An approach for output decoding of neural networks

Tobias Strauß* Roger Labahn Gundram Leifert Welf Wustlich

December 21, 2012
(date of revision: August 23, 2013)

Abstract

We review two decoding methods for recurrent neural networks – based on the well known weighted Levenshtein distance and the connectionist temporal classification (CTC) – and we introduce a new method – the dynamic Levenstein distance (DynWL). We compare these three methods analytically in terms of time complexity and error performance. Although the approaches are different, there are deep connections between these ways of decoding. Finally, we test on the Arabic and French ICDAR data sets. Our experiments show that CTC yields the smallest error rates. Nevertheless, there are scenarios where DynWL is a good choice between performance and time complexity.

1 Output decoding via dictionaries

In this section, we describe how to choose a lexicon item from a given lexicon $\mathcal{D}$ using the output matrix of the RNN. We look at four different methods which differ in running time and performance: Levenshtein distance, Hamming, DynWL and CTC.

Let $\mathcal{A}$ be the Alphabet. The RNN $\mathcal{N} : \mathcal{I} \subset \bigcup_{t\in\mathbb{N}} \mathbb{R}^{m\times t} \to \mathcal{C} \subset \bigcup_{t\in\mathbb{N}} \mathbb{R}^{|\mathcal{A}|+1\times t}$ distinguishes between $|\mathcal{A}| + 1$ different classes namely the different letters and one artificial class called: blank, no label, or *NaC*. Further, let $\mathcal{A}' := \mathcal{A} \cup \{NaC\}$ be the set of class labels. A word is a sequence of elements of $\mathcal{A}$. To symbolize a *NaC* within a word, we use $\sqcup$. To keep it simple, we abbreviate the number of columns of $\mathcal{N}(\boldsymbol{x})$ by $T = T(\boldsymbol{x})$.

Decoding problem: We define our test data to be $(\boldsymbol{x}, \boldsymbol{z}) \in S' \subset \mathcal{I} \times \mathcal{D}$ where S' is the set of all words $\boldsymbol{z} \in \mathcal{D}$ and writings $\boldsymbol{x}$ of $\boldsymbol{z}$. The decoding algorithm $d_\mu : \mathcal{C} \to \bigcup_{t\in\mathbb{N}} \mathcal{A}^t$ is an algorithm which chooses the lexicon item from the output matrix $\mathcal{N}(\boldsymbol{x})$ of the RNN. Usually, one defines a function $\mu : \mathcal{C} \to \bigcup_{t\in\mathbb{N}} \mathcal{A}^t \to \mathbb{R}$ whereas $\mu(\mathcal{N}(\boldsymbol{x}), \boldsymbol{s})$ characterizes the confidence that $\mathcal{N}(\boldsymbol{x})$ encodes $\boldsymbol{s} \in \bigcup_{t\in\mathbb{N}} \mathcal{A}^t$. Typically, the decoding function d_μ is defined as

$$d_\mu(\mathcal{N}(\boldsymbol{x})) := \operatorname{argmin}_{\boldsymbol{s}\in\mathcal{D}} \mu(\mathcal{N}(\boldsymbol{x}), \boldsymbol{s}).$$

Thus, d_μ returns the lexicon item with the minimum value $\mu(\mathcal{N}(\boldsymbol{x}), \boldsymbol{s})$ for a network output $\mathcal{N}(\boldsymbol{x})$. To introduce examples functions, we need to provide some terms: The elements of the output matrix $\mathcal{N}(\boldsymbol{x})$ are $(y_c^t)_{t,c}$. If the symbol c describes a class, we usually use c also as subscript symbolizing the related index of class c of a vector or matrix i.e. y_c^t is the output value of class c at position t. The vector $\boldsymbol{y}^t$ denotes the confidences for all labels. A *path* $\boldsymbol{\pi} = (\pi^t)_t \in \bigcup_{t\in\mathbb{N}} \mathcal{A}'^t$ is a sequence of class labels from $\mathcal{A}'$. The activations of the path $(\pi^t)_t$ are denoted by $y^t_{\pi^t}$. The *best path* $\boldsymbol{\beta} = \boldsymbol{\beta}(\mathcal{N}(\boldsymbol{x})) := (\beta^t)_t$ is the path with the highest activations $\max_c y_c^t$ at each position t. To *collapse* a path $(\pi^t)_t$ means to merge consecutive identical π^t and delete the *NaC*s. For the related function, we use the notation of

*Universität Rostock

[Gra12]: Let $\mathcal{F} : \bigcup_{t\in\mathbb{N}} \mathcal{A}'^t \to \bigcup_{t\in\mathbb{N}} \mathcal{A}^t$ define the many to one function which maps a path to a word. For example, $\mathcal{F}(\sqcup a \sqcup ab \sqcup) = \mathcal{F}(\sqcup aa \sqcup aabb \sqcup) = aab$. The symbol $\boldsymbol{s}_{1:u}$ denotes the subword $s^1 s^2 \dots s^u$ of $s^1 s^2 \dots s^U = \boldsymbol{s} \in \mathcal{A}^U$.

In the following, we want to review several functions μ.

1.1 Levenshtein distance

The first function $\mathrm{WL}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z})$ works as follows: We collapse the best path and calculate the Levenshtein distance (also edit distance, see [MRS08]) $\mathrm{WL}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) := ED(\mathcal{F}(\boldsymbol{\beta}), \boldsymbol{z})$ between $\boldsymbol{z}$ and the collapsed best path $\boldsymbol{\beta}$. The Levenshtein distance can be calculated efficiently using dynamic programming. Since it is well known how to compute the Levenshtein distance, we omit a detailed description here. This method does not incorporate all of the information of the output of the neural net. Thus, we would expect d_{WL} to yield more errors than the following methods. Better results may be obtained by adapting the edit costs (weighted Levenshtein distance). The Levenshtein distance is still the most used decoding scheme for the output neural networks.

1.2 Hamming distance

Similarly to the Levenshtein distance, one can also use the Hamming distance. But in this case, we have to consider paths instead of words. Given two vectors (of paths in our case) $\boldsymbol{\pi}, \overline{\boldsymbol{\pi}}$ from $\mathcal{A}'$, the Hamming distance is defined to be the number of distinct positions $\pi^t \neq \overline{\pi}^t$. We define

$$\mathrm{ham}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) := \min_{\boldsymbol{\pi}\in\mathcal{F}^{-1}(\boldsymbol{z})} \mathrm{ham}(\boldsymbol{\pi}, \boldsymbol{\beta}).$$

1.3 Connectionist Temporal Classification

The *Connectionist Temporal Classification (CTC)* was introduced in [GFGS06]. To train neural networks for sequence labeling, one usually needs position depended targets which often are difficult to generate since they are expensive and the creating consumes a lot of time. CTC overcomes this difficulty by an automatic separation of the inputs. All you need is the target string but not the precise position of the letter. Additionally, a great benefit of CTC is that it interprets output activations as probabilities such that we can ask for the probability of any reference given image $\boldsymbol{x}$. To choose the maximum probability lexicon item is not a new approach (it is a standard approach for HMMs, see for example [KLSS02]) but it is not very common for neural networks.

To normalize the output matrix to entries between 0 and 1, every column (which means the output at a certain position) of the neural net is applied to softmax. The normalized activations sum to 1 such that we will interpret them as probabilities $p(c|\boldsymbol{y}^k)$ of c given $\boldsymbol{y}^k$. Let s_c^t be

$$s_c^t := \frac{\exp(\mathcal{N}(\boldsymbol{x})_{t,k})}{\sum_j \exp(\mathcal{N}(\boldsymbol{x})_{t,j})}.$$

We define $\mathrm{CTC}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) := -p(\boldsymbol{z}|\boldsymbol{x})$ to formulate a minimization problem. Assume $p(c_1|\boldsymbol{y}^k)$ and $p(c_2|\boldsymbol{y}^l)$ are independent for any $c_1, c_2 \in \mathcal{A} \cup \{NaC\}$ and $k \neq l$, then

$$p(\boldsymbol{z}|\boldsymbol{x}) := \sum_{\boldsymbol{\pi}\in\mathcal{F}^{-1}(\boldsymbol{z})} p(\boldsymbol{\pi}|\boldsymbol{x})$$

whereas

$$p(\boldsymbol{\pi}|\boldsymbol{x}) := \prod_{k=1}^{T} s_{\pi^k}^k.$$

Fortunately, we do not have to evaluate every path individually. The probability $p(\boldsymbol{z}|\boldsymbol{x})$ can be calculated using dynamic programming. For a detailed description see [GFGS06] or [Gra12].

1.4 Dynamic Weighted Levenshtein distance

In this subsection, we introduce a new function - *the Dynamic Weighted Levenshtein Distance (DynWL)* - which is strongly inspired by the Levenshtein Distance but it turns out to have also connections to the CTC approach. Just like the previous functions, DynWL is also calculated in a dynamic programming way. But we just have to compute half as much values as CTC. This results from the fact that we do not need to extend $\boldsymbol{z}$ by inserting *NaC*s.

We introduce the new term "squash". Squashing a path $\boldsymbol{\pi}$ means almost the same as collapsing $\boldsymbol{\pi}$. The result is a word. The difference between both terms is the treatment of repeated letters: If a *NaC* is between two other repeated labels the path collapses to two letters but squashing the same path yields both one and two letters. The other way around, two repeated labels without a *NaC* inbetween collapse to one letter but it squashes to both one letter and two letters. Formally, we model this by the mapping $\mathcal{G} : \mathcal{A}'^* \to \mathcal{P}(\mathcal{A}^*)$. The mapping $\mathcal{G}$ deletes or substitutes all the *NaC*s by neighboring labels and outputs all possibilities for keeping or deleting repeated labels. With the above example $\mathcal{G}(\sqcup\, a \sqcup\, ab \sqcup) = \{aab, ab\}$ and $\mathcal{G}(\sqcup\, aa \sqcup\, aabb \sqcup) = \{aaaabb, aaabb, aabb, abb, aaaab, aaab, aab, ab\}$.

Therefore, one path can squash to a set of words (containing more than one element). The idea behind this is a more careful treatment with the *NaC*. If the *NaC* does not work as expected - namely as a separator between different characters - we allow several possibilities to be valid.

The costs of any character c depends on the maximum activation $y^t_{\beta_t}$ at position t and the activation y^t_c. I.e. DynWL uses *dynamical*, output dependent costs instead of constant costs as Hamming or Levenshtein. Substituting β^t at position t with label c generates the costs

$$d^t_c := \mathcal{N}(\boldsymbol{x})_{t,\beta^t} - \mathcal{N}(\boldsymbol{x})_{t,c}.$$

We denote the costs $y^t_{\beta_t} - y^t_{\pi^t}$ as distance (to the bestpath) hereafter. The Dynamic Weighted Levenshtein Distance calculates the minimum distance of any path squashing to the reference $\boldsymbol{z}$. Now, let

$$\text{dynwl}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) := \min_{\boldsymbol{\pi} \in \mathcal{G}^{-1}(\boldsymbol{z})} \sum_{t=1}^{|\pi|} d^t_{\beta^t}.$$

An efficient way to calculate this distance is given in section 2.

2 DynWL Algorithm

The most efficient way to calculate $\text{dynwl}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z})$ is a Viterbi-style algorithm. We denote the entries of the related matrix by γ^t_u describing the minimal "distance" of the best path at position t to a path $\boldsymbol{\pi}_{\boldsymbol{z},u}$ where $\boldsymbol{z}_{1:u} \in G(\boldsymbol{\pi}_{\boldsymbol{z},u})$. γ^t_0 specifies the distance that the word does not begin until position t. The initial values are

$$\gamma^0_k = \begin{cases} d^0_{\sqcup} & k = 0 \\ d^0_{z_1} & k = 1 \\ 0 & \text{else} \end{cases}.$$

The recursive formula for γ^t_u is

$$\gamma^t_u := \min\left\{\gamma^{t-1}_u + d^{t-1}_{z_u}\,,\, \gamma^{t-1}_{u-1} + d^{t-1}_{z_u}\,,\, \gamma^{t-1}_u + d^{t-1}_{\sqcup}\right\},$$

with the initial values $\gamma^t_u = \infty$ if $t < u$, and $\gamma^t_0 = \sum_{s\in[t]} d^s_{\sqcup}$ and $\gamma^1_1 = d^1_{\boldsymbol{z}_1}$. The overall costs for $\boldsymbol{z}$ can be read from the last item of the matrix $\text{dynwl}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) = \gamma^T_{|\boldsymbol{z}|}$. Figure 2 illustrates the DynWL algorithm and Algorithm 1 provides the pseudocode for DynWL.

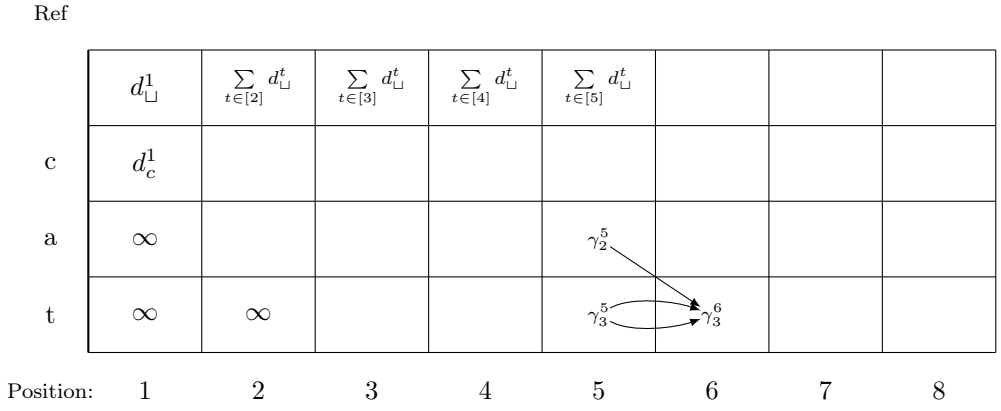

Figure 1: Scheme of the dynamic programming matrix for DynWL. The varable $d^k_\sqcup$ denotes the distance of the *NaC* activation to the maximum and $[n] := \{1, \ldots, n\}$. There are 2 previously calculated γ values which determine a table entry γ^t_u in the center. Either you already read the letter z_u , then you continue with a *NaC* or again with the same letter, or you read the previous letter z_{u-1}. Then you have to continue with the letter z_u.

Algorithm 1 DynWL

Require: $\boldsymbol{z}$, $(d^t_c)_{c,t}$
  for $t := 1 \to T$ **do**
    $\gamma^t_0 \leftarrow \sum_{i=1}^t d^t_\sqcup$
  end for
  $\gamma^1_1 \leftarrow d^1_{z_1}$
  for $u := 2 \to |\overline{\boldsymbol{z}}|$ **do**
    $\gamma^1_u \leftarrow \infty$
  end for
  for $t := 2 \to T$ **do**
    for $u := 2 \to |\overline{\boldsymbol{z}}|$ **do**
      $\gamma^t_u \leftarrow \min\left\{\gamma^{t-1}_u + d^{t-1}_{z_u}\,,\ \gamma^{t-1}_{u-1} + d^{t-1}_{z_u}\,,\ \gamma^{t-1}_u + d^{t-1}_\sqcup\right\}$
    end for
  end for
  return $\gamma^T_{|\overline{\boldsymbol{z}}|}$

3 Comparision

DynWL is an extension to the Hamming and Levenshtein decoding whereas the substitution, deletion and insertion costs depend on the activations.

Similarly to CTC, DynWL returns the dictionary item which is also optimal in some way. It returns the dictionary item $\boldsymbol{z}$ whose optimal path yields the highest softmax probability under all paths squashing to any dictionary item. This is because the distance d^t_c of the activations can be rewritten as the distance of logarithmic softmax values:

$$\begin{aligned}\mathrm{dynwl}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{z}) &= \min_{\boldsymbol{z}=G(\boldsymbol{\pi})} \sum_{t=1}^{T} (\max_{c'} y^t_{c'} - y^t_c) \\ &= \min_{\boldsymbol{z}=G(\boldsymbol{\pi})} \sum_{t=1}^{T} (\max_{c'} \ln(s^t_{c'}) - \ln(s^t_c)) \end{aligned} \tag{1}$$

So if neural net works well and the softmax probabilities approximate the real probability of a character at the current position reliably, DynWL will do better than Hamming and Levenshtein decoding.

	WLD	Hamming	DynWL	CTC
average error	18,32%	17,68%	11,22%	10,77%
minimum error	13,84%	15,73%	7,95%	7,62%
maximum error	35,71%	34,23%	23,92%	22,83%

Table 1: Word error rate on the IFN/ENIT dataset. The minimum error represents the smallest error of the ten randomly initialized networks. maximum error is analogously defined.

DynWL could also be seen as modified CTC but there are two main differences: the NaCs role and the way different paths are combined. CTC interprets the NaC as letter separator (paths are collapsed) whereas DynWL sees this special label also as placeholder for an unsure character (squashed paths). As a result of squashing the paths, several words may yield the same DynWL costs and a special treatment for these cases is required. On the other hand, DynWL also incorporates paths which CTC does not. This could be of advantage if for example two repeated letters are written narrow, there might be not enough space to insert a separation NaC such that CTC will generate more costs and might return the wrong dictionary item.

As another difference, CTC sums all distinct paths collapsing to $\boldsymbol{s}$ to get the overall probability of $\boldsymbol{s}$ in $\mathcal{N}(\boldsymbol{x})$. In contrast, DynWL takes the minimum of both paths, returning only the distance to $\boldsymbol{s}$ of the "nearest" path.

4 Experiments

In this section, we compare the decoding methods on two experiments from ICDAR 2009. We use the same neural network architectures as in [Gra12]. The network consists of MDLSTM Layer and two Feedforward Layer. We omit a detailed description and refer to [Gra12] where the interested reader will find all important information. The program is written in JAVA. Note that we use exactly the same input to all decoding methods.

4.1 Offline Arabic Handwriting Recognition

Data We compare the performance of WL, Hamming, DynWL and CTC decoding on the IFN/ENIT database of handwritten Arabic words (see [PMM$^+$02]). It contains 32492 different imagines of Tunisian places. Unlike the original task, we compare the names of the places and do not incorporate the zip codes.

Setup We divide the data into a training set of size 30,000 and a validation set containing 2492 items. We train 10 randomly initialized neural networks on the training set. The tests are only performed on the validation set. The lexicon was created from all words occurring in the whole data set. It contains 1508 different words.

Evaluation Table 1 shows the average minimum and maximum word error rates of the 10 neural networks on the IFN/ENIT dataset. Levenshtein and Hamming decoding perform worst. CTC performs slightly better than DynWL. Note that the RNNs are trained to minimize the CTC error. This might be a disadvantage for DynWL. However, we did not implement network training version of DynWL. The computation times are almost equal. CTC and Hamming decoding need a little bit more computation time since the dynamic programming table is about 2 times bigger than for DynWL. WL is the fastest algorithm.

	Test 1				Test 2			
	WLD	Hamming	DynWL	CTC	WLD	Hamming	DynWL	CTC
average error	16,49%	16,05%	9,42%	8,66%	19,82%	19,72%	12,52%	10,87%
minimum error	14,98%	14,39%	8,34%	7,66%	18,23%	17,82%	11,31%	9,77%
maximum error	18,67%	17,86%	10,77%	10,00%	22,13%	21,61%	14,17%	12,33%

Table 2: Word error rate for the two French handwriting recognition test. The minimum error represents the smallest error of the ten randomly initialized networks. Maximum error is analogously defined.

4.2 French Handwriting Recognition

Data We compare the performance of WL, Hamming, DynWL and CTC decoding on a subset of the RIMES database of handwritten mail snippets (see [GA09]) published at ICDAR 2009.

Setup Again the neural network architecture for this task was provided by Graves (see [Gra12]). The systems parameter are reported there. We divide the data into a training set of size 44,196 and a validation set containing 7542 items. We train 10 randomly initialized neural networks on the training set. The tests are only performed on the validation set. Similar to the original task, we test with two different lexicons: The first one (Test 1) contains all elements from the validation list (1636 words) and the second one (Test 2) was created from all words occurring in the whole data set (4936 words).

Evaluation Table 2 shows the average minimum and maximum word error rates of the 10 neural networks on the dataset for the different dictionaries. The results are similar to those of Section 4.1. Levenshtein and Hamming decoding perform worst. CTC performs slightly better than DynWL.

5 Conclusion

We presented 4 different algorithms for output decoding for neural networks and tested it on two handwriting recognition tasks. The newly introduced DynWL performs slightly worse than CTC. In case of very long output matrices, it still makes sense to use DynWL since CTC is often used in log scale to avoid underflows as reported in [Gra12]. This dramatically increases the time complexity. This is unnecessary for DynWL since underflows cannot appear.

We also hope to convince the reader of the great benefits of CTC. Decoding / spelling correction works much better compared to Levenshtein where much of the network information is lost.

References

[GA09] Emmanuele Grosicki and Haikal El Abed. Icdar 2009 handwriting recognition competition. In *ICDAR*, pages 1398–1402. IEEE Computer Society, 2009.

[GFGS06] Alex Graves, Santiago Fernández, Faustino J. Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In William W. Cohen and Andrew Moore, editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 369–376. ACM, 2006.

[Gra12] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence*. Springer, 2012.

[KLSS02] Alessandro L. Koerich, Yann Leydier, Robert Sabourin, and Ching Y. Suen. A hybrid large vocabulary handwritten word recognition system using neural networks with hidden markov models. In *In proceedings of IWFHR'2002*, pages 99–104, 2002.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[PMM+02] M. Pechwitz, S. S. Maddouri, V. Mrgner, N. Ellouze, and H. Amiri. Ifn/enit-database of handwritten arabic words. In *7th Colloque International Francophone sur l'Ecrit et le Document (CIFED 2002)*, Hammamet, Tunis, 2002.