



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo y experimentación de un sistema de aprendizaje profundo para redes neuronales convolucionales y recurrentes.

Development and experimentation of a deep learning system for convolutional and recurrent neural networks

Degree final work

Degree in Informatics Engineering

Author: Mocholí Calvo, Carlos

Tutor: Vidal Ruiz, Enrique

Experimental director: Puigcerver i Pérez, Joan

Course 2017/2018

This work would not have been possible without the opportunity to be part of the PRHLT research center.

I am especially indebted to Enrique Vidal, who offered this possibility to me, and to Joan Puigcerver, who was a teacher and mentor throughout the making of this work.

I would also like to express my gratitude to my parents, who always encouraged me to keep going over any problems that I might have faced as well as to my close friends, who are always there for me even when I prioritized other things over them.

Thank you.

Resum

En l'actualitat, hi ha molt pocs toolkits d'aprenentatge profund centrat en la tasca de Reconeixement de Text Manuscrit (HTR). HTR es referix al problema de reconèixer una seqüència de caràcters en una imatge d'entrada. Per aquest motiu, hem decidit crear PyLaia, un toolkit per a realitzar experiments d'anàlisi de documents manuscrits.

PyLaia és flexible, de codi obert, independent del dispositiu en què s'executa i es pot utilitzar per a expressar una àmplia varietat d'experiments, inclòs l'entrenament i la inferència sobre models de xarxes neuronals profundes convolucionals i recurrents. S'ha utilitzat per a realitzar investigacions sobre els conjunts de dades IAM i RIMES. PyLaia també és un successor de Laia, escrit en Lua.

Este treball descriu la implementació del sistema que hem construït utilitzant PyTorch com a base per al nostre toolkit. El programari és extensible i fàcilment configurable i proporciona un ampli conjunt de capes funcionals amb un enfocament particular en HTR. A més, també descrivim la implementació de l'arquitectura del nostre model personalitzat que combina capes convolucionals i recurrents per a competir amb les arquitectures actuals d'avantguarda en el camp.

Una àmplia gamma d'experiments s'han dut a terme per a validar la nostra implementació. Els experiments presentats aconseguixen millors resultats que els obtinguts amb el predecessor del toolkit. També comparem l'impacte de diverses característiques, com l'ús del model que permet l'entrada d'imatges d'altura variable o l'ús del dropout.

PyLaia es manté com un paquet de codi obert sota la llicència de MIT i està disponible en <https://github.com/jpuigcerver/PyLaia>

Paraules clau: aprenentatge automàtic, xarxes neuronals, xarxes neuronals convolucionals, xarxes neuronals recurrents, reconeixement de text manuscrit, PyTorch

Resumen

En la actualidad, hay muy pocos toolkits de aprendizaje profundo centrado en la tarea de Reconocimiento de Texto Manuscrito (HTR). HTR se refiere al problema de reconocer una secuencia de caracteres en una imagen de entrada. Por este motivo, hemos decidido crear PyLaia, un conjunto de herramientas para realizar experimentos de análisis de documentos de texto manuscrito.

PyLaia es flexible, de código abierto, independiente del dispositivo en el que se ejecuta y se puede utilizar para expresar una amplia variedad de experimentos, incluido el entrenamiento y la inferencia sobre modelos de redes neuronales profundas convolucionales y recurrentes. Se ha utilizado para realizar investigaciones sobre los conjuntos de datos IAM y RIMES. PyLaia también es un sucesor de Laia, escrito en Lua.

Este trabajo describe la implementación del sistema que hemos construido utilizando PyTorch como base para nuestro toolkit. El software es extensible y fácilmente configurable y proporciona un amplio conjunto de capas funcionales con un enfoque particular en HTR. Además, también describimos la implementación de la

arquitectura de nuestro modelo personalizado que combina capas convolucionales y recurrentes para competir con las arquitecturas actuales de vanguardia en el campo.

Una amplia gama de experimentos se han llevado a cabo para validar nuestra implementación. Los experimentos presentados logran mejores resultados que los obtenidos con el predecesor del toolkit. También comparamos el impacto de varias características, como el uso del modelo que permite la entrada de imágenes de altura variable o el uso del dropout.

PyLaia se mantiene como un paquete de código abierto bajo la licencia de MIT y está disponible en <https://github.com/jpuigcerver/PyLaia>

Palabras clave: aprendizaje automático, redes neuronales, redes neuronales convolucionales, redes neuronales recurrentes, reconocimiento de texto manuscrito, PyTorch

Abstract

At present, there are very few deep learning toolkits focused on the task of Handwritten Text Recognition (HTR). HTR refers to the problem of recognizing a sequence of characters in an input image. For this reason, we have decided to build PyLaia, a toolkit for performing handwritten text document analysis experiments.

PyLaia is flexible, open-source, device-agnostic, and can be used to express a wide variety of experiments, including training and inference over Convolutional and Recurrent based deep Neural Network models. It has been used for conducting research over the IAM and RIMES datasets. PyLaia is also a successor to Laia, written in Lua.

This work describes the implementation of the system that we have built using PyTorch as the basis for our toolkit. The software is extensible and easily configurable and provides a rich set of functional layers with a particular focus on HTR. Additionally, we also describe the implementation of our custom model architecture which combines convolutional and recurrent layers to compete with current state-of-the-art architectures in the field.

A wide array of experiments have been carried out to validate our implementation. The experiments presented achieve improved results over those done using the toolkit's predecessor. We also compare the impact of several features such as the usage of a model which allows inputs of variable height or the use of dropout.

PyLaia is maintained as an open-source package under the MIT license and is available at <https://github.com/jpuigcerver/PyLaia>

Key words: machine learning, neural networks, convolutional neural network, recurrent neural network, handwritten text recognition, PyTorch

Contents

Contents	vii
List of Figures	ix
List of Tables	x
<hr/>	
1 Introduction	1
1.1 Background	1
1.2 Motivation	1
1.3 Objectives	2
1.4 Thesis structure	2
1.5 Contributions	3
2 Pattern Recognition	5
2.1 Neural Network (NN)	5
2.2 Convolutional Neural Network (CNN)	6
2.3 Recurrent Neural Network (RNN)	7
2.4 The multidimensional case	8
2.5 The Long Short Term Memory (LSTM) block	8
2.6 The Connectionist Temporal Classification (CTC) loss function	9
3 Handwritten Text Recognition	11
3.1 History	11
3.2 Data	12
3.2.1 The IAM dataset	12
3.2.2 The RIMES dataset	13
4 State-of-the-art	15
4.1 Critique	17
5 Proposal	19
5.1 Chosen toolkit backend	19
5.2 Chosen technologies	20
5.3 System architecture	20
5.4 Detailed design	23
6 Solution development	25
6.1 Model architecture	25
6.2 Training	27
6.3 Experiments	28
6.3.1 Dropout comparison	28
6.3.2 Sweep over learning rate (η)	30
6.3.3 Input height comparison	32
6.3.4 PyLaia compared to Laia	35
7 Conclusion and future work	39
8 Degree relationship	41

Bibliography

45

List of Figures

1.1	Commit graph for the Torch and PyTorch GitHub repository respectively as of the date of this writing. Notice the y axis scale difference.	2
2.1	Mathematical model of artificial neuron. Figure adapted from SPINN: a straightforward machine learning solution to the pulsar candidate selection problem	6
2.2	3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Source: Stanford’s Convolutional Neural Networks for Visual Recognition course	6
2.3	Example of a 2D convolution operation. The kernel size is 2×2 and the stride is 2	7
2.4	In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). Source: Stanford’s Convolutional Neural Networks for Visual Recognition course	7
2.5	A recurrent neural network and the unfolding in time of the computation involved in its forward computation. The artificial neurons get inputs from other neurons at previous time steps. Adapted from LeCun, Bengio, and G. Hinton 2015	8
2.6	Instead of having a single neural network layer, there are four different gates interacting between each other (represented in yellow). Source: Christopher Olah	9
2.7	Example of how the CTC function might collapse an input image. The image represents the input of the network and the box represents the CTC output. This is later decoded by removing duplicates and ϵ , a CTC specific symbol. Figure adapted from Hannun, “Sequence Modeling with CTC”, Distill, 2017.	10
3.1	A sample dataset image	12
3.2	A sample dataset image	13
4.1	General model architecture used in most of the SOTA results. Figure taken from Théodore Bluche’s paper “Joint line segmentation and transcription for end-to-end handwritten paragraph recognition” [21].	16
5.1	PyTorch’s logo. Image taken from [22]	20
6.1	Architecture of the neural network presented in this work. Figure adapted from [3]	25
6.3	Comparison of the loss function between Laia and PyLaia. N is the number of samples in the batch, T is the number of frames in each sample of the batch	30

6.5	Example of how deformation may occur. Imagine the case where figures 6.5a and 6.5c are part of our dataset, take into account that their character size is exactly the same but 6.5c's bounding box was poorly chosen. If we were to resize the height of all images to that of figure 6.5c we would end up with a very stretched character in the case of 6.5a. On the other hand, if we did the opposite resizing to the height of 6.5a the opposite effect would happen	32
6.8	Combination of two 2D tensors into a single 3D tensor. The images whose width is less than the widest image in the batch are centered and padded with zeros	37

List of Tables

4.1	Comparison of the character and word error rate (%) on IAM and RIMES paragraphs of previously published competitive state-of-the-art results.	16
6.1	Details of the configuration used in the convolutional blocks of our architecture.	26
6.2	Dropout comparison of the character and word error rate on IAM and RIMES paragraphs for the test partition.	28
6.3	Comparison between the toolkits' processed samples per second in both the training and decoding steps using the IAM dataset.	35

CHAPTER 1

Introduction

1.1 Background

Due to the recent success of deep learning, a large number of Deep Learning focused software has emerged. Even though they are open source software and allow external contributions, they are mostly maintained by employees of multinational technology companies such as Google and Facebook. Their purpose is to provide a general purpose computing framework that fulfills most of the users' necessities. However, for a field-focused research group, having an in-house system tailored to the groups' needs is a very valuable asset to ensure code re-utilization and to improve the speed of research and ease of collaboration within the group members.

This is the case for the Pattern Recognition and Human Language Technology (PRHLT) research center. In 2016, three of its members released Laia, an open source deep learning toolkit to transcribe handwritten text images [1]. This toolkit, which was built using Torch, aimed to facilitate common HTR experiments and has been extensively used through the research center's projects.

1.2 Motivation

Torch is a versatile numeric computing framework and machine learning library that extends the Lua scripting language [2]. It supports both CPU and GPU devices. However, in October 2016, Facebook Artificial Intelligence Research (FAIR) released PyTorch, a successor to Torch, thus most of the experiments done on Torch have been migrated to PyTorch. Additionally, Torch's development has stopped completely as seen in Figure 1.1

The key differences are the improved bridge to the core implementation written in performing languages such as C and CUDA as well as the possibility to use the vast ecosystem of Python scientific libraries.

Based on the experience with Laia and a more complete understanding of the desirable system properties and requirements for HTR experiments, we have decided to build **PyLaia** as a second-generation system.

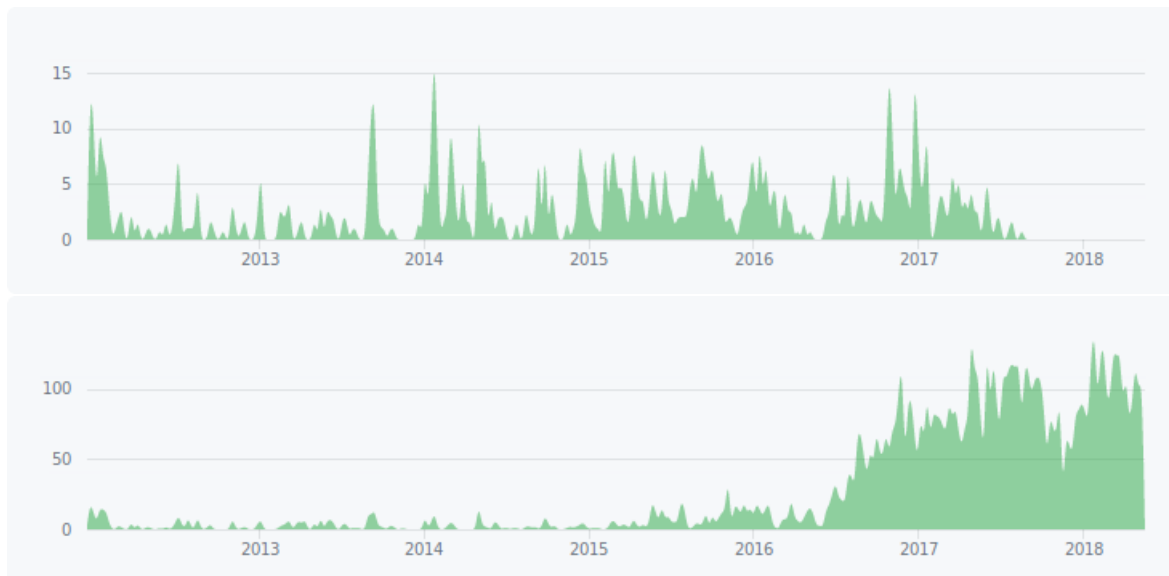


Figure 1.1: Commit graph for the Torch and PyTorch GitHub repository respectively as of the date of this writing. Notice the y axis scale difference.

1.3 Objectives

The aim of this work is to further explore the task of statistical modeling of handwritten text and to convert handwritten text into the digital format by writing a general purpose HTR toolkit.

The toolkit is supposed to be built in a modular way. Allowing the user to perform the same tasks that are available in Laia. Even though the focus of this work is handwriting recognition, the system must not be tied to this task, allowing similar handwriting document analysis tasks such as Keyword Spotting or Line Segmentation to be performed reusing at least part of the codebase.

Handwritten text is a very general term so to validate the results in this project we have decided to validate our experiments against those done previously for Laia with the IAM and RIMES datasets [3].

1.4 Thesis structure

In Chapter 2 of this document we first review the basics of Pattern Recognition by introducing the basic model architectures used throughout this work.

In Chapter 3, we introduce the field of HTR to the less familiar readers and then describe the datasets used in this work.

Chapter 4 discusses current state-of-the-art results in the field of HTR.

Chapter 5 lays out the technological alternatives for this work and briefly mentions the system architecture.

Chapter 6 contains the results obtained for each experiment performed.

In Chapter 7, we present the conclusions drawn from the obtained results and what additional research is necessary is to further the goals of this work.

Finally, in Chapter 8, we mention how the skills learned through the degree were applicable to this work's development.

Additionally, a list of the acronyms used in this work is present before the bibliography.

1.5 Contributions

The project's first commit is dated as of 30th of October, 2017. The reasons why it was created to replace Laia are those aforementioned in Section 1.2

I was not the sole developer working in PyLaia. Joan Puigcerver, experimental director of this work and creator of both Laia and PyLaia, had already written 2192 lines of code¹ by the time I entered the project, around the 20th of February.

The foundations and main structure of the software were already laid out so my main (but not exclusive) focus was to write all of the code necessary to perform robust and reproducible experiments necessary for this work. These relate mostly to the packages `common`, `conditions`, `engine`, `experiments`, `hooks`, and `meters`, as described in Section 5.3. By the time of this writing, we have reached 10170 lines of code.

¹Counting only the number of lines of the core library source code files

CHAPTER 2

Pattern Recognition

Pattern recognition is the science of making inferences from perceptual data based on either a priori knowledge or on statistical information. Neural networks have become a key component in modern pattern recognition systems.

In a typical pattern recognition application, the raw data is processed and converted into a form that is amenable for a machine to use, usually features taken together into vectors. Features may be represented as continuous or discrete variables. A feature can be obtained from one or more measurements that quantify some significant characteristics of the object.

There are several neural network architectures which can be used to accomplish different tasks with different techniques. I will focus on those designed for classification tasks.

2.1 Neural Network (NN)

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems but has since diverged and become a matter of engineering and achieving good results in Machine Learning tasks.

The network is based on neurons, also called units. Each neuron receives input signals (which get all summed) and produces output signals after altering its internal state. The idea is that the connection strengths (the weights \mathbf{W}) are learnable and control the strength of influence of one neuron on another. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. We model the firing rate of the neuron with an activation function σ , which represents the frequency of the spikes along the axon. In practice, this activation function must be differentiable (continuous) to be able to apply optimization algorithms to it.

The simplest type of Neural Networks are Feedforward Neural Networks, where the connections between the neurons do not form a cycle. The signal chain is transmitted from the inputs through the hidden layers all the way to the output layer. This layer consists of the set of nodes that we measure and whose output is not connected to any other neurons.

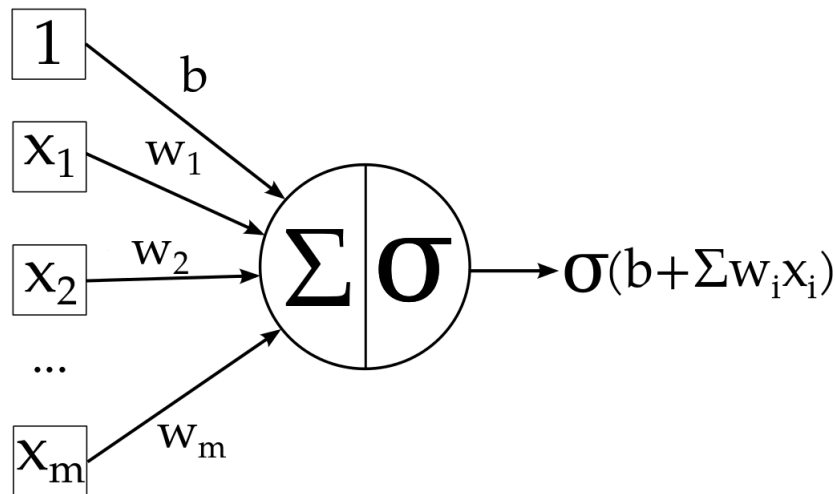


Figure 2.1: Mathematical model of artificial neuron. Figure adapted from SPINN: a straightforward machine learning solution to the pulsar candidate selection problem

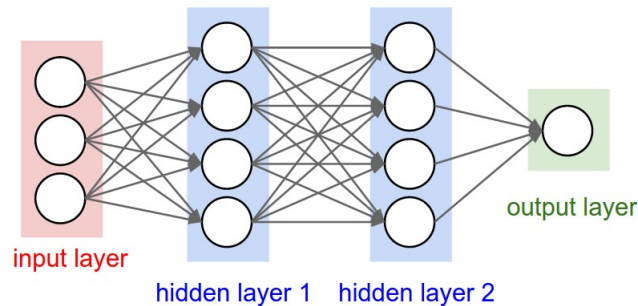


Figure 2.2: 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Source: Stanford's Convolutional Neural Networks for Visual Recognition course

2.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks are very similar to ordinary Feedforward Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw signal elements on one end to class scores at the other. And they still have a loss function on the last (fully-connected) layer. One essential property of CNNs is that the layers are exclusively connected to their immediate neighbours.

The key is that parameters are reused independently of their position in the image, meaning that a neuron may activate with the same magnitude when a certain feature is recognized even in different positions. This helps with generalization thus avoiding overfitting. Moreover, this architecture makes an explicit assumption about the input dimensions, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$* \begin{matrix} \begin{matrix} 0.5 & 0.1 \\ -0.1 & 0.8 \end{matrix} \\ = \begin{matrix} \begin{matrix} 5 & 7.6 \\ 15.4 & 18 \end{matrix} \end{matrix}$$

$$[\mathbf{x} * \mathbf{W}]_{i,j} = \sum_l \sum_m \mathbf{x}_{i-j, j-m} \mathbf{W}_{l-m}$$

Figure 2.3: Example of a 2D convolution operation. The kernel size is 2×2 and the stride is 2

A CNN arranges its neurons in three dimensions (width, height, depth). Every layer transforms the 3D input volume to a 3D output volume of neuron activations.

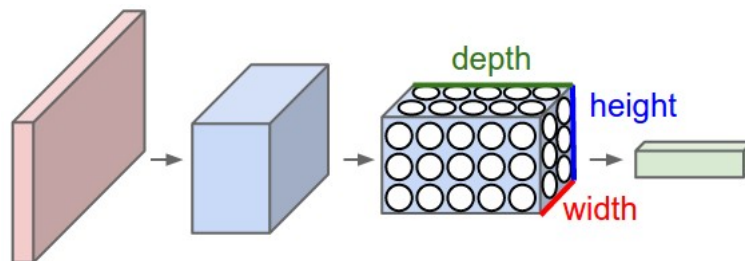


Figure 2.4: In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). Source: Stanford's Convolutional Neural Networks for Visual Recognition course

2.3 Recurrent Neural Network (RNN)

While Feedforward Neural Networks are able to process individual vectors of a fixed size (the input from the previous layer), this is not the case for RNNs which can cope with unlimited sequences of such vectors. RNNs can in principle make use of an arbitrary amount of context by storing information in their internal state. This, in addition to the ability to process sequences of variable length, makes them ideal for processing sequences of values as is required in HTR.

The key idea is parameter sharing (the weights \mathbf{W}) across different time steps. Parameter sharing makes it possible to extend and apply the model to examples of different shapes while generalizing across them. Thus a RNN can be considered as a layered, Feedforward Neural Network with shared weights

The recurrence formula at every time step is defined by:

$$s_t = \sigma(\mathbf{U}x_t, \mathbf{W}s_{t-1})$$

where s_t is the new state, σ is an activation function, s_{t-1} is the state at the previous step, x_t is the input vector at the t -th time step and \mathbf{W} , \mathbf{U} are weight matrices.

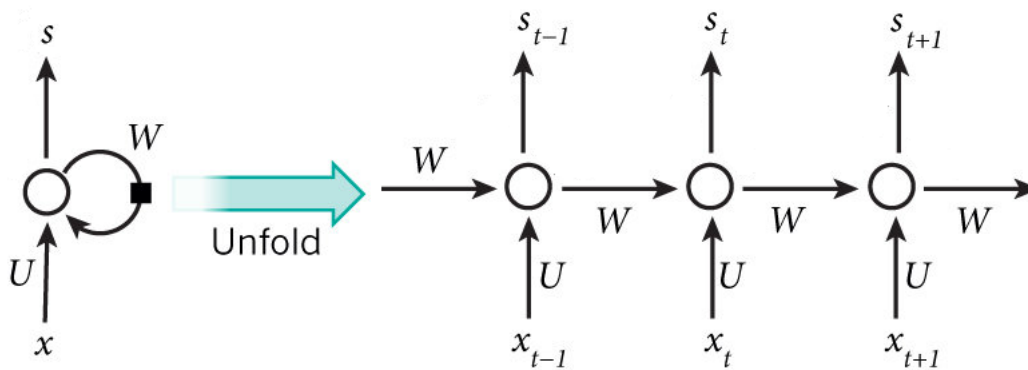


Figure 2.5: A recurrent neural network and the unfolding in time of the computation involved in its forward computation. The artificial neurons get inputs from other neurons at previous time steps. Adapted from LeCun, Bengio, and G. Hinton 2015

Although their main purpose is to learn long-term dependencies, theoretical and empirical evidence shows that it is difficult to learn to store information for very long [4].

2.4 The multidimensional case

A Multidimensional Recurrent Neural Network is a generalization of a RNN, which can deal with higher-dimensional data. These can use a recurrence over either one or two dimensions in order to model the variations on both axes [5] and potentially capturing long-term dependencies across both axes.

Restricting ourselves to the 2D case, commonly used for HTR, we can describe how does it process an input. A 2D RNN scans the input image along both axes and produces a transformed output of the same size. The hidden state $s_{u,v}$ for a position (u, v) of a 2D layer is computed based on the previous hidden states of both axes, $s_{u-1,v}$ and $s_{u,v-1}$, and the current input $x_{u,v}$ by:

$$s_{u,v} = \sigma(\mathbf{U}x_{u,v} + \mathbf{W}s_{u-1,v} + \mathbf{V}s_{u,v-1} + b)$$

where \mathbf{U} , \mathbf{W} and \mathbf{V} are weight matrices, b a bias vector and σ a nonlinear activation function

In the HTR field, these networks are used to transcribe images of text lines, where multiple multidimensional layers are stacked in combination with other types of layers.

2.5 The Long Short Term Memory (LSTM) block

Long Short Term Memory Recurrent Neural Network (LSTM-RNN) have been very successful both in general machine learning tasks and in the HTR community. The use of LSTM blocks in a RNN allows the network to store information for longer amounts of time by exploiting more context and leads to more stable training by

avoiding the vanishing/exploding gradient problem [6], where the influence of a given input on the hidden layer either decays or blows up exponentially as it cycles around the network’s recurrent connections.

These use special hidden layers which consist of recurrently connected subnets, called memory blocks, the natural behaviour of which is to remember inputs for a long time. A special unit called the memory cell acts like an accumulator or a gated leaky neuron: it has a connection to itself at the next time step that has a weight of one, so it copies its own real-valued state and accumulates the external signal, but this self-connection is multiplicatively gated by another unit that learns to decide when to clear the content of the memory.

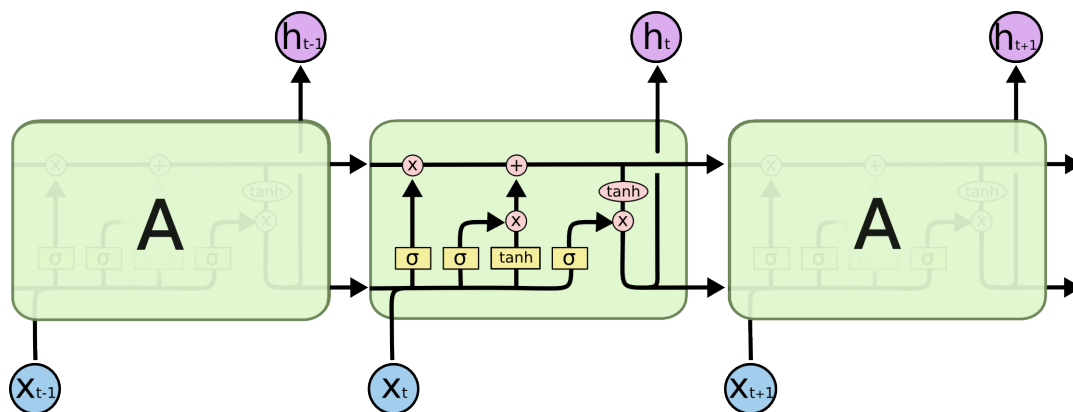


Figure 2.6: Instead of having a single neural network layer, there are four different gates interacting between each other (represented in yellow). Source: Cristopher Olah

For many tasks, it is useful to have access to “future” as well as “past” context¹. The identification of a given letter is helped by knowing the letters beside itself. Bidirectional RNNs (and LSTMs) are able to access context in both directions along the input sequence. They work by using two separate hidden layers, each one processing the input in each direction. Both of these layers are connected to the same output layer, providing it with access to the past and future context of every point in the sequence.

2.6 The Connectionist Temporal Classification (CTC) loss function

Unfortunately, in HTR, we don’t know how the characters in the transcript align to the image. This makes training an image transcriber harder than it might at first seem. We could devise a rule like “one character corresponds to ten inputs” but people’s rates of handwriting vary. Another alternative is to hand-align each character to its location in the image thus we would know the ground truth for each input time-step. However, for any reasonably sized dataset, this is prohibitively time-consuming. Also, in the case of very cursive handwriting, the input segmentation is very difficult to determine.

¹Processing an image from right-to-left and left-to-right in the case of HTR

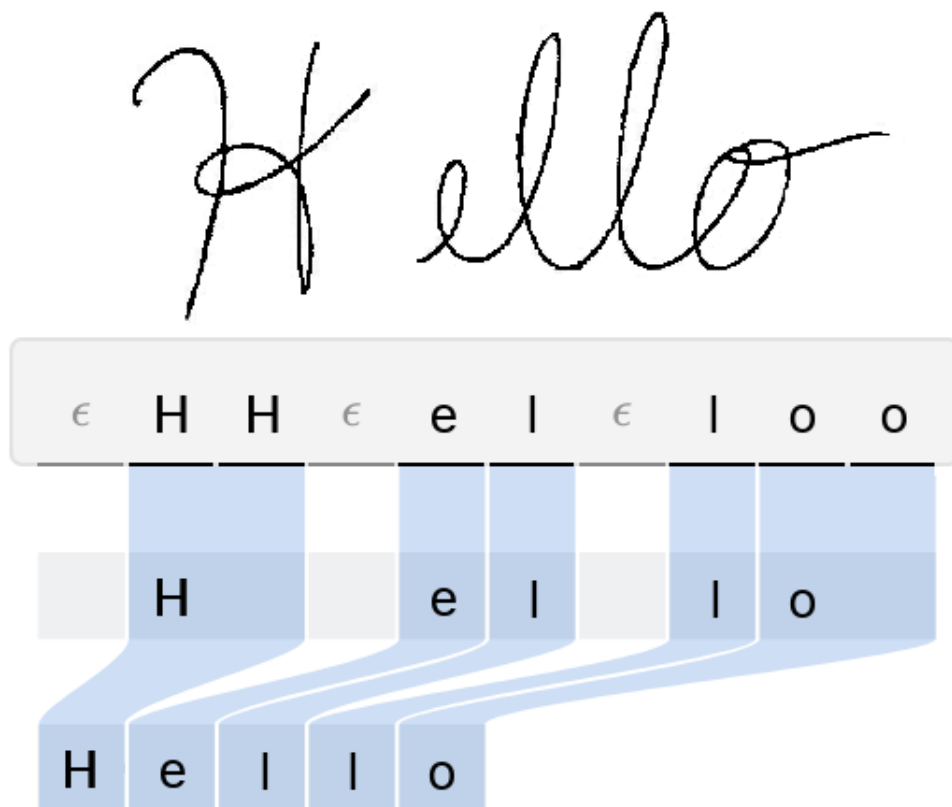


Figure 2.7: Example of how the CTC function might collapse an input image. The image represents the input of the network and the box represents the CTC output. This is later decoded by removing duplicates and ϵ , a CTC specific symbol. Figure adapted from Hannun, “Sequence Modeling with CTC”, Distill, 2017.

CTC is a way to get around not knowing the alignment between the input and the output. Consider mapping input sequences $\mathbf{x} = [x_1, x_2, \dots, x_n]$, to corresponding output sequences $\mathbf{y} = [y_1, y_2, \dots, y_m]$. We want to find an accurate mapping between these given the following limitations:

- Both \mathbf{x} and \mathbf{y} can vary in length.
- The ratio of the lengths of \mathbf{x} and \mathbf{y} can vary.
- We don’t have an accurate alignment (correspondence of the elements of \mathbf{x} and \mathbf{y}).

The CTC algorithm overcomes these challenges. For a given sequence \mathbf{x} it outputs a distribution over all possible labels \mathbf{y} . We can use this distribution either to infer a likely output or to assess the probability of a given output and thus calculate a loss. To get the probability of an output given an input, CTC works by summing over the probability of all possible alignments between the two, this is possible because it assumes that every output alignment is conditionally independent on each other, which might sometimes be a bad assumption [7]. Most state-of-the-art HTR systems use CTC as the loss function with which the model is trained [8].

CHAPTER 3

Handwritten Text Recognition

For centuries, the most common way of preserving and disseminating ideas and facts through posterity was to write it by hand. Nonetheless, with the advent of digital storage, it has been replaced by technological writing tools. This has the convenience of allowing the indexing of the documents so searches and analysis can be quickly performed in addition to the ability to instantly create duplicates and to share them with others.

Handwritten Text Recognition (HTR) is an area of pattern recognition which defines an ability of a machine to analyze patterns and recover the handwritten text represented in an input signal. Particularly, In the case of offline¹ handwriting recognition, the input signal is typically a variable-sized two dimensional image in the form of a segmented line of a scanned document (e.g. forms, historical manuscripts, etc), and the output is a sequence of characters.

This problem can be addressed statistically by solving the following optimization problem:

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \Pr(\mathbf{y} | \mathbf{x})$$

Where \mathbf{x} is the input signal, representaed as a sequence of frames, and \mathbf{y} is a sequence of symbols. Since the real probability distribution $\Pr(\mathbf{y} | \mathbf{x})$ is unknown, it is modeled by a parametric distribution $P_{\Theta}(\mathbf{y} | \mathbf{x})$ whose parameters are fit according to the Maximum Likelihood Estimation criterion.

It is important to keep in mind that handwriting recognition research will not be finished once most of the historic documents have been digitalized and published in online digital libraries since some data types are much more comfortable to input by a pen. Examples of these are mathematical equations, music composition, etc. These are the main reasons why HTR has proven to be an increasingly important task.

3.1 History

The first occurrences of handwritten text classification were done with Optical Character Recognition (OCR) machines, these were primitive mechanical devices with

¹As opposed to online recognition, where the input is a time series of coordinates, representing the movement of the pen-tip.

fairly high failure rates. After some false starts, OCR became a competitive commercial enterprise in the 1950's [9].

The next major upgrade in producing high OCR accuracies was the use of a Hidden Markov Model for the task of OCR. This approach uses letters as a state, which then allows for the context of the character to be accounted for when determining the next hidden variable [10].

The cursive nature of handwriting makes it hard to first segment the text into characters to recognize them individually. This method has been progressively replaced by the sliding window approach, in which features are extracted from vertical frames [11]. For instance, a CNN can be applied to identify text by moving the sliding window across the image to find a potential instance of a character being present.

Finally, the most recent advances in deep learning and the new architectures have allowed building models that can handle both the 2D aspect of the input and the sequential aspect of the prediction. In particular, Long Short Term Memory based Recurrent Neural Networks, using the Connectionist Temporal Classification objective function [12] have yielded low error rates and became the state-of-the-art model for HTR [13]

3.2 Data

In this work, we used two popular resources to validate our work. They are both quite large datasets, which is a very important characteristic for deep learning tasks.

3.2.1. The IAM dataset

Compiled by the FKI-IAM Research group², this dataset consists of handwritten (modern) English sentences based on the Lancaster-Oslo/Bergen (LOB) corpus. The corpus is a collection of texts that comprise about one million word instances.

The dataset includes 1 066 forms produced by approximately 400 different writers. A total of 82 227 word instances out of a vocabulary of 10 841 words occur in the collection. The database consists of full English sentences [14]. This database, given its breadth, depth, and quality tends to serve as the basis for many handwriting recognition tasks.

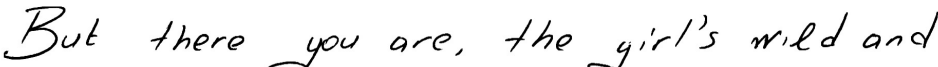


Figure 3.1: A sample dataset image

It is split into writer-independent training, validation and test partitions of 6 161, 966 and 2 915 lines, respectively. The original lines images in the training set have an average width of 1 751 pixels and an average height of 124 pixels. There are 79 different characters in the dataset including the white-space.

²<https://www.fki.inf.unibe.ch/databases/iam-handwriting-database>

3.2.2. The RIMES dataset

Compiled by A2iA³, an HTR focused business, this dataset consists of handwritten (modern) French sentences [15].

The database was collected by asking volunteers to write handwritten letters. They were given a fictional identity (same sex as the real one) and up to 5 scenarios regarding business interaction themes such as change of personal information, modification of contract or complaint among others. The volunteers composed a letter with those pieces of information using their own words. The layout was free and it was only asked to use white paper and to write in a readable way with black ink.

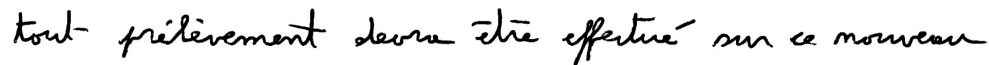


Figure 3.2: A sample dataset image

The dataset split consists of 11 333 training lines and 778 test lines. Since the original release does not include a separated validation partition, 10% of the total training lines were sampled for validation purposes. Thus, the final division of the dataset into training, validation, and test consists of 10 171, 1 162 and 778 lines, respectively. The original line images in the training set have an average width of 1 658 pixels and an average height of 113 pixels. There are 99 different characters in the dataset including the white-space.

³<https://www.a2ia.com>

CHAPTER 4

State-of-the-art

The metrics used to evaluate and compare any system are critical. In particular, for HTR, two different metrics are used:

- Character Error Rate (CER). Computed with the minimum number of operations required to transform the reference text into the hypothesis generated (a number which is known as the Levenshtein distance [16]). It is defined by the formula:

$$\text{CER} = (i_c + s_c + d_c) / n_c$$

where n_c is the number of characters in the reference, s_c is the number of substitutions, d_c the number of deletions and i_c the number of insertions required to transform the output hypothesis into the reference.

- Word Error Rate (WER). This is defined in a similar way, but using words as the Levenshtein unit instead of characters.

$$\text{WER} = (i_w + s_w + d_w) / n_w$$

Note that the number of errors can be larger than the length of the reference and lead to percentages larger than the unit.

Current state-of-the-art results for the datasets used in this work are shown in Table 4.1. These have been achieved using a language model (unlike the results in this work), the number of epochs has not been disclosed by the authors.

Most of these¹ use a MDLSTM-RNN, with bidirectional LSTMs. These may be optionally followed by convolutions as is the case for Pham et al. [20] and Voigtlaender et al. [18] All of them use the CTC loss function to calculate the backward gradients.

Most of these authors have not disclosed the results obtained without the language model applied after training the network. However, assuming that NN training is independent to the language model training we can conclude that our model would benefit equally if a language model was applied to it. The application of the language model is equal to the application of a monotonically increasing function

¹RIMES does not have a predefined validation set, so the validation sets are likely to be different.

¹Excluding Puigcerver's results, whose LSTM layer is one dimensional

Table 4.1: Comparison of the character and word error rate (%) on IAM and RIMES paragraphs of previously published competitive state-of-the-art results.

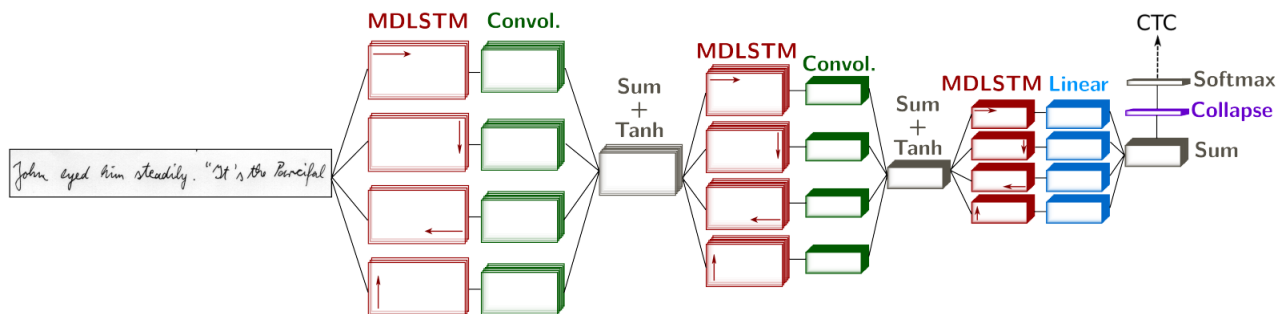
IAM Dataset System	CER (%)		WER (%)	
	Validation	Test	Validation	Test
Puigcerver. [3]	2.9	4.4	9.2	12.2
Bluche. [17]	—	4.4	—	10.9
Voigtlaender et al. [18]	2.4	3.5	7.1	9.3
Doetsch et al. [19]	2.5	4.7	8.4	12.2
Pham et al. [20]	3.7	5.1	11.2	13.6
RIMES Dataset System	CER (%)		WER (%)	
	Validation ¹	Test	Validation ¹	Test
Puigcerver. [3]	2.3	2.5	8.9	9.0
Bluche. [17]	—	3.5	—	11.2
Voigtlaender et al. [18]	—	2.8	—	9.6
Doetsch et al. [19]	—	4.3	—	12.9
Pham et al. [20]	3.3	3.3	13.1	12.3

thus if it were to be applied to an inequality, this inequality would uphold. This can be expressed as:

$$\forall N_A, N_B : e_{N_A} > e_{N_B} \implies e_{L(N_A)} > e_{L(N_B)}$$

where N_A and N_B are two neural networks trained over the same dataset, e is the error obtained and L is the same language model for both cases.

Since a language model is not going to be used after training our network. We can not directly compare our results to the results displayed in Table 4.1. Luckily, we can compare our results with those performed by Puigcerver using Laia. With the previously mentioned property, if our results are similar to his, we then know that our results compete with current state-of-the-art results if a language model was applied to them.

**Figure 4.1:** General model architecture used in most of the SOTA results. Figure taken from Théodore Bluche’s paper “Joint line segmentation and transcription for end-to-end handwritten paragraph recognition” [21].

4.1 Critique

Multidimensional LSTM networks are quite computationally expensive, especially compared to convolutions. This is due to the fact that the convolutions computational cost is one or two orders of magnitude smaller. Even if we had an infinite number of processing units, a clever implementation of a 2D LSTM would have a computational complexity of $\mathcal{O}((W + H) \cdot D + C)$ whereas in the case of a 2D convolution its $\mathcal{O}(C \cdot S)$ where W and H are the width and height of the image, D and C the number of input and output channels and S the kernel size of the convolution.

Human language is also naturally one-dimensional, it is composed by sequences which can be read from either left-to-right or right-to-left. This suggests a lack of need to learn long-term dependencies in the extra dimension that a MDLSTM would provide.

Since the first layers of a Neural Network have to process a larger amount of inputs, we can greatly improve our model efficiency by stacking a few convolutional layers at the start of our net, especially knowing that they have been shown to extract similar features to those extracted by a LSTM at these initial layers.

As Puigcerver laid out in his paper [3], the combination of a CNN and a LSTM-RNN can achieve the same results whilst drastically reducing the memory consumption and inference runtime.

CHAPTER 5

Proposal

PyLaia is meant to be a platform for research on deep learning methods in handwritten document analysis which aims to significantly lower the barrier to high-quality research. It is also designed to support researchers who want to build experiment pipelines as quickly and easily as possible.

Most existing deep learning toolkits are designed for general machine learning, and a significant effort can be required to develop research infrastructure for particular model classes. PyLaia provides custom operators to fill the gaps that are required for HTR research.

5.1 Chosen toolkit backend

There are multiple competing toolkits for building deep learning models. These explore different tradeoffs between usability and expressiveness, research or production focus and supported hardware. All of them operate on a Directed Acyclic Graph (DAG) of computational operators, wrapping high-performance libraries such as CUDNN (for NVIDIA GPUs) and automate memory allocation, synchronization, and distribution.

The two largest toolkits as of the time of this writing are Tensorflow and PyTorch. Both of them represent a computation in terms of a graph, where data flows between nodes and dependencies are defined between individual operations. The nodes in the graph represent variables (tensors, scalars, etc.) and the edges represent some mathematical operations. During the optimization step, the chain rule and the graph are combined to compute the derivative of the output with respect to the learnable nodes in the graph.

PyTorch makes use of *dynamic* computation graphs meaning that they are built and executed at runtime. It provides a flexible data management API that handles intelligent batching and padding, high-level abstractions for common operations and a modular and extensible experiment framework. It is imperative, meaning that the computation is run immediately, this allows the use of debugging tools. Being dynamic, it is very easy to modify and execute different nodes as the computation is done.

Several neural network architectures can benefit from this, for example, in RNN based models where the input sequence length is variable. This means that the

dataset samples do not have to be resized to some fixed length (either by cutting or padding)

Other declarative frameworks such as TensorFlow use a *static* computation graph. This is given and run by an execution engine provided by the framework. This approach gives easier deployment, potentially improved efficiency and the ability to do compilation ahead of time.

5.2 Chosen technologies

We chose to build PyLaia on PyTorch. This allows swift model development by reusing PyTorch's heavily tested modules and adds functionality for data management and experimentation on common HTR problems.



Figure 5.1: PyTorch's logo. Image taken from [22]

The main reasons why we chose it are the aforementioned dynamic nature, necessary for HTR where all of the input sequences lengths may vary and the fact that since Laia was written in Torch we could keep the overall design and structure of the software, principally because of the API similarity between Torch and PyTorch.

Moreover, when we started writing PyLaia¹, some benchmarks were performed comparing the memory usage and speed of the different LSTM implementations in both toolkits. The results showed that TensorFlow consumed around double the memory yet still was slower. The reason for this is that their implementation supporting variable length input sequences is not as optimized as their basic LSTM implementation, written on top of CUDA. Considering that the ability to process variable length inputs was one of the main motivations of the rewrite², PyTorch came as a clear winner in this front. However, keep in mind that this might have changed significantly due to both software's active release cycle.

We also use existing implementations of several operations to reduce the cost of implementation and risk of bugs. Most notably we use Baidu's CTC loss function implementation, which is compatible with both the CPU and GPU.

5.3 System architecture

PyLaia's core is structured into separate packages designed to be as minimalist and modular as possible, aiming to follow the UNIX philosophy:

¹Around October 2017

²As we describe in Section 6.3.3

- **common**. Contains all of the general purpose classes that might be used anywhere in the codebase such as `logging`, `arguments` parsing utilities and the state `saver` and `loaders`.
- **conditions**. Contains several classes that act as preconditions to fire an action. See `hooks`.
- **data**. Contains any classes whose purpose is to feed data into the experiment pipeline. In addition, it also contains any classes whose purpose is to apply a transformation to an input.
- **decoders**. As the name implies, this contains the different decoders to transform an input feature representation into an output. As of the time of this writing, there are only CTC related decoders, for example, the one represented in Figure 2.7.
- **engine**. This contains the `Engine` and `Trainer`. They perform the model inference by running the forward and backward steps and notifying the system through the `hooks`.
- **experiments**. This package contains the different high-level classes to run any kind of experiment. Their responsibility is to keep track of whatever metrics are appropriate by taking control of their internal model and trainer. They also take care of running the validation step during the training.
- **hooks**. Hooks are a critical part of the training phase in an experiment. They are designed to activate at some pre-defined steps during the training³. They might make use of a condition in order to perform the action that they have been assigned. This way a PyLaia user can run their own code during the training without modifying the source code or rewriting it.
- **losses**. Contains any loss functions. Mostly wrappers to PyTorch's loss functions and Baidu's CTC loss implementation.
- **meters**. Contains the different classes that compute any kind of metric to be used during the experiment. Some notable examples are the `SequenceErrorMeter` used to compute the model's CER and WER and the `RunningAverageMeter` which is used to compute the average and standard deviation of the loss.
- **models**. Contains the different model architectures and layers related to handwritten document analysis that PyLaia provides.
- **nn**. This package consists of general NN layers and utilities. The main difference with `models` is that these are not tied to handwritten document analysis.
- **utils**. This is where utility functions which do not completely belong anywhere else are located.

Furthermore, PyLaia provides generic plug-and-play scripts to be executed without having to build a custom pipeline.

³Concretely, at each epoch and iteration start and end

A main design goal of PyLaia is to allow custom configurations for the abstractions provided by the toolkit to ease the important decisions that define a new model without having to implement all of the details from scratch.

5.4 Detailed design

Since the codebase is too large to be described with detail, we are only going to describe the critical parts of the code related to the experiments performed.

Below is the pseudocode of the script provided by PyLaia to start the HTR model training:

```
1 # Create the model
2 model = Model(hyperparameters)
3
4 # Prepare the trainer with its input training partition.
5 # It also contains the model which will be trained,
6 # the loss function and the gradient optimizer.
7 tr_dataset_loader = ImageDataLoader(...)
8 trainer = Trainer(
9     model=model,
10    criterion=CTCLoss(), # HTR loss function
11    optimizer=RMSprop(...),
12    data_loader=tr_dataset_loader,
13    ...)
14 # Same for the evaluator with its validation partition.
15 # We don't update any parameters so the criterion
16 # and optimizer are not necessary.
17 va_dataset_loader = ImageDataLoader(...)
18 evaluator = Evaluator(
19     model=model,
20     data_loader=va_dataset_loader,
21     ...)
22 # We can now create the experiment
23 experiment = HTRExperiment(trainer, evaluator)
24
25 # Set the trainer hooks. This will save the
26 # experiment if the model accuracy has improved
27 trainer.add_hook(EPOCH_END, SaveIfBestModel(...))
28
29 # We are ready to start training. Run!
30 experiment.run()
31
32 # Experiment finished. Save the model!
33 ModelCheckpointSaver(model).save()
```

Listing 5.1: Pseudocode for the pylaia-htr-train-ctc script

One can see how PyLaia's API abstracts the implementation details of the experiment in a short and concise manner thus allowing the user to focus on other duties to accomplish their goals.

As far as how the Trainer is implemented, below is the (simplified) core routine which performs the training:

```
1 while must_train:
2     call_hooks(EPOCH_START)
3     # For each input signal and its reference
4     for input, target in batches:
5         if not must_train: break
6         call_hooks(ITER_START)
7         # Zero all of the optimizer gradients for the tensors
8         # it will update (the learnable weights of the model)
9         optimizer.gradients = 0
10        # Forward pass. Compute the output of the model
11        # for the input
12        prediction = model(input)
13        # Compute the loss
14        loss = compute_loss(target, prediction)
15        # Compute gradient of the loss with respect to
16        # model parameters
17        loss.backward()
18        # Update the model parameters
19        optimizer.step()
20        # An iteration is defined as a forward and backward
21        # pass. Increment the counter
22        iterations += 1
23        call_hooks(ITER_END)
24        # An epoch is defined as a pass through the entire
25        # dataset. Increment the counter
26        epochs += 1
27        call_hooks(EPOCH_END)
```

Listing 5.2: PyLaiia's training algorithm

Throughout the codebase our custom code is interweaved with PyTorch's API, the later doing the heavy computation. Some examples are the call to `.backward()`, `.step()` or that our model is implemented as a PyTorch `nn.Module` which overrides the `__call__` operator so that you can use it as a function (see line 12 as an example).

The hooks execute any custom code that the user may have defined before starting the experiment. For example, the validation pass is set to be run by default when the `EPOCH_END` hook gets called.

CHAPTER 6

Solution development

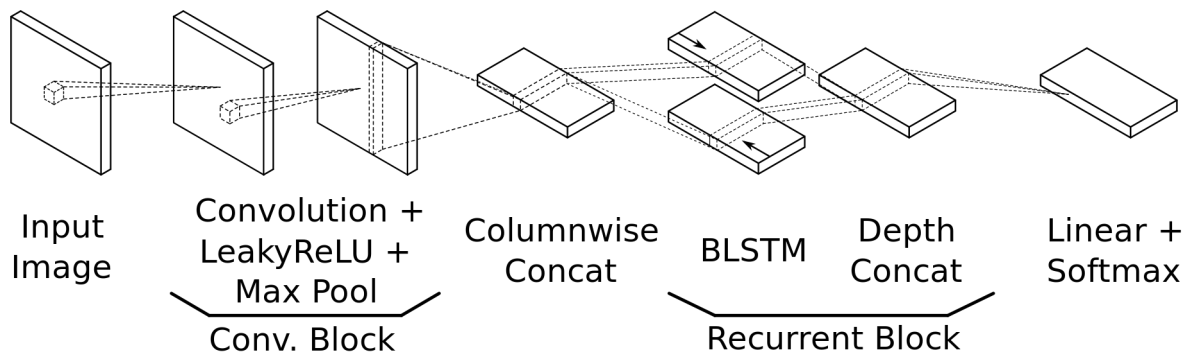


Figure 6.1: Architecture of the neural network presented in this work. Figure adapted from [3]

A general overview of the architecture is depicted in Figure 6.1. In this section, we explain the details of each of the blocks of our architecture and some other details of our system, we then explain the conditions under which our training was performed and display the results obtained with our system.

6.1 Model architecture

Convolutional blocks

Each convolutional block contains a two-dimensional convolutional layer (Conv) with a kernel size of 3×3 pixels, with both horizontal and vertical stride of 1 pixel. The number of filters at the n -th Conv layer is equal to $16n$.

In order to reduce overfitting, Dropout may be applied [23] at the input of some Conv layers (with dropout probability equal to 0.2). Dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. This helps prevent overfitting where the model has learnt some features of the training data which do not generalize well to the problem.

We use Leaky Rectifier Linear Units (LeakyReLU) [24] as the activation function (σ) in the convolutional blocks. The LeakyReLU function is defined by the following

Configuration	Values
Conv. filters	16 – 32 – 48 – 64 – 80
Maxpool (2×2)	Yes – Yes – Yes – No – No
Dropout (if used)	0 – 0 – 0.2 – 0.2 – 0.2

Table 6.1: Details of the configuration used in the convolutional blocks of our architecture.

expression:

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ a \cdot x & \text{otherwise} \end{cases}$$

where a controls the angle of the negative slope (we use $1 \cdot 10^{-2}$).

Finally, the output of the activation function is fed to a Maximum Pooling layer (Maxpool) with non-overlapping kernels of 2×2 pixels. The Maxpool layer is commonly used to reduce the dimensionality of the input images.

Table 6.1 shows the configuration used in each convolutional block. We use a total of 5 convolutional blocks in our architecture.

Recurrent blocks

Recurrent blocks are formed by bidirectional LSTM layers, that process the input image columnwise (meaning the different columns are arranged in rows, see “Columnwise concat” in Figure 6.1) in left-to-right and right-to-left order. The output of the two directions is concatenated depth-wise (see “Depth concat” in Figure 6.1). Thus, if D is the number of hidden units in each direction, the output of the bidirectional LSTM block has a depth of $2D$ channels.

Before each LSTM layer, Dropout is also applied here (with probability 0.5).

The number of hidden units of all 1D-LSTM layers is fixed to $D = 256$. We use a total number of 5 recurrent blocks.

Linear layer

Finally, each column after the recurrent LSTM blocks must be mapped to an output label. In order to do so, the depth is transformed from $2 \cdot D$ to L using an affine transformation (where L is equal to the number of characters + 1, for the CTC blank symbol).

As we did in the recurrent blocks, Dropout is applied before the Linear layer to prevent overfitting (also with probability 0.5).

The total number of model parameters for IAM and RIMES is 9601 508 and 9591 248, respectively. The difference lies in the number of output symbols.

6.2 Training

Some notes on the training process:

- All parameters of the network are trained by minimizing the CTC loss.
- Random distortions were not used to augment the training data.
- Batch normalization was not used in any of the experiments
- The RMSProp optimization algorithm [25] was used in all of the executions to incrementally update the parameters of the model using the gradients of the CTC loss.
- All of the experiments were performed on NVIDIA GTX 1080 GPUs
- The batch size varied between executions due to memory limitations since the GPUs were shared. However, the results are comparable among different batch sizes since no batch normalization was used in any execution.
- All of the images had some pre-processing done to increase the sharpness and contrast.
- In the case of the fixed height experiments, the images were resized to 128px, the median of all the samples.
- No lexicon or language model was applied to the network in any of the experiments.

During the training, we measure the label error rate, i.e. the lowest Character Error Rate (CER) of the network itself on the validation data. Training stops when the measure does not improve for 20 epochs.

Images for which the output sequence have a shorter length than the number of characters in the reference are a problem during training, as the loss cannot be computed correctly by the CTC function. Hence, a few¹ training images from RIMES were ignored, while on IAM this problem didn't occur.

¹Images [train2011-686-10, train2011-58-08] for the fixed height images and images [train2011-686-03, train2011-58-04, train2011-58-08, train2011-101-00] for the variable height images. All of them erroneously transcribed

6.3 Experiments

6.3.1. Dropout comparison

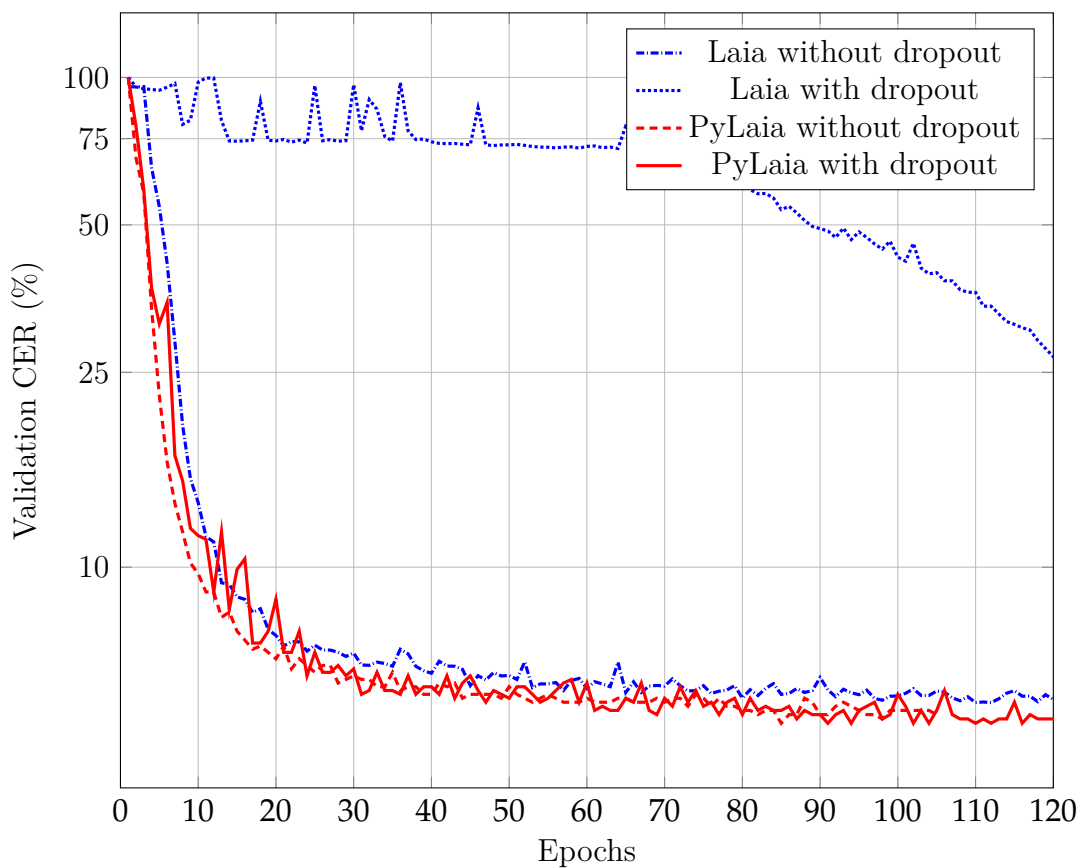
First, a comparison of the use of dropout in both Laia and PyLaia is shown. For both datasets, a learning rate of $\eta = 3 \cdot 10^{-4}$ was used. The CER is computed at line-level.

It seems like there is an implementation bug in Laia as seen by the number of epochs it takes the error rate to converge for both datasets compared to that of PyLaia. In the case of IAM, the error does ultimately converge around epoch 280 but reaching a higher CER than our implementation.

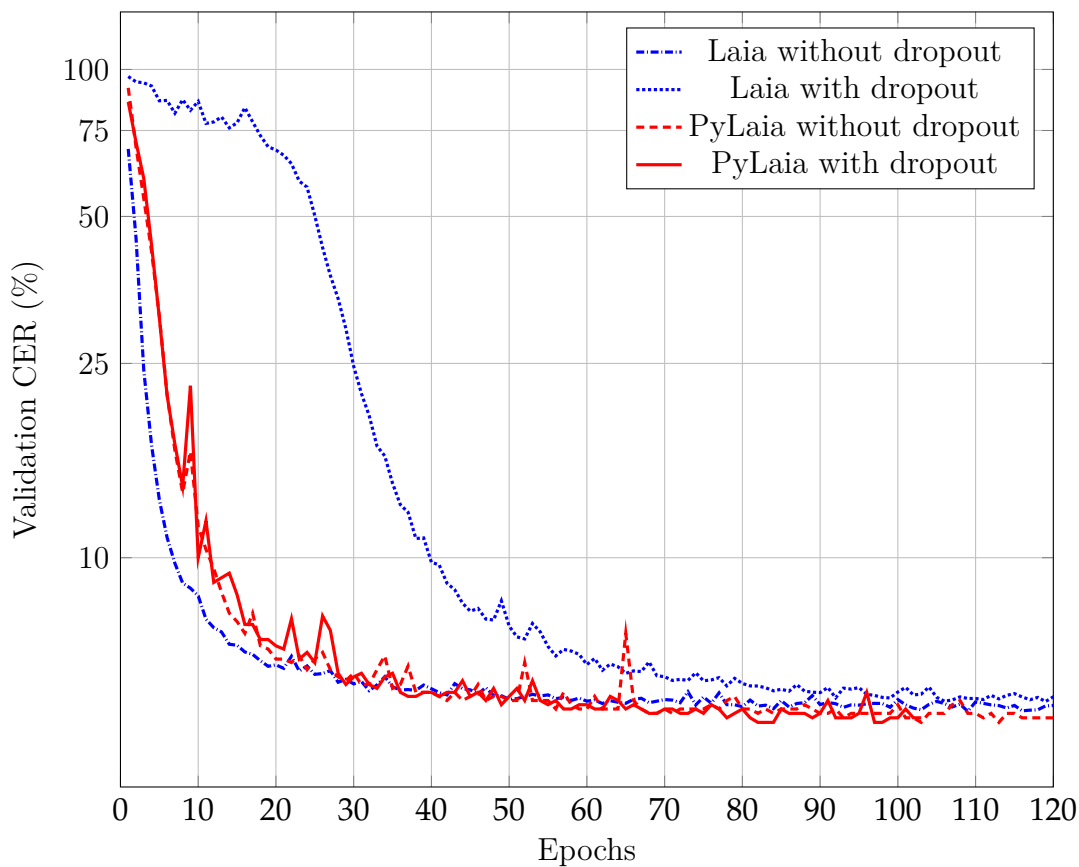
For both datasets, PyLaia achieves a good validation CER in a fair number of epochs. Table 6.2 displays the achieved test partition paragraph-level results after 20 consecutive epochs without an improved validation CER. As is expected the use of dropout improves the model accuracy for the test partition, however, the difference is not very significant especially in the case of the RIMES dataset.

Table 6.2: Dropout comparison of the character and word error rate on IAM and RIMES paragraphs for the **test** partition.

Toolkit	IAM		RIMES	
	CER (%)	WER (%)	CER (%)	WER (%)
Laia without dropout	8.2 [7.5–8.9]	25.1 [23.5–26.7]	3.4 [2.6–4.2]	13.6 [11.5–15.7]
Laia with dropout	8.5 [7.8–9.2]	25.8 [24.3–27.3]	3.7 [2.9–4.4]	14.2 [12.3–16.1]
PyLaia without dropout	7.6 [7.0–8.3]	23.8 [22.4–25.3]	3.2 [2.5–3.9]	12.8 [10.9–14.8]
PyLaia with dropout	7.4 [6.8–8.1]	23.4 [21.8–24.7]	3.1 [2.4–3.9]	13.0 [10.9–15.0]



(a) Dropout comparison between toolkits for the IAM dataset



(b) Dropout comparison between toolkits for the RIMES dataset

6.3.2. Sweep over learning rate (η)

In PyLaia, the loss function is slightly different from that of Laia, which can have an effect on the magnitude of the learning rate.

$$\text{Laia: } \frac{1}{N \cdot T} \sum_{n=1}^N \log P_{\Theta}(\mathbf{y} \mid \mathbf{x})$$

$$\text{PyLaia: } \frac{1}{N} \sum_{n=1}^N \log P_{\Theta}(\mathbf{y} \mid \mathbf{x})$$

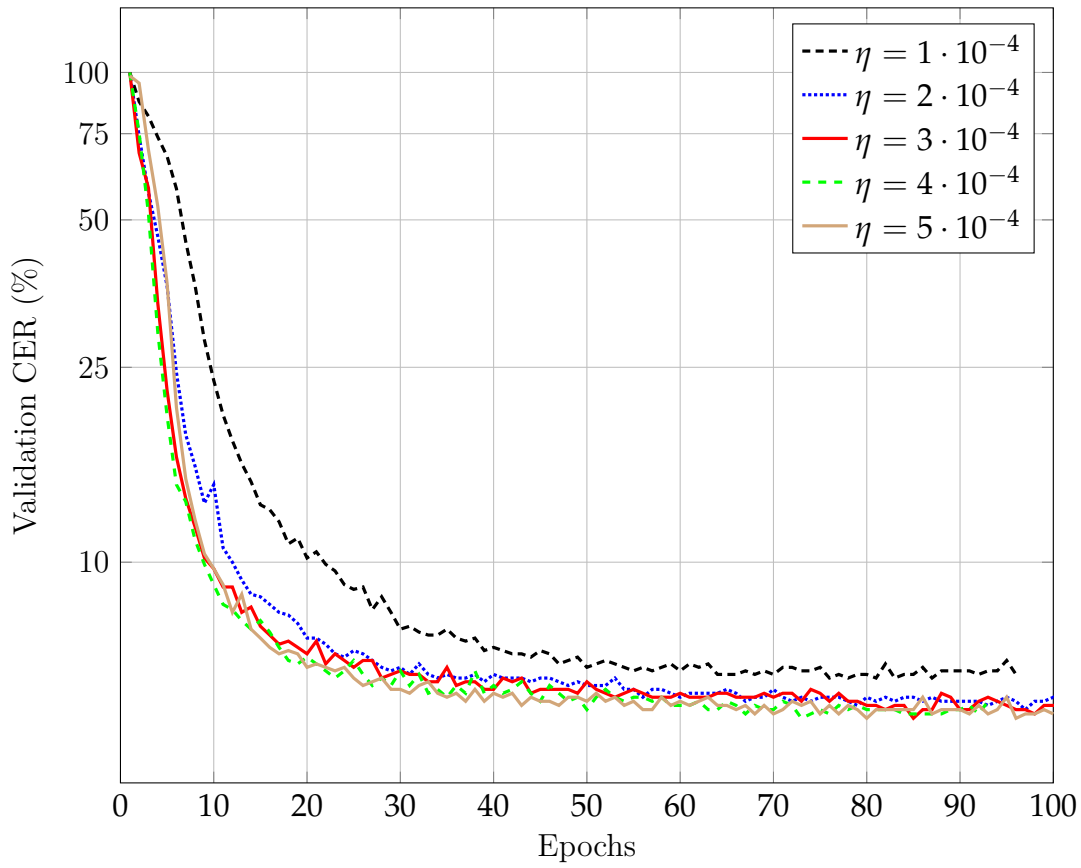
Figure 6.3: Comparison of the loss function between Laia and PyLaia. N is the number of samples in the batch, T is the number of frames in each sample of the batch

The main difference is that Laia’s loss function has a constant T which depends on the number of frames in the batch while PyLaia does not. The reason for this is that in PyLaia we do not assume that all of the samples have the same size, so T is not a constant for the whole batch. T would then have to be inside the sum as it would vary per sample. This requires the modification of the CTC library used.

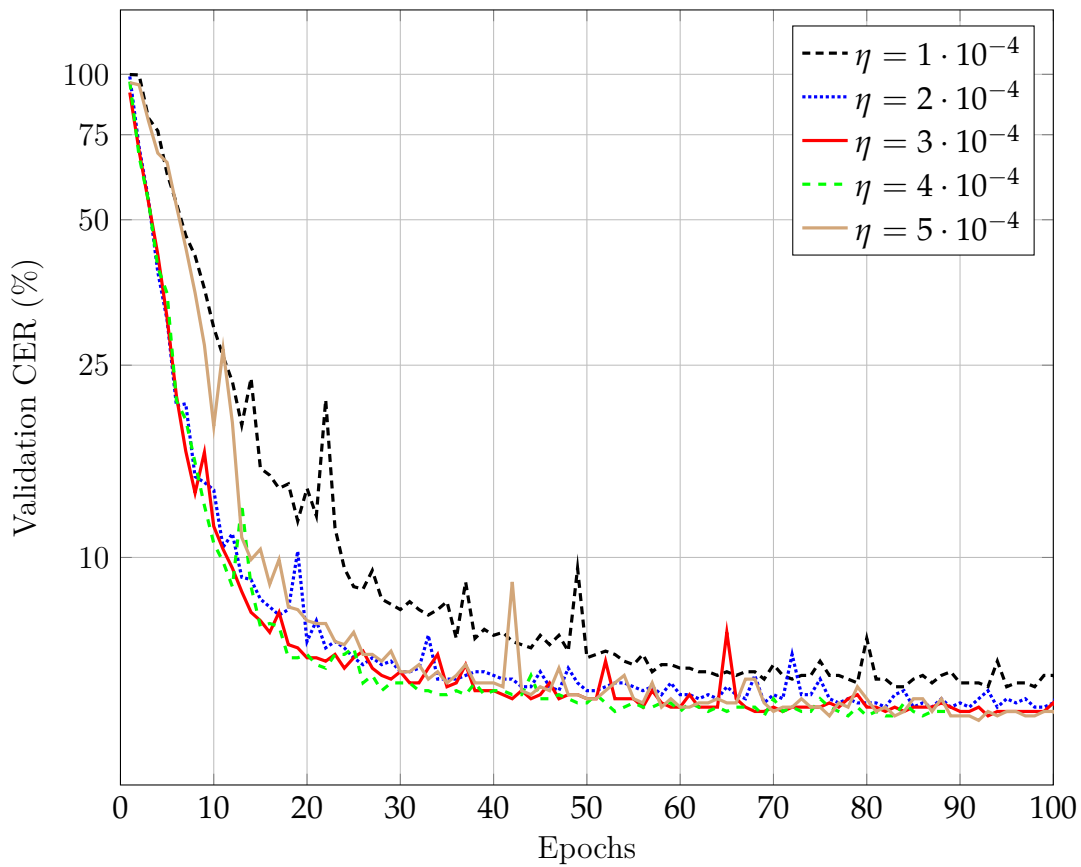
Considering that we are not dividing by T , the gradients computed in PyLaia have a larger magnitude which suggests that a smaller step size (learning rate) would be more appropriate. For this reason, we decided to validate the choice of learning rate for PyLaia instead of just using Laia’s value ($\eta = 3 \cdot 10^{-4}$).

Below, a comparison of the impact of η in PyLaia per dataset is shown. The CER is computed at line-level. No dropout was used.

Judging by the results for both dataset, a learning rate of $3 \cdot 10^{-4}$ seems to still be a good generic choice. Increasing or decreasing the learning rate too much seemed to cause bad curves in some executions. This is expected due to our choice of the optimizer. RMSprop divides the learning rate by an exponentially decaying average of squared gradients thus adapting itself to the magnitude of the gradients.



(a) Learning rate comparison for the IAM dataset



(b) Learning rate comparison for the RIMES dataset

6.3.3. Input height comparison

One of the biggest advantages of PyLaia is the ability to use input images of different heights. This circumvents the pre-processing step where all of the images are resized to fit a median height.

Issues may arise from the pre-processing in some cases, for example, if images heights are widely varying for a given character size, (maybe because the writers did not write in a straight line but diagonally, or because the dataset's bounding boxes were not chosen properly) then some input deformation occurs as shown by Figure 6.5. This also happens when images of the same height have varying character sizes. For HTR, it is in our best interest that the character size to image height proportion does not change much throughout the dataset.

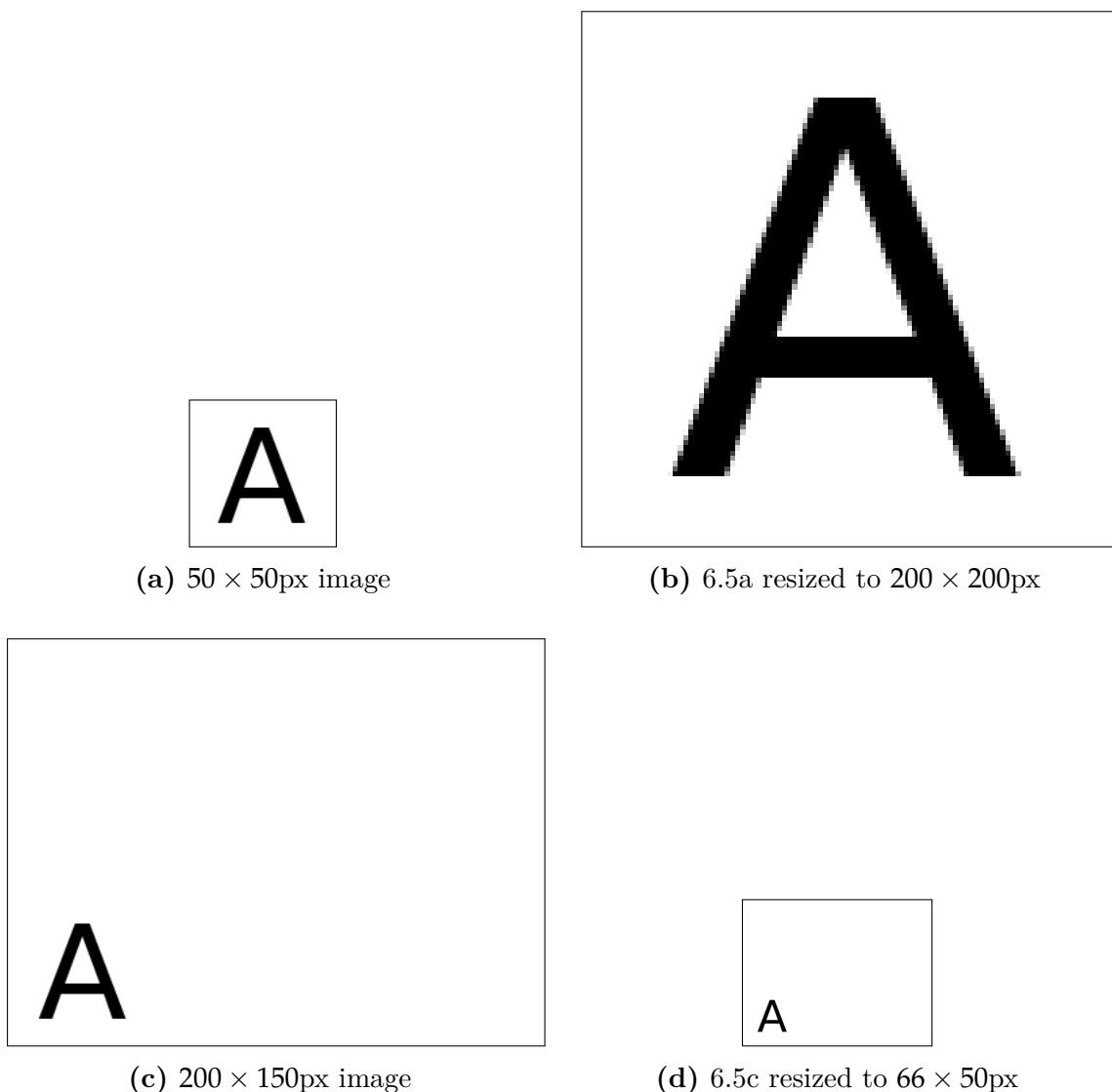


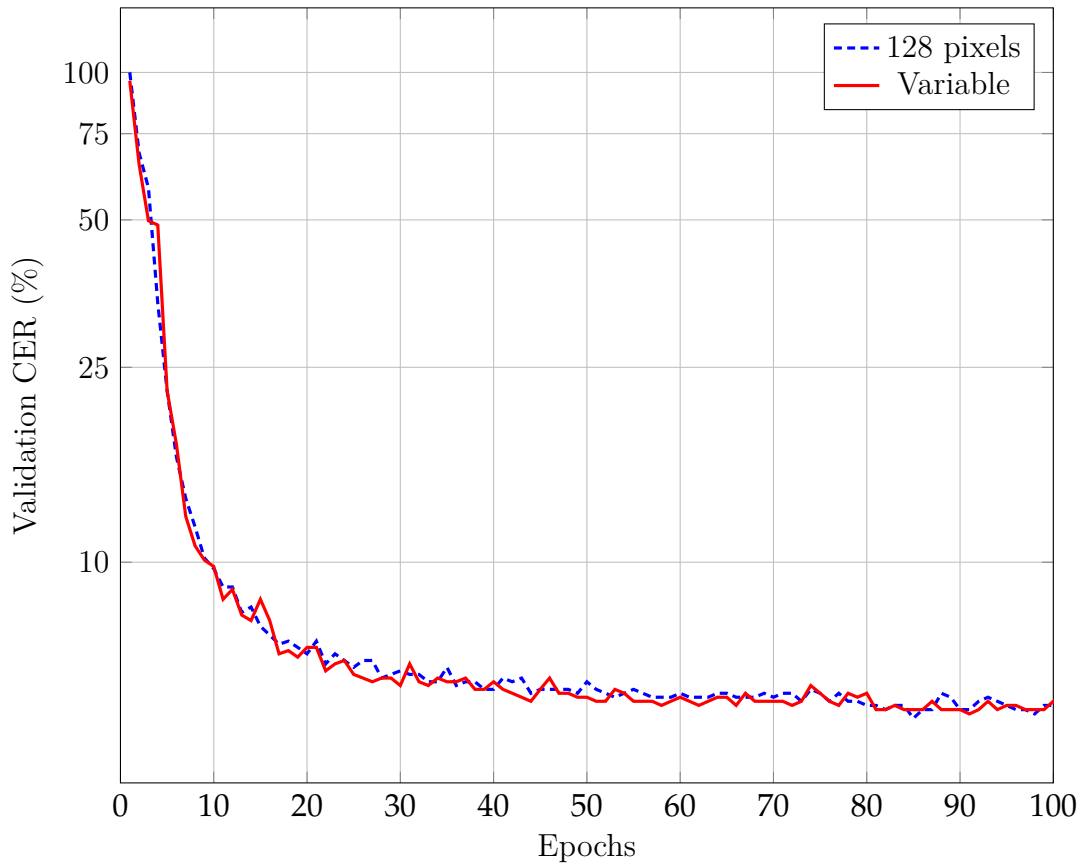
Figure 6.5: Example of how deformation may occur. Imagine the case where figures 6.5a and 6.5c are part of our dataset, take into account that their character size is exactly the same but 6.5c's bounding box was poorly chosen. If we were to resize the height of all images to that of figure 6.5c we would end up with a very stretched character in the case of 6.5a. On the other hand, if we did the opposite resizing to the height of 6.5a the opposite effect would happen

The solution to this issue is to use custom adaptive pooling layers at the end of the convolutional block since its the recurrent block who requires a fixed input height. Our implementation takes into account the size of each individual image within the batch to apply the adaptive pooling and divides each image into k sections where k is the desired image height.

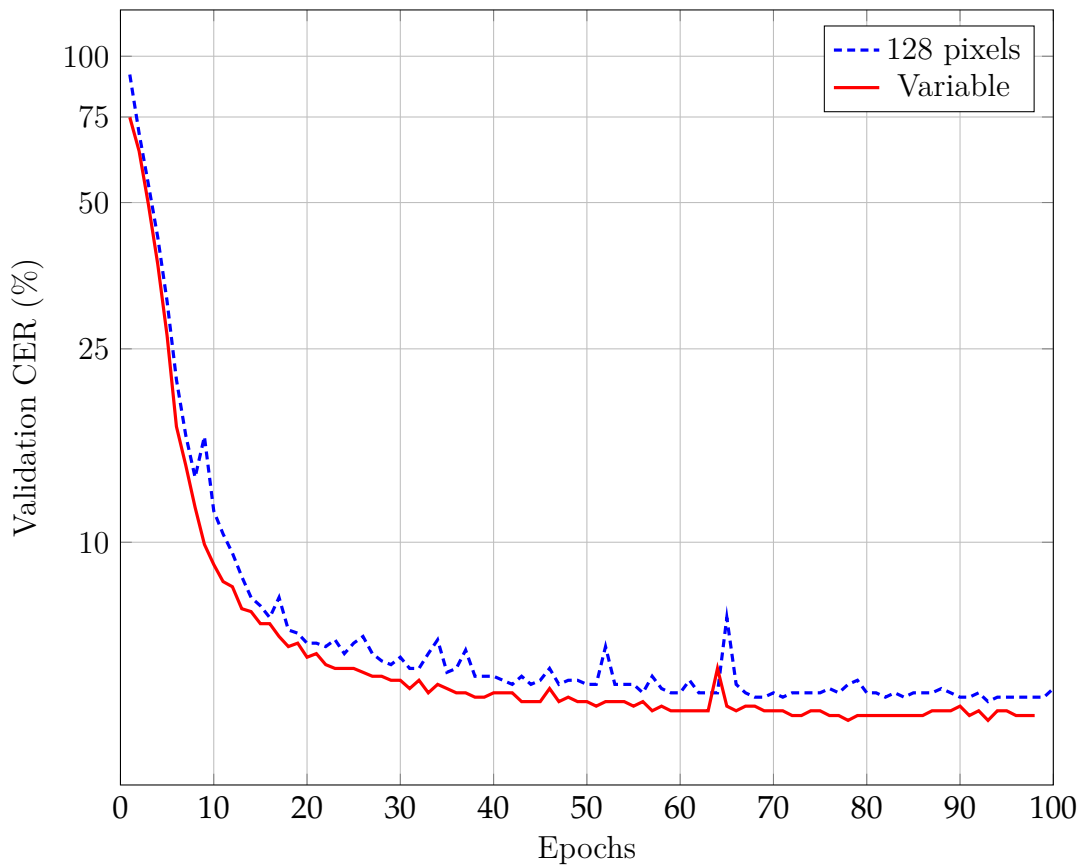
In the experiments showcased below, average adaptive pooling was used which is equivalent to a linear interpolation. For both datasets, a learning rate of $\eta = 3 \cdot 10^{-4}$ was used. The CER is computed at line-level.

A difference can be appreciated in the case of RIMES. This indicates that a greater amount of pre-processing noise is added, thus making the use of a variable height model worthwhile.

Keep in mind that the cost of the extra computation performed every epoch instead of once before training is negligible compared to that of the backward pass.



(a) Input height comparison for the IAM dataset



(b) Input height comparison for the RIMES dataset

6.3.4. PyLaia compared to Laia

The following figures directly compare the toolkit’s results. The error rates are computed at paragraph-level for a fixed height model. A learning rate of $\eta = 3 \cdot 10^{-4}$ was used. Below we demonstrate the performance of PyLaia. Our aim is to show that PyLaia successively obtains comparable results to Laia.

Our system considerably improves Laia’s results. The main reason for this is that Baidu’s CTC algorithm implementation can handle sequences of different lengths, however, Torch’s standard API for neural networks does not support this. Thus, in Laia, all of the input images in a single batch are centered and padded with zeros to match the size (width) of the largest image in the minibatch (as shown in Figure 6.8). PyLaia, being built on top of PyTorch did not have this limitation.

The results obtained show that our system competes with the current state-of-the-art approaches shown in Table 4.1 considering the lack of an explicit language model. We can make this assumption because of the improvement over Puigcerver’s results without a language model applied. In his paper, he provides metrics for Laia with and without a language model applied [3]. Meaning that if he already challenged previous best publications when a language model was applied, we would gain similar improvements if we were to do the same. However, additional research should be performed to confirm this hypothesis.

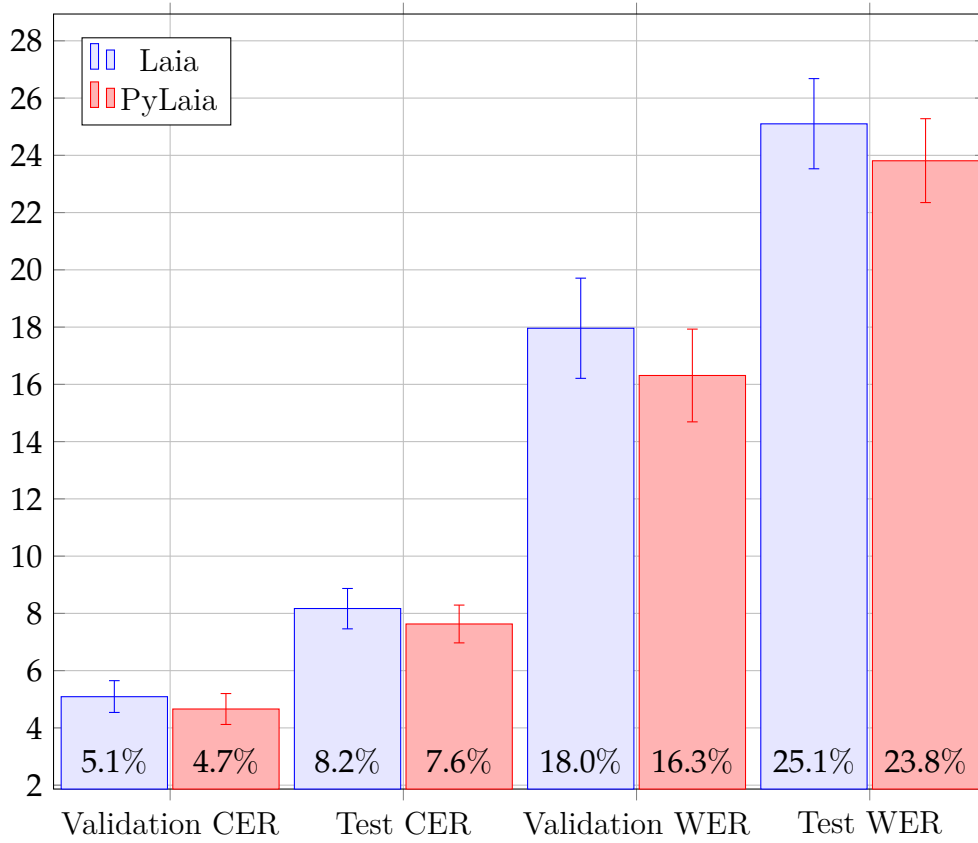
Finally, a comparison of the samples per second processed in each toolkit for the IAM dataset can be seen in Table 6.3. The results were obtained using an Intel Core i5-7500 CPU (3.40GHz) for the CPU decoding. PyLaia is shown to be 2 times faster in training and around 3.6 times faster in the decoding step over Laia. In addition, PyLaia benefits much more of having more GPU memory. The importance of using a GPU can be appreciated by looking at the results. Only GPU training was performed because CPU training would be too slow for the experiments to be finished in a reasonable time. In the case of CPU training, the clock speed is the main bottleneck in performance.

Table 6.3: Comparison between the toolkits’ processed samples per second in both the training and decoding steps using the IAM dataset.

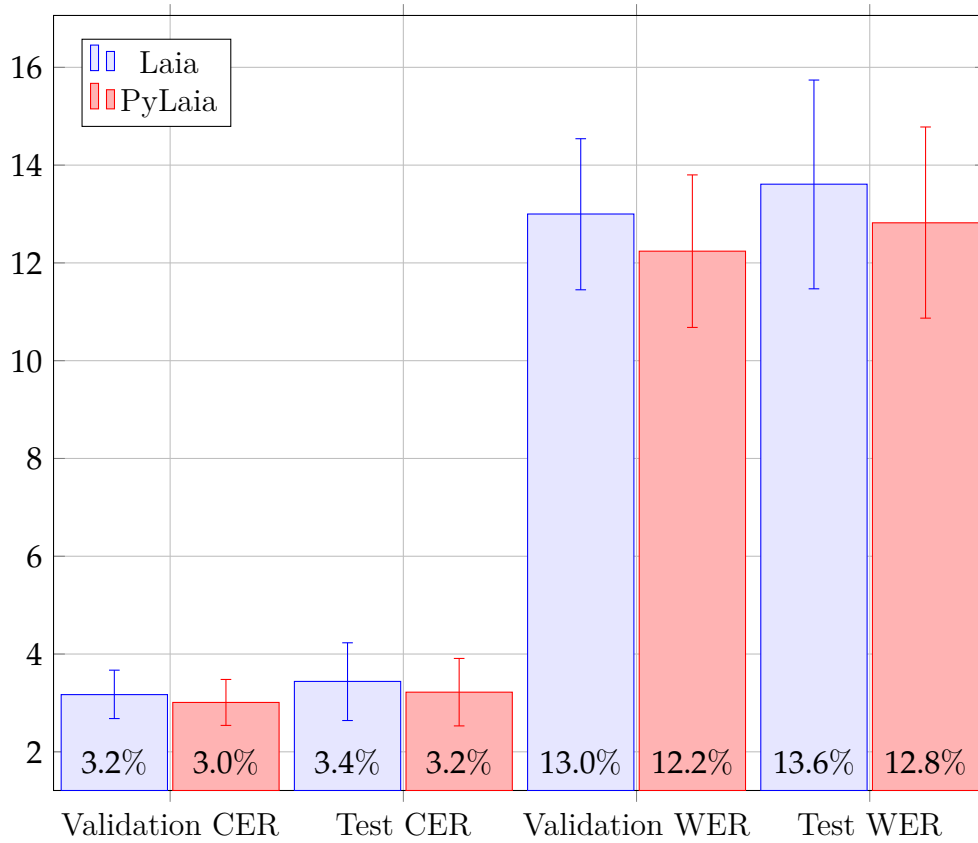
Batch size	Toolkit	Training		Decoding	
		GPU	CPU ¹	CPU ¹	GPU
5	Laia	12.2	—	—	21.5
	PyLaia	22.5	4.6	4.6	69.0
10	Laia	15.71	—	—	26.1
	PyLaia	30.8	5.8	5.8	91.1
15	Laia	— ²	—	—	28.4
	PyLaia	37.6	6.1	6.1	107.3

¹Laia does not support running using the CPU

²Laia ran out of memory for this batch size using our 8GB memory GPU



(a) Toolkit comparison for the IAM dataset



(b) Toolkit comparison for the RIMES dataset

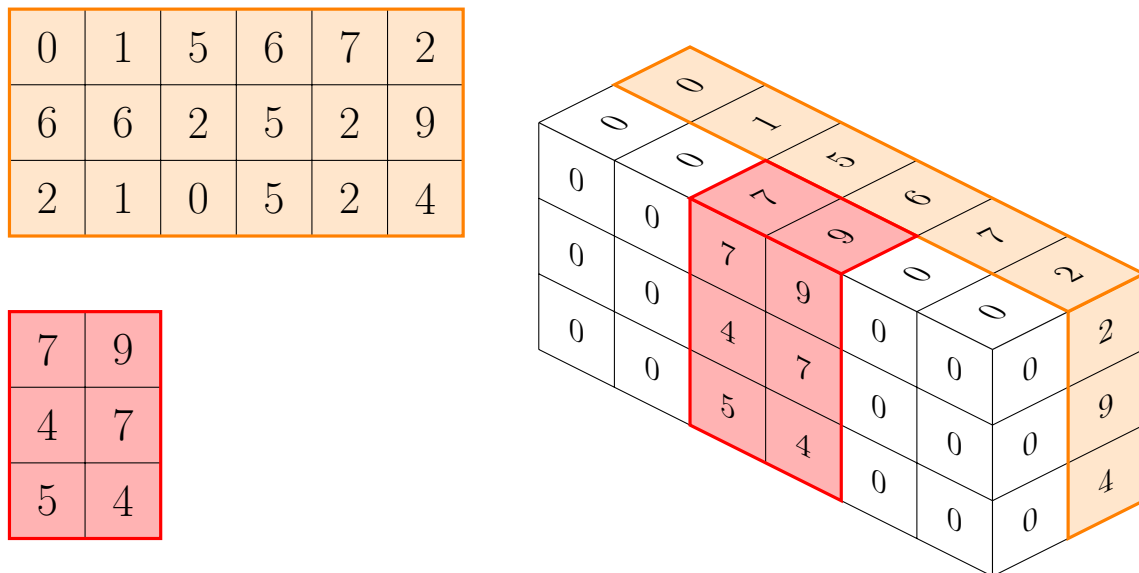


Figure 6.8: Combination of two 2D tensors into a single 3D tensor. The images whose width is less than the widest image in the batch are centered and padded with zeros

CHAPTER 7

Conclusion and future work

We have developed a fully-fledged deep learning toolkit for convolutional and recurrent neural networks, capable of performing handwritten document analysis experiments with proper architectural design and usability.

We have successfully replicated the experiments previously done using Laia to check their validity. By using two widely accepted datasets for HTR experiments, the correctness of our PyLaia implementation has been validated by comparing its results to Laia's and have found an improvement over all of the metrics evaluated.

Multiple PRHLT researchers have already started using PyLaia successfully in their experiments, even for other tasks different from HTR such as Keyword Spotting, which was one of the goals we aimed to achieve. What is more, a few research publications are being prepared using the toolkit.

Several questions remain unanswered, such as the real impact of using a language model to further improve the recognition accuracy and to explicitly compare our results to those performed by other papers without making any assumptions over the language model independence with the model training process. Furthermore, we have not explored the impact of either the usage of distortions for data augmentation nor the use of batch normalization.

A question is raised as well on the comparison between toolkits for other metrics such as memory usage. We could not scientifically compare this in this work due to environment limitations regarding shared graphics processing units with other users. However, we have perceived that the memory usage is significantly reduced.

Finally, as is the case for any piece of software, its development is never completely finished. Bugs could appear and new features may be implemented. This is critical to ensure that the software stays up to date and to avoid legacy software.

CHAPTER 8

Degree relationship

Having taken the degree specialization most closely related to the subject of this work, I have had the opportunity to employ a large array of the skills that have been taught to me. This is especially the case for the subjects “Perception” and “Machine Learning” which directly relate to the concepts used throughout this work.

In addition, by writing PyLaia, I have had the chance to apply many algorithms, data structures, and software engineering concepts. The internal architecture components were designed to pursue a clean and clear interaction, achieving a versatile system to the best of my ability. Moreover, relevant software development technologies and practices were used such as git for version control, Travis for continuous integration, Docker for easy deployment and the usage of unit testing to validate the code correctness.

Lastly, during the making of this work, I got to develop myself in relation to some critical transversal competencies such as “Design and project”, “Application and critical thinking” and “Continuous learning” among others.

Acronyms

- CER** Character Error Rate. 15, 16, 21, 27–31, 33, 34, 36
- CNN** Convolutional Neural Network. vii, 6, 7, 12, 17
- CTC** Connectionist Temporal Classification. vii, 9, 10, 12, 15, 20, 21, 26, 27, 30, 35
- CUDA** Compute Unified Device Architecture. 1, 20
- DAG** Directed Acyclic Graph. 19
- FAIR** Facebook Artificial Intelligence Research. 1
- HTR** Handwritten Text Recognition. iv, v, 1, 2, 7–13, 15, 19, 20, 23, 32, 39
- IAM** Institut für Informatik und angewandte Mathematik. iv, v, vii, x, 2, 12, 16, 26–29, 31, 34–36
- LOB** Lancaster-Oslo/Bergen. 12
- LSTM** Long Short Term Memory. vii, 8, 9, 12, 15, 17, 20, 26
- LSTM-RNN** Long Short Term Memory Recurrent Neural Network. 8, 17
- MDLSTM-RNN** Multidimensional Long Short Term Memory Recurrent Neural Network. 15
- NN** Neural Network. vii, 5, 15, 17, 21
- OCR** Optical Character Recognition. 11, 12
- PRHLT** Pattern Recognition and Human Language Technology. 1, 39
- RIMES** Reconnaissance et Indexation de données Manuscrites et de fac similÉS. iv, v, vii, x, 2, 13, 15, 16, 26–28, 31, 33, 34, 36
- RNN** Recurrent Neural Network. vii, 7–9, 12, 19
- WER** Word Error Rate. 15, 16, 21, 28, 36

Bibliography

- [1] J. Puigcerver, D. Martin-Albo, and M. Villegas. “Laia: A deep learning toolkit for htr”. <https://github.com/jpuigcerver/Laia>, 2016. GitHub repository.
- [2] R. Collobert, K. Kavukcuoglu, and C. Farabet. “Torch7: A matlab-like environment for machine learning”. In *BigLearn, NIPS Workshop*, 2011.
- [3] J. Puigcerver. “Are Multidimensional Recurrent Layers Really Necessary for Handwritten Text Recognition?” In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, IEEE, nov 2017
- [4] 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, Mar 1994.
- [5] P. Voigtlaender, P. Doetsch, and H. Ney. “Handwriting recognition with large multidimensional long short-term memory recurrent neural networks”. In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 228–233, Oct 2016.
- [6] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. *Neural Computation*, 9(8):1735–1780, November 1997.
- [7] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. “Listen, attend and spell”. *CoRR*, abs/1508.01211, 2015.
- [8] 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, May 2009.
- [9] George Nagy. “Disruptive developments in document recognition”. *Pattern Recognition Letters*, 79:106 – 112, 2016.
- [10] H. Bunke, M. Roth, and E.G. Schukat-Talamazzini. “Off-line cursive handwriting recognition using hidden markov models”. *Pattern Recognition*, 28(9):1399 – 1413, 1995.
- [11] Théodore Bluche, Jérôme Louradour, and Ronaldo O. Messina. “Scan, attend and read: End-to-end handwritten paragraph recognition with MDLSTM attention”. *CoRR*, abs/1604.03286, 2016.

- [12] Alex Graves and Jürgen Schmidhuber. "Handwriting recognition with multidimensional recurrent neural networks". In *Proceedings of the 21st International Conference on Neural Information Processing Systems, NIPS'08*
- [13] Joan Andreu Sánchez, Verónica Romero, Alejandro Toselli, and Enrique Vidal. "ICFHR 2014 HTRtS: Handwritten Text Recognition on tranScriptorium Datasets". In *International Conference on Frontiers in Handwriting Recognition (ICFHR)*, 2014.
- [14] U.-V. Marti and H. Bunke. "The IAM-database: an english sentence database for offline handwriting recognition". *International Journal on Document Analysis and Recognition*, 5(1):39–46, Nov 2002.
- [15] a2ia "The RIMES dataset". https://www.a2ialab.com/doku.php?id=rimes_database:start
- [16] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [17] Théodore Bluche. "*Deep Neural Networks for Large Vocabulary Handwritten Text Recognition*". Theses, Université Paris Sud - Paris XI, May 2015.
- [18] P. Voigtlaender, P. Doetsch, and H. Ney. "Handwriting Recognition with Large Multidimensional Long Short- Term Memory Recurrent Neural Networks". In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 228–233, Oct 2016.
- [19] P. Doetsch, M. Kozielski, and H. Ney. "Fast and Robust Training of Recurrent Neural Networks for Offline Handwriting Recognition". In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 279–284, Sept 2014.
- [20] V. Pham, T. Bluche, C. Kermorvant, and J. Louradour. "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 285–290, Sept 2014.
- [21] Théodore Bluche. "Joint line segmentation and transcription for end-to-end handwritten paragraph recognition". *CoRR*, abs/1604.08352, 2016.
- [22] A. Paszke, S. Gross, and S. Chintala "PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration". <https://pytorch.org>
- [23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research*, 15:1929–1958, 2014.
- [24] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In *Proc. ICML*, volume 30, 2013.
- [25] T. Tieleman and G. Hinton. Lecture 6.5—"RmsProp: Divide the gradient by a running average of its recent magnitude". COURSERA: Neural Networks for Machine Learning, 2012.