Appeared in the proceedings of NAACL 2016 (San Diego, June). This version is slightly clarified from the published version; both were prepared in April 2016.

Weighting Finite-State Transductions With Neural Context

Pushpendre Rastogi and **Ryan Cotterell** and **Jason Eisner**
Department of Computer Science, Johns Hopkins University
{pushpendre,ryan.cotterell,eisner}@jhu.edu

Abstract

How should one apply deep learning to tasks such as morphological reinflection, which stochastically edit one string to get another? A recent approach to such sequence-to-sequence tasks is to compress the input string into a vector that is then used to generate the output string, using recurrent neural networks. In contrast, we propose to keep the traditional architecture, which uses a finite-state transducer to score *all possible output strings*, but to augment the scoring function with the help of recurrent networks. A stack of bidirectional LSTMs reads the input string from left-to-right and right-to-left, in order to summarize the *input context* in which a transducer arc is applied. We combine these learned features with the transducer to define a probability distribution over *aligned* output strings, in the form of a weighted finite-state automaton. This reduces hand-engineering of features, allows learned features to examine unbounded context in the input string, and still permits exact inference through dynamic programming. We illustrate our method on the tasks of morphological reinflection and lemmatization.

1 Introduction

Mapping one character sequence to another is a structured prediction problem that arises frequently in NLP and computational linguistics. Common applications include grapheme-to-phoneme (G2P), transliteration, vowelization, normalization, morphology, and phonology. The two sequences may have different lengths.

Traditionally, such settings have been modeled with weighted finite-state transducers (WFSTs) with parametric edge weights (Mohri, 1997; Eisner, 2002). This requires manual design of the transducer states and the features extracted from those states. Alternatively, deep learning has recently been tried for sequence-to-sequence transduction (Sutskever et al., 2014). While training these systems could discover contextual features that a hand-crafted parametric WFST might miss, they dispense with important structure in the problem, namely the *monotonic input-output alignment*. This paper describes a natural hybrid approach that marries simple FSTs with features extracted by recurrent neural networks.

Our novel architecture allows efficient modeling of globally normalized probability distributions over string-valued output spaces, simultaneously with automatic feature extraction. We evaluate on morphological reinflection and lemmatization tasks, showing that our approach strongly outperforms a standard WFST baseline as well as neural sequence-to-sequence models with attention. Our approach also compares reasonably with a state-of-the-art WFST approach that uses task-specific latent variables.

2 Notation and Background

Let Σ_{x} be a discrete input alphabet and Σ_{y} be a discrete output alphabet. Our goal is to define a conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ where $\mathbf{x} \in \Sigma_{\mathrm{x}}^*$ and $\mathbf{y} \in \Sigma_{\mathrm{y}}^*$ and x and y may be of different lengths.

We use italics for characters and boldface for strings. x_j denotes the j^{th} character of $\mathbf{x}$, and $\mathbf{x}_{i:j}$ denotes the substring $x_{i+1}x_{i+2}\cdots x_j$ of length $j-i \geq 0$. Note that $\mathbf{x}_{j:j} = \varepsilon$, the empty string. Let $n = |\mathbf{x}|$.

Our approach begins by hand-specifying an *unweighted* finite-state transducer (FST), F, that nondeterministically maps any well-formed input $\mathbf{x}$ to all appropriate outputs $\mathbf{y}$. An FST is a directed graph whose vertices are called states, and whose arcs are each labeled with some pair $\mathbf{s}:\mathbf{t}$, representing a possible edit of a source substring $\mathbf{s} \in \Sigma_{\mathrm{x}}^*$ into a target substring $\mathbf{t} \in \Sigma_{\mathrm{y}}^*$. A path π from the FST's initial state to its final state represents an alignment of $\mathbf{x}$ to $\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ (respectively) are the concatenations of the $\mathbf{s}$ and $\mathbf{t}$ labels of the arcs along π. In

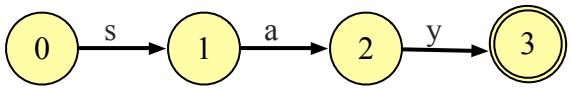

Figure 1: An automaton encoding the English word say.

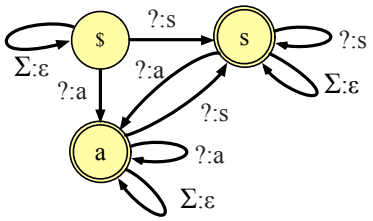

Figure 2: An example transducer F, whose state remembers the most recent *output* character (or \$ if none). Only a few of the states are shown, with all arcs among them. The Σ wildcard matches any symbol in $\Sigma_{\mathtt{x}}$; the "?" wildcard matches the empty string ε *or* any symbol in $\Sigma_{\mathtt{x}}$.

general, two strings $\mathbf{x}, \mathbf{y}$ can be aligned through exponentially many paths, via different edit sequences.

If we represent $\mathbf{x}$ as a straight-line finite-state automaton (Figure 1), then composing $\mathbf{x}$ with F (Figure 2) yields a new FST, G (Figure 3). The paths in G are in 1-1 correspondence with exactly the paths in F that have input $\mathbf{x}$. G can have cycles, allowing outputs of unbounded length.

Each path in G represents an alignment of $\mathbf{x}$ to some string in $\Sigma_{\mathtt{y}}^*$. We say $p(\mathbf{y} \mid \mathbf{x})$ is the *total* probability of all paths in G that align $\mathbf{x}$ to $\mathbf{y}$ (Figure 4).

But how to define the probability of a path? Traditionally (Eisner, 2002), each arc in F would also be equipped with a weight. The weight of a path in F, or the corresponding path in G, is the sum of its arcs' weights. We would then define the probability $p(\pi)$ of a path π in G as proportional to $\exp w(\pi)$, where $w(\cdot) \in \mathbb{R}$ denotes the weight of an object.

The weight of an arc $\textcircled{h} \xrightarrow{\mathbf{s}:\mathbf{t}} \textcircled{h'}$ in F is traditionally defined as a function of features of the edit $\mathbf{s}:\mathbf{t}$ and the names (h, h') of the source and target states. In effect, h summarizes the alignment between the prefixes of $\mathbf{x}$ and $\mathbf{y}$ that precede this edit, and h' summarizes the alignment of the suffixes that follow it.

Thus, while the weight of an edit $\mathbf{s}:\mathbf{t}$ may depend on context, it traditionally does so only through h and h'. So if F has k states, then the edit weight can only distinguish among k different types of preceding or following context.

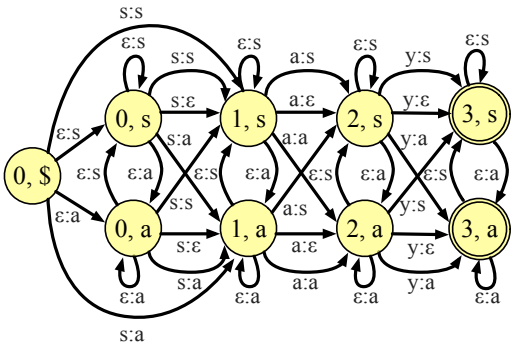

Figure 3: An example of the transducer G, which pairs the string $\mathbf{x}$=say with infinitely many possible strings $\mathbf{y}$. This G was created as the composition of the straight-line input automaton (Figure 1) and the transducer F (Figure 2). Thus, the state of G tracks the states of those two machines: the position in $\mathbf{x}$ and the most recent output character. To avoid a tangled diagram, this figure shows only a few of the states (the start state plus all states of the form $\textcircled{i,s}$ or $\textcircled{i,a}$), with all arcs among them.

3 Incorporating More Context

That limitation is what we aim to correct in this paper, by augmenting our representation of context. Our *contextual weighting* approach will assign weights directly to G's arcs, instead of to F's arcs.

Each arc of G can be regarded as a "token" of an edit arc in F: it "applies" that edit to *a particular substring of* $\mathbf{x}$. It has the form $\textcircled{i,h} \xrightarrow{\mathbf{s}:\mathbf{t}} \textcircled{j,h'}$, and represents the replacement of $\mathbf{x}_{i:j} = \mathbf{s}$ by $\mathbf{t}$. The finite-state composition construction produced this arc of G by combining the arc $\textcircled{h} \xrightarrow{\mathbf{s}:\mathbf{t}} \textcircled{h'}$ in F with the path $\textcircled{i} \overset{\mathbf{s}}{\rightsquigarrow} \textcircled{j}$ in the straight-line automaton representing $\mathbf{x}$. The latter automaton uses integers as state names: it is $\textcircled{0} \xrightarrow{x_1} \textcircled{1} \xrightarrow{x_2} \ldots \xrightarrow{x_n} \textcircled{n}$.

Our top-level idea is to make the weight of this arc in G depend also on $(\mathbf{x}, i, j)$, so that it can consider unbounded input context around the edit's location. Arc weights can now consider arbitrary features of the input $\mathbf{x}$ and the position i, j—exactly like the potential functions of a linear-chain conditional random field (CRF), which also defines $p(\mathbf{y} \mid \mathbf{x})$.

Why not just use a CRF? That would only model a situation that enforced $|\mathbf{y}| = |\mathbf{x}|$ with each character y_i aligned to x_i, since the emissions of a CRF correspond to edits $\mathbf{s} : \mathbf{t}$ with $|\mathbf{s}| = |\mathbf{t}| = 1$. An FST can also allow edits with $|\mathbf{s}| \neq |\mathbf{t}|$, if desired, so it can be fit to $(\mathbf{x}, \mathbf{y})$ pairs of different lengths with unknown alignment, summing over their possible alignments.

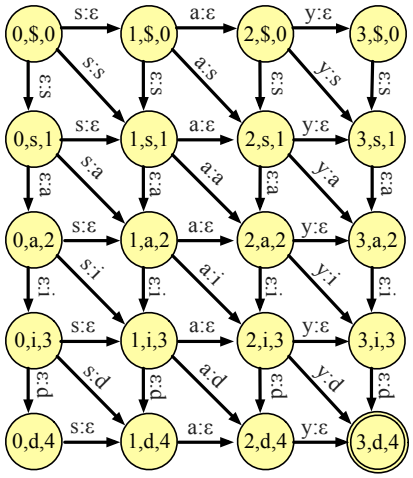

Figure 4: A compact lattice of the exponentially many paths in the transducer G of Figure 3 that align input string $\mathbf{x}$=`say` with output string $\mathbf{y}$=`said`. To find $p(\mathbf{y} \mid \mathbf{x})$, we must sum over these paths (i.e., alignments). The lattice is created by composing G with $\mathbf{y}$, which selects all paths in G that output $\mathbf{y}$. Note that horizontal movement makes progress through $\mathbf{x}$; vertical movement makes progress through $\mathbf{y}$. The lattice's states specialize states in G so that they also record a position in $\mathbf{y}$.

A standard weighted FST F is similar to a dynamic linear-chain CRF. Both are unrolled against the input $\mathbf{x}$ to get a dynamic programming lattice G. But they are not equivalent. By weighting G instead of F, we combine the FST's advantage (aligning unequal-length strings $\mathbf{x}, \mathbf{y}$ via a latent path) with the CRF's advantage (arbitrary dependence on $\mathbf{x}$).

To accomplish this weighting in practice, sections 4–5 present a trainable neural architecture for an arc weight function $w = f(\mathbf{s}, \mathbf{t}, h, h', \mathbf{x}, i, j)$. The goal is to extract continuous features from *all* of $\mathbf{x}$. While our specific architecture is new, we are not the first to replace hand-crafted log-linear models with trainable neural networks (see section 9).

Note that as in a CRF, our arc weights cannot consider arbitrary features of $\mathbf{y}$, only of $\mathbf{x}$. Still, a weight's dependence on states h, h' does let it depend on a *finite* amount of information about $\mathbf{y}$ (also possible in CRFs/HCRFs) and its alignment to $\mathbf{x}$.

In short, our model $p(\mathbf{y} \mid \mathbf{x})$ makes the weight of an $\mathbf{s}:\mathbf{t}$ edit, applied to substring $\mathbf{x}_{i:j}$, depend *jointly* on $\mathbf{s}:\mathbf{t}$ and *two* summaries of the edit's context:

- a finite-state summary (h, h') of its context in the aligned $(\mathbf{x}, \mathbf{y})$ pair, as found by the FST F
- a vector-valued summary of the context in $\mathbf{x}$ only, as found by a recurrent neural network

The neural vector is generally a richer summary of the context, but it considers only the *input*-side context. We are able to efficiently extract these rich features from the *single* input $\mathbf{x}$, but not from each of the *very many* possible outputs $\mathbf{y}$. The job of the FST F is to compute additional features that also depend on the output.[1] Thus our model of $p(\mathbf{y} \mid \mathbf{x})$ is defined by an FST together with a neural network.

4 Neural Context Features

Our arc weight function f will make use of a vector $\boldsymbol{\gamma}_{i:j}$ (computed from $\mathbf{x}, i, j$) to characterize the substring $\mathbf{x}_{i:j}$ that is being replaced, in context. We define $\boldsymbol{\gamma}_{i:j}$ as the concatenation of a left vector $\boldsymbol{\alpha}_j$ (describing the prefix $\mathbf{x}_{0:j}$) and a right vector $\boldsymbol{\beta}_i$ (describing the suffix $\mathbf{x}_{i:n}$), which characterize $\mathbf{x}_{i:j}$ jointly with its left or right context. We use $\boldsymbol{\gamma}_j$ to abbreviate $\boldsymbol{\gamma}_{j-1:j}$, just as x_j abbreviates $\mathbf{x}_{j-1:j}$.

To extract $\boldsymbol{\alpha}_j$, we read the string $\mathbf{x}$ one character at a time with an LSTM (Hochreiter and Schmidhuber, 1997), a type of trainable recurrent neural network that is good at extracting relevant features from strings. $\boldsymbol{\alpha}_j$ is the LSTM's output after j steps (which read $\mathbf{x}_{0:j}$). Appendix A reviews how $\boldsymbol{\alpha}_j \in \mathbb{R}^q$ is computed for $j = 1, \ldots, n$ using the recursive, differentiable update rules of the LSTM architecture.

We also read the string $\mathbf{x}$ *in reverse* with a second LSTM. $\boldsymbol{\beta}_i \in \mathbb{R}^q$ is the second LSTM's output after $n - i$ steps (which read reverse$(\mathbf{x}_{i:n})$).

We regard the two LSTMs together as a BiLSTM function (Graves and Schmidhuber, 2005) that reads $\mathbf{x}$ (Figure 5). For each bounded-length substring $\mathbf{x}_{i:j}$, the BiLSTM produces a characterization $\boldsymbol{\gamma}_{i:j}$ of that substring in context, in $O(n)$ *total* time.

We now define a "deep BiLSTM," which stacks up K BiLSTMs. This deepening is aimed at extracting the kind of rich features that Sutskever et al. (2014) and Vinyals et al. (2015) found so effective in a different structured prediction architecture.

The k^{th}-level BiLSTM (Figure 6) reads a sequence of input vectors $\mathbf{x}_1^{(k)}, \mathbf{x}_2^{(k)}, \ldots, \mathbf{x}_n^{(k)} \in \mathbb{R}^{d^{(k)}}$, and produces a sequence of vectors $\boldsymbol{\gamma}_1^{(k)}, \boldsymbol{\gamma}_2^{(k)}, \ldots, \boldsymbol{\gamma}_n^{(k)} \in \mathbb{R}^{2q}$. At the initial level

[1] Each arc of G is used in a known input context, but could be reused in many output contexts—different *paths* in G. Those contexts are only guaranteed to share h, h'. So the arc weight cannot depend on any *other* features of the output context.

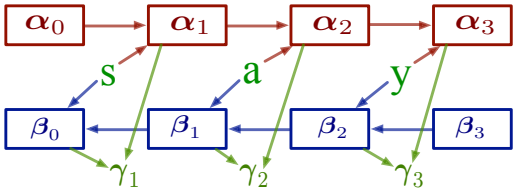

Figure 5: A level-1 BiLSTM reading the word $\mathbf{x}$=say.

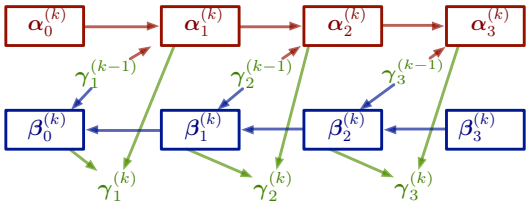

Figure 6: Level $k > 1$ of a deep BiLSTM. (We augment the shown input vectors with level $k-1$'s input vectors.)

$k = 1$, we define $\mathbf{x}_j^{(1)} = \mathbf{e}_{x_j} \in \mathbb{R}^{d^{(1)}}$, a vector embedding of the character $x_j \in \Sigma_{\mathbf{x}}$. For $k > 1$, we take $\mathbf{x}_j^{(k)}$ to be $\boldsymbol{\gamma}_j^{(k-1)}$, concatenated with $\mathbf{x}_j^{(k-1)}$ for good measure. Thus, $d^{(k)} = 2q + d^{(k-1)}$.

After this deep generalization, we define $\boldsymbol{\gamma}_{i:j}$ to be the concatenation of all $\boldsymbol{\gamma}_{i:j}^{(k)}$ (rather than just $\boldsymbol{\gamma}_{i:j}^{(K)}$).

This novel deep BiLSTM architecture has more connections than a pair of deep LSTMs, since $\boldsymbol{\alpha}_j^{(k)}$ depends not only on $\boldsymbol{\alpha}_j^{(k-1)}$ but also on $\boldsymbol{\beta}_j^{(k-1)}$. Thus, while we may informally regard $\boldsymbol{\alpha}_j^{(k)}$ as being a deep summary of the prefix $\mathbf{x}_{0:j}$, it actually depends indirectly on all of $\mathbf{x}$ (except when $k = 1$).

5 The Arc Weight Function

Given the vector $\boldsymbol{\gamma}_{i:j}$, we can now compute the weight of the edit arc $(i,h) \xrightarrow{\mathbf{s}:\mathbf{t}} (j,h)$ in G, namely $w = f(\mathbf{s}, \mathbf{t}, h, h', \mathbf{x}, i, j)$. Many reasonable functions are possible. Here we use one that is inspired by the log-bilinear language model (Mnih and Hinton, 2007):

$$w = (\mathbf{e}_{\mathbf{s}}, \boldsymbol{\gamma}_{i,j}, \mathbf{e}_{x_i}, \mathbf{e}_{x_{j+1}}) \cdot \mathbf{r}_{\mathbf{t},h,h',\text{type}(\mathbf{s}:\mathbf{t})} \quad (1)$$

The first argument to the inner product is an embedding $\mathbf{e}_{\mathbf{s}} \in R^{d^{(1)}}$ of the source substring $\mathbf{s}$, concatenated to the edit's neural context and also (for good measure) its local context.[2] For example, if $|\mathbf{s}| = 1$, i.e. $\mathbf{s}$ is a single character, then we would use the embedding of that character as $\mathbf{e}_{\mathbf{s}}$. Note that the embeddings $e_{\mathbf{s}}$ for $|\mathbf{s}| = 1$ are also used to encode the local context characters and the level-1 BiLSTM input. We learn these embeddings, and they form part of our model's parameter vector $\boldsymbol{\theta}$.

The second argument is a *joint* embedding of the other properties of the edit: the target substring $\mathbf{t}$, the edit arc's state labels from F, and the type of the edit (INS, DEL, or SUB: see section 8). When replacing $\mathbf{s}$ in a particular context, which fixes the first argument, we will prefer those replacements whose $\mathbf{r}$ embeddings yield a high inner product w. We will learn the $\mathbf{r}$ embeddings as well; note that their dimensionality must match that of the first argument.

The model's parameter vector $\boldsymbol{\theta}$ includes the $O((d^{(K)})^2)$ parameters from the $2K$ LSTMs, where $d^{(K)} = O(d^{(1)} + Kq)$. It also $O(d^{(1)}S)$ parameters for the embeddings $\mathbf{e}_{\mathbf{s}}$ of the S different input substrings mentioned by F, and $O(d^{(K)}T)$ for the embeddings $\mathbf{r}_{\mathbf{t},h,h',\text{type}(\mathbf{s}:\mathbf{t})}$ of the T "actions" in F.

6 Training

We train our model by maximizing the conditional log-likelihood objective,

$$\sum_{(\mathbf{x},\mathbf{y}^*)\in\text{dataset}} \log p(\mathbf{y}^* \mid \mathbf{x}) \quad (2)$$

Recall that $p(\mathbf{y}^* \mid \mathbf{x})$ sums over all alignments. As explained by Eisner (2002), it can be computed as the pathsum of the composition $G \circ \mathbf{y}^*$ (Figure 4), divided by the pathsum of G (which gives the normalizing constant for the distribution $p(\mathbf{y} \mid \mathbf{x})$). The pathsum of a weighted FST is the total weight of all paths from the initial state to a final state, and can be computed by the forward algorithm.[3]

Eisner (2002) and Li and Eisner (2009) also explain how to compute the partial derivatives of $p(\mathbf{y}^* \mid \mathbf{x})$ with respect to the arc weights, essentially by using the forward-backward algorithm. We backpropagate further from these partials to find the

[2] Our present implementation handles INS edits (for which $j = i$) a bit differently, using $(\mathbf{e}_{x_{i+1}}, \boldsymbol{\gamma}_{i:i+1}, \mathbf{e}_{x_i}, \mathbf{e}_{x_{i+2}})$ rather than $(\mathbf{e}_{\varepsilon}, \boldsymbol{\gamma}_{i:i}, \mathbf{e}_{x_i}, \mathbf{e}_{x_{i+1}})$. This is conveniently the same vector that is used for all other competing edits at this i position (as they all have $|\mathbf{s}| = 1$ in our present implementation); it provides an extra character of lookahead.

[3] If an FST has cycles, such as the self-loops in the example of Figure 3, then the forward algorithm's recurrence equations become cyclic, and must be solved as a linear system rather than sequentially. This is true regardless of how the FST's weights are defined. (For convenience, our experiments in this paper avoid cycles by limiting consecutive insertions: see section 8.)

gradient of (2) with respect to all our model parameters. We describe our gradient-based maximization procedure in section 10.3, along with regularization.

Our model does not have to be trained with the conditional log likelihood objective. It could be trained with other objectives such as empirical risk or softmax-margin (Li and Eisner, 2009; Gimpel and Smith, 2010), or with error-driven updates such as in the structured perceptron (Collins, 2002).

7 Inference and Decoding

For a new input $\mathbf{x}$ at test time, we can now construct a weighted FST, G, that defines a probability distribution over all *aligned* output strings. This can be manipulated to make various predictions about $\mathbf{y}^*$ and its alignment.

In our present experiments, we find the most probable (highest-weighted) path in G (Dijkstra, 1959), and use its output string $\hat{\mathbf{y}}$ as our prediction. Note that Dijkstra's algorithm is exact; no beam search is required as in some neural sequence models.

On the other hand, $\hat{\mathbf{y}}$ may not be the most probable string—extracting that from a weighted FST is NP-hard (Casacuberta and de la Higuera, 1999). The issue is that the total probability of each $\mathbf{y}$ is split over many paths. Still, this is a well-studied problem in NLP. Instead of the Viterbi approximation, we could have used a better approximation, such as crunching (May and Knight, 2006) or variational decoding (Li et al., 2009). We actually did try crunching the 10000-best outputs but got no significant improvement, so we do not report those results.

8 Transducer Topology

In our experiments, we choose F to be a simple contextual edit FST as illustrated in Figure 2. Just as in Levenshtein distance (Levenshtein, 1966), it allows all edits $\mathbf{s} : \mathbf{t}$ where $|\mathbf{s}| \leq 1, |\mathbf{t}| \leq 1, |\mathbf{s}| + |\mathbf{t}| \neq 0$. We consider the edit type to be INS if $\mathbf{s} = \varepsilon$, DEL if $\mathbf{t} = \varepsilon$, and SUB otherwise. Note that copying of a character is a SUB edit with $\mathbf{s} = \mathbf{t}$.

For a "memoryless" edit process (Ristad and Yianilos, 1996), the FST would require only a single state. By contrast, we use $|\Sigma_{\mathbf{x}}| + 1$ states, where each state records the most recent output character (initially, a special "beginning-of-string" symbol \$). That is, the state label h is the "history" output character immediately before the edit $\mathbf{s} : \mathbf{t}$, so the state label h' is the history before the *next* edit, namely the final character of $h\mathbf{t}$. For edits other than DEL, $h\mathbf{t}$ is a bigram of $\mathbf{y}$, which can be evaluated (in context) by the arc weight function $w = f(\mathbf{s}, \mathbf{t}, h, h', \mathbf{x}, i, j)$.

Naturally, a weighted version of this FST F is far too simple to do well on real NLP tasks (as we will show in our experiments). The magic comes from instead weighting G so that we can pay attention to the input context $\boldsymbol{\gamma}_{i:j}$.

The above choice of F corresponds to the "$(0, 1, 1)$ topology" in the more general scheme of Cotterell et al. (2014). For practical reasons, we actually modify it to limit the number of consecutive INS edits to 3.[4] This trick bounds $|\mathbf{y}|$ to be $< 4 \cdot (|\mathbf{x}| + 1)$, ensuring that the pathsums in section 6 are finite regardless of the model parameters. This simplifies both the pathsum algorithm and the gradient-based training (Dreyer, 2011). Less importantly, since G becomes acyclic, Dijkstra's algorithm in section 7 simplifies to the Viterbi algorithm.

9 Related Work

Our model adds to recent work on linguistic sequence transduction using deep learning.

Graves and Schmidhuber (2005) combined BiLSTMs with HMMs. Later, "sequence-to-sequence" models were applied to machine translation by Sutskever et al. (2014) and to parsing by Vinyals et al. (2015). That framework did not model any alignment between $\mathbf{x}$ and $\mathbf{y}$, but adding an "attention" mechanism provides a kind of soft alignment that has improved performance on MT (Bahdanau et al., 2015). Faruqui et al. (2016) apply these methods to morphological reinflection (the only other application to morphology we know of). Grefenstette et al. (2015) recently augmented the sequence-to-sequence framework with a continuous analog of stack and queue data structures, to better handle long-range dependencies often found in language.

Some recent papers have used LSTMs or BiLSTMs, as we do, to define probability distributions over action sequences that operate directly on an input sequence. Such actions are aligned to the in-

[4] This multiplies the number of states 4-fold, since each state must also record a count in $[0, 3]$ of immediately preceding INS edits. No INS edit arc is allowed from a state with counter 3. The counter is not considered by the arc weight function.

put. For example, Andor et al. (2016) score edit sequences using a globally normalized model, while Yao and Zweig (2015) and Dyer et al. (2015) evaluate the local probability of a transduction or parsing action given past actions (and any structure they built) and future words. These architectures are powerful because their LSTMs can globally assess the *output*; but as a result they do not permit dynamic programming and must fall back on beam search.

Our use of dynamic programming for efficient exact inference has long been common in non-neural architectures for sequence transduction, including FST systems that allow "phrasal" replacements $\mathbf{s} : \mathbf{t}$ where $|\mathbf{s}|, |\mathbf{t}| > 1$ (Chen, 2003; Jiampojamarn et al., 2007; Bisani and Ney, 2008). Our work augments these FSTs with neural networks, much as others have augmented CRFs. In this vein, Durrett and Klein (2015) augment a CRF parser (Finkel et al., 2008) to score constituents with a feedforward neural network. Likewise, FitzGerald et al. (2015) employ feedforward nets as a factor in a graphical model for semantic role labeling. Many CRFs have incorporated feedforward neural networks (Bridle, 1990; Peng et al., 2009; Do and Artieres, 2010; Vinel et al., 2011; Fujii et al., 2012; Chen et al., 2015, and others). Some work augments CRFs with BiLSTMs: Huang et al. (2015) report results on part-of-speech tagging and named entity recognition with a linear-chain CRF-BiLSTM, and Kong et al. (2015) on Chinese word segmentation and handwriting recognition with a semi-CRF-BiLSTM.

10 Experiments

We evaluated our approach on two morphological generation tasks: reinflection (section 10.1) and lemmatization (section 10.2). In reinflection, the goal is to transduce verbs from one inflected form into another, whereas in lemmatization, the goal is to reduce an inflected verb to its root form.

We compare our WFST-LSTM against two standard baselines, a WFST with hand-engineered features and the Moses phrase-based MT system (Koehn et al., 2007), as well as the more complex latent-variable model of Dreyer et al. (2008). The comparison with Dreyer et al. (2008) is of noted interest since their latent variables are structured particularly for morphological transduction tasks—we are directly testing the ability of the LSTM to structure its hidden layer as effectively as linguistically motivated latent-variables. Additionally, we provide detailed ablation studies and learning curves which show that our neural-WFSA hybrid model can generalize even with very low amounts of training data.

10.1 Morphological Reinflection

Following Dreyer (2011), we conducted our experiments on the following transduction tasks from the CELEX (Baayen et al., 1993) morphological database: 13SIA $\mapsto$ 13SKE, 2PIE $\mapsto$ 13PKE, 2PKE $\mapsto$ z and rP $\mapsto$ pA.[5] We refer to these tasks as 13SIA, 2PIE, 2PKE and rP, respectively.

Concretely, each task requires us to map a German inflection into another inflection. Consider the 13SIA task and the German verb `abreiben` ("to rub off"). We require the model to learn to map a past tense form `abrieb` to a present tense form `abreibe`—this involves a combination of stem change and affix generation. Sticking with the same verb `abreiben`, task 2PIE requires the model to transduce `abreibt` to `abreiben`—this requires an insertion and a substitution at the end. The tasks 2PKE and rP are somewhat more challenging since performing well on these tasks requires the model to learn complex transduction: `abreiben` to `abzureiben` and `abreibt` to `abgerieben`, respectively. These are complex transductions with phenomenon like infixation in specific contexts (`ab`**`zu`**`rieben`) and circumfixation (`ab`**`ge`**`rieb`**`en`**) along with additional stem and affix changes. See Dreyer (2011) for more details and examples of these tasks.

We use the datasets of Dreyer (2011). Each experiment sampled a different dataset of 2500 examples from CELEX, dividing this into 500 training + 1000 validation + 1000 test examples. Like them, we report exact-match accuracy on the test examples, averaged over 5 distinct experiments of this form. We also report results when the training and validation data are swapped in each experiment, which doubles the training size.

[5] Glossary: 13SIA=*1st/3rd sg. ind. past*; 13SKE=*1st/3rd sg. subjunct. pres.*; 2PIE=*2nd pl. ind. pres.*; 13PKE=*1st/3rd pl. subjunct. pres.*; 2PKE=*2nd pl. subjunct. pres.;* z=*infinitive*; rP=*imperative pl.*; pA=*past part*.

10.2 Lemmatization

Lemmatization is a special case of morphological reinflection where we map an inflected form of a word to its lemma (canonical form), i.e., the target inflection is fixed. This task is quite useful for NLP, as dictionaries typically list only the lemma for a given lexical entry, rather than all possible inflected forms. In the case of German verbs, the lemma is taken to be the infinitive form, e.g., we map the past participle `abgerieben` to the infinitive `abreiben`.

Following Dreyer (2011), we use a subset of the lemmatization dataset created by Wicentowski (2002) and perform 10-fold experiments on four languages: Basque (5843), English (4915), Irish (1376) and Tagalog (9545), where the numbers in parenthesis indicate the total number of data pairs available. For each experimental fold the total data was divided into train, development and test sets in the proportion of 80:10:10 and we report test accuracy averaged across folds.

10.3 Settings and Training Procedure

We set the hyperparameters of our model to $K = 4$ (stacking depth), $d^{(1)} = 10$ (character embedding dimension), and $q = 15$ (LSTM state dimension). The alphabets Σ_{x} and Σ_{y} are always equal; their size is language-dependent, typically ≈ 26 but larger in languages like Basque and Irish where our datasets include properly accented characters. With $|\Sigma| = 26$ and the above settings for the hyperparameters, the number of parameters in our models is $352,801$.

We optimize these parameters through stochastic gradient descent of the negative log-likelihood objective, and regularize the training procedure through dropout accompanied with gradient clipping and projection of parameters onto L2-balls with small radii, which is equivalent to adding a group-ridge regularization term to the training objective. The learning rate decay schedule, gradient clipping threshold, radii of L2-balls, and dropout frequency were tuned by hand on development data.

In our present experiments, we made one change to the architecture. Treating copy edits like other SUB edits led to poor performance: the system was unable to learn that all SUB edits with $\mathbf{s} = \mathbf{t}$ were extremely likely. In the experiments reported here, we addressed the problem by simply tying the weights of all copy edits regardless of context, bypassing (1) and instead setting $w = c$ where c is a learned parameter of the model. See section 11 for discussion.

Model	13SIA	2PIE	2PKE	rP
Moses15	85.3	94.0	82.8	70.8
Dreyer (Backoff)	82.8	88.7	74.7	69.9
Dreyer (Lat-Class)	84.8	93.6	75.7	81.8
Dreyer (Lat-Region)	**87.5**	93.4	**87.4**	**84.9**
BiLSTM-WFST	85.1	**94.4**	85.5	83.0
Model Ensemble	85.8	**94.6**	86.0	83.8
BiLSTM-WFST (2 × Data)	*87.6*	*94.8*	*88.1*	*85.7*

Table 1: Exact match accuracy on the morphological reinflection task. All the results in the first half of the table are taken from Dreyer (2011), whose experimental setup we copied exactly. The Moses15 result is obtained by applying the SMT toolkit Moses (Koehn et al., 2007) over letter strings with 15-character context windows. Dreyer (Backoff) refers to the *ngrams+x* model which has access to all the "backoff features." Dreyer (Lat-Class) is the *ngrams+x+latent class* model, and Dreyer (Lat Region) refers to the *ngrams+x+latent class + latent change region* model. The "Model Ensemble" row displays the performance of an ensemble including our full model and the 7 models that we performed ablation on. In each column, we boldfaced the highest result and those that were not significantly worse (sign test, $p < 0.05$). Finally, the last row reports the performance of our BiLSTM-WFST when trained on twice as much data.

10.4 Results

Table 1 and 2 show our results. We can see that our proposed BiLSTM-WFST model always outperforms all but the most complex latent-variable model of Dreyer (2011); it is competitive with that model, but only beats it once individually. All of Dreyer's models include output trigram features, while we only use bigrams.

Figure 7 shows learning curves for the 13SIA and 2PKE tasks: test accuracy when we train on less data. Curiously, at 300 data points the performance of our model is tied to Dreyer (2011). We also note that our model always outperforms the Moses15 baseline on all training set sizes except on the 2PKE task with 50 training samples.

In general, the results from our experiments are a promising indicator that LSTMs are capable of extracting linguistically relevant features for morphology. Our model outperforms all baselines, and is competitive with and sometimes surpasses the

Model	Basque	English	Irish	Tagalog
Base (W)	85.3	91.0	43.3	0.3
WFAffix (W)	80.1	93.1	70.8	81.7
ngrams (D)	91.0	92.4	96.8	80.5
ngrams + x (D)	91.1	93.4	97.0	83.0
ngrams + x + l (D)	**93.6**	**96.9**	**97.9**	88.6
BiLSTM-WFST	91.5	94.5	**97.9**	**97.4**

Table 2: Lemmatization results on Basque, English, Irish and Tagalog. Comparison systems marked with (W) are taken from Wicentowski (2002) and systems marked with a (D) are taken from Dreyer (2011). We outperform baselines on all languages and are competitive with the latent-variable approach (ngrams + x + l), beating it in two cases: Irish and Tagalog.

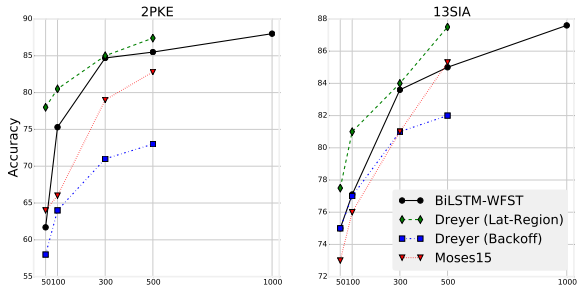

Figure 7: Learning Curves

latent-variable model of Dreyer et al. (2008) *without* any of the hand-engineered features or linguistically inspired latent variables.

On morphological reinflection, we outperform all of Dreyer et al's models on 2PIE, but fall short of his latent-change region model on the other tasks (outperforming the other models). On lemmatization, we outperform all of Wicentowski's models on all the languages, and all of Dreyer et al.'s models on Irish and Tagalog, but not Dreyer et al.'s latent-variable approach on Basque or English. This suggests that perhaps further gains are possible through using something like Dreyer's FST as our F. Indeed, this would be compatible with recent work that gets best results by *combining* automatically learned neural features and hand-engineered features.

10.5 Analysis of Results

We analyzed our lemmatization errors for all the languages on one fold of the datasets. On the English lemmatization task, 7 of our 27 errors simply copied the input word to the output: `ate`, `kept`, `went`, `taught`, `torn`, `paid`, `strung`. This suggests

Systems	13SIA	2PIE	2PKE	rP
Deep BiLSTM w/ Tying	**86.8**	**94.8**	**87.9**	**81.1**
Deep BiLSTM (No Context)	**86.5**	**94.3**	**87.8**	78.8
Deep BiLSTMs w/o Copying	**86.5**	**94.6**	86.5	**80.7**
Shallow BiLSTM	**86.4**	**94.7**	86.1	**80.6**
Bi-Deep LSTM	86.1	94.2	86.5	78.6
Deep MonoLSTM	84.0	93.8	85.6	67.3
Shallow MonoLSTM	84.2	**94.5**	84.9	68.2
No LSTM (Local Context)	83.6	88.5	83.2	68.0
Deep BiLSTM w/o Tying	69.7	78.5	77.9	66.7
No LSTM (No Context)	70.7	84.9	72.4	64.1
Seq2Seq-Att-4Layer	76.0	91.4	81.2	79.3
Seq2Seq-Att-1Layer	77.2	89.6	82.1	79.1
Seq2Seq-4Layer	2.5	5.2	11.5	6.4
Seq2Seq-1Layer	9.1	11.1	14.1	11.9

Table 3: Ablation results for reinflection: Exact-match accuracy on the validation data of different systems. Like the test accuracies, these average over 5 experiments.

that our current aggressive parameter tying for copy edits may predict a high probability for a copy edit even in contexts that should not favor it.

Also we found that the FST sometimes produced non-words while lemmatizing the input verbs. For example it mapped `picnicked` ↦ `picnick`, `happen` ↦ `hapen`, `exceed` ↦ `excy` and `lining` ↦ `lin`. Since these strings would be rare in a corpus, many such errors could be avoided by a reranking approach that combined the FST's path score with a string frequency feature.

In order to better understand our architecture and the importance of its various components, we performed an ablation study on the validation portions of the morphological induction datasets, shown in Table 3. We can see in particular that using a BiLSTM instead of an LSTM, increasing the depth of the network, and including local context all helped to improve the final accuracy.

"Deep BiLSTM w/ Tying" refers to our complete model. The other rows are ablation experiments—architectures that are the same as the first row except in the specified way. "Deep BiLSTM (No Context)" omits local context $\mathbf{e}_{x_i}, \mathbf{e}_{x_{j+1}}$ from (1). "Deep BiLSTMs w/o Copying" does not concatenate a copy of $\mathbf{x}_i^{(k-1)}$ into $\mathbf{x}_i^{(k)}$ and simplifies $\boldsymbol{\gamma}_{i:j}$ to be $\boldsymbol{\gamma}_{i:j}^{(K)}$ only. "Shallow BiLSTM" reduces K from 4 to 1. "Bi-Deep LSTM" replaces our deep BiLSTM with two deep LSTMs that run in opposite directions but do not interact with each other. "Deep MonoLSTM" redefines β_i to be the empty vector,g i.e. it replaces

the deep BiLSTM with a deep left-to-right LSTM. "Shallow MonoLSTM" replaces the deep BiLSTM with a shallow left-to-right LSTM. "No LSTM (Local Context)" omits $\gamma_{i:j}$ from the weight function altogether. "Deep BiLSTM w/o Tying" does not use the parameter tying heuristic for copy edits. "No LSTM (No Context)" is the simplest model that we consider. It removes $\gamma_{i:j}, \mathbf{e}_{x_i}$ and $\mathbf{e}_{x_{j+1}}$ and in fact, it is precisely a weighting of the edits in our original FST F, without further considering the context in which an edit is applied.

Finally, to compare the performance of our method to baseline neural encoder-decoder models, we trained 1-layer and 4-layer neural sequence-to-sequence models with and without attention, using the publicly available `morph-trans` toolkit (Faruqui et al., 2016). We show the performance of these models in the lower half of the table. The results consistently show that sequence-to-sequence transduction models that lack the constraints of monotonic alignment perform worse than our proposed models on morphological transduction tasks.

11 Future Work: Possible Extensions

As neither our FST-LSTM model or the latent-variable WFST model of Dreyer et al. (2008) uniformly outperforms the other, a future direction is to improve the FST we use in our model—e.g., by augmenting its states with explicit latent variables. Other improvements to the WFST would be to increase the amount of history h stored in the states, and to allow $\mathbf{s}, \mathbf{t}$ to be longer than a single character, which would allow the model to segment $\mathbf{x}$.

We are not committed to the arc weight function in (1), and we imagine that further investigation here could improve performance. The goal is to define the weight of the arc $\textcircled{h} \xrightarrow{\mathbf{s}:\mathbf{t}} \textcircled{h'}$ in the context summarized by $\boldsymbol{\alpha}_i, \boldsymbol{\beta}_j$ (the context around $\mathbf{x}_{i:j} = \mathbf{s}$) and/or by $\boldsymbol{\alpha}_j, \boldsymbol{\beta}_i$ (which incorporate $\mathbf{s}$ as well).

Eq. (1) could be replaced by any parametric function of the variables $(\mathbf{s}, \mathbf{t}, h, h', \boldsymbol{\alpha}_i, \boldsymbol{\alpha}_j, \boldsymbol{\beta}_i, \boldsymbol{\beta}_j)$—e.g., a neural network or a multilinear function. This function might depend on learned embeddings of the separate objects $\mathbf{s}, \mathbf{t}, h, h'$, but also on learned joint embeddings of *pairs* of these objects (which adds finer-grained parameters), or hand-specified *properties* of the objects such as their phonological features (which adds backoff parameters).

Such a scheme might set $w = \mathbf{f}(\mathbf{c}) \cdot \mathbf{g}(\mathbf{r})$, where $\mathbf{r}$ is some encoding of the arc $(\mathbf{s}, \mathbf{t}, h, h')$, $\mathbf{c}$ is some encoding of the context $(\boldsymbol{\alpha}_i, \boldsymbol{\alpha}_j, \boldsymbol{\beta}_i, \boldsymbol{\beta}_j, \mathbf{s}, h, h')$, and $\mathbf{f}$ and $\mathbf{g}$ are learned linear or nonlinear vector-valued functions. Note that this formulation lets the objects $\mathbf{s}, h, h'$ play a dual role—they could appear as part of the arc and also as part of the context. This is because we must judge whether $\mathbf{s} = \mathbf{x}_{i:j}$ (with "label" $\textcircled{h} \longrightarrow \textcircled{h'}$) is an appropriate input segment given the string context around $\mathbf{x}_{i:j}$, but if this segment is chosen, in turn it provides additional context to judge whether $\mathbf{t}$ is an appropriate output.

One could also revisit the stacking mechanism. Rather than augmenting a character x_j at the next level with its context $(\boldsymbol{\alpha}_j, \boldsymbol{\beta}_i)$ (where $i = j-1$), one might augment it with the richer context $\mathbf{f}(\mathbf{c})$ or even the arc-in-context representation $\mathbf{f}(\mathbf{c}) \odot \mathbf{g}(\mathbf{r})$, where $\mathbf{t}, h, h'$ are chosen to maximize w (max-pooling).

High-probability edits $\mathbf{s} : \mathbf{t}$ typically have $\mathbf{t} \approx \mathbf{s}$: a perfect copy, or a modified copy that changes just one or two features such as phonological voicing or orthographic capitalization. Thus, we may wish to learn a shared set of embeddings for $\Sigma_{\mathrm{x}} \cup \Sigma_{\mathrm{y}}$, and make the weight of $\mathbf{s} : \mathbf{t}$ arcs depend on features of the "discrepancy vector" $\mathbf{e}_{\mathbf{t}} - \mathbf{e}_{\mathbf{s}}$, such as this vector's components[6] and their absolute magnitudes, which would signal discrepancies of various sorts.

12 Conclusions

We have presented a hybrid FST-LSTM architecture for string-to-string transduction tasks. This approach combines classical finite-state approaches to transduction and newer neural approaches. We weight the same FST arc *differently* in different contexts, and use LSTMs to *automatically* extract features that determine these weights. This reduces the need to engineer a complex topology for the FST or to hand-engineer its weight features. We evaluated one such model on the tasks of morphological reinflection and lemmatization. Our approach outperforms several baselines and is competitive with (and sometimes surpasses) a latent-variable model hand-crafted for morphological transduction tasks.

[6] In various directions—perhaps just the basis directions, which might come to encode distinctive features, via training.

Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency under the Deep Exploration and Filtering of Text (DEFT) Program, agreement number FA8750-13-2-001; by the National Science Foundation under Grant No. 1423276; and by a DAAD fellowship to the second author. We would like to thank Markus Dreyer for providing the datasets and Manaal Faruqui for providing implementations of the baseline sequence-to-sequence models. Finally we would like to thank Mo Yu, Nanyun Peng, Dingquan Wang and Elan Hourticolon-Retzler for helpful discussions and early prototypes for some of the neural network architectures.

References

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. Available at `arXiv.org` as `arXiv:1603.06042`, March.

R. Harald Baayen, Richard Piepenbrock, and Rijn van H. 1993. The CELEX lexical data base on CD-ROM.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*.

Maximilian Bisani and Hermann Ney. 2008. Joint-sequence models for grapheme-to-phoneme conversion. *Speech Communication*, 50(5):434–451.

John S. Bridle. 1990. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Proceedings of NIPS*, pages 211–217.

Francisco Casacuberta and Colin de la Higuera. 1999. Optimal linguistic decoding is a difficult computational problem. *Pattern Recognition Letters*, 20(8):813–821.

Liang-Chieh Chen, Alexander Schwing, Alan Yuille, and Raquel Urtasun. 2015. Learning deep structured models. In *Proceedings of ICML*, pages 1785–1794.

Stanley F. Chen. 2003. Conditional and joint models for grapheme-to-phoneme conversion. In *Proceedings of EUROSPEECH*.

Michael Collins. 2002. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Philadelphia, July.

Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic contextual edit distance and probabilistic FSTs. In *Proceedings of ACL*, pages 625–630, June.

E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1).

Trinh-Minh-Tri Do and Thierry Artieres. 2010. Neural conditional random fields. In *Proceedings of AISTATS*. JMLR.

Markus Dreyer, Jason R. Smith, and Jason Eisner. 2008. Latent-variable modeling of string transductions with finite-state methods. In *Proceedings of EMNLP*, pages 1080–1089, October.

Markus Dreyer. 2011. *A Non-Parametric Model for the Discovery of Inflectional Paradigms from Plain Text Using Graphical Models over Strings*. Ph.D. thesis, Johns Hopkins University, Baltimore, MD, April.

Greg Durrett and Dan Klein. 2015. Neural CRF parsing. In *Proceedings of ACL*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of ACL-IJCNLP*, pages 334–343, July.

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of ACL*, pages 1–8, July.

Manaal Faruqui, Yulia Tsvetkov, Graham Neubig, and Chris Dyer. 2016. Morphological inflection generation using character sequence to sequence learning. In *Proceedings of NAACL*. Code available at `https://github.com/mfaruqui/morph-trans`.

Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. 2008. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL*, pages 959–967.

Nicholas FitzGerald, Oscar Täckström, Kuzman Ganchev, and Dipanjan Das. 2015. Semantic role labeling with neural network factors. In *Proceedings of EMNLP*, pages 960–970.

Yasuhisa Fujii, Kazumasa Yamamoto, and Seiichi Nakagawa. 2012. Deep-hidden conditional neural fields for continuous phoneme speech recognition. In *Proceedings of IWSML*.

Kevin Gimpel and Noah A. Smith. 2010. Softmax-margin CRFs: Training log-linear models with cost functions. In *Proceedings of NAACL-HLT*, pages 733–736, June.

Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610.

Alex Graves. 2012. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proceedings of NIPS*, pages 1819–1827.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*.

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Sittichai Jiampojamarn, Grzegorz Kondrak, and Tarek Sherif. 2007. Applying many-to-many alignments and hidden Markov models to letter-to-phoneme conversion. In *Proceedings of NAACL-HLT*, pages 372–379.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of ACL (Interactive Poster and Demonstration Sessions)*, pages 177–180.

Lingpeng Kong, Chris Dyer, and Noah A. Smith. 2015. Segmental recurrent neural networks. *arXiv preprint arXiv:1511.06018*.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Zhifei Li and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of EMNLP*, pages 40–51, August.

Zhifei Li, Jason Eisner, and Sanjeev Khudanpur. 2009. Variational decoding for statistical machine translation. In *Proceedings of ACL*, pages 593–601.

Jonathan May and Kevin Knight. 2006. A better n-best list: Practical determinization of weighted finite tree automata. In *Proceedings of NAACL*, pages 351–358.

Andriy Mnih and Geoffrey Hinton. 2007. Three new graphical models for statistical language modelling. In *Proceedings of ICML*, pages 641–648. ACM.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Jian Peng, Liefeng Bo, and Jinbo Xu. 2009. Conditional neural fields. In *Proceedings of NIPS*, pages 1419–1427.

Eric Sven Ristad and Peter N. Yianilos. 1996. Learning string edit distance. Technical Report CS-TR-532-96, Princeton University, Department of Computer Science, October. Revised October 1997.

Ilya Sutskever, Oriol Vinyals, and Quoc Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of NIPS*.

Antoine Vinel, Trinh Minh Tri Do, and Thierry Artieres. 2011. Joint optimization of hidden conditional random fields and non-linear feature extraction. In *Proceedings of ICDAR*, pages 513–517. IEEE.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Proceedings of NIPS*, pages 2755–2763.

Richard Wicentowski. 2002. *Modeling and Learning Multilingual Inflectional Morphology in a Minimally Supervised Framework*. Ph.D. thesis, Johns Hopkins University.

Kaisheng Yao and Geoffrey Zweig. 2015. Sequence-to-sequence neural net models for grapheme-to-phoneme conversion. In *Proceedings of INTERSPEECH*.

Appendices

A LSTM Formulas

Lowercase bold letters represent real vectors. Uppercase letters represent real matrices of the apprporiate dimension. Subscripts distinguish different vectors and matrices and not entries of a vector. $\odot$ denotes element-wise multiplication, and σ is the sigmoid function $\sigma(z) = \frac{1}{1+\exp(-z)}$, applied element-wise. See Graves (2012) for the rationale behind these definitions:

$$\mathbf{c}_i \leftarrow \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \mathbf{z}_i \in \mathbb{R}^q \tag{3}$$

$$\boldsymbol{\alpha}_i \leftarrow \mathbf{o}_i \odot (2\sigma(2\mathbf{c}_i) - 1) \in \mathbb{R}^q \tag{4}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ and

$$\mathbf{i}_i \leftarrow \sigma(W_{\mathrm{i}}\mathbf{x}_i + U_{\mathrm{i}}\boldsymbol{\alpha}_{i-1} + \mathbf{d}_{\mathrm{i}}) \in \mathbb{R}^q \tag{5}$$

$$\mathbf{f}_i \leftarrow \sigma(W_{\mathrm{f}}\mathbf{x}_i + U_{\mathrm{f}}\boldsymbol{\alpha}_{i-1} + \mathbf{d}_{\mathrm{f}}) \in \mathbb{R}^q \tag{6}$$

$$\mathbf{z}_i \leftarrow 2\sigma(W_{\mathrm{z}}\mathbf{x}_i + U_{\mathrm{z}}\boldsymbol{\alpha}_{i-1} + \mathbf{d}_{\mathrm{z}}) - 1 \in \mathbb{R}^q \tag{7}$$

$$\mathbf{o}_i \leftarrow \sigma(W_{\mathrm{o}}\mathbf{x}_i + U_{\mathrm{o}}\boldsymbol{\alpha}_{i-1} + \mathbf{d}_{\mathrm{o}}) \in \mathbb{R}^q \tag{8}$$

$\mathbf{c}_i$ denotes the LSTM's hidden state, which is is obtained by interpolating $\mathbf{c}_{i-1}$ element-wise with a vector $\mathbf{z}_i$ that is derived from the input. We take $\mathbf{c}_0 = \mathbf{0}$. $\mathbf{c}_i$ is then squashed and rescaled element-wise to determine the output $\boldsymbol{\alpha}_i$. The vector $\mathbf{z}_i$ depends on the entire previous output $\boldsymbol{\alpha}_{i-1}$ and the entire input $\mathbf{x}_i$, as do the rescaling and interpolation coefficients—these non-element-wise definitions are in the last 4 lines. The parameter vector $\boldsymbol{\theta}$ includes the elements of M_m (for all $M \in \{W, U, \mathbf{d}\}$ and $m \in \{\mathrm{i}, \mathrm{f}, \mathrm{o}, \mathrm{z}\}$) and the elements of $\boldsymbol{\alpha}_0$.