Urdu Handwriting Recognition Using Deep Learning

Submitted by:

Shehryar Malik	2015-EE-167
M. Naeem Maqsood	2015-EE-168
Abdur Rehman Ali	2015-EE-188

Supervised by: Dr. Ubaid Ullah Fayyaz

Department of Electrical Engineering

University of Engineering and Technology Lahore

Urdu Handwriting Recognition Using Deep Learning

Submitted to the faculty of the Electrical Engineering Department
of the University of Engineering and Technology Lahore
in partial fulfillment of the requirements for the Degree of

Bachelor of Science

in

Electrical Engineering.

Internal Examiner

External Examiner

Director
Undergraduate Studies

Department of Electrical Engineering

University of Engineering and Technology Lahore

Declaration

We declare that the work contained in this thesis is our own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed: _____

Date: _____

Signed: _____

Date: _____

Signed: _____

Date: _____

Acknowledgments

This work would not have been possible without the support of several individuals and organizations.

Foremost, we would like to express our gratitude to our supervisor Dr. Ubaid Ullah Fayyaz for his guidance in this project. We also thank our co-supervisor Dr. Kashif Javed for his support.

We would also like to thank Mrs. Qurat-ul-Ain Akram of the Center for Language Engineering for her continuous support throughout the project. Apart from providing us with space on the Center's premises, she also arranged for the Center to provide funding for this project.

In order to prepare the dataset for this project, we contacted several schools and colleges. We are indebted to all those who agreed to help us in this regard.

Last, but not the least, our sincerest gratitude to all those individuals and organizations who have helped ensure high quality Open CourseWares (OCWs) for all those who want to study artificial intelligence, in general, and deep learning, in particular. Had it not been for them, we would not even have had the technical expertise to begin with this project. As such, this thesis is dedicated to them.

For those who create and ensure OCWs for AI enthusiasts

Contents

List of Figures

List of Tables

Abbreviations

OCR	**O**ptical **C**haracter **R**ecognition
ANN	**A**rtificial **N**eural **N**etwork
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural N**e**twork
RNN	**R**ecurrent **N**eural **N**etwork
BRNN	**B**idirectional **R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory
BLSTM	**B**idirectional **L**ong **S**hort **T**erm **M**emory
MDLSTM	**M**ulti-**D**imensional **L**ong **S**hort **T**erm **M**emory
CTC	**C**onnectionist **T**emporal **C**lassification
CE	**C**ross **E**ntropy
ReLU	**Re**ctified **L**inear **U**nit
LM	**L**anguage **M**odel

Abstract

Optical character recognition aims to recognize text in images. Recent breakthroughs in deep learning have revolutionized OCR systems for languages such as English. However, their impact on Urdu has been minimal. This thesis aims to bridge this gap. We develop a new dataset comprising of around $15,000$ images of Urdu handwritten text lines and use it to train different deep learning architectures. The first is the standard CNN-RNN architecture that optimizes the Connectionist Temporal Classification function. We also incorporate a trigram language model with this architecture to further improve performance. The second architecture is an attention-based encoder-decoder network that optimizes the cross-entropy function for each character in the transcription. We achieve accuracies of 91.51% and 90.07% on the two architectures respectively. These results are comparable to the state-of-the-art results on English datasets.

Chapter 1

Introduction

Consider the problem of reading text written on a piece of paper. The human mind accomplishes this task through a series of steps. It might begin by first identifying which parts of the paper contain text. Next, it would decide where to start reading from. To do that, it would also have to recognize the fact that the paper might contain many lines and that lines needs to be read individually in a certain sequential order. If a word is not written clearly then the mind knows to use previous and future contexts of the text to try to identify it. However, if the text is just a collection of random words, then the mind recognizes that and understands that it should not take into account the context. If the text is multilingual, then the mind also has to decide which parts of it belong to which language. It also needs to take into account cases where some of the languages are read left-to-right and the others right-to-left. In short, several complex decisions need to be made which also require an *understanding* of the text.

Recent breakthroughs in the field of machine learning, especially in the form of deep learning, have made systems that can read text documents more realizable than ever before. Apart from bringing artificial intelligence one step closer to human intelligence, these *optical character recognition* systems can assist people in a wide range of affairs. For example, state institutions can use these systems to digitize old records, thus sparing them the need of large storage areas. Similarly, libraries can easily create electronic versions of any old books they might have, thus preserving them for eternity.

Urdu is the national language of Pakistan and is spoken by over a 100 million people [1]. It is written from right to left using the Persian script and has 58 letters [2]. Characters physically join together to form ligatures. Each word contains one or more ligatures. The shape of each character varies according to its position in the ligature. Unlike English, no space is inserted between words in Urdu.

This thesis aims to develop an optical recognition system for Urdu using deep learning.

Chapter 2

Supervised Learning for Classification

Machine learning offers a powerful set of algorithms to analyze and model data. These algorithms can be used in a variety of settings. One such setting is supervised learning. Concretely, in supervised learning we assume a training set $S_{train} = \{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$ drawn from some distribution $\mathcal{D}_{x,y}$. Each input-output pair $(x^{(i)}, y^{(i)})$ is referred to as a training example. All training examples are assumed to be independent and identically distributed (i.i.d.). Supervised learning aims to find a function f that maps inputs $x^{(i)}$ to their corresponding outputs $y^{(i)}$.

We define the training error $\hat{\epsilon}_{train}$ to be the error in f on the training set. However, we are usually more interested in the error in f on any example drawn from the distribution $D_{x,y}$. To that end, we define ϵ to be the generalization error in f on $D_{x,y}$. We may approximate the generalization error by testing f on a test set S_{test} also drawn from $D_{x,y}$. We denote the error on this set as $\hat{\epsilon}_{test}$. Note that it is essential that f is only evaluated on the test set at the very end of the supervised learning process (when no more changes to f are to be made). In order to approximate the generalization error of f during the supervised learning process, a third set, known as the validation set and denoted with S_{val}, is drawn from $D_{x,y}$. The errors are sometimes also referred to as losses.

For the purpose of this thesis we shall assume a discrete-valued setting for the outputs y (we will drop the superscript i when there is no fear of ambiguity). Specifically, we shall assume that $y \in \{0, ..., |V| - 1\}$. The problem of supervised learning, thus, becomes of classifying inputs x into their correct classes. As such, we shall refer to f as a classifier.

2.1 Probabilistic Classifiers

There are many approaches to setting up a classifier. One approach involves having it output the correct label y for each input x. Such a classifier is known as a discriminant function. However, we may also use a probabilistic approach where we have the classifier

output probabilities with which x belongs to each of the $|V|$ classes. We may then select the most probable class.

Denote the parameters of a probabilistic classifier with $\boldsymbol{\theta}$. Then from the assumption that all training examples are i.i.d. it follows that:

$$p(S_{train}|\boldsymbol{\theta}) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \boldsymbol{\theta}) \tag{2.1}$$

which, using Bayes' rule, gives:

$$p(\boldsymbol{\theta}|S_{train}) = \frac{p(S_{train}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(S_{train})} \tag{2.2}$$

Each class probability C_k can then found by integrating over all possible values of $\boldsymbol{\theta}$:

$$P(C_k|x, S_{train}) = \int_{\boldsymbol{\theta}} p(C_k|x, \boldsymbol{\theta})p(\boldsymbol{\theta}|S_{train}) \tag{2.3}$$

However, in practice, $\boldsymbol{\theta}$ is very high dimensional and consequently the integral in Equation 2.3 becomes intractable. In such cases, it is common to approximate it using a single value of $\boldsymbol{\theta}$.

$$\int_{\boldsymbol{\theta}} p(C_k|x, \boldsymbol{\theta})p(\boldsymbol{\theta}|S_{train}) \approx p(C_k|x, \boldsymbol{\theta}_{MAP}) \tag{2.4}$$

where:

$$\boldsymbol{\theta}_{MAP} = \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|S_{train}) \tag{2.5}$$

This is known as the maximum a priori (MAP) approximation. We obtain the following after substituting Equation 2.2 in the equation above:

$$\boldsymbol{\theta}_{MAP} = \operatorname*{argmax}_{\boldsymbol{\theta}} p(S_{train}|\boldsymbol{\theta}) \tag{2.6}$$

where we have assumed a uniform prior on the values of $\boldsymbol{\theta}$ and noted that $\boldsymbol{\theta}_{MAP}$ is independent of $p(S_{train})$.

Substituting Equation 2.1 into Equation 2.6 gives:

$$\boldsymbol{\theta}_{MAP} = \operatorname*{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \boldsymbol{\theta}) \tag{2.7}$$

We can, therefore, find $\boldsymbol{\theta}_{MAP}$ by minimizing the following *objective* function with respect to $\boldsymbol{\theta}$:

$$O = -\ln \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \boldsymbol{\theta}) = -\sum_{i=1}^{m} \ln p(y^{(i)}|x^{(i)}, \boldsymbol{\theta}) \tag{2.8}$$

Note that the logarithm is a monotonic function and consequently minimizing a function is the same as minimizing its logarithm.

2.2 The Cross-Entropy Function

Let $q(x^{(i)})_j$ denote the actual probability of the input $x^{(i)}$ belonging to the class j. In the case when $x^{(i)}$ can only belong to one class (denoted with k):

$$q(x^{(i)})_j = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

Also, let $p(x^{(i)})_j$ denote the probability a classifier assigns to class j for the input $x^{(i)}$. Then the cross-entropy objective function is defined as:

$$H = -\ln \prod_{i=1}^{m} \sum_{j=1}^{|V|} q(x^{(i)})_j p(x^{(i)})_j \tag{2.10}$$

For the special case in Equation 2.9, Equation 2.10 will reduce to Equation 2.8. This is because the inner summation will reduce to $p(x^{(i)})_k$ which is equal to $p(y^{(i)}|x^{(i)}, \boldsymbol{\theta})$. Here, $y^{(i)}$ is the true class of $x^{(i)}$ (denoted by k above).

Usually, the outputs of a classifier are real-valued numbers (without any restrictions on their range). The softmax function is often used to convert these numbers into probabilities:

$$\text{softmax}(o_j) = \frac{e^{o_j}}{\sum_{k=1}^{|V|} e^{o_k}} \tag{2.11}$$

where o_j denotes the probability that the classifier assigns to class j for some input.

For the remainder of this thesis, we shall assume that the cross-entropy function first converts the classifier's outputs to valid probabilities through the softmax function and then applies the actual cross-entropy function.

2.3 Sequence-to-Sequence Problems

Consider the optical character recognition problem outlined in Chapter 1. Let the image be the input to a classifier and the transcription be the desired output. Let $\boldsymbol{x} \in \mathbb{R}^{H \times W}$ be the image matrix. We shall denote its ith column by $\boldsymbol{x}_i$ and the jth element in the ith column by x_{ij}. Also, let $\boldsymbol{y}$ denote a vector $[y_1, y_2, ..., y_L]$ whose ith element corresponds to the ith character of the transcription. Note that each $y_i \in \{1, ..., |V|\}$ (we assign appropriate ids to each of the $|V|$ characters).

In this example, both the input and the output can be thought of as *sequences*. The input is a sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_W$ whereas the output is a sequence of character ids. Such problems are referred to as sequence-to-sequence problems.

The main challenge in sequence-to-sequence problems is usually that of alignment. In the example above, it may not be easy to divide the image into segments each with only one character. In this section, we describe two approaches to solving the alignment problem.

2.3.1 Connectionist Temporal Classification (CTC)

The CTC approach [3] defines a new so-called *blank* token in the vocabulary and assigns it the id $|V| + 1$. We denote this extended vocabulary with V'. For each element in the input sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T$, the classifier predicts a probability for observing each character in V'. Therefore, if $\boldsymbol{o_t}$ denotes the output of the classifier for input $\boldsymbol{x}_t$, then o_{tj} denotes the probability of observing the character j for that input. Let L'^T be the set of all possible sequences of length T over the vocabulary V'. Then, for any *path* $\boldsymbol{\pi} \in L'^T$:

$$p(\boldsymbol{\pi}|\boldsymbol{x}^{(i)}, \boldsymbol{\theta}, S_{train}) = \prod_{t=1}^{T} o_{t\pi_t}^{(i)} \tag{2.12}$$

where π_t denotes the tth element in π. Note that we assume that the labels at different time steps are conditionally independent of each other (where we are conditioning on $\boldsymbol{x}$ and S_{train}).

Let us define a many-to-one map $\beta : L'^T \mapsto L^{\leqslant T}$. The β function maps each $\boldsymbol{\pi} \in L'^T$ (which is of length T) to a sequence of length less than or equal to T. It does so by first merging all repeated characters and then deleting the blank tokens. For example, $[aa - abcb - -d]$ will map onto $[aabcbd]$. From this example, it is clear that the blank token is useful because it (i) separates two consecutive instances of the same character, and (ii) can represent a gap in the input sequence (e.g. the white space between two characters).

Note that more than one path in L'^T may be *collapsed* onto the same sequence $\boldsymbol{l} \in L^{\leqslant T}$ by the β function. As all paths are mutually exclusive (only one can occur at a time), we have:

$$p(\boldsymbol{l}|\boldsymbol{x}^{(i)}, \boldsymbol{\theta}, S_{train}) = \sum_{\boldsymbol{\pi} \in \beta^{-1}(\boldsymbol{l})} p(\boldsymbol{\pi}|\boldsymbol{x}^{(i)}, \boldsymbol{\theta}, S_{train}) \tag{2.13}$$

Substituting Equation 2.13 into Equation 2.8 will give us an objective function, that when minimized will yield $\boldsymbol{\theta}_{MAP}$. However, the problem with Equation 2.13 is that for a given $\boldsymbol{y}$ we would have $2^{T-|\boldsymbol{y}|^2+|\boldsymbol{y}|(T-3)}3^{(|\boldsymbol{y}|-1)(T-|\boldsymbol{y}|)-2}$ possible paths. Computing probabilities for each of these paths would be too time-consuming. [3] presents a dynamic-programming algorithm, that recursively evaluates the RHS of Equation 2.13.

Classifiers having *recurrent* connections best work for CTC as they allow it to condition on previous predictions at each time step.

2.3.2 Decoding Strategies for CTC

In the CTC approach, the classifier outputs a probability distribution over the entire vocabulary for each time step. The goal of decoding is to find the sequence that has the highest probability. Here, we distinguish between a path and a sequence. A path is simply constructed by selecting one character at each time step while a sequence is what we get after applying the β function on a path.

The best path decoding (greedy search) selects the most probable character at each time step. However, it is not guaranteed to find the most probable sequence.

In contrast, beam search keeps track of k sequences. At each time step, it first extends each path with all of the characters in the vocabulary and then chooses the paths that correspond to the top k sequences with the highest probability.

In some cases, it may also be desirable to select the most probable sequence that satisfies some constraints (such as language constraints). To that end, [4] proposes a token-passing algorithm.

However, in tasks such as handwriting recognition, constraining the outputs may not be a good idea. Firstly, because it is usually not possible to create a dictionary that contains all words in a certain language. Secondly, new words get created often and hence we would need to regularly update our dictionary Thirdly, we might have to deal with cases where the handwritten text might not contain any valid words (for example a random string of characters and numbers).

To solve this problem, [5] proposes a word-based language model that only partially takes into account a language model during decoding. [6] proposes an alternative approach to handle this problem by only constraining words in the output by a dictionary, while simultaneously allowing for random strings of special characters and numbers.

2.3.3 Encoder-Decoder Architecture

The idea behind an Encoder-Decoder architecture is to have the classifier handle the alignment problem internally. We present a detailed discussion on this approach in Chapter 4.

2.4 Training Classifiers

Up till now, we have only discussed how to set up an objective function, that when minimized yields the desired $\boldsymbol{\theta}_{MAP}$. In most cases, the objective function does not have a closed-form solution. Instead, we have to resort to numerical methods. In this section, we present the gradient descent method and some of its variants.

In the following discussion, $\boldsymbol{\theta} \in \mathbb{R}^d$ denotes the parameters of the classifier and O denotes the objective function we are trying to minimize.

2.4.1 Gradient Descent

The gradient descent method performs the following update:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} O(\boldsymbol{\theta}) \tag{2.14}$$

i.e. it repeatedly takes a *step* towards the minimum point. α controls the step size and is known as the learning rate. There are several ways of applying gradient descent: (i) vanilla gradient descent computes $O(\boldsymbol{\theta})$ for the entire training set and then applies the

update rule, (ii) stochastic gradient descent computes $O(\boldsymbol{\theta})$ for one example, performs the update rule and then moves onto the next example, and (iii) mini-batch gradient descent divides the training data into *minibatches*, performs the update for one minibatch and then moves onto the next one.

However, there are several problems associated with this simple version of gradient descent (see for e.g. [7] and[8]). We now discuss some variants of gradient descent that try to address these challenges.

2.4.2 Momentum

At time step t, momentum [9] first computes:

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \alpha \nabla_{\boldsymbol{\theta}_t} O(\boldsymbol{\theta}_t) \tag{2.15}$$

where γ is a (scalar) hyperparameter and then performs the update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t \tag{2.16}$$

2.4.3 Adagrad

Adagrad [10] uses the following update rule:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{\sqrt{G + \epsilon}} \odot \nabla_{\boldsymbol{\theta}} O(\boldsymbol{\theta}) \tag{2.17}$$

where ϵ is a smoothing constant (usually of the order of 10^{-8}) and $\boldsymbol{G} \in \mathbb{R}^{d \times d}$ is a diagonal matrix whose each entry i, i is the sum of the squares of the gradients of θ_i up till this point. Here, θ_i denotes the ith element of $\boldsymbol{\theta}$ and not its value at the ith time step (as is used elsewhere).

2.4.4 Adam

At time step t, Adam [11] first computes:

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} O(\boldsymbol{\theta}_t) \tag{2.18}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \left(\nabla_{\boldsymbol{\theta}} O(\boldsymbol{\theta}_t)\right)^2 \tag{2.19}$$

where β_1 and β_2 are hyperparameters and then applies bias correction:

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t} \tag{2.20}$$

$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t} \tag{2.21}$$

and finally performs the following update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \hat{\boldsymbol{m}}_t \tag{2.22}$$

Chapter 3

Artificial Neural Networks

Artificial neural networks are a class of algorithms in machine learning that are inspired by the structure and function of the human brain. In this chapter, we present an overview of some of these algorithms.

In the following discussion we shall assume a supervised learning setting where we define $S = \{(x^{(i)}, y^{(i)}); i = 1, ..., M\}$ to be a set of inputs x and corresponding labels y. When x and y are multi-valued we shall denote their jth value with the subscript j. We shall also assume a probabilistic setting, where we output a probability for each of the $|V|$ classes in our vocabulary.

3.1 A Single Neuron

Figure 3.1 shows a single neuron which is the basic building block of a neural network. The scalars w and b are known as the *weight* and *bias* of the neuron. The non-linear

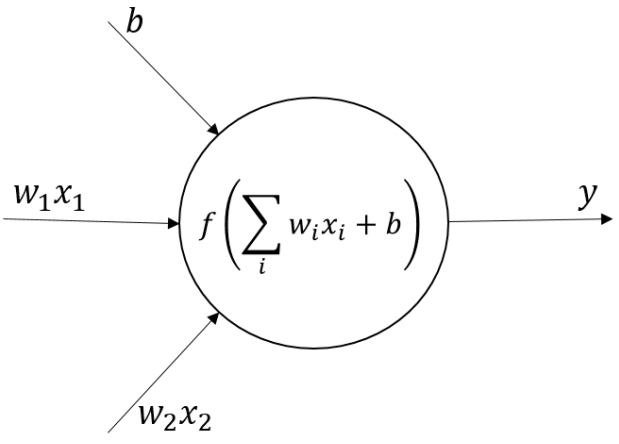


FIGURE 3.1: A Single Neuron

function f is called as the activation function. Table 3.1 summarizes some of the most common activation functions (see for e.g. [12] for a more detailed discussion on different activation functions and their advantages and disadvantages).

Sigmoid	$1/(1+\exp(-x))$
tanh	$\tanh(x)$
Rectified Linear Unit (ReLU)	$\max(0, x)$
Leaky ReLU	$\max(0.01x, x)$

TABLE 3.1: Some Common Activation Functions

3.2 Stacking Neurons

Neurons can be stacked together to form a single layer, known as a *feed-forward* or a *fully-connected* layer. A typical neural network has many of these layers. Figure 3.2 shows a two-layer neural network (the input layer is usually not counted as a layer).

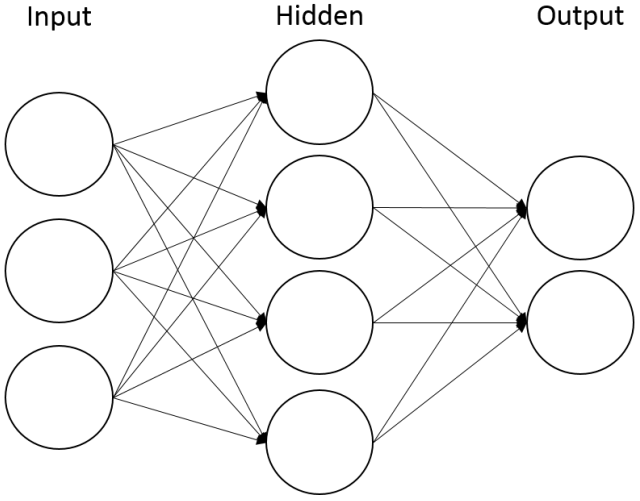


FIGURE 3.2: A Two-Layer Neural Network

The output layer usually does not have an activation function associated with it. This is because the outputs of this layer are usually interpreted as *class scores* (where the o_j neuron outputs a score/probability for class j).

We may represent a neural network using vectors, matrices and tensors. This will help keep the notation clean for larger neural networks. Specifically, let $\boldsymbol{x}$ denote the input vector $[x_1, ..., x_N]^T$ and $\boldsymbol{o}$ the output vector $[o_1, ..., o_{|V|}]^T$. Let $\boldsymbol{W}$ be the weight matrix of some layer whose ij entry denotes the weight from the input neuron i to the layer's jth neuron. Finally, define $\boldsymbol{b}$ to be the bias vector of a layer whose ith entry denotes the bias for the layer's ith neuron. Using this notation, we may express the neural network in Figure 3.2 as:

$$\boldsymbol{o} = \boldsymbol{W}_o^T f(\boldsymbol{W}_h^T \boldsymbol{x} + \boldsymbol{b}_h) + \boldsymbol{b}_o \tag{3.1}$$

where f is the activation function for the hidden layer and the subscripts h and o denote the hidden and output layers respectively.

3.3 Training Neural Networks

Let O denote the objective function that we are trying to optimize for a neural network. Training a network involves two processes: a forward pass and a backward pass.

In the forward pass, we feed the network some input and calculate its output. Next, the objective function takes in this output and produces a scalar, known as the *loss* of the network. The weights are usually initialized randomly.

For the backward pass, we calculate the derivative $\frac{\partial O}{\partial o_j}$. This is the derivative of the objective function with respect to the output of jth neuron in the output layer. We then use the chain rule to calculate the derivative of the objective function with respect to the other weights of the network. We shall use the following notation from now on:

$$\delta_j := \frac{\partial O}{\partial w_j} \tag{3.2}$$

where j is some unit in the network. δ is often referred to as the error signal received by the network. For some weight w_{ij} in the last hidden layer we have:

$$\delta_{w_{ij}} = \sum_{k=1}^{|V|} \frac{\partial O}{\partial o_k} \frac{\partial o_k}{\partial w_{ij}} \tag{3.3}$$

The backward pass is also known as backpropagation.

Usually, some form of minibatch gradient descent is used for training neural networks. We define an *iteration* as the process of feeding one minibatch to the network and using it to perform an update on the network's parameters. Furthermore, we define an *epoch* as the process of feeding all minibatches in the training data to the network one-by-one. Therefore, one epoch is comprised of many iterations. Training usually lasts several epochs. The network is periodically evaluated on a validation set after a certain number of epochs.

3.4 Recurrent Neural Networks (RNNs)

A recurrent neural network is a type of neural network that uses cyclical connections to maintain a so-called *hidden state*. At each time step, the RNN is fed in some input. The hidden state allows the RNN to condition the output for this input on all previous inputs that it has seen. Figure 3.3 shows an RNN.

Assume that each $\boldsymbol{x}_j \in \mathbb{R}^D$. At each time step t, the output (which is also the hidden state) of an RNN is given by:

$$\boldsymbol{h}_t = f\left(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{b}\right) \tag{3.4}$$

where $\boldsymbol{W}_{hh} \in \mathbb{R}^{d\times d}$, $\boldsymbol{W}_{hx} \in \mathbb{R}^{d\times D}$ and $\boldsymbol{b} \in \mathbb{R}^d$ are the parameters of the RNN and f is a non-linear activation function. Here, d is the desired dimensionality of the output

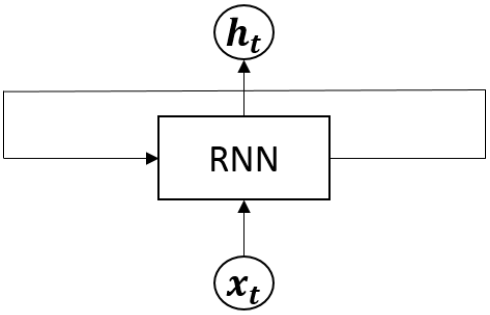


FIGURE 3.3: A Recurrent Neural Network

($\boldsymbol{h}_t \in \mathbb{R}^d$). Sometimes d is also referred to as the number of units of the RNN.

Figure 3.4 *unrolls* an RNN cell over different time steps (here $T = N$).

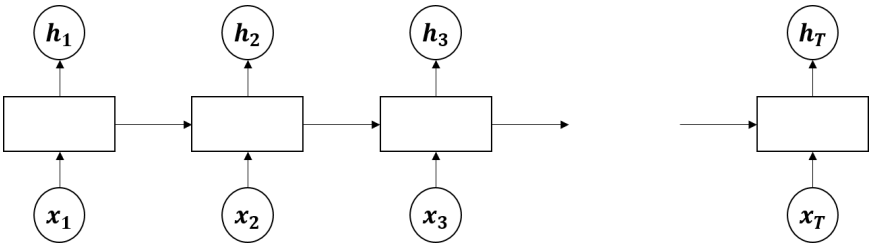


FIGURE 3.4: The Recurrent Neural Network Unrolled

The output of the RNN is sometimes written as $\boldsymbol{H} = [\boldsymbol{h}_1, ..., \boldsymbol{h}_T]$. Note that $\boldsymbol{H} \in \mathbb{R}^{T \times d}$.

The gradient of the objective function O with respect to $\boldsymbol{W}$ is given by:

$$\frac{\partial O}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial O_t}{\partial \boldsymbol{W}} \tag{3.5}$$

where $O_t = O[f_t(h_t)]$ for some appropriate function f_t and:

$$\frac{\partial O_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial O_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}} \tag{3.6}$$

and:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}} \tag{3.7}$$

Note that we have assumed that $O = \sum_t O_t$.

However, there are two major problems with RNNs that prevent it from learning long-term dependencies: (i) information morphing, and (ii) vanishing and exploding gradients.

Information morphing refers to the fact that an RNN will always morph (change) state even in the absence of external inputs. Let $g(\boldsymbol{h})$ denote an RNN as a function of its hidden state only. For $g(\boldsymbol{h}) = \boldsymbol{h}$ to be true, g must be an identity function. However, an identity function is linear in nature whereas RNNs are non-linear functions. This means that an RNN will always morph state.

Vanishing and exploding gradients refer to fact that for long sequences the RHS of Equation 3.7 either approaches 0 or ∞. See [13] for a detailed proof of this.

In order to deal with the problem RNNs have in learning long-term dependencies, [14] propose the Long-Short Term Memory (LSTM). The LSTM maintains two states at each time step: a hidden state $\boldsymbol{h}_t$ and a cell state $\boldsymbol{c}_t$. There are many variants of the LSTM. We reproduce the equations of one such variant [15]:

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ci}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i\right) \tag{3.8}$$

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{cf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f\right) \tag{3.9}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t\boldsymbol{c}_{t-1} + \boldsymbol{i}_t \tanh\left(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c\right) \tag{3.10}$$

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{co}\boldsymbol{c}_t + \boldsymbol{b}_o\right) \tag{3.11}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \tanh(\boldsymbol{c}_t) \tag{3.12}$$

where σ is the logistic sigmoid function and $\boldsymbol{i}_t$, $\boldsymbol{f}_t$ and $\boldsymbol{o}_t$ are the input, forget and output gates respectively.

The output of a RNN is only conditioned on the previous inputs. In order to condition it on future inputs also, a bidirectional RNN (BRNN) [16] is often used. Specifically, two RNNs are employed. Each begins traversing the sequence from a different end. Denote the RNN function with f. Then, the output $\boldsymbol{H}$ is found by concatenating $\overrightarrow{\boldsymbol{H}}$ and $\overleftarrow{\boldsymbol{H}}$, where:

$$\overrightarrow{\boldsymbol{H}} = f_{forward}([x_1, x_2, ..., x_T]) \tag{3.13}$$

$$\overleftarrow{\boldsymbol{H}} = f_{backward}([x_T, x_{T-1}, ..., x_1]) \tag{3.14}$$

Bidirectional LSTMs (BLSTMs) simply replace the RNN cell in BRNNs with a LSTM cell [17]. Similarly, mutli-dimensional RNNs further extend the concept of BRNNs by traversing the input from multiple directions [18].

3.5 Convolutional Layers

Let $\boldsymbol{x} \in \mathbb{R}^{H \times W \times C}$ be the input to a convolutional layer. When the inputs are images, H and W denote their height and width respectively and C denotes the number of color channels (usually 3, corresponding to red, green and blue). Let $\boldsymbol{F} \in \mathbb{R}^{H' \times W'}$ denote a filter (also called as a kernel) of height H' and width W'. The convolution layer applies the following operation on each color channel of the input separately (the kth channel

being denoted with x_k):

$$\boldsymbol{y}_k(i,j) = (\boldsymbol{x}_k * \boldsymbol{F})(i,j) = \sum_{n_1}\sum_{n_2} \boldsymbol{x}[i-n_1, j-n_2]\boldsymbol{F}[n_1,n_2] \tag{3.15}$$

The final output is given by:

$$\boldsymbol{y} = \sum_k \boldsymbol{y}_k \tag{3.16}$$

Note that $\boldsymbol{y}$ is a rank 2 tensor. Using more than one filter at each layer gives several of these rank 2 tensors, which can then be stacked on top of one another. This produces a rank 3 tensor. See Chapter 9 of [19] for a more detailed explanation on convolutional layers.

3.6 Pooling Layers

Pooling layers are used to reduce the dimensionality of inputs. They are many variants of pooling. One of the most common type is the *max* pooling which divides inputs into different (possibly overlapping) blocks and then simply replaces each block with the highest number in that block. Chapter 9 of [19] and the references therein provide a more detailed explanation of pooling.

3.7 Techniques Used for Training Neural Networks

Training neural networks within reasonable time is usually tricky. One of the reasons for this is *overfitting*. Consider Figure 3.5. Note that initially both the training and validation losses decrease. However, after a certain point (specifically, to the right of the dashed line), the training loss continues to decrease while the validation loss starts to increase. This phenomenon is known as overfitting. Overfitting indicates a high variance in the network (see section 14.6 of [20] for a more detailed discussion on this).

In this section, we outline some of the techniques used to combat overfitting. We also discuss some other techniques that are used to help networks converge faster.

3.7.1 L1 and L2 Regularizations

L1 regularization adds:

$$\lambda \sum_i ||\boldsymbol{W}_i|| \tag{3.17}$$

and L2 regularization adds:

$$\lambda \sum_i ||\boldsymbol{W}_i||^2 \tag{3.18}$$

to the objective function. Here, the $\boldsymbol{W}$'s are the parameters (usually excluding the biases) of the network and λ control how much of regularization to use (see Chapter 7.1 of [19] for more details).

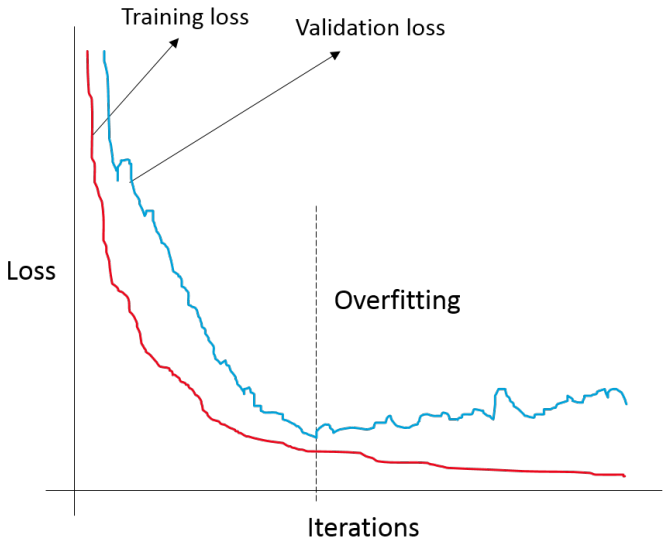


FIGURE 3.5: Demonstrating Overfitting

3.7.2 Dropout

For any given layer in the network, dropout [21] randomly sets some of its inputs to zero with some probability p. This adds some noise to the network and helps prevent overfitting. Note that dropout is only applied during training and disabled during evaluation and inference.

3.7.3 Early Stopping

Suppose that a network is evaluated on some validation set after each epoch. We may terminate training if there is no further improvement on the validation set for a certain number of epochs. This is known as early stopping.

3.7.4 Restoring the Best Model

Suppose that a network gives a certain accuracy on the validation set at the end of some epoch. Now suppose that the difference between this accuracy and the accuracy at the end of the previous epoch is greater than some number. We may then discard this epoch completely and restart training from the last epoch (i.e restore the last best model). This has the intuition of *forcing* the network to take the best path towards convergence.

3.7.5 Decaying the Learning Rate

This technique decays the learning rate as training progresses. This has the intuition of making the network take smaller steps as it approaches the minimum point, thus reducing the probability that the network *steps over* it.

3.7.6 Bucketing

Bucketing deals with the problem of variable length sequences in the input. Usually, because of programming constraints the inputs in a particular minibatch need to be of the same length. One way of solving this problem is to pad all sequences to the maximum sequence length in the input data. However, this results in increased training times. A better approach is to divide the input data into a discrete number of buckets. Inputs that are closer to each other in terms of their sequence lengths are grouped together in one bucket. Each input is then only padded to the maximum sequence length in the bucket they belong too. Note that this means that all inputs in a minibatch belong to the same bucket. Bucketing allows minibatches with smaller sequence lengths to be processed faster.

3.7.7 Xavier Weight Initialization

The weights of a network are usually initialized randomly. In practice, the Xavier initialization [22] is known to result in faster convergences. For weights in the ith layer the Xavier initialization uses a normal distribution with zero mean and the following variance:

$$\text{Var} = \frac{2}{n_i + n_{i+1}} \tag{3.19}$$

where n_i and n_{i+1} are the number of input and output neurons in the ith layer.

3.7.8 Batch Normalization

For a layer with H-dimensional output $\boldsymbol{o} = [o_1, o_2, ..., o_H]$, batch normalization [23] computes:

$$\hat{o}_k = \frac{o_k - \mathbb{E}[o_k]}{\sqrt{\text{Var}[o_k]}} \tag{3.20}$$

and then finds:

$$z_k = \gamma_k \hat{o}_k + \beta_k \tag{3.21}$$

for each k. Here, γ_k and β_k are parameters that are to be jointly learned with the rest of the model, During training $\mathbb{E}[i_k]$ and $\text{Var}[i_k]$ are computed using the minibatch at that iteration only. During evaluation and inference, these values are calculated using the entire training set. While [23] assumes a setting where batch normalization is applied at the beginning of the layer (to its input) we shall assume that batch normalization is applied at the end of the layer (as presented above). This will fit in better with the discussions hereafter (including that in the next subsection).

3.7.9 Layer Normalization

For a layer with H-dimensional outputs $\boldsymbol{o} = [o_1, o_2, ..., o_H]$, layer normalization [24] first computes:

$$\hat{o}_k = \frac{o_k - \mu}{\sigma} \tag{3.22}$$

where:

$$\mu = \frac{1}{H}\sum_{j=1}^{H} o_j \tag{3.23}$$

$$\sigma = \sqrt{\frac{1}{H}\sum_{j=1}^{H}(o_j - \mu)^2} \tag{3.24}$$

and then applies Equation 3.21. Note that unlike batch normalization, layer normalization does not depend upon the minibatch size. Also, layer normalization does not distinguish between the training and the evaluation and inference processes.

For recurrent neural networks, for each time step t we first compute:

$$\boldsymbol{a}_t = \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{b} \tag{3.25}$$

as in Equation 3.4, then calculate:

$$\mu_t = \frac{1}{H}\sum_{i=0}^{H} a_{it} \tag{3.26}$$

$$\sigma_t = \sqrt{\frac{1}{H}\sum_{j=0}^{H}(a_{it} - \mu_t)^2} \tag{3.27}$$

where a_{it} denotes the ith element of $\boldsymbol{a}$ and H is the number of units of the RNN and finally find the next hidden state:

$$\boldsymbol{h}_t = f\left[\frac{\boldsymbol{g}}{\sigma_t} \circ (\boldsymbol{a}_t - \mu_t) + \boldsymbol{b}\right] \tag{3.28}$$

where $\boldsymbol{g}$ and $\boldsymbol{b}$ are defined to be of the same dimension as $\boldsymbol{h}_t$, $\circ$ denotes the element-wise vector product (also known as the Hadamard product) and f is the RNN's non-linear activation function.

3.7.10 Residual Connections

Let $g(\boldsymbol{x})$ denote the output of some layer g in a network. Residual connections [25] directly connect the input of the layer g with its output. The output of a layer with residual connections is given by:

$$\boldsymbol{z} = g(\boldsymbol{x}) + \boldsymbol{x} \tag{3.29}$$

When $g(\boldsymbol{x})$ and $\boldsymbol{x}$ are of different dimensions, an appropriate affine transformation on x may be used:

$$z = g(\boldsymbol{x}) + \boldsymbol{W}_r\boldsymbol{x} \tag{3.30}$$

where $\boldsymbol{W}_r$ is to be jointly learned with the rest of the network. Residual connections allow very deep networks to be trained efficiently by countering the so-called degradation problem (see for e.g. [25]).

3.7.11 Clipping Gradients

In Section 3.4, we discussed the exploding gradients problem. One way to prevent gradients from exploding is to scale them down whenever a gradient norm exceeds a certain threshold [13].

Chapter 4

Encoder-Decoder Architectures

Figure 4.1 shows the concept behind the Encoder-Decoder architecture. The encoder takes in an input and *encodes* it in some way. The decoder takes this new (more useful) representation of the input and produces some output. This output is fed into some objective function. Gradients of the output of the objective function with respect to the parameters of both the encoder and decoder are calculated and used to update them using some update rule (such as those outlined in Section 2.4).

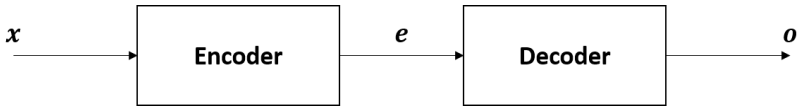


FIGURE 4.1: The Encoder-Decoder Architecture

We shall specifically discuss the encoder-decoder architecture in reference to the problem of generating a transcription corresponding to the image of some handwritten text. Let $\boldsymbol{x} \in \mathbb{R}^{H \times W}$ denote the input image matrix of height H and width W. We shall denote the jth column in $\boldsymbol{x}$ with $\boldsymbol{x}_j$. Note that each $\boldsymbol{x}_j \in \mathbb{R}^H$. Also, let $\boldsymbol{y}$ denote a vector whose jth element denotes the jth character in the transcription.

4.1 Implementation Using Recurrent Neural Networks

Figure 4.2 shows an encoder decoder network that uses recurrent neural networks (RNNs) [26].

The encoder and the decoder are two separate RNNs, or its variants such as the LSTM. The final hidden state of the encoder RNN encodes the entire image in a single vector. The initial hidden state of the decoder RNN is initialized with this vector. The decoder is then required to predict the tth output in the transcription at time step t.

At each time step t during training, the decoder is fed the $(t-1)$th character of the transcription and asked to predict the next character. Usually, the first character in the transcription is a special start token that acts as a starting signal to the decoder.

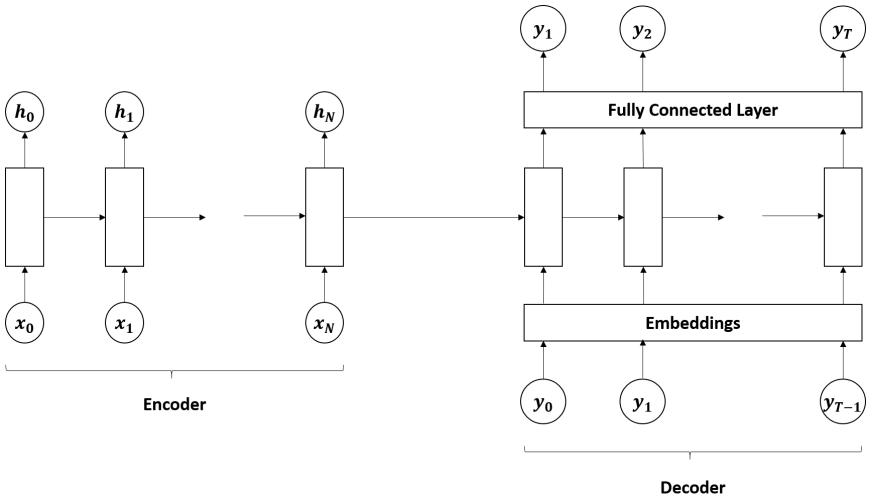


FIGURE 4.2: Implementation of the Encoder-Decoder Architecture Using Recurrent Neural Networks

Another special token used is the end token. Once the last character is predicted, the decoder is required to predict this end token at the next time step, indicating that its done with this input. This technique of feeding the previous correct output at each time step is known as *teacher forcing*.

Each character in the vocabulary is assigned a vector. All of these vectors are concatenated to produce the so-called *embedding matrix*. Note that it is this embedding vector that is fed into the decoder at each time step. The embedding matrix is jointly learned with the rest of the network's parameters.

During inference, we do not have access to the true transcription. Hence, teacher forcing cannot be used. So instead, at each time step of the decoder, the character with the highest probability is selected and fed to the decoder at the next time step.[1] Note however that this creates a distributional mismatch. During training the decoder always had access to the true character at the previous time step. However, this is not the case during inference. Hence, during inference the decoder might find itself in an unknown state space (such as a combination of the input image and a sequence of previous [incorrect] predictions; something it has never seen during training). Such a distribution mismatch can lead to an accumulation of errors over time, because each subsequent output is conditioned on preceding outputs.

Several techniques have been proposed to solve this distributional mismatch problem. One obvious way of eliminating this problem is to not use teacher forcing at all. However,

[1] An alternative approach is this: at the first time step select the top k characters with the highest probability; feed each of these characters to the next time step individually; extend each of these k characters with all characters in the vocabulary and of all the resulting $k|V|$ sequences select the top k ones with the highest probability. Repeat this process with all subsequent time steps. The probability of a sequence can be calculated using Equation 2.12. This decoding strategy is known as beam search.

this, in practice, leads to poor results (see for example the results in [27]).

[27] instead proposes a technique called scheduled sampling in which the network randomly decides with probability ϵ at each time step during training whether to use teacher forcing or to sample a character from the probability distribution at the output of the previous time step. [27] proposes to decrease ϵ from 1 to 0 in some fashion. Sometimes ϵ is just set to some constant value less than 1. While scheduled sampling works well in practice, [28] shows that its underlying objective function leads to an inconsistent objective function.

[29] introduces the Dataset Aggregation (DAGGER) method to solve the distributional mismatch problem which repeatedly updates the training set to include the new distributions (sequences) encountered during validation. [30] introduces yet another solution called professor forcing that trains two networks — one with teacher forcing and the other without it — in an adversarial fashion.

4.2 Adding an Attention Mechanism

One obvious flaw with encoder-decoder networks is that the encoder has to encode the entire input sequence into only one vector. For longer input sequences, it might not be possible to create a useful representation using a single vector only.

One way to solve this problem is to increase the size of this vector. However, this would only lead to bigger networks and increased training times.

Another method to solve this problem is to recognize the fact that for each prediction, the decoder needs access to only a small subset of the input sequence. For example, in order to predict the first character in a handwriting recognition task, the decoder only needs to look at the first few columns in the image. Attention-based models build upon this insight by adding another neural network to the existing architecture, that at each time step decides which part of the image to pay attention to.

4.2.1 Global Attention

[31] proposes a soft-alignment method. Let $\boldsymbol{H} \in \mathbb{R}^{W \times D}$ denote the concatenation $[\boldsymbol{h}_0, \boldsymbol{h}_1, ..., \boldsymbol{h}_W]$ where $\boldsymbol{h}_j$ is the encoder output for $\boldsymbol{x}_j$ (see for example 4.2). Here, D denotes the size of the encoder (the number of units in an RNN). For each step t of the decoder, we calculate a context vector:

$$\boldsymbol{c}_t = \sum_{j=1}^{W} \alpha_{jt} \boldsymbol{h}_j \tag{4.1}$$

Each scalar weight α_{jt} (collectively known as the alignments) is computed by:

$$\alpha_{jt} = \frac{\exp(e_{jt})}{\sum_{k=1}^{W} e_{kt}} \tag{4.2}$$

where:

$$e_{jt} = a(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j) \tag{4.3}$$

where $\boldsymbol{s}_{t-1}$ is the previous hidden state of the decoder and a is an alignment model that is jointly learned with the rest of the network. a can simply be a feed forward neural network (as in [31]) given by:

$$a = \boldsymbol{V}_a^T f_a(\boldsymbol{W}_a^T[\boldsymbol{s}_{t-1}; \boldsymbol{h}_j] + \boldsymbol{b}_a) \tag{4.4}$$

where $\boldsymbol{V}_a$, $\boldsymbol{W}_a$ and $\boldsymbol{b}_a$ are the weights and biases of the alignment model, f is some non-liner function and $[\boldsymbol{s}_{t-1}; \boldsymbol{h}_j]$ represents a concatenation of $\boldsymbol{s}_{t-1}$ and $\boldsymbol{h}_j$. The context vector is then fed to the decoder along with the previous outputs.

Each α_{jt} indicates the relevance of (the encoding of) the jth column of the image to the prediction that the decoder has to make at time step t.

Note that the alignments at each step t are calculated *after* the decoder makes the predictions for that step. Therefore, for the first prediction the decoder needs to rely on the final hidden state of the encoder (with which its own hidden state is initialized).

This attention mechanism is often called as the Bahdanau attention-mechanism.

[32] proposes a slightly alternative approach. In this approach, the decoder is only fed its previous output. The output of the decoder at each time step is fed to the alignment model. The alignment model uses this decoder output and the encoder outputs to calculate a context vector using Equations 4.1 to 4.3 (except that instead of $\boldsymbol{s}_{t-1}$ we use $\boldsymbol{s}_t$). This context vector and the decoder output at that time step are fed to a feed forward layer that makes the final decision. Additionally, [32] experiments with two different alignment models also that are reproduced below:

$$a = \boldsymbol{s}_t^T \boldsymbol{h}_j \tag{4.5}$$

$$a = \boldsymbol{s}_t^T \boldsymbol{W}_a \boldsymbol{h}_j \tag{4.6}$$

4.2.2 Local Attention

Global attention requires the alignment model to process all of the encoder outputs at any given time step and assign weights to each of them. In contrast, local attention only selects a subset of the encoder outputs that fall within the *window* $[p_t - D, p_t + D]$ at any time step and uses them to calculate the context vector. In [32] D is determined empirically, while two different approaches are used to find p_t. The monotonic alignment methods simply sets p_t to t (i.e. the current time step of the decoder) where as the predictive alignment methods calculates p_t using:

$$p_t = W \text{sigmoid}(\boldsymbol{v}_p^T \tanh(\boldsymbol{W}_p \boldsymbol{s}_t)) \tag{4.7}$$

where $\boldsymbol{v}_p$ and $\boldsymbol{W}_p$ are parameters to be learned and W is the width of the image. In order to favor positions near p_t each alignment weight α_{jt} is multiplied by:

$$\exp\left(-\frac{(j-p_t)^2}{2\sigma^2}\right) \tag{4.8}$$

where σ is set to $D/2$.

Chapter 5

N-Gram Language Models

Language models can help increase the accuracy of systems that deal with language (such as handwritten text recognition systems). The following discussion is mainly drawn from Chapter 4 of [33].

5.1 The N-Gram Language Model

Assume a large text corpus consisting of V distinct words. Denote a particular sequence of words $[w_1, ..., w_L]$ with w_1^L. Using the chain rule, we have:

$$P(w_1^L) = \prod_{k=1}^{L} P(w_k | w_1^{k-1}) \tag{5.1}$$

n-gram models estimate the conditional probabilities on the RHS in the equation above. Specifically:

$$P(w_k | w_1^{k-1}) \approx P(w_k | w_{k-n+1}^{k-1}) \tag{5.2}$$

Equation 5.2 essentially approximates the probability of a word appearing in a text by only taking into account the last $n-1$ words. Intuitively, this makes sense for language models. Words appearing at the beginning of a paragraph hardly ever influence the occurrence of words at the end of the paragraph. We may estimate these n-gram conditional probabilities by using:

$$P(w_k | w_{k-n+1}^{k-1}) = \frac{N(w_{k-n+1}^{k})}{N(w_{k-n+1}^{k-1})} \tag{5.3}$$

where $N(\circ)$ is the number of times $\circ$ appears in the text corpus. $N(\circ)$ is often referred to as the counts of $\circ$.

5.2 Perplexity

In order to check how accurate the probabilities calculated in the previous section are, a metric known as perplexity is often used. The perplexity is always calculated on a

separate test set (that was not used for finding the n-gram probabilities). Let $W^{'} = [w_1^{'}, ..., w_{L'}^{'}]$ denote the test set. The perplexity is given as:

$$\text{Perplexity}(W^{'}) = P(w_1^{'}, ..., w_{L'}^{'})^{-\frac{1}{L'}} \tag{5.4}$$

where the RHS can be evaluated using Equation 5.1.

5.3 Smoothing

The naïve n-gram model possesses a serious flaw. The test set might contain a valid sequence of words that was not present in training corpus — the corpus that was used to calculate the n-gram probabilities. In such a case, the n-gram model will assign a zero probability to this sequence of words resulting in the entire probability of the test set being 0 (as per Equation 5.1). This would mean that we can not calculate the perplexity on test set (as we can not divide by 0). One way of solving this problem is to use smoothing. Smoothing removes some probability mass from more frequent sequences and assigns it sequences that were not present in the training set. We present some smoothing algorithms below (see [34] for a more detailed analysis including some more algorithms).

5.3.1 Laplace (Add-One) Smoothing

Laplace smoothing adds 1 to all counts:

$$P(w_k|w_{k-n+1}^{k-1}) = \frac{N(w_{k-n+1}^{k}) + 1}{N(w_{k-n+1}^{k-1}) + V} \tag{5.5}$$

Note that adding V to the denominator is necessary to ensure that the probabilities sum up to 1.

5.3.2 Backoff and Interpolation

Backoff and interpolation use the idea that if a particular n-gram is absent from the training corpus, we can approximate its probability from a lower order n-gram. In backoff, we only 'back-off' to a lower order estimate in case the higher order n-gram is missing. In interpolation, we combine all n-gram estimates in some fashion. Note that in either case we have to ensure that all probabilities sum up to 1.

5.3.3 Kneser-Ney Smoothing

[35] proposes the so-called Kneser-Ney smoothing algorithm. The original algorithm was backoff which [34] later modified to be interpolated. We first present the interpolated method. For a bigram model define:

$$P_{\text{continuation}}(w_i) = \frac{|\{w_{i-1} : N(w_{i-1}w_i) > 0\}|}{|\{(w_{j-1}w_j) : N(w_{j-1}w_j) > 0\}|} \tag{5.6}$$

i.e. $P_{\text{continuation}}(w_i)$ is the probability that w_i appears after a new word (or a new sequence of $n-1$ words in the general n-gram case). The Interpolated Kneser-Ney smoothing equation is:

$$P(w_i|w_{i-1}) = \frac{\max(N(w_{i-1}w_i) - d, 0)}{N(w_{i-1})} + \lambda(w_{i-1})P_{\text{continuation}}(w_i) \tag{5.7}$$

The first term above is the same as in Equation 5.3 except that we *discount* a constant d from the numerator. The Kneser-Ney smoothing then reassigns this *probability mass* via the second term. Subtracting a constant d from all probabilities has the intuition that it doesn't affect n-grams for which we already have a high probability too much. For n-grams with low probability (i.e. lesser counts) we already have a less reliable estimate anyways and as such making those probabilities smaller should not have that much of an adverse effect.

In the second term, $\lambda(w_{i-1})$ is defined as:

$$\lambda(w_{i-1}) = \frac{d \times |\{w : N(w_{i-1}w_i) > 0\}|}{N(w_{i-1})} \tag{5.8}$$

Note that the numerator in the equation above is the total probability mass subtracted. Multiplying this with $P_{\text{continuation}}(w_i)$ essentially assigns w_i an additional probability mass proportional to the number of new *contexts* it appears in. Finally, the denominator $N(w_{i-1})$ ensures that $\sum_j P(w_j|w_{i-1}) = 1$.

The general recursive equation for the Interpolated Kneser-Ney technique is:

$$P(w_i|w_{i-n+1}^{i-1}) = \frac{\max[N(w_{i-n+1}^{i}) - d, 0]}{N(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1})P(w_i|w_{i-n+2}^{i-1}) \tag{5.9}$$

The original backoff algorithm is given by:

$$P(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \frac{\max(N(w_{i-n+1}^{i})-d,0)}{N(w_{i-n+1}^{i-1})} & \text{if } N(w_{i-n+1}^{i}) > 0 \\ \lambda(w_{i-n+1}^{i-1})P(w_i|w_{i-n+2}^{i-1}) & \text{if } N(w_{i-n+1}^{i}) = 0 \end{cases} \tag{5.10}$$

Chapter 6

Optical Character Recognition Systems

Optical Character Recognition (OCR) typically involves five steps: preprocessing, segmentation, feature extraction, classification and recognition and post-processing. In this chapter, we present a brief review of the OCR literature.

6.1 Preprocessing

Preprocessing involves a number of steps [36] that help improve the accuracy of later stages by removing noise and unnecessary details from an image.

Binarization is the process of converting a colored or gray-scale image to a binary image. In a binary image, all pixels can only take on two values: 0 or 1. One way of doing this is through Otsu's method [37] where we assume that an image contains two classes of pixels and then calculate the optimum threshold that separates them.

Sometimes scanning introduces a *skew* in images that needs to be corrected. Several different techniques exist for this purpose (see for e.g [36], [38]).

Other techniques involved in preprocessing include noise removal, background elimination, removal of black boundaries and extra white spaces, gray-scale normalization, size-normalization, smoothing and thinning (see for e.g. [36] and [39]).

6.2 Segmentation

Instead of feeding an image of an entire page of handwritten text to some classifier, it is usually useful to segment it into pieces first. These pieces could be individual lines, words or ligatures [36]. One way of doing this is through horizontal projection. Pixel values in each row are summed up. Assuming that 0 corresponds to a white pixel, a row summing up to zero would indicate a white line. This information can be used to segment the image into individual lines. Similarly, vertical projection can be used to segment images of lines into individual words and/or ligatures. However, in the case

of images of handwritten text, this is generally harder (as lines/words/ligatures may overlap). [36] reviews different segmentation techniques.

6.3 Feature Extraction

Instead of feeding raw images (that might contain noise) to a classifier, one may first extract information that is relevant to the task-at-hand and only feed in that information. The goal of feature extraction is to extract this information from raw images.

Approaches to feature extraction include the computation of curvature, slope, end-points axes ratio, the length variations of strokes, shape context, discrete cosine transform and discrete wavelet transform and zoning features etc. (see for e.g. [40] and [36]).

However, more recent research does not extract features explicitly and instead feeds raw images to the classifier (see for e.g. [41] and [42]).

6.4 Recognition

Traditionally, Hidden Markov Models (HMM) were used for the recognition phase (see [43] for a review of some OCR systems that used HMM). However, recent research uses deep learning (neural nets) for this purpose.

Depending on the segmentation step, the classifier will either need to recognize images of either single characters ([44] and [45]) or complete lines ([38] and [46]) or entire paragraphs [41].

6.5 Post Processing

Once the text in an image has been recognized, additional steps such as spell-checking and grammar corrections can be carried out to improve the accuracy of the recognition system. This of course assumes that the text in the image is grammatically correct and contains valid words.

Chapter 7

Dataset Collection

The performance of a deep neural model is dependent upon the quality of the dataset it was trained on.

[38] introduces the Urdu Nastaleeq Handwritten Dataset (UNHD). While the UNHD dataset consists of $10,000$ text lines, only $4,240$ are publicly available which are not enough to train a robust deep neural network. As a result, we create a new dataset for the purpose of this thesis. The dataset will be available for further research.[1]

For this new dataset, $500,000$ text lines were selected from Urdu literature. $10,000$ lines were picked from these lines in such a way that the ratios of the frequencies of words remained the same. These lines (after some filtering) were divided into 490 pages, each consisting of 20 lines. Each page was given a unique 4-digit i.d. and was written by a distinct writer. Each writer too got a unique 4-digit i.d.

The writers ranged between 15 and 30 years of age, were of both sexes and mostly belonged to schools, colleges and universities. The writers were given pages with black lines drawn on them for writing. Red pens with 6 different stroke widths were used for writing. The writers were instructed to leave one blank line after every line. Writers usually took 1 to 3 lines to write each printed text line.

Each page was scanned using a flatbed scanner at 300 dots per inch (dpi) and saved using the *.jpg* format. Only the red pixels were extracted from each page. This removed the black lines in the background. The images were then segmented into text lines using horizontal projection. Each image was assigned a unique 10 digit i.d. of the format *aaaa_bbbb_cc*, where *aaaa* was the i.d. of the writer who wrote them, *bbbb* was the i.d. of the 20-line page that the writer wrote and *cc* was the line number of the writer's page.

The final dataset contains $15,164$ text lines written by 490 different writers in 6 different strokes and has $13,497$ trigrams, $1,674$ bigrams and 61 unigrams.

[1] Contact http://cle.org.pk/ for this dataset.

The dataset is further divided into training and test sets consisting of 13,351 and 1,813 images respectively. 440 writers contributed to the training set while 86 contributed to the test set. 288 images in the test set are of writers who also contributed to the training set.

Table 7.1 contains some statistics of the images in the dataset.

	Height			Width		
	Min	Max	Mean	Min	Max	Mean
Train	44	328	167.71	65	2509	1859.99
Test	72	291	166.86	104	2439	1818.30

TABLE 7.1: Statistics of Images in the Dataset

Appendix D shows how the dataset was prepared. Appendix C gives the frequencies of each character in the dataset according to its positions in its ligatures.

Chapter 8

Experiments

All experiments discussed below were carried out on the Urdu handwritten dataset from Chapter 7. Figure 8.1 shows the system during the training phase.

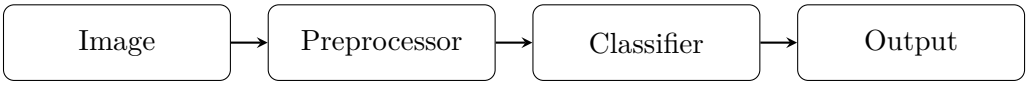


FIGURE 8.1: The System During the Training Process

In this chapter, we only present an outline of the processes in the preprocessor and classifier stages during training and evaluation. The inference process is discussed in Chapter 9. The exact details of the system can be found in the configuration files in Appendix A.

8.1 Generating Labels

8.1.1 Character-Based Approach

In Urdu, each character has a different shape based on their position in a ligature. In the character-based approach to generating labels, each shape is assigned a different i.d. (note that in this case this is more of a *shape*-based approach). Appendix C lists the shapes in the vocabulary.

8.1.2 Ligature-Based Approach

In this approach, we assign each distinct ligature an i.d. This has the advantage that it reduces the length of the output sequence. Also, because we are constraining ourselves to a certain set of *valid* ligatures, this approach can lead to a lower ligature error rate. However, there are several drawbacks to this approach. Firstly, the training data might not contain all possible valid ligatures. Secondly, most ligatures might only occur a few times, which may prevent the classifier from properly learning their shapes. Thirdly, during inference an image might contain an invalid ligature. However, we would still want our classifier to correctly predict this invalid ligature, which is not possible if we only restrict ourselves to a certain set of ligatures.

8.2 Accuracy Metric

Levenshtein (edit) distance is the total number of insertions, deletions and substitutions needed to make two strings equal [47]. We calculate the accuracy of the classifier on a single example using:

$$\text{Accuracy} = \frac{\text{Levenshtein}(\text{prediction}, \text{label})}{\text{Length of label}} \tag{8.1}$$

We average the accuracy over all examples.

8.3 Preprocessor

All images are trimmed to remove any white columns in them. All images are re-sized to a fixed height. Images are also divided into buckets. The width of all images in a bucket are padded upto the maximum width of that bucket. Furthermore, all images are binarized using Otsu's method.

8.4 Classifiers

We experiment with two different classifiers. Furthermore, we also incorporate an n-gram language model with the first classifier. In all experiments, we use a character-based classification approach. The learning rate is decayed exponentially. Table 8.1 gives the number of trainable parameters of each architecture.

CNN-RNN-CTC	$7,075,374$
Encoder-Decoder	$9,432,832$

TABLE 8.1: Number of Trainable Parameters

8.4.1 CNN-RNN-CTC Architecture

The preprocessed images are fed to CONV-POOL-ReLU blocks. Each of these blocks has a convolutional layer, a max-pooling operator and a ReLU activation function in that order. Some of these blocks use a batch normalization layer at the end. Residual connections may optionally be added. Appropriate affine transformations are used in that case. The final output of the CONV-POOL-ReLU block is fed to a recurrent neural network which is followed by a feed forward layer. The number of hidden units of the feed forward layer is the size of the vocabulary plus one. The extra unit is needed for the Connectionist Temporal Classification function, which is the objective function of this network. Note that as Urdu is read from right to left we flip all images before feeding them to the classifier.

8.4.2 Language Model

While the classifier is character-based, the language model we use is ligature-based i.e. it models the probability of a particular ligature following a specific sequence of ligatures. We use a trigram model with Backoff Kneser-Ney smoothing. We use the same prefix

beam search algorithm outlined in [5] except that instead of applying the language model whenever we encounter a space, we apply it whenever a ligature ends. Note that because of our character-based approach to generating labels, we can identify which ids signal the end of a ligature.

In the configuration file in Appendix A.1 'alpha' and 'beta' are the same as defined in [5]. The 'discard probability' is the minimum probability that a character must have at a given time step in order for it to be considered by the decoding algorithm.

8.4.3 Encoder-Decoder Architecture

In this architecture, we first feed the preprocessed images to several CONV-POOL-ReLU blocks as discussed in Section 8.4.1. We pass on the outputs of these blocks to an encoder-decoder network. We also incorporate the Bahdanau attention mechanism. In this case, we do not flip images. We leave it to the classifier to learn the direction of the script. For scheduled sampling, we use an exponential decaying function. The outputs of the cross entropy function for all decoder outputs are summed up to give the loss of the network.

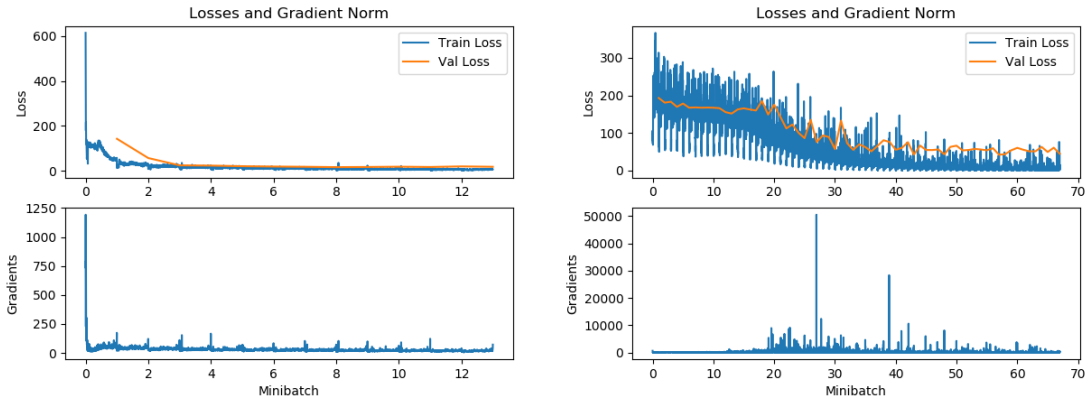
8.5 Results

Table 8.2 shows the results of the experiments.

	CNN-RNN-CTC	Encoder-Decoder
Greedy Search	88.50	89.52
Beam Search	88.75	**90.07**
Beam Search + Language Modeling	**91.51**	-
On English (IAM dataset) [46]	93.80	91.90

TABLE 8.2: Results

Figure D.1 shows the losses and gradient norms during training.



(A) CNN-RNN-CTC (B) Encoder-Decoder

FIGURE 8.2: Training Plots

Figure 8.3 shows the alignments assigned to an image overtime by the alignment model. Note that the classifier learns to *read* from right to left. Appendix B shows how the attention mechanism works on an image.

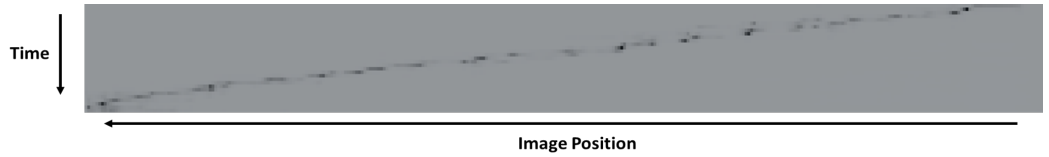


FIGURE 8.3: Visualization of the Attention Mechanism

We train the trigram language model on 10,000 Urdu text lines and achieve a perplexity of 47.621 on a held-out (test) set.

Table 8.4 shows the breakdown of the results on the test set.

Accuracy Bin	Number of Samples			Number of Distinct Writers		
	Greedy	Beam	Beam + LM	Greedy	Beam	Beam + LM
100	148	161	428	50	50	71
90	767	770	748	78	79	80
80	579	573	426	84	83	82
70	219	219	133	65	68	57
60	74	71	55	35	33	28
50	19	10	15	10	6	9
40	3	4	4	3	4	3
30	2	3	1	1	2	1
20	2	2	3	1	1	3
10	0	0	0	0	0	0
0	0	0	0	0	0	0

TABLE 8.3: The CNN-RNN-CTC Architecture on the Test Set

Accuracy Bin	Number of Samples		Number of Distinct Writers	
	Greedy	Beam	Greedy	Beam
100	261	278	60	62
90	721	758	79	82
80	516	507	84	84
70	175	161	66	63
60	81	69	42	37
50	35	25	20	16
40	6	2	6	2
30	11	8	7	4
20	3	3	3	3
10	2	0	2	0
0	2	2	2	2

TABLE 8.4: The Encoder-Decoder Architecture on the Test Set

Chapter 9

Deployment

Figure 9.1 shows the final system after the classifier has been trained.

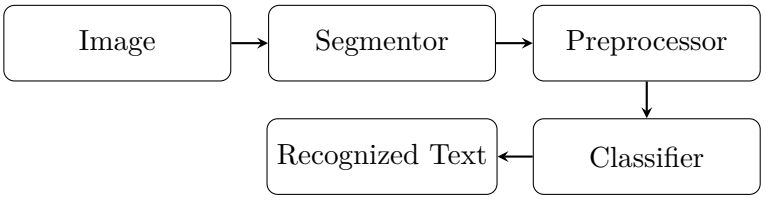


FIGURE 9.1: The Final System

Note that we add a segmentation step. For segmentation, we use a connected-component approach. We first smear the image horizontally. As a result, the characters in each line get physically connected. We then construct boundary boxes around each connected component. The coordinates of the edges of the boundary boxes are then used to segment the original image.

The rest of the stages are the same as discussed in previous chapters.

Chapter 10

Conclusion and Future Work

We presented a new dataset consisting of Urdu handwritten text lines and their corresponding ground truths. This dataset was used to train two deep learning based architectures. An n-gram language model was also incorporated. The results show that deep learning-based methods are highly effective in building optical character recognition systems for Urdu.

A number of improvements may be made in the future.

Firstly, the dataset may be expanded to include, for example, images of multi-lingual handwritten documents. In cases where some languages are written from right-to-left and others from left-to-right attention based models will prove to be particularly useful because of their flexibility in choosing which part of the image to consider at any given time.

Secondly, one major drawback of the architectures presented is that future decisions made by the network cannot affect its past decisions. This is the opposite of what the human mind does, especially in the case of reading.

Thirdly, future work may involve combining the predictions of different networks. This is because different networks (like human minds) usually learn different things. Some are good at identifying certain characters while others are good at identifying others. The main challenge in this regard is that the output sequences produced by these different networks may not be aligned.

Fourthly, some characters in the dataset have a very low frequency count. As such, the networks have very little time to properly learn to recognize them. In this regard, the objective function may be modified in such a way that the network is forced to pay more attention to the examples of these low frequent characters.

Appendix A

Configuration Files

A.1 CNN-RNN-CTC Architecture

```
"model": "CNN_RNN_CTC",

"char_or_lig": "char",

"save_best": "True",
"restore_best_model": "False",
"early_stopping": "False",

"batch_size": "32",
"buckets": "5",

"image_size": "None, 64",
"image_width_range": "None",
"flip_image": "True",

"optimizer": "adam",

"lr": "0.001",
"anneal_lr": "True",
"anneal_lr_every": "1000",
"anneal_lr_rate": "0.96",

"dropout": "0.2",
"max_grad_norm": "5",
"l2_regularizer_scale": "0.0",

"cnn_num_layers": "7",
"cnn_num_residual_layers": 0,
"cnn_activation": "leaky_relu",
"cnn_num_filters": "32, 64, 128, 128, 256, 256, 512",
"cnn_filter_sizes": "5, 5, 5, 5, 3, 3, 3",
"cnn_strides": "(1,1), (1,1), (1,1), (1,1), (1,1), (1,1), (1,1)",
"cnn_paddings": "SAME, SAME, SAME, SAME, SAME, SAME, SAME",
"pool_sizes": "(2,2), (1,2), (1,2), (1,2), (1,2), (1,2), (1,1)",
"pool_strides": "(2,2), (1,2), (1,2), (1,2), (1,2), (1,2), (1,1)",
"pool_paddings": "SAME, SAME, SAME, SAME, SAME, SAME, SAME",
"do_batch_norm": "False, False, True, False, False, False, True",
```

```
"rnn_num_layers": "2",
"rnn_unit_type": "lstm",
"rnn_type": "bi",
"rnn_num_residual_layers": "0",
"rnn_num_units": "512",
```

For beam search:

```
"beam_width": "10",
```

For beam search with language modeling:

```
"beam_width": "10",
"ngrams": "3",
"alpha": "0.5",
"beta": "4",
"discard_probability": 0.001
```

A.2 Encoder-Decoder Architecture

```
"model": "Encoder_Decoder",

"char_or_lig": "char",

"save_best": "True",
"restore_best_model": "False",
"early_stopping": "False",

"batch_size": "32",
"buckets": "5",

"image_size": "None, 64",
"image_width_range": "None",
"flip_image": "False",

"optimizer": "adam",

"lr": "0.001",
"anneal_lr": "True",
"anneal_lr_every": "4000",
"anneal_lr_rate": "0.96",

"dropout": "0.2",
"max_grad_norm": "5",
"l2_regularizer_scale": "0.0",

"cnn_num_layers": "7",
"cnn_num_residual_layers": "0",
"cnn_activation": "leaky_relu",
"cnn_num_filters": "16, 32, 64, 64, 128, 128, 128",
"cnn_filter_sizes": "5, 5, 5, 3, 3, 3, 2",
"cnn_strides": "(1,1), (1,1), (1,1), (1,1), (1,1), (1,1), (1,1)",
"cnn_paddings": "SAME, SAME, SAME, SAME, SAME, SAME, SAME",
"pool_sizes": "(2,2), (1,2), (1,2), (1,2), (1,2), (1,2), (1,1)",
"pool_strides": "(2,2), (1,2), (1,2), (1,2), (1,2), (1,2), (1,1)",
"pool_paddings": "SAME, SAME, SAME, SAME, SAME, SAME, SAME",
"do_batch_norm": "False, True, False, True, False, False, True",

"encoder_num_layers": "2",
```

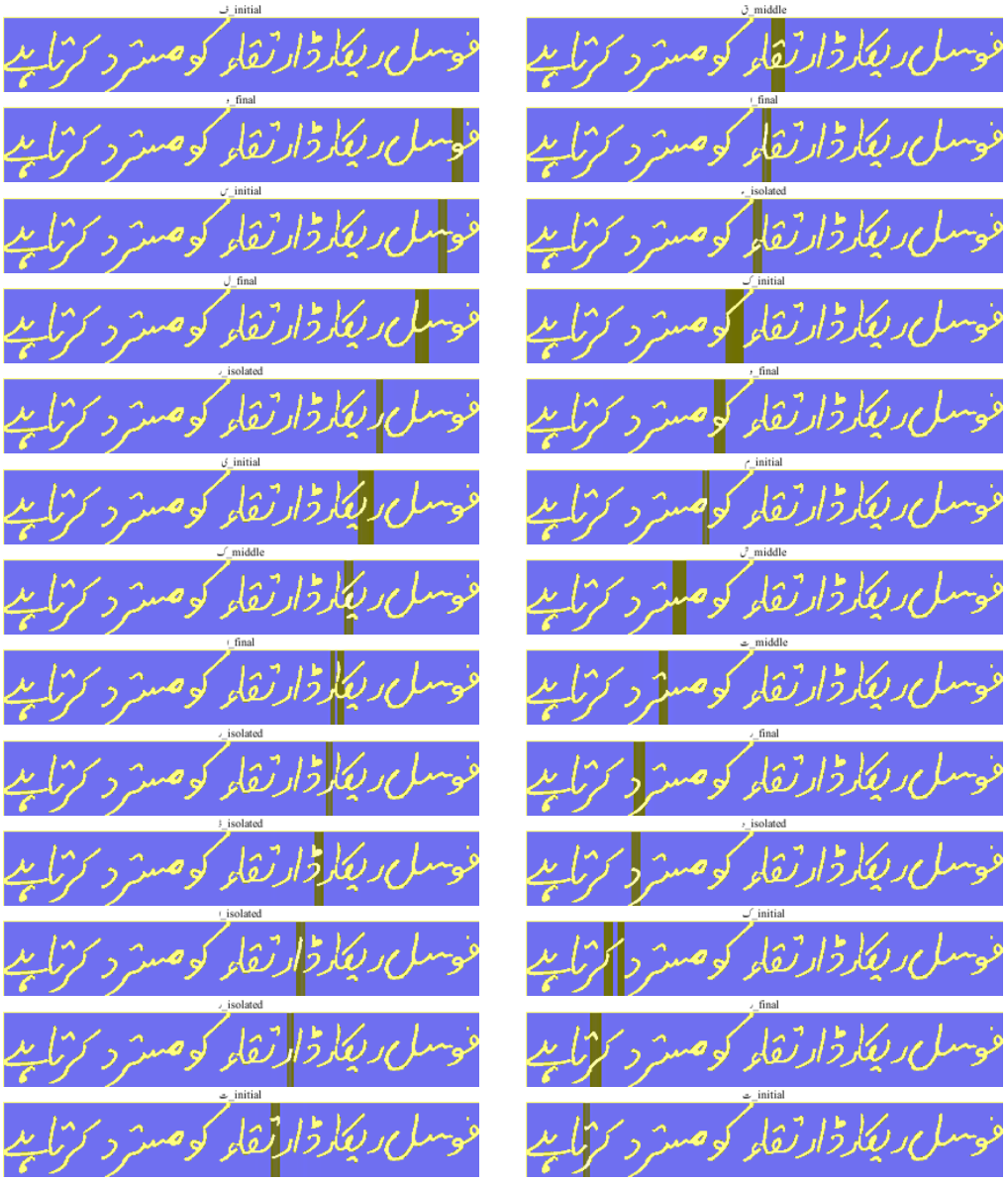
```
"encoder_unit_type": "layer_norm_lstm",
"encoder_type": "bi",
"encoder_num_residual_layers": "0",
"encoder_num_units": "512",

"embed_size": "256",

"use_attention": "True",
"attention_type": "bahdanau",
"attention_num_units": "512",

"pass_hidden_state": "True",
"decoder_num_layers": "2",
"decoder_unit_type": "layer_norm_lstm",
"decoder_num_residual_layers": "0",
"decoder_num_units": "512",

"do_scheduled_sampling": "True",
"initial_not_sampling_prob": "0.80",
"anneal_not_sampling_prob": "True",
"anneal_not_sampling_prob_every": "4000",
"anneal_not_sampling_prob_rate": "0.96",
```

For beam search:

```
"beam_width": "10",
```

Appendix B

Demonstrating Attention




ہ final
فوسل ریکارڈ ارتقاء کو مسترد کرتا ہے
ے final
فوسل ریکارڈ ارتقاء کو مسترد کرتا ہے
ہ initial
فوسل ریکارڈ ارتقاء کو مسترد کرتا ہے
<END>
فوسل ریکارڈ ارتقاء کو مسترد کرتا ہے

Appendix C

Character Frequencies

Character	Position	Train	Test
ص	isolated	72	18
ث	final	90	19
ظ	isolated	92	24
ط	final	99	21
خ	final	103	14
ض	isolated	105	15
ص	final	113	20
ّ	hijja	115	11
ٹ	isolated	125	15
ض	final	133	12
چ	final	141	17
!	isolated	145	24
ج	final	146	15
ڑ	isolated	156	5
ھ	isolated	168	19
ع	isolated	173	27
و	initial	183	12
ا	hijja	189	49
ظ	initial	195	27
ح	final	195	24
(	isolated	199	35
گ	isolated	201	33
ً	isolated	201	16
)	isolated	202	29
َ	hijja	210	27
ِ	isolated	218	15

Character	Position	Train	Test
ش	isolated	221	10
؟	isolated	224	29
ث	middle	224	45
غ	middle	229	29
ع	final	234	47
ث	initial	235	46
ج	isolated	252	33
ء	isolated	255	85
:	isolated	256	17
ٹ	final	260	24
پ	middle	283	27
ک	isolated	286	21
ق	isolated	294	29
ڈ	final	296	26
ش	final	306	53
گ	final	320	32
ف	final	327	61
ذ	final	348	65
ُ	hijja	370	23
ح	isolated	371	55
ڑ	isolated	373	31
ڈ	isolated	394	78
ذ	isolated	425	61
ِ	hijja	458	76
ضَ	middle	466	48
ق	final	483	82

TABLE C.1: Character Frequencies in Training and Test Sets
Key: initial: the first character of its ligature; final: the final character of its ligature; middle: between the first and final characters of its ligature; isolated: a one-letter ligature; hijja (aerab): associated with some character that is part of a ligature (usually separately written either on the top or the bottom of the character)

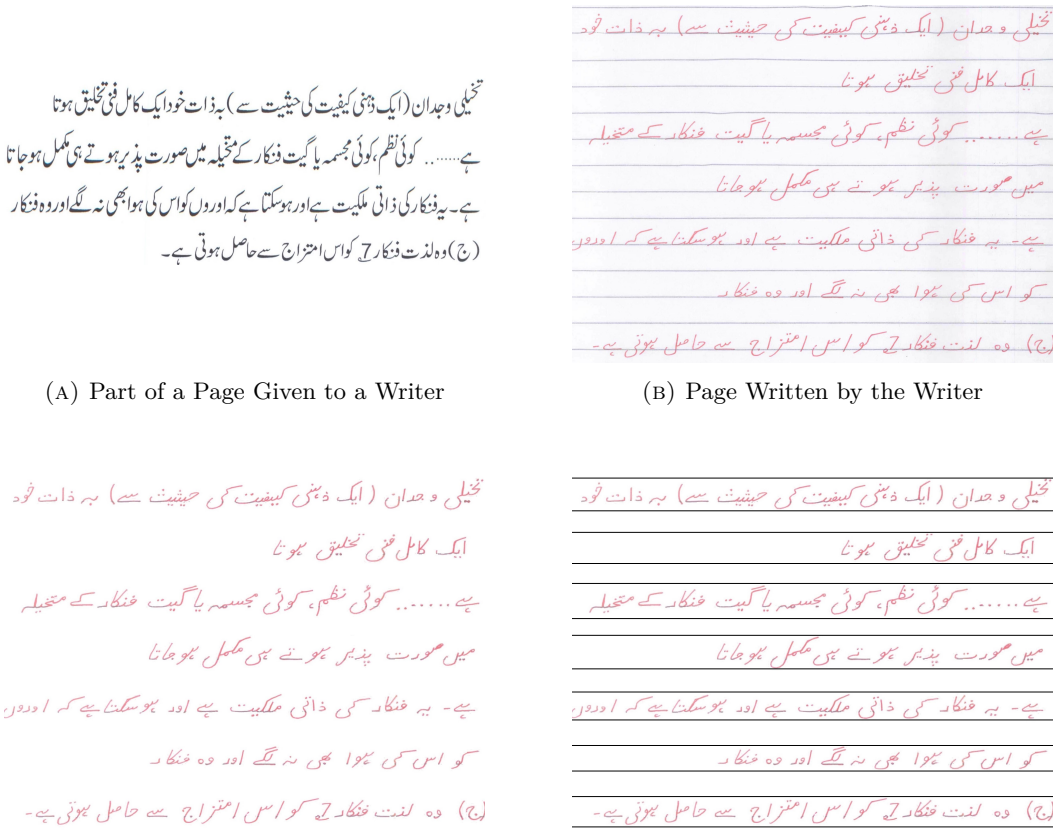
Character	Position	Train	Test
ض	initial	512	72
ظ	middle	522	96
غ	initial	536	66
پ	isolated	564	81
ھ	initial	578	57
ط	middle	616	94
خ	middle	629	131
ٹ	initial	649	43
ف	isolated	678	90
ٹ	middle	794	98
ف	middle	795	112
چ	middle	823	83
ص	middle	841	175
ب	isolated	902	134
ئ	middle	907	124
ش	middle	921	97
س	final	927	122
ح	middle	1028	149
ھ	final	1114	90
ز	final	1120	126
ط	initial	1121	176
ڑ	final	1165	120
گ	middle	1195	124
ب	final	1213	169
ج	middle	1291	167
ص	initial	1320	234
ل	isolated	1375	200
ق	middle	1380	276
ے	isolated	1470	172
م	isolated	1575	254
ل	final	1576	278
ز	isolated	1757	214
ب	middle	1765	291
م	final	1789	241
ک	final	1794	234
چ	initial	1875	186
ن	final	1905	242
ء	isolated	1966	255
ع	middle	1966	308
ف	initial	2002	259
ت	isolated	2029	345
خ	initial	2030	267
ی	isolated	2104	242
ق	initial	2119	358
ش	initial	2313	294

Character	Position	Train	Test
آ	isolated	2335	324
ح	initial	2354	406
ع	initial	2403	352
ت	final	2403	350
ہ	isolated	2424	347
'	isolated	2479	172
س	isolated	2690	416
ن	isolated	2762	400
ئ	initial	2890	392
س	middle	3104	413
ک	middle	3260	495
گ	initial	3386	361
م	middle	3509	478
ں	isolated	3604	433
ہ	middle	3996	490
د	final	4389	581
ت	middle	4405	585
ج	initial	5051	739
ل	middle	5145	726
ل	initial	5600	693
ن	middle	5624	646
پ	initial	5644	681
۔	isolated	6460	710
ہ	final	6501	951
د	isolated	6517	753
ھ	middle	6745	709
ں	final	6928	979
ب	initial	7892	959
ی	initial	8249	1185
س	initial	8297	1097
ت	initial	9108	1289
و	isolated	9962	1321
ن	initial	10388	1340
ہ	initial	11021	1570
ر	isolated	11879	1637
م	initial	11916	1596
ی	final	13254	1717
ر	final	15163	1936
و	final	16372	2185
ی	middle	16873	2390
ے	final	19944	2658
ک	initial	21489	2862
ا	isolated	21750	3183
ا	final	30058	4035

TABLE C.1: Character Frequencies in Training and Test Sets (Continued)

Appendix D

Dataset Preparation Process



(A) Part of a Page Given to a Writer

(B) Page Written by the Writer

(C) After Extracting the Red Pixels

(D) After Segmentation

FIGURE D.1: How the Dataset was Prepared

Bibliography

[1] "Urdu," http://www.omniglot.com/writing/urdu.htm, accessed: 2019-04-08.

[2] "Controversy over number of letters in Urdu alphabet," https://www.dawn.com/news/919270, accessed: 2019-04-08.

[3] 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*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 369–376.

[4] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 5, pp. 855–868, May 2009.

[5] A. L. Maas, A. Y. Hannun, D. Jurafsky, and A. Y. Ng, "First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns," *CoRR*, vol. abs/1408.2873, 2014. [Online]. Available: http://arxiv.org/abs/1408.2873

[6] H. Scheidl, S. Fiel, and R. Sablatnig, "Word beam search: A connectionist temporal classification decoding algorithm," in *16th International Conference on Frontiers in Handwriting Recognition*. IEEE, 2018, pp. 253–258.

[7] C. Darken, J. Chang, and J. Moody, "Learning rate schedules for faster stochastic gradient search," in *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, Aug 1992, pp. 3–12.

[8] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Proceedings of the 27th International Conference on Neural Information Processing Systems*, ser. NIPS'14, vol. 2. Cambridge, MA, USA: MIT Press, 2014, pp. 2933–2941.

[9] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, Jan. 1999.

[10] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Jul. 2011.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2015.

[12] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *CoRR*, vol. abs/1811.03378, 2018.

[13] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, ser. ICML'13, vol. 28. JMLR.org, 2013, pp. III–1310–III–1318.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[15] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," *CoRR*, vol. abs/1303.5778, 2013.

[16] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *Trans. Sig. Proc.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[17] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, pp. 5–6, 2005.

[18] A. Graves, S. Fernández, and J. Schmidhuber, "Multi-dimensional recurrent neural networks," in *Artificial Neural Networks – ICANN 2007*, J. M. de Sá, L. A. Alexandre, W. Duch, and D. Mandic, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 549–558.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[20] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[21] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International*

Conference on International Conference on Machine Learning - Volume 37, ser. ICML'15. JMLR.org, 2015, pp. 448–456.

[24] L. J. Ba, R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[26] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[27] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1171–1179.

[28] F. Huszar, "How (not) to train your generative model: Scheduled sampling, likelihood, adversary?" *CoRR*, vol. abs/1511.05101, 2015.

[29] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 627–635.

[30] A. Goyal, A. Lamb, Y. Zhang, S. Zhang, A. Courville, and Y. Bengio, "Professor forcing: A new algorithm for training recurrent networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 4608–4616.

[31] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR*, 2015.

[32] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421.

[33] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009.

[34] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ser. ACL '96. Stroudsburg, PA, USA: Association for Computational Linguistics, 1996, pp. 310–318.

[35] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling." in *ICASSP*. IEEE Computer Society, 1995, pp. 181–184.

[36] S. Naz, K. Hayat, M. I. Razzak, M. W. Anwar, S. A. Madani, and S. U. Khan, "The optical character recognition of urdu-like cursive scripts," *Pattern Recogn.*, vol. 47, no. 3, pp. 1229–1248, Mar. 2014.

[37] N. Otsu, "A Threshold Selection Method from Gray-level Histograms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[38] S. B. Ahmed, S. Naz, S. Swati, and M. I. Razzak, "Handwritten urdu character recognition using one-dimensional blstm classifier," *Neural Computing and Applications*, pp. 1–9, 2017.

[39] Y. Alginahi, "Preprocessing techniques in character recognition," in *Character Recognition*, M. Mori, Ed. Rijeka: IntechOpen, 2010, ch. 1. [Online]. Available: https://doi.org/10.5772/9776

[40] A. Lawgali, Bouridane, M. Angelova, and Z. Ghassemlooy, "Handwritten arabic character recognition: Which feature extraction method," *International Journal of Advanced Science and Technology*, vol. 34, pp. 1–8, 01 2011.

[41] T. Bluche, J. Louradour, and R. O. Messina, "Scan, attend and read: End-to-end handwritten paragraph recognition with MDLSTM attention," in *14th IAPR International Conference on Document Analysis and Recognition, ICDAR 2017, Kyoto, Japan, November 9-15, 2017*. IEEE, 2017, pp. 1050–1055. [Online]. Available: https://doi.org/10.1109/ICDAR.2017.174

[42] M. Jain, M. Mathew, and C. V. Jawahar, "Unconstrained ocr for urdu using deep cnn-rnn hybrid networks," *2017 4th IAPR Asian Conference on Pattern Recognition (ACPR)*, pp. 747–752, 2017.

[43] L. M. Lorigo and V. Govindaraju, "Offline arabic handwriting recognition: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 5, pp. 712–724, May 2006.

[44] A. Elsawy, M. Loey, and H. El-Bakry, "Arabic handwritten characters recognition using convolutional neural network," *WSEAS TRANSACTIONS on COMPUTER RESEARCH*, vol. 5, pp. 11–19, 01 2017.

[45] A. Sahlol and C. Suen, "A novel method for the recognition of isolated handwritten arabic characters," *CoRR*, vol. abs/1402.6650, 2014.

[46] A. Chowdhury and L. Vig, "An efficient end-to-end neural model for handwritten text recognition," in *British Machine Vision Conference 2018, BMVC 2018, Northumbria University, Newcastle, UK, September 3-6, 2018*, 2018, p. 202. [Online]. Available: http://bmvc2018.org/contents/papers/0606.pdf

[47] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, February 1966.