Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm

Harald Scheidl
Student of Computer Science
Vienna University of Technology
e0725084@student.tuwien.ac.at

ABSTRACT

Recurrent Neural Networks (RNNs) are used for sequence recognition tasks such as handwritten text recognition or speech recognition. The output of such a RNN is a matrix containing label probabilities for each time-step. To decode these per-frame predictions into the final text multiple connectionist temporal classification decoding algorithms exist. If the text should be constrained to a sequence of dictionary words, the token passing algorithm can be used. However, token passing has a high running time which depends quadratically on the dictionary size and is not able to decode arbitrary character strings such as punctuation marks and numbers. This paper proposes word beam search decoding, which is able to tackle these problems. It constrains the words contained in its beams with help of a prefix tree. The algorithm can integrate a word-level Language Model (LM). It has four LM scoring-modes with different running times regarding the dictionary size. The proposed algorithm outperforms best path decoding, beam search decoding and token passing on a historical handwritten text recognition dataset.

KEYWORDS

connectionist temporal classification, decoding, language model,
recurrent neural network, speech recognition, handwritten text recognition.

1 Introduction

Sequence recognition is the task of transcribing sequences of data with sequences of labels [4]. Graves et al. [5] introduce the Connectionist Temporal Classification (CTC) operation which enables neural network training from pairs of data and target labelings (text). The neural network is trained to output the labelings in a specific coding scheme. Decoding algorithms are used to get the final labeling. Hwang and Sung [8] present a beam search decoding algorithm which can be extended with a character-level Language Model (LM). Graves et al. [7] introduce the token passing algorithm, which constraints its output to a sequence of dictionary words and uses a word-level LM. The motivation to propose the *word beam search decoding algorithm* is twofold:

- Beam search decoding works on character-level and does not constrain its beams to dictionary words.
- The running time of token passing depends quadratically on the dictionary size, which is not feasible for large dictionaries as is shown in Section 7 [7]. Further, the algorithm does not handle non-word characters. This makes it impossible to e.g. recognize arbitrary large numbers, as it is not practicable to put them all into a dictionary.

Word beam search decoding uses a dictionary to constrain the words and uses a word-level bigram LM. It has four different LM scoring-modes with $\mathcal{O}(1)$, $\mathcal{O}(log(W))$, $\mathcal{O}(W)$ and $\mathcal{O}(W \cdot$

$log(W)$) running time with respect to the dictionary size W. Further, it has a free mode in which arbitrary non-words such as numbers and punctuation are recognized. The proposed algorithm is able to outperform token passing and beam search decoding on the Bentham dataset. Further, the running time outperforms token passing.

The rest of the paper is organized as follows: first, a brief introduction to sequence recognition and the CTC operations (loss and decoding) is given. Then, LMs and prefix trees are discussed. Afterwards, the CTC decoding algorithms are presented with a strong focus on the proposed algorithm. Evaluation is done using a historical Handwritten Text Recognition (HTR) dataset. Finally, the conclusion summarizes the paper.

2 Sequence Recognition

Sequential data occurs in the form of handwritten text, speech and gestures among others [5]. Hidden Markov Models (HMMs) show good performance on tasks such as HTR [7]. However, Recurrent Neural Networks (RNNs) are also capable of modeling sequences and have advantages over HMMs from a theoretical and from a practical point of view [7]. When using RNNs without the CTC loss function, there are two disadvantages: the first one is that the training data must be annotated for each time-step and the second one is that the per-frame predictions must be postprocessed to yield the final labeling, i.e. the text in the case of HTR. When using the CTC loss function, dataset creation does not require specifying the exact position of the characters in the input as shown in Figure 1. The annotation "The evidence" is assigned to the image as a whole. While training, the neural network figures out on its own where the text lies in the image, the same applies for the speech signal.

Figure 1: Two common sequence recognition problems. Top: part of a speech signal and its annotation. Bottom: an image of handwritten text and its annotation [10].

3 CTC Coding Scheme

To understand the loss and decoding algorithms, it is essential to understand the coding scheme used by the CTC operation. A RNN outputs a sequence of length T with $C+1$ character probabilities per sequence element, where C denotes the number of characters [7]. An additional character is added to the RNN output which is called *blank* and is denoted by "-" in this paper. For each sequence element the character probabilities sum to 1 [7]. Picking one character for each time-step from the RNN output and concatenating these characters form a path π [7]. The probability of a path is defined as the product of all character probabilities on this path. A single character from a labeling is encoded by one or multiply adjacent occurrences of this character on the path, possibly followed by a sequence of blank labels [7]. A way to model this encoding is by using a Finite State Machine (FSM) [8]. The FSM is created as follows: the labeling is extended by adding a blank label in between all characters and at the beginning and at the end of the labeling. For each character (and blank) in the labeling, a state is inserted into the FSM. A self-loop is added to each state to allow repeated labels on a path. For consecutive characters, a direct transition and a transition through a blank is added. The only exception is the case of repeated characters (like in pi*zz*a), then only a transition through the blank is allowed to avoid ambiguousness. Figure 2 shows a FSM which produces valid paths for the labeling "ab": this is achieved by proceeding from a start state through arbitrary transitions and states to the final state and outputting the label of a state each time a state is reached. A valid path for "ab" is " - a a - - b - -", another one is "a b".

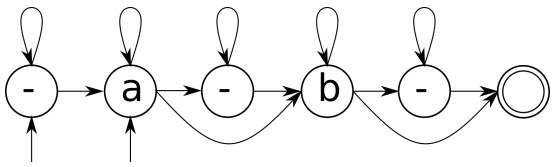

Figure 2: FSM which produces valid paths (encodings) for the labeling "ab". The two left-most states are the initial states while the right-most state is the final state.

To decode a path into a labeling, the encoding operation implemented by the FSM has to be inverted. This is done by the collapsing function B [8]. It is applied to a path π and yields a labeling l by first removing repeated labels and then removing blanks on the path. To give an example, the path "a - b b - -" is collapsed to B("a - b b - -")="ab". The loss is calculated by taking all paths yielding the target

labeling (i.e. all paths π for which $B(\pi) = l$ holds) and summing over their probabilities. This enables training the RNN without knowing the character-positions of the target labeling in the input.

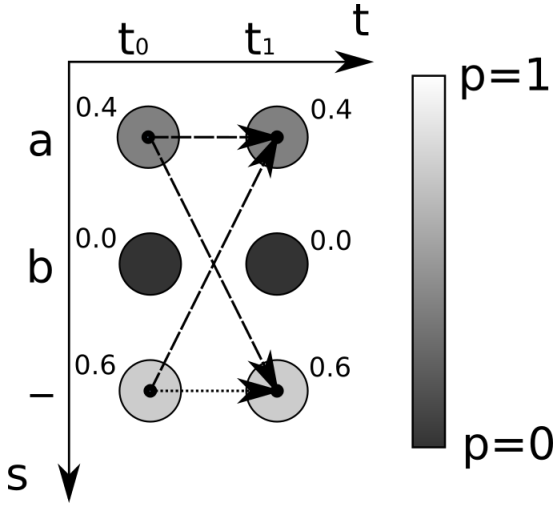

Figure 3: An example of a RNN output. The sequence has a length of 2 with time-steps t_0 and t_1. For each time-step a probability distribution over the possible characters ("a" and "b" and the blank character "-") is outputted by the RNN. Lines indicate four different paths through the RNN output: the dotted line indicates the best path yielding the labeling "" while the dashed lines indicate paths yielding the labeling "a".

After the RNN is trained by the CTC loss, new samples are presented to the neural network to recognize the handwritten or spoken text. A first approximation of decoding the RNN output is to take the most probable label per time-step forming the so called best path and then apply the collapsing function B to this path [7]. This approximation algorithm is called best path decoding [7]. However, there are situations for which this yields the wrong result as illustrated in Figure 3. The given RNN output has a length of 2 and has 2 possible characters "a" and "b" and further the blank "-". Taking the most probable characters yields the best path "- -" and therefore the empty labeling B("- -")="" with probability $0.6 \cdot 0.6 = 0.36$. However, the correct answer is "a", this can be seen by summing up the probabilities of all paths yielding this labeling: "a -", "- a" and "a a" with probability $2 \cdot 0.6 \cdot 0.4 + 0.4 \cdot 0.4 = 0.64$.

4 Language Model

A LM is able to predict upcoming words given previous words and it is also able to assign probabilities to given sequences of words [9]. For example, a well trained LM should assign a higher probability to the sentence "There are ..." than to "Their are ...". A LM can be queried to give the probability $P(w|h)$ that a word sequence (history) h is followed by the word w. Such a model is trained from a text by counting how often w follows h. The probability of a sequence is then $P(h) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdot ... \cdot P(w_n|w_1, w_2, ..., w_{n-1})$ [9].

It is not feasible to learn all possible word sequences, therefore an approximation called N-gram is used [9]. Instead of using the complete history, only a few words from the past are used to predict the next word. N-grams with N=2 are called bigrams. Bigrams only take the last word into account, that means they approximate $P(w_n|h)$ by $P(w_n|w_{n-1})$. The probability of a sequence is then given by $P(h) = P(w_1) \cdot \prod_{n=2}^{|h|} P(w_n|w_{n-1})$. Another special case is the unigram LM, which does not consider the history at all but only the relative frequency of a word in the training text, i.e. $P(h) = \prod_{n=1}^{|h|} P(w_n)$.

The N-gram distributions are learned from a training text. For the unigram distribution, the number of occurrences of a word is counted and normalized by the total number of words in the text. The bigram distribution is calculated by counting how often a word w_1 is followed by a word w_2 and normalizing by the total number of words following w_1. If a word is contained in the test text but not in the training text, it is called Out-Of-Vocabulary (OOV) word. In this case a zero probability is assigned to the sequence, even if only one OOV word occurs. To overcome this problem *smoothing* can be applied to the N-gram distribution, more details are available in Jurafsky [9].

5 Prefix Tree

A prefix tree or trie (from retrieval) is a basic tool in the domain of string processing [3]. Figure 4 shows a prefix tree containing 5 words. It is a tree datastructure and therefore consists of edges and nodes. Each edge is labeled by a character and points to the next node. A node has a word-flag which indicates if this node represents a word. A word is encoded by a path starting from the root node and following edges labeled with the corresponding characters of the word. Querying characters which follow a given prefix is easy: the node corresponding to the prefix is identified and the edge labels determine the characters which can follow the prefix. It is also possible to identify all words which contain a given prefix: starting from the corresponding node of the pre-

fix, all descendant nodes are collected which have the word-flag set. As an example, the characters and words following the prefix "th" in Figure 4 are determined. The node is found by starting at the root node and following the edges "t" and "h". Possible following characters are the outgoing edges of this node, in this example "i" and "a". Words starting with the given prefix are "this" and "that".

Aoe et al. [1] show an efficient implementation of this data structure. Finding the node for a prefix with length L needs $\mathcal{O}(L)$ time in their implementation. The time to find all words containing a given prefix depends on the number of nodes of the tree. An upper bound for the number of nodes is the number of words W times the maximum number of characters M of a word, therefore the time to find all words is $\mathcal{O}(W \cdot M)$.

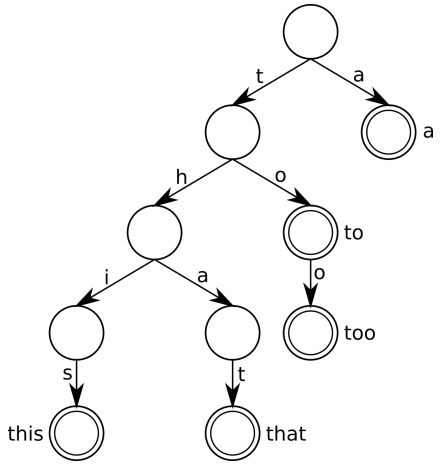

Figure 4: Prefix tree containing the words "a", "to", "too", "this" and "that". Double circles indicate that the word-flag is set.

6 CTC Decoding Algorithms

This section first briefly introduces token passing and beam search decoding. Afterwards, the proposed word beam search algorithm is described in detail.

6.1 Token Passing

This algorithm is proposed by Graves et al. [7], however the following discussion is based on another publication from Graves [4]. A dictionary containing words is given and the output of the algorithm is constrained to a sequence of these words. For each word a word-model is created, which essentially is a state machine connecting consecutive characters with regard to the CTC coding scheme: a character of a word is represented by a path on which the character occurs one or multiply times and is optionally followed by one or multiply blanks (see CTC coding scheme). A word sequence is modeled by putting multiple word-models in parallel, connecting all end-states with all begin-states. The information flow is implemented by tokens, which are passed from state to state. Each token holds the score and the history of already visited words. Figure 5 shows three word-models which are put in parallel. This algorithm limits its output to a sequence of dictionary words and does not allow arbitrary character strings which are needed for numbers or punctuation marks. The time-complexity of this algorithm is $\mathcal{O}(T \cdot W^2)$, where T denotes the sequence length and W the dictionary size [4].

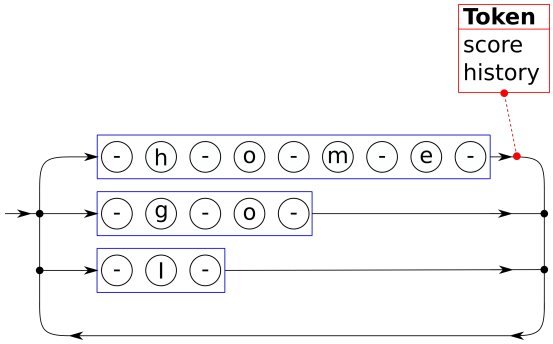

Figure 5: Three word models (blue) are put in parallel. Information flow is implemented by tokens (red) which are passed through the states and between the words.

6.2 Beam Search Decoding

The basic beam search decoding algorithm is described in the paper of Hwang and Sung [8]. An illustration is shown is Figure 6. The RNN output is a matrix of size $T \times (C + 1)$ (number of time-steps times number of characters including the blank), each entry at position (t, c) represents the probability of seeing character c at time-step t. This matrix is fed into the algorithm. At each time-step, each beam-labeling is extended by all possible characters. Additionally, the original beam is also copied to the next time-step. This forms a tree as shown in Figure 6. To avoid the exponential growths due to the branching of the beams, only the best beams are kept at each time-step. The Beam Width BW governs the number of beams to keep. If two beam-labelings at a given time-step are equal, they are merged by first summing up the probabilities and then deleting one of them. When extending a beam, a character-level LM can be used to assign a probability to the beam-labeling. It is also possible to break the beam-labeling into words and assign zero probability if unknown words are contained [6].

The time-complexity is not specified in the paper from Hwang and Sung [8], however it can be derived from the pseudo-code of Algorithm 1 when using all possible characters for the beam-extension. At each time-step, the beams are sorted according to their score. In the previous time-step, each of the BW beams is extended by C characters, therefore $BW \cdot C$ beams have to be sorted which accounts for $\mathcal{O}(BW \cdot C \cdot log(BW \cdot C))$. As this sorting happens for each of the T time-steps, the overall time-complexity is $\mathcal{O}(T \cdot BW \cdot C \cdot log(BW \cdot C))$. The two inner loops are ignored because they only account for $\mathcal{O}(BW \cdot C)$.

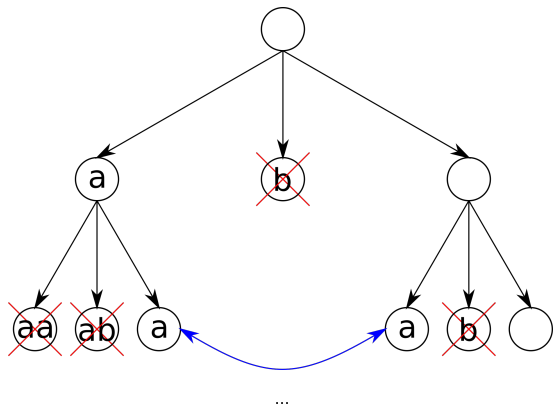

Figure 6: Iteratively extending beam-labelings (from top to bottom) forms this tree. Only the two best-scoring beams are kept per time-step (i.e. $BW = 2$), all others are removed (red). Equal labelings get merged (blue).

6.3 Word Beam Search

Word beam search decoding is a modification to beam search decoding and has the following properties:

- Words are constrained to dictionary words.
- Arbitrary many non-word characters are allowed between words.
- A word-level bigram LM can optionally be integrated.
- Better running time than token passing (regarding time-complexity and the real running time on a computer).

To constrain words to dictionary words and also allow arbitrary many non-word characters between words, each beam is in one of two states. If a beam gets extended by a word-character (typically "a", "b", ...), then the beam is in the word-state, otherwise it is in the non-word-state. Figure 7 shows the two beam-states and the state transitions. A beam is extended by a set of characters which depends on the beam-state. If the beam is in the non-word-state, the beam-labeling can be extended by all possible non-word-characters (typically " ", ",", ...). Further, it can be extended by each character which occurs as the first character of a word. These characters are retrieved from the prefix tree as the labels of the edges leaving the root node. If the beam is in the word-state, the prefix tree is queried to give the list of possible next characters. Figure 8 shows an example for a beam in the word-state. The last word-characters form the prefix "th", the corresponding node in the prefix tree is found by following the edges "t" and "h". The outgoing edges of this node determine the next characters, which are "i" and "a" in this example. Further, if the current prefix represents a complete word (e.g. "to"), then the next characters also include all non-word-characters.

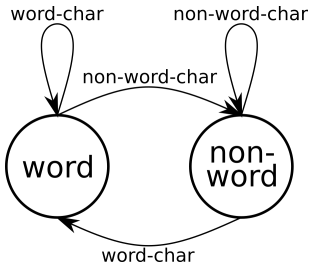

Figure 7: A beam can be in one of two states.

Optionally, a word-level LM can be integrated. A bigram LM is assumed in the following text. The more words a beam contains, the more often it gets scored by the LM. To avoid penalizing beams with many words more than beam with few words, the LM score gets normalized. Four possible LM scoring-modes exist, names are assigned to the modes which are used throughout the paper:

- Words: no LM scoring is used.
- N-grams: each time a beam makes a transition from the word-state to the non-word-state, the beam-labeling gets scored by the LM. The bigram LM scores seeing the second-but-last word and the last word next to each other.
- N-grams + Forecast: each time a beam is extended by a word-character, all possible next words are queried from the prefix tree. Figure 8 shows an example for the prefix "th" which can be extended to the words "this" and "that". The bigram LM scores seeing the last word in the beam-labeling and each possible next word next to each other. All scores are summed up to score the beam-

labeling. This scoring scheme can be regarded as a LM forecast.

- N-grams + Forecast + Sample: in the worst case, all words of the dictionary have to be taken into account for the forecast. All possible next words are randomly sampled, yielding a subset (with limited size) of these words. The LM only scores the words from this subset. Finally, the sum of the scores is corrected to account for the sampling process.

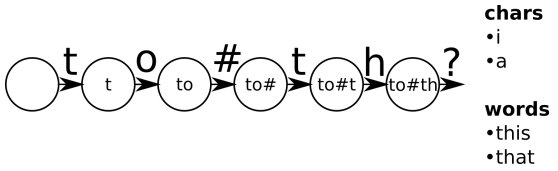

Figure 8: A beam currently in the word-state. The "#" character represents a non-word character. The current prefix "th" can be extended by "i' and "a" and can be extended to form the words "this" and "that". The prefix tree from Figure 4 is assumed in this example.

Algorithm 1 shows the pseudo-code for word beam search decoding. The set B holds the beams of the current time-step, and P holds the probabilities for the beams. P_b is the probability that the paths of a beam end with a blank label, P_{nb} that they end with a non-blank label, and P_{txt} is the probability assigned by the LM. P_{tot} is the abbreviation for $P_b + P_{nb}$. The algorithm iterates from $t = 1$ through $t = T$. The best beams are obtained by sorting them with regard to $P_{tot} \cdot P_{txt}$ and only keep the BW best ones. For each of the kept beams, the probability of seeing this beam-labeling at the current time-step is calculated. Separately book-keeping for paths ending with a blank and paths ending with non-blank accounts for the CTC coding scheme. Each beam gets extended by a set of possible next characters, depending on the beam-state. When extending a beam, the LM calculates a score P_{txt} depending on the scoring-mode. The normalization of the LM score is achieved by taking P_{txt} to the power of $1/numWords(b)$, where $numWords(b)$ is the number of words contained in the beam b. After the algorithm finished its iteration through time, the beam-labelings get completed if necessary: if a beam-labeling ends with a prefix not representing a complete word, the prefix tree is queried to give the list of possible words which contain the prefix. Two different ways to implement the completion exist: either the beam-labeling is extended by the most likely word (according to unigram and/or bigram LM), or the beam-labeling is only completed if the list of possible words contains exactly one entry.

Algorithm 1: Word Beam Search

Data: Character distribution matrix mat, BW and LM
Result: most probable labeling

$B = \{\varnothing\}$;
$P_b(\varnothing, 0) = 1$;
for $t = 1...T$ **do**
  $bestBeams = bestBeams(B, BW)$;
  $B = \{\}$;
  for $b \in bestBeams$ **do**
    if b != $\varnothing$ **then**
      $P_{nb}(b,t)$ += $P_{nb}(b,t-1) \cdot mat(b(t),t)$;
    end
    $P_b(b,t)$ += $P_{tot}(b,t-1) \cdot mat(blank,t)$;
    $B = B \cup b$;
    $nextChars = nextChars(b)$;
    for $c \in nextChars$ **do**
      $b' = b + c$;
      $P_{txt}(b') = applyLM(b,c)$;
      if $b(t) == c$ **then**
        $P_{nb}(b',t)$ += $mat(c,t) \cdot P_b(b,t-1)$;
      else
        $P_{nb}(b',t)$ += $mat(k,t) \cdot P_{tot}(b,t-1)$;
      end
      $B = B \cup b'$;
    end
  end
end
$B = completeBeams(B)$;
return $bestBeams(B, 1)$;

The time-complexity depends on the scoring-mode used. If no LM is used, the only difference to vanilla beam search decoding is to query the next possible characters. This takes $\mathcal{O}(M \cdot C)$ time, where M is the maximum length of a word and C is the number of characters. The overall running time therefore is $\mathcal{O}(T \cdot BW \cdot C \cdot (log(BW \cdot C) + M))$. If a LM is used, then a lookup in a unigram and/or bigram table is performed when extending a beam. Searching such a table takes $\mathcal{O}(log(W))$ and is done at most C times for each beam. This sums to the overall running time of $\mathcal{O}(T \cdot BW \cdot C \cdot (log(BW \cdot C) + M + log(W)))$. In the case of LM forecasting, the next words have to be searched which takes $\mathcal{O}(M \cdot W)$ and is done at most C times for each beam. The LM is queried S times, where S is the size of the word sample. If no sampling is used, then $S = W$. The running time sums to $\mathcal{O}(T \cdot BW \cdot C \cdot (log(BW \cdot C) + S \cdot log(W) + W \cdot M))$.

7 Results

Evaluation is done using the test-set of the Bentham HTR dataset, a sample is shown in Figure 9. It contains 93 different characters. Character Error Rate (CER) and Word Error Rate (WER) are used as error measures. CER is the Levenshtein distance between ground truth text and recognized text, divided by the length of the ground truth text [2]. WER goes accordingly, but on word-level. The neural network is implemented using the TensorFlow framework and trained with the CTC loss function. The output sequence of the neural network has a length of 100 time-steps. There are 93 characters in the Bentham dataset, therefore the output matrix has a size of $T \times (C + 1) = 100 \times 94$. Beam search decoding, token passing and word beam search decoding are implemented as custom TensorFlow operations in C++ with the beam width set to 10. The LM uses add-k smoothing with a smoothing value of 0.01. The prefix tree is implemented using sorted lists for the outgoing edges to keep memory usage low. Searching an outgoing edge therefore takes $\mathcal{O}(log(C))$ using binary search.

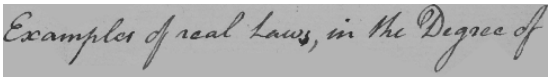

Figure 9: A sample from the Bentham HTR dataset [10].

Four experiments are conducted, the first one analyzes the influence of the LM scoring-modes, the second and third one analyze the influence of the training text of the LM and the last one compares best path decoding, beam search decoding and token passing to the proposed algorithm.

Table 1 compares the scoring-modes. The text contained in the test-set of the Bentham dataset is used to train the LM. Training the LM with the text that has to be recognized is of course a simplification, however, the objective of this experiment is to evaluate the proposed algorithm and not the performance of the LM. As can be seen, the *Words* and *N-grams* modes are the fastest ones with $40ms$ and $50ms$ per sample. The *N-Grams* mode shows the best WER, while the *N-grams + Forecast + Sample* mode shows the best CER. However, forecasting the unigram and bigram distributions at each time-step makes the algorithm around 10 times slower, even when sampling is enabled. This suggests that looking up the next word in the prefix tree is the bottleneck.

Tables 2 and 3 show the influence of different training texts for the LM, the former one for the *Words* mode, the latter one for the *N-grams* mode. The LM is trained with the text of the training-set, the concatenation of training-set and validation-set and the concatenation of training-set, validation-set and a list of 466000 English words [1]. The *Words* mode yields better results than the *N-grams* mode. The reason is that the training text is too short to learn enough word-pairs and their probability. Smoothing the N-gram distributions, as applied in the form of add-k smoothing, decreases this problem. However, only restricting the words to dictionary words still yields better results. Regarding the training texts, the results get better the longer the texts are.

Mode	CER/WER [%]	Time [ms]
W	4.65/7.98	40
N	4.51/7.30	50
N+F	4.55/7.42	544
N+F+S	4.50/7.46	449

Table 1: Comparison of the four LM scoring-modes: Words (W), N-grams (N), N-grams + Forecast (N+F), N-grams + Forecast + Sample (N+F+S).

Text	CER/WER [%]	Time [ms]
T	7.32/17.10	45
T+V	7.01/16.37	45
T+V+L	5.72/13.72	52

Table 2: Comparison between three different texts to train the LM: training text (T), training and validation text (T+V), training and validation text and English word list (T+V+L). Words mode is used.

A comparison between the proposed algorithm and three other algorithms is given in Table 4. As in the first experiment, the text of the test-set is used to train the LM. Best path decoding is an approximation algorithm and can be seen as a baseline result. It only takes $14ms$ per sample, which already includes the evaluation of the TensorFlow computation graph of the neural network. Beam search decoding uses a character-

[1] Taken from https://github.com/dwyl/english-words

Text	CER/WER [%]	Time [ms]
T	7.77/16.83	73
T+V	7.41/15.93	75
T+V+L	6.91/14.19	76

Table 3: Comparison between three different texts to train the LM: training text (T), training and validation text (T+V), training and validation text and English word list (T+V+L). N-grams mode is used.

Algorithm	CER/WER [%]	Time [ms]
Best Path	5.60/16.61	14
Beam	5.35/15.63	106
Token	8.66/9.45	389
Word Beam	4.51/7.30	50

Table 4: Comparison between four different decoding algorithms: best path decoding, beam search decoding, token passing and word beam search decoding. N-grams mode is used.

level LM and achieves slightly better results than best path decoding. Token passing has a 6% lower WER than beam search decoding, however, the CER is worse than the one of best path decoding. The reason is that token passing only uses words from a dictionary and does not allow other character strings such as arbitrary numbers. Word beam search decoding is used in *N-grams* mode and achieves the best results regarding CER and WER. This shows that the proposed algorithm is able to outperform the other algorithms if a well-trained LM is available for the dataset.

8 Conclusion

A decoding algorithm for CTC trained neural networks was proposed which works on word-level, restricts words to dictionary words, allows arbitrary character strings between words and can optionally integrate a word-level LM. It comes with four different scoring-modes which govern the effect of the optional LM and can also be used to govern the running time of algorithm. The restriction to dictionary words is achieved with a prefix tree which is used to retrieve characters given a word-prefix. The algorithm was evaluates on the Bentham HTR dataset. Experiments have shown that the algorithm is able to outperform best batch decoding, beam search decoding and token passing when a well-trained LM is available. If no LM is available, the *Words* mode can be used which only constrains the words contained in the beam-labelings.

References

[1] J.I. Aoe, K. Morimoto, and T. Sato. An efficient implementation of trie structures. *Software: Practice and Experience*, 22(9):695–721, 1992.

[2] T. Bluche. *Deep Neural Networks for Large Vocabulary Handwritten Text Recognition*. PhD thesis, Université Paris Sud-Paris XI, 2015.

[3] P. Brass. *Advanced data structures*. Cambridge University Press Cambridge, 2008.

[4] A. Graves. *Supervised sequence labelling with recurrent neural networks*. Springer, 2012.

[5] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist Temporal Classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 369–376. ACM, 2006.

[6] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1764–1772, 2014.

[7] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, 2009.

[8] K. Hwang and W. Sung. Character-level incremental speech recognition with recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5335–5339. IEEE, 2016.

[9] D. Jurafsky and J. Martin. *Speech and Language Processing*. Pearson London, 2014.

[10] J. Sánchez, V. Romero, A. Toselli, and E. Vidal. ICFHR2014 competition on handwritten text recognition on transcriptorium datasets. In *14th International Conference on Frontiers in Handwriting Recognition*, pages 785–790. IEEE, 2014.