Minimally Supervised Number Normalization

Kyle Gorman and **Richard Sproat**
Google, Inc.
111 8th Ave., New York, NY, USA

Abstract

We propose two models for verbalizing numbers, a key component in speech recognition and synthesis systems. The first model uses an end-to-end recurrent neural network. The second model, drawing inspiration from the linguistics literature, uses finite-state transducers constructed with a minimal amount of training data. While both models achieve near-perfect performance, the latter model can be trained using several orders of magnitude less data than the former, making it particularly useful for low-resource languages.

1 Introduction

Many speech and language applications require text tokens to be converted from one form to another. For example, in text-to-speech synthesis, one must convert digit sequences (*32*) into number names (*thirty-two*), and appropriately verbalize date and time expressions (*12:47* → *twelve forty-seven*) and abbreviations (*kg* → *kilograms*) while handling allomorphy and morphological concord (e.g., Sproat, 1996). Quite a bit of recent work on SMS (e.g., Beaufort et al., 2010) and text from social media sites (e.g., Yang and Eisenstein, 2013) has focused on detecting and expanding novel abbreviations (e.g., *cn u plz hlp*). Collectively, such conversions all fall under the rubric of *text normalization* (Sproat et al., 2001), but this term means radically different things in different applications. For instance, it is not necessary to detect and verbalize dates and times when preparing social media text for downstream information extraction, but this is essential for speech applications.

While expanding novel abbreviations is also important for speech (Roark and Sproat, 2014), numbers, times, dates, measure phrases and the like are far more common in a wide variety of text genres. Following Taylor (2009), we refer to categories such as cardinal numbers, times, and dates—each of which is semantically well-circumscribed—as *semiotic classes*. Some previous work on text normalization proposes minimally-supervised machine learning techniques for normalizing specific semiotic classes, such as abbreviations (e.g., Chang et al., 2002; Pennell and Liu, 2011; Roark and Sproat, 2014). This paper continues this tradition by contributing minimally-supervised models for normalization of cardinal number expressions (e.g., *ninety-seven*). Previous work on this semiotic class include formal linguistic studies by Corstius (1968) and Hurford (1975) and computational models proposed by Sproat (1996; 2010) and Kanis et al. (2005). Of all semiotic classes, numbers are by far the most important for speech, as cardinal (and ordinal) numbers are not only semiotic classes in their own right, but knowing how to verbalize numbers is important for most of the other classes: one cannot verbalize times, dates, measures, or currency expressions without knowing how to verbalize that language's numbers as well.

One computational approach to number name verbalization (Sproat, 1996; Kanis et al., 2005) employs a cascade of two finite-state transducers (FSTs). The first FST factors the integer, expressed as a digit sequence, into sums of products of powers of ten (i.e., in the case of a base-ten number system). This is composed with a second FST that defines how the

Transactions of the Association for Computational Linguistics, vol. 4, pp. 507–519, 2016. Action Editor: Jason Eisner.
Submission batch: 3/2016; Revision batch: 5/2016; Published 11/2016.
©2016 Association for Computational Linguistics. Distributed under a CC-BY 4.0 license.

numeric factors are verbalized, and may also handle allomorphy or morphological concord in languages that require it. Number names can be relatively easy (as in English) or complex (as in Russian; Sproat, 2010) and thus these FSTs may be relatively easy or quite difficult to develop. While the Google text-to-speech (TTS) (see Ebden and Sproat, 2014) and automatic speech recognition (ASR) systems depend on hand-built number name grammars for about 70 languages, developing these grammars for new languages requires extensive research and labor. For some languages, a professional linguist can develop a new grammar in as little as a day, but other languages may require days or weeks of effort. We have also found that it is very common for these hand-written grammars to contain difficult-to-detect errors; indeed, the computational models used in this study revealed several long-standing bugs in hand-written number grammars.

The amount of time, effort, and expertise required to produce error-free number grammars leads us to consider machine learning solutions. Yet it is important to note that number verbalization poses a dauntingly high standard of accuracy compared to nearly all other speech and language tasks. While one might forgive a TTS system that reads the ambiguous abbreviation *plz* as *plaza* rather than the intended *please*, it would be inexcusable for the same system to ever read *72* as *four hundred seventy two*, even if it rendered the vast majority of numbers correctly.

To set the stage for this work, we first (§2–3) briefly describe several experiments with a powerful and popular machine learning technique, namely recurrent neural networks (RNNs). When provided with a large corpus of parallel data, these systems are highly accurate, but may still produce occasional errors, rendering it unusable for applications like TTS.

In order to give the reader some background on the relevant linguistic issues, we then review some cross-linguistic properties of cardinal number expressions and propose a finite-state approach to number normalization informed by these linguistic properties (§4). The core of the approach is an algorithm for inducing language-specific number grammar rules. We evaluate this technique on data from four languages.

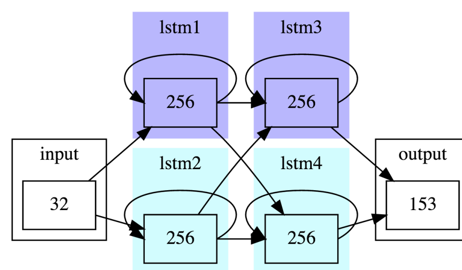

Figure 1: The neural net architecture for the preliminary Russian cardinal number experiments. Purple LSTM layers perform forwards transitions and green LSTM layers perform backwards transitions. The output is produced by a CTC layer with a softmax activation function. Input tokens are characters and output tokens are words.

2 Preliminary experiment with recurrent neural networks

As part of a separate strand of research, we have been experimenting with various recurrent neural network (RNN) architectures for problems in text normalization. In one set of experiments, we trained RNNs to learn a mapping from digit sequences marked with morphosyntactic (case and gender) information, and their expression as Russian cardinal number names. The motivation for choosing Russian is that the number name system of this language, like that of many Slavic languages, is quite complicated, and therefore serves as a good test of the abilities of any text normalization system.

The architecture used was similar to a network employed by Rao et al. (2015) for grapheme-to-phoneme conversion, a superficially similar sequence-to-sequence mapping problem. We used a recurrent network with an input layer, four hidden feed-forward LSTM layers (Hochreiter and Schmidhuber, 1997), and a connectionist temporal classification (CTC) output layer with a softmax activation function (Graves et al., 2006).[1] Two of the hidden layers modeled forward sequences and the other two backward sequences. There were 32 input nodes—corresponding to characters—and 153 output nodes—corresponding to predicted number name words. Each of the hidden layers had 256 nodes. The full architecture is depicted in Figure 1.

The system was trained on 22M unique digit se-

[1] Experiments with a non-CTC softmax output layer yielded consistently poor results, and we do not report them here.

quences ranging from one to one million; these were collected by applying an existing TTS text normalization system to several terabytes of web text. Each training example consisted of a digit sequence, gender and case features, and the Russian cardinal number verbalization of that number. Thus, for example, the system has to learn to produce the feminine instrumental form of *60*. Examples of these mappings are shown in Table 1, and the various inflected forms of a single cardinal number are given in Table 2. In preliminary experiments, it was discovered that short digit sequences were poorly modeled due to undersampling, so an additional 240,000 short sequence samples (of three or fewer digits) were added to compensate. 2.2M examples (10%) were held out as a development set.

The system was trained for one day, after which it had a 0% label error rate (LER) on the development data set. When decoding 240,000 tokens of held-out test data with this model, we achieved very high accuracy (LER $<$ .0001). The few remaining errors, however, are a serious obstacle to using this system for TTS. The model appears to make no mistakes applying inflectional suffixes to unseen data. Plausibly, this task was made easier by our positioning of the morphological feature string at the end of the input, making it local to the output inflectional suffix (at least for the last word in the number expression). But it does make errors with respect to the numeric value of the expression. For example, for `9801__plu.ins.` (девятью тысячами восьмьюстами одними), the system produced девятью тысячами семьюстами одними (`9701__plu.ins.`): the morphology is correct, but the numeric value is wrong.[2]

This pattern of errors was exactly the opposite of what we want for speech applications. One might forgive a TTS system that reads *9801* with the correct numeric value but in the wrong case form: a listener would likely notice the error but would usually not be misled about the message being conveyed. In contrast, reading it as *nine thousand seven hundred and one* is completely unacceptable, as this would actively mislead the listener.

It is worth pointing out that the training set used here—22M examples—was quite large, and we were only able to obtain such a large amount of labeled data because we already had a high-quality hand-built grammar designed to do exactly this transduction. It is simply unreasonable to expect that one could obtain this amount of parallel data for a new language (e.g., from naturally-occurring examples, or from speech transcriptions). This problem is especially acute for low-resource languages (i.e., most of the world's languages), where data is by definition scarce, but where it is also hard to find high-quality linguistic resources or expertise, and where a machine learning approach is thus most needed.

In conclusion, the system does not perform as well as we demand, nor is it in any case a practical solution due to the large amount of training data needed. The RNN appears to have done an impressive job of learning the complex inflectional morphology of Russian, but it occasionally chooses the wrong number names altogether.

3 Number normalization with RNNs

For the purpose of more directly comparing the performance of RNNs with the methods we report on below, we chose to ignore the issue of allomorphy and morphological concord, which appears to be "easy" for generic sequence models like RNNs, and focus instead on verbalizing number expressions in whatever morphological category represents the language's citation form.

3.1 Data and general approach

For our experiments we used three parallel data sets where the target number name was in citation form (in Russian, nominative case):

- A **large** set consisting of 28,000 examples extracted from several terabytes of web text using an existing TTS text normalization system
- A **medium** set of 9,000 randomly-generated examples (for details, see Appendix A)
- A **minimal** set of 300 examples, consisting of the counting numbers up to 200, and 100 carefully-chosen examples engineered to cover a wide variety of phenomena

A separate set of 1,000 randomly-generated examples were held out for evaluation.

[2] The exact number of errors and their particular details varied from run to run.

5_neu.gen.	→	пяти	five
24_mas.acc.	→	двадцать четыре	twenty-four
99_plu.ins.	→	девяноста девятью	ninety-nine
11_fem.nom.	→	одиннадцать	eleven
81_fem.gen.	→	восьмидесяти одной	eighty-one
60_fem.ins.	→	шестьюдесятью	sixty
91_neu.ins.	→	девяноста одним	ninety-one
3_mas.gen.	→	трёх	three

Table 1: Example input and output data (and glosses) for the Russian RNN experiments.

шестьдесят	nominative (nom.)
шестидесяти	genitive (gen.)
шестидесяти	dative (dat.)
шестьдесят	accusative (acc.)
шестьюдесятью	instrumental (ins.)
шестидесяти	prepositional (pre.)

Table 2: Inflectional forms of the cardinal number number “60” in Russian.

The minimal set was intended to be representative of the sort of data one might obtain from a native-speaker when asked to provide all the essential information about number names in their language.[3]

In these experiments we used two different RNN models. The first was the same LSTM architecture as above (henceforth referred to as “LSTM”), except that the numbers of input and output nodes were 13 and 53, respectively, due to the smaller input and output vocabularies.

The second was a TensorFlow-based RNN with an attention mechanism (Mnih et al., 2014), using an overall architecture similar to that used in a system for end-to-end speech recognition (Chan et al., 2016). Specifically, we used a 4-layer pyramidal bidirectional LSTM reader that reads input characters, a layer of 256 attentional units, and a 2-layer decoder that produces word sequences. The reader is referred to Chan et al., 2016 for further details. Henceforth we refer to this model as “Attention”.

All models were trained for 24 hours, at which point they were determined to have converged.

[3]Note that the native speaker in question merely needs to be able to answer questions of the form “how do you say ‘23’ in your language?”; they do not need to be linguistically trained. In contrast, hand-built grammars require at least some linguistic sophistication on the part of the grammarian.

3.2 Results and discussion

Results for these experiments on a test corpus of 1,000 random examples are given in Table 3.

The RNN with attention clearly outperformed the LSTM in that it performed perfectly with both the medium and large training sets, whereas the LSTM made a small percentage of errors. Note that since the numbers were in citation form, there was little room for the LSTM to make inflectional errors, and the errors it made were all of the “silly” variety, in which the output simply denotes the wrong number. But neither system was capable of learning valid transductions given just 300 training examples.[4]

We draw two conclusions from these results. First, even a powerful machine learning model known to be applicable to a wide variety of problems may not be appropriate for *all* superficially-similar problems. Second, it remains to be seen whether any RNN could be designed to learn effectively from an amount of data as small as our smallest training set. Learning from minimal data sets is of great practical concern, and we will proceed to provide a plausible solution to this problem below. We note again that very low error rates do not ensure that a system is

[4]The failure of the RNNs to generalize from the minimal training set suggests they are evidently not expressive enough for the sort of “clever” inference that is needed to generalize from so little data. It is plausible that an alternative RNN architecture could learn with such a small amount of data, though we leave it to future research to discover just what such an architecture might be. In an attempt to provide the RNNs with additional support, we also performed an evaluation with the minimal training set in which inputs were encoded so that each decimal position above 0 was represented with a letter (`A` for *10*, `B` for 100, and so forth). Thus *2034* was represented as `2C3A4`. In principle, this ought to have prevented errors which fail to take positional information into account. Unfortunately, this made no difference whatsoever.

Training size	LSTM Acc.	Attention Acc.	Overlap
28,000	0.999	1.000	56%
9,000	0.994	1.000	0%
300	< 0.001	< 0.001	$< 1\%$

Table 3: Accuracies on a test corpus of 1,000 random Russian citation-form number-name examples for the two RNN architectures. "Overlap" indicates the percentage of the test examples that are also found in the training data.

usable, since not all errors are equally forgivable.

4 Number normalization with finite-state transducers

The problem of number normalization naturally decomposes into two subproblems: factorization and verbalization of the numeric factors. We first consider the latter problem, the simpler of the two.

Let λ be the set of *number names* in the target language, and let ν be the set of *numerals*, the integers denoted by a number name. Then let $L : \nu^* \to \lambda^*$ be a transducer which replaces a sequence of numerals with a sequence of number names. For instance, for English, L will map *90 7* to *ninety seven*. In languages where there are multiple allomorphs or case forms for a numeral, L will be non-functional (i.e., one-to-many); we return to this issue shortly. In nearly all cases, however, there are no more than a few dozen numerals in ν,[5] and no more than a few names in λ for the equivalent numeral in ν. Therefore, we assume it is possible to construct L with minimal effort and minimal knowledge of the language. Indeed, all the information needed to construct L for the experiments conducted in this paper can be found in English-language Wikipedia articles.

The remaining subproblem, factorization, is responsible for converting digit sequences to numeral factors. In English, for example, *97000* is factored as *90 7 1000*. Factorization is also language-specific. In Standard French, for example, there is no simplex number name for '90'; instead this is realized as *quatre-vingt-dix* "four twenty ten", and thus *97000* (*quatre-vingt-dix-sept mille*) is factored as *4 20 10 7 1000*. It is not a priori obvious how one might go about learning language-specific factorizations. For inspiration, we turn to a lesser-known body of linguistics research focusing on number grammars.

Hurford (1975) surveys cross-linguistic properties of number naming and proposes a syntactic representation which directly relates verbalized number names to the corresponding integers. Hurford interprets complex number constructions as arithmetic expressions in which operators (and the parentheses indicating associativity) have been elided. By far the two most common arithmetic operations are multiplication and addition.[6] In French, for example, the expression *dix-sept*, literally 'ten seven', denotes *17*, the sum of its terms, and *quatre-vingt(s)*, literally 'four twenty', refers to *80*, the product of its terms. These may be combined, in *quatre-vingt-dix-sept*. To visualize arithmetic operations and associativities, we henceforth write factorizations using s-expressions—pre-order serializations of *k*-ary trees—with numeral terminals and arithmetic operator non-terminals. For example, *quatre-vingt-dix-sept* is written `(+ (* 4 20) 10 7)`.

Within any language there are cues to this elided arithmetic structure. In some languages, some or all addends are separated by a word translated as *and*. In other languages it is possible to determine whether terms are to be multiplied or summed depending on their relative magnitudes. In French (as in English), for instance, an expression *XY* usually is interpreted as a product if $X < Y$, as in *quatre-vingt(s)* '80', but as a sum if $X > Y$, as in *vingt-quatre* '24'. Thus the problem of number *denormalization*—that is, recovering the integer denoted by a verbalized number—can be thought of as a special case of grammar induction from pairs of natural language expressions and

[5] At worst, a small number of languages, such as several Indic languages of North India, effectively use unique numerals for all counting numbers up to 100.

[6] Some languages make use of *half-counting*, or multiplication by one half (e.g., Welsh *hanner cant*, '50', literally 'half hundred'), or *back-counting*, i.e., subtraction (e.g., Latin *undevīgintī*, '19', literally 'one from twenty'; Menninger, 1969, 94f.). But these do not reduce the generality of the approach here.

their denotations (e.g., Kwiatkowski et al., 2011).

4.1 FST model

The complete model consists of four components:

1. A language-independent covering grammar F, transducing from integers expressed as digit sequences to the set of possible *factorizations* for that integer
2. A language-specific numeral map M, transducing from digit sequences to numerals
3. A language-specific verbalization grammar G, accepting only those factorizations which are licit in the target language
4. A language-specific lexical map L, transducing from sequences of numerals (e.g., *20*) to number names (already defined)

As the final component, the lexical map L, has already been described, we proceed to describe the remaining three components of the system.

4.1.1 Finite-state transducer algorithms

While we assume the reader has some familiarity with FSTs, we first provide a brief review of a few key algorithms we employ below.

Our FST model is constructed using *composition*, denoted by the $\circ$ operator. When both arguments to composition are transducers, composition is equivalent to chaining the two relations described. For example, if A transduces string x to string y, and B transduces y to z, then $A \circ B$ transduces from string x to string z. When the left-hand side of composition is a transducer and the right-hand side is an acceptor, then their composition produces a transducer in which the range of the left-hand side relation is intersected with the set of strings accepted by the right-hand side argument. Thus if A transduces string x to strings $\{y, z\}$, and B accepts y then $A \circ B$ transduces from x to y.

We make use of two other fundamental operations, namely *inversion* and *projection*. Every transducer A has an inverse denoted by A^{-1}, which is the transducer such that $A^{-1}(y) \to x$ if and only if $A(x) \to y$. Any transducer A also has input and output projections denoted by $\pi_i(A)$ and $\pi_o(A)$, respectively. If the transducer A has the domain α^* and the range β^*, then $\pi_i(A)$ is the acceptor over α^* which accepts x if and only if $A(x) \to y$ for some $y \in \beta^*$; output projection is defined similarly. The inverse, input projection, and output projection of an FST (or a push-down transducer) are computed by swapping and/or copying the input or output labels of each arc in the machine. See Mohri et al., 2002 for more details on these and other finite-state transducer algorithms.

4.1.2 Covering grammar

Let A be an FST which, when repeatedly applied to an arithmetic s-expression string, produces the s-expression's value. For example, one application of A to `(+ (* 4 20) 10 7)` produces `(+ 80 10 7)`, and a second application produces `97`. Let μ be the set of s-expression markup symbols {'(', ')', '+', '*'} and Δ be the set $\{0, 1, 2, \dots, 9\}$. Then,

$$F : \Delta^* \to (\mu \cup \Delta)^* = A^{-1} \circ A^{-1} \circ A^{-1} \dots \quad (1)$$

is an FST which transduces an integer expressed as a digit string to all its candidate factorizations expressed as s-expression strings.[7] Let $\mathcal{C}(d) = \pi_o(d \circ F)$, which maps from a digit sequence d to the set of all possible factorizations—in any language—of that digit sequence, encoded as s-expressions. For example, $\mathcal{C}($`97`$)$ contains strings such as:

```
(+ 90 7)
(+ 80 10 7)
(+ (* 4 20) 10 7)
...
```

4.1.3 Grammar inference

Let $M : (\mu \cup \Delta)^* \to \nu^*$ be a transducer which deletes all markup symbols in μ and replaces sequences of integers expressed as digit sequences with the appropriate numerals in ν. Let $\mathcal{D}(l) = \pi_i(M \circ L \circ l)$, which maps from a verbalization l to the set of all s-expressions which contain l as terminals. For example, $\mathcal{D}($`4 20 10 7`$)$ contains:

[7] In practice, our s-expressions never have a depth exceeding five, so we assume $F = A^{-1} \circ A^{-1} \circ A^{-1} \circ A^{-1} \circ A^{-1}$.

S	$\rightarrow$	(7 | 90 | * | +)
$*$	$\rightarrow$	(7 | 90 | +) 1000
$+$	$\rightarrow$	90 7

Table 4: A fragment of an English number grammar which accepts factorizations of the numbers {7, 90, 97, 7000, 90000, and 97000}. *S* represents the start symbol, and '|' denotes disjunction. Note that this fragment is regular rather than context-free, though this is rarely the case for complete grammars.

```
(+ 4 20 10 7)
(+ 4 20 (* 10 7))
(+ (* 4 20) 10 7)
...
```

Then, given (d, l) where $d \in \Delta^*$ is an integer expressed as a digit sequence, and $l \in \lambda^*$ is d's verbalization, their intersection

$$\mathcal{E}(d, l) = \mathcal{C}(d) \circ \mathcal{D}(l) \quad (2)$$

will contain the factorization(s) of d that verbalizes as l. In most cases, $\mathcal{E}$ will contain exactly one path for a given (d, l) pair. For instance, if d is `97000` and l is `ninety seven thousand`, $\mathcal{E}(d, l)$ is `(* (+ 90 7) 10000)`.

We can use $\mathcal{E}$ to induce a context-free grammar (CFG) which accepts only those number verbalizations present in the target language. The simplest possible such CFG uses '*' and '+' as non-terminal labels, and the elements in the domain of L (e.g., *20*) as terminals. The grammar will then consist of binary productions extracted from the s-expression derivations produced by $\mathcal{E}$. Table 4 provides a fragment of such a grammar.

With this approach, we face the familiar issues of ambiguity and sparsity. Concerning the former, the output of $\mathcal{E}$ is not unique for all outputs. We address this either by applying normal form constraints on the set of permissible productions, or ignoring ambiguous examples during induction. One case of ambiguity involves expressions involving addition with 0 or multiplication by 1, both identity operations that leave the identity element (i.e., 0 or 1) free to associate either to the left or to the right. From our perspective, this ambiguity is spurious, so we stipulate that identity elements may only be siblings to (i.e., on the right-hand side of a production with) another terminal. Thus an expression like *one thousand one hundred* can only be parsed as `(+ (* 1 1000) (* 1 100))`. But not all ambiguities can be handled by normal form constraints. Some expressions are ambiguous due to the presence of "palindromes" in the verbalization string. For instance, *two hundred two* can either be parsed as `(+ 2 (* 100 2))` or `(+ (* 2 100))`. The latter derivation is "correct" insofar as it follows the syntactic patterns of other English number expressions, but there is no way to determine this except with reference to the very language-specific patterns we are attempting to learn. Therefore we ignore such expressions during grammar induction, forcing the relevant rules to be induced from unambiguous expressions. Similarly, multiplication and addition are associative so expressions like *three hundred thousand* can be binarized either as `(* (* 3 100) 10000)` or `(* 3 (* 100 1000))`, though both derivations are equally "correct". Once again we ignore such ambiguous expressions, instead extracting the relevant rules from unambiguous expressions.

digit	$\rightarrow$	(2 | 3 | 4 | …9)
teen	$\rightarrow$	(11 | 12 | 13 | …19)
decade	$\rightarrow$	(20 | 30 | 40 |…90)
century	$\rightarrow$	(200 | 300 | 400 | …900)
power_of_ten	$\rightarrow$	(1000 | 10000 | …)

Table 5: Optional preterminal rules.

Since we only admit two non-terminal labels, the vast majority of our rules contain numeral terminals on their right-hand sides, and as a result, the number of rules tends to be roughly proportional to the size of the terminal vocabulary. Thus it is common that we have observed, for instance, *thirteen thousand* and *fourteen million* but not *fourteen thousand* or *thirteen million*, and as a result, the CFG may be deficient simply due to sparsity in the training data, particularly in languages with large terminal vocabularies. To enhance our ability to generalize from a small number of examples, we optionally insert preterminal labels during grammar induction to form classes of terminals assumed to pattern together in all productions. For instance, by introducing 'teen' and 'power_of_ten' preterminals, all four of the previous expressions are generated by the same top-level production. The full set of preterminal labels we use here are shown in Table 5.

In practice, obtaining productions using $\mathcal{E}$ is inefficient: it is roughly equivalent to a naïve algorithm which generates all possible derivations for the numerals given, then filters out all of those which do not evaluate to the expected total, violate the aforementioned normal form constraints, or are otherwise ambiguous. This fails to take advantage of top-down constraints derived from the particular structure of the problem. For example, the naïve algorithm entertains many candidate parses for *quatre-vingt-dix-sept* '97' where the root is '*' and the first child is '4', despite the fact that no such hypothesis is viable as 4 is not a divisor of 97.

We inject arithmetic constraints into the grammar induction procedure, as follows. The inputs to the modified algorithm are tuples of the form $(T, v_0, \ldots, v_n)$ where T is the numeric value of the expression and $v_0, \ldots, v_n$ are the $n + 1$ numerals in the verbalization. Consider a hypothesized numeric value of the leftmost child of the root, $T_{0\ldots i}$, which dominates $v_0, \ldots, v_i$ where $i < n$. For this to be viable, it must be the case that $T_{0\ldots i} \leq T$. And, if we further hypothesize that the root node is '+', then the remaining children must evaluate to $T - T_{0\ldots i}$. Similarly, if we hypothesize that the root node is '*', then the remaining children must evaluate to $T / T_{0\ldots i}$, and this quantity must be integral.

This approach can be implemented with a backtracking recursive descent parser enforcing the aforementioned normal form constraints and propagating the top-down arithmetic constraints. In practice, however, we implement the search using a straightforward dynamic programming algorithm. The algorithm proceeds by recursively generating all possible leftmost children of the tree and then using these top-down constraints to prune branches of the search space which have no viable completion (though our implementation does not fully propagate these constraints). While the number of left subtrees is exponential in the length of the verbalization, our implementation remains feasible since real-world examples tend to have verbalizations consisting of relatively few terminals. Pseudocode for our implementation is provided in Appendix B.

4.1.4 Grammar compilation

Once productions have been collected, they are used to specify a recursive transition network (Woods, 1970) which is then compiled into a pushdown acceptor (Allauzen and Riley, 2012) over ν^*, henceforth G. An example is shown in Figure 2.

4.1.5 Final model and remaining issues

Then, the verbalization for d is given by

$$\mathcal{V}(d) = \pi_o (d \circ F \circ M \circ G \circ L). \quad (3)$$

As noted above, the lexicon transducer L is non-functional when there are multiple number names for a single numeral, as may arise in number systems with allomorphy or morphological concord. When this is the case, we compose the lattice produced by $\mathcal{V}$ with a language model (LM) of verbalized numbers (over λ^*) and then decode using the shortest path algorithm. Note that whereas the construction of G requires parallel data, the LM requires only "spoken" data. While it is not common in most languages to write out complex cardinal numbers in their verbalized form, it is nonetheless possible to find a large sample of such expressions at web scale (Sproat, 2010); such expressions can be identified by matching against the unweighted $\pi_o (F \circ M \circ G \circ L)$.

4.2 Materials and methods

The FST-based verbalizer $\mathcal{V}$ was constructed and evaluated using four languages: English, Georgian, Khmer, and Russian (the latter targeting citation forms only). The **medium** and **minimal** sets are used for all four languages; in Russian, we also reuse the **large** data set (see §3.1). In all cases the test sets consisted of 1,000 randomly generated examples, the same examples as in previous experiments.

The size of $\mathcal{V}$ varied by language, with the smallest, English, consisting of roughly 8,000 states and arcs, and the largest, Russian, measuring roughly 80,000 states and arcs and comprising approximately a megabyte of uncompressed binary data.

No LM was required for either English or Khmer as both have a functional L. However, the Georgian and Russian L are both ambiguous, so the best path through the output lattice was selected according to a trigram language model with Witten-Bell smoothing. The language models were constructed using the **medium** training set.

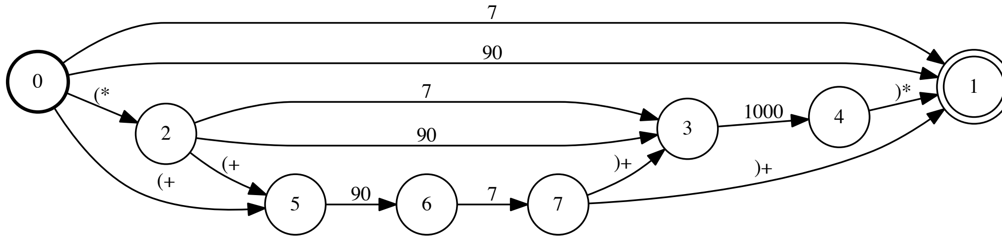

Figure 2: A pushdown acceptor that accepts all the language of the grammar fragment in Table 4. Arc labels that contain parentheses indicate “push” and “pop” stack operations, respectively, and must balance along a path.

Locale	Training size	Num. acc.	Morph. acc.	Overlap
eng_us	9,000	1.000	1.000	0%
	300	1.000	1.000	< 1%
kat_ge	9,000	1.000	1.000	0%
	300	1.000	1.000	< 1%
khm_kh	9,000	1.000	1.000	0%
	300	1.000	1.000	< 1%
rus_ru	28,000	1.000	1.000	56%
	9,000	1.000	0.998	0%
	300	1.000	0.998	< 1%

Table 6: Evaluation of the FST verbalizer on English (eng_us), Georgian (kat_ge), Khmer (kmh_kh), and Russian (rus_ru); errors are separated into those which affect the numeric denotation (“Num. acc.”) and those which merely use incorrect morphological forms (“Morph. acc.”).

4.3 Results and discussion

The results were excellent for all four languages. There were no errors at all in English, Georgian, and Khmer with either data set. While there were a few errors in Russian, crucially *all were agreement errors rather than errors in the factorization itself*, exactly the opposite pattern of error to the ones we observed with the LSTM model. For example, 70,477,170 was rendered as семьдесят миллион четыреста семьдесят семь тысяч сто семьдесят; the second word should be миллионов, the genitive plural form. More surprisingly, verbalizers trained on the 300 examples of the minimal data set performed just as well as ones trained with two orders of magnitude more labeled data.

5 Discussion

We presented two approaches to number normalization. The first used a general RNN architecture that has been used for other sequence mapping problems, and the second an FST-based system that uses a fair amount of domain knowledge. The RNN approach can achieve very high accuracy, but with two caveats: it requires a large amount of training data, and the errors it makes may result in the wrong number. The FST-based solution on the other hand can learn from a tiny dataset, and never makes that particularly pernicious type of error. The small size of training data needed and the high accuracy make this a particularly attractive approach for low-resource scenarios. In fact, we suspect that the FST model could be made to learn from a smaller number of examples than the 300 that make up the "minimal" set. Finding the minimum number of examples necessary to cover the entire number grammar appears to be a case of the set cover problem—-which is NP-complete (Karp, 1972)—but it is plausible that a greedy algorithm could identify an even smaller training set.

The grammar induction method used for the FST verbalizer is near to the simplest imaginable such procedure: it treats rules as well-formed if and only if they have at least one unambiguous occurrence in the training data. More sophisticated induction methods could be used to improve both generalization and robustness to errors in the training data. Generalization might be improved by methods that "hallucinate" unobserved productions (Mohri and Roark, 2006), and robustness could be improved using manual or automated tree annotation (e.g., Klein and Manning, 2003; Petrov and Klein, 2007). We leave this for future work.

Above, we focused solely on cardinal numbers, and specifically their citation forms. However, in all four languages studied here, ordinal numbers share the same factorization and differ only superficially from cardinals. In this case, the ordinal number verbalizer can be constructed by applying a trivial transduction to the cardinal number verbalizer. However, it is an open question whether this is a universal or whether there may be some languages in which the discrepancy is much greater, so that separate methods are necessary to construct the ordinal verbalizer.

The FST verbalizer does not provide any mechanism for verbalization of numbers in morphological contexts other than citation form. One possibility would be to use a discriminative model to select the most appropriate morphological variant of a number in context. We also leave this for future work.

One desirable property of the FST-based system is that FSTs (and PDTs) are trivially *invertible*: if one builds a transducer that maps from digit sequences to number names, one can invert it, resulting in a transducer that maps number names to digit sequences. (Invertibility is not a property of any RNN solution.) This allows one, with the help of the appropriate target-side language model, to convert a normalization system into a *denormalization* system, that maps from spoken to written form rather than from written to spoken. During ASR decoding, for example, it is often preferable to use spoken representations (e.g., *twenty-three*) rather than the written forms (e.g., *23*), and then perform denormalization on the resulting transcripts so they can be displayed to users in a more-readable form (Shugrina, 2010; Vasserman et al., 2015). In ongoing work we are evaluating FST verbalizers for use in ASR denormalization.

6 Conclusions

We have described two approaches to number normalization, a key component of speech recognition and synthesis systems. The first used a recurrent neural network and large amounts of training data, but very little knowledge about the problem space. The second used finite-state transducers and a learning

method totally specialized for this domain but which requires very little training data. While the former approach is certainly more appealing given current trends in NLP, only the latter is feasible for low-resource languages which most need an automated approach to text normalization.

To be sure, we have not demonstrated than RNNs—or similar models—are inapplicable to this problem, nor does it seem possible to do so. However, number normalization is arguably a sequence-to-sequence transduction problem, and RNNs have been shown to be viable end-to-end solutions for similar problems, including grapheme-to-phoneme conversion (Rao et al., 2015) and machine translation (Sutskever et al., 2014), so one might reasonably have expected them to have performed better without making the "silly" errors that we observed. Much of the recent rhetoric about deep learning suggests that neural networks obviate the need for incorporating detailed knowledge of the problem to be solved; instead, one merely needs to feed pairs consisting of inputs and the required outputs, and the system will self-organize to learn the desired mapping (Graves and Jaitly, 2014). While that is certainly a desirable ideal, for this problem one can achieve a much more compact and data-efficient solution if one is willing to exploit knowledge of the domain.

Acknowledgements

Thanks to Cyril Allauzen, Jason Eisner, Michael Riley, Brian Roark, and Ke Wu for helpful discussion, and to Navdeep Jaitly and Haşim Sak for assistance with RNN modeling. All finite-state models were constructed using the OpenGrm libraries (`http://opengrm.org`).

References

Cyril Allauzen and Michael Riley. 2012. A pushdown transducer extension for the OpenFst library. In *CIAA*, pages 66–77.

Richard Beaufort, Sophie Roekhaut, Louise-Amélie Cougnon, and Cédrick Fairon. 2010. A hybrid rule/model-based finite-state framework for normalizing SMS messages. In *ACL*, pages 770–779.

William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *ICASSP*, pages 4960–4964.

Jeffrey T. Chang, Hinrich Schütze, and Russ B. Altman. 2002. Creating an online dictionary of abbreviations from MEDLINE. *Journal of the American Medical Informatics Association*, 9(6):612–620.

H. Brandt Corstius, editor. 1968. *Grammars for number names*. D. Reidel, Dordrecht.

Peter Ebden and Richard Sproat. 2014. The Kestrel TTS text normalization system. *Natural Language Engineering*, 21(3):1–21.

Alex Graves and Navdeep Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. In *ICML*, pages 1764–1772.

Alex Graves, Santiago Fernández, Gaustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist temporal classification: Labeling unsegmented sequence data with recurrent neural networks. In *ICML*, pages 369–376.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

James R. Hurford. 1975. *The Linguistic Theory of Numerals*. Cambridge University Press, Cambridge.

Jakub Kanis, Jan Zelinka, and Luděk Müller. 2005. Automatic number normalization in inflectional languages. In *SPECOM*, pages 663–666.

Richard M. Karp. 1972. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *ACL*, pages 423–430.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *EMNLP*, pages 1512–1523.

Karl Menninger. 1969. *Number Words and Number Symbols*. MIT Press, Cambridge. Translation of *Zahlwort und Ziffer*, published by Vanderhoeck & Ruprecht, Breslau, 1934.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent models of visual attention. In *NIPS*, pages 2204–2212.

Mehryar Mohri and Brian Roark. 2006. Probabilistic context-free grammar induction based on structural zeros. In *NAACL*, pages 312–319.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Deana Pennell and Yang Liu. 2011. Toward text message normalization: Modeling abbreviation generation. In *ICASSP*, pages 5364–5367.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *NAACL*, pages 404–411.

Kanishka Rao, Fuchun Peng, Haşim Sak, and Françoise Beaufays. 2015. Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *ICASSP*, pages 4225–4229.

Brian Roark and Richard Sproat. 2014. Hippocratic abbreviation expansion. In *ACL*, pages 364–369.

Maria Shugrina. 2010. Formatting time-aligned ASR transcripts for readability. In *NAACL*, pages 198–206.

Richard Sproat, Alan W. Black, Stanley Chen, Shankar Kumar, Mari Ostendorf, and Christopher Richards. 2001. Normalization of non-standard words. *Computer Speech and Language*, 15(3):287–333.

Richard Sproat. 1996. Multilingual text analysis for text-to-speech synthesis. *Natural Language Engineering*, 2(4):369–380.

Richard Sproat. 2010. Lightly supervised learning of text normalization: Russian number names. In *IEEE Workshop on Speech and Language Technology*, pages 436–441.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112.

Paul Taylor. 2009. *Text to Speech Synthesis*. Cambridge University Press, Cambridge.

Lucy Vasserman, Vlad Schogol, and Keith Hall. 2015. Sequence-based class tagging for robust transcription in ASR. In *INTERSPEECH*, pages 473–477.

William A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Yi Yang and Jacob Eisenstein. 2013. A log-linear model for unsupervised text normalization. In *EMNLP*, pages 61–72.

A Random sampling procedure

Random-generated data sets were produced by sampling from a Yule-Simon distribution with $\rho = 1$, then rounding each sample's k trailing digits, where k is a random variable in the discrete uniform distribution $\mathcal{U}\{0, n\}$ and n is the order of the sampled number. Duplicate samples were then removed. The following R function implements this procedure.

```
require(VGAM)

EPSILON <- 1e-12
rnumbers <- function(n) {
  x <- ryules(n, rho=1)
  num.digits <- floor(log10(x + EPSILON)) + 1
  sig.digits <- ceiling(runif(n, min=0, max=num.digits))
  unique(signif(x, sig.digits))
}
```

B Parse generation algorithm

The following algorithm is used to generate parses from parallel (written/spoken) data. It depends upon a procedure GetSubtrees(…) generating all possible labeled binary subtrees given a sequence of terminals, which is left as an exercise to the reader.

1: **procedure** GetOracles(T, v_0, …, v_n) ▷ Total T, terminals v_0, …, v_n.
2: **if** $n = 1$ **then**
3: **if** eval(v_0) $= T$ **then**
4: **yield** s-expression v_0
5: **end if**
6: **return**
7: **end if**
8: **for** $i \in 1 \ldots n-1$ **do** ▷ Size of left child.
9: **for all** $L \in$ GetSubtrees(v_0, …, v_i) **do**
10: $T_L \leftarrow$ eval(L)
11: $T_R \leftarrow T - T_L$ ▷ Hypothesizes + root.
12: **if** $T_R > 0$ **then**
13: **for all** $R \in$ GetOracles(T_R, v_{i+1}, …, v_n) **do**
14: **yield** s-expression (+ L R)
15: **end for**
16: **end if**
17: $T_R \leftarrow T / T_L$ ▷ Hypothesizes ∗ root.
18: **if** $T_R \in \mathcal{N}$ **then** ▷ "is a natural number".
19: **for all** $R \in$ GetOracles(T_R, v_{i+1}, …, v_n) **do**
20: **yield** s-expression (* L R)
21: **end for**
22: **end if**
23: **end for**
24: **end for**
25: **end procedure**