Compact and Efficient WFST-based Decoders for Handwriting Recognition

Meng Cai and Qiang Huo
Microsoft Research Asia, Beijing, China
Email: {meng.cai, qianghuo}@microsoft.com

***Abstract*—We present two weighted finite-state transducer (WFST) based decoders for handwriting recognition. One decoder is a cloud-based solution that is both compact and efficient. The other is a device-based solution that has a small memory footprint. A compact WFST data structure is proposed for the cloud-based decoder. There are no output labels stored on transitions of the compact WFST. A decoder based on the compact WFST data structure produces the same result with significantly less footprint compared with a decoder based on the corresponding standard WFST. For the device-based decoder, on-the-fly language model rescoring is performed to reduce footprint. Careful engineering methods, such as WFST weight quantization, token and data type refinement, are also explored. When using a language model containing 600,000 *n*-grams, the cloud-based decoder achieves an average decoding time of 4.04 ms per text line with a peak footprint of 114.4 MB, while the device-based decoder achieves an average decoding time of 13.47 ms per text line with a peak footprint of 31.6 MB.**

I. Introduction

The framework of a state-of-the-art segmentation-free handwriting recognition (HWR) system often includes a character model, a language model, a lexicon and a decoder, which is similar to an automatic speech recognition (ASR) system (e.g., [1]). The decoder is a crucial part in this framework. A decoder generates recognition result by computing the best path in a decoding network. There has been extensive studies in decoders in speech recognition (e.g., [2], [3], [4], [5], [6], [7], [8], [9], [10]). Different decoding algorithms and decoder implementations have a great impact on the speed and memory footprint of a practical HWR system.

A highly successful technique widely used in decoders is the weighted finite-state transducer (WFST) [7]. A WFST is a directed graph consisting of a set of states and a set of transitions (arcs) connecting the states. The WFST provides a unified representation of language model, lexicon, and the topology of character model. Each of the components can be compiled into a WFST format. After the WFSTs of the components are generated, the individual WFSTs are often composed to generate a single WFST. Optimization operations such as determinization and minimization [7] are applied to obtain the final WFST. This final WFST is used as the decoding network by the HWR decoder.

By the definition of a WFST, there are four attributes stored on each WFST transition. The four attributes are an input label, an output label, a weight and a pointer to the next state. For the WFST decoding network of an English HWR system, the input labels are characters and the output labels are words. The weight on each WFST transition is the composed cost of language model score, lexicon score and transition score in the character model. Some literatures also refer to the weights on the WFST transitions as the graph costs [8]. The actual cost used during HWR decoding is a weighted sum of the graph cost and the character cost. The character cost is typically the negative log likelihood produced by the character model. The decoder tries to find the path with the lowest cost using Viterbi search with pruning. After the best path is found, the sequence of the output labels on the WFST transitions are used as the decoding result.

The advantage of using WFST for decoding is that searching on a single, pre-compiled and optimized decoding network can be efficient and simple. However, WFST-based decoding may result in big memory consumption because the WFST is a fully-expanded decoding network. As a result, reducing the memory footprint while maintaining the efficiency of WFST-based decoders is important for a practical HWR system.

Attempts have been made previously to reduce the footprint of WFST-based decoders. The most well-known method is WFST-based decoding with on-the-fly language model rescoring [9], [10]. In this method, the WFST decoding network is built with a small language model. The decoder loads the small WFST and performs on-the-fly language model rescoring using a normal language model. The footprint of the small WFST decoding network plus the language model is significantly smaller than the single WFST built with the normal language model, therefore the overall footprint of the decoder is reduced. With careful engineering, ASR decoding is even made possible on mobile devices (e.g., [11], [12]). The on-the-fly language model rescoring method is suitable for device-based HWR system. However, performing on-the-fly language model rescoring inevitably requires some extra computation compared with decoding on a single graph. Thus the method of decoding on a single graph should be considered for cloud-based HWR system. Moreover, an HWR task has some unique properties that are different from an ASR task, which may enable some specific optimizations for the design of the HWR decoders.

In this paper, we present two decoders for HWR. The first decoder is suitable for deployment on the cloud. This decoder uses a single decoding network to maximize the efficiency. We discover in this work that a compact WFST data structure is suitable for HWR task. Each transition in the compact WFST data structure only contains an input label, a

2379-2140/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDAR.2017.32

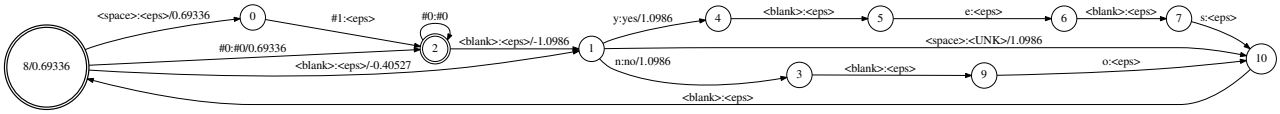

Fig. 1. Illustration of a lexicon WFST with an optional <space> symbol. This WFST only contains words yes and no. <UNK> symbol is for out-of-vocabulary (OOV) words. Epsilon symbol, <eps>, stands for no input or no output. Symbols #0 and #1 are disambiguation symbols for subsequent determinization operation.

weight and a pointer to the next state. The cloud-based HWR decoder is built upon the compact WFST data structure and the corresponding algorithms. Several techniques to speed-up the cloud-based decoders are also explored. The second decoder is suitable for deployment on devices with constrained memory resource. It uses on-the-fly language model rescoring in order to reduce footprint. It also uses other footprint reduction methods such as WFST weight quantization, token and data type refinement.

The remainder of this paper is organized as follows. In Section II, a brief description of an HWR system is presented. Section III and Section IV discuss the cloud-based decoder and the device-based decoder, respectively. Detailed experiments are given in Section V. Finally Section VI concludes the paper.

II. Overview of an HWR System

Our HWR baseline system is based on a deep bidirectional long short-term memory (DBLSTM) recurrent neural network (RNN) character model and an *n*-gram language model [13], [14], [15], [16]. The character model is trained using the connectionist temporal classification (CTC) objective function [17] and a stochastic gradient descent (SGD) optimization method. The input of our DBLSTM-CTC model is a sequence of PCA feature vectors extracted from each text line [14], [15], while its output consists of normal characters and an extra <blank> symbol. The <blank> symbols occur between two same characters and optionally occur between two different characters in each training label sequence. Because of the use of <blank> symbol and CTC training, the softmax output of the DBLSTM-CTC model produces sharp spikes for promising characters. This phenomenon can speed-up decoding as pruning strategies can be effectively used for Viterbi search.

WFST generation follows standard recipe as described in e.g. [7], [8]. The overall procedure is summarized as

$$S = min(det(H \circ C \circ L \circ G)), \tag{1}$$

where L and G are lexicon and language model WFSTs, respectively. These two WFSTs are compiled using the standard method [7], [8]. C is a WFST modeling context dependency in the character model [18]. As context dependency is not explicitly modeled in our HWR task, C is a trivial WFST with the same input label and output label on each transition. H is a WFST for the hidden Markov model (HMM). Our character model is viewed as an HMM with a single state for each character, where its transition and self-loop probabilities are both set to 0.5. The symbols, $\circ$, $det()$, and $min()$ stand for WFST composition, WFST determinization, and WFST minimization, respectively. In our HWR lexicon, <blank> symbol is inserted between characters in each lexical item to match the CTC criterion. We optionally attach a <space> symbol to the tail of every lexical item in order to model the space between two words. An illustration of a WFST L is given in Fig. 1. The insertion of <space> symbols enables important optimizations in this work, which is presented in detail in Subsection III-A.

III. HWR Decoder for Cloud Deployment

HWR decoders can be deployed in either cloud-based or device-based ways. For cloud-based deployment, the decoder runs on powerful servers, therefore it is usually designed to minimize runtime latency. However, memory footprint is still an important issue even for cloud-based decoders as smaller decoder footprint reduces deployment cost. The method of decoding on a single WFST is chosen for our cloud-based HWR decoder. In this section, we first propose a compact WFST data structure designed for HWR. The proposed method can reduce memory footprint significantly compared with the standard method, while producing the same decoding result. We then give some detailed optimizations to further speed-up the cloud-based decoder.

A. Compact WFST Data Structure

According to Eq. (1), the output labels of the decoding network S are words and the input labels of S are characters. In a standard decoding scheme for HWR or ASR, the output label sequence on the transitions of the best path is the decoding result, while the input label sequence is the alignment. The alignment is used to calculate the word boundaries together with the word sequence. Recording the WFST output label sequence for ASR is necessary for two reasons. First, the lexicon of speech recognition may contain word items with the same pronunciation, making it impossible to get the word sequence using the alignment. Second, the alignment of the ASR decoding result does not necessarily contain a silence phoneme between two adjacent words, resulting in ambiguity of word boundaries if only the alignment is given. However, the two issues do not exist for HWR. The concatenation of characters naturally forms each word in an HWR lexicon, and there is always a space between two words in a text line for languages such as English. As a result, the word sequence in an HWR task can be completely recovered using the decoding alignment, as long as the <space> symbol is modeled in the WFST decoding network.

Based on the above analysis, it is not necessary to record the word sequence for the HWR decoding. It is neither necessary

Algorithm 1 Compact WFST for HWR decoding.

1: Compile the lexicon WFST L with the optional <space> symbol as shown in Fig. 1;
2: Build the standard WFST decoding network S by composition and optimization according to Eq. (1);
3: Remove the output labels on the transitions of S to generate the compact WFST P;
4: Perform Viterbi search with pruning on the compact WFST P to compute the best path;
5: Record the alignment sequence $\mathbf{A}$ for the best path;
6: Split the alignment sequence $\mathbf{A}$ using the <space> symbols to generate n sub-sequences;
7: Apply the CTC mapping function [17] on each of the n sub-sequences to obtain n words, forming the word sequence $\mathbf{W}_n$;
8: **return** $\mathbf{W}_n$.

to store the output labels on the transitions of the WFST for HWR. The HWR decoder only needs to produce the alignment using the input labels on the transitions of the WFST. Then a post-processing step can be used to generate the word sequence using the alignment. We thus propose a data structure called **compact WFST** for HWR. Given a WFST decoding network, the compact WFST is generated by simply removing the output labels from the transitions. The corresponding algorithm for the compact WFST-based decoder is given in Algorithm 1.

As output labels are not stored on the transitions of the decoding network or the paths of Viterbi search, the compact WFST data structure reduces footprint compared with the standard WFST. Decoding speed is also slightly faster than the standard method because memory management cost is smaller.

B. Optimizations for Decoder Speed-up

Besides the compact WFST data structure, there are also several optimization strategies to speed-up the cloud-based HWR decoder, which are presented as follows:

1) Decoding network storage: The decoding network is stored as three arrays in memory. The first array contains the non-epsilon transitions of the compact WFST. Each non-epsilon transition stores the input label, the weight and the pointer to the next state. The second array contains the epsilon transitions of the compact WFST. Each epsilon transition stores the weight and the pointer to the next state. The third array contains the states of the compact WFST. Two pointers are stored in each state. One points to the first non-epsilon transition from that state and the other points to the first epsilon transition from that state. This structure is similar to the one described in [19] except that we deal with the non-epsilon transitions and the epsilon transitions separately.

2) An exhaustive hash table: During Viterbi search, a hash table is typically needed to determine whether a next state has been pointed by any of the current transitions. The key of the hash table is the state index of the compact WFST and the value of the hash table is the pointer to the current token. It is noted that a token records a path in Viterbi algorithm as described in [20]. The time complexity of searching an element in a hash table is $\mathcal{O}(1)$. But the time complexity of inserting an element in a hash table is not $\mathcal{O}(1)$, which degrades the performance. We avoid using conventional hash tables in the cloud-based HWR decoder. In our method, an array of flags and an array of pointers are created, whose numbers of elements both equal to the number of states in the decoding network. Inserting an element is equivalent to setting the corresponding flag to `true` and recording the pointer of the token. Searching an element is equivalent to checking whether the flag is `true` and fetching the pointer of the token if available. In this way, the time complexity of inserting an element or searching an element are both $\mathcal{O}(1)$. We call this data structure an **exhaustive hash table** in this paper. The exhaustive hash table is possible for the cloud-based HWR decoder when we do not use a huge language model thus the WFST only has several million states.

3) Pruning strategy: The pruning strategy of the HWR decoder is a combination of two pruning methods: the standard beam pruning and the histogram pruning. For the beam pruning, the decoder always records the current best path ρ_{opt} for each frame. A path ρ is pruned if the cost between ρ and ρ_{opt} is greater than a beam threshold η. The histogram pruning is applied after beam pruning to control the maximum number of active paths at each frame. This pruning strategy is also similar to that in [19].

IV. HWR Decoder for On-Device Deployment

Besides cloud-based HWR, another important scenario is HWR on devices with limited memory. The footprint of a device-based decoder is thus much more critical than the cloud-based decoder. We present our device-based HWR decoder in this section, which is designed to minimize memory footprint. We discuss on-the-fly language model rescoring method first and several footprint reduction methods next.

A. On-the-fly Language Model Rescoring

The WFST S built by Eq. (1) is optimized and fully-expanded. Decoding on a single WFST S can be viewed as a method of trading space for time. The size of the WFST S is dominated by the size of the language model G. To reduce the footprint of the device-based HWR decoder, a straightforward idea is to compose the WFST $N = min(det(H \circ C \circ L))$ and apply the language model scores of G dynamically during decoding. However, removing the language model scores from the decoding network hurts the effectiveness of pruning [21]. A better trade-off and practical solution is to compose a small language model into the decoding network, while use a normal language model for on-the-fly rescoring.

The equivalent decoding network of the on-the-fly language model rescoring method is as follows:

$$S_{\text{otf}} = min(det(H \circ C \circ L \circ G_s)) \circ (G_s^{-1} \circ G_n), \quad (2)$$

where G_s is the WFST of the small language model. The G_n is the WFST of the normal language model used for rescoring.

The G_s^{-1} is the WFST that has the same structure as G_s, but the weights on the transitions of G_s^{-1} are the opposite numbers of the weights on the corresponding transitions of G_s. The purpose of G_s^{-1} is to compensate for the score of G_s so that the decoding network S_{otf} is equivalent to S in Eq. (1).

The decoding network S_{otf} is never really generated in a practical decoder implementation. Instead, the WFST H, C, L and G_s are composed and optimized off-line to generate a small decoding network S'. The language model scores of G_n are applied when processing the non-epsilon transitions during Viterbi search. The pseudocode of the on-the-fly language model rescoring when processing the non-epsilon transitions for a single frame is given in Algorithm 2. The s represents a state in the small decoding network S'. The $E(s)$ is the set of transitions from the state s. The symbols $i[e]$, $o[e]$, $w[e]$ and $n[e]$ are the input label, the output label, the weight and the next state of the transition e. The symbol $\otimes$ means multiplication in the semiring, which is commonly used in the WFST representation [7]. We use log semiring [7] in the HWR task and the $\otimes$ is actually the addition. The $CMScore(data, i[e])$ stands for the scaled negative log likelihood produced by the character model. The function $PropagateLM(s_\lambda, o[e])$ performs the on-the-fly language model score computation given the language model state s_λ and the output label $o[e]$. The output label $o[e]$ on the transition of the decoding network S' is also the input label on the transition from the state s_λ in the language model WFST G_n. If there is no transition from the state s_λ whose input label is $o[e]$, we follow the epsilon transition from s_λ to the next state $\hat{s}_\lambda$ and then search for the transition from the state $\hat{s}_\lambda$ that has the input label of $o[e]$. The epsilon transition in the language model WFST represents the back-off in the *n*-gram language model.

Algorithm 2 On-the-fly language model rescoring.

for all $\langle s, s_\lambda \rangle$ such that $\langle s, s_\lambda \rangle \in PreviousHash$ **do**
  $\Theta \leftarrow PreviousHash(\langle s, s_\lambda \rangle)$; // get previous token
  if Θ is pruned by the beam η **then**
    continue;
  end if
  for each $e \in E(s)$ such that $i[e] \neq \varepsilon$ **do**
    $\langle s'_\lambda, w_\lambda \rangle \leftarrow PropagateLM(s_\lambda, o[e])$;
    $w' \leftarrow \Theta.w \otimes w[e] \otimes w_\lambda \otimes CMScore(data, i[e])$;
    $\Theta' \leftarrow \langle pointer(\Theta), i[e], w' \rangle$; // the current token
    if Θ' is pruned by the beam η **then**
      continue;
    end if
    if $\langle n[e], s'_\lambda \rangle \in CurrentHash$ **then**
      if $w' < CurrentHash(\langle n[e], s'_\lambda \rangle).w$ **then**
        $CurrentHash(\langle n[e], s' \rangle) \leftarrow \Theta'$;
      end if
    else
      $CurrentHash.insert(\langle n[e], s'_\lambda \rangle, \Theta')$;
    end if
  end for
end for

B. Decoder Footprint Reduction

Several methods are explored to further reduce the footprint of the device-based HWR decoder. These methods include:

1) WFST weight quantization: Even for the HWR decoder with on-the-fly language model rescoring, the footprint of the WFST is still a major part of memory consumption. We try to quantize the weights on the transitions of the WFSTs to make the networks more compact. The WFST weights are mainly determined by the language model scores, which do not have a large dynamic range. We thus use a simple linear quantization method. The weights of 32-bit floating point numbers are scaled by a constant δ and then converted to 8-bit signed integers for the memory storage. These integers are converted back to floating point numbers in Viterbi search.

2) Token refinement: The information stored in the tokens during decoding is similar to the information stored in the transitions of the WFST. Typically an input label, an output label, an accumulated weight, a pointer to the next state and a pointer to the previous token are stored in a token. We argue that the output label is not needed to be stored in the token. The reason is the same as the principle of the compact WFST in Subsection III-A. The pointer to the next state is used for the transition iteration. It is also not needed if we always maintain two hash tables for the previous frame and the current frame respectively, as shown in Algorithm 2.

3) Data type refinement: For languages such as English, there are only about 100 characters to be modeled. So it is enough to use an 8-bit unsigned integer to encode all those characters for the input labels in the WFST transitions or the tokens. However, the WFST transitions or the tokens are typically implemented as structures. Simply replacing the data type from 32-bit integer to 8-bit integer may not be enough to reduce the footprint because of the structure alignment in computer memory. In conjunction with replacing the data type of the input labels to 8-bit integer, we also use several arrays of basic data types to replace an array of structure in the decoder implementation to achieve footprint reduction.

V. Experiments

A. Experimental Setup

We study the performance of our decoders on an offline English HWR task. Two data sets are used for evaluation: One is the benchmark IAM testing set [22], the other is an in-house testing set consisting of transcribed whiteboard and handwritten notes data. We refer to the in-house testing set as the E2E (end-to-end) set. There are 1,861 text lines in the IAM testing set and 4,028 text lines in the E2E testing set. For the DBLSTM-CTC character model, a training set containing about 283k text lines is used. The character model contains 5 hidden layers with 256 memory cells per layer. The CTC output layer has 96 nodes, representing all the 95 displayable ASCII characters plus the `<blank>` symbol. The text lines are normalized to have a height of 60 pixels. We use a sliding window of 30 pixels wide and the window shift is 3 pixels.

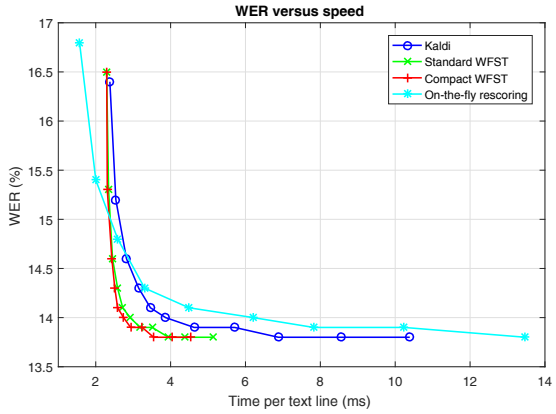

Fig. 2. Illustration of WER versus decoding speed for different decoders on IAM testing set.

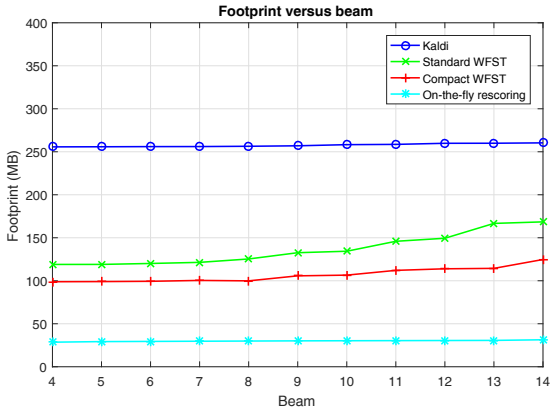

Fig. 3. Illustration of memory footprint versus decoding beam for different decoders on IAM testing set.

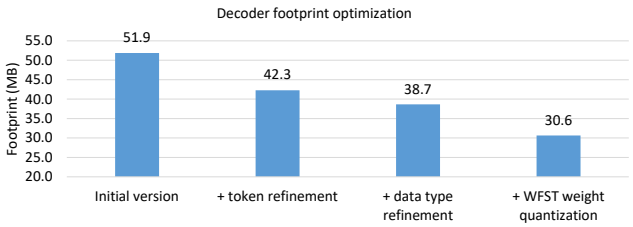

Fig. 4. Effects of applying different memory footprint reduction methods for device-based HWR decoders on IAM testing set.

The 50-dimensional PCA features are extracted for each frame of the text line image as the input to the character model.

A hybrid word-subword bigram language model with a lexicon of 131 thousand items is built to mitigate the out-of-vocabulary (OOV) word problem [16]. This model is trained on a large set of text corpora consisting of about 7 billion words. The text corpora include the standard LDC2008T15, LDC2011T07 and LDC2015T13 corpora plus some self-collected corpora. The language model contains about 600 thousand n-grams. The perplexities on the IAM and E2E testing sets are 466 and 526, respectively. The OOV word rates on the IAM and E2E testing sets are 0.34% and 1.18%, respectively. This language model is used for both cloud-based and device-based decoders to produce comparable results.

The decoder in the Kaldi open source toolkit [23] is employed as a baseline decoder. We compute character model scores off-line in order to only compare the performance of the beam search. A Linux server with a 2.8 GHz Intel Xeon E5-2680 v2 CPU is used for evaluations. All the decoders run on a single-thread mode. The recognition accuracy is measured by word error rate (WER).

B. Results of Cloud-based Decoder

The WFST decoding network built for the cloud-based decoder consists of 1,969,232 states and 5,224,414 transitions. The size of the WFST decoding network is 103 MB.

We first evaluate the speed of the decoders. By trying various beam settings, the relationship between the decoding speed and the WER is illustrated in Fig. 2. It is shown that both the cloud-based decoder with the standard WFST and the cloud-based decoder with the compact WFST are faster than the Kaldi decoder when producing the same accuracy. This is due to the speeding-up strategies in Subsection III-B and careful engineering efforts. The speed of the decoder with the compact WFST is slightly faster than the decoder with the standard WFST because of the smaller cost of memory management.

We also record the peak memory footprint of the decoders, as shown in Fig. 3. It is shown that the footprint of the decoder with the compact WFST is significantly less than the footprint of the decoder with the standard WFST when using the same beam. The memory footprint reduction demonstrates the effectiveness of the proposed method.

C. Results of Device-based Decoder

For the device-based decoder, we build the WFST decoding network with a unigram language model. This WFST has 357,824 states and 972,866 transitions. A bigram language model WFST is also built, which has 131,431 states and 708,463 transitions. The sizes of the WFST decoding network and the language model WFST are 19 MB and 13 MB, respectively. Both the weights in the WFST decoding network and the weights in the language model WFST are quantized. The scaling constant δ for the quantization is set to 4.

The overall speed and footprint of all the decoders are shown in Fig. 2 and Fig. 3. From the results in Fig. 2, we can see that the speed of the device-based decoder is slower than the baseline. This is because the language model rescoring requires additional computation compared with decoding on a single WFST. All the decoders achieve the same WER with a wide beam, suggesting that the WFST weight quantization does not incur any accuracy degradation. The results in Fig. 3 show that the footprint of the device-based decoder is much

TABLE I
COMPARISON OF DIFFERENT DECODERS ON IAM TESTING SET.

Decoder	WER (%)	Footprint (MB)	Speed (ms/line)
Kaldi	13.8	259.9	8.57
Cloud-based	13.8	114.4	4.04
Device-based	13.8	30.6	13.47

TABLE II
COMPARISON OF DIFFERENT DECODERS ON E2E TESTING SET.

Decoder	WER (%)	Footprint (MB)	Speed (ms/line)
Kaldi	19.3	259.5	3.78
Cloud-based	19.3	114.4	2.91
Device-based	19.3	31.6	6.34

smaller than other decoders. The small memory footprint enables the deployment of the HWR system on devices.

We further study the impact of different footprint reduction methods in Subsection IV-B. We create an initial version of the device-based HWR decoder without the footprint reduction method and then apply these methods. The beam of 13 is used. The footprint comparison is illustrated in Fig. 4. The results in Fig. 4 show that the footprint reduction methods are effective for the device-based decoder.

Finally, the performance of the decoders on the IAM and E2E testing sets are given in Table I and Table II, respectively. It is shown that different decoders achieve the same WERs but have quite different speeds and footprints. The cloud-based decoder has the maximum speed while the device-based decoder has the minimum footprint. The differences of the decoding speed for the E2E set are less significant than those for the IAM set because the text lines in the E2E set tend to be shorter on average.

VI. CONCLUSION

We have proposed a cloud-based decoder and a device-based decoder for HWR. The cloud-based decoder benefits from a compact WFST data structure to achieve reduced footprint and faster speed. The compact WFST and the corresponding algorithms designed for HWR have been presented in detail. The device-based decoder utilizes an on-the-fly language model rescoring technique to reduce footprint. Several engineering methods for footprint reduction have also been investigated. Evaluations on two testing sets demonstrate the effectiveness of the proposed methods. Exploring ways to further speed-up device-based HWR decoder will be our future work.

ACKNOWLEDGMENT

We would like to thank our Microsoft colleagues, Kai Chen, Wenping Hu, Lei Sun, Sen Liang and Xiongjian Mo for building the DBLSTM-CTC character model used in this study, and Zhenghao Wang and Huaming Wang for helpful discussions on WFST-based decoders.

REFERENCES

[1] T. Bluche, "Deep neural networks for large vocabulary handwritten text recognition," Ph.D. dissertation, Univ. Paris Sud - Paris XI, 2015.
[2] D. Nolden, H. Soltau, and H. Ney, "Progress in dynamic network decoding," in *Proc. ICASSP*, 2014, pp. 3276–3280.
[3] D. Rybach, R. Schluter, and H. Ney, "A comparative analysis of dynamic network decoding," in *Proc. ICASSP*, 2011, pp. 5184–5187.
[4] S. Ortmanns, H. Ney, and A. Eiden, "Language-model look-ahead for large vocabulary speech recognition," in *Proc. ICSLP*, 1996, pp. 2095–2098.
[5] H. Ney and S. Ortmanns, "Progress in dynamic programming search for lvcsr," *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1224–1240, 2000.
[6] D. Rybach, "Investigation on search methods for speech recognition using weighted finite-state transducers," Ph.D. dissertation, RWTH Aachen Univ., 2014.
[7] M. Mohri, F. Pereira, and M. Riley, "Speech recognition with weighted finite-state transdusers," in *Springer Handbook on Speech Processing and Speech Communication*. Springer Berlin Heidelberg, 2008, pp. 559–584.
[8] D. Povey, M. Hannemann, G. Boulianne, L. Burget, A. Ghoshal, M. Janda, M. Karafiat, S. Kombrink, P. Motlicek, Y. Qian, K. Riedhammer, K. Vesely, and N. T. Vu, "Generating exact lattices in the WFST framework," in *Proc. ICASSP*, 2012, pp. 4213–4216.
[9] T. Hori, C. Hori, Y. Minami, and A. Nakamura, "Efficient WFST-based one-pass decoding with on-the-fly hypothesis rescoring in extremely large vocabulary continuous speech recognition," *IEEE Transactions on Audio, Speech and Language Processing*, vol. 15, no. 4, pp. 1352–1365, 2007.
[10] T. Hori and A. Nakamura, "Speech recognition algorithms using weighted finite-state transducers," in *Synthesis Lectures on Speech and Audio Processing*, 2013, pp. 1–164.
[11] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen, "Accurate and compact large vocabulary speech recognition on mobile deivces," in *Proc. Interspeech*, Lyon, France, 2013, pp. 662–665.
[12] I. McGraw, R. Prabhavalkar, R. Alvarez, M. G. Arenas, K. Rao, D. Rybach, O. Alsharif, H. Sak, A. Gruenstein, F. Beaufays, and C. Parada, "Personalized speech recognition on mobile devices," in *Proc. ICASSP*, 2016, pp. 5955–5959.
[13] W. Hu, M. Cai, K. Chen, H. Ding, L. Sun, S. Liang, X. Mo, and Q. Huo, "Sequence discriminative training for offline handwriting recognition by an interpolated CTC and lattice-free MMI objective function," in *Proc. ICDAR*, 2017.
[14] Q. Liu, L.-J. Wang, and Q. Huo, "A study on effects of implicit and explicit language model information for DBLSTM-CTC based handwriting recognition," in *Proc. ICDAR*, 2015, pp. 461–465.
[15] K. Chen, Z.-J. Yan, and Q. Huo, "A context-sensitive-chunk BPTT approach to training deep LSTM/BLSTM recurrent neural networks for offline handwriting recognition," in *Proc. ICDAR*, 2015, pp. 411–415.
[16] M. Cai, W. Hu, K. Chen, L. Sun, S. Liang, X. Mo, and Q. Huo, "An open vocabulary OCR system with hybrid word-subword language models," in *Proc. ICDAR*, 2017.
[17] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," in *Proc. ICML*, 2006, pp. 369–376.
[18] J. Odell, "The use of context in large vocabulary speech recognition," Ph.D. dissertation, University of Cambridge, 1995.
[19] G. Saon, D. Povey, and G. Zweig, "Anatomy of an extremely fast LVCSR decoder," in *Proc. Interspeech*, Lisbon, Portugal, 2005, pp. 549–552.
[20] S. J. Young, N. H. Russell, and J. H. S. Thornton, Eds., *Token passing: a simple conceptual model for connected speech recognition systems*. Cambridge, UK: Cambridge University Engineering Department, 1989.
[21] H. J. G. A. Dolfing and I. L. Hetherington, "Incremental language models for speech recognition using finite-state transducers," in *Proc. ASRU*, 2001, pp. 194–197.
[22] U.-V. Marti and H. Bunke, "The IAM-database: an English sentence database for offline handwriting recognition," *International Journal on Document Analysis and Recognition*, vol. 5, pp. 39–46, 2002.
[23] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The Kaldi speech recognition toolkit," in *Proc. ASRU*, 2011.