	Kernel	Stride	Filters
Conv 1	3×3	1×2	16
Gated Conv 1	3×3	1×1	16
Conv 2	3×3	1×1	32
Gated Conv 2	3×3	1×1	32
Conv 3	2×4	2×4	40
Gated Conv 3	3×3	1×1	40
Conv 4	3×3	1×1	48
Gated Conv 4	3×3	1×1	48
Conv 5	2×4	2×4	56
Gated Conv 5	3×3	1×1	56
Conv 6	3×3	1×1	64

Table 2.2: The configuration of the convolutional layers in the Flor encoder.

Note that, unlike traditional CNNs (Section 2.3), Flor does not use pooling layers but, instead, there are some convolutional layers with larger strides than others (Table 2.2). Larger-strided convolutions can be used to down-sample feature maps while also extracting features. It has been shown that the pooling layers of a CNN can be replaced with such convolutions with no loss in accuracy [138].

2.6 SANscript

In this section, we propose the SANscript architecture: an optical model that uses self-attention networks instead of recurrent neural networks in its decoder. We motivate its design and hypothesise how its performance will compare to the current state-of-the-art.

2.6.1 Motivation

The take-away message of the original Transformer paper [148] is that SANs can replace RNNs for sequence modelling (Section 2.4.3). If it can be done in a seq2seq architecture such as the Transformer, then maybe the same can be done in an encoder-decoder architecture such as Flor or Puigcerver. However, this poses several problems. Firstly, the Transformer does not use CNNs; unlike optical models. Secondly, the Transformer was originally evaluated on purely linguistic tasks; optical models have to model both visual and linguistic data. Lastly, the Transformer was not trained using CTC; it is not clear if doing so is tractable for a SAN-based optical model.

Fortunately, the compatibility of CNNs with Transformers (in turn, with SANs) is well documented in more recent literature. They have been combined successfully in models for visual tasks such as image captioning [156, 160], image classification [37, 85], image object detection [25, 165], video captioning [76, 164], video action labelling [54] and scene text recognition [17, 44, 151]. There is even a CNN-Transformer architecture that is claimed to achieve state-of-the-art results at offline HTR [79]. By contrast, the available literature on the training of attention-based networks using CTC is limited and primarily in the context of automatic speech recognition (ASR). Most such *acoustic* models only employ CTC as an auxiliary loss function and still heavily rely on RNNs [33, 84, 162]. Despite this, J. Salazar et al propose a nonrecurrent, SAN-based acoustic model that is trained end-to-end with CTC [133].

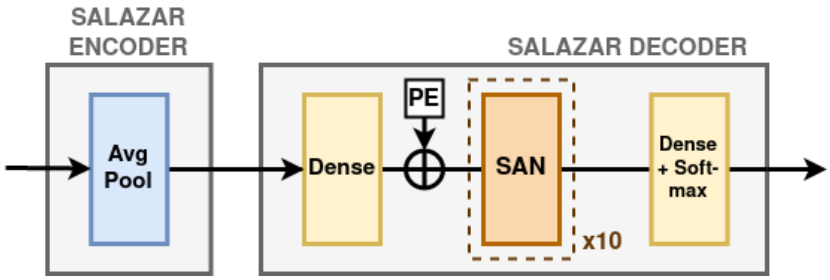


Figure 2.19: The Salazar architecture for ASR [133]; an encoder-decoder acoustic model with an entirely SAN-based decoder that is trained end-to-end with CTC.

This work, which we refer to as the *Salazar* architecture, is the main inspiration for our proposed optical model, SANscript. We take its results as evidence that training a SAN-based architecture with CTC is possible. As discussed in Section 1.4, the forms of data and model architecture used for ASR and for HTR are closely related; therefore, it is sound to base an optical model on an acoustic one.

The Salazar architecture (Figure 2.19) is briefly defined as follows. The encoder contains only an average pooling (avgpool; Section 2.3) layer to down-sample input sequences; however, the authors have suggested incorporating convolutional layers as a possible improvement [133]. The decoder has a position-wise dense layer, an additive sinusoidal positional encoding (Section 2.4.3.5), a stack of ten SANs and finally a softmax dense layer.

2.6.2 Proposed Architecture

Inspired by the Salazar acoustic model (Section 2.6.1), we propose the *SANscript* optical model: an encoder-decoder-CTC architecture for HTR based on self-attention networks. This architecture is entirely nonrecurrent, unlike the current state-of-the-art (Section 2.5). We believe that this the first attempt at such a design within HTR research, and fills a gap in the literature. SANscript follows the general structure presented in Section 2.5.2, but replaces the RNNs with SANs. More specifically, it has a convolutional encoder and self-attentional decoder. In order, the decoder (Figure 2.6) consists of:

- a dense layer (Section 2.1) of size D,
- a sinusoidal positional encoding (PE; Section 2.4.3.1) of size $M \times D$,
- N-many pairs of dropout (Section 2.1.4) and SAN layers, and
- a softmax dense layer (Section 2.1.3) of size $|\mathcal{A}'|$.

Dropout is applied with rate R and $|\mathcal{A}'|$ is the size of the intermediate CTC alphabet (Section 2.5.1). While the Salazar architecture does not use dropout, we decided to include it in SANscript because of its use by Flor and Puigcerver. We define a SAN (Figure 2.6) as per [133, 148]. That is:

- a multi-head attention layer (MHA; Section 2.4.3.1) with H-many heads,
- a layer normalisation (LN; Section 2.4.3.4),
- a position-wise feed-forward network (FFN; Section 2.4.3.2) with hidden size F, and
- another LN.

There are residual connections (Section 2.4.3.3) over the MHA and over the FFN. The number of SAN layers N, number of heads H, input depth D, hidden size F and dropout rate R are all hyperparameters. Meanwhile, the length of the incoming feature sequence M depends on the structure of the encoder.

Instead of proposing a specific convolutional encoder for SANscript, we choose to use the encoders from our baseline optical models (Section 2.5). Our architecture, therefore, has two variants:

- $\text{SANscript}_{\text{PUIG}}$, which uses the convolutional encoder of Puigcerver (Section 2.5.2.1); and
- $\text{SANscript}_{\text{FLOR}}$, which uses the gated convolutional encoder of Flor (Section 2.5.2.2).

By keeping the state-of-the-art convolutional encoders the same and only changing the recurrent decoders, the impact of using SANs in optical models can be more accurately evaluated.

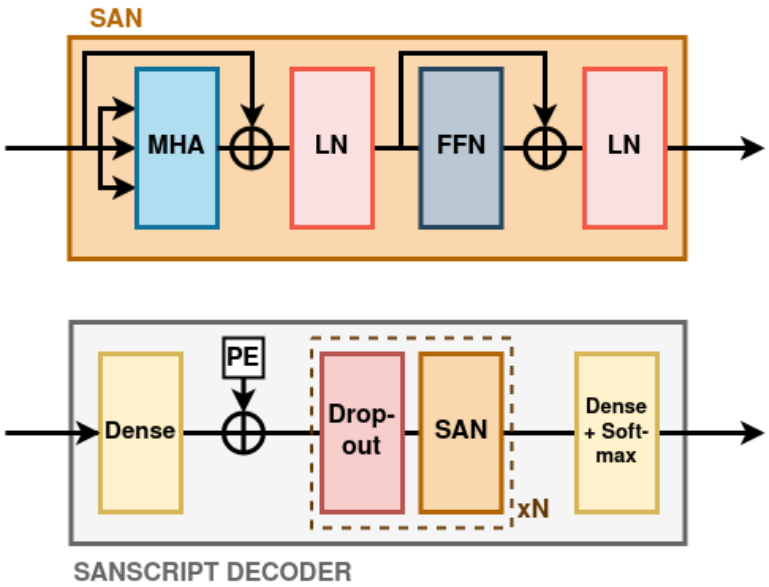


Figure 2.20: The architecture of the SANscript decoder (bottom), and that of its underlying SAN components (top).

2.6.3 Hypotheses

We hypothesized that, under the same experimental setup, both variants of SANscript would require less time to train than Flor or Puigcerver. This is because, as measured by the number of sequential steps required per input sequence, the computation of a SAN can be parallelized more than that of an RNN. As noted in Table 2.3, a self-attention layer considers all positions of the sequence in a constant number of sequential operations, while the number for a recurrent layer scales linearly. In a SAN (Sections 2.4.3.1 and 2.4.3.2), the MHA component compares entire sequences at once and the FFN is applied to each position independently (i.e. it can be applied in parallel). Meanwhile, an RNN looks at each position one-at-a-time (Section 2.2). Moreover, a SAN is generally less computationally expensive than a recurrent layer (Table 2.3): feature dimension D is often smaller than the sequence length M [148]. For example, in our subsequent experiments (Section 3.4), we used $M = 128$ and $D = 192$.

Layer type	Complexity per layer	Sequential operations
Self-Attention	$O(m^2 d)$	$O(1)$
Recurrent	$O(md^2)$	$O(m)$

Table 2.3: For different types of layer, the per-layer complexity and minimum number of sequential operations required process an M length sequence of D-dimensional features; taken from [148].

We further hypothesized that SANscript could achieve similar recognition rates as Flor and Puigcerver (the current state-of-the-art; Section 2.5), while requiring no more parameters[13] than Puigcerver (9.6 million). However, we expected that achieving this would be challenging. Both the Salazar and Transformer architectures were comparatively much larger, with 30 million and 210 million trainable parameters respectively [133, 148]. They were also trained on very large amounts of data — this is very characteristic of Transformer-based architectures. Unfortunately, public HTR datasets are notoriously limited [38, 79, 120]. For example, in our later experiments, the total amount of real handwriting data available is in the order of 10^4 samples (Section 3.1). In comparison, the seminal BERT model (Section 1.3.3) has 350 million parameters and achieves state-of-the-art language modelling performance after training on 10^9 many samples [35]. There is no precedent for the use of SANs in such a limited scope as ours. Furthermore, these architectures are known to be unstable during training; requiring careful learning rate scheduling to prevent the loss from completely diverging [133, 148]. The Flor and Puigcerver architectures, meanwhile, are trained with a constant learning rate [50, 127]. It may not be possible to optimally train all models under the same experimental setup, as desired.

[13] The number of parameters in SANscript depends on the configuration of its hyperparameters (Section 2.6.2).

Chapter 3

Project Execution

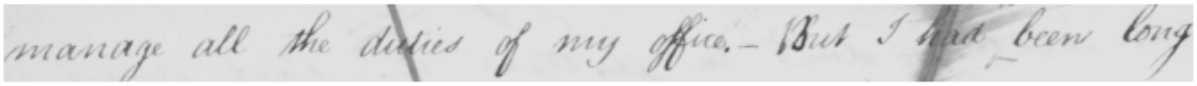
In this chapter, we outline some of the key components involved in the execution of our project. This includes the datasets (Section 3.1) and the experimental setup (Section 3.2) used for training and evaluating optical models. We also establish the baseline performance of our selected state-of-the-art models and compare these results to their original claimed performances (Section 3.3). Lastly, we perform an ablation study into the hyperparameter configurations of our proposed model, SANscript (Section 3.4). The content of this chapter is significantly less dense than that of the other chapters in this dissertation. Our aim for it is to provide an easy point-of-reference for our project implementation and decisions; all technical details are reserved to Chapter 2. Meanwhile, our main results and their evaluations are left for Chapter 4.

3.1 Datasets

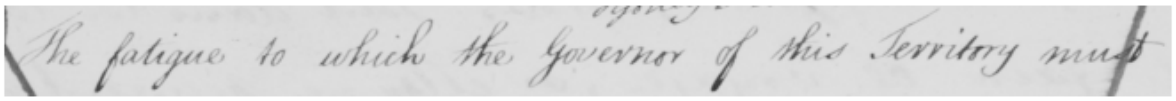
A variety of line-level, offline HTR datasets have been used in this project, each with a unique set of characteristics. We believe that it is important to benchmark all architectures on a wide range of datasets in order to get a clear understanding of each of their strengths and weaknesses. We have used both modern and historical data, and primarily of English text written in a Latin script. We use four public datasets of real handwriting (Sections 3.1.1, 3.1.2, 3.1.3, 3.1.4), however these are limited in size. As discussed in Section 2.6.3, our architecture may require larger amounts of data in order to perform well. To this end, we have created and used a large synthetic dataset (Section 3.1.5). We have also compared the characteristics of each dataset and speculated on the challenges they may pose (Section 3.1.6).

3.1.1 Bentham Dataset

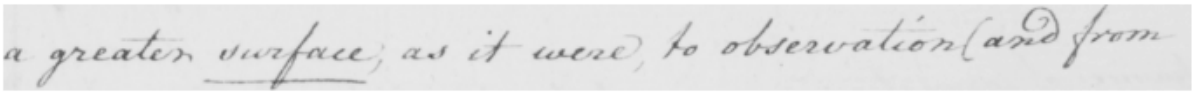
The *Bentham* dataset [52, 144], compiled by University College London, was created from manuscripts written by English philosopher Jeremy Bentham in the late 18th and early 19th Centuries. It was written cursively by a single writer. The transcriptions were crowd-sourced by volunteers. The images in the dataset are, qualitatively, more complex than those in the other datasets presented; due to the unrestricted format of the text, noisy backgrounds and many non-text pen marks (Figure 3.1).



(a) `manage all the duties of my office - But I had been long`



(b) `The fatigue to which the Governor of this Territory must`

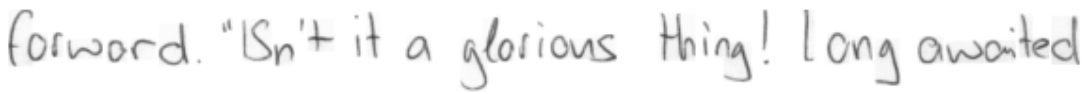


(c) `a greater surface; as it were, to observation (and from`

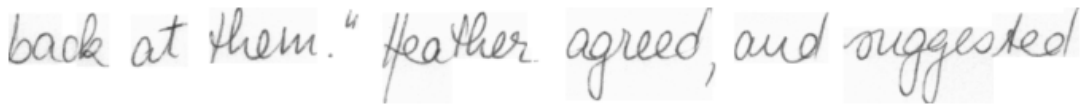
Figure 3.1: Some example images from the Bentham dataset, labelled with their corresponding ground-truth transcriptions.

3.1.2 IAM Dataset

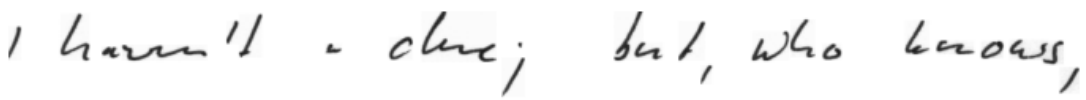
The *IAM* dataset [105] (Figure 3.2) is widely used in HTR research [19, 50, 127]. It is based on the Lancaster-Oslo-Bergen corpus, and contains contemporary English sentences written in mixed styles by over 500 writers. The provided dataset partitioning, known as the *Large Writer Independent Text Line Recognition Task*, ensures each writer only contributes to a single partition.



(a) `forward. "Isn't it a glorious thing! Long awaited`



(b) `back at them." Heather agreed, and suggested`

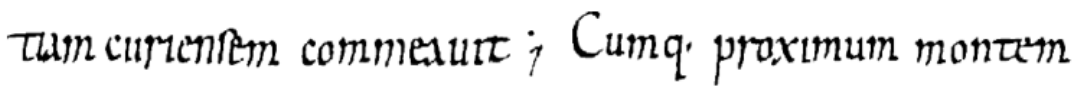


(c) `I haven't a clue; but, who knows,`

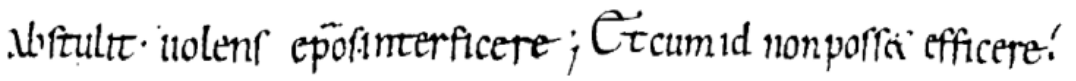
Figure 3.2: Some example images from the IAM dataset, labelled with their corresponding ground-truth transcriptions.

3.1.3 Saint Gall Dataset

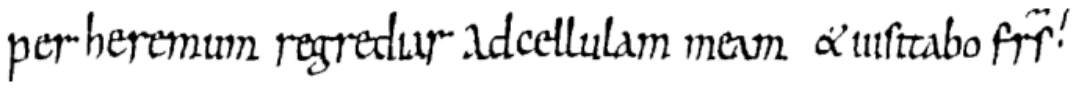
The *Saint Gall* dataset [45] is the only non-English dataset used: it holds Latin text written in Carolingian script by a single writer in the late 9th Century. It samples the Codex Sangallensis 562 manuscript; the oldest existing copy of the vitae of Saint Gall and Saint Othmar. It was written by the medieval monk Walafrid Strabo and is currently housed at the Abbey Library of Saint Gall, Switzerland. The images provided in the Saint Gall dataset already come binarised[1] and normalised; however, as noted later in Section 3.1.6, the ground-truths do not include punctuation.



(a) `tiam curiensem commeavit Cumq proximum montem`



(b) `abstulit volens epos interficere Et cum id non posset efficere`

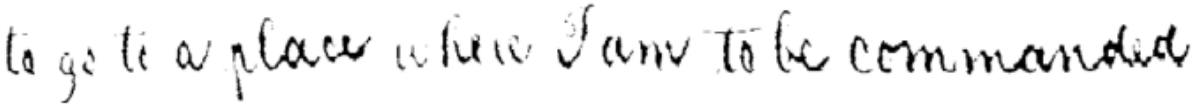


(c) `per heremum regrediar ad cellulam meam et visitabo frs`

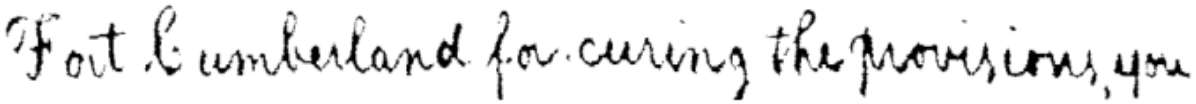
Figure 3.3: Some example images from the Saint Gall dataset, labelled with their corresponding ground-truth transcriptions.

3.1.4 Washington Dataset

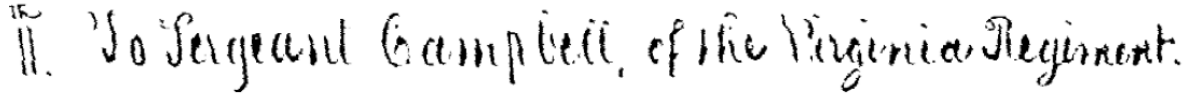
The *Washington* dataset [46] was created from the George Washington Papers at the Library of Congress, United States of America. Namely, from Series 2 Letterbook 1; which consists of recompiled letters written by George Washington in 1755 as a young colonial officer during the French and Indian War. It is written in a cursive, longhand script by a single hand. The images in the Washington dataset are also binarised and normalised.



(a) `to go to a place where I am to be commanded`



(b) `Fort Cumberland for curing the provisions, you`



(c) `11th. To Sergeant Campbell, of the Virginia Regiment.`

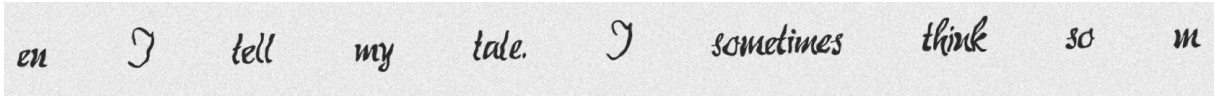
Figure 3.4: Some example images from the Washington dataset, labelled with their corresponding ground-truth transcriptions.

[1] Binarising an image involves mapping all pixels values above some threshold to 1 and the rest to 0.

3.1.5 Synthetic Dataset

One of the outcomes of this project was the creation of a large, entirely *synthetic* dataset for offline HTR, some examples of which are provided in Figure 3.5. In total, it has 50,000 labelled samples; we wished for it to have more, but we were limited by the amount of memory available during our experiments (Section 3.2). As discussed previously in Section 2.6.3 and later in Section 3.1, public HTR datasets are limited in size. We expected for our proposed model to perform best when first pre-trained on a larger dataset. This kind of improvement has been previously demonstrated on the Puigcerver [6] and Transformer [79] architectures for HTR. Unfortunately, manually annotating any substantial amount of real handwriting data is both prohibitively expensive and beyond the scope of this project. A cheaper alternative, popular in text recognition literature, is to generate this data synthetically [3, 15, 78, 79, 88].

(a) `or seek`



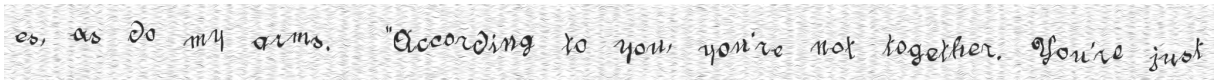
(b) `en I tell my tale. I sometimes think so m`

(c) `n recovery for up to 30 days. * Under development. I`

(d) `nd 'All the wor`

(e) `ting? D'you think he wished to war`

(f) `love sears through us like a flame, enabling us to`



(g) `es, as do my arms. "According to you, you're not together. You're just`

Figure 3.5: Some example images from our synthetic dataset, labelled with their corresponding ground-truth transcriptions.

3.1.5.1 Generating Images

Inspired by the work of L. Kang et al [79], we created our synthetic dataset by first generating grayscale images of text using handwritten-style electronic fonts and then randomly distorting them until they looked sufficiently "realistic". We used E. Belval's open-source tool, *Text Recognition Data Generator* [12], to generate the initial images from strings of text. A large public corpus of English books [86] was used as the source of text and over 300 freely-available handwriting-style fonts were collected from the Internet. We believed that the corpus could provide rich linguistic information for our models to learn from; while the large number of fonts could approximate a varied set of handwriting styles. We randomly sampled strings from the corpus such that their lengths L are distributed uniformly $L \sim U[1, 100]$. This was done to ensure that there are sufficient examples of text-lines of many different lengths; as shown later in Section 3.1.6, the other datasets tend to lack samples of shorter and longer lengths.

For each image generated, the height was set to 128 and the following properties were uniformly sampled:

- the font used from our collected set,
- the number of pixels between characters from $[-15, 0]$,
- and the number of pixels in each margin from $[0, 30]$.

The tool also provides some basic text distortions; the following of which were used and also randomly configured:

- the skewing of the texts horizontal direction by an angle from $[-4, 4]$,
- the vertical displacement of each character based on a sine or cosine signal,
- and Gaussian background noise.

3.1.5.2 Elastic Deformations

In order to further distort our images, we employed the *elastic deformation* (Figure 3.6) method proposed by P. Simard et al [137]. They postulate that, when applied to images of synthetic handwriting, the elastic deformation corresponds to *"uncontrolled oscillations of the hand muscles, dampened by inertia"* [137]. This method deforms an image by updating each pixel position (x, y) according to a displacement field $(\boldsymbol{\Delta x}, \boldsymbol{\Delta y})$ as per Equation 3.1; where α controls the intensity of the deformation.

$$\begin{aligned} x &\leftarrow x + \alpha \boldsymbol{\Delta x}(x) \\ y &\leftarrow y + \alpha \boldsymbol{\Delta y}(y) \end{aligned} \tag{3.1}$$

The displacement field is first initialised randomly from a uniform distribution (i.e. $\boldsymbol{\Delta x}(x) \sim U[-1, 1]$ and $\boldsymbol{\Delta y}(y) \sim U[-1, 1]$) and then its values are convolved with a Gaussian filter of standard deviation σ. In our work, we use M. Ernestus' open-source implementation [39] of the elastic deformation and, as recommended by P. Simard et al [137], set $\sigma = 4$ and $\alpha = 34$.

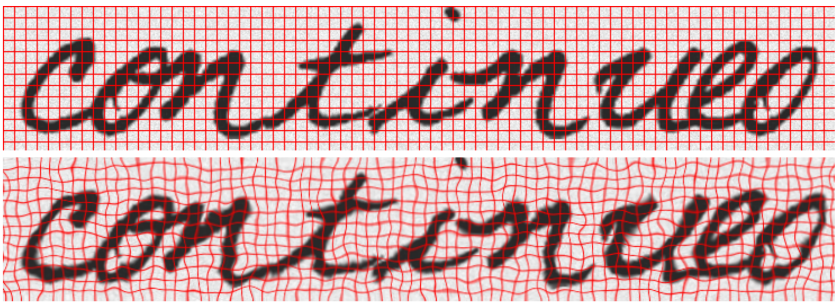


Figure 3.6: A demonstration of elastic deformation on a sample of synthetic handwriting; a red grid is overlaid on the original image (top) to highlight the effects of the deformation on the final image (bottom).

3.1.5.3 Multiprocessing

Our initial naïve method for creating the synthetic dataset was incredibly inefficient: a single process would generate, deform and persist each image one-at-a-time. This method was projected to take over 12 hours to create the entire dataset on an AMD Ryzen 5 1600 CPU with 16GB of RAM. We improved our method through process-based parallelism using Python's in-built multiprocessing library. Instead of just one process handling all tasks, the work was distributed over many worker processes.

More specifically, a pool of four workers were used to generate the initial images, another pool of four workers were used to deform the images and a single, dedicated worker persisted the final images to local storage. The intermediate images are communicated between the different stages asynchronously via message queues. Each image generation worker selects the next line of the corpus and font to use independently — the access to these shared resources are guarded by binary semaphores. This parallel method (Figure 3.7) generated the entire dataset in under 2 hours.

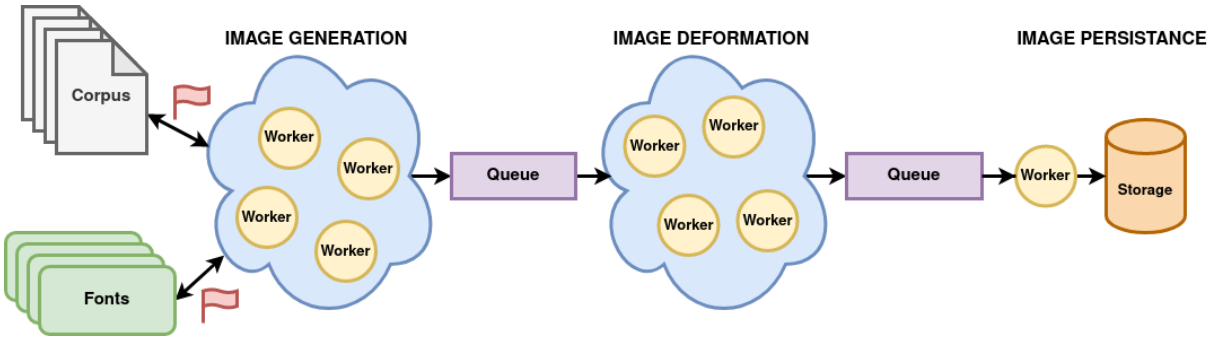


Figure 3.7: A high-level diagram of the multiprocessing method, as described in Section 3.1.5.3, used for generating our synthetic dataset.

3.1.6 Comparison of Datasets

The public datasets are split into training, validation and testing sets according to the official partitions provided with them. For our synthetic dataset, we randomly picked two unique subsets of 5000 samples for training and validation. The sizes of these sets are detailed in Table 3.1, along with the number of unique characters in each dataset.

Dataset	Number of samples			Number of unique characters	
	Train	Test	Valid	All	Punctuation
Bentham	8808	1372	820	91	21
IAM	6161	1861	1840	79	16
Saint Gall	468	707	235	48	0
Washington	325	168	163	68	8
Synthetic	40000	5000	5000	180	30

Table 3.1: A breakdown of the different partition sizes and the statistical properties of the samples, for each dataset used in this study.

Excluding the synthetic dataset, Bentham and IAM have by far the most samples; the Saint Gall and Washington datasets are significantly smaller (Table 3.1). Models trained on only the smaller datasets are at a higher risk of overfitting due to the lack of substantial data. One could expect these datasets to be the most challenging. However, they also use fewer unique characters; especially Saint Gall, which does not have any punctuations in its ground-truths (Table 3.1). While there is less information to learn from a smaller dataset, we believed that a more limited vocabulary would make learning easier.

The synthetic dataset, meanwhile, is significantly larger; both in terms of samples and unique characters (Table 3.1). This makes it ideal for pre-training on: models can first learn a broad and generic understanding of handwriting, before being fine-tuned to a specific set of real styles. However, it is crucial to fine-tune pre-trained models; we do not expect models only trained on synthetic data to perform well on real handwriting, as the synthetic data is far less complex. Moreover, real images of handwriting tend to have more significant background noise; such as those in the Bentham dataset (Figure 3.1).

The Bentham, Saint Gall and Washington datasets each contain samples from only a single writer, as mentioned in Sections 3.1.1, 3.1.3 and 3.1.4. We believed that this would make them, overall, less variable and, therefore, easier to learn than the IAM or synthetic datasets; which sample many different writers[2] (Sections 3.1.2 and 3.1.5). Despite this, we expected models to generalise better if they are trained on many different styles. This is another reason why the synthetic dataset is ideal for pre-training. Also, we firmly believe that the IAM dataset is the most comprehensive benchmark for HTR performance; given its many and varied handwriting styles and the fact that each style is only present in a single dataset partition (Section 3.1.2).

We have also analysed the distributions of characters and words[3] across all datasets. It was noted that the Bentham and Saint Gall datasets tend to have more characters per text-line than the others

[2]Here, for the synthetic dataset, we equate the number of writers to the number of unique fonts used.

[3]We defined the words in a string to be the groups of one or more non-empty-space characters delimited by a space character.

(Figure 3.8). As is discussed later in Section 3.2.5, an optical model is evaluated on a dataset based on the character and word error rates (CER and WER, respectively) of their predictions on the test set. The greater the number of characters in a ground-truth, the less of an impact a single, incorrectly predicted character has on the CER. Therefore, for all models, we expected CERs on the Bentham and Saint Gall datasets to be generally lower than those on other datasets. Unsurprisingly, given how our corpus was sampled (Section 3.1.5), the number of characters per synthetic text-line is distributed normally (Figure 3.8). This is yet another reason why the synthetic dataset is ideal for pre-training: it has sufficient examples of comparatively shorter and longer sequences, unlike the other datasets.

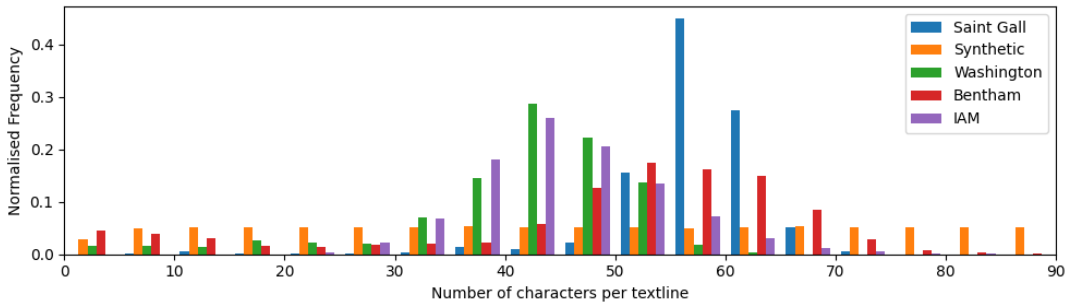


Figure 3.8: For each dataset, the distribution of the number of characters per ground-truth text-line.

Bentham's longer ground-truths are due to it, generally, having more words per text-line than the other datasets (Figure 3.9). The more words there are in a text-line, the more likely it is that two incorrectly predicted characters belong to different words; therefore, we expect Bentham WERs to be higher on average. Meanwhile, Saint Gall has similar numbers of words per line as IAM and Washington (Figure 3.9), however, its Latin words are usually longer than the words in the English datasets (Figure 3.10). The longer the words are, the more likely that two errors occur in the same word. For this reason, we expect models to generally score a lower WER on Saint Gall.

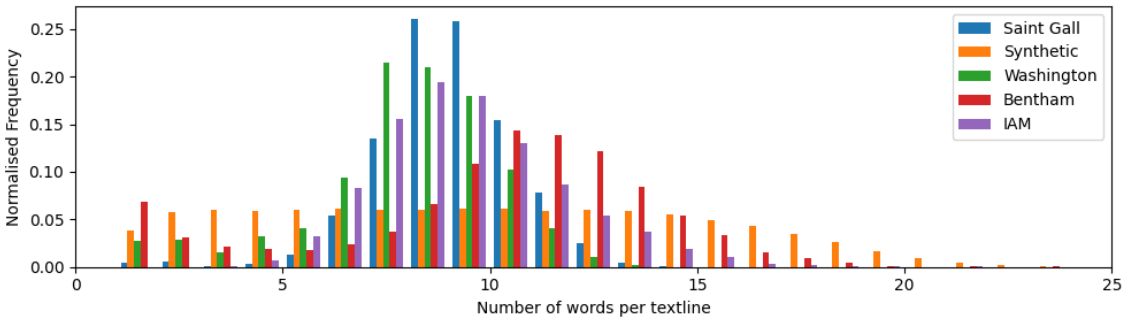


Figure 3.9: For each dataset, the distribution of the number of words per text-line.

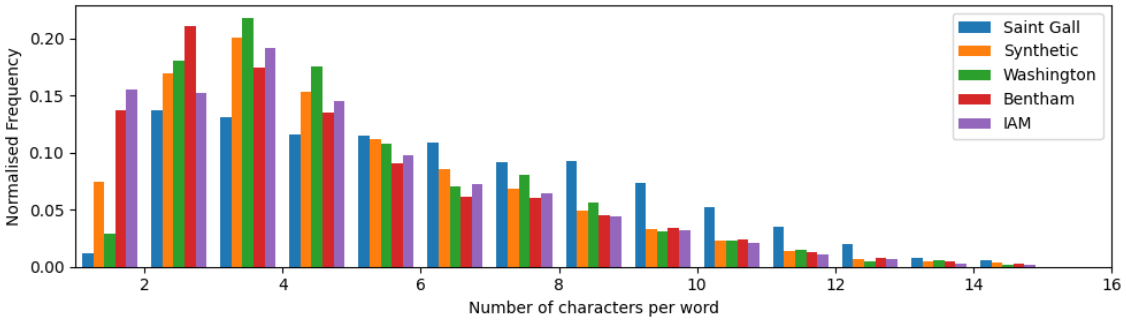


Figure 3.10: For each dataset and across all text-lines within it, the distribution of the number of characters per word.

On an interesting side note, the distribution of character frequencies is approximately *Zipfian* across all datasets (Figure 3.11). That is, in a collection of C unique elements, the normalised frequency f of the element with rank[4] k is given by 3.2. For example, in the Bentham dataset and when we only consider the top 50 characters, `e` is the second most common character with a normalised frequency of roughly 0.102 (Figure 3.11). Using $C = 50$ and $k = 2$ gives us $f(2; 50) \approx 0.111$.

$$f(k; C) = \frac{1}{k^{-1} \sum_{c=1}^{C} c^{-1}} \tag{3.2}$$

This reaffirms *Zipf's Law* [125] that, in a sufficiently large corpus, the frequency of a linguistic construct is inversely proportional to its rank.

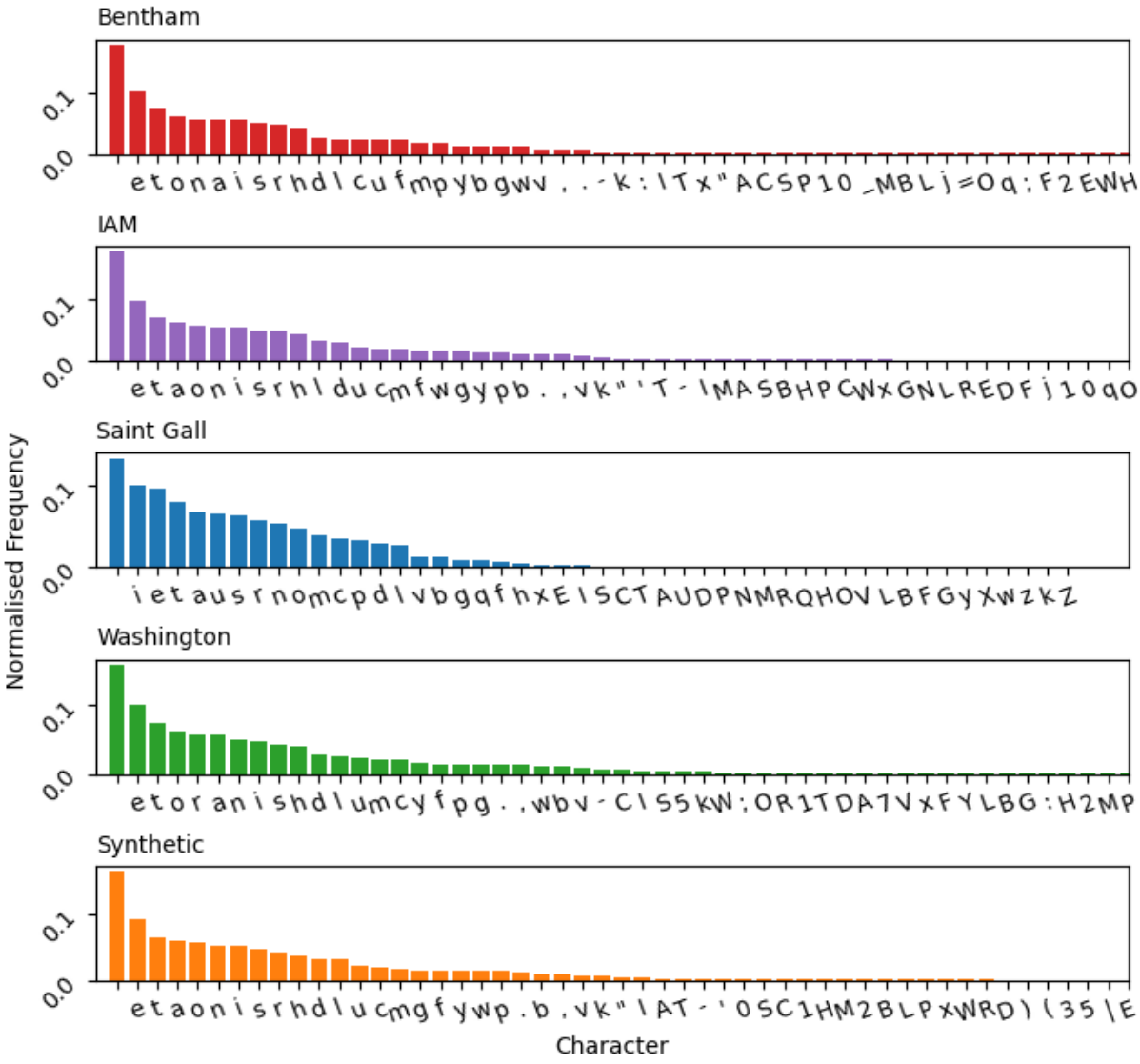


Figure 3.11: For each dataset, the top 50 most frequent characters and their frequencies.

[4]The rank of an element is its position index in a sequence of all elements in a decreasing order of frequency.

3.2 Experimental Setup

In order to compare the different architectures fairly, this study adopts a methodology similar to A. Flor et al [50], where all models are trained and evaluated under the exact same experimental setup. The experimental setup we used differed from that used originally for the Puigcerver and Flor architectures.

3.2.1 Implementation

Our work has been implemented using Tensorflow 2.3.0 on Python 3.7.3; one of the standard libraries for deep learning applications in industry. During development, our code was maintained in a GitHub repository [32]. Experiments were conducted using a single Nvidia Tesla P100 GPU with 16GB of memory on the University of Bristol's *BlueCrystal Phase 4* [2] high performance computing machine.

The code we used was initially forked from an open-source framework for offline HTR [47]. It provided the means to load and augment a user-supplied dataset, as well as to train and evaluate custom optical models on that dataset using CTC. This framework was chosen as the starting point for our implementation because of its developmental maturity and to reduce the amount of unnecessary heavy-lifting we had to do. We nevertheless discovered and fixed several bugs in the framework throughout the project; such as in its data loaders and dynamic image augmentations. These solutions have been integrated into the official framework repository as open-source contributions. Also, at the time of writing, our proposed architecture is in the process of being integrated into the framework; subject to peer-review.

BlueCrystal's resources are managed by the *SLURM* [159] scheduler and users submit computing jobs into a queue. During our project, a highly-configurable job scheduling pipeline was developed. This allowed us to efficiently submit experiments to BlueCrystal and ensured consistency across all attempts. For example, we could run the same experiment on all architectures multiple times using a single console command. This pipeline was incredibly useful for our later ablation study (Section 3.4), where a large number of different hyperparameter configurations had to be evaluated; the pipeline allowed us to do this without ever changing the underlying code.

The official Tensorflow implementations of neural network components were used where possible, however, we implemented the following components ourselves:

- Self-Attention Network (Section 2.6.2); including its sub-components:
  - Multi-Head Attention (Section 2.4.3.1),
  - Sinusoidal Positional Encoding (Section 2.4.3.5),
  - Residual Connection (Section 2.4.3.3).
- Gated convolution (Section 2.3.2),
- Learning rate scheduling (presented later in Section 3.2.4).

3.2.2 Preprocessing Techniques

All input images are preprocessed by normalising their pixel values into the range $[0, 1]$ and by resizing them into grayscale images of size $(W, H, C) = (1024, 128, 1)$ whilst maintaining the original aspect ratio through zero-padding. Meanwhile, for all ground-truth transcription strings, accents are removed from characters (e.g. é → e).

For training and validation, the ground-truths are encoded from strings into fixed-length sequences of one-hot vectors using a character-level tokenizer. The tokenizer uses a base alphabet of the first 95 printable ASCII characters[5]. This is extended with two special tokens, *pad* ¶ and *unknown* ¤; which are used to pad ground-truth transcriptions to the fixed length of $M = 128$ and to replace any characters in a ground-truth that are not in the base alphabet (respectively). Therefore, the base alphabet is of size $|\mathcal{A}| = 97$ and the intermediate CTC alphabet (Section 2.5.1) is of size $|\mathcal{A}'| = 98$. These special tokens are only used internally within the optical models, and are removed from their decoded predictions.

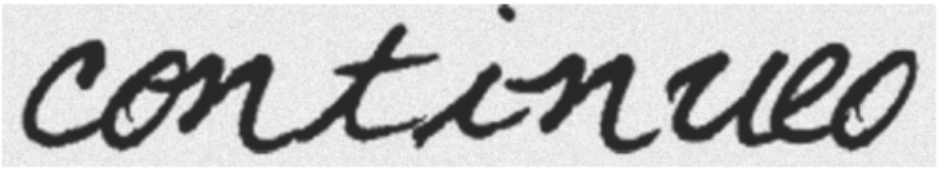
[5]This includes the numerical digits, all lower-case and upper-case unaccented letters of the English alphabet, basic punctuation and the space character.

3.2.3 Dynamic Augmentations

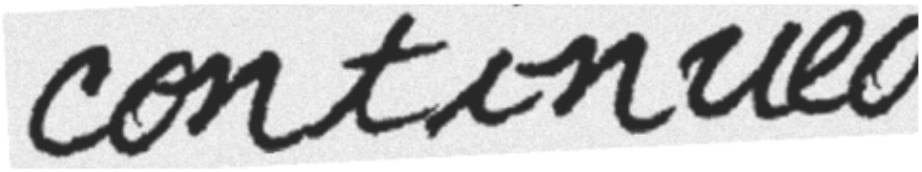
It is well established in the HTR literature that using dynamic image augmentations during training can greatly improve the final performance of an optical model [108, 127, 154]. This makes sense intuitively, as these augmentations artificially increase the number of unique training samples available. Therefore, in our experiments, we randomly[6] and dynamically apply the following augmentations to each image of every training batch:

- *affine warping*, including rotations of up to 3° in either direction, resizing of up to 5%, and displacement along either dimension by up to 5% each;
- *dilation* with a kernel of up to size 3×3,
- *erosion* with a kernel of up to size 5×5.

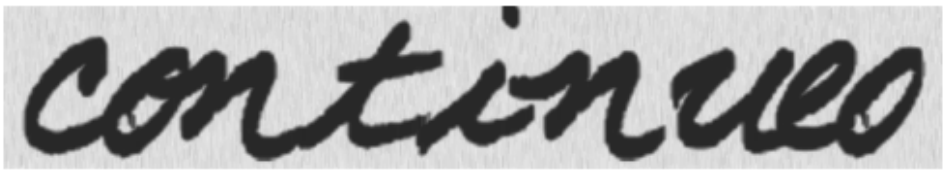
We provide examples of each such augmentation in Figure 3.12; however, due to their ubiquity in image processing, we reserve their formalisation to the works by T. Myers et al [110] and R. Srisha et al [139].



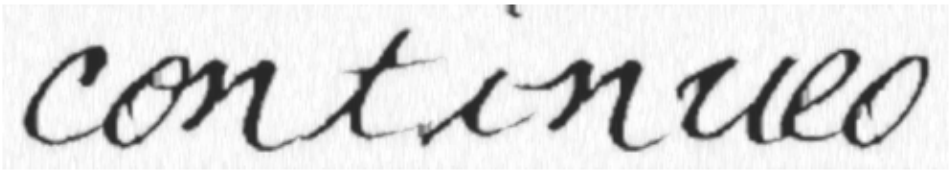
(a) The original image.



(b) The original image with affine warping applied.



(c) The original image with dilation applied.



(d) The original image with erosion applied.

Figure 3.12: A demonstration of the effects of the image augmentation techniques used dynamically during training.

[6] Random in the sense that each augmentation is applied to each image with a probability of 0.5 and its parameters are sampled from an uniform distribution.

3.2.4 Training

The optical models are trained to minimize the CTC loss function using the *RMSprop* gradient descent method [71]. RMSprop was chosen as it is used to train both Flor and Puigcerver in their original papers [50, 127]. As discussed in Section 2.6.3, SAN-based architectures are known to be unstable during training but this can be controlled through the careful scheduling of the learning rate.

Preemptively, we initially decided to schedule our learning rate using the method proposed by the original Transformer authors [148]. This method first linearly increases the learning rate for a number of warm-up training steps w, and then reduces it proportionally to the inverse square root of the training step number s (Figure 3.13). The learning rate $\eta(s)$ at step s is formalised by Equation 3.3, where the warm-up period $\mathcal{W}$ and scaling factor $\mathcal{K}$ are hyperparameters. As per [148], we used $\mathcal{W} = 4000$ and $\mathcal{K} = 256$. **Only later, during our ablation study (Section 3.4.8), would we discover that this scheduling function was not required. In fact, for our evaluations in Chapter 4, we used a fixed learning rate of 0.0003.**

$$\eta(s) = \frac{1}{\sqrt{\mathcal{K}}} \times \min\left(\frac{1}{\sqrt{s}}, \frac{s}{\mathcal{W}^{1.5}}\right) \tag{3.3}$$

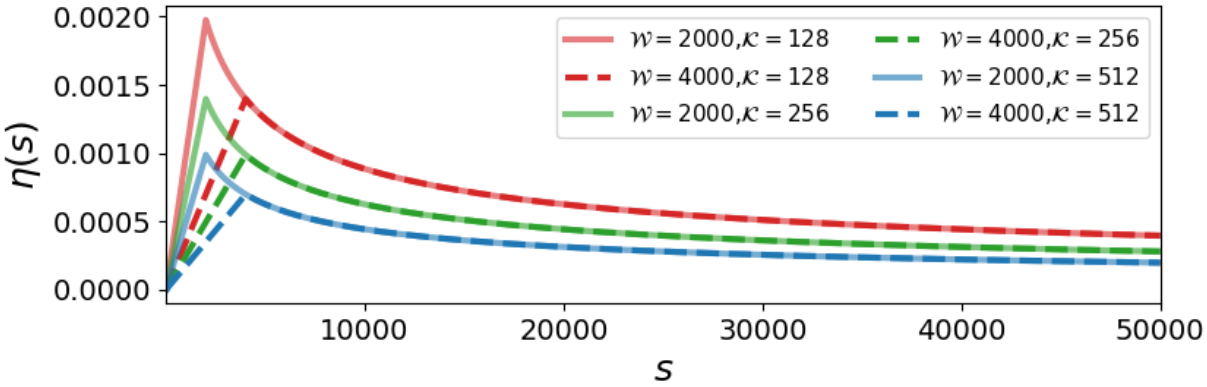


Figure 3.13: For a range of training steps s, the learning rate $\eta(s)$ as scheduled by Equation 3.3; where $\mathcal{W} \in \{2000, 4000\}$ and $\mathcal{K} \in \{128, 256, 512\}$.

Batches of size 16 are used at every training step, and a maximum of 1000 epochs are performed during training; where an epoch is considered as a full pass of the training set. There are $\lceil \frac{S}{B} \rceil$ training steps per epoch, where S is the size of the training set and B is the batch size. At the end of an epoch, the average loss across the training and validation sets are logged. The order of the training set is also randomly shuffled after every epoch. An early stopping mechanism is employed, ending the training process if the logged validation loss does not improve for 20 consecutive epochs.

3.2.5 Evaluation

The models are evaluated using the weight configuration which achieved the lowest validation loss during training. Their outputs are CTC decoded into predicted transcriptions using the *vanilla beam search* algorithm [75] with a beam width of 10. The standard HTR metrics of *character error rate* (CER) and *word error rate* (WER) are used for evaluation. CER and WER are calculated as the character-level and word-level Levenshtein distance [98] between predicted and ground-truth transcriptions (respectively). More formally, the CER is defined as the number of character-level substitutions S_C, insertions I_C and deletions D_C required to transform the prediction into the ground-truth, normalised by the total number of characters in the ground-truth N_C (Equation 3.4). The WER is defined equivalently but on the word-level (Equation 3.5). The lower the error rate, the better the model is at recognising handwriting.

$$CER = \frac{S_C + I_C + D_C}{N_C} \tag{3.4}$$

$$WER = \frac{S_W + I_W + D_W}{N_W} \tag{3.5}$$

3.2.6 Differences to Original Setups

For the Puigcerver and Flor architectures, our experimental setup differs in some ways to the setup used in their original papers [50, 127]. Here we present these differences; if we do not explicitly mention a certain configuration, you can assume that it is the same as in our experiments.
Firstly, as opposed to a scheduled rate, the original setups used the following initial learning rates:

- Puigcerver with 0.0003,
- and Flor with 0.001.

Both setups also used an early stopping mechanism, like in our setup, as well as an additional mechanism that reduces the learning rate by a certain factor whenever the validation loss plateaus. The configuration of the latter mechanism is not provided.

While Flor was originally evaluated on the Bentham, IAM, Saint Gall and Washington datasets, Puigcerver was only evaluated on IAM. Unlike our and Puigcervers work, Flor used an n-gram language model for post-processing spelling correction (Section 1.1). Puigcerver used an unspecified base alphabet and was implemented in the Lua language using the Torch deep learning library. We use the same implementation library and base alphabet as Flor.

There are also differences with the pre-processing and augmentations used. For example, Puigcerver resized input images to a fixed height of 128 without any padding. For the most part, our work and that of Flor and Puigcerver use the same image augmentation techniques (i.e. affine warping, dilation, erosion). While Puigcerver does not specify the individual configurations of these methods, we use the same configurations as Flor. However, Flor employed two additional techniques:

- *illumination compensation* [27] to remove shadows and balance contrast,
- and *deslanting* [150] to correct cursive-styles.

We reserve the technical details of these techniques to their cited works.

3.3 Replication Study

One of the aims of this project is to compare SANscript to the current state-of-the-art, but first we must establish the baseline performance to compare against. In this section, the previously identified state-of-the-art architectures (Section 2.5) are evaluated on the selected public dataset (Section 3.1). This is what we benchmark SANscript against. This was done to verify if the claimed performance can be replicated and if using our experimental setup has any significant impact on this performance. All results provided in this section are averaged over 3 attempts.

3.3.1 Using Our Experimental Setup

First, we trained and evaluated Flor and Puigcerver on each public dataset; using the experimental setup outlined in Section 3.2. The results are given in Table 3.2. Both models are consistently more accurate on the larger datasets (Bentham and IAM) than on the smaller ones (Saint Gall and Washington). On these smaller datasets, there tends to be a large gap between the training and validation set losses; these are signs of overfitting (Section 2.1.4). This confirms our initial suspicions (Section 3.1.6), in particular for Puigcerver on Washington. What is unusual, however, is the training times of Flor. This architecture has 0.1× as many parameters as Puigcerver, and its underlying GRUs are computationally less expensive than the LSTMs used by Puigcerver (Section 2.5.2.2). Yet, in our experiments, Flor generally requires more time per epoch during training than Puigcerver. We explore this issue in depth in Section 3.3.3.

The original authors of Puigcerver claim that it achieves a CER of 8.20% and WER of 25.40% on the IAM dataset [127]; our results for this are almost identical. The authors do not provide results for the other datasets we use, however, we assumed that the original implementation would have performed similarly on them; as is the case in our experiments. Therefore, we are confident that the claimed performance of Puigcerver has been replicated.

While Flor was originally evaluated on the same public datasets that we use, the authors do not provide results without the use of a post-processing language model; we cannot compare our results directly. However, under that setup, they claim that Flor achieves the same (if not slightly better) recognition rates as Puigcerver on all datasets [50]. For the most part, this is also observed in our experiments (Table 3.3). Given the assumption that the effects of a language model are independent of the optical model architecture (Section 1.1), we believe that claimed accuracy of Flor has also been replicated.

Dataset	Architecture	Test error (%)		Best loss		Time to train (hour:min:sec)		Total epochs
		CER	WER	Train	Valid	Per epoch	Total	
Bentham	Puigcerver	7.96	32.21	7.46	9.67	0:01:23	1:32:23	66.67
	Flor	8.25	33.47	5.71	8.91	0:01:31	3:01:34	119.67
IAM	Puigcerver	8.10	24.57	10.36	16.13	0:01:02	1:03:07	61.33
	Flor	7.92	25.90	8.85	13.78	0:01:07	1:48:57	98.33
Saint Gall	Puigcerver	8.54	40.71	11.56	22.36	0:00:06	0:09:34	103.67
	Flor	7.85	37.84	11.46	23.35	0:00:06	0:10:32	110.33
Washington	Puigcerver	33.52	66.90	30.11	61.83	0:00:04	0:07:33	112.00
	Flor	9.85	31.45	10.97	11.30	0:00:04	0:12:25	189.67

Table 3.2: The average performance results of training the Puigcerver and Flor architectures on each of the public datasets, under our experimental setup.

3.3.2 Using Original Experimental Setup

As discussed in Section 3.2.6, our experimental setup is different to that used in the original Puigcerver and Flor papers. The most notable difference is that we use learning rate scheduling (Section 3.2.4) where, previously, Puigcerver used an initial rate of 0.0003 and Flor 0.001 [50, 127]. To approximate the original setups, we replace the learning rate scheduler with the corresponding initial learning rate, and introduce a mechanism to reduce the learning rate by a factor of 0.2 if the validation loss does not improve for 15 consecutive epochs (Section 3.2.6). The rest of such a setup is identical to ours. Table 3.3 provides the performance results of Puigcerver and Flor under an approximation of their original setups.

Dataset	Architecture	Test error (%)		Best loss		Time to train (hour:min:sec)		Total epochs
		CER	WER	Train	Valid	Per epoch	Total	
Bentham	Puigcerver	7.55	31.99	5.82	9.53	0:01:27	2:05:36	86.00
	Flor	7.93	33.06	5.41	8.73	0:01:32	3:26:57	135.67
IAM	Puigcerver	7.68	23.48	7.21	16.32	0:01:03	1:31:55	88.00
	Flor	7.36	24.11	6.79	12.95	0:01:12	2:52:10	142.67
Saint Gall	Puigcerver	9.40	40.70	13.65	29.81	0:00:06	0:13:01	141.00
	Flor	7.98	40.00	11.51	18.16	0:00:06	0:04:38	48.00
Washington	Puigcerver	97.35	100.00	139.48	140.17	0:00:05	0:01:50	22.00
	Flor	9.18	29.38	3.25	11.86	0:00:04	0:09:03	101.67

Table 3.3: The average performance results of training the Puigcerver and Flor architectures, on each of the public datasets, under an approximation of their original setups (as described in 3.3.2)

Overall, Puigcerver and Flor achieve lower error rates when using their original setups than when using ours (Tables 3.2 and 3.3). The exception to this is on the Saint Gall and Washington datasets: overfitting is far more significant for the original setups. In fact, Puigcerver completely diverges on Washington. This is better illustrated in Figure 3.14.

It first appeared that using our setup reduced the number of epochs required to converge; for example, on the Bentham and IAM datasets, the loss converges faster during the warm-up period (roughly the first 20 epochs; Figure 3.14). However, after this point, the curves are very similar. The difference occurs between epochs 40 and 80, when the LR reducing mechanism of the original setup activates; in Figure 3.14, this corresponds to a sudden dip in the loss. It tends to happen around the same time that the training would terminate under our setup. Generally, after the LR is reduced, the training under the original setup continues for longer than under ours; this is how a lower error rate is achieved but also how the models begin to overfit.

Despite achieving higher error rates overall, we continued using the learning rate scheduling as part of our experimental setup. At this point, we still believed that it would be required to stabilise SANscript during training (Section 3.2.4).

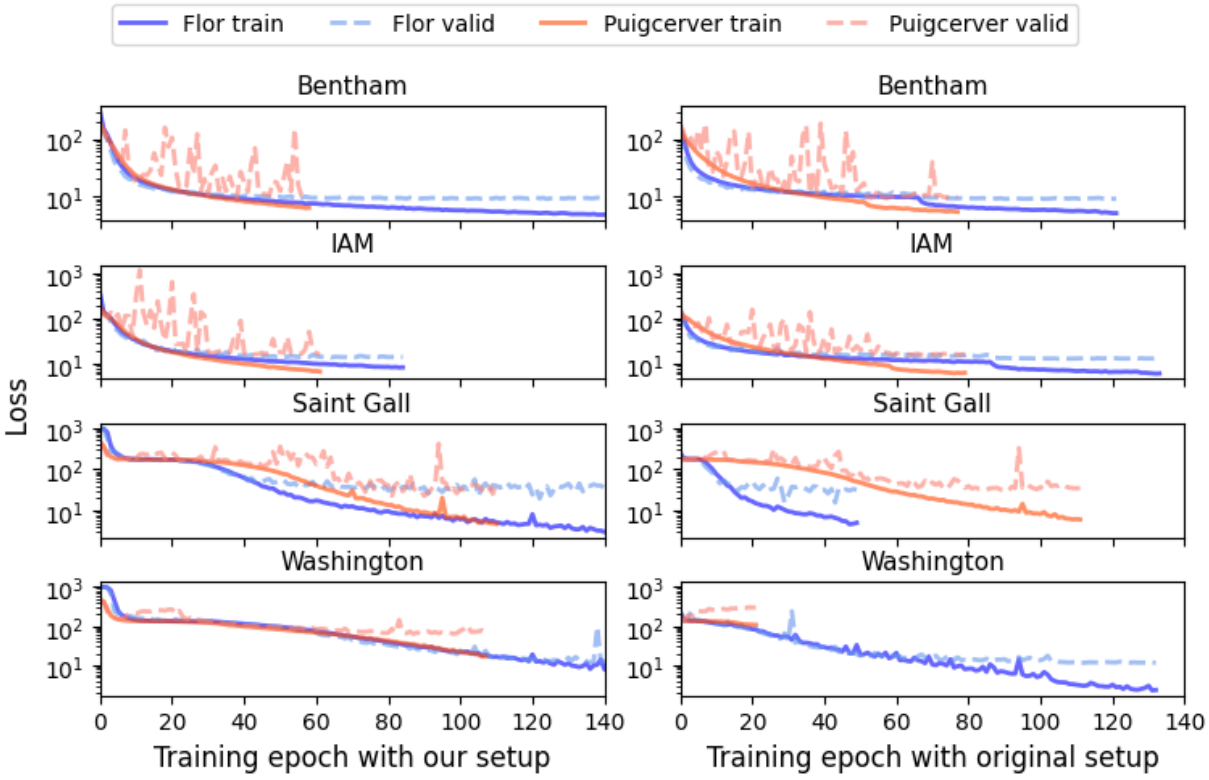


Figure 3.14: For the Puigcerver and Flor architectures and for each public dataset, the test and validation losses during training when using either our proposed (left) or their original (right) experimental setups.

3.3.3 Evaluation of Gated Convolutions

In our experiments, the Flor architecture has been observed to generally require more time per epoch than the Puigcerver architecture; as explained in Section 3.3.1, this is counter-intuitive. Flor is supposed to be a smaller, faster alternative to Puigcerver (Section 2.5.2.2). We discovered that this is due to Flors use of gated convolutions: if we remove them from its decoder, the time required per epoch reduces significantly (Table 3.4). These gated convolutions were implemented using the high-level Tensorflow framework but it is likely that our implementation does not fully utilize the GPU capacity when compiled. For comparison, the official Tensorflow implementations of neural network components (such as the standard convolution) are highly optimised for GPU use. To resolve this bottleneck, we could re-implement the gated convolution directly in a lower-level GPU-acceleration library; however, this is beyond the scope of our project. Removing the gated convolutions also leads to higher error rates (Table 3.4); therefore, despite their bottleneck, we continued to use them in Flor.

Dataset	Gated Convolutions	Test error (%)		Best loss		Time to train (hour:min:sec)		Total epochs
		CER	WER	Train	Valid	Per epoch	Total	
Bentham	Yes	8.25	33.47	5.71	8.91	0:01:31	3:01:34	119.67
	No	8.31	33.71	6.58	9.26	0:01:12	2:56:48	145.33
IAM	Yes	7.92	25.90	8.85	13.78	0:01:07	1:48:57	98.33
	No	9.68	30.53	12.59	16.19	0:00:44	1:02:06	84.33
Saint Gall	Yes	7.85	37.84	11.46	23.35	0:00:06	0:10:32	110.33
	No	7.36	34.81	7.84	24.17	0:00:04	0:07:41	118.33
Washington	Yes	9.85	31.45	10.97	11.30	0:00:04	0:12:25	189.67
	No	12.90	39.96	14.16	13.89	0:00:03	0:10:03	163.67

Table 3.4: The average performance results of training the Flor architecture on each of the public datasets, under our experimental setup and both with and without the use of gated convolutions.

3.3.4 Conclusions

The differences in our experimental setup and that of the Puigcerver and Flor papers do lead to different results for those architectures. However, when using either setup, we have shown that both architectures tend to have comparable error rates. This suggests we have successfully replicated the state-of-the-art performance for HTR. Therefore, at this point in the project, we were not concerned with these experimental discrepancies and continued using our setup. We have also explored the performance bottleneck in our implementation of the Flor architecture, caused by gated convolutions. While removing these layers from Flor improves its training speed considerably, it is at the loss of accuracy; we continued using this architecture as is. In the subsequent ablation study, we observed the same bottleneck in SANscript and further investigated the different experimental setups.

3.4 Ablation Study

In this section, we provide an ablation study into the configuration of our proposed architecture. Placing this section at this point of the dissertation is somewhat unusual: such work is often left to the end of the evaluation chapter. However, we believed that it is important to do this first. The aim of our later evaluation is to compare SANscript to the state-of-the-art, but this requires us to first optimise its many hyperparameters. Including this content in the evaluation chapter would detract from its overall aim. It also allows us to discuss the impact that each of SANscripts components has on its performance. As a reminder, SANscript (Section 2.6.2) has the following hyperparameters:

- the number of SAN layers N (Section 3.4.1),
- the number of attention heads per SAN H (Section 3.4.2),
- the dimension of the feature vectors D (Section 3.4.3),
- the hidden layer size of the FNNs F (Section 3.4.4), and
- the dropout rate R (Section 3.4.5).

Moreover, we have proposed two variants of our architecture:

- SANscript$_{\text{PUIG}}$ which uses the Puigcerver encoder,
- and SANscript$_{\text{FLOR}}$ which uses the Flor encoder.

For this ablation study, we performed a grid-search of the hyperparameter for both variants. We also evaluated the impact of using positional encoding (Section 3.4.6) and learning rate scheduling (Section 3.4.8). Lastly, we look into the use of gated convolutions in SANscript$_{\text{FLOR}}$ (Section 3.4.7).

The initial configuration[7] is $(N, H, D, F, R) = (1, 2, 128, 512, 0.2)$. Our aim is to find an optimal configuration that does not impose more parameters than the Puigcerver architecture (9.6 million; Section 2.6.3). We optimised hyperparameters on the *combined*[8] public dataset. While this does not guarantee that the final configuration is optimal for the datasets individually, optimising SANscript on each dataset is far too time-consuming. All results presented are averaged across 3 attempts.

3.4.1 Number of SAN Layers

First, we explore the number of SAN layers N (Table 3.5). Increasing N consistently reduces the error rate for both variants; however, it also increases the time per epoch. This makes sense as increasing a neural network's depth generally improves its performance [69] and the computational complexity of SANscript grows linearly with the number of SANs (Section 2.6.3). We select $N = 4$ as our optimal value to balance the error rate decrease and training time increase.

N	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
1	PUIG	14.35	47.75	0:00:48	2:14:34	167.00	0.468
	FLOR	11.81	41.43	0:02:05	5:42:02	164.00	0.407
2	PUIG	12.99	45.00	0:00:55	2:39:28	174.67	0.667
	FLOR	10.19	37.25	0:02:07	4:57:02	140.67	0.605
4	PUIG	11.66	42.03	0:00:56	2:26:20	157.33	1.063
	FLOR	9.96	37.11	0:02:18	4:14:04	110.33	1.002
6	PUIG	10.19	37.84	0:01:06	3:33:58	193.00	1.460
	FLOR	9.61	35.85	0:02:38	6:30:02	149.67	1.399

Table 3.5: For both variants and for $n \in \{1, 2, 4, 6\}$, the average performance results of training SANscript with $(N, H, D, F, R) = (n, 2, 128, 512, 0.2)$ on the combined dataset under our experimental setup.

[7] We have chosen this somewhat arbitrarily, based on the configuration $(N, H, D, F, R) = (6, 8, 512, 2048, 0.2)$ of the original Transformer [148]. Our initial hyperparameter values, with the exception of the dropout rate, are approximately 25% of these values.

[8] This is simply a combination of the different partitions of the Bentham, IAM, Saint Gall and Washington datasets.

3.4.2 Number of Attention Heads

Next, we evaluated the number of attention heads H in each SAN component; the results of which are given in Table 3.6. Increasing the number of attention heads generally reduces the error rate of both variants. As demonstrated in Section 2.4.3.1, each head of a SAN learns to model a different type of sequence dependency. We suspect that SANscript performs better with more heads as it is able to capture more complex combinations of dependencies. However, increasing the number of heads provides diminishing returns. This is likely because the dimensionality of each head D_H gets smaller as H increases (Section 2.4.3.1). In other words, the more heads a SAN has, the more types of relationship it can learn but each relationship is expressed in a lower-resolution. We selected $H = 4$ as our optimal value.

H	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
1	PUIG	12.28	43.41	0:00:58	2:25:26	150.67	1.063
	FLOR	9.92	36.81	0:02:24	5:29:59	137.33	1.002
2	PUIG	11.66	42.03	0:00:56	2:26:20	157.33	1.063
	FLOR	9.96	37.11	0:02:18	4:14:04	110.33	1.002
4	PUIG	9.90	37.26	0:01:00	2:36:20	153.33	1.063
	FLOR	9.38	35.40	0:02:25	4:54:05	122.33	1.002
8	PUIG	10.92	39.90	0:01:00	2:51:25	172.67	1.063
	FLOR	9.52	35.80	0:02:24	6:22:42	159.33	1.002

Table 3.6: For both variants and for $h \in \{1, 2, 4, 8\}$, the average performance results of training SANscript with $(N, H, D, F, R) = (4, h, 128, 512, 0.2)$ on the combined dataset under our experimental setup.

3.4.3 Feature Vector Dimensionality

We found that increasing the feature vector dimensionality D, to an extent, improves error rates and reduces the total number of epochs required (Table 3.7). However, it has a significant impact on the number of parameters. This is unsurprising, given that the FNN in each SAN has $2DF + F + D$ many parameters; where F is the hidden size. Also, there is a dense projection at the beginning of the SANscript decoder (Section 2.6.2) with $ED + D$ many parameters; where E is the dimensionality of the incoming encoder features. E is dependent on the configuration of the convolutional layers: $E = 1280$ for the Puigcerver encoder and $E = 256$ for the Flor encoder. This explains why $\text{SANscript}_{\text{PUIG}}$ has more parameters than $\text{SANscript}_{\text{FLOR}}$, despite sharing the same hyperparameters. Interestingly, increasing the feature dimensionality has little-to-no effect on the time required per epoch. Doing so does increase the number of individual calculations to perform, but we expect most of these to be performed in parallel. In the end, we chose $D = 192$ as our optimal value: a larger value may lead to better accuracy, but it imposes more parameters and increases the risk of overfitting.

D	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
128	PUIG	9.90	37.26	0:01:00	2:36:20	153.33	1.063
	FLOR	9.38	35.40	0:02:25	4:54:05	122.33	1.002
192	PUIG	9.75	36.70	0:01:01	2:31:04	149.33	1.744
	FLOR	9.08	34.44	0:02:29	4:18:14	103.33	1.617
256	PUIG	9.91	37.08	0:01:03	2:11:08	125.67	2.555
	FLOR	8.97	33.95	0:02:28	4:30:56	110.00	2.364

Table 3.7: For both variants and for $d \in \{128, 192, 256\}$, the average performance results of training SANscript with $(N, H, D, F, R) = (4, 4, d, 512, 0.2)$ on the combined dataset under our experimental setup.

3.4.4 FFN Hidden Size

As discussed in Section 2.4.3.2, the FFN inside a SAN has a hidden dimension of F that is considerably larger than the input/output dimensions D. However, our results (Table 3.4.4) show that increasing F actually tends to harm accuracy. In fact, using a comparatively smaller F can lead to better error rates in some scenarios. This is counter-intuitive: many state-of-the-art applications of the Transformer architecture use $F = 4D$ [79, 35, 133, 148]; in our case, this would be $4 \times 192 = 768$. We suspect that the benefits of using such a large hidden size come with having larger amounts of training data. For similar reasons as the feature dimensionality D (Section 3.4.3), increasing F also has a significant impact on the number parameters. In the end, we continued using $F = 512$.

F	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
256	PUIG	10.53	39.01	0:01:01	2:19:33	133.67	1.349
	FLOR	8.93	33.94	0:02:22	4:19:55	110.33	1.223
512	PUIG	9.75	36.70	0:01:01	2:31:04	149.33	1.744
	FLOR	9.08	34.44	0:02:29	4:18:14	103.33	1.617
768	PUIG	9.97	37.61	0:00:58	2:16:22	141.00	2.138
	FLOR	9.62	35.91	0:02:24	4:07:41	102.33	2.011
1024	PUIG	10.06	37.62	0:01:03	2:25:155	137.33	2.555
	FLOR	9.24	34.85	0:02:25	4:40:09	115.67	2.405

Table 3.8: For both variants and for $f \in \{256, 512, 768, 1024\}$, the average performance results of training SANscript with $(N, H, D, F, R) = (4, 4, 192, f, 0.2)$ on the combined dataset under our experimental setup.

3.4.5 Dropout Rate

Increasing the dropout rate R generally leads to a greater number of epochs required to converge (Table 3.9). This makes sense: when you train a model with dropout, only a subset of its parameters will be updated at each iteration; the larger R is, the smaller these subsets are on average. A model with a higher dropout rate is expected to require more training iterations to converge than a model with a lower rate because a smaller proportion of its weights are updated at one time.

Dropout is supposed to help prevent overfitting (Section 2.1.4). However, at this point in the ablation study, overfitting was not a problem; likely due to use of a combination of many datasets. We expected dropout to be more useful later, when training on one of the smaller datasets individually. Generally, a lower error rate is achieved when using $R \in \{0.1, 0.2\}$ than when using $R = 0$ (i.e. no dropout). It appears that using $R = 0.4$ significantly harms accuracy, however, this is a red herring. This occurs because the early stopping threshold is too small. Remember, the higher the dropout rate is, the more epochs that are required to converge. This could be resolved by increasing the threshold; however, we did not have enough time to try this. All things considered, we chose $R = 0.1$.

R	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
0	PUIG	10.34	37.85	0:00:58	1:30:18	93.67	1.744
	FLOR	10.19	37.77	0:02:28	2:39:44	64.33	1.617
0.1	PUIG	9.66	36.44	0:00:58	1:42:58	107.00	1.744
	FLOR	9.05	34.61	0:02:28	3:18:07	80.00	1.617
0.2	PUIG	9.75	36.70	0:01:01	2:31:04	149.33	1.744
	FLOR	9.08	34.44	0:02:29	4:18:14	103.33	1.617
0.4	PUIG	23.62	65.36	0:01:00	0:38:31	38.33	1.744
	FLOR	23.75	65.54	0:02:30	1:38:56	39.67	1.617

Table 3.9: For both variants and for $r \in \{0, 0.1, 0.2, 0.4\}$, the average performance results of training SANscript with $(N, H, D, F, R) = (4, 4, 192, 512, r)$ on the combined dataset under our experimental setup.

3.4.6 Use of Positional Encoding

The use of sinusoidal positional encoding (PE) has minimal effect on the final performance of SANscript (Table 3.10). This is surprising: remember that SANs are inherently content-based and not position-based (Section 2.4.3.5). The original Transformer authors argue that positional information must be injected somehow for a SAN-based model to learn position-based dependencies [148]. We suspect that the strict monotonic alignment enforced by the CTC loss function acts as an inductive bias of sequence position in our experiments, circumventing the need for PE. In the original Transformer paper, CTC was not used (Section 2.6.1). Nevertheless, using PE has no significant impact on the training times or number of parameters, and may even reduce the total number of epochs required (Table 3.10); therefore, we continued to use it in SANscript.

PE	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
Yes	PUIG	9.66	36.44	0:00:58	1:42:58	107.00	1.744
	FLOR	9.05	34.61	0:02:28	3:18:07	80.00	1.617
No	PUIG	9.67	36.30	0:01:01	2:11:26	129.33	1.744
	FLOR	9.17	34.75	0:02:21	4:49:06	122.67	1.617

Table 3.10: For both variants with $(N, H, D, F, R) = (4, 4, 192, 512, 0.1)$, the average performance results of training SANscript on the combined dataset under our experimental setup, both with and without the use of PE.

3.4.7 Use of Gated Convolutions

Throughout this ablation study, $\text{SANscript}_{\text{FLOR}}$ has required substantially more time per epoch than $\text{SANscript}_{\text{PUIG}}$ to train. A similar bottleneck was observed in the recurrent Flor architecture; the cause of which was attributed to our implementation of its underlying gated convolutions (Section 3.3.3). Indeed, when these layers are removed, $\text{SANscript}_{\text{FLOR}}$ requires significantly less time per epoch (Table 3.4.7). Surprisingly, it also achieves lower error rates; when the gated convolutions were removed from Flor, its errors rates increased (Section 3.3.3). We cannot explain this discrepancy.

Despite these gains, the FLOR variant without its gated convolutions was still much slower than the PUIG variant (Table 3.4.7). In this scenario, the main difference between the two is that FLOR uses large-strided convolutions for down-sampling while PUIG uses average pooling (Section 2.5.2.2). Previous work has shown that the pooling layers in a CNN can be replaced with such convolutions with no detriment [138]; our results suggested otherwise, but this may be due to our specific implementation. Moreover, the modified $\text{SANscript}_{\text{FLOR}}$ still requires a similar total time to train as before, due to the increase in the number of epochs. Therefore, we did not change the encoder of $\text{SANscript}_{\text{FLOR}}$.

Gated Convolutions	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
	CER	WER	Per epoch	Total		
Yes	9.05	34.61	0:02:28	3:18:07	80.00	1.617
No	8.79	34.11	0:01:57	3:23:41	103.67	1.542

Table 3.11: The average performance results of training $\text{SANscript}_{\text{FLOR}}$ with $(N, H, D, F, R) = (4, 4, 192, 512, 0.1)$ on the combined dataset under our experimental setup, either with and without the use of gated convolutions in its encoder.

3.4.8 Use of Learning Rate Scheduling

SAN-based architectures are known to be unstable during the early stages of training. In anticipation of this issue, we implemented a function $\eta(s)$ to carefully schedule the learning rate (LR) at each training step s (Section 3.2.4). To validate its use, we have compared this experimental setup to the approximated setups of the original Flor and Puigcerver papers defined in Section 3.3.2. The results are provided in Table 3.4.8:

Setup	Variant	Test error (%)		Time to train (hour:min:sec)		Total epochs	Params ($\times 10^6$)
		CER	WER	Per epoch	Total		
Ours	PUIG	9.66	36.44	0:00:58	1:42:58	108.00	1.744
	FLOR	9.05	34.61	0:02:28	3:18:07	80.00	1.617
Flor	PUIG	97.79	100.00	0:00:58	1:04:14	66.33	1.744
	FLOR	97.80	100.00	0:02:24	3:27:45	86.00	1.617
Puigcerver	PUIG	8.36	32.70	0:01:01	2:01:43	120.67	1.744
	FLOR	7.87	31.14	0:02:30	4:17:11	102.33	1.617

Table 3.12: The average performance results of training both variants of SANscript with $(N, H, D, F, R) = (4, 4, 192, 512, 0.1)$ on the combined dataset under either our, Flors or Puigcervers experimental setup.

Under the original Flor experimental setup, the training of both SANscript variants diverged (Table 3.12): this could suggest instability. However, qualitatively speaking, the SANscript loss curves for the other setups do not seem unstable (Figure 3.15). There is some noise in the validation loss for $\text{SANscript}_{\text{PUIG}}$, but this was also observed in the Puigcerver architecture (Section 3.3.2); we can, therefore, attribute this to the encoder. Moreover, SANscript can converge using the Puigcerver setup and even achieve lower errors rates (Table 3.12). This was when we realised that our architecture may not suffer from training instability as much as other, larger SAN-based architectures (Section 3.2.4) and, in turn, that learning rate scheduling may not be required.

The loss curves under our and the Puigcerver setups are very similar until around epoch 80, where the plateau mechanism of the latter activates (Figure 3.15). This suggests that, towards the end of training, a smaller LR is required to achieve better recognition. This could be achieved by exploring and optimising the hyperparameters of $\eta(s)$ (Section 3.2.4). There was not enough time to do this and we reserved this for future work. **The learning rate scheduling was removed from subsequent experiments; the approximated Puigcerver method, as defined in Section 3.3.2, is used instead.**

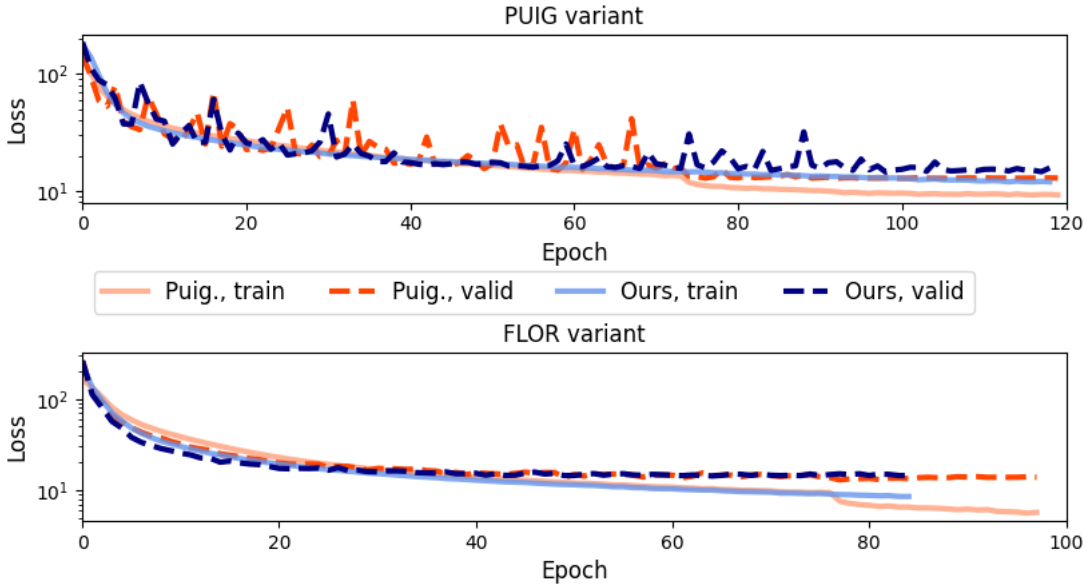


Figure 3.15: The test and validation losses during the training of the PUIG (top) and FLOR (bottom) variants of SANscript on the combined dataset, using either our or Puigcervers experimental setup.

3.4.9 Conclusions

Despite achieving better recognition rates than SANscript$_{\text{PUIG}}$, SANscript$_{\text{FLOR}}$ is severely limited by its slower training. The average time it requires per epoch can be reduced significantly by removing the gated convolutions from the encoder, although it is still twice that of SANscript$_{\text{PUIG}}$. We therefore concluded that the optimal configuration of SANscript is the PUIG variant with $(N, H, D, F, R) = (4, 4, 192, 512, 0.1)$. Henceforth, when SANscript is mentioned, it is in reference to this configuration. At the start of this study, we believed that SANscript would be unstable during training, in turn requiring learning rate scheduling. Our results have contradicted this. For subsequent experiments, an initial learning rate of 0.0003 and a plateau mechanism with a tolerance of 15 epochs were used instead.

Chapter 4

Critical Evaluation

In this chapter, we provide a critical evaluation of our results and answer the following research questions:

- Can SANs be used to replace RNNs in the current state-of-the-art optical models for HTR?
- Does it achieve the same level of recognition?
- Does it require less time to train?

We have compared the relative improvement of using one optical model over another based on how it reduces the final error rates and loss, and the time required to train. In other words, the smaller these performance results are, the better a model is.

First, in Section 4.1, we compared our proposed architecture (SANscript; Section 2.6.2) to our selected recurrent architectures (Puigcerver and Flor; Section 2.5) by training them on each public dataset (Section 3.1) individually. Secondly, in Section 4.2, we compared these same architectures after they had been pre-trained on our synthetic dataset (Section 3.1.5). Lastly, in Section 4.3, we visualise the self-attention mechanism that underpins our proposed architecture and offer interpretations of its behaviour.

In the previous ablation study (Section 3.4), we determined the optimal configuration of SANscript to be the so-called PUIG variant with the following hyperparameters: $(N, H, D, F, R) = (4, 4, 192, 512, 0.1)$. This is what is used throughout this chapter. We also decided to change our experimental setup (Section 3.2) to use an initial learning rate of 0.0003 and a mechanism that reduces the learning rate if the validation loss does not improve for 15 epochs; instead of a scheduling function. Unless stated otherwise, all architectures were trained and evaluated under this setup, with their results averaged across 3 attempts.

Overall, our research has that shown that SANscript

- can closely approximate the state-of-the-art recognition rates, and
- consistently requires significantly less time to train than recurrent architectures;

This demonstrates that self-attention networks can indeed replace recurrent neural networks for handwritten text recognition.

4.1 Evaluation on Public Datasets

We hypothesized that SANscript would be able to achieve the state-of-the-art recognition rates of Puigcerver and Flor, whilst requiring less time to train and no more parameters than Puigcerver (Section 2.6.3). As for the latter regard, SANscript imposes considerably fewer parameters than Puigcerver (Table 4.1). However, it is also twice as large as Flor, making SANscript comparatively more likely to overfit during training.

Architecture	Parameters ($\times 10^6$)
Puigcerver	9.591
SANscript	1.744
Flor	0.847

Table 4.1: The total number of trainable parameters of each architecture evaluated in this chapter.

Each architecture was trained on each public dataset from randomly initialised weights (Table 4.2). Puigcerver and SANscript completely diverged on the Washington dataset, Flor did not. This is likely due to the limited size of the dataset, and the significantly fewer parameters used by Flor. With the exception of Washington, Flor and Puigcerver generally achieve comparable error rates on all datasets, but Flor requires more time per epoch and in total to train. Meanwhile, on average, the CER and WER of SANscript were 1.23× and 1.21× (respectively) that of Flor and Puigcerver together. However, it required around 0.5× the time per epoch required by either Flor or Puigcerver. This also meant that it required less time in total to train, although the speedup here was less significant due to a slight increase in the overall number of epochs.

Dataset	Architecture	Test error (%)		Best loss		Time to train (hour:min:sec)		Total epochs
		CER	WER	Train	Valid	Per epoch	Total	
Bentham	Puigcerver	7.55	31.99	5.82	9.53	0:01:27	2:05:36	86.00
	Flor	7.82	32.87	4.73	8.59	0:01:34	4:18:45	165.00
	SANscript	9.19	35.24	8.65	11.59	0:00:41	1:13:19	116.00
IAM	Puigcerver	7.68	23.48	7.21	16.32	0:01:03	1:31:55	88.00
	Flor	7.53	24.47	7.20	13.16	0:01:06	2:58:46	161.33
	SANscript	10.26	32.62	10.52	19.89	0:00:28	1:03:07	134.00
Saint Gall	Puigcerver	9.40	40.70	13.65	29.81	0:00:06	0:13:01	141.00
	Flor	9.39	42.05	15.16	31.44	0:00:06	0:07:10	74.33
	SANscript	10.76	49.73	15.83	28.90	0:00:03	0:04:59	111.00
Washington	Puigcerver	97.34	100.00	139.48	140.17	0:00:05	0:01:50	22.00
	Flor	9.12	28.91	4.12	10.91	0:00:04	0:15:10	230.67
	SANscript	97.34	100.00	141.85	147.32	0:00:02	0:00:53	22.00

Table 4.2: For each dataset and architecture, the average performance results of training that architecture on that dataset from random initial weights.

In Section 3.1.6, we theorised on the relative difficulties that each of the public datasets would present our models. It was first expected for the WERs achieved on Bentham to be comparatively higher than those on other datasets; our results confirm this. Despite achieving similar CERs on IAM, all models have a larger WER on Bentham (Table 4.2). We also theorised that WERs would be generally lower on the Saint Gall dataset, however, this has been contradicted (Table 4.2). We incorrectly assumed that, due to the longer words of Saint Gall, character errors were more likely to occur within the same word. The opposite appears to be the case.

Despite providing a significant training speedup over them, SANscript did not sufficiently approximate the recognition of Flor and Puigcerver. However, in the earlier ablation study (Section 3.4.8), the average[1] SANscript error rate across all datasets was 0.83× that achieved in these experiments. There, SANscript was trained on a combination of all public datasets, whereas, here it was trained on each one individually. This suggested that our architectures' performance could improve given more substantial training data; further motivating synthetic pre-training.

[1] Excluding any experiments where the training loss diverged.

4.2 Evaluation of Synthetic Pre-training

To pre-train a model on the synthetic dataset, the weights are randomly initialised as usual. However, the experimental setup is modified as follows: the maximum number of epochs is set to 100, and the early stopping mechanism is disabled. In other words, a fixed set of 100 epochs are performed during pre-training. To then fine-tune a model on another dataset, the weights are initialised to the configuration that achieved the lowest validation loss during pre-training. The experimental setup is not modified for tuning.

4.2.1 Pre-training Results

As first expected (Section 3.1.6), it is important to fine-tune models after pre-training. All models performed poorly on the public datasets when they had only been pre-trained synthetically (Table 4.3). Note that these untuned models performed considerably better on IAM than on all other datasets. This makes sense as, qualitatively-speaking, the electronic fonts used in the synthetic dataset (Section 3.1.5) most closely resemble the handwriting styles of IAM (Section 3.1.2). Flor performed significantly worse than the other untuned models. We suspected that this was due to its comparatively smaller size, and the simplicity of the synthetic data relative to real handwriting. During pre-training, Flor converges both faster and to a significantly lower level than either Puigcerver or SANscript (Figure 4.1). The latter two, meanwhile, share very similar loss curves and have substantially more parameters than Flor. We believe that Flor became too confident on the less complex synthetic samples, such that it did not immediately adapt well to real inputs. This could be prevented by pre-training for fewer epochs, or by using a different experimental setup. It is also important to note how much faster SANscript was to pre-train: requiring 4 hours to perform the 100 epochs, while Flor and Puigcerver required over 10 hours each.

Dataset	Architecture	Test Error (%)	
		CER	WER
Bentham	Puigcerver	47.37	82.51
	Flor	98.45	100.00
	SANscript	53.33	91.52
IAM	Puigcerver	27.73	69.37
	Flor	59.54	92.77
	SANscript	33.50	80.63
Saint Gall	Puigcerver	46.60	98.64
	Flor	99.33	99.85
	SANscript	43.31	97.58
Washington	Puigcerver	51.33	93.99
	Flor	94.11	100.00
	SANscript	51.98	96.21

Table 4.3: For each architecture, the error rates achieved on each public dataset without any fine-tuning after pre-training on the synthetic database.

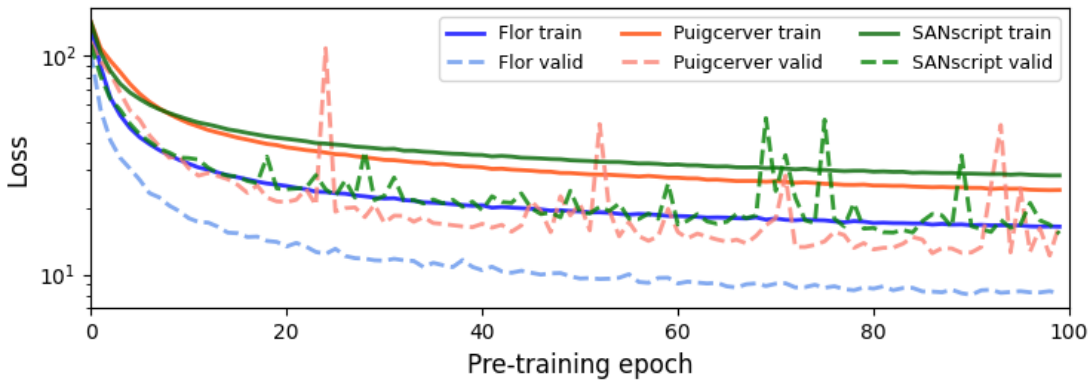


Figure 4.1: For each architecture, the training and validation losses during synthetic pre-training.

4.2.2 Fine-Tuning Results

The performance results of fine-tuning the pre-trained models is provided in Table 4.4. Clearly, synthetic pre-training greatly benefits all architectures. Compared to training from random weights (Table 4.2), the error rates and epochs required are generally lower here. Note that Puigcerver and SANscript achieved strong recognition on the Washington dataset (Table 4.4) where previously, without pre-training, their losses would completely diverge (Table 4.2).

With pre-training and fine-tuning, the differences between the error rates of SANscript and that of Flor and Puigcerver become significantly smaller. Previously, the average CER and WER achieved by SANscript were 1.23× and 1.21× (respectively) that of Puigcerver and Flor combined (Section 4.1). Here, these same ratios are 1.08× and 1.07× (Table 4.4). In other words, SANscript was less than 10% away from the state-of-the-art performance. Therefore, given sufficient training data, we argue that it is indeed possible to replace the RNNs in optical models with SANs and still achieve comparable recognition.

For all architectures, fine-tuning a pre-trained model required substantially fewer epochs than when training it from randomly initialised weights (Figure 4.2). Pre-training also appears to have reduced the variability in the validation loss for the SANscript and Puigcerver architectures. As for the total time required to fine-tune, SANscript is unmatched (Table 4.4). On average across all datasets, it required 0.35× the time of Puigcerver and Flor together. Despite using highly GPU-optimised LSTM[2] and GRU[3] layers in the implementation of these recurrent architectures (Section 3.2.1), SANscript consistently achieved a superior training speed thanks to the highly parallelisable nature of its underlying SANs (Section 2.6.3).

Moreover, on the Saint Gall dataset, SANscript outperformed Puigcerver and Flor on all regards (Table 4.4). This is significant, given that Saint Gall is in a different language and script to the synthetic dataset (Section 3.1.3); and has limited training samples (Section 3.1.6). This supports the findings of L. Kang et al [79]. That is, after generic pre-training, a SAN-based optical model can adapt to a new style more effectively than an RNN-based one when there is little labelled data.

Dataset	Architecture	Test error (%)		Best loss		Time to train (hour:min:sec)		Total epochs
		CER	WER	Train	Valid	Per epoch	Total	
Bentham	Puigcerver	6.51	30.03	5.44	7.60	0:01:24	1:49:49	78.33
	Flor	7.46	32.18	4.12	7.90	0:01:31	3:32:31	140.33
	SANscript	7.97	34.36	6.85	9.58	0:00:41	0:57:24	84.33
IAM	Puigcerver	5.61	17.78	7.83	11.43	0:01:02	0:54:15	52.33
	Flor	6.84	22.87	6.75	11.84	0:01:07	1:37:58	88.33
	SANscript	7.50	25.46	9.20	14.48	0:00:25	0:25:42	60.67
Saint Gall	Puigcerver	8.46	39.58	17.38	27.79	0:00:06	0:02:58	28.00
	Flor	8.39	41.10	17.46	26.11	0:00:06	0:02:59	29.67
	SANscript	7.12	35.68	6.35	18.79	0:00:03	0:01:56	39.00
Washington	Puigcerver	6.65	21.33	5.88	7.67	0:00:04	0:03:55	56.33
	Flor	6.27	21.85	2.03	6.91	0:00:04	0:06:00	90.00
	SANscript	7.66	27.00	4.06	8.51	0:00:02	0:01:53	66.00

Table 4.4: For each dataset and architecture, the average performance results of fine-tuning that architecture on that dataset after pre-training on the synthetic dataset.

In summary, we have demonstrated that — by pre-training on a large synthetic dataset and fine-tuning on a real dataset — SANscript can:

- closely approximate the state-of-the-art error rates;
- require significantly less time to train than the leading recurrent models; and
- adapt to low-resource handwriting styles effectively.

[2] https://www.tensorflow.org/api_docs/python/tf/compat/v1/keras/layers/CuDNNLSTM
[3] https://www.tensorflow.org/api_docs/python/tf/compat/v1/keras/layers/CuDNNGRU

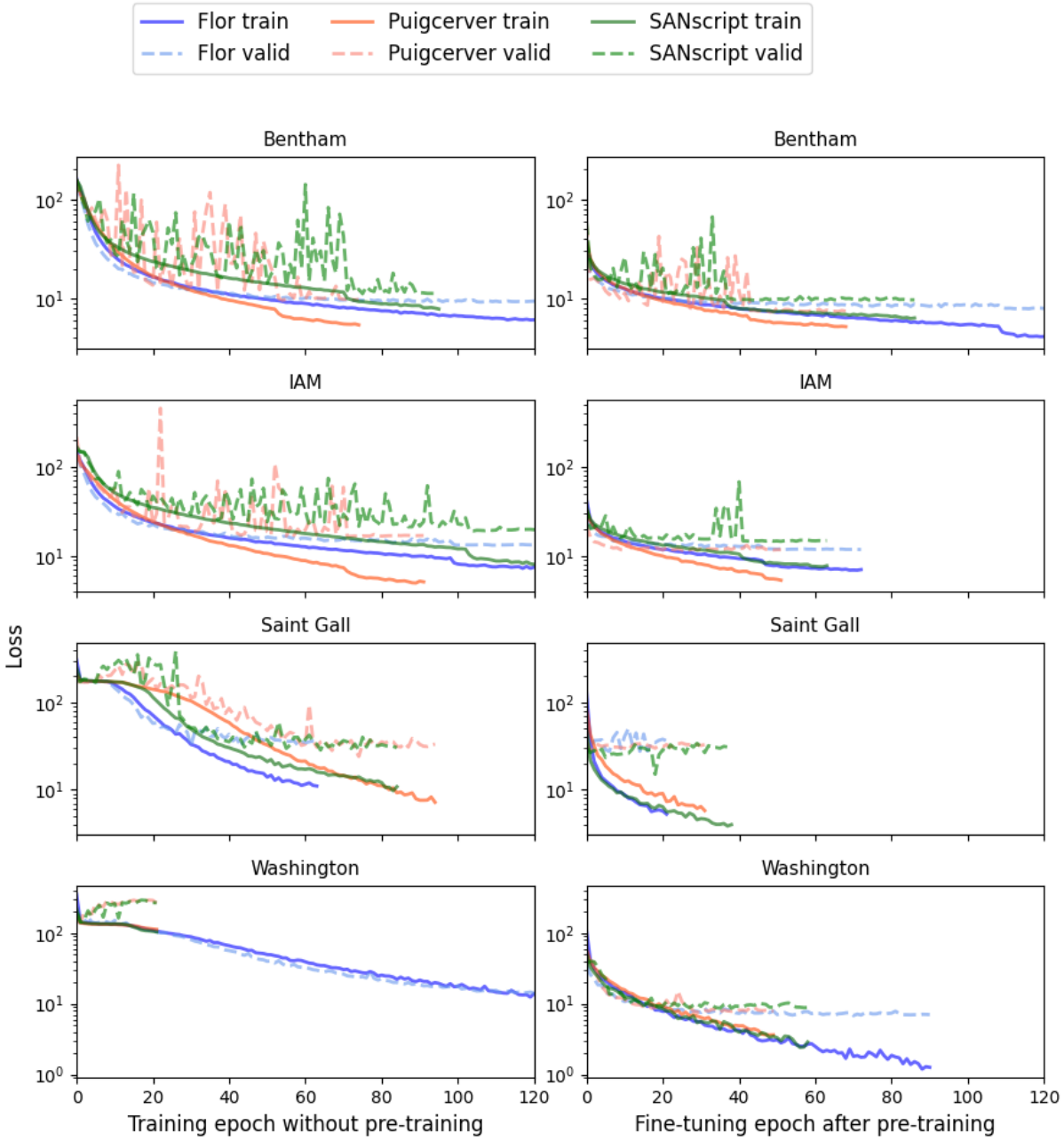


Figure 4.2: For each architecture and each public dataset, the training and validation loss curves when either training without any pre-training (Section 4.1), or when fine-tuning after pre-training (Section 4.2.2).

Tables 4.5 and 4.6 provide some examples of the transcriptions that were predicted by the fine-tuned models used in this Section, as well as by the models from Sections 4.1 and 4.2.1. Note how poorly the pre-trained Flor model performed without any fine-tuning; this was discussed in Section 4.2.1. Also, despite achieving high error rates, the untuned Puigcerver and SANscript models were still able to recognise the general forms of the text line; suggesting that they have learnt some generic understanding of handwriting during pre-training. Moreover, it appears that the prominent background noise in the Bentham dataset (Section 3.1.1) made it challenging for all models to transcribe certain words.

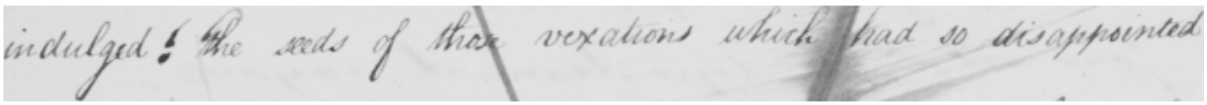


Figure 4.3: An example image from the Bentham testing set with the following ground-truth transcription: `indulged!  The seeds of those vexations which had so disappointed`

Training setup	Architecture	Prediction
Random initial weights	Puigcerver	`indulged of the seds of these verations which had so disappointed`
	Flor	`indulged !  the seeds of those vexations which fhad so disappointed`
	SANscript	`indulged 1 the seds of thon vexations whichad so disappointed`
Only pre-trained	Puigcerver	`winatuGed the seds of thir verations whicAUhad so disappointed`
	Flor	`moufelth mas yhomxaum unapypdsappontd`
	SANscript	`sivatuled / He ueds of MP vexations uhicAWlad i disappornted`
Pre-trained and fine-tuned	Puigcerver	`indulged ) The seeds of thie verations which had so disappointed`
	Flor	`indulged!  the seeds of thos vexations which fad so disappointed`
	SANscript	`indulged!  the seeds of those vexations which had so disappointed`

Table 4.5: For each architecture, examples of their predictions for the input image in Figure 4.3 when either trained from random initial weights (Section 4.1), only pre-trained synthetically (Section 4.2.1), or pre-trained and then fine-tuned (Section 4.2.2).

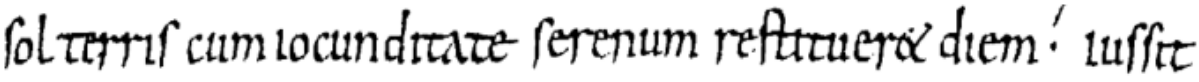


Figure 4.4: An example image from the Saint Gall testing set with the following ground-truth transcription: `sol terris cum iocunditate serenum restitueret diem iussit`

Training setup	Architecture	Prediction
Random initial weights	Puigcerver	`solterris cum iocunditate serenum reftitueret diem iussit`
	Flor	`sol terris cumiocunditate serenum reftitueret diempt iussit`
	SANscript	`sol terris cum iocunditate serenum restitueret duem lussit`
Only pre-trained	Puigcerver	`/ol-teprifcumocundrace-ferenumreAtcuepidem' wlhe`
	Flor	`p`
	SANscript	`bl-rerfcumocundrare-ferenum reAtruerxduem' uffe`
Pre-trained and fine-tuned	Puigcerver	`sol terris cum iocunditate serenum restitueret diem iussit`
	Flor	`sol terris cumiocunditate serenum restitueret diem iussit`
	SANscript	`sol terris cum iocunditate serenum restitueret diem iussit`

Table 4.6: For each architecture, examples of their predictions for the input image in Figure 4.4 when either trained from random initial weights (Section 4.1), only pre-trained synthetically (Section 4.2.1), or pre-trained and then fine-tuned (Section 4.2.2).

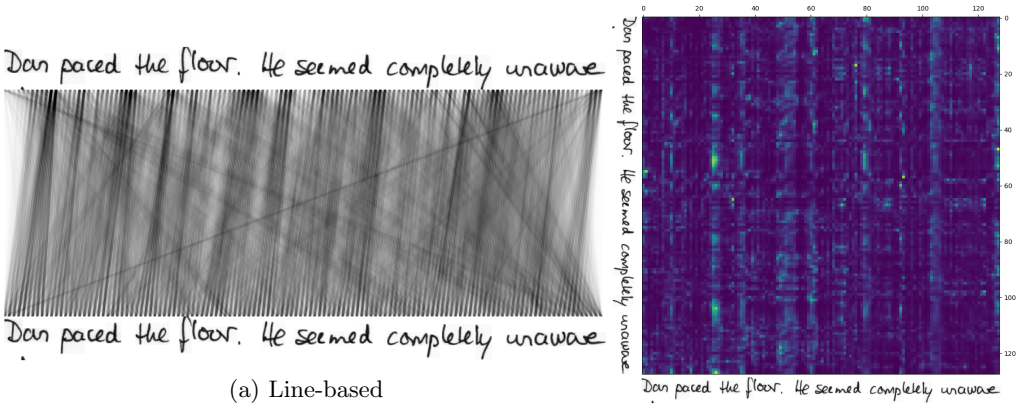
4.3 Visualising Self-Attention

One of the key advantage of using SANs over RNNs is their superior human interpretability. The features that are salient to a SAN are encoded explicitly in its attention weights (Section 2.4). Understanding what an RNN implicitly focuses on requires the backpropagation of loss gradients [60]. Visualising SANscripts attention weights has offered an interesting insight into its sequence modelling behaviour. To generate the attention weights, we used a SANscript model that was pre-trained synthetically and fine-tuned on the IAM dataset (Section 4.2).

We summarise Section 2.4 as a reminder: an attention mechanism takes sequences of query, key and value vectors as input. The attention weights are calculated by measuring the similarity of each query to all keys. The larger the attention weight between a query-key pair, the more semantically-related the two vectors are. The attention weights are used to transform the values for the final output. In a self-attention mechanism, the queries, keys and values are the same sequence; in the case of SANscript, the encoder features. SANscript has four SAN components in its decoder, each of which use four self-attention mechanisms (or heads) in parallel. Each head learns to consider its inputs from its own perspective. More technically, for each head, the inputs are projected to a different vector subspace. During training, these projections are optimised such that queries and keys which share semantic dependencies have similar vector representation and, therefore, share a large attention weight.

The attention weights of SANscript have been visualised using two different methods (Figure 4.5). The first, inspired by Figure 2.13, shows the queries (bottom) and keys (top) on two parallel axes. A line is drawn from each query position to each value position, the opacity of which reflects the magnitude of the attention weight calculated between them. The second method, inspired by Figure 2.10, visualises the weights as a simple 2D heatmap, with the queries on the x-axis and the keys on the y-axis. After all, the attention weights were already in the form of a 128×128 matrix (Section 2.4.3.1). The brighter the colour at a coordinate is, the larger the attention weight is between the corresponding query-key pair.

For both visualisation methods, we have used the same example image from the IAM test set. It has the following ground-truth transcription: `Dan paced the floor.  He seemed completely unaware` To aid the interpretation of our visualisations, this image has been aligned to their axes. This is valid, given that the image was resized to a width of 128 on input (Section 3.2.2), and that the encoder convolutions are applied locally and are spatially equivariant (Section 2.3). In other words, a position in the keys/values corresponds to roughly the same position along the width of the input image. We have found the line-based method to be better suited for visualising simpler dependency patterns (Figure 4.5a), and the heatmap-based method better for more complex patterns (Figure 4.5b).



(a) Line-based

(b) Heatmap-based

Figure 4.5: Examples of the different visualisation methods used. Note, each example shows the attention weights of a different head.

As demonstrated by the original Transformer authors, each head of a SAN learns to capture a different kind of linguistic dependency (Section 2.4.3.1). The same has been observed for SANscript, but in this case, the dependencies are visual-linguistic. For example, some heads clearly attend to local groups immediately before (Figure 4.7) or after (Figure 4.6) each query.

Others relate the spans of empty spaces between individual characters (Figure 4.8) or words (Figure 4.9) to the rest of the sequence. Certain heads even appear to capture real syntactic dependencies: `Dan` is the subject of both sentences, explicitly in the first and implicitly through `He` in the second (Figure 4.6). While some heads tend to distribute their attention weights to local regions (Figure 4.7), others are more diffuse and take into account global information (Figure 4.8). Each head of the same SAN layer is usually distinct, and some pairs of them are somewhat symmetric (Figures 4.7 and 4.6). This is interesting, as we do not specify for heads to be unique *a priori*. This behaviour emerges naturally during training.

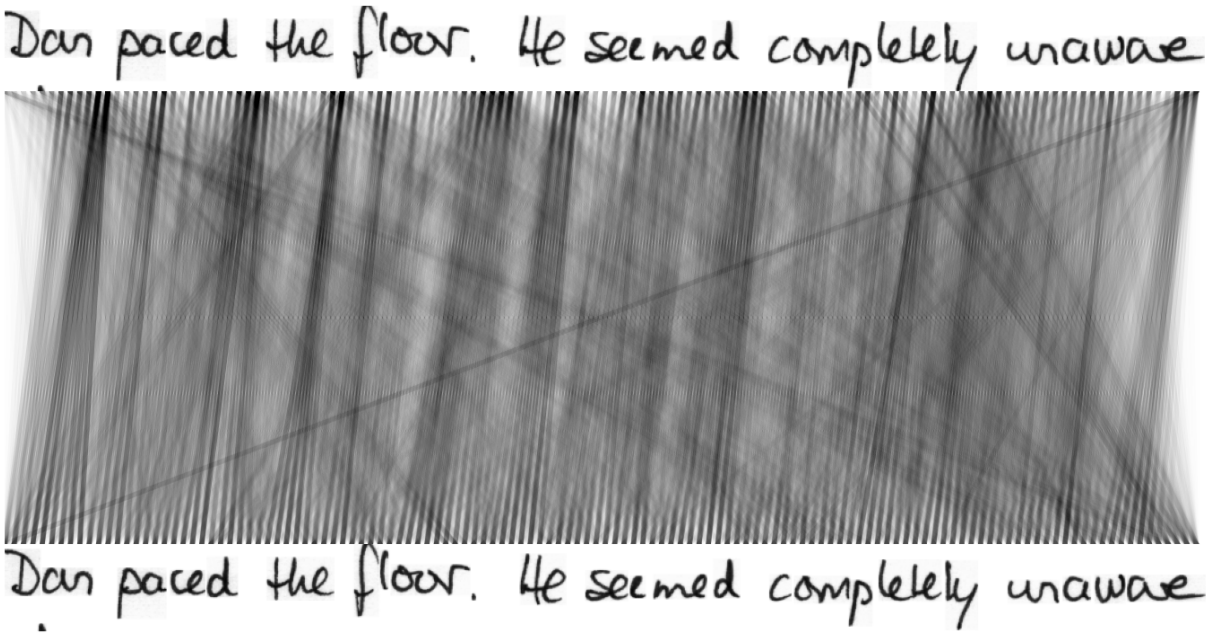


Figure 4.6: The self-attention weights calculated by the second attention head of the first SAN component in the SANscript decoder. Note the general forward-attending pattern, and the strong connections from `the floor` and `completely unaware` to `Dan`.

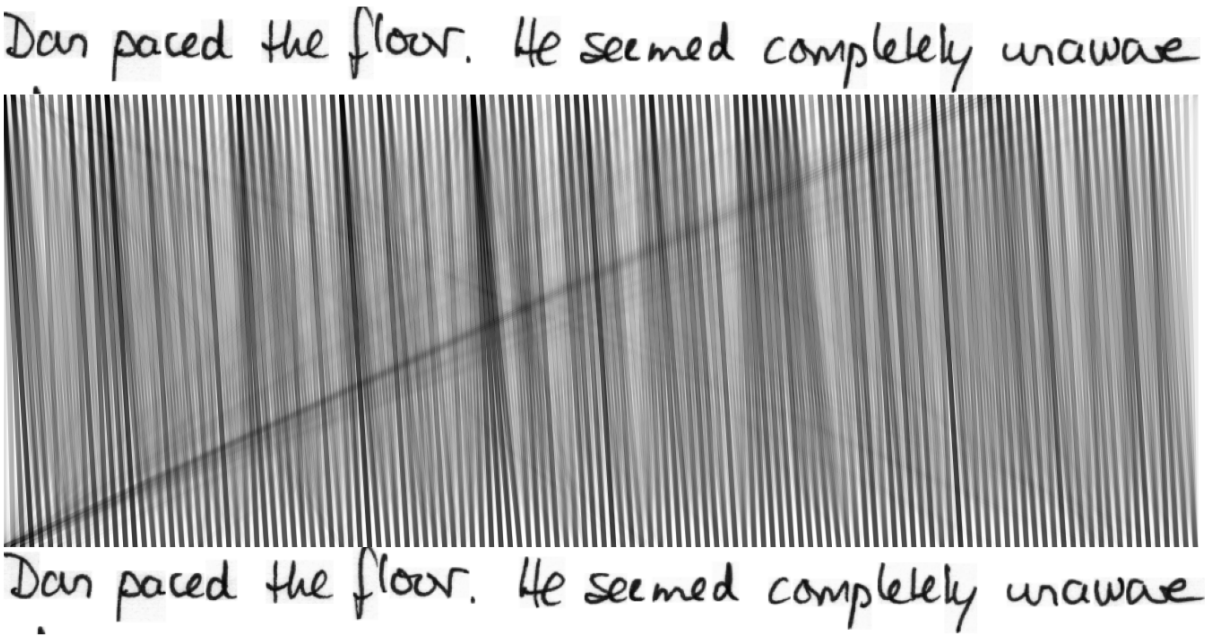


Figure 4.7: The self-attention weights calculated by the third attention head of the first SAN component in the SANscript decoder. Note the backward-attending pattern and the saliency of the full-stop.

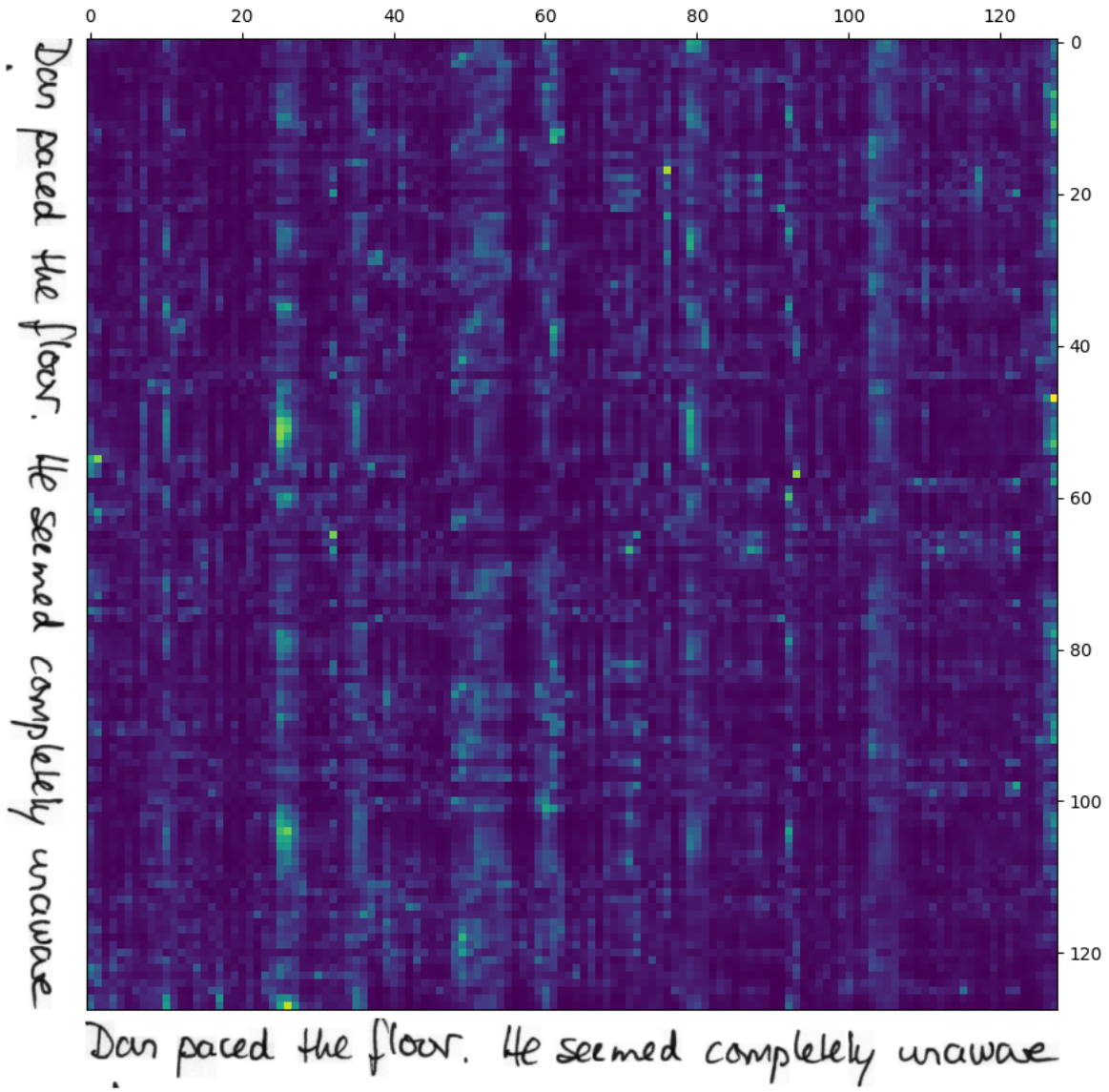


Figure 4.8: The self-attention weights calculated by the first attention head of the first SAN component in the SANscript decoder. Note the saliency of the spaces between individual characters.

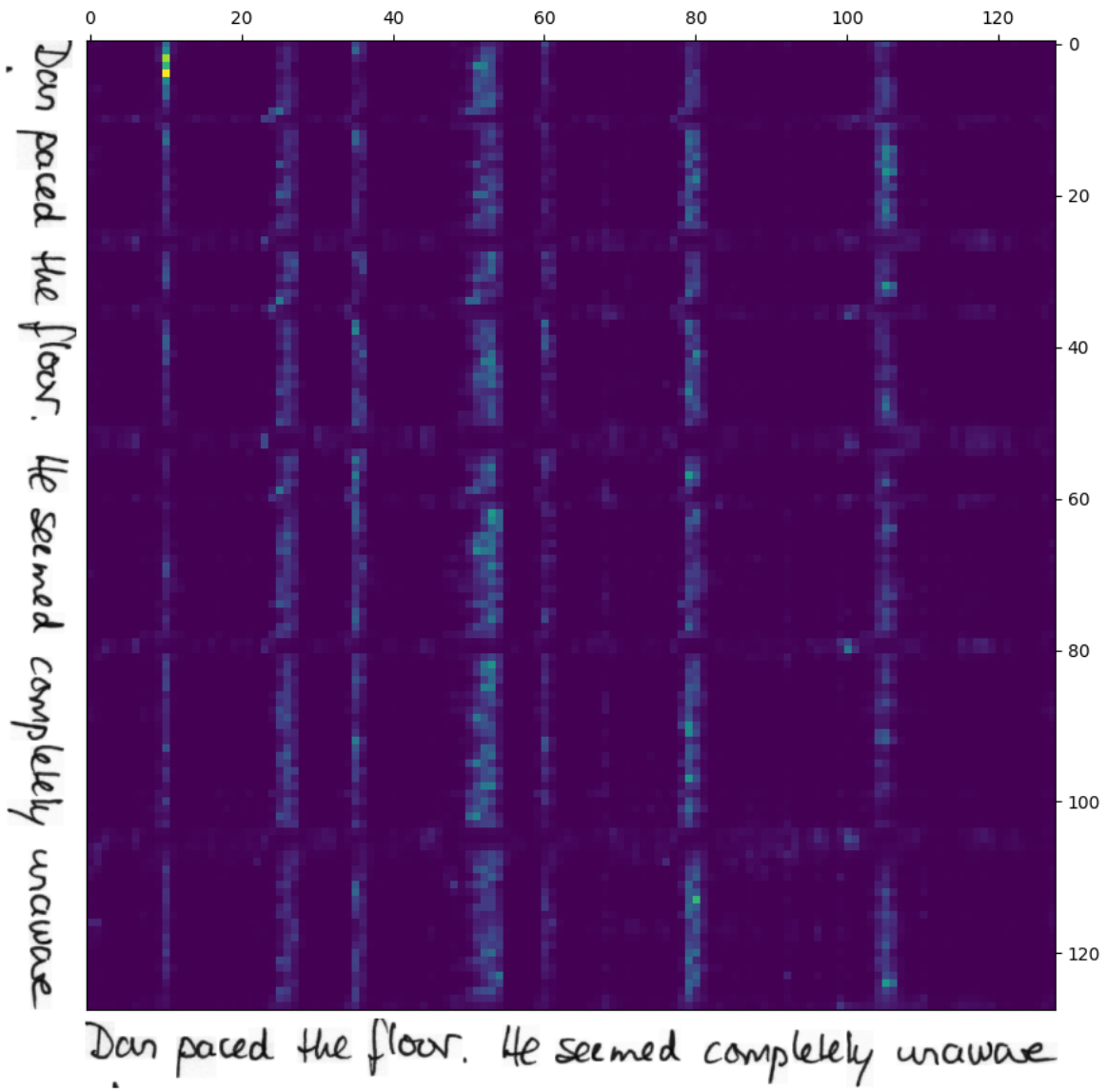


Figure 4.9: The self-attention weights calculated by the fourth attention head of the first SAN component in the SANscript decoder. Note the saliency of the spaces between whole words.

We conjecture that a stack of SAN layers, such as in SANscript, learns to model hierarchies of dependencies; similarly to the features in a multi-stage CNN (Section 2.3). More specifically, the level of semantic abstraction increases with depth: earlier SAN layers capture lower-level patterns, whilst later SANs operate on higher-levels. For example, the third head of the first SAN layer connects local neighborhoods of positions (Figure 4.7). In comparison, the fourth head of the third layer connects regions corresponding to whole words with one another, whilst seemingly ignoring the spaces between them (Figure 4.10). By applying the attention weights to the values, a SAN encodes the dependencies it has extracted directly into its output. The input of a later SAN layer is based on the outputs of all preceding SANs. Therefore, subsequent SAN layers are able to consider wider ranges of semantic structure at once and, in turn, combine them into increasingly complex patterns. This is somewhat reminiscent of the pooling layers in a CNN: by down-sampling intermediate features maps, subsequent filters can effectively consider larger areas of the original input and extract more abstract patterns (Section 2.3). Further visualisations and their interpretations are provided in the pages that follow.

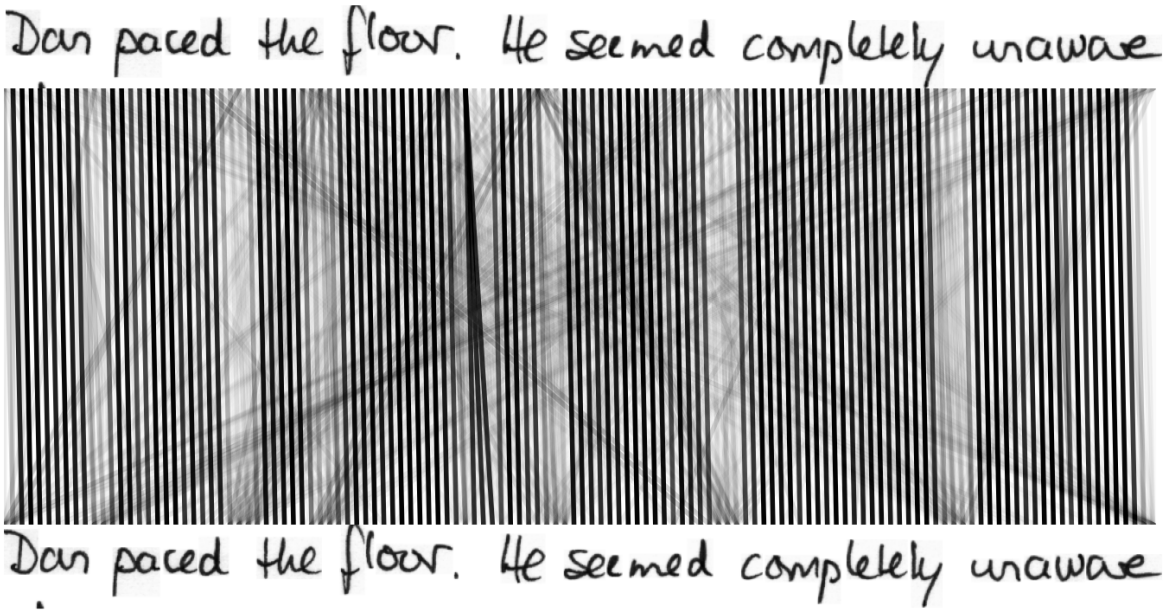


Figure 4.10: The self-attention weights calculated by the fourth attention head of the third SAN component in the SANscript decoder. Note the alignment between whole words but not the spaces between them.

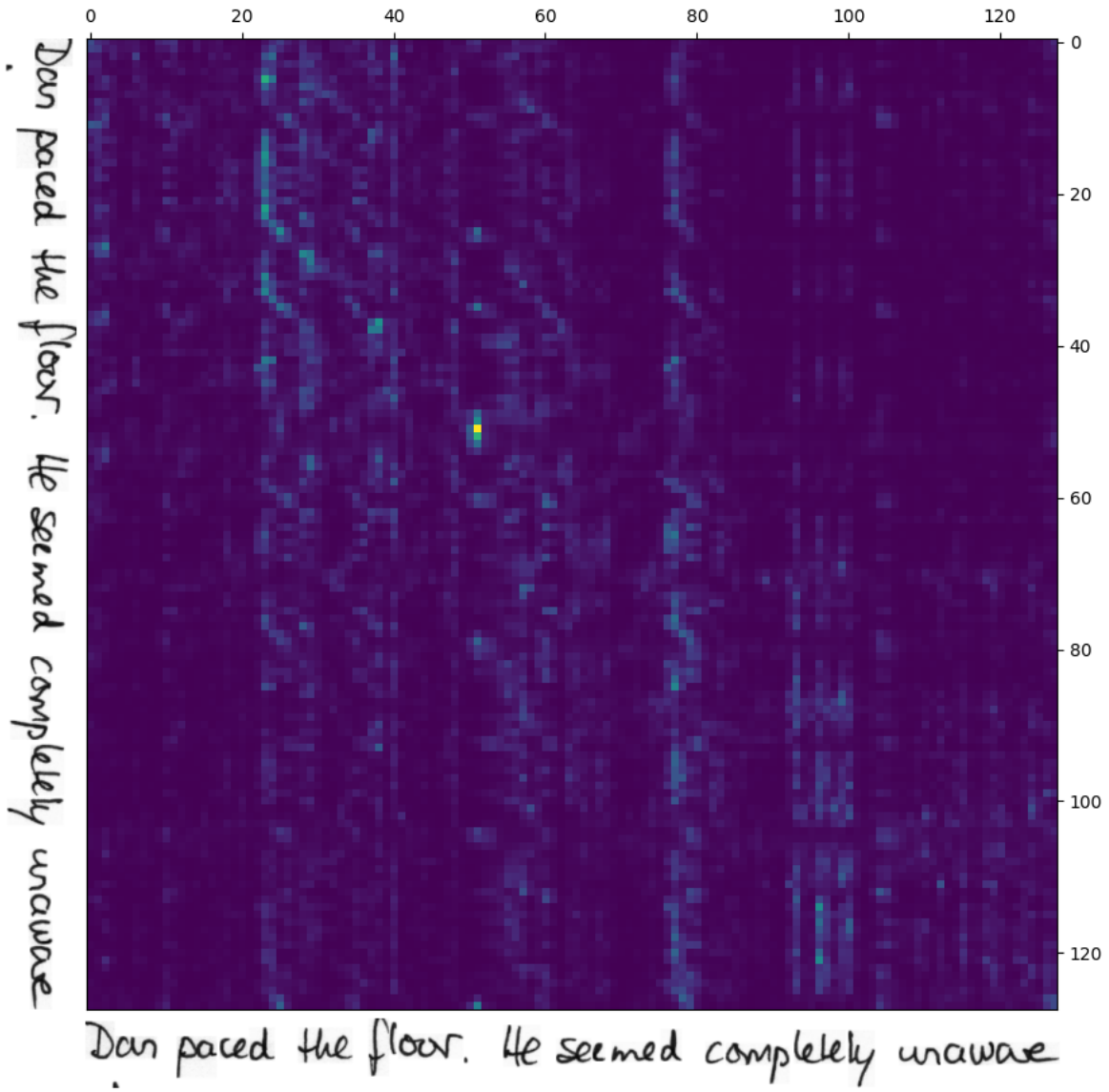


Figure 4.11: The self-attention weights calculated by the fourth attention head of the second SAN component in the SANscript decoder. Note the saliency of the full-stop.

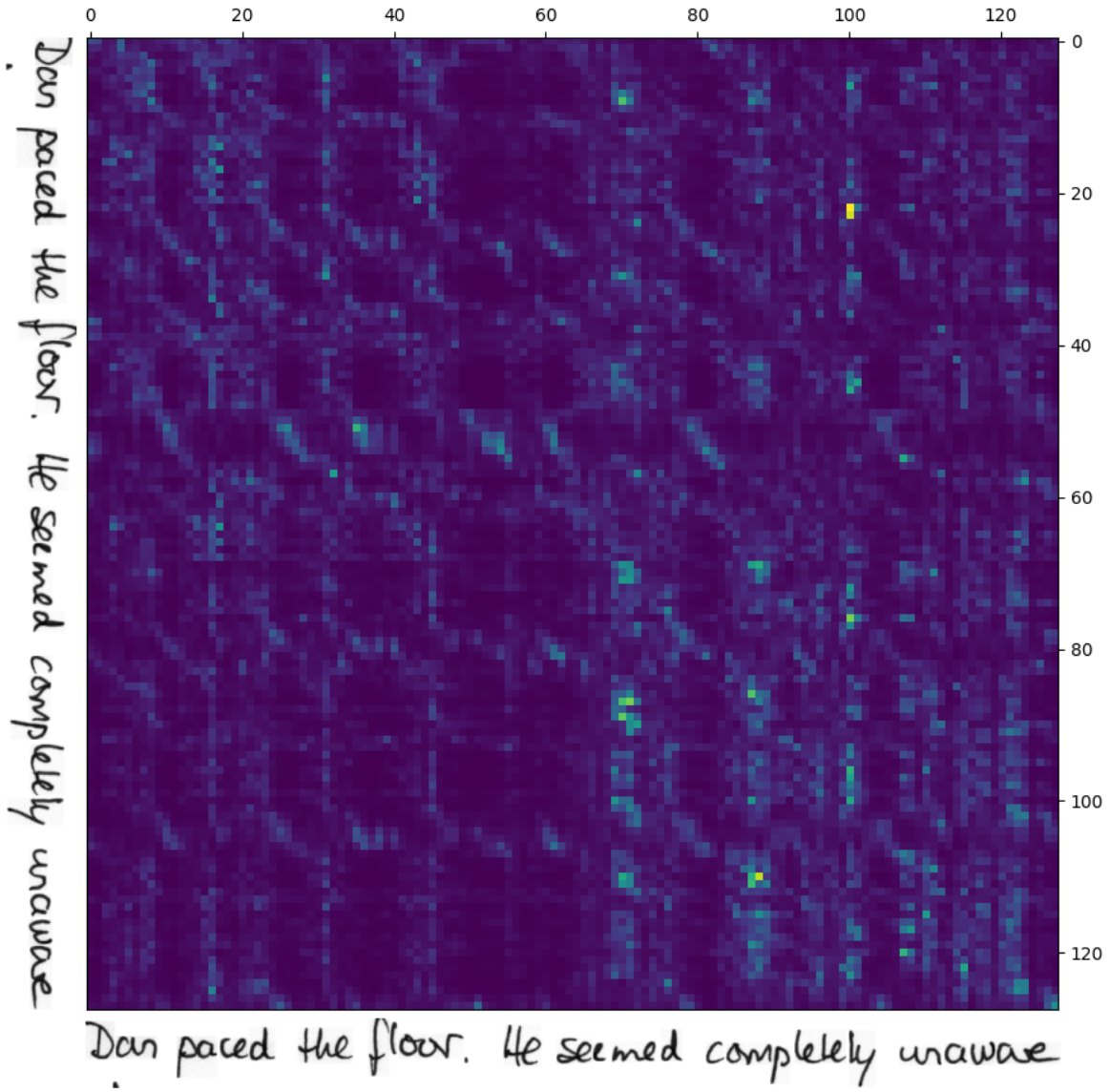


Figure 4.12: The self-attention weights calculated by the second attention head of the second SAN component in the SANscript decoder. Note the increased complexity relative to previous heads: it appears to consider whole words and the spaces between them, as well as distinguish between the separate sentences.

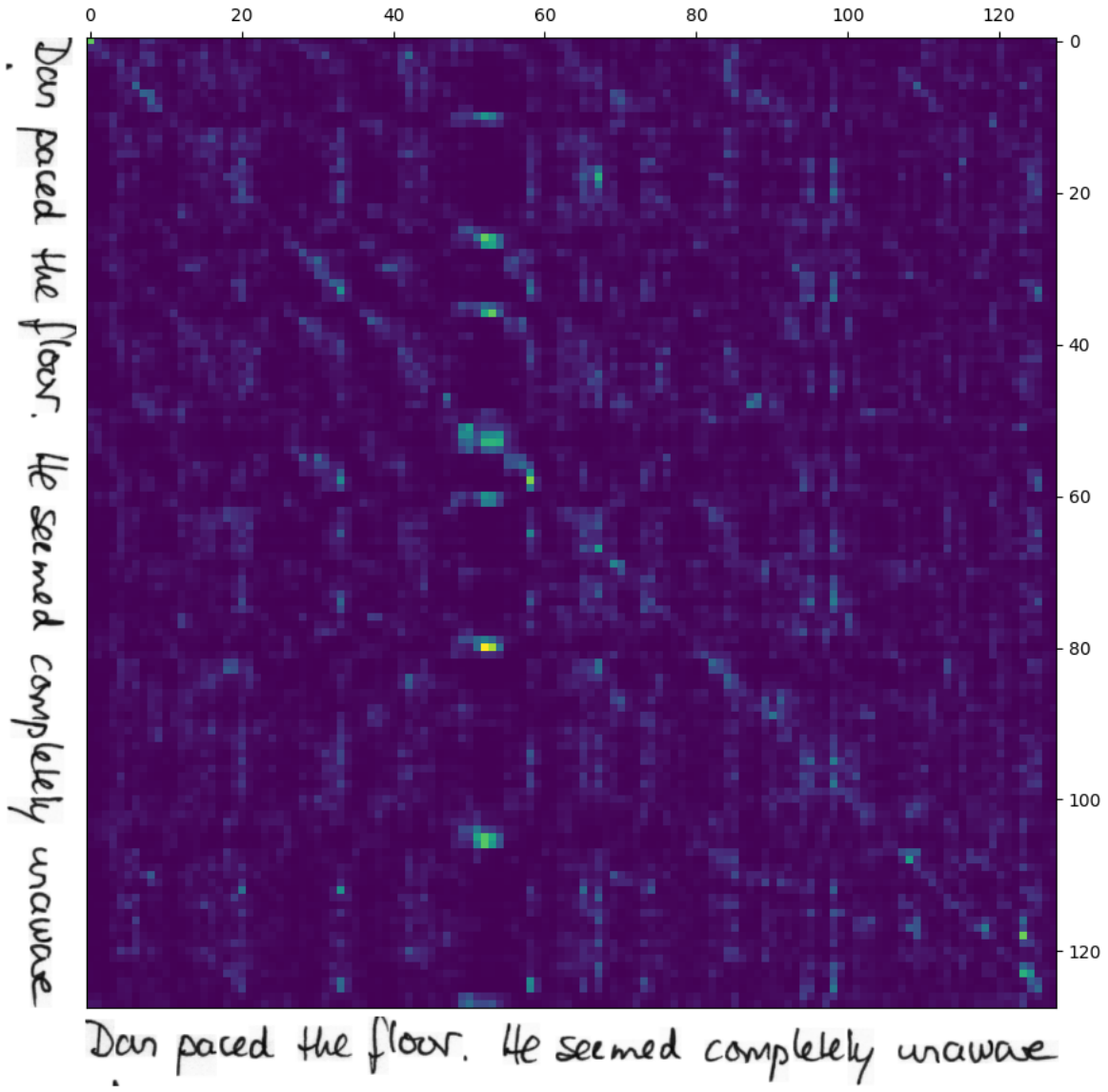


Figure 4.13: The self-attention weights calculated by the second attention head of the third SAN component in the SANscript decoder. Note how it shares many similar patterns to the previous heads (Figures 4.11 and 4.12), supporting the idea that SANs learn hierarchies of features.

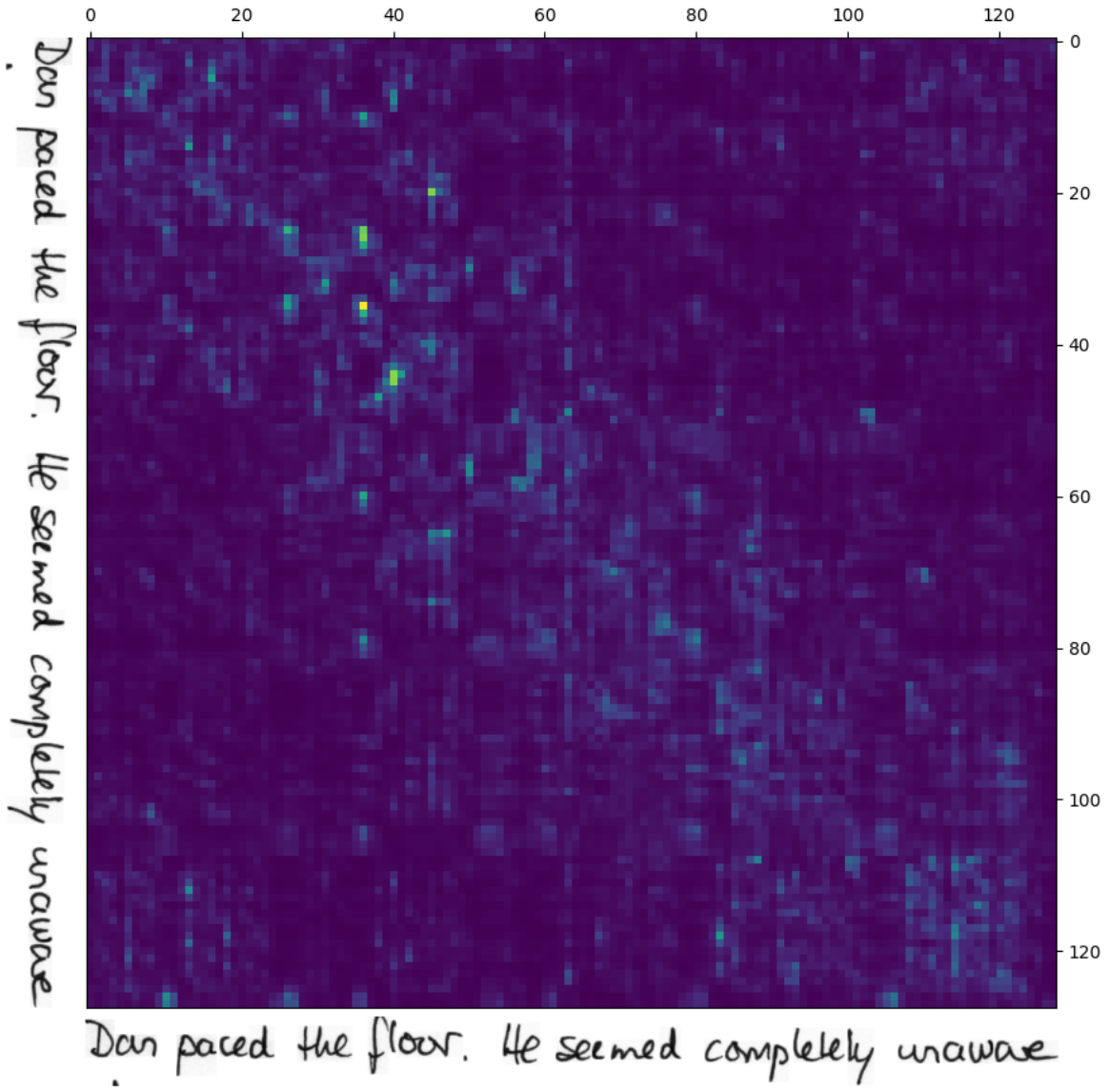


Figure 4.14: The self-attention weights calculated by the second attention head of the fourth SAN component in the SANscript decoder. Note the diffuseness of the head, taking into account global information.

Chapter 5

Conclusion

Here, we provide a summary (Section 5.1) and discussion into the potential impacts (Section 5.2) of our work, as well as suggest some potential future work on this topic (Section 5.3).

5.1 Summary of Contributions

As part of this project, we have notably:

- proposed SANscript; the first non-autoregressive and nonrecurrent optical model architecture for offline HTR, based on self-attention networks (Section 2.6);
- demonstrated that SANscript can closely approximate the current state-of-the-art recognition rates, while requiring significantly less time to train than the leading recurrent architectures (Section 4.2.2);
- created a large, generic and entirely synthetic dataset, which can be used to pre-train and improve the performance of any architecture (Section 3.1.5);
- demonstrated that a pre-trained SANscript model can be adapted to a new handwriting style with limited labelled data more effectively than a pre-trained recurrent model can (Section 4.2.2); and
- contributed to an open-source HTR framework (Section 3.2.1).

5.2 Potential Impact

Our findings are relevant for all those who are dependent on HTR technology: using SANscript reduces the cost of training by requiring less time and hand-labelled data. One key audience that could benefit from this are the users and providers of the Transkribus HTR platform (Section 1.1). Currently, Transkribus depends on recurrent models to provide its HTR service, the use of which has been identified as a major bottleneck [108]. A recent case study of Transkribus [112] has suggested the following:

> "*The next goals for computer scientists are to optimise methods so that they require less training data to achieve comparable results, and build generic models that can work on similar fonts and hands. In the future 'out-of-the-box' models could make it easier for even more users to engage with and benefit from HTR...*"

Our work takes a significant step towards meeting this goal. Its findings are also applicable outside of HTR research: SANs can potentially replace RNNs in any neural network architecture.

As is discussed later in Section 5.3, we suggest developing a larger synthetic dataset to improve the recognition of SANscript further. This is common for SAN-based architectures. In the wider research community, the efficient training provided by SANs has encouraged the use of datasets and models of exponentially increasing size [35, 128, 148]. In fact, between 2012 and 2018, the computational resources required to train a state-of-the-art deep learning system increased by $300,000\times$ [4]. However, bigger is not better in all regards. At the time of writing, GPT3 is one of the most powerful models for NLP [22]; it is Transformer-based (Section 1.3.3) and has over 1.75 billion parameters. It has been estimated that training GPT3 would require over 300 GPUs running non-stop for 90 days, the energy consumption of which can emit as much carbon dioxide as driving a car for over 7×10^5 kilometres [5]. Collectively

reducing humanity's carbon footprint is more important than ever, given the existential threats posed by the on-going climate crisis [122]. We believe that this trend in deep learning research is unsustainable, and that future advancements must attempt to reduce this resource-intensiveness.

5.3 Future Work

We hope that our work encourage future work on SAN-based optical models, and we recommend researching into improvements in the following areas:

Architecture

As proposed, SANscript uses the convolutional encoder of an existing state-of-the-art recurrent architecture (Section 2.6.2). This was decided so that the impact of replacing RNNs with SANs could be more accurately evaluated. However, it is likely that a more optimal configuration of convolutional layers exists for SANscript. This could take inspiration from recent advancements in convolutional networks for image processing [69, 73].

Experimental setup

In our experiments, the same setup was used to train and evaluate all architectures; for similar reasons as given above. This experimental setup can have a significant impact on the final performance of a model, especially for SANscript (Section 3.4.8). For example, SAN-based architectures are known to be unstable during training and can require the careful scheduling of learning rates. While this was not the case in our project, other work suggests that the problem is more pronounced in larger architectures, and that the hyperparameters of the scheduling function are dependent on the training dataset [124, 133]. Our research into this was limited, but we expect that a deeper exploration of how SANscript is trained will lead to better results.

Implementation

SANscript achieved faster speeds over its recurrent counterparts using a custom implementation of self-attention networks, written in a high-level deep learning framework. It is possible that this speedup could be improved by re-implementing the SANs in a lower-level GPU-acceleration library. We recommend doing the same for gated convolutions; despite improving recognition, the implementation of them we used was significant bottleneck for a SANscript (Section 3.4.7).

Synthetic dataset

The synthetic dataset (Section 3.1.5) was incredibly useful in our experiments; using it for pre-training can greatly improve the performance of HTR models and was necessary for SANscript to achieve state-of-the-art performance (Section 4.2.2). However, due to resource constraints, it was somewhat limited in size and complexity. This is probably the most profitable area of improvement. Our same methodology can be used to generate an even larger synthetic dataset. Pre-training SANscript on it may require multiple GPUs, but will provide the model a richer collection of linguistic data to learn from. As for the quality of the generated text, this could be improved through further image augmentations or by adding more background noise elements. There even exists neural network models for generating impressively realistic samples of handwriting (Figure 5.1) [51, 58, 80]. While promising, these methods have their limitations; we therefore recommend to use a mix of image generation techniques.

Figure 5.1: An entirely synthetic image of handwriting generated using S. Vasquez's implementation [147] of the neural architecture proposed by A. Graves [58], trained on the IAM dataset.

Bibliography

[1] M. A. Abuzaraida, A. M. Zeki, and A. M. Zeki. Feature extraction techniques of online handwriting arabic text recognition. In *2013 5th International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, pages 1–7, 2013.

[2] Advanced Computing Research Centre. Bluecrystal phase 4. https://www.acrc.bris.ac.uk/acrc/phase4.htm, 2017.

[3] I. Ahmad and G. Fink. Training an Arabic handwriting recognizer without a handwritten training data set. *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 476–480, 2015.

[4] D. Amodei, D. Hernandez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever. AI and Compute. https://openai.com/blog/ai-and-compute/, 2018.

[5] L. F. W. Anthony, B. Kanding, and R. Selvan. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. *CoRR*, abs/2007.03051, 2020.

[6] J. C. Aradillas Jaramillo, J. J. Murillo-Fuentes, and P. M. Olmos. Boosting Handwriting Text Recognition in Small Databases with Transfer Learning. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 429–434, 2018.

[7] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[8] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[9] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. End-to-End Attention-based Large Vocabulary Speech Recognition. *CoRR*, abs/1508.04395, 2015.

[10] B. Balci, D. Saadati, and D. Shiferaw. Handwritten text recognition using deep learning. *CS231n: Convolutional Neural Networks for Visual Recognition, Stanford University, Course Project Report, Spring*, pages 752–759, 2017.

[11] A. Baldominos, Y. Saez, and P. Isasi. A Survey of Handwritten Character Recognition with MNIST and EMNIST. *Applied Sciences*, 9(15), 2019.

[12] E. Belval. Text recognition data generator. https://github.com/Belval/TextRecognitionDataGenerator, 2019.

[13] Y. Bengio, Y. LeCun, and D. Henderson. Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden markov models. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1994.

[14] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[15] A. K. Bhunia, A. Das, P. S. R. Kishore, and P. P. Roy. Handwriting Recognition in Low-resource Scripts using Adversarial Learning, 2019.

[16] W. W. Bledsoe and I. Browning. Pattern Recognition and Reading by Machine. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), page 225–232, New York, NY, USA, 1959. Association for Computing Machinery.

[17] M. Bleeker and M. de Rijke. Bidirectional Scene Text Recognition with a Single Decoder. *CoRR*, abs/1912.03656, 2019.

[18] T. Bluche, J. Louradour, and R. Messina. Scan, Attend and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention. *CoRR*, abs/1604.03286, 2016.

[19] T. Bluche and R. Messina. Gated Convolutional Recurrent Neural Networks for Multilingual Handwriting Recognition. pages 646–651, 11 2017.

[20] T. Bluche, H. Ney, and C. Kermorvant. Feature extraction with convolutional neural networks for handwritten word recognition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 285–289, 2013.

[21] T. Bluche, H. Ney, and C. Kermorvant. Tandem HMM with convolutional neural network for handwritten word recognition. pages 2390–2394, 10 2013.

[22] 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 Da. Amodei. Language Models are Few-Shot Learners, 2020.

[23] H. Bunke, S. Bengio, and A. Vinciarelli. Offline recognition of unconstrained handwritten texts using HMMs and statistical language models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(6):709–720, 2004.

[24] C. M. Calvo. *Development and experimentation of a deep learning system for convolutional and recurrent neural networks*. PhD thesis, 2018.

[25] N. Carion, F. Massa, Gabriel S., N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-End Object Detection with Transformers, 2020.

[26] S. Chaudhari, G. Polatkan, R. Ramanath, and V. Mithal. An attentive survey of attention models. *CoRR*, abs/1904.02874, 2019.

[27] K. Chen, C. Chen, and C. Chang. Efficient Illumination Compensation Techniques for Text Images. *Digit. Signal Process.*, 22(5):726–733, September 2012.

[28] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, abs/1406.1078, 2014.

[29] A. Chowdhury and L. Vig. An Efficient End-to-End Neural Model for Handwritten Text Recognition. *CoRR*, abs/1807.07965, 2018.

[30] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR*, abs/1412.3555, 2014.

[31] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926, 2017.

[32] R. J. C. d'Arce. `handwritten-text-recognition`. https://github.com/rafaeljcdarce/handwritten-text-recognition, 2021.

[33] A. Das, J. Li, G. Ye, R. Zhao, and Y. Gong. Advancing Acoustic-to-Word CTC Model with Attention and Mixed-Units. *CoRR*, abs/1812.11928, 2018.

[34] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. Language Modeling with Gated Convolutional Networks. *CoRR*, abs/1612.08083, 2016.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[36] L. Dong, S. Xu, and B. Xu. Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *CoRR*, abs/2010.11929, 2020.

[38] J. A. Edwards. *Easily Adaptable Handwriting Recognition in Historical Manuscripts*. PhD thesis, EECS Department, University of California, Berkeley, May 2007.

[39] M. Ernestus. Elastic distortion. https://gist.github.com/ernestum/601cdf56d2b424757de5, 2016.

[40] European Commision. tranScriptorium. https://cordis.europa.eu/project/id/600707, 2019.

[41] European Commision. Opening up Europe's written cultural heritage to people all over the world. https://cordis.europa.eu/article/id/411587-opening-up-europe-s-written-cultural-heritage-to-people-all-over-the-world, 2020.

[42] European Commision. Periodic Reporting for period 3 - READ (Recognition and Enrichment of Archival Documents). https://cordis.europa.eu/project/id/674943/reporting, 2020.

[43] European Commision. Recognition and Enrichment of Archival Documents. https://cordis.europa.eu/project/id/674943, 2020.

[44] X. Feng, H. Yao, Y. Qi, J. Zhang, and S. Zhang. Scene Text Recognition via Transformer, 2020.

[45] A. Fischer, V. Frinken, A. Fornés, and H. Bunke. Transcription Alignment of Latin Manuscripts Using Hidden Markov Models. In *Proceedings of the 2011 Workshop on Historical Document Imaging and Processing*, page 29–36. Association for Computing Machinery, 2011.

[46] A. Fischer, A. Keller, V. Frinken, and H. Bunke. Lexicon-free handwritten word spotting using character HMMs. *Pattern Recognition Letters*, 33(7):934–942, 2012.

[47] A. Flor. `handwritten-text-recognition`. https://github.com/arthurflor23/handwritten-text-recognition, 2020.

[48] A. Flor, B. L. D. Bezerra, E. B. Lima, and A. H. Toselli. HDSR-Flor: A robust end-to-end system to solve the Handwritten Digit String Recognition problem in real complex scenarios. *IEEE Access*, 8:208543–208553, 2020.

[49] A. Flor, B. L. D. Bezerra, and A. H. Toselli. Towards the Natural Language Processing as Spelling Correction for Offline Handwritten Text Recognition Systems. *Applied Sciences*, 10(21), 2020.

[50] A. Flor, 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)*, pages 54–61, 2020.

[51] S. Fogel, H. Averbuch-Elor, S. Cohen, S. Mazor, and R. Litman. ScrabbleGAN: Semi-Supervised Varying Length Handwritten Text Generation. *CoRR*, abs/2003.10557, 2020.

[52] B. Gatos, G. Louloudis, T. Causer, K. Grint, V. Romero, J. A. Sánchez, A. H. Toselli, and E. Vidal. Ground-truth production in the transcriptorium project. In *2014 11th IAPR International Workshop on Document Analysis Systems*, pages 237–241, 2014.

[53] R. Ghosh, C. Panda, and P. Kumar. Handwritten Text Recognition in Bank Cheques. In *2018 Conference on Information and Communication Technology (CICT)*, pages 1–6, 2018.

[54] R. Girdhar, J. Carreira, C. Doersch, and A. Zisserman. Video Action Transformer Network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[55] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[56] I. Goodfellow, Y. Bengio, and A. Courville. 6.2. 2.3 softmax units for multinoulli output distributions. *Deep learning*, pages 180–184, 2016.

[57] A. Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012.

[58] A. Graves. Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013.

[59] A. Graves, S. Fernández, 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*, ICML 2006, page 369–376, New York, NY, USA, 2006. Association for Computing Machinery.

[60] A. Graves, S. Fernández, M. Liwicki, H. Bunke, and J. Schmidhuber. Unconstrained Online Handwriting Recognition with Recurrent Neural Networks. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, page 577–584, 2007.

[61] A. Graves, S. Fernández, and J. Schmidhuber. Multi-Dimensional Recurrent Neural Networks. *CoRR*, abs/0705.2011, 2007.

[62] A. Graves and N. Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1764–1772, Bejing, China, 22–24 Jun 2014. PMLR.

[63] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, 2009.

[64] A. Graves and J. Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2009.

[65] R. L. Grimsdale and J. M. Bullingham. Character Recognition by Digital Computer using a Special Flying-Spot Scanner. *The Computer Journal*, 4(2):129–136, 01 1961.

[66] R. Grosse. Lecture 15: Exploding and vanishing gradients. *University of Toronto Computer Science*, 2017.

[67] Ç. Gülçehre, K. Cho, R. Pascanu, and Y. Bengio. Learned-norm pooling for deep neural networks. *CoRR*, abs/1311.1780, 2013.

[68] I. Hadji and R. P. Wildes. What do we understand about convolutional networks? *CoRR*, abs/1803.08834, 2018.

[69] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[70] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, abs/1502.01852, 2015.

[71] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent.

[72] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[73] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[74] M. Huckvale. Purpose: the missing link in speech and handwriting recognition. In *AISB Workshop on Computational Linguistics for Speech and Handwriting Recognition*, 1994.

[75] K. Hwang and W.Sung. Character-Level Incremental Speech Recognition with Recurrent Neural Networks. *CoRR*, abs/1601.06581, 2016.

[76] V. Iashin and E. Rahtu. A Better Use of Audio-Visual Cues: Dense Video Captioning with Bi-modal Transformer. *arXiv e-prints*, 2020.

[77] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[78] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman. Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition. *CoRR*, abs/1406.2227, 2014.

[79] L. Kang, P. Riba, M. Rusiñol, A. Fornés, and M. Villegas. Pay Attention to What You Read: Non-recurrent Handwritten Text-Line Recognition, 2020.

[80] L. Kang, P. Riba, Y. Wang, M. Rusiñol, A. Fornés, and M. Villegas. GANwriting: Content-Conditioned Generation of Styled Handwritten Word Images. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 273–289. Springer International Publishing, 2020.

[81] L. Kang, J. I. Toledo, P. Riba, M. Villegas, A. Fornés, and Ma. Rusiñol. Convolve, Attend and Spell: An Attention-based Sequence-to-Sequence Model for Handwritten Word Recognition. In *Pattern Recognition*, pages 459–472. Springer International Publishing, 2019.

[82] D. Keysers, T. Deselaers, H. A. Rowley, L. Wang, and V. Carbune. Multi-Language Online Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(06):1180–1194, jun 2017.

[83] G. Kim, V. Govindaraju, and S. N. Srihari. An architecture for handwritten text recognition systems. *International Journal on Document Analysis and Recognition*, 2(1):37–44, 1999.

[84] S. Kim, T. Hori, and S. Watanabe. Joint CTC-Attention based End-to-End Speech Recognition using Multi-task Learning. *CoRR*, abs/1609.06773, 2016.

[85] W. Kim, B. Son, and I. Kim. ViLT: Vision-and-Language Transformer Without Convolution or Region Supervision, 2021.

[86] S. Kobayashi. Homemade BookCorpus. https://github.com/soskek/bookcorpus, 2018.

[87] A. L Koerich, R. Sabourin, and C. Y Suen. Large vocabulary off-line handwriting recognition: A survey. *Pattern Analysis & Applications*, 6(2):97–121, 2003.

[88] P. Krishnan and C. V. Jawahar. HWNet v2: An Efficient Word Image Representation for Handwritten Documents. *CoRR*, abs/1802.06194, 2018.

[89] A. Krizhevsky, I. Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[90] B. Kumar and V. Singh. Feature extraction techniques for handwritten text in various scripts: A survey. *International Journal of Soft Computing and Engineering*, Volumn 3:238–241, 03 2013.

[91] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019.

[92] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1990.

[93] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[94] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[95] Y. LeCun, C. Cortes, and C. J. C. Burges. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[96] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256, 2010.

[97] G. Leifert, T. Strauß, T. Grüning, and R. Labahn. CITlab ARGUS for historical handwritten documents. *CoRR*, abs/1605.08412, 2016.

[98] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[99] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[100] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.

[101] N. T. Ly, C. T. Nguyen, and M. Nakagawa. Training an End-to-End Model for Offline Handwritten Japanese Text Recognition by Generated Synthetic Patterns. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 74–79, 2018.

[102] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[103] A. Makhzani and B. Frey. k-sparse autoencoders. 2013.

[104] U. Marti and H. Bunke. *Using a Statistical Language Model to Improve the Performance of an HMM-Based Cursive Handwriting Recognition Systems*, page 65–90. World Scientific Publishing Co., Inc., USA, 2001.

[105] U. Marti and H. Bunke. The IAM-database: an English sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5:39–46, 2002.

[106] S. Marukatat, T. Artieres, R. Gallinari, and B. Dorizzi. Sentence recognition through hybrid neuro-Markovian modeling. In *Proceedings of Sixth International Conference on Document Analysis and Recognition*, pages 731–735, 2001.

[107] J. Michael, R. Labahn, T. Grüning, and J. Zöllner. Evaluating Sequence-to-Sequence Models for Handwritten Text Recognition. *CoRR*, abs/1903.07377, 2019.

[108] J. Michael, M. Weidemann, and R. Labahn. D7.9 HTR Engine Based on NNs P3: Optimizing speed and performance. https://readcoop.eu/wp-content/uploads/2018/12/D7.9_HTR_NN_final.pdf, 2018.

[109] B. Moysset, T. Bluche, M. Knibbe, M. F. Benzeghiba, R. Messina, J. Louradour, and C. Kermorvant. The A2iA Multi-lingual Text Recognition System at the Second Maurdor Evaluation. In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 297–302, 2014.

[110] T. Myers and P. Spiegel. Image metamorphosis by affine transformations. 2005.

[111] G. Mühlberger. D2.6. dissemination and awareness plan. http://transcriptorium.eu/pdfs/deliverables/tranScriptorium-D7.3-31December2015.pdf, 2019.

[112] G. Mühlberger, Louise Seaward, Melissa Terras, Sofia Ares Oliveira, Bosch Vicente, Sebastian Colutto, Hervé Déjean, Markus Diem, Stefan Fiel, Basilis Gatos, Tobias Grüning, Albert Greinoecker, Guenter Hackl, Vili Haukkovaara, Gerhard Heyer, Lauri Hirvonen, Tobias Hodel, Matti Jokinen, Philip Jokinen, Mario Kallio, Frederic Kaplan, Florian Kleber, Roger Labahn, Eva Maria Lang, Sören Laube, Gundram Leifert, Georgios Louloudis, Rory McNicholl, Jean-Luc Meunier, Elena Mühlbauer, Nathanael Philipp, Ioannis Pratikakis, Joan Puigcerver Pérez, Hannelore Putz, George

Retsinas, Verónica Romero, Robert Sablatnig, Joan Andreu Sánchez, Philip Schofield, Georgios Sfikas, Christian Sieber, Nikolaos Stamatopoulos, Tobias Strauss, Tamara Terbul, Alejandro Hector Toselli, Berthold Ulreich, Mauricio Villega, Enrique Vidal, Johanna Walcher, Max Weidemann, Herbert Wurster, Konstantinos Zagoris, Maximilian Bryan, and Johannes Michael. Transforming scholarship in the archives through handwritten text recognition: Transkribus as a case study. *Journal of Documentation*, 75(5):954–976, December 2019.

[113] R. Nallapati, B. Xiang, and B. Zhou. Sequence-to-Sequence RNNs for Text Summarization. *CoRR*, abs/1602.06023, 2016.

[114] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[115] D. Nguyen, N. Tran, and H. Le. Improving Long Handwritten Text Line Recognition with Convolutional Multi-way Associative Memory. *CoRR*, abs/1911.01577, 2019.

[116] B. Ogilvie. Scientific archives in the age of digitization. *Isis*, 107(1):77–85, 2016.

[117] U. Pal, R. K. Roy, and F. Kimura. Multi-lingual city name recognition for Indian postal automation. In *2012 International Conference on Frontiers in Handwriting Recognition*, pages 169–173, 2012.

[118] J. Pan, C. Canton-Ferrer, K. McGuinness, N. E. O'Connor, J. Torres, E. Sayrol, and X. Giró-i-Nieto. SalGAN: Visual Saliency Prediction with Generative Adversarial Networks. *CoRR*, abs/1701.01081, 2017.

[119] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[120] H. Pham, A. Setlur, S. Dingliwal, T.-H. Lin, B. Poczos, K. Huang, Z. Li, J. Lim, Collin M., and T. Vu. Robust Handwriting Recognition with Limited and Noisy Data. 08 2020.

[121] V. Pham, C. Kermorvant, and J. Louradour. Dropout improves Recurrent Neural Networks for Handwriting Recognition. *CoRR*, abs/1312.4569, 2013.

[122] R. Pierrehumbert. There is no Plan B for dealing with the climate crisis. *Bulletin of the Atomic Scientists*, 75(5):215–221, 2019.

[123] R. Plamondon and S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):63–84, 2000.

[124] M. Popel and O. Bojar. Training tips for the transformer model. *CoRR*, abs/1804.00247, 2018.

[125] D. M. W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*, NeMLaP3/CoNLL '98, page 151–160, USA, 1998. Association for Computational Linguistics.

[126] J. R. Prasad and U. V. Kulkarni. Trends in handwriting recognition. In *2010 3rd International Conference on Emerging Trends in Engineering and Technology*, pages 491–495, 2010.

[127] J. Puigcerver. Are Multidimensional Recurrent Layers Really Necessary for Handwritten Text Recognition? In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 01, pages 67–72, 2017.

[128] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[129] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. Zero-Shot Text-to-Image Generation. *CoRR*, abs/2102.12092, 2021.

[130] V. Romero, J. A. Sánchez, V. Bosch, K. Depuydt, and J. de Does. Influence of text line segmentation in Handwritten Text Recognition. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 536–540, 2015.

[131] K. Roy and K. Majumder. Trilingual Script Separation of Handwritten Postal Document. In *2008 Sixth Indian Conference on Computer Vision, Graphics Image Processing*, pages 693–700, 2008.

[132] K. Roy, U. Pal, and B. B. Chaudhuri. Neural network based word-wise handwritten script identification system for Indian postal automation. In *Proceedings of 2005 International Conference on Intelligent Sensing and Information Processing, 2005.*, pages 240–245, 2005.

[133] J. Salazar, K. Kirchhoff, and Z. Huang. Self-attention Networks for Connectionist Temporal Classification in Speech Recognition. *2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2019.

[134] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[135] A. W. Senior and A. J. Robinson. An Off-Line Cursive Handwriting Recognition System. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(3):309–321, March 1998.

[136] A. Sharma and D. B. Jayagopi. Towards efficient unconstrained handwriting recognition using dilated temporal convolution network. *Expert Systems with Applications*, 164:114004, 2021.

[137] P.Y. Simard, D. Steinkraus, and J.C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 958–963, 2003.

[138] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv:1412.6806*, 2014.

[139] R. Srisha and A. Khan. Morphological Operations for Image Processing : Understanding and its Applications. 12 2013.

[140] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 2014.

[141] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway Networks. *CoRR*, abs/1505.00387, 2015.

[142] S. Srivastava, J. Priyadarshini, S. Gopal, S. Gupta, and H. Dayal. Optical Character Recognition on Bank Cheques Using 2D Convolution Neural Network. In *Applications of Artificial Intelligence Techniques in Engineering*, pages 589–596, Singapore, 2019. Springer Singapore.

[143] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. *CoRR*, abs/1409.3215, 2014.

[144] J. A. Sánchez, V. Romero, A. H. Toselli, and E. Vidal. ICFHR2014 Competition on Handwritten Text Recognition on Transcriptorium Datasets (HTRtS). In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 785–790, 2014.

[145] 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*, 94:122–134, 2019.

[146] C. Tensmeyer, C. Wigington, B. L. Davis, S. Stewart, T. R. Martinez, and W. Barrett. Language Model Supervision for Handwriting Recognition Model Adaptation. *CoRR*, abs/1808.01423, 2018.

[147] S. Vasquez. Handwriting Synthesis with Recurrent Neural Networks. https://github.com/sjvasquez/handwriting-synthesis, 2018.

[148] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need, 2017.

[149] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks, 2018.

[150] A. Vinciarelli and J. Luettin. A new normalization technique for cursive handwritten words. *Pattern Recognition Letters*, 22(9):1043–1050, 07 2001.

[151] P. Wang, L. Yang, H. Li, Y. Deng, C. Shen, and Y. Zhang. A Simple and Robust Convolutional-Attention Network for Irregular Text Recognition. *CoRR*, abs/1904.01375, 2019.

[152] Y. Wang, X. Deng, S. Pu, and Z. Huang. Residual Convolutional CTC Networks for Automatic Speech Recognition. *CoRR*, abs/1702.07793, 2017.

[153] M. Weidemann, J. Michael, T. Grüning, and R. Labahn. D7.8 HTR Engine Based on NNs P2: Building deep architectures with TensorFlow. https://readcoop.eu/wp-content/uploads/2017/12/Del_D7_8.pdf, 2017.

[154] C. Wigington, S. Stewart, B. Davis, B. Barrett, B. Price, and S. Cohen. Data Augmentation for Recognition of Handwritten Words and Lines Using a CNN-LSTM Network. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 01, pages 639–645, 2017.

[155] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C.t Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[156] Y. Xiong, B. Du, and P. Yan. Reinforced transformer for medical image captioning. In *Machine Learning in Medical Imaging*, pages 673–680. Springer International Publishing, 2019.

[157] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *CoRR*, abs/1502.03044, 2015.

[158] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR*, abs/1906.08237, 2019.

[159] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*. Springer Berlin Heidelberg, 2003.

[160] J. Yu, J. Li, Z. Yu, and Q. Huang. Multimodal Transformer With Multi-View Visual Representation for Image Captioning. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(12):4467–4480, 2020.

[161] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[162] A. Zeyer, K. Irie, R. Schlüter, and H. Ney. Improved training of end-to-end attention models for speech recognition. *CoRR*, abs/1805.03294, 2018.

[163] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio, and A. C. Courville. Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks. *CoRR*, abs/1701.02720, 2017.

[164] L. Zhou, Y. Zhou, J. J. Corso, R. Socher, and C. Xiong. End-to-End Dense Video Captioning With Masked Transformer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[165] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai. Deformable DETR: Deformable Transformers for End-to-End Object Detection, 2021.