Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 02, June 2020.

ACCURATE AND FAST RECURRENT NEURAL NETWORK SOLUTION FOR THE AUTOMATIC DIACRITIZATION OF ARABIC TEXT

Gheith Abandah[1] and Asma Abdel-Karim[2]

(Received: 2-Sep.-2019, Revised: 27-Oct.-2019 and 21-Nov.-2019, Accepted: 16-Dec.-2019)

ABSTRACT

Arabic is mostly written now without its diacritics (short vowels). Adding these diacritics decreases reading ambiguity among other benefits. This work aims to develop a fast and accurate machine learning solution to diacritize Arabic text automatically. This paper uses long short-term memory (LSTM) recurrent neural networks to diacritize Arabic text. Intensive experiments are performed to evaluate proposed alternative design and data encoding options towards a fast and accurate solution. Our experiments involve investigating and handling problems in sequence lengths, proposing and evaluating alternative encodings of the diacritized output sequences and tuning and evaluating neural network options including architecture, network size and hyper-parameters. This paper recommends a solution that can be fast trained on a large dataset and uses four bidirectional LSTM layers to predict the diacritics of the input sequence of Arabic letters. This solution achieves a diacritization error rate of 2.46% on the LDC ATB3 dataset benchmark and 1.97% on the larger new Tashkeela dataset. This latter rate is 47% improvement over the best-published previous result.

KEYWORDS

Automatic diacritization, Arabic natural language processing, Sequence transcription, Arabic text, Recurrent neural networks, Long short-term memory, Bidirectional neural network.

1. INTRODUCTION

Automatic diacritization of Arabic text is one of the challenging and important tasks in Arabic Natural Language Processing (NLP). Arabic scripts consist of sequences of words written from right to left using two types of symbols: letters and diacritics. Letters should always be written, whereas diacritics can be omitted, resulting in partially diacritized or undiacritized texts [1]. Except for educated native speakers, lack of diacritization often causes incorrect pronunciation and consequently ambiguity in understanding the text. This is especially true for children and non-native speakers who lack sufficient mastery of the language grammar and lexicon.

Arabic text has two categories: *Classical Arabic* (CA) and *Modern Standard Arabic* (MSA) [1]. CA is the language of the Qur'an and old books and poems. MSA is the primary language used today in the media, education, news and formal speeches in Arabic-speaking countries. MSA is the modern form of CA and is based on it syntactically, morphologically and phonologically. As opposed to CA, most MSA texts are written with partial or no diacritization. In addition to these two categories, there are many informal spoken Arabic dialects. These dialects vary significantly geographically and socially and are neither standardized nor taught in schools. However, they are becoming more popular in writing Arabic texts on smart phones and over the internet [2]-[3].

The Arabic language has 28 letters and eight basic diacritics. Table 1 shows parts of the Unicode Block 0600-06FF that includes the Arabic letters and diacritics [2]. There are 36 variants of the 28 Arabic letters, which have Unicode hexadecimal codes 0621–063A and 0641–064A. These variants come from adding the six *Hamza* letters (ئ، إ، ؤ، أ، آ، ء), the *Teh Marbuta* (ة) and the *Alef Maksura* (ى) to the basic 28 letters. Arabic diacritics have Unicode codes 064B–0652. There are three types of Arabic diacritics: Vowel diacritics, Nunation diacritics and *Shadda*. Vowel diacritics include short vowels (*Fatha* ◌َ, *Damma* ◌ُ, *Kasra* ◌ِ) and the absence of vowel (*Sukun* ◌ْ). Nunation diacritics look like double versions of their corresponding short vowels (*Fathatan* ◌ً, *Dammatan* ◌ٌ, *Kasratan* ◌ٍ). The *Shadda* diacritic (◌ّ)

G. Abandah and A. Abdel-Karim are with Computer Engineering Department, The University of Jordan, Amman, Jordan. Emails: [1]abandah@ju.edu.jo and [2]a.abdelkarim@ju.edu.jo

implies doubling the letter it appears on [1].

Table 1. Unicode Arabic code block showing 36 letter variants and the basic eight diacritics.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+062x		ء	آ	أ	ؤ	إ	ئ	ا	ب	ة	ت	ث	ج	ح	خ	د
U+063x	ذ	ر	ز	س	ش	ص	ض	ط	ظ	ع	غ					
U+064x		ف	ق	ك	ل	م	ن	ه	و	ى	ي	ً	ٌ	ٍ	َ	ُ
U+065x	ِ	ّ	ْ													

In Arabic, words that have the same letters but different diacritics have different pronunciations and meanings. For example, the undiacritized word كتب has several meanings based on the way it is diacritized. If it is diacritized as كَتَبَ, it is pronounced "kataba" and means "wrote". However, if it is diacritized as كُتُب, then it is pronounced "kutub" and means "books". It can also be diacritized as كُتِبَ, pronounced "kutiba" and means "was written" indicating the past passive voice. A native reader can infer which diacritization form to use for a word based on the context. For example, for the statement كتب الطالب رسالة, the reader can infer that this is a verb-subject-object sentence "The student wrote a message" and hence the correct diacritization of the word كتب is كَتَبَ "kataba" [4].

The diacritization problem is even more complex when considering the *inflectional diacritics*. This type of diacritics is based on rules that inflect the word according to the context. Arabic words are inflected according to the word's tense, person, voice, gender, number, case and definiteness. The inflectional diacritics mostly occur on the last letter. For instance, the past verb كَتَبَ "kataba", which was not inflected in the last example, is inflected for first person as كَتَبْتُ "katabtu" and for feminine second person as كَتَبْتِ "katabti". In some cases, the inflectional diacritics do not appear on the last letter. For example, the composite word of noun and pronoun كتابه "his book" is diacritized كِتابُهُ "kitabuhu" when it is a subject and كِتابَهُ "kitabahu" when it is an object [5].

In order to correctly diacritize a sentence, the entire sentence context should be analyzed. Table 2 shows an example sentence where the diacritization of the first four words depends on the fifth word. In (a), the fifth word "كَثِيرَةٌ" "numerous" is adjective and implies that كتب is a noun and should be diacritized كُتُبُ "kutubu". In (b), the fifth word "الدَّرْسَ" "lesson" is noun and implies that كتب is a verb and should be diacritized كَتَبَ "kataba". This example also shows how the last letter diacritic of each word in the sentence differs based on the last word of the sentence. In fact, obtaining the correct diacritics of words' last letters, also known as *end cases*, is considered the most challenging part of automatic text diacritization. Therefore, automatic text diacritization requires an approach that takes into account both past and future contexts. Moreover, long-term context should be checked, since diacritics may depend on three or more distant words [4].

Table 2. Example of two different diacritizations of four words based on the fifth word.

Sentence	***Possible Diacritizations***	***Meaning in English***
كتب أحمد وعلي وعمر _____	كُتُبُ أَحْمَدَ وَعَلِيٍ وَعُمَرَ كَثِيرَةٌ	(a) Books of Ahmed, Ali and Omar are numerous.
	كَتَبَ أَحْمَدُ وَعَلِيٌ وَعُمَرُ الدَّرْسَ	(b) Ahmed, Ali and Omar wrote the lesson.

The objective of this work is to develop a fast and accurate model that uses *recurrent neural networks* (RNNs) to transcribe raw undiacritized sequences into fully diacritized sequences. We concentrate on networks that exploit long-term past and future contexts to make diacritics predictions and that can be

trained using datasets in reasonable times. We train and test RNN models using two datasets: Linguistic Data Consortium's Arabic Treebank part 3 (LDC-ATB3) [6], which serves as an example of MSA and the cleaned subset of *Tashkeela* [7], which serves as an example of CA. As mentioned earlier, automatic diacritization of Arabic text provides help to children and non-native speakers in learning the language. In addition, it is an important step in text-to-speech (TTS) software and automatic speech recognition (ASR) engines.

Throughout our experiments, we explore and analyze the effect of tuning several network parameters, such as the number of network layers and using dropout, on the accuracy and execution time of the tested models. We experiment alternative approaches to handle problems in sequence lengths and propose wrapping of sequences to solve the problem. We also use multiple encoding methods for the diacritized output sequences and propose two new encoding methods. In addition, we experiment with three network architectures: unidirectional *long short-term memory* (LSTM), *bidirectional* LSTM and *encoder/decoder* LSTM networks.

The rest of this paper is organized as follows. In the next section, we provide a review of automatic Arabic diacritization systems proposed in the literature. Section 3 provides background information of the RNN models we use in this work. Section 4 describes our experimental setup. Section 5 presents and discusses the results of our experiments. Section 6 compares our best results with the results of previous best-performing models and analyzes the errors. Finally, Section 7 gives the conclusions.

2. LITERATURE REVIEW

Systems developed for automatic diacritization of Arabic text can be classified into three categories: *rule-based* systems, *statistical* systems and *hybrid* systems.

2.1 Rule-based Systems

Rule-based approaches require defining a set of well-formed rules that exploit human knowledge in the form of morphological analyzers, dictionaries and grammar modules. Although rule-based approaches solve the problem with acceptable results, they rely on linguistic knowledge or parsing tools and require rules to be continuously maintained and updated [8]-[9].

2.2 Statistical Systems

Statistical approaches, on the other hand, predict the probable diacritics for a sequence of characters without the need for language-specific knowledge or parsing tools. Instead, they require a large corpus of diacritized text. Machine learning statistical methods that have been applied to Arabic text diacritization include hidden Markov models (HMMs), n-grams, finite state transducers (FSTs) and more recently RNNs [8].

Gal [10] used an HMM to restore Arabic diacritics with the Holy Quran as a corpus. His system restores only short vowels and correctly diacritizes 89% of the words in the test set. Elshafei et al. [11] proposed a similar approach that uses an HMM for modeling and Viterbi search algorithm to find the most optimal diacritics of a sentence. Their training data was taken from multiple knowledge domains and the tests used randomly picked verses from the Quran. They achieved a 4.1% diacritization error rate. Refer to Subsection 4.5 for the definition of diacritization error rates.

Hifny [12] proposed an automatic diacritization system that combines dynamic programming (DP) with n-gram language model and smoothing. He used n-gram language modelling to assign scores to possible diacritized word sequences. Dynamic programming is then used to search for the most likely sequence. Different smoothing algorithms are tested to solve the problem of unseen n-grams. The author used the Tashkeela dataset [13] for training and testing his model with a corpus of 5.25 million words for training and a testing set of 1.9 million words. His approach achieved a word error rate of 3.4% when end cases' are excluded and 8.9% when these cases are included. This is due to the difficulty of retrieving end cases diacritics as outlined in the previous section. Definition of word error rate is also provided in subsection 4.5.

Azim et al. [14] proposed a statistical approach that uses weighted combination of two diacritizers: one is text-based and the other is speech-based. The system uses a correctly vocalized speech of the text to

complement and correct errors generated by the text-based model. The text-based diacritizer is modelled by *conditional random fields* (CRFs) and the speech-based diacritizer is modelled by HMM. Using LDC ATB3, their approach achieves very accurate diacritization and word error rates of 1.5% and 4.9%, respectively. However, their system requires the availability of an acoustic signal that corresponds to the raw text data.

2.3 Hybrid Systems

Most current systems use hybrid approaches that make use of language-specific rules to guide statistical techniques. Vergyri and Kirchhof [15] explored multiple combinations of acoustic information with morphological and contextual sources. In their experiments, they used two corpora: the Foreign Broadcast Information Service (FBIS) corpus of MSA speech and the LDC Call Home Egyptian Colloquial Arabic (ECA) corpus. Without modelling the *Shadda* diacritic, they achieved diacritization and word error rates of 11.5% and 27.3%, respectively. Nelken et al. [16] proposed a hybrid model that uses a cascade of finite state transducers with integrated word-based model, letter-based model and morphological model. The diacritization and word error rates of their model using LDC Arabic Treebank of diacritized news stories (Part 2) are 12.8% and 23.6%, respectively.

Zitouni et al. [17] proposed an approach based on maximum entropy framework that learns the correlation between several input features and the output diacritics. These features include lexical features, segment-based features and part-of-speech (POS) features. They trained and tested their model using LDC ATB3. They provided an in-detail description of their usage of the LDC ATB3 and produced a clearly defined split of the dataset into training and testing subsets. This split established LDC ATB3 as a benchmark in this area and allowed for reproduction of results and accurate comparison with subsequent techniques. Their approach achieves diacritization and word error rates of 5.5% and 18.0%, respectively.

In [5], Habash and Rambow extended their morphological analysis and disambiguation of Arabic system (MADA) such that it consults the Buckwalter Arabic Morphological Analyzer (BAMA) to get a list of all potential analysis of a word. Fourteen Support Vector Machine (SVM) predictors are then used to narrow this list to a smaller one. Finally, n-gram language models are used to select one solution from the narrowed list. They trained and tested their approach using LDC ATB3 as proposed by Zitouni et al. [17]. Their approach achieves diacritization and word error rates of 4.8% and 14.9%, respectively.

Rashwan et al. [18] introduced a system that uses two stochastic layers in order to perform automatic diacritization. The first layer is an un-factorized layer that diacritizes letters by searching a dictionary that was built offline. It retrieves all diacritized forms of the word if it is found. The most likely sequence is then selected using the n-gram probability estimation and A* lattice search. The second layer factorizes words that were not diacritized in the first layer into their morphological components (prefix, root, pattern and suffix). N-gram probability estimation and A* lattice search are also used in this layer to select the most likely diacritization from the generated factorizations. The reported diacritization and word error rates of their approach using LDC ATB3 are 3.8% and 12.5%, respectively.

The hybrid system developed by Said et al. [19] includes an automatic corrector, rule-based and statistical morphological analyzers, a POS tagger and an out-of-vocabulary diacritizer. Their rule-based analyzer was formed based on comprehensive lexicon and handcrafted rules. The statistical analyzer was trained using LDC ATB3. Given an input word, these analyzers produce a lattice of diacritized forms. The POS tagger disambiguates this lattice and selects the most likely diacritized form for the word using HMM and Viterbi algorithm. Their approach achieved diacritization and word error rates of 3.6% and 11.4%, respectively.

Our previous work in [4] was the first to use RNN to solve the diacritization problem as a sequence transcription problem. More specifically, we proposed, trained and tested a bidirectional LSTM network that takes as an input raw undiacritized sequences and transcribes them into diacritized sequences. Our approach did not apply lexical, morphological or syntactical analysis prior to or in line with the data training. We used error-correction techniques to post-process the output of the network. We used LDC ATB3, the simple version of the holy Quran and ten books drawn from the Tashkeela dataset [13]. We achieved state-of-the-art performance with diacritization and word error rates of 2.09% and 5.82%, respectively, for Tashkeela and 2.72% and 9.07%, respectively, for ATB3. A follow up work shows that

the RNN accuracy can even be slightly improved when a morphological and syntactical analyzer preprocesses the RNN input [20].

Rashwan et al. [21] proposed a system that consists of two frameworks: a deep learning framework that uses the confused sub-set resolution (CSR) method to improve the classification accuracy and an Arabic part-of-speech (PoS) tagging using deep neural networks. Their system achieves an accuracy of 97% using LDC ATB3. In [7], Fadel et al. compared the performance of some publicly available rule-based systems with the neural-based approach Shakkala [22] using their cleaned subset of Tashkeela. Shakkala outperformed the best performing rule-based approaches, mainly Mishkal [23] and Harakat, with diacritization and word error rates of 3.73% and 11.19%, respectively.

More recently, Mubarak et al. [24] implemented a sequence-to-sequence model using an encoder-decoder LSTM RNN with attention mechanism. They used sliding window to divide sequences into fixed lengths. The most likely diacritic form of a word is selected using n-gram probability estimation. They trained their model using 4.5 million tokens and tested it using the freely available WikiNews corpus of 18,300 words [25]. They do not identify their training data or refer to its source. Their best reported results are 1.21% and 4.49% diacritization and word error rates, respectively. Although these are the best results reported so far, we do not include them in our comparison, because they are not generated using the same datasets commonly used in this domain.

In this paper, we build on our previous work in [4] by implementing our model using state-of-the-art tools. Moreover, we perform intensive experiments that explore alternative implementation options (e.g., network architecture, optimization techniques and number of layers) and data preparation options (e.g., encoding methods and handling sequence lengths) towards a faster and more accurate model.

3. SEQUENCE TRANSCRIPTION

Sequence Transcription is the process of translating an input sequence into the corresponding output sequence of a different type. This includes language translation, voice recognition and diacritizing Arabic texts. Recurrent neural networks have proved to perform best on sequence transcription [26]. This is due to their ability to preserve correlations between data points in the sequence, as their hidden states are functions of all previous states with respect to time [27].

3.1 Recurrent Neural Networks

Given a sequence of inputs $(x_1, x_2, \dots, x_T)$, a standard RNN computes a sequence of outputs $(y_1, y_2, \dots, y_T)$ based on the computation of a sequence of hidden vectors by iterating the following equations from $t = 1$ to T:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \quad (1)$$

$$y_t = W_{hy}h_t + b_y \quad (2)$$

where W terms denote weight matrices and b terms denote bias matrices. For computing each hidden state, there are two sets of weights: one for the inputs x_t and one for the previous hidden state h_{t-1} [28]. A sigmoid function σ is normally used as the activation function for basic RNNs [26]. RNNs can be used to solve four types of sequence transcription problems according to the lengths of the input and the output [28]. These types are shown in Figure 1.

The first RNN type, shown in Figure 2a, takes an input sequence and produces an output sequence of the same length. These networks are referred to as *one-to-one* networks. The second type is the *sequence-to-vector* network, where input sequences are transcribed into one final output by ignoring all previous outputs. The third type is the *vector-to-sequence* network, where one input vector is used to produce an output sequence. The fourth type is the general *sequence-to-sequence* network, where the output sequence is generally not of the same length as the input sequence. This type is often implemented using the *encoder-decoder* architecture.

Given that automatic diacritization of Arabic text is a sequence-to-sequence problem, both the one-to-one and the encoder-decoder networks can be used to solve the problem. In this work, we implement both types using multiple encoding methods for the output sequences. In this problem, the encoder-decoder approach is implemented with output sequences (that include letters and diacritics) that are

longer than the input undiacritized sequences and hence is considered a *one-to-many* sequence-to-sequence transcription.

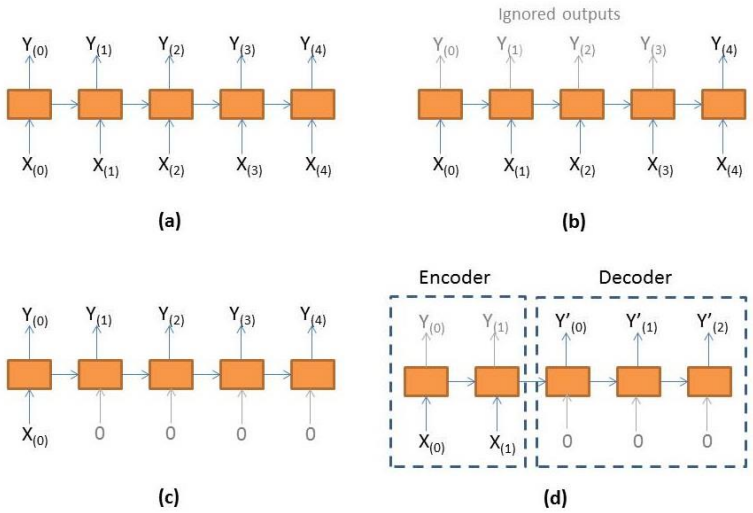

Figure 1. RNN types based on lengths of the input and the output: (a) one-to-one sequence-to-sequence, (b) sequence-to-vector, (c) vector-to-sequence and (d) general sequence-to-sequence.

3.2 Long Short-Term Memory Cells

The basic recurrent networks described above consist of memory cells that can store representation of recent inputs only. Hence, they have a short-term memory that results in slowly changing weights [29]. LSTM networks, on the other hand, which use purpose-built memory cells, are capable of learning from long-term contexts [27]. Each memory cell has an *input* gate, a *forget* gate, an *output* gate and a *cell activation* unit. These units are represented by the vectors i, f, o *and* c, respectively, which are of the same size as the hidden vector h_t. The following equations show how the hidden vector's activation function for LSTM is a composite function that results from computing the aforementioned vectors.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \quad (3)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad (4)$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (5)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \quad (6)$$

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

Notice that these vectors depend on the layer input x_t and the previous states: short-term state h_{t-1} and long-term state c_{t-1}.

3.3 Encoder/Decoder Networks

Encoder-decoder RNNs are designed such that the network consists of two RNNs. The first RNN represents the encoder which maps an input sequence into a fixed-length vector acting as a sequence-to-vector network. More specifically, this vector is a summary of the input sequence. The second RNN is the decoder which maps the encoder vector into an output sequence forming a vector-to-sequence network. The two networks are jointly trained to maximize the probability of an output sequence given an input sequence. As introduced above, this architecture allows mapping input sequences to output sequences of different lengths [30].

3.4 Bidirectional RNNs

Conventional unidirectional RNNs can make use only of previous context. However, many sequence transcription problems, including diacritization, require exploiting future context as well. Bidirectional RNN layers achieve this by comprising two adjacent unidirectional networks in each layer. One network is trained by presenting the sequence in the forward direction and the other is trained by presenting it in

the backward direction. The output is a function of both networks and, consequently, exploits past and future contexts. Specifically, the forward hidden vector is computed by iterating in the positive time direction (i.e., from $t = 1$ to T), while the backward hidden vector is computed by iterating in the negative time direction (i.e., from $t = T$ to 1) [31]. Both vectors are used to update the output vector y_t, as specified in the following equations:

$$\vec{h}_t = \sigma\left(W_{x\vec{h}}x_t + W_{\vec{h}\vec{h}}\vec{h}_{t-1} + b_{\vec{h}}\right) \quad (8)$$

$$\overleftarrow{h}_t = \sigma\left(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h}_{t-1} + b_{\overleftarrow{h}}\right) \quad (9)$$

$$y_t = W_{\vec{h}y}\vec{h}_t + W_{\overleftarrow{h}y}\overleftarrow{h}_t + b_o \quad (10)$$

3.5 Deep RNNs

RNNs are made even more powerful by stacking multiple layers on top of each other, forming a deep RNN. Deep networks are necessary to solve complex transcription functions. In such architectures, the output sequence of one layer acts as the input sequence for the next layer. Assuming that the same hidden function σ is used for all N layers in the stack, the hidden vectors h^n are computed by iterating from $n = 1$ to N and from $t = 1$ to T, as shown in Equation 11, where $h^0 = x$. The network final output y_t is computed according to Equation 12.

$$h_t^n = \sigma(W_{h^{n-1}h^n}h_t^{n-1} + W_{h^nh^n}h_{t-1}^n + b_h^n) \quad (11)$$

$$y_t = W_{h^Ny}h_t^N + b_y \quad (12)$$

4. EXPERIMENTAL SETUP

We use an experimental setup similar to that used in our previous work in [4]. During the training process, both input undiacritized sequences and target diacritized sequences are presented to the model after encoding. The model is tested by applying undiacritized sequences to its input and comparing the transcribed output sequences with the correctly diacritized sequences. Our final reported results are obtained after post-processing is performed on the predicted output sequences to correct some transcription errors. Post-processing techniques include *Sukun* correction, *Fatha* correction and dictionary correction, which were proposed and discussed in our previous work in [4]. The processing and memory specifications of the platform on which our experiments were performed are shown in Table 3. The following subsections describe other aspects of our experimental setup.

Table 3. Processing and memory specifications of the experimental platform.

CPU	Intel Core i7-6700 @ 3.4 GHz, 4 cores (8 threads), 8 MB cache
GPU	Nvidia GeForce RTX 2080 @ 2.1 GHz, 2944 CUDA cores, 8 GB memory
Memory	32 GB DDR4-SDRAM @ 1066MHz

4.1 Data

Our experimental data consists mainly of text from the Linguistic Data Consortium's (LDC) Arabic Treebank (LDC2010T08) [6] and the cleaned subset of Tashkeela corpus extracted in [7]. More specifically, we use the LDC's Arabic Treebank Part 3 (ATB3) v3.2, which consists of 599 distinct newswire stories from the Lebanese publication An Nahar. Text in this dataset is an example of the modern standard Arabic (MSA). We split this dataset, as proposed by Zitouni et al. [17], such that the first 509 newswire stories, in chronological order, are used for training the model and the last 90 stories are used for validation and testing. This accounts for 22,170 sequences for training and 3,857 sequences for validation.

The used Tashkeela dataset includes 55K lines randomly chosen by Fadel et al. [7] from the classical Arabic (CA) and Holy Quran datasets. The provided dataset is a processed subset of the original datasets with some file formatting errors removed and many diacritization issues fixed. The dataset is split into 50K lines for training, 2,500 lines for validation and 2,500 lines for testing. Table 4 shows size statistics of these two datasets: word count, sequence count, average letters per word and average words per sequence.

Table 4. Datasets' size statistics.

Dataset	*Word Count*	*Sequence Count*	*Letters per Word*	*Words per Sequence*
LDC ATB3	305 K	26,027	4.6	11.3
Tashkeela	2,312 K	55 K	4.0	42.1

Table 5 provides statistics of diacritics usage in these datasets in terms of the percentage of letters without diacritics, with one diacritic and with two diacritics.

Table 5. Datasets' diacritics usage statistics.

Dataset	*No Diacritics*	*One Diacritic*	*Two Diacritics*
LDC ATB3	39.8%	54.8%	5.4%
Tashkeela	17.8%	77.2%	5.0%

Tashkeela is larger than ATB3 in number of sequences and sequence lengths. However, both datasets have close average number of letters per word, which is a property of the Arabic language [4]. Tashkeela has smaller percentage of characters with no diacritics compared to ATB3. This is due to the extraction process conducted in [7] of the used Tashkeela subset which ensured diacritics to characters rate greater than 80%.

One of the aspects that we address in this study is the effect of variation in sequence lengths and having very long sequences on both the execution time and the accuracy. The maximum sequence length for ATB3 is 695 letters, whereas Tashkeela has a maximum sequence length of 7,542 letters. Nevertheless, both datasets have small percentages of very long sequences. Figure 2 shows the cumulative distribution function (CDF) of the sequence length for ATB3 and Tashkeela datasets. Only 1% of ATB3's sequences are longer than 233 characters, whereas only 1% of Tashkeela's sequences are longer than 1,194 characters.

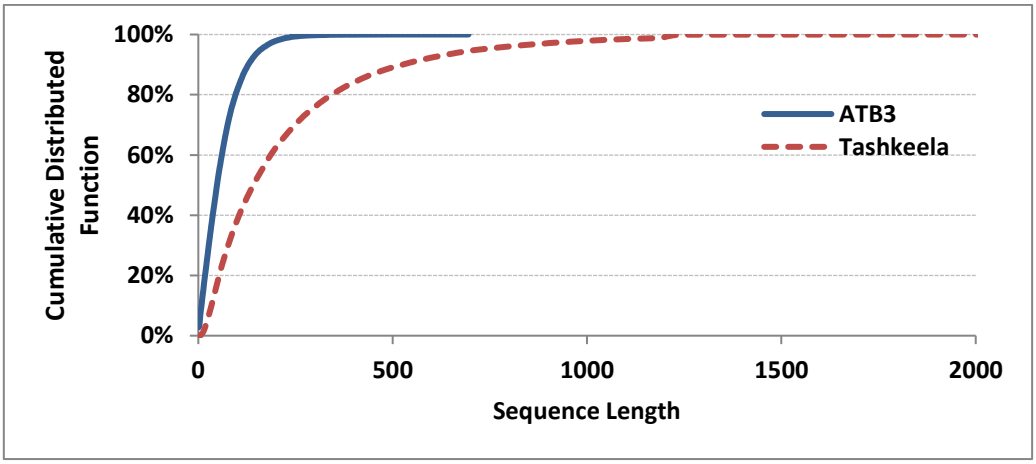

Figure 2. Cumulative distribution function (CDF) of sequence length of ATB3 and Tashkeela datasets.

4.2 Data Preparation

The Tashkeela dataset was cleaned by Fadel et al. [7]. Their cleaning process included solving some diacritization issues, such as fixing misplaced diacritics and removing the first diacritic in cases of letters with multiple diacritics. They also prepared the dataset by removing English letters, separating numbers from words by adding whitespaces and removing multiple whitespaces. In addition, they performed some file formatting, such as removing tags from HTML files and removing URLs.

In order to perform machine learning using datasets larger than the computer memory, we converted both datasets into TensorFlow records (TFRecords) [32]. The TFRecord format is a format for storing data on the disk and allows reading huge data efficiently during training and testing. Our TFRecords consist of sequences, each sequence consists of tokens and each token is one-hot encoded in a vector. Storing these datasets, in their original dense format, results in consuming very large disc spaces. For

ATB3, the training TFRecord file is 3.5 GB and is 102 GB for Tashkeela. To reduce this space, we considered skipping sequences longer than 300 characters. This reduces ATB3 training TFrecord file to 1.5 GB and to 4 GB for Tashkeela. However, skipping long sequences reduces the number of sequences used in training.

Therefore, we experimented with using a sparse format that exploits the fact that the one-hot encoding gives vectors that mostly consist of 0's. The size when using the sparse format depends on the number of tokens, but unlike the dense format, does not depend on the maximum sequence length. Moreover, for the dense case when sequences longer than 300 characters are not skipped, we wrap long sequences to a maximum of 400 characters (see Section 5.1 for more detail).

Figure 3 shows ATB3 and Tashkeela TFrecord training file sizes, in logarithmic scale, for the dense and sparse formats both with and without skipping of sequences longer than 300 characters. Sparse format is much smaller than the dense format for both datasets even when sequences longer than 300 characters are not skipped. Hence, we use sparse files to store the entire datasets while maintaining reasonable usage of the disc space. Moreover, the execution time when using the sparse format is comparable to that of the dense format. The overhead of converting sparse format back into dense matrices is hidden by the time saved when avoiding the long disc access time of the large dense files.

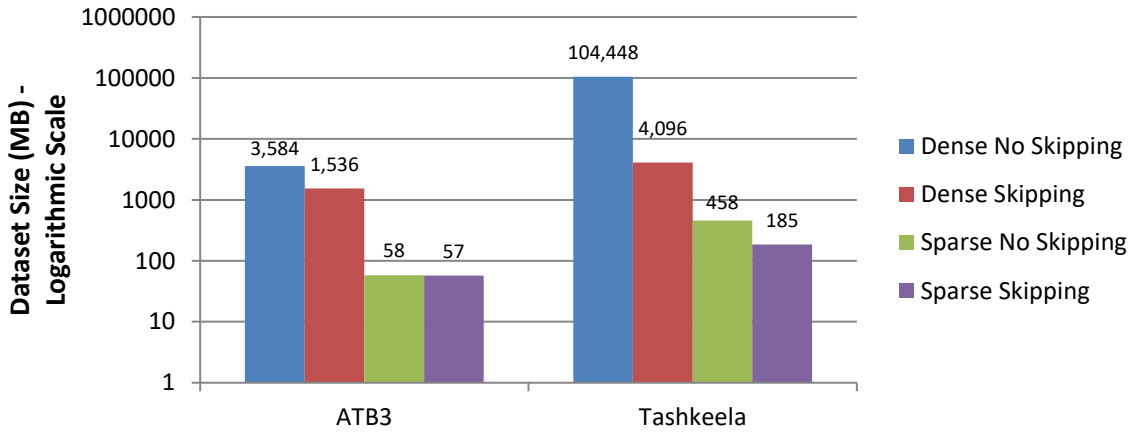

Figure 3. Training dataset sizes when using dense and sparse TFRecords with and without skipping of sequences longer than 300 characters.

4.3 Data Encoding

This subsection describes how input and output sequences are encoded. Input undiacritized sequences are obtained by removing diacritics from target diacritized sequences. Since they consist of letters only, input sequences are encoded using the Unicode representations of their letters. For example, the undiacritized word "ثم" is encoded as "062B 0645": the Unicodes of the letters ث and م, respectively.

For diacritized target sequences, we experiment with four encoding methods as described below. Table 6 shows the eight main Arabic diacritics and their shapes, sounds, hexadecimal Unicode numbers and the binary bit codes used to encode them in this work. Each letter in Arabic may have no diacritics, one diacritic or two diacritics. When a letter has two diacritics, one of these letters must be *Shadda*. The diacritized sequence encoding methods used in this work are a one-to-many encoding method and three one-to-one encoding methods.

1) One-to-many encoding

In one-to-many encoding, we use separate symbols for the letter and its diacritics as in Unicode. The Unicode representations of the letter's diacritic(s) follow the letter Unicode representation. For example, the diacritized word "ثُمَّ" is encoded as "062B 064F 0645 0651 064E". Therefore, diacritized target sequences are usually longer than undiacritized input sequences. In this work, we use one-to-many encoding with the encoder-decoder network.

Table 6. The main eight Arabic diacritics with their shapes, sounds, hexadecimal Unicode numbers and used binary bit codes.

Name	*Shape*	*Sound*	*Unicode*	*Bit code*
Fathatan	ً	/an/	064B	0001
Dammatan	ٌ	/un/	064C	0010
Kasratan	ٍ	/in/	064D	0011
Fatha	َ	/a/	064E	0100
Damma	ُ	/u/	064F	0101
Kasra	ِ	/i/	0650	0110
Sukun	ْ	None	0652	0111
Shadda	ّ	Doubling	0651	1000

2) Many classes, one-to-one encoding

In one-to-one encoding, one symbol is used to encode each letter with its diacritic and hence input and target sequences have the same length. The first method is the encoding method used in our previous work [4]. Using this method, each diacritized letter is encoded in a symbol that results from combining the letter code with its diacritic(s) bit code(s), as shown in Equation 13. Each letter is encoded into a unique code L that is formed by clearing the most significant byte of the letter's Unicode number l, which is 0x06 for all Arabic letters. Then, the masked code is shifted four-bit positions to the left and ORed with the bit code of the letter diacritic d_1 if it has one diacritic or the bit codes of its two diacritics d_1 and d_2 if it has two diacritics. Notice that this encoding method gives many output classes in the order of the number of letters times the number of diacritics.

$$L = \begin{cases} (l \wedge \text{0x00ff} \ll 4), & \text{no diacritic} \\ (l \wedge \text{0x00ff} \ll 4) \vee d_1, & \text{one diacritic} \\ (l \wedge \text{0x00ff} \ll 4) \vee d_1 \vee d_2, & \text{two diacritics} \end{cases} \tag{13}$$

For example, in order to encode the letter مَّ of the word ثُمَّ, the Unicode of the letter م which is 0645 is masked into 0045. Then, the code is shifted 4 bits into 0450 and the combined bit code of *Fatha* and *Shadda* (0100 ∨ 1000 = 1100 = C) is inserted in the least significant four bits of the letter code (0450) to form the code 045C.

3) Diacritics only, one-to-one encoding

In this work, we propose and test two other one-to-one encoding methods. In the first method, each diacritized letter is encoded using its diacritics only. This encoding scheme relies on the fact that letters in the undiacritized input sequence do not change in the target diacritized sequence except for adding diacritics to them. This has the advantage of limiting the number of possible output classes to the number of possible diacritics and hence simplifies the output layer. Equation 14 shows the way in which a unique code *L* is formed using this encoding scheme using the diacritics bit code(s) without involving the letter unicode representation. For example, the letter مَّ of the word ثُمَّ is encoded using its diacritics bit codes only which is C (0100 ∨ 1000 = 1100 = C).

$$L = \begin{cases} 0, & \text{no diacritic} \\ d_1, & \text{one diacritic} \\ d_1 \vee d_2, & \text{two diacritics} \end{cases} \tag{14}$$

4) Multiple label, one-to-one encoding

The second one-to-one encoding method that we propose in this work assumes that each bit in the code represents a label that contributes to the diacritization of the letter. Table 7 illustrates the labels assigned to each bit position in this encoding method. A value of 1 in a bit position indicates that the

corresponding diacritic is present. For example, the letter مَّ of the word ثُمَّ is encoded using this method by placing 1s in the *Shadda* label (bit position 5) and the *Fatha* label (bit position 1) to form the binary code 100010 (hexadecimal 22).

Table 7. Labels assigned to bits in multiple label encoding.

Bit Position	***5***	***4***	***3***	***2***	***1***	***0***
Label	*Shadda*	*Nunation*	*Kasra*	*Damma*	*Fatha*	*Sukun*

To summarize, Table 8 shows the example word ثُمَّ encoded using these four encoding methods.

Table 8. Encoding the diacritized word ثُمَّ using one-to-many, many classes, diacritics only and multiple label encoding.

Encoding Method	***Encoding of letters***		***Word Encoding***
	ثُ	مَّ	
One-to-many	062B 064F	0645 0651 064E	062B 064F 0645 0651 064E
Many Classes	02B5	045C	02B5 045C
Diacritics Only	5	C	5 C
Multiple Labels	04	22	04 22

4.4 Base Model

We use Keras (Python deep learning library) with TensorFlow at the backend to develop our machine learning models [32]. This combination implements the state-of-the-art algorithms in deep machine learning. Our baseline model is an LSTM RNN with two bidirectional layers and 256 cells per layer preceded by a masking layer and followed by a fully-connected output layer. This model uses *softmax* as the activation of the output layer, the *RMSprop* optimizer in training, categorical cross entropy as the loss function and a batch size of 64 sequences [28]. Figure 4 shows the core code of this model.

```
model = Sequential()
model.add(Masking(mask_value= 0, input_shape=(seq_len, num_inp_tokens)))
model.add(Bidirectional(LSTM(256, return_sequences=True), merge_mode='concat'))
model.add(Bidirectional(LSTM(256, return_sequences=True), merge_mode='concat'))
model.add(TimeDistributed(Dense(num_tar_tokens, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['acc'])
```

Figure 4. Python Keras core code of the base model.

4.5 Evaluation Metrics

We evaluate models in terms of execution time required to train the model and the accuracy of the model in diacritizing the input sequences. Throughout our experiments, we evaluate the accuracy of multiple designs and data handling/encoding options using the *diacritization error rate*. DER is the percentage of characters with incorrectly predicted diacritics. For all experiments, we use the original DER definition where punctuation marks and numbers are counted [17]. We also report the *word error rate* of our best performing model. WER is the percentage of incorrectly diacritized words. A word is considered incorrectly diacritized if it has at least one incorrectly diacritized letter.

For the experiment that evaluates alternative network architectures, we use the accuracy as the evaluation metric, because DER and WER cannot be obtained for the encoder/decoder network, as explained in Section 5.3.

5. Experiments and Results

The following subsections present the experiments and their results.

5.1 Handling Sequence Lengths

As discussed earlier, both datasets have high variation in their sequence lengths. We conducted three experiments to find the best approach to handle this variation. Figure 5 shows the training time required and the diacritization error rate obtained for the three experiments for both datasets. We first included all sequences for the ATB3 dataset (i.e., maximum sequence length is 695) and sequences not exceeding 1,260 characters for Tashkeela (i.e., this accounts for 99.94% of Tashkeela sequences). Then, we included only sequences with a maximum of 300 characters for both datasets. Finally, we experimented including all sequences, but wrapping long sequences to have a maximum of 400 characters per input. The network input is arranged in tensors (dense matrices) with a width that equals the longest sequence, e.g., 695, 300 and 400, respectively for the three ATB3 experiments.

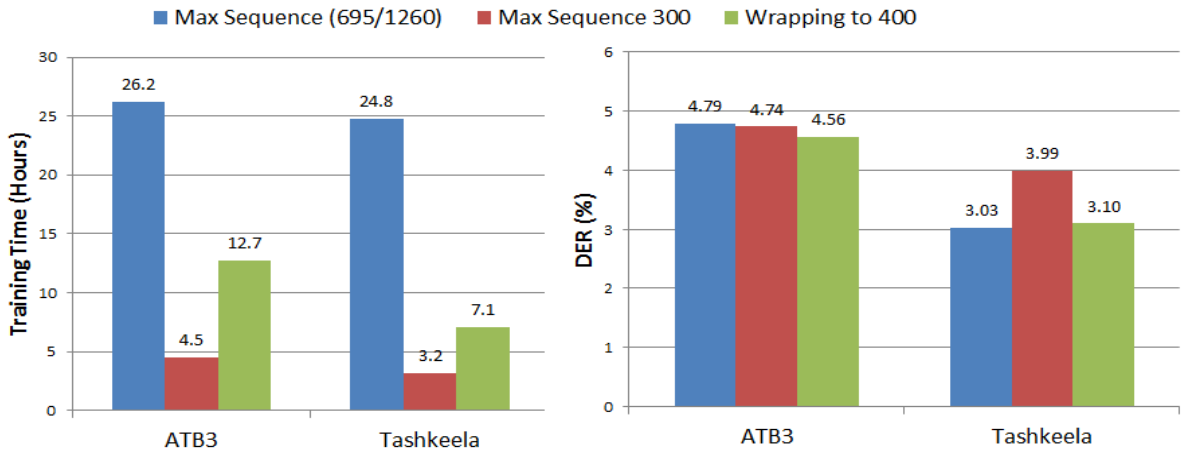

Figure 5. Training time and DER for ATB3 and Tashkeela datasets when including: (a) Maximum sequence length (695 for ATB3 and 1260 for Tashkeela), (b) Sequences not longer than 300 and (c) All sequences and wrapping long sequences to a maximum of 400 characters.

As expected, the execution time of the long sequences experiment is the highest: 26.2 hours for ATB3 and 24.8 hours for Tashkeela. However, including only sequences shorter that 300 adversely affects Tashkeela accuracy (a DER of 3.99%). In fact, the best DER for Tashkeela is obtained when all sequences are included (3.03% DER). Wrapping sequences to 400 characters is a good compromise; it gives reasonable training times of 12.7 hours for ATB3 and 7.1 hours for Tashkeela. At the same time, obtained DER values using wrapping is very close to the best DER. These are 4.56% for ATB3 and 3.10% for Tashkeela. We use this third method in the following experiments. Notice that Tashkeela achieves lower DER than ATB3. Tashkeela always achieves better accuracy than ATB3, because its

training set is larger and its diacritized letters ratio is higher.

5.2 Data Encoding Methods

Figure 6 shows the results of experiments using the three one-to-one target sequences encoding methods: many classes encoding, diacritics only encoding and multiple label encoding. For both datasets, encoding using diacritics only achieves the best results with DER of 4.83% for ATB3 and 3.10% for Tashkeela. The new diacritics only encoding is consistently better than the many classes encoding. It simplifies the network function from predicting letters and diacritics to predicting the diacritics only. We first had high expectations for multiple label encoding, because it exposes the contextual significance of the diacritics. However, this manual split of diacritics to multiple labels turned out to be less efficient than machine learning with one-hot encoding.

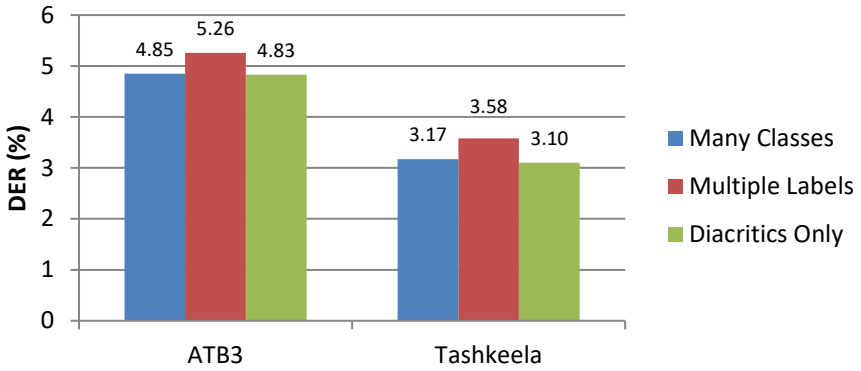

Figure 6. DER for ATB3 and Tashkeela using one-to-one encoding methods: many classes, multiple labels and diacritics only.

5.3 Network Architecture

This subsection presents the results of experimenting with three network architectures. The tested architectures are the encoder/decoder model, unidirectional LSTM and bidirectional LSTM. The tested encoder/decoder network consists of two encoder layers and two decoder layers. We limit the sequences lengths to 100 characters for this model, because longer sequences do not converge to useful output. Notice that for this set of experiments, we use the validation accuracy to evaluate the three network architectures, because the DER cannot be calculated for the encoder/decoder network. This network often outputs sequences that are not only wrong in diacritics, but also are wrong in the letter sequence output, making the DER calculation impossible. Figure 7 shows the validation accuracy of the three tested architectures. The encoder/decoder architecture has the worst validation accuracy among the three architectures. Moreover, as expected, unidirectional LSTM has inferior performance compared to bidirectional LSTM, since diacritization of a word depends on future context as well as on past context. Bidirectional LSTM is better than unidirectional LSTM by at least 11%.

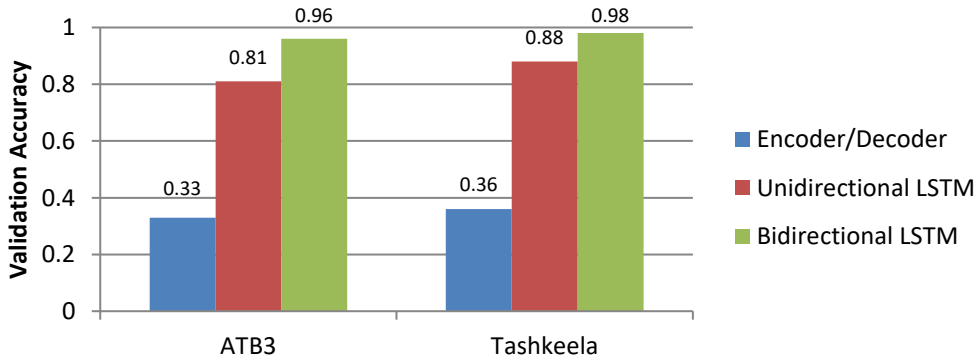

Figure 7. Accuracy of the validation set for ATB3 and Tashkeela using three network architectures.

Although there is a recent work that used encoder/decoder architecture to add the diacritics [24], we do not recommend it. Bidirectional LSTM gives better results without the trouble of employing sophisticated techniques, such as sliding windows, voting and attention.

5.4 Network Size

Figure 8 shows the results of changing the network size by varying the number of bidirectional LSTM layers. As expected, increasing the number of layers increases the time required to train the network. However, a deeper network with four layers achieves better accuracy than fewer layers; a DER of 4.19% for ATB3 and 2.83% for Tashkeela.

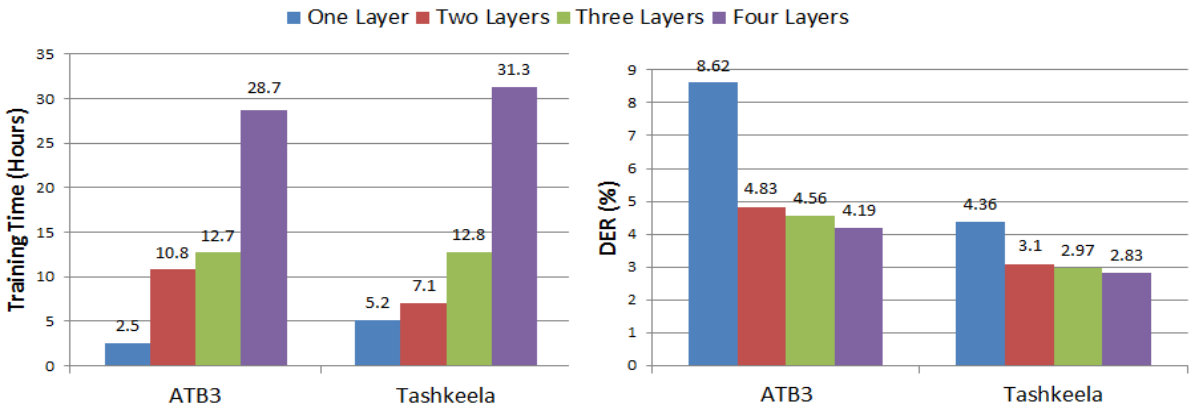

Figure 8. Training time and DER when varying network size from one layer to four layers.

5.5 Influence of Dropout

We used dropout regularization to overcome training overfitting [28]. We used grid search to find the best dropout option. Best results are obtained when a dropout rate of 0.1 is used for the layer input and a dropout rate of 0.3 is used for the recurrent state. Figure 9 shows the result of applying this dropout to the network with varying number of layers. The results show that applying dropout improves the accuracy, achieving a DER of 3.19% for ATB3 and 2.03% for Tashkeela with a network of four layers. We experimented going for a deeper network of five layers when dropout is used. Results did not show improvement in the obtained DER for both ATB3 and Tashkeela, as shown in Figure 9.

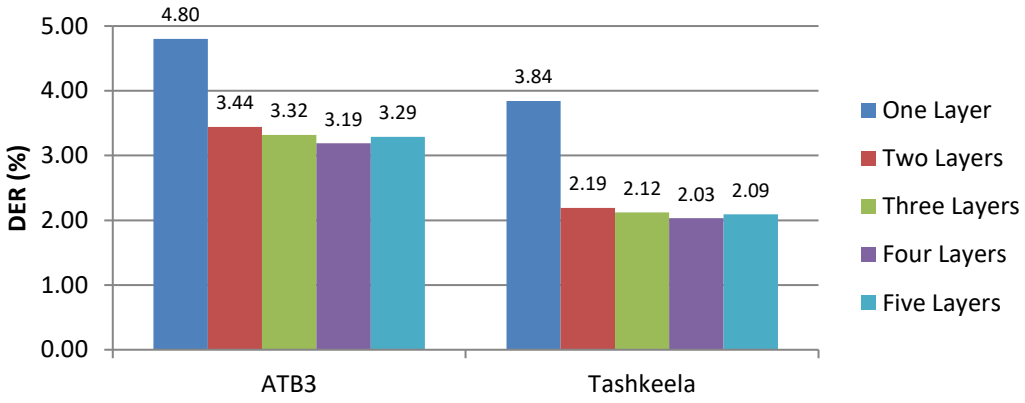

Figure 9. DER of five network sizes with dropout.

6. Discussion

Our best results are reported here using four bidirectional LSTM layers with dropout, diacritics only encoding of the target sequences and wrapping long input sequences to 400 characters. The following three subsections compare the results of this work with previous work, analyze the output diacritization errors and summarize other studied model hyper-parameters in this work.

6.1 Comparison with Existing Systems

Table 9 summarizes the comparison of this work and previous work. With the post processing techniques proposed in our previous work [4], the best DER and WER are 2.46% and 8.12%, respectively, for ATB3. This improves over our previous work that previously reported a DER of 2.7% for ATB3. For Tashkeela, the best DER and WER are 1.97% and 5.13%, respectively. This provides 47% DER and 54% WER improvement over the best-reported DER of the Shakkala framework tested by Fadel et al. [7]. In addition, Table 9 shows DER and WER when errors in diacritizing the last letter of each word are ignored. Especially for ATB3, error rates significantly improve when these errors are ignored. Last letter diacritization depends on the context and hence is considered more difficult than diacritizing other letters. The last column in Table 9 shows DER resulting from last-letter diacritization errors only. For

both ATB3 and Tashkeela, our model provides better last-letter diacritization error rates compared to other systems.

Table 9. Comparison with previous work on LDC ATB3 and Tashkeela (TKL) datasets

System	*All Diacritics*				*Ignore Last*				*DER Last*	
	ATB3		*TKL*		*ATB3*		*TKL*		*ATB3*	*TKL*
	DER	*WER*	*DER*	*WER*	*DER*	*WER*	*DER*	*WER*	*DER*	*DER*
Zitouni et al. (2006) [17]	5.5	18	-	-	2.5	7.9	-	-	3.0	-
Habash & Rambow (2007) [5]	4.8	14.9	-	-	2.2	5.5	-	-	2.6	-
Rashwan et al. (2011) [18]	3.8	12.5	-	-	**1.2**	**3.1**	-	-	2.6	-
Said et al. (2013) [19]	3.6	11.4	-	-	1.6	4.4	-	-	2.0	-
Abandah et al. (2015) [4]	2.7	9.1	-	-	1.38	4.34	-	-	1.34	-
Fadel et al. (2019) [7]	-	-	3.73	11.19	-	-	2.88	6.53	-	0.85
This work	**2.46**	**8.12**	**1.97**	**5.13**	1.24	3.81	**1.22**	**3.13**	**1.22**	**0.75**

6.2 Diacritization Error Analysis

We analyze errors of our system by tallying the errors according to the number of errors per word and the presence of last-letter diacritization error. The results of this analysis for ATB3 and Tashkeela are shown in Table 10. The table shows that most of the miss-diacritized words have one diacritic error at 79.4% and 74.3% for ATB3 and Tashkeela, respectively. Words with three or more diacritic errors are not frequent at 4.8% and 4.4% for ATB3 and Tashkeela, respectively. The table also shows that in ATB3, 62.8% of word errors include an error in last letter diacritic. Tashkeela has a smaller ratio of these errors (49.6%), because Tashkeela is a larger dataset and has a lower ratio of missing diacritics (see Table 5). Also, notice that when there is an error in the last letter diacritics, the error distribution tail is longer reaching, e.g., 0.5% for four errors or more *versus* 0.2% or 0.1% when the last letter is OK.

Table 10. Distribution of word errors in percent (%).

Dataset	*Errors per word*	*One*	*Two*	*Three*	*Four +*	*Total*
ATB3	**Last letter OK**	26.3	9.1	1.6	0.2	37.2
	Error in last letter	53.1	6.6	2.5	0.5	62.8
	Total	79.4	15.7	4.1	0.7	100.0
Tashkeela	**Last letter OK**	35.2	13.5	1.6	0.1	50.4
	Error in last letter	39.1	7.8	2.2	0.5	49.6
	Total	74.3	21.3	3.8	0.6	100.0

We have inspected 200 diacritization error samples of the ATB3 test set. For these samples, we analyzed the sources of the errors in the last-letter and other letters (internal) diacritics. Additionally, we report ratios of some specific error types, such as errors in words that have *Shadda*, errors that are not harmful and errors in composite words. Table 11 shows the results of this analysis.

Almost three quarters of the errors in end word diacritics are due to incorrect prediction by the proposed model. For the example output sentence shown in the table, there is no reason for the underlined miss-diacritized word خُطُورَةٍ to have *Kasratan* ٍ instead of Fathatan ً. Another source of errors is having undiacritized letters in the training and testing sequences. Having undiacritized letters in the training set leads to a model that does not diacritize some letters (13% of the selected samples). Target sequences with undiacritized letters are responsible of 8% of the sample errors. The underlined word المجد of the

third example target sentence is not diacritized, whereas the output word المَجْدِ is correctly diacritized. The rest end-word diacritic errors (6%) are due to not having enough context for the model to predict the last-letter diacritics. For example, our model fails to diacritize the last letter of the word السبت, since it comes alone, not included in a proper sentence.

Table 11. Analysis of sample errors.

Criteria	*Ratio*	*Examples*	
		Target	***Output***
End word diacritics			
Incorrect prediction	73%	إِنَّنا نُواجِهُ هَجْمَةً أَكْثَرَ خُطُورَةً	إِنَّنا نُواجِهُ هَجْمَةً أَكْثَرَ خُطُورَةٍ
Not diacritized	13%	الَّذِي رَأَى أَوَّلَ مِن أَمْسِ	الَّذِي رَأَى أَوَّلَ مِن أَمْس
Target error	8%	بَيْنَما سَجَّلَ إِصابَةَ المجد	بَيْنَما سَجَّلَ إِصابَةَ المَجْدِ
Not enough context	6%	السَّبْتُ	السَّبْت
Internal diacritics			
Incorrect prediction	35%	إِلَى عَشْرِ كُراتٍ مُرْتَدَّةٍ	إِلَى عَشْرِ كُرّاتٍ مُرْتَدَّةٍ
Valid word	31%	وَلَوْ كُنّا نُسَلِّمُ بِهذا الأَمْرِ	وَلَوْ كُنّا نُسَلَّمَ بِهذا الأَمْرِ
Name word	18%	وَانْتَقَلَ بَعْدَ ذلِكَ مِن بيناروول إِلَى ميلان	وَانْتَقَلَ بَعْدَ ذلِكَ مِن بينارُول إِلَى ميلانٍ
Possible alternative	7%	مَعْلُوماتٍ تُدَعِّمُ حُجَجَ بُوش	مَعْلُوماتٍ تَدْعَمُ حُجَجَ بُوش
Target error	6%	عادَ مُجَدِّداً إِلَى تكتل لِيكُود	عادَ مُجَدِّداً إِلَى تَكَتُّلِ لِيكُود
Not diacritized	3%	وَمِن نَفْسِ مادَّةِ الدِّينامِيت	وَمِن نَفْسِ مادَّةِ الدِّينامِيت
Error types			
Has *Shadda*	23%	يُجَنِّبُهُ ضَرْبَةً عَسْكَرِيَّةً	يُجَنِّبُهُ ضَرْبَةً عَسْكَرِيَّةً
Not harmful	21%	أَوْضَحَ بورعد	أَوْضَحَ بُورعد
In composite word	12%	حَصَرَ مَجْلِسُ الوُزَراءِ مُناقَشاتِهِ	حَصَرَ مَجْلِسُ الوُزَراءِ مُناقَشاتِهُ

For the diacritization errors in the internal letters, incorrect prediction that gives invalid words is at 35% of the samples. In 31% of the cases, the proposed model produces a diacritization output that gives a valid Arabic word, but the output word is not suitable for the context. For example, the word نسلم was diacritized as نُسَلَّم (passive voice meaning "we will be given") instead of نُسَلِّمُ (active voice meaning "we accept"). In 7% of the cases, the model produces a diacritization output that is a correct diacritization alternative, but is different from the target diacritics. For example, the model diacritizes the word تدعم as تَدْعَمُ instead of تُدَعِّمُ and both words provide the same meaning of "it supports". Other internal letter errors are in words that represent names (18%). Diacritizing foreign names such as بينارول (Club Atlético Peñarol) is hard, because they are often out-of-the-vocabulary and not diacritized in the dataset. The rest internal diacritics errors are due to lack of diacritization in the target or output sequences at 6% and 3%, respectively.

Of the selected samples, 23% of the errors are in words that have *Shadda*. Restoring diacritics of these words is more difficult, as the word diacritics tend to be more complex when *Shadda* is present. We observed also that 21% of the sample errors are not harmful, such that the miss-diacritized words can still be correctly read and understood. For example, the model diacritizes the name word بورعد as بُورعد. Adding the *Damma* in this case does not affect the word pronunciation. Finally, errors in composite words contribute 12% of the error samples. Predicting the diacritics of composite words that have prefixes and/or suffixes is harder than that of simple words. For example, in a composite word with a suffix, the inflection diacritic of the stem word is on the letter before the suffix, not on the last letter. In the last example of the table above, the model fails to retrieve the correct diacritic for the pronoun suffix ـه in the word مناقشاته by diacritizing it as مُناقَشاتِهُ instead of مُناقَشاتِهِ.

6.3 Other Experiments

In addition to the experiments reported in the previous sections, we performed other experiments for which results are not reported here, because they do not improve the accuracy of our model. These experiments included testing *Adam* optimizer instead of the RMSprop optimizer [28]. For all experiments, RMSprop performed better than Adam optimizer did. Moreover, we experimented with adding L1 and L2 regularization in lieu of dropout to overcome overfitting instead of dropout [28]. Results show that regularization does not improve training and even produces worse accuracy in some cases.

7. CONCLUSIONS

We performed intensive experiments to find a fast and accurate solution. Our experiments used the LDC ATB3 dataset as an example of MSA and a clean subset of Tashkeela dataset as an example of CA. Our experiments included studying the variation in sequence lengths of the used datasets. We experimented with handling this variation using different approaches and tested both the accuracy and training time. We recommend wrapping very long sequences to segments not longer than 400 letters. We also proposed two new encoding methods for target diacritized sequences. Our experiments show that the proposed encoding using diacritics only improves the accuracy, since it simplified the network output layer.

We tested different network architectures and the results show the superiority of the bidirectional LSTM network over the encoder/decoder network and the unidirectional LSTM. We also tuned our model by going for a deeper network and applying dropout. Our best results are reported for a bidirectional RNN LSTM with four layers that uses dropout. Best achieved DER is 2.46% and 1.97% for ATB3 and Tashkeela, respectively. Our best DER for Tashkeela provides an improvement of 47% over the best-published result.

The results of this work open doors for future work. The proposed dataset file sparse encoding, wrapping long sequences, efficient bidirectional deep LSTM and tuned hyper-parameters allow efficient training using large datasets. We intend to improve the accuracy of the proposed model based on the insights gained from the diacritization error analysis above. The diacritization accuracy should improve when we use larger MSA dataset. Note that Tashkeela has better accuracy and is larger than ATB3. Additionally, we need to solve the problem of having missing diacritics in some of training sequences. Such sequences confuse the network and result in some undiacritized output. Finally, as some diacritization differences between the output and the target sequences are not more harmful than other differences, we need to develop a better loss function that considers this issue when training the network.

REFERENCES

[1] N. Y. Habash, Introduction to Arabic Natural Language Processing, Synthesis Lectures on Human Language Technologies, Morgan and Claypool Publishers, 2010.

[2] G. Abandah, M. Khedher, W. Anati, A. Zghoul, S. Ababneh and M. Hattab, "The Arabic Language Status in the Jordanian Social Networking and Mobile Phone Communications," Proc. of the 7th Int'l Conference on Information Technology (ICIT 2015), pp. 449-456, 2015.

[3] G. A. Abandah and F. Khundakjie, "Issues Concerning Code System for Arabic Letters," Dirasat-Eng. Sci. J., vol. 31, no. 1, pp. 165-177, 2004.

[4] G. A. Abandah, A. Graves, B. Al-Shagoor, A. Arabiyat, F. Jamour and M. Al-Taee, "Automatic Diacritization of Arabic Text Using Recurrent Neural Networks," International Journal on Document Analysis and Recognition (IJDAR), vol. 18, no. 2, pp. 183-197, 2015.

[5] N. Habash and O. Rambow, "Arabic Diacritization through Full Morphological Tagging," Proc. of Conference on North American Chapter of the Association for Computational Linguistics, pp. 53-56, 2007.

[6] M. Maamouri, A. Bies, T. Buckwalter and W. Mekki, "The Penn Arabic Treebank: Building a Large-scale Annotated Arabic Corpus," Proc. of Conference on Arabic Language Resources and Tools (NEMLAR), pp. 102-109, 2004.

[7] A. Fadel, I. Tuffaha, B. Al-Jawarneh and M. Al-Ayyoub, "Arabic Text Diacritization Using Deep Neural Networks," arXiv: 1905.01965v1, 2019.

[8] A. M. Azmi and R. S. Almajed, "A Survey of Automatic Arabic Diacritization Techniques," Natural Language Engineering, vol. 21, pp. 477-495, 2013.

[9] O. Hamed and T. Zesch, "A Survey and Comparative Study of Arabic Diacritization Tools," JLCL: Special Issue-NLP for Perso-Arabic Alphabets, vol. 32, no. 1, pp. 27-47, 2017.

[10] Y. Gal, "An HMM Approach to Vowel Restoration in Arabic and Hebrew," Proceedings of the ACL-02 Workshop on Computational Approach to Semitic Languages (SEMITIC '02), pp. 27-33, 2002.

[11] E. Elshafei, H. Al-Muhtaseb and M. Alghamdi, "Statistical Methods for Automatic Diacritization of Arabic Text," Proceedings of Saudi 18th National Computer Conference (NCC18), pp. 301-306, 2006.

[12] Y. Hifny, "Smoothing Techniques for Arabic Diacritics Restoration," Proceedings of the 12th Conference on Language Engineering (ESOLEC '012), pp. 6-12, 2012.

[13] T. Zerrouki, "Arabic Corpora Resources, Tashkila Collection from the Arabic Al-Shamela Library, [Online], Available: "http://aracorpus.e3rab.com/, [Accessed Aug. 27, 2019].

[14] A. S. Azim, X. Wang and K. C. Sim, "A Weighted Combination of Speech with Text-based Models for Arabic Diacritization," Proceedings of the 13th Annual Conference of International Speech Communication Association, pp. 2334-2337, 2012.

[15] D. Vergyri and K. Kirchhoff, "Automatic Diacritization of Arabic for Acoustic Modelling in Speech Recognition," Proceedings of the Workshop on Computational Approaches to Arabic Script-based Languages, pp. 66-73, 2004.

[16] R. Nelken and S. M. Shieber, "Arabic Diacritization Using Weighted Finite-state Transducers," Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages, pp. 79-85, 2005.

[17] I. Zitouni, J. S. Sorensen and R. Sarikaya, "Maximum Entropy-based Restoration of Arabic Diacritics," Proceedings of the 21st International Conference on Computational Linguistics, pp. 577-584, 2006.

[18] M. Rashwan, M. Al-Badrashiny, M. Attia, S. Abdou and A. Rafea, "A Stochastic Arabic Diacritizer Based on a Hybrid of Factorized and Unfactorized Textual Features IEEE/ACM Transactions on Audio, Speech and Language Processing, vol. 19, no. 1, pp. 166-175, 2011.

[19] A. Said, M. El-Sharqwi, A. Chalabi and E. Kamal, "A Hybrid Approach for Arabic Diacritization," In: E. Mtai, F. Mezaine, M. Saraee, V. Sugumaran and S. Vadera (Eds.), "Natural Language Processing and Information Systems," Lecture Notes in Computer Science, vol. 7934, pp. 53-64, Springer, 2013.

[20] S. Alquda, G. Abandah and A. Arabiyat, "Investigating Hybrid Approaches for Arabic Text Diacritization with Recurrent Neural Networks," Proceedings of the 2017 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT), pp. 1-6, 2017.

[21] M. Rashwan, A. Sallab, H. Raafat and A. Rafea, "Deep Learning Framework with Confused Sub-set Resolution Architecture for Automatic Arabic Diacritization," Proceedings of the IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP), vol. 23, no. 3, pp. 505-516, 2015.

[22] A. Barqawi and T. Zerrouki, "Shakkala, Arabic Text Vocalization," [Online], Available: https://github.com/Barqawiz/Shakkala, 2017.

[23] Tahadz, "Mishkal," [Online], Available: https://tahadz.com/mishkal, [Accessed on October 16, 2019].

[24] H. Mubarak, A. Abdelali, H. Sajjad, Y. Samih and K. Darwish, "Highly Effective Arabic Diacritization Using Sequence-to-Sequence Modeling," Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, vol. 1, pp. 2390-2395, 2019.

[25] K. Darwish, H. Mubarak and A. Abdelali, "Arabic Diacritization: Stats, Rules and Hacks," Proceedings of the 3rd Arabic Natural Language Processing Workshop, pp. 9-17, 2017.

[26] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence-to-Sequence Learning with Neural Networks," Advances in Neural Information Processing Systems (NIPS), arXiv: 1409.3215v3, 2014.

[27] A. Graves, A. R. Mohamed and G. Hinton, "Speech Recognition with Deep Recurrent Neural Networks," Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6645-6649, 2013.

[28] A. Geron, Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools and Techniques to Build Intelligent Systems, USA: O'Reilly, 2017.

[29] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997.

[30] K. Cho, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, "Learning Phase Representations Using RNN Encoder-Decoder for Statistical Machine Translation," arXiv: 1406.1078v3, 2014.

[31] M. Schuster and K. K. Paliwal, "Bidirectional Recurrent Neural Networks," IEEE Transactions on Signal Processing, vol. 45, no. 11, pp. 2673-2681, 1997.

[32] Google, "TensorFlow," [Online], Available: https://www.tensorflow.org/, [Accessed on Aug. 27, 2019].

ملخص البحث:

تُكتب النصوص بالعربية اليوم دون وضع الحركات على الأحرف، علماً بأنّ وضع هذه الحركات من شأنه أن يقلل من الغموض في قراءة النص، الى جانب فوائد أخرى. يهدف هذا العمل البحثي الى تطوير حلّ دقيق وسريع باستخدام تعلُّم الآلة من أجل وضع الحركات على أحرف النّصّ أوتوماتيكياً. يستخدم هذا البحث الشّبكات العصبيّة المتكرّرة اعتماداً على الذاكرة الطويلة قصيرة الأمد لوضع الحركات على أحرف النّصوص المكتوبة بالعربية.

تم إجراء تجارب مكثّفة لتقييم التّصميم المقترح وخيارات ترميز البيانات للحصول على حلّ دقيق وسريع. وتتضمن تلك التجارب البحث في المشكلات المتعلقة بطول التّتابُع، واقتراح وتقييم خيارات التّرميز للمُخرجات التي تم وضع الحركات على أحرف النّصّ فيها، وضبط الخيارات المتعلقة بالشبكات العصبية وتقييمها من حيث البِنية وحجم الشبكة والمتغيرات العُليا.

وتوصي هذه الدراسة بحل يمكن تدريبُهُ بسرعةٍ على مجموعة بياناتٍ ضخمة باستخدام طبقات ذاكرة طويلة قصيرة الأمد ثنائية الاتّجاه عددها (4) طبقات؛ من أجل توقُّع الحركات على أحرف تتابع المدخل لنصوص اللُّغة العربية. وتميَّز الحلّ المقترح بنسبة خطأ في وضع الحركات على الأحرف مقدارها 2.46% و 1.97% عند تطبيقه على مجمةعة البيانات (LDC ATB3) ومجموعة البيانات الجديدة الأضخم المعروفة باسم (تشكيلة)، على الترتيب. وهذه النسبة الأخيرة تمثّل تحسيناً بنسبة 47% مقارنة بأفضل النتائج المنشورة سابقاً.