Word Beam Search: A CTC Decoding Algorithm

Improve text recognition results: avoid spelling mistakes, allow arbitrary numbers and punctuation marks and make use of a word-level language model.

Let’s suppose we have a neural network (NN) for text recognition (OCR or HTR) which is trained with the connectionist temporal classification (CTC) loss function. We feed an image containing text into the NN and get the digitized text out of it. The NN simply outputs the characters it sees in the image. This purely optical model might introduce errors because of similar looking characters like “a” and “o”.

Problems like the one shown in Fig. 1 will look familiar to you if you have already worked on text-recognition: the result is of course wrong, however, when we look closely at the handwritten text, we can imagine why the NN confuses “a” with “oi” in the word “random”.

![A random number: 1234.

Fig. 1: Why does the NN make such dumb mistakes?

Can we improve our result?

So, how to tackle such situations? There are different decoding algorithms available, some also include a language model (LM). Let’s look what the algorithms output for our example:

- **Best path decoding**: “A roindan nunibr: 1234.”. This is the result shown in Fig. 1. Best path decoding only uses the output of the NN (i.e. no language model) and computes an approximation by
taking the most likely character at each position.

**Beam search:** “A roindan numbr: 1234.” It also only uses the NN output, but it uses more information from it and therefore produces a more accurate result.

**Beam search with character-LM:** “A randan number: 1234.” It additionally scores character-sequences (e.g. “an” is more probable than “oi”) which further improves the result.

**Token passing:** “A random number”. This algorithm uses a dictionary and word-LM. It searches for the most probable sequence of dictionary words in the NN output. But it can’t handle arbitrary character sequences (numbers, punctuation marks) like “: 1234.”

None of the algorithms gets it right. But we observe nice properties of beam search and token passing:

Beam search allows **arbitrary character strings**, which is needed to decode **numbers and punctuation** marks.

Token passing limits its output to **dictionary words**, which **avoids spelling mistakes**. Of course, the dictionary must contain all words which have to be recognized.

A combination of both properties would be nice: when we see a word, we only allow words from a dictionary, but in any other case, we allow arbitrary character strings. Let’s look at an algorithm which implements our desired behavior: word beam search.

**Proposed algorithm: Word Beam Search**

We use the vanilla beam search algorithm as a starting point. This algorithm iterates through the NN output and creates text candidates (called beams) which are scored. Fig. 2 shows an illustration of the evolution of beams: we start with the empty beam, then add all possible characters (we only have “a” and “b” in this example) to it in the first iteration and only keep the best scoring ones. The beam width controls the number of surviving beams. This is repeated until the complete NN output is processed.
Let’s look at how we have to adapt vanilla beam search to get the desired behavior. We want the algorithm to behave differently when it recognizes a word and when it recognizes a number or punctuation marks. Therefore we add a state variable to each beam (state diagram see Fig. 3). A beam is either in word-state or in non-word-state. If the beam-text currently is “Hel”, then we are in word-state and only allow adding word-characters until we have a complete word like “Hell” or “Hello”. A beam currently in non-word-state could look like this: “She is 3”.

Going from word-state to non-word-state is allowed when a word is completed: then we can add a non-word-character to it, like if we have “Hello” and add “ “ to it we get “Hello “. The other direction from non-word-state to word-state is always allowed, like in “She is 33 “, where we may add the word-character “y” to get “She is 33 y” (and later, maybe, will get “She is 33 years old.”).
When in word-state, we only allow adding characters which eventually will form words. In each iteration, we at most add one character to each beam, so how do we know which characters will, after adding enough characters, form a word? A prefix tree as shown in Fig. 4 can solve this task: all words from the dictionary are added to the tree.
When adding a word like “too”, we start at the root node, add (if it does not yet exist) an edge labeled with the first character “t” and a node, then add an edge “o” to this new node, and again add an “o”. The final node gets marked to indicate the end of a word (double circles). If we repeat this for the words “a”, “to”, “too”, “this” and “that”, we get the tree shown in Fig. 4.
It is now easy to answer two questions:

Given a prefix, **which characters can be added** to eventually form a word? Simply go to the prefix's node in the prefix tree by following the edges labeled with the prefix's characters, and then collect the labels of the outgoing edges of this node. If we are given the prefix “th”, then the possible next characters are “i” and “a”.

Given a prefix, **which words can be created** out of it? From the prefix's node, collect all descendant nodes which are marked as a word. If we are given the prefix “th”, then the possible words are “this” and “that”.

With this information, we can constrain the characters added to beams (which are in word-state): we only add characters which eventually will form a word. See Fig. 5 for an illustration.
We only keep the best-scoring beams per iteration. The score depends on the NN output. But we can also incorporate a word-LM with different scoring modes (the provided abbreviations for these modes are used in the results section):

- **Don't use the LM at all (W)**, only constrain the words by a dictionary. This is what we have discussed so far.

- **Score by LM each time a word is fully recognized (N)**. Suppose we have a beam “My nam”. As soon as an “e” is added and therefore the last word “name” is finished, the LM scores seeing “My” and “name” next to each other.

- **Look into the future (N+F)**: with the help of the prefix tree, we know which words might occur in later iterations. A beam “I g” might be extended to “I go”, “I get”, … So instead of waiting until a word is finished to apply the LM-score to a beam, we can also do it in each iteration by summing over the LM-scores of all possible words, this would be “I” and “go”, “I” and “get”, … in the mentioned example. A small performance improvement can be achieved by only taking a random subset (N+F+S) of the words.

We now have an algorithm which is able to recognize the correct text “A random number: 1234.” shown in Fig. 1. The words are constrained by a dictionary, but all the other characters are just added in the way the NN sees them.

**How well does Word Beam Search perform?**
It is nice that the algorithm is able to correctly recognize the sample from Fig. 1. But we are more interested in how well the algorithm performs on a complete dataset. We will use the Benthem HTR dataset. It contains handwritten text from the time around the year 1800, a sample is shown in Fig. 6.

![Sample from Benthem dataset.](image)

We feed the same NN output into all decoding algorithms. This way, a fair comparison is possible. We use two different training texts for the LM: 1.) the text from the test-set which is an ideal training text as it contains all words to be recognized and 2.) a rudimentary training text created from the text of the training-set concatenated with a word list containing 370000 words. The error measures are character error rate (CER) and word error rate (WER). CER is the edit distance between ground truth text and recognized text, normalized by the ground truth length. WER is defined the same way, but on word-level.

Results are shown in Table 1. Word beam search outperforms the other algorithms on this dataset. If we have a good training text for the LM, it makes sense to use it to score the beams (modes N, N+F or N+F+S). Otherwise, it is best to just constrain the words and don’t use the LM to score the beams (mode W).

If we don’t know which words we have to recognize, or if we don’t have to recognize words at all, other algorithms will perform better.
<table>
<thead>
<tr>
<th></th>
<th>Perfect LM</th>
<th>Rudimentary LM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Path</strong></td>
<td>5.60 / 17.06</td>
<td>5.60 / 17.06</td>
</tr>
<tr>
<td><strong>Token Passing</strong></td>
<td>8.16 / 9.24</td>
<td>(not feasible)</td>
</tr>
<tr>
<td><strong>VBS</strong></td>
<td>5.35 / 16.02</td>
<td>5.55 / 16.39</td>
</tr>
<tr>
<td><strong>WBS W</strong></td>
<td>4.22 / 7.90</td>
<td>5.47 / 14.09</td>
</tr>
<tr>
<td><strong>WBS N</strong></td>
<td>4.07 / 7.08</td>
<td>6.15 / 13.90</td>
</tr>
<tr>
<td><strong>WBS N+F</strong></td>
<td>4.05 / 7.36</td>
<td>6.76 / 18.00</td>
</tr>
<tr>
<td><strong>WBS N+F+S</strong></td>
<td>4.06 / 7.39</td>
<td>6.75 / 18.06</td>
</tr>
</tbody>
</table>

Table 1: Results given as CER [%] / WER [%] for different algorithms (VBS: vanilla beam search, WBS: word beam search, W: no LM used, N: LM used, N+F: LM with forecasting, N+F+S: LM with forecasting and sampling) and training texts for the LM (perfect and rudimentary LM). A small CER/WER means that we have a well performing algorithm.

**Implementation**

Code can be found on GitHub. A Python, C++ and TensorFlow implementation is provided.

https://github.com/githubarald/CTCWordBeamSearch

**References and further reading**

Code and publications:

- Implementation of word beam search
- ICFHR 2018 paper
- Poster
- Thesis: evaluation of word beam search on 5 datasets

Articles on text recognition and CTC:

- Introduction to CTC
- Vanilla beam search
- Implementing a text recognition system