Neural Models of Text Normalization for Speech Applications

Hao Zhang
Google Inc.
Research & Machine Intelligence
haozhang@google.com

Richard Sproat
Google Inc.
Research & Machine Intelligence
rws@google.com

Axel H. Ng
Google Inc.
Research & Machine Intelligence
axelhng@google.com

Felix Stahlberg
University of Cambridge
Department of Engineering
fs439@cam.ac.uk

Xiaochang Peng
Facebook, Inc.
NLP Research
xiaochang@fb.com

Kyle Gorman
Google Inc.
Research & Machine Intelligence
kbg@google.com

Brian Roark
Google Inc.
Research & Machine Intelligence
roark@google.com

Submission received: 27 June 2018; revised version received: 18 January 2019; accepted for publication: 21 February 2019.

doi:10.1162/COLI_a_00349

Machine learning, including neural network techniques, have been applied to virtually every domain in natural language processing. One problem that has been somewhat resistant to effective machine learning solutions is text normalization *for speech applications such as text-to-speech synthesis (TTS). In this application, one must decide, for example, that* 123 *is verbalized as* one hundred twenty three *in* 123 pages *but as* one twenty three *in* 123 King Ave. *For this task, state-of-the-art industrial systems depend heavily on hand-written language-specific grammars.*

We propose neural network models that treat text normalization for TTS as a sequence-to-sequence problem, in which the input is a text token in context, and the output is the verbalization of that token. We find that the most effective model, in accuracy and efficiency, is one where the sentential context is computed once and the results of that computation are combined with the computation of each token in sequence to compute the verbalization. This model allows for a great deal of flexibility in terms of representing the context, and also allows us to integrate tagging and segmentation into the process.

These models perform very well overall, but occasionally they will predict wildly inappropriate verbalizations, such as reading 3 cm *as* three kilometers. *Although rare, such verbalizations are a major issue for TTS applications. We thus use finite-state* covering grammars *to guide the neural models, either during training and decoding, or just during decoding, away from such "unrecoverable" errors. Such grammars can largely be learned from data.*

1. Introduction

A key question in the use of machine learning components within applications is what accuracy level is required in order for the learned models to be useful. The answer can vary dramatically depending on the application. For example, predicting the preferred internal heat for the passengers of a vehicle based on past preferences is likely useful as long as the prediction is within a degree or two of the actual preference, and worse performance on occasion is hardly catastrophic. The same cannot be said for the navigational systems of a self-driving vehicle, where even rare errors cannot be tolerated. Such thresholds for usefulness have been at play in speech and language processing since the earliest real-world applications. For example, Munteanu et al. (2006) attempted to establish a maximum word-error rate at which automatic speech recognition transcriptions are useful for tasks such as skimming online videos for content, concluding that at 25% word-error rate the transcripts were still useful for such a task. The word-error rate threshold for usefulness will be far lower for spoken assistant applications and presumably lower still for spoken interfaces to navigation systems. Similarly, machine translation may be found to be very useful when it comes to getting the gist of, say, a newspaper article, but direct use of its output is risky, illustrated by menu translation disasters. [1] The threshold of acceptable risk depends on the use of the system output, much like the threshold for acceptable latency.

Text normalization is a ubiquitous pre-processing stage for a range of speech and language processing applications, and its requirements depend quite heavily on the application for which it is designed—one reason, perhaps, that general machine learning methods to address the problem are not quite as common as in other areas of

1 For example, `http://languagelog.ldc.upenn.edu/nll/?p=4136`.

the field. Most text normalization methods will involve tokenization and matching of tokens against an existing lexicon. Beyond that, the downstream application may require mapping of certain tokens to new token sequences, including such trivial changes as de-casing to more complex mappings (e.g., expanding abbreviations or handling non-standard spellings such as those found in social media). In this article, we focus on text normalization for speech applications, such as sending text through a text-to-speech synthesis engine to be read aloud. This use scenario has some characteristics that make it different from text normalization scenarios that do not have a spoken target. The handling of numbers, for example, is quite different. When normalizing text for, say, syntactic parsing or other natural language algorithms operating solely within the written domain, one common approach is to replace the numerical value (e.g., *197*) with one or more placeholder class labels (e.g., *N*) so that all numerical values within a class are processed similarly by the parser. When normalizing text for speech synthesis, however, the written numeric value must be mapped to its spoken form (e.g., *one hundred ninety seven* or *one nine seven*), which is often called *number naming*. Thus, in the parsing scenario the normalization is deterministic; in the speech synthesis scenario it is contextually ambiguous. Number naming becomes especially tricky in inflected languages, such as Russian, where numbers take on the case of their predicates. Furthermore, writing long numbers out as they are spoken is not something people tend to do, so training data must be curated (rather than harvested, as in machine translation)—large, naturally occurring parallel corpora for this do not exist.

Text normalization for speech synthesis comes with some application demands that dictate acceptable latencies and error rates. As it is heavily used in mobile and spoken assistant applications, latency is a key consideration. Further, some errors are catastrophic, impacting not only the naturalness of the voice but the accuracy of the rendition. Inflecting number names incorrectly in, say, Russian does lead to a lack of naturalness, but the result would be usable in most cases. However, producing number names that are not value-preserving renders the result worse than unusable, since the user would be grossly misinformed. For example, when reading a written address (e.g., 197 Pine Ave.), if the numeric value is not preserved in the normalization (*nineteen hundred seven Pine Avenue*), then a driver using spoken directions in a map application may be led astray. Text normalization is thus a problem for which machine learning holds promise for contextual disambiguation, but under some relatively strict application demands.

In this article, we present several novel sequence-to-sequence architectures to address this problem, which outperform standard Transformer-based methods heavily used in neural machine translation (Vaswani et al. 2017). Further, some of the architectures exploit special characteristics of the problem to achieve significant further speedups without sacrificing accuracy. In addition, we explore methods for avoiding the kinds of catastrophic errors mentioned above (and presented in more detail subsequently) to which the neural methods are prone. We present methods for learning finite-state covering grammars, which help avoid solutions that are catastrophic (e.g., not value-preserving). Our approach represents a feasible combination of linguistic knowledge and data-driven methods that yield efficient architectures that meet demanding application performance requirements.

The specific contributions of this article include:

- The presentation of a large-scale, publicly available data set for this problem.

- Multiple new neural architectures that significantly improve upon the accuracy and efficiency of the models presented in Sproat and Jaitly (2016, 2017), as well as a set of other baselines.
- New general methods for finite-state covering grammar induction from data, used to avoid catastrophic errors, which extend the methods of Gorman and Sproat (2016) to other kinds of input that require normalization besides just numbers, such as dates or measure expressions.
- Extensive and informative evaluation of the models in a range of use scenarios.

In the next section we provide more background on the problem of text normalization before presenting our methods.

2. Text Normalization, and Why It Is Hard

Ever since the earliest invention of writing, in Mesopotamia over 5,000 years ago, people have used various sorts of abbreviatory devices. In most ancient writing systems, numbers were almost exclusively written with numerical symbols, rather than with number words representing the way one would *say* the number. Weights and measures often had standard short representations, and ordinary words could also be abbreviated to save space or time. That tradition has survived into modern writing systems. In English, whereas there are prescriptive conventions on, say, writing the full words for numbers when they begin a sentence (*Seventy-two people were found …* rather than *72 people were found*), there are nonetheless many things that would typically not be written out in words. These include numbers whose verbalization requires many words, as well as times, dates, monetary amounts, measure expressions, and so on. It is not so unusual to find *three million* written out, but *3,234,987* is much easier to read and write as a digit sequence. One might write *three thirty* or *half past three* for a time, but *3:30* is just as likely, and even preferable in many contexts. It would be much more likely to see *$39.99* written thus, rather than as *thirty-nine dollars and ninety-nine cents*, or *thirty-nine, ninety-nine*. For many of these cases there are differences across genres: Fully verbalized forms are much more likely in fictional prose, whereas numerical or abbreviated forms are much more likely in scientific texts, news, or on Wikipedia.

Following Taylor (2009), we use the term **semiotic class** to denote things like numbers, times, dates, monetary amounts, etc., that are often written in a way that differs from the way they are verbalized. **Text normalization** refers to the process of verbalizing semiotic class instances (e.g., converting something like *3 lb* into its verbalization *three pounds*). As part of a text-to-speech (TTS) system, the text normalization component is typically one of the first steps in the pipeline, converting raw text into a sequence of words, which can then be passed to later components of the system, including word pronunciation, prosody prediction, and ultimately waveform generation.

Work on text normalization for TTS dates to the earliest complete text-to-speech system, MITalk (Allen, Hunnicutt, and Klatt 1987). The earliest systems were based entirely on rules hard-coded in Fortran or C. The Bell Labs multilingual TTS system (Sproat 1996, 1997) introduced the use of weighted finite-state transducers for text normalization, and this approach is still in use in deployed systems, such as Google's Kestrel text-normalization system (Ebden and Sproat 2014). Sproat et al. (2001) describe an early attempt to apply machine learning to text normalization for TTS. The main challenge in text normalization is the variety of semiotic classes. Sproat et al. give an

Table 1
A taxonomy of *non-standard words*, from Sproat et al. (2001), Table 1, page 293. Used with permission.

	EXPN	abbreviation	*adv, N.Y, mph, gov't*
alpha	LSEQ	letter sequence	*CIA, D.C, CDs*
	ASWD	read as word	*CAT*, proper names
	MSPL	misspelling	*geogaphy*
	NUM	number (cardinal)	*12, 45, 1/2, 0·6*
	NORD	number (ordinal)	*May 7, 3rd, Bill Gates III*
	NTEL	telephone (or part of)	*212 555-4523*
	NDIG	number as digits	*Room 101*
N	NIDE	identifier	*747, 386, I5, pc110, 3A*
U	NADDR	number as street address	*5000 Pennsylvania, 4523 Forbes*
M	NZIP	zip code or PO Box	*91020*
B	NTIME	a (compound) time	*3·20, 11:45*
E	NDATE	a (compound) date	*2/2/99, 14/03/87* (or US) *03/14/87*
R	NYER	year(s)	*1998, 80s, 1900s, 2003*
S	MONEY	money (US or other)	*$3·45, HK$300, Y20,000, $200K*
	BMONEY	money tr/m/billions	*$3·45 billion*
	PRCT	percentage	*75%, 3·4%*
	SPLT	mixed or "split"	*WS99, x220, 2-car*
			(see also SLNT and PUNC examples)
	SLNT	not spoken,	word boundary or emphasis character:
M		word boundary	M.bath, KENT*RLTY, _really_
I	PUNC	not spoken,	non-standard punctuation: "***" in
S		phrase boundary	$99,9K***Whites, "..." in DECIDE... Year
C	FNSP	funny spelling	sllooooww, sh*t
	URL	url, pathname or email	*http://apj.co.uk, /usr/local, phj@tpt.com*
	NONE	should be ignored	ascii art, formatting junk

initial taxonomy (Table 1) with three major categories: "mostly alphabetic," "numeric," and "miscellaneous." Within these broad categories finer-grained classifications depend in part on how the input maps to the output verbalization, and in part on the kind of entity denoted by the token.

From a modern perspective, this taxonomy has a number of obvious omissions. Some of these omitted types did not exist, or were considerably less common, at the time of writing, such as hashtags or "funny spellings" like *sllooooww*. The increasing prominence of such categories has led to a considerable body of work on normalizing SMS and social media text (Xia, Wong, and Li 2006; Choudhury et al. 2007; Kobus, Yvon, and Damnati 2008; Beaufort et al. 2010; Liu et al. 2011; Pennell and Liu 2011; Aw and Lee 2012; Liu, Weng, and Jiang 2012; Liu et al. 2012; Hassan and Menezes 2013; Yang and Eisenstein 2013; Chrupala 2014; Min and Mott 2015, inter alia).[2] Text normalization is thus a task of great importance for many diverse real-world applications, although the requirements of large-scale speech applications such as TTS and automatic speech recognition (ASR) have received comparatively little attention in the natural language

2 Text normalization of social media tends to focus on different problems from those that are the main concern of normalization aimed at speech applications. For example, how one pronounces number sequences is generally of little or no concern in the normalization of social media text, though it is essential for most speech applications.

processing literature. A recent update to this taxonomy is presented by van Esch and Sproat (2017). Among the new categories are season/episode designations (*S01/E02*), ratings (*4.5/5*), vision (*20/20*), and chess notation (*Nc6*). What is noteworthy about these novel categories is that each of them has idiosyncratic ways of verbalization. Thus a rating *4.5/5* is read *four point five out of five*, whereas the vision specification *20/20* is read *twenty twenty* and the season/episode designation *S01/E02* is read *season one, episode two*, and so on.

To a first approximation, one can handle all of these cases using hand-written grammars—as in Sproat (1997) and Ebden and Sproat (2014). However, different aspects of the problem are better treated using different approaches. Text normalization can be basically broken down into three notional components. The first is *tokenization*: How do we segment the text into tokens each of which is a word, a punctuation token, or an instance of a semiotic class? Second, what are the reasonable ways to *verbalize* that token? And third, which reading is most appropriate for the given context?

The first component, tokenization, can be treated with hand-written grammars, but this may be brittle and challenging to maintain. Thus it seems desirable to use machine-learned taggers, similar to those used to segment text in writing systems that do not separate words with spaces; see, for example, Wang and Xu (2017), who use convolutional neural networks for Chinese word segmentation.

The second component, verbalization, can be treated with hand-written language-specific grammars. But as is well known, such grammars may require some degree of linguistic expertise, can be complex to develop, and with sufficient complexity become difficult to maintain. And as noted above, there is great diversity between different semiotic classes. [3]

Finally, selecting the appropriate verbalization in context can also be framed as a sequence labeling problem, since the verbalization of a token depends on what it represents. For instance, *4/5* can be a date, a fraction, or a rating, and once we know which it is, the verbalization is fairly straightforward. In languages with complex inflectional morphology, one may also need to know which morphosyntactic category a string falls into—in Russian, for example, it is not enough to know that *323* is to be read as a cardinal number, because one also needs to know what grammatical case to use—but again, this is a sequence-labeling problem. [4]

Although this three-component system is feasible, it is desirable—as we do in some experiments herein—to treat earlier stages (such as tokenization or semiotic class labeling) as a latent variable passed to later stages; thus it would be preferable to train all three models jointly. Furthermore, because these three components involve both labeling with a fixed tag set (for segmentation and semiotic classification) and string-to-string transductions (for verbalization), a neural network approach, which is suited to both types of relations, is a priori desirable.

We note two points at the outset. First, the required data, namely, raw text and its verbalized equivalent, cannot in general be expected to occur naturally. Machine translation can to a large extent rely on "found" data because people translate texts for practical reasons, such as providing access to documents to people not able to read the source language. In contrast, there is no reason why people would spend resources producing verbalized equivalents of ordinary written text: in most cases, English speakers

3 As a consequence, much of the subsequent work on applying machine learning to text normalization for speech applications focuses on specific semiotic classes, like letter sequences (Sproat and Hall 2014), abbreviations (Roark and Sproat 2014), or cardinal numbers (Gorman and Sproat 2016).

4 In fact, Kestrel (Ebden and Sproat 2014) uses a machine-learned morphosyntactic tagger for Russian.

do not need a gloss to know how to read *$10 million*. Thus, if one wants to train neural models to verbalize written text, one must produce the data.[5] Second, the bar for success in this domain seems to be higher than it is in other domains in that users expect TTS systems to correctly read numbers, dates, times, currency amounts, and so on. As we show subsequently, deep learning models produce good results overall; but as we shall also show, neural networks tend to make the occasional error that would be particularly problematic for a real application.

We distinguish between two types of error that a text normalization system might make. The first, and less serious, kind involves picking the wrong form of a word, while otherwise preserving the meaning. For example, if the system reads *the road is 45 km long* as *the road is forty five* ***kilometer*** *long*, this is wrong, but humans can easily recover from the error and still understand the message. Such errors are particularly likely to occur in languages with complex inflectional morphology. Errors of this first kind, however, do not generally reduce the intelligibility of the resulting speech insofar as the same message is conveyed, albeit ungrammatically. Contrast this with the second kind of error, where the error results in a totally different message being conveyed. If the system reads the sentence as *the road is* ***thirty five*** *kilometers long*, getting the number wrong completely, it would convey the wrong meaning entirely. Unfortunately, we find that neural network models are particularly prone to the latter type of error; similar issues in neural machine translation are discussed by Arthur, Neubig, and Nakamura (2016). We refer to the latter class as **unrecoverable errors** because they miscommunicate information in a way that is hard for the hearer to recover from. Even if unrecoverable errors occur only infrequently, one never knows when to expect such an error, and, because such errors are sporadic, it is hard to guard against them if the neural model is left to its own devices.

It is worth emphasizing up front that unrecoverable errors are one reason why approaches like Char2Wav (Sotelo et al. 2017) that purport to provide a complete "end-to-end" neural network that maps directly from raw text to waveforms are unlikely to replace industrial TTS systems in the near term. Text normalization by itself is a hard problem. If one is expecting to solve not only that, but also other aspects of speech such as word pronunciation and prosody by inferring a model from a limited amount of aligned text and speech data, then one is setting one's expectations higher than what currently seems reasonable to expect of neural models. Although research on "end-to-end" neural TTS has produced impressive *demonstrations*, our own results suggest that they will make embarrassing errors when applied to arbitrary text, and such errors would be hard to fix in an end-to-end system. The **covering grammar** solution to these errors that we present here depends on the fact that we are producing *symbolic* (string)

5 It has been suggested that raw text paired with speech—such as produced by ASR systems—does exist in large volume. However, several problems prevent use of such data for text normalization at the present time. First, many genres of text, including literary text, and even news stories, are not particularly rich in text normalization problems. We have used Wikipedia text in many of our experiments because it is rich in at least some semiotic classes, such as dates, measure expressions, and various other numerical expressions. Unfortunately, while there is a spoken subset of Wikipedia, volume is very low and quality is variable. Furthermore, many of the chosen articles are, again, not very rich in text normalization issues, and there is often a mismatch between the version of the article that was read, and what is currently available on (written) Wikipedia. Closed captioning is more promising, but it is notorious for not-infrequent departures from the spoken text, and much closed captioned text is, again, not particularly rich in text normalization issues. Finally, we note a recent trend toward "end-to-end" ASR systems (Chan et al. 2016; Chiu et al. 2017, inter alia) that map from audio directly to written form, without an intermediate "spoken-form" transcription. For a text written as *In 1983, ...* the recognizer would transcribe *In 1983*, rather than *in nineteen eighty three*, making it useless for our purposes.

output: It is hard to see how one would apply such methods to correct continuous output, such as a waveform or a sequence of speech parameters. There would therefore be no hope for solving such errors in a fully end-to-end system, other than augmenting the training data with targeted examples and hoping for the best.

Sequence-to-sequence text normalization has the promise of providing accurate normalization with far less development and maintenance effort than hand-written grammars. For deployment as part of an actual TTS system, however, such an approach must also be sufficiently fast at time of inference, yield very high accuracy, and be reliable and predictable—that is, without producing the kinds of unrecoverable errors illustrated earlier. In this article, we explore a range of neural architectures, and demonstrate an accuracy similar to that of heavily engineered hand-written methods. We achieve substantial speedups with certain methods for encoding sentence context, as well as methods for integrating constraints that forbid the more egregious kinds of unrecoverable errors. In sum, this article presents methods for text normalization that meet most of the engineering requirements for use in large-scale text-to-speech systems.

3. Previous Approaches to Text Normalization

3.1 Standard Approaches

The standard approach in industrial text-normalization systems uses complex hand-written grammars to verbalize input tokens. An example of such a system is Google's Kestrel TTS text normalization system (Ebden and Sproat 2014). Kestrel is powered by classification grammars, which tokenize the input and classify text tokens according to their semiotic class; and verbalization grammars, which determine the contextually appropriate verbalization for a token, in conjunction with machine learned morphosyntactic taggers.

Tokens in Kestrel need not correspond to whitespace-delimited tokens, even in those languages where whitespace or punctuation is used to separate words. Thus the Kestrel grammars recognize *Jan.1, 2012* as a date and parse it as a single token, identifying the month, day, and year, and represent it internally using a protocol-buffer representation like the following:[6]

```
date { month: "January" day: "1" year: "2012"}
```

Verbalization grammars then convert from a serialization of the protocol buffer representation into actual word sequences, such as *January the first twenty twelve*. Tokenization/classification and verbalization grammars are compiled into weighted finite-state transducers (WFSTs) using the Thrax grammar development library (Roark et al. 2012).

One advantage of separating tokenization/classification from verbalization via the intermediate protocol buffer representation is that it allows for reordering of elements, something that is challenging with WFSTs.[7] The need for reordering arises, for example, in the treatment of currency expressions where currency symbols such as '$' or '' often occur before digits, but are verbalized after the corresponding digits. An input *$30* might be parsed as something like

```
money { currency: "USD" amount { integer: "30" } }
```

6 `https://developers.google.com/protocol-buffers/`.

7 As we will show subsequently, pushdown transducers can be used for this purpose, and are finite-state equivalent if the vocabulary of elements to be reordered is itself finite.

which would then be reordered prior to verbalization as

```
money { amount { integer: "30" } currency: "USD" }
```

An open-source version of Kestrel includes sample grammars for a subset of English text normalization problems.[8]

Similar approaches have been adopted for ASR verbalization (Sak et al. 2013, inter alia). In this case, the problem is not generally to verbalize text at runtime, but rather as a component of an acoustic- and language-model training procedure that ingests written text as training data and converts it to possible spoken forms. Unlike the situation in TTS where one usually wants just one output verbalization for a given context, in ASR it may be desirable to produce a set of possible verbalizations; for example, *1999* might be verbalized as *nineteen ninety nine* if it is a date, or as *one thousand nine hundred ninety nine* if it is an ordinary number.

3.2 Previous Neural Approaches to Text Normalization

There has been some work on neural network methods for social media text normalization (Chrupala 2014; Min and Mott 2015) that demonstrate competitive performance in shared tasks. However, as we noted earlier, these systems have a somewhat different focus than speech applications.

Sproat and Jaitly (2016) explore two neural models of text normalization. The first involves a long short-term memory (LSTM) (Hochreiter and Schmidhuber 1997), recurrent neural network (RNN) that produces a lattice-like list of possible verbalizations which are then rescored by a separate LSTM language model. The second is an RNN with an attention mechanism modeled on Chan et al. (2016). A subsequent study (Sproat and Jaitly 2017) focuses solely on the latter model, which outperformed the former; despite this, it still produces a substantial number of unrecoverable errors as described above. This paper also describes a method for preventing unrecoverable errors using WFST-based *covering grammars*, a method described in much greater detail in Section 6.

Sproat and Jaitly (2016) and Sproat and Jaitly (2017) have been starting points for our work, and our "sliding window" baseline (Section 5.2) is based on this prior work, but the present work goes well beyond it in a number of respects. First, we develop further neural architectures that outperform this prior system. Second, though the *covering grammar* mechanism was introduced in the prior work, we extend it here both in that we introduce a method for inducing most of the covering grammar models from data, and in that we define more precisely how the covering grammars are actually used during decoding. Finally, we report results on new data sets.

Arik et al. (2017) present a neural network TTS system that mimics the traditional separation into linguistic analysis (or front-end) and synthesis (or back-end) modules. It is unclear to what degree this system in fact performs text normalization since the only front-end component they describe is grapheme-to-phoneme conversion, which is a separate process from text normalization and usually performed later in the pipeline.

Some prior work, such as Shugrina (2010), focuses on the inverse problem of *denormalizing* spoken sequences into written text in the context of ASR so that *two hundred fifty* would get converted to *250*, or *three thirty* as a time would get formatted as *3:30*. Pusateri et al. (2017) describe a system in which denormalization is treated as a neural network sequence labeling problem using a rich tag set.

8 See `http://github.com/google/sparrowhawk`.

The data we report on in this article was recently released and was the subject of a Kaggle competition (see later in this article), and a few recent studies (Pramanik and Hussain 2018; Yolchuyeva, Németh, and Gyire-Tóth 2018, inter alia) propose neural network approaches to text normalization using this data set. We postpone discussion of this work until a later section (Section 7.7).

4. A Transformer Model

Before we turn to a discussion of our own neural text normalization models, we present some results using a different approach, one that probably has already occurred to the reader. Namely: Why not treat the text normalization problem as a *machine translation* task, where the source language is raw text and the target language is normalized text, in the same language as the source language? Thus in our case a source sentence might be

```
John lives at 123 King Ave next to A&P.
```

and the corresponding target would be

```
John lives at one twenty three King Avenue next to A_letter and P_letter sil
```

Clearly, this is a much easier problem than real translation, especially when it comes to translation between two very dissimilar languages such as English and Japanese.

To this end, we trained a Transformer model (Vaswani et al. 2017) on our English training set, as described in Section 7.1, and tested on our standard English test set. Because the problem is a full sequence-to-sequence task, the training corpus was transformed into a sequence of pairs of raw input sentences, and normalized output sentences, as in the given example.

The details of our Transformer model are summarized in Appendix A.2.

Because there was no simple way to align the input tokens with the output verbalization, we evaluated for *sentence accuracy* only. Note that the amount of training data—about 10 million tokens—is perhaps an order of magnitude less data than is commonly used for translation between high-resource languages. On the other hand, our task is significantly easier than real translation. It is not clear how these two factors trade off against one another.

The overall sentence accuracy of the Transformer system on our data was 96.53% (sentence error rate of 3.47%), which is lower than the accuracy of 97.75% we report here for our best system (see Table 5). As with our own purely neural systems, the Transformer model is prone to unrecoverable errors. Among these are:

- A predilection for replacing the letter *i* with *u* so that the letter sequence *IUCN*, for instance, is verbalized as *u u c n*.
- Inappropriate expansions such as verbalizing *they've been outside* as *they avenue been outside*, or *Cherokee's 2.8 V 6* as *cherokee's two point eight volts six*.
- Complete substitution of different lexical items: *Yahoo!* verbalized as *o m l*.

A similar experiment with our Russian data produced similar results. Here, the sentence accuracy of the system is 93.35% (error rate of 6.65%), compared with our own best sentence accuracy of 95.46% from Table 5. Common types of errors in Russian are:

- *километр в квадрате* 'kilometer squared' (and morphological variants) instead of *квадратный километр* 'square kilometer' (and morphological variants). This is due to mistokenized examples of *км*^{2} in the training data being translated as *к м в квадрате*, with the Transformer model generalizing that pattern. Our own models do not make this error.
- Wrong choice of letters in letter sequences, as in English.
- Stopping problems in transliterations. For example, the English word *narrative* transliterated not as *нарратив* (literally *narrativ*), but as *нарративтитивтивтивтивтивтив*. Such stopping problems are familiar in sequence-to-sequence models (see, e.g., Section 5.2.3 of Xie [2017] for a discussion of this problem in neural natural language generation).

As already noted—and see, especially, Sections 7.6 and 7.8—our own neural models make similar kinds of errors, but the fact that the Transformer model is prone to such errors, coupled with slightly higher overall error rates, at least serves as an answer to the question that people often ask, namely, why we do not treat this as a machine translation task? Simply put, state-of-the-art machine translation models do not solve the problem because we would still need to have some mechanism to correct for these errors.

This then leads us to the more serious issue, namely, that treating the problem as a sequence-to-sequence problem where the input and output are full sentences makes it much more difficult to correct such errors, since it would be harder to reconstruct which output token(s) correspond to which input token(s), and thus which portion of the input is responsible for the errorful output. This in turn motivates architectures that treat each input token separately. With such an architecture, one has a chance of correcting the output by an approach such as the covering grammar approach we will discuss in Section 6, or in the worst case by simply adding the input token to a whitelist that forces it to be verbalized in a particular way. We view these issues as compelling motivations for not adopting an "out-of-the-box" solution, but rather designing models that keep the relation between input and output tokens clear. We turn immediately to a discussion of our models.

5. Our Models

In the previous section we argued against using an architecture designed for machine translation for text normalization. In this section we present novel models and their components that are specifically designed for this problem. We start with a discussion of segmentation.

5.1 Segmentation

Most of our models (except the one presented in Section 5.4) assume pre-segmented input. Table 2 gives an example of the data used for these models. We assume the same segmentation standard as Ebden and Sproat (2014), which generally splits off

Table 2
An example training sentence.

John	`<self>`
lives	`<self>`
at	`<self>`
123	`one twenty three`
King	`<self>`
Ave	`avenue`
next	`<self>`
to	`<self>`
A&P	`a_letter and p_letter`
.	`sil`

punctuation and separates words by whitespace, but also treats as a single segment multiword sequences that represent dates (*Jan. 3, 2016*), certain money expressions (*$5 million*), and so on. We use a special token `<self>` to indicate that the input is to be passed through. The token `sil` is used to represent silence, which is typically associated with punctuation. We denote the set of all possible Kestrel (input) segments as S, and the set of all possible output words (the target vocabulary) as W. We represent each sentence in the training corpus as a sequence of pairs $\langle (x_1, y_1), \ldots, (x_l, y_l) \rangle$ where l is the sentence length in segments. Each $x_i \in S$ is a single segment such as a complete date, address, money expression, and so forth (e.g., $x_i = \texttt{123}$). $y_i \in \{\texttt{<self>}, \texttt{sil}\} \cup W^+$ contains the normalized form of x_i as a sequence of words (e.g., $y_i = \langle \texttt{one}, \texttt{twenty}, \texttt{three} \rangle$).

The output word vocabulary W in our setup is relatively small because it only contains words used for non-trivial normalization such as number names, and so on. On the input side, however, we need to tokenize segments into smaller chunks to obtain a fixed-size input vocabulary for the neural models. We denote the input vocabulary to the neural model as T, and the mapping from segment to token sequence as $\text{tok} : S \rightarrow T^+$. T can be a short list of full words, word features (Section 5.3.1), subword units (Sennrich, Haddow, and Birch 2016), or characters.

We will use the $\oplus$-operator to denote string concatenation.

5.2 The Sliding Window Model

In the next few sections we detail the various architectures that we have applied to the problem of text normalization, starting with our baseline model.

We use the model of Sproat and Jaitly (2016, 2017) as a strong neural baseline, which consists of a bidirectional RNN encoder and an attention mechanism decoder. Appendix A.3 lists the hyper-parameters we use in our experiments.

We refer to this model as **sliding window model** (Figure 1) as we normalize each segment by feeding in a context window of n segments to the left and right around the segment of interest (where n is typically 3). The token of current interest is enclosed in the tags `<norm>...</norm>` as in this example:

```
John lives at <norm> 123 </norm> King Ave next to A&P .
```

The output being predicted is just the verbalization of the current segment:

```
one twenty three
```

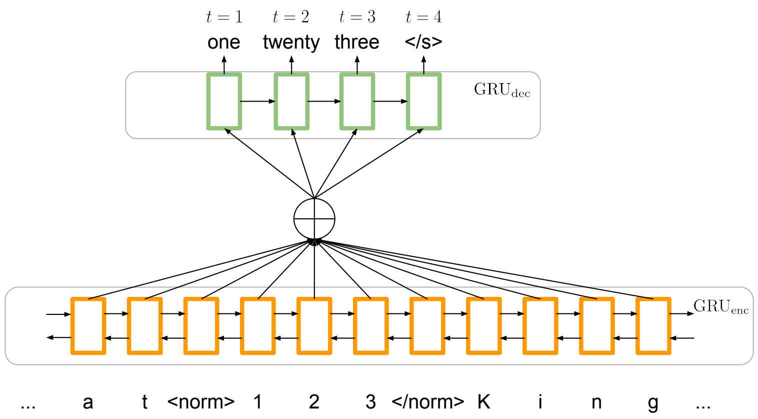

Figure 1
The **sliding window model**, with a bidirectional RNN encoder and an attention mechanism decoder.

More formally, the model normalizes the i-th segment by learning to predict the following probability:

$$P(y_i|x_{i-n},\ldots,x_{i+n}) = \prod_{t=1}^{|y_i|} \underbrace{P((y_i)_t|(y_i)_1^{t-1},x_{i-n},\ldots,x_{i+n})}_{=g(\cdot)} \tag{1}$$

where $g(\cdot)$ is modeled by an RNN-based attentional encoder-decoder network (Bahdanau, Cho, and Bengio 2015) in which the encoder consumes the context window around the to-be-normalized segment:

$$\bigoplus_{k=\max(1,i-n)}^{i-1} \mathrm{tok}(x_k) \oplus < \mathtt{norm} > \oplus\, \mathrm{tok}(x_i) \oplus < /\mathtt{norm} > \oplus \bigoplus_{k=i+1}^{\min(l,i+n)} \mathrm{tok}(x_k) \tag{2}$$

If the length of the resulting sequence is l, the encoder built on gated recurrent units (GRU$_{\text{enc}}$ in Figure 1) generates l hidden states $((h_i)_1,\ldots,(h_i)_l)$. The attention mechanism works as follows:

$$\begin{aligned} P((y_i)_t|(y_i)_1^{t-1},x_{i-n},\ldots,x_{i+n}) &= P((y_i)_t|(y_i)_1^{t-1},(h_i)_1^l) \\ &= g((y_i)_{t-1},s_{i,t},c_{i,t}) \end{aligned}$$

where g is a nonlinear function that outputs the probability of $(y_i)_t$, given state $s_{i,t}$ of the decoder RNN (GRU$_{\text{dec}}$ in Figure 1) at time step t after producing $(y_i)_1^{t-1}$ and attending over $(h_i)_1^l$ to result in the context vector $c_{i,t}$. Concretely, $c_{i,t}$ is a weighted sum of the encoder hidden states.

$$c_{i,t} = \sum_{j=1}^{l} (\alpha_i)_{tj} (h_i)_j \tag{3}$$

The weight of each hidden state is computed as

$$(\alpha_i)_{tj} = \frac{\exp((e_i)_{tj})}{\sum_{k=1}^{l} \exp((e_i)_{tk})}$$

where

$$(e_i)_{tj} = a(s_{i,t-1}, (h_i)_j)$$

and where a is the alignment model that computes the matching score between input position j and output position t for the i-th example, which is implemented as a single-layer multilayer perceptron with tanh(·) as the activation function.

Note that even though each segment is considered separately from the other segments, the use of an input context of three words is generally sufficient to disambiguate in cases where a given segment might have more than one possible verbalization. Thus, if the input were:

```
I raised <norm> 123 </norm> goats .
```

the context around *123* in this instance would be sufficient to determine that the correct reading is:

```
one hundred twenty three
```

One advantage of processing each segment separately is that it allows for parallel processing during decoding, which significantly speeds up decoding.

The input sequence for the sliding window model is a character sequence (i.e., tok(·) maps to characters, and T is the English character set), because for a segment like *123*, one needs to see the individual digits to know how to read it. It is also useful to see the individual characters for out-of-vocabulary words to determine whether they should be mapped to `<self>`.[9]

5.3 Contextual Sequence-to-Sequence Models

The previous section described our baseline sequence-to-sequence model where data are presented to the model one segment at a time using a sliding window, with each segment enclosed in `<norm>...</norm>` tags. In this and the following sections, we propose alternative neural network architectures for this problem.

The models proposed in this section can be characterized as *contextual sequence-to-sequence models* illustrated schematically in Figure 2. Here the problem is cast as a

9 We have also experimented with *word piece* and other encodings for surrounding context, as discussed later in the paper.

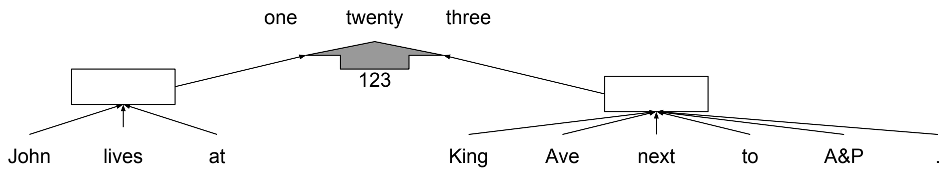

Figure 2
Text normalization with a contextual sequence-to-sequence model.

context-aware sequence-to-sequence mapping task from the input (character) sequence `1 2 3` to the output (word) sequence `one twenty three`. The context of this sequence-to-sequence problem is encoded with two vector representations: one for the left context `John lives at`, and one for the right context `King Ave next to A&P`.

The core of this architecture is an RNN-based attentional sequence-to-sequence network with a bidirectional encoder $\mathrm{GRU}_{\mathrm{mid}}$ built from gated recurrent units, or GRUs (Cho et al. 2014). Appendix A.4 lists the hyper-parameters we use in our experiments.

5.3.1 Encoding the Context. We propose to use two additional RNNs to produce distributed representations of the full sentence context. Figure 3 shows the complete model architecture. In addition to the core sequence-to-sequence encoder $\mathrm{GRU}_{\mathrm{mid}}$ (shown with orange boxes in the figure) we use two more GRU networks ($\mathrm{GRU}_{\rightarrow}$ and $\mathrm{GRU}_{\leftarrow}$) to encode the context. The hidden states of $\mathrm{GRU}_{\rightarrow}$ and $\mathrm{GRU}_{\leftarrow}$ that are adjacent to the to-be-normalized segment are used as context representations. Note that the tokenization for the context is independent of the middle segment, which makes it possible to use coarse-grained tokenization such as words or word pieces for the context and characters for the to-be-normalized part. In our later experiments, we always use characters for the middle segment (i.e., $\mathrm{tok}_{\mathrm{mid}}(\cdot)$ maps to the character set T_{mid}), but vary the granularity for the context tokenization $\mathrm{tok}_{\mathrm{context}} : S \rightarrow T_{\mathrm{context}}$. The choices for T_{context} include (from coarse to fine) words, word pieces (Schuster and Nakajima 2012; Sennrich, Haddow, and Birch 2016; Wu et al. 2016), or even characters. We also experiment with word features, which represent words by extracting character n-gram features for n up to 3 within each word, hashing and embedding them, and finally applying a transformation to produce a fixed-length dense vector. The simplest transformation is a summation, but it can also be a neural network such as an RNN. Figure 4 shows the character n-gram based word contextual model.

Unlike the sliding window model (Equation 2), our new contextual sequence-to-sequence model conditions on the full sentence to normalize the i-th segment:

$$P(y_i|x_1,\ldots,x_l) = \prod_{t=1}^{|y_i|} P((y_i)_t \mid (y_i)_1^{t-1}, c_{i,t}) \qquad (4)$$

where $c_{i,t}$ is the concatenation of three vectors (see Figure 3):

$$c_{i,t} = (\overrightarrow{h_{i-1}}; h_{i,t}^{\mathrm{mid}}; \overleftarrow{h_{i+1}}) \qquad (5)$$

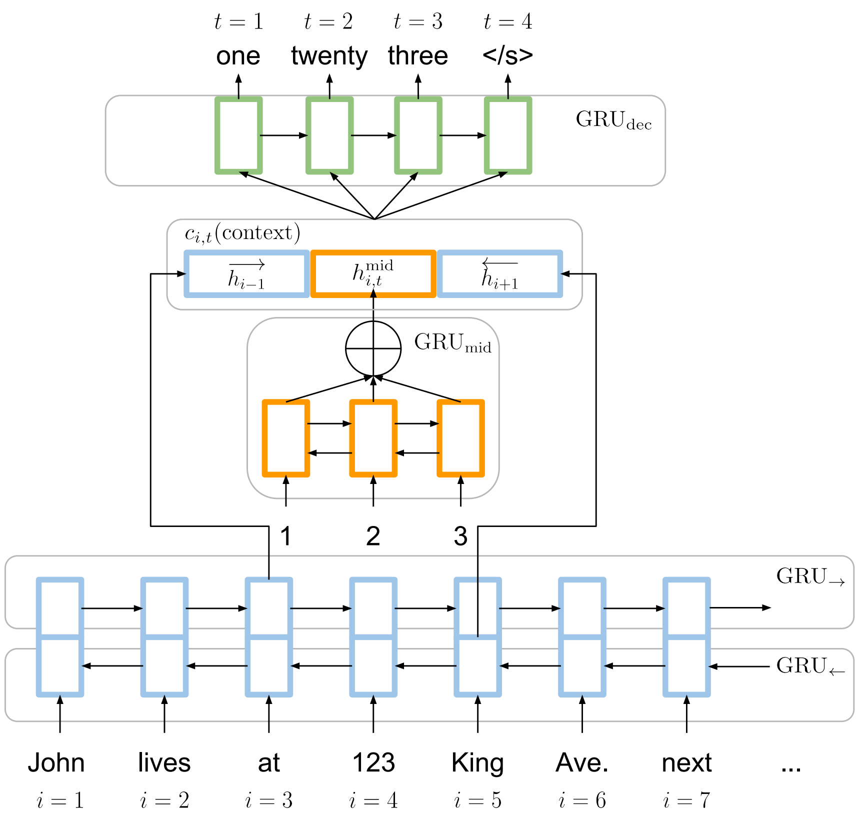

Figure 3
Encoding the context with RNNs $\text{GRU}_{\rightarrow}$ and $\text{GRU}_{\leftarrow}$.

Here, $\overrightarrow{h_k}$ and $\overleftarrow{h_k}$ for $1 \leq k \leq l$ are the hidden states of $\text{GRU}_{\rightarrow}$ and $\text{GRU}_{\leftarrow}$ that run over the complete input sequence $\bigoplus_{k=1}^{l} \text{tok}_{\text{context}}(x_k)$.[10]

$$\overrightarrow{h_i} = \text{GRU}_{\rightarrow}(\overrightarrow{h_{i-1}}, \text{tok}_{\text{context}}(x_i)) \tag{6}$$

$$\overleftarrow{h_i} = \text{GRU}_{\leftarrow}(\overleftarrow{h_{i+1}}, \text{tok}_{\text{context}}(x_i)) \tag{7}$$

The representation of the middle segment $h_{i,t}^{\text{mid}}$ at time t is computed using attention over the hidden states of yet another bidirectional RNN GRU_{mid}, which only consumes the segment to normalize $\text{tok}_{\text{mid}}(x_i)$. It is the same attention mechanism as described in Section 5.2, with the crucial difference that the region of attention is restricted to the segment to normalize. The difference between the class of contextual models described here and the baseline sliding window model is clear from the constrast between Equation (3)

10 Because $\text{tok}_{\text{context}}(\cdot)$ can yield multiple tokens for a single segment, these equations may require multiple steps.

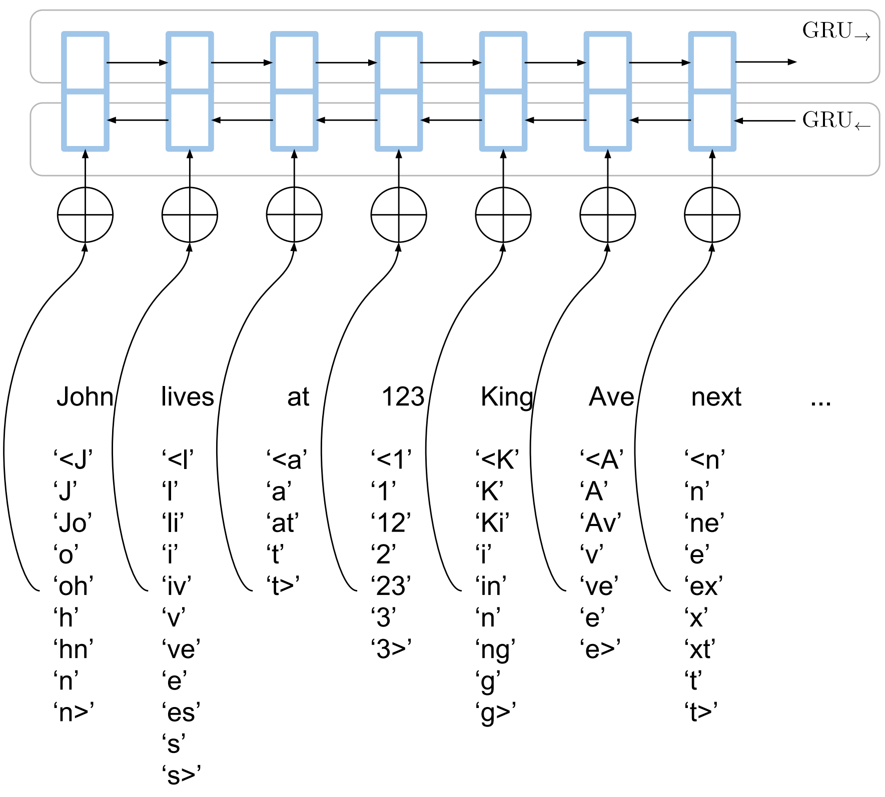
GRU→
GRU←
John
lives
at
123
King
Ave
next
...
'<J' 'J' 'Jo' 'o' 'oh' 'h' 'hn' 'n' 'n>'
'<l' 'l' 'li' 'i' 'iv' 'v' 've' 'e' 'es' 's' 's>'
'<a' 'a' 'at' 't' 't>'
'<1' '1' '12' '2' '23' '3' '3>'
'<K' 'K' 'Ki' 'i' 'in' 'n' 'ng' 'g' 'g>'
'<A' 'A' 'Av' 'v' 've' 'e' 'e>'
'<n' 'n' 'ne' 'e' 'ex' 'x' 'xt' 't' 't>'

Figure 4
Word contextual model based on character *n*-grams within each word.

and Equation (5). While the sliding window model uses surrounding context through the attention mechanism, the contextual models use context directly as a constant vector.

5.3.2 Integrating Context Encodings. The context-aware sequence-to-sequence models presented in the previous section concatenate a vector $h^{\text{mid}}_{i,t}$ with a constant context vector to obtain the input $c_{i,t}$ to the decoder network at each time step. We refer to this approach as the 'concat' strategy. This gives the decoder access to the context outside the to-be-normalized segment at each time step but in doing so increases the dimensionality of the decoder input. An alternative way is to only use the attention-based vector as input to the decoder network (i.e., $c_{i,t} = h^{\text{mid}}_{i,t}$). The constant context is only used to initialize the decoder RNN state; we refer to this as the 'init' strategy. Figure 5 contrasts both methods. We will compare these strategies in terms of both speed and accuracy.

5.4 Stacking Tagging and Contextual Models

In the previous sections, we have assumed that the input sequence is pre-segmented into tokens. However, in real-world applications the normalization model needs to be fed with segmented output produced by a segmenter. A segmenter can be rule-based

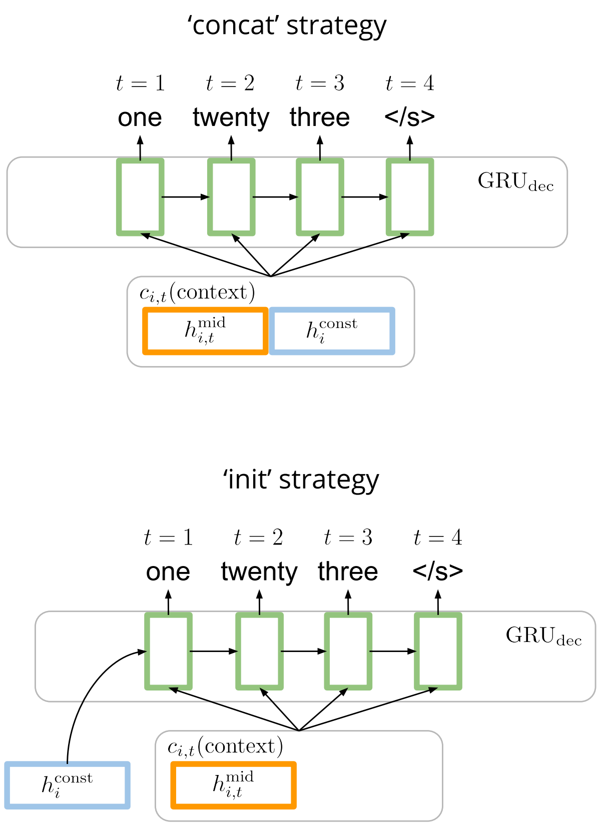

Figure 5
Using the context representations in the decoder network.

like that in Kestrel (Ebden and Sproat 2014), or a neural network based sequence tagging model trained together with a normalization model.

Generally speaking, the majority of tokens are either normal words that are left alone (`<self>`) or punctuation symbols that are mapped to silence (`sil`). We refer to these two types of tokens as *trivial cases* and the remaining tokens as *difficult cases*. Thus, we have a three-class coarse-grained classification problem on input words: `<self>`, `sil`, and *other* (difficult cases).

We treat input segmentation and coarse-grained classification as a joint tagging problem, using a stacked multi-task strategy in modeling. In this model, the hidden states of the sentence context RNNs are shared between the tagger and the normalizer. At training time, the tagging loss and the normalization loss sum up to become the total loss to minimize. At decoding time, we stack the two models. In the first stage, the tagger predicts segmentation and class labels for trivial tokens. In the second stage, the sequence-to-sequence normalization model predicts the normalized output for the *other* input tokens.

The two-stage approach has two potential benefits. It can be more computationally efficient because the tagging RNN has a much smaller output vocabulary than the normalization decoder RNN and does not require an attention mechanism. With multi-task training of a shared input encoder, the speed-up is more significant. The model

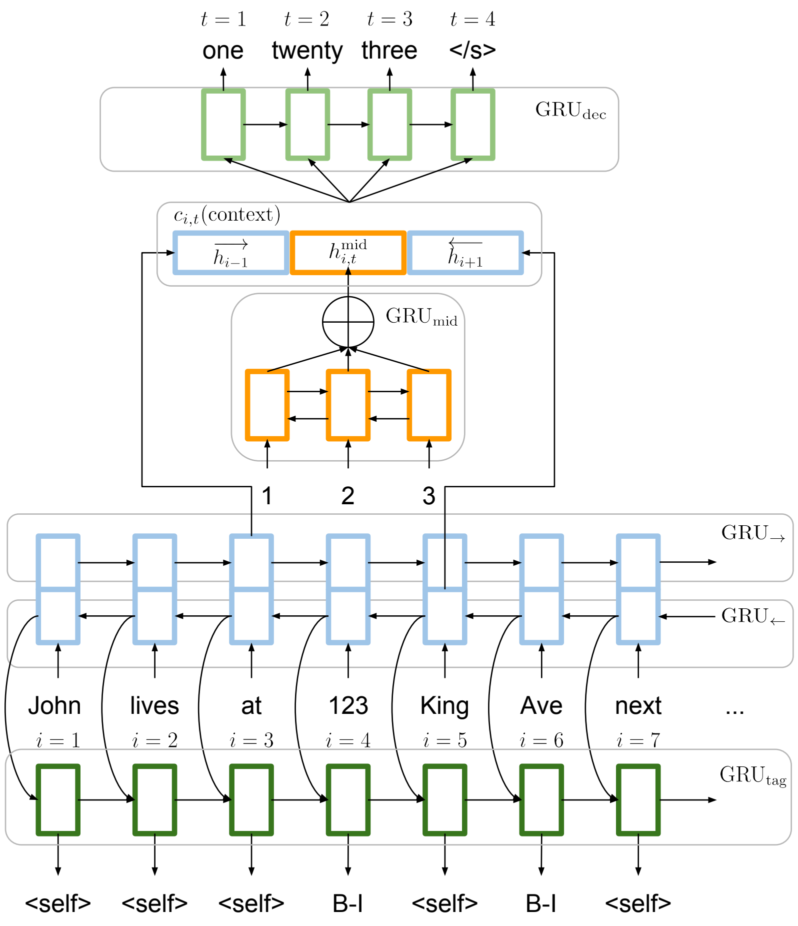

Figure 6
Multi-task encoder RNNs $\mathrm{GRU}_{\rightarrow}$ and $\mathrm{GRU}_{\leftarrow}$ for tagging and contextual text normalization.

has the potential to be more accurate in classifying trivial cases because classification crucially depends on the surrounding context.

The multi-task network is visualized in Figure 6. For the sequence tagging model, the transition actions are the sequence of tag labels. To differentiate trivial cases from difficult cases, we use a three-label tag set: `<self>`, `sil`, and `I`; the latter represents a difficult case. To distinguish the beginning position of a segment from non-beginning positions, we use the labels `B` and `M`. In order to learn the token boundaries and classes jointly, we simply conjoin these two tag sets; for instance, `B-I` refers to the start of a difficult case. The lower portion of Figure 6 shows the RNN architecture for the tagger. We reuse the backward RNN encoder ($\mathrm{GRU}_{\leftarrow}$) to encode the right-to-left context for each word position. In addition to the recurrent link to its previous time step, each hidden state of $\mathrm{GRU}_{\mathrm{tag}}$ also attends to the encoder state at the same word position from the $\mathrm{GRU}_{\leftarrow}$ layer.

5.5 Incorporating Reconstruction Loss

We observe that unrecoverable errors usually involve linguistically coherent output, but simply fail to correspond to the input. In the terminology used in machine translation, one might say that they favor fluency over adequacy. The same pattern has been identified in neural machine translation (Arthur, Neubig, and Nakamura 2016), which motivates a branch of research that can be summarized as enforcing the attention-based decoder to pay more "attention" to the input. Tu et al. (2016) and Mi et al. (2016) argue that the root problem lies in the attention mechanism itself. Unlike traditional phrase-based machine translation, there is no guarantee that the entire input can be "covered" at the end of decoding, and thus they strengthen the attention mechanism to approximate a notion of input coverage. Tu et al. (2017) suggest that the fix can also be made in the decoder RNN. The key insight here is that the hidden states in the decoder RNN should keep memory of the correspondence with the input. In addition to the standard translation loss, there should also be a **reconstruction loss**, which is the cost of translating back to the input from the decoder hidden states. The new training objective is minimizing a (weighted) combination of translation loss and reconstruction loss. This is visualized in Figure 7. The relative weights of reconstruction

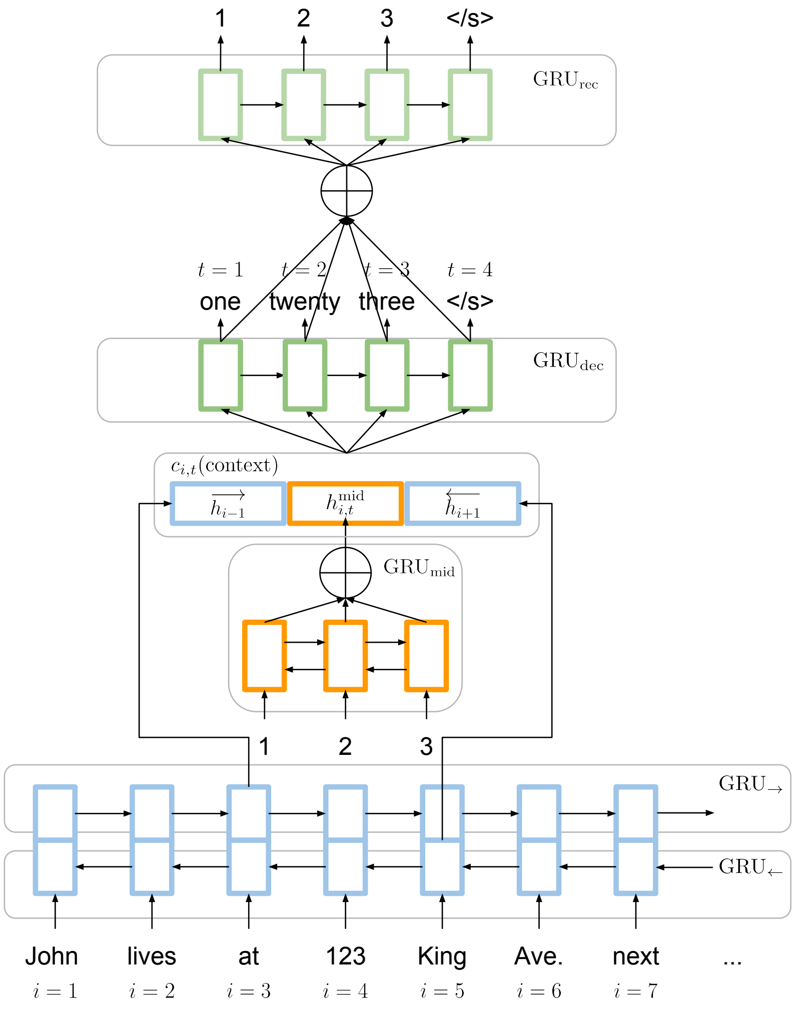
1
2
3
</s>
$\mathrm{GRU}_{\mathrm{rec}}$
$t=1$
$t=2$
$t=3$
$t=4$
one
twenty
three
</s>
$\mathrm{GRU}_{\mathrm{dec}}$
$c_{i,t}$(context)
$\overrightarrow{h}_{i-1}$
$h^{\mathrm{mid}}_{i,t}$
$\overleftarrow{h}_{i+1}$
$\mathrm{GRU}_{\mathrm{mid}}$
1
2
3
$\mathrm{GRU}_{\rightarrow}$
$\mathrm{GRU}_{\leftarrow}$
John
lives
at
123
King
Ave.
next
...
$i=1$
$i=2$
$i=3$
$i=4$
$i=5$
$i=6$
$i=7$

Figure 7
Model with translation loss and reconstruction loss.

loss vs. translation loss represent the relative importance of adequacy and fluency. The reconstruction component of this model can also be used in two ways. First, it can act as a regularizer on the translation model parameters; while it slows down training it does not affect decoding speed. Second, the jointly trained reconstruction model can be used to rerank the translation model outputs by adding the reconstruction loss term to hypotheses in the beam. We report results for the first setting in Section 7.

One caveat is in order, however; our translation problem is not fully symmetric. For words that should be translated to `<self>` and `sil`, reconstruction is infeasible. For such cases, the decoder state sequence is of length 1, and it is unrealistic to expect that a single decoder state is capable of reconstructing the entire input sequence. Hence, in implementation, we skip all examples that are `<self>` and `sil` and only apply reconstruction loss on the remaining examples. This is mathematically equivalent to setting the reconstruction loss to zero for all trivial examples.

In this section we have presented a suite of neural models for text normalization. Before we turn to the experiments and results with these models, we describe how weak covering grammars can be used to mitigate against some of the unrecoverable errors to which the neural models are prone.

6. Inferring Language-Particular Covering Grammars from Data

One effective strategy for mitigating against some of the unrecoverable errors produced by pure neural models is to constrain the output for difficult cases with **covering grammars**, which are grammars that are intended to *cover* the set of verbalizations that are reasonable for a given input. In principle such grammars can be implemented as any kind of constraint on the output, but we focus on finite-state models.

The reasons for the occurrence of unrecoverable errors are not fully understood but plausible culprits are lack of sufficient training data for given expressions, and contexts that might favor a given output enough to make the system disregard the input.[11] For example, the model may have learned to verbalize *3 kg* as *three kilograms*, but it may not generalize what it has learned to cases that have occurred less often in the training data, such as *3mA* (*three milliamperes*) or *329,132 kg* (*three hundred twenty nine thousand one hundred thirty two kilograms*). Large numbers are particularly challenging because it is unlikely the RNN will have seen enough examples of numbers in the training data to fully grasp the number name system of the language. In general, there may not have been enough instances of any given semiotic class (measure, money, cardinal number, date, time . . .) for the system to learn a complete model of that class. Also problematic are cases where a known expression is written in a way that the system has not seen many examples of. If *kilograms* is usually written *kg* in the training data, then the model may simply not know what to do when it sees *3.5 kilograms* in new data.

In other cases, the context may introduce a bias. Consider a date like *April 12, 908*, where the neural system might verbalize this as *April twelfth sil* ***nineteen*** *o eight*, even though it has no problem correctly verbalizing three digit numbers in other contexts. In data sources such as that in Wikipedia perhaps as many as 99% of the examples of the form month-day-year have a four-digit year, reflecting a bias toward events occurring after AD 1000. Thus it is not surprising that the neural system would have a bias to

11 In addition, inconsistencies in training (*2010* as a year could be either *twenty ten* or *two thousand ten*) may also confuse the model.

interpret something as a four-digit date in such contexts, even if the input is a three-digit date.

The role of covering grammars is to limit the choices that the neural system has to choose from to only those reasonable given the input.

The grammars used in a largely hand-crafted text normalization system tend to be quite complex because they need to produce the contextually appropriate rendition of a particular token. In contrast, covering grammars—henceforth CGs—are lightweight grammars, which enumerate the reasonable possible verbalizations for an input, rather than trying to specify the correct version for a specific instance. It is sufficient if the CG tells us that *3mA* could be *three milliampere* or *three milliamperes*, and restricts us from reading it as, for example, *three million liters*.

Clearly it would be possible to construct CGs completely by hand using a grammar development toolkit as was done for the more detailed Kestrel grammars, which were developed using the Thrax grammar compiler (Roark et al. 2012). However, it would obviously be desirable if we could minimize the handwork needed to provide a minimal amount of language-specific information (how one reads measure expressions or months of the year, for example), a small number of hand-developed rules, and otherwise rely on training data to induce the final CG. In this section we explore this approach, and we build off our prior work on inducing number name grammars reported by Gorman and Sproat (2016), generalizing the concept to semiotic classes in general.

6.1 Inducing Number Name Covering Grammars

Gorman and Sproat (2016) reported on a system that can induce a number-name grammar expressed as an FST from a minimal set of training pairs consisting of digit sequences and their verbalizations. The method requires knowing the meanings of all basic number words—*1* is *one*, *20* is *twenty*, *100* is *hundred*, and so forth; and a list of about 300 examples of complex number names and their digit representation built out of these basic number words. In languages that inflect numbers, like Russian, all inflected forms of all number words should be given, so that the entry for *100*, for example, would list all the forms in which the word meaning *100* might appear.

When composing complex number names, languages use a limited set of bases (base 10 is overwhelmingly the most common across the world's languages, with base 20 being a distant second place), and a limited set of arithmetic operations that can be applied to those bases. Summation and multiplication are overwhelmingly the most common operations (with subtraction being a much rarer option), and indeed one typically constructs a complex number name via *sums of products of the bases*; see Hurford (1975) for extensive discussion of the operations used by various languages in the construction of number names. Thus the English number name *three thousand two hundred fifty four* is interpreted as the sum $3 \times 1{,}000 + 2 \times 100 + 50 + 4$. Similarly French *quatre-vingt-dix-sept* (97) is $4 \times 20 + 10 + 7$.

If we take the French case and consider the digit representation *97*, and assuming we allow bases 10 and 20, and potential base words for 80 and 90, then some reasonable ways to factor this are:

$90 + 7$
$80 + 10 + 7$
$4 \times 20 + 10 + 7$
$\ldots$

To compute such factors, assume that we have a (universal) arithmetic FST $\mathscr{A}$ that transduces from a digit string to a lattice of possible arithmetic expressions.

Now consider the verbalization *quatre-vingt-dix-sept* (4 20 10 7). If one knew nothing about French except the meanings of these individual words, there would theoretically be a number of ways that the terms could be combined arithmetically, including:

$4 + 20 + 10 \times 7$
$4 + 20 + 10 + 7$
$4 \times 20 + 10 + 7$
...

Similarly, assume that we have a language-particular lexical map $\mathscr{L}$ that maps a sequence of number words in the language to a lattice of possible arithmetic combinations.

For any training pair (i, o) (e.g., *quatre-vingt-dix-sept* → *97*), we seek a set of paths $\mathscr{P}$ defined as follows

$$\mathscr{P} = \pi_{output}[i \circ \mathscr{L}] \cap \pi_{input}[\mathscr{A}^{-1} \circ o] \tag{8}$$

where π is the projection operation over regular relations. In practice $\mathscr{P}$ will usually contain just one path for a given training pair, though this requires some care in choosing the training pair set; see Gorman and Sproat (2016) for further details.

Given a set of training pairs, we obtain a set of paths, from which a grammar $\mathscr{G}$ can be extracted. A number name reader $\mathscr{C}$ can then be defined as:

$$\mathscr{C} = \mathscr{L} \circ \mathscr{G} \circ \mathscr{A}^{-1} \tag{9}$$

Again, see Gorman and Sproat (2016) for further details on the method sketched above.

Note that in this discussion we have been implicitly assuming that we are talking about *cardinal* (counting) numbers, but this method works equally well for *ordinals* (*first*, *sixth*, *twenty third* ...). Other ways of reading numbers such as digit-by-digit readings can be handled by incorporating the learned number-name transducer $\mathscr{C}$ into a hand-built rule using Thrax or similar grammar development tools.

6.2 Extending CG Induction to Other Semiotic Classes

To see how this approach might be extended to more general semiotic classes, consider the reading of dates, as in the following examples, which give plausible written forms and their verbalization:

Jan 4, 1999	→	January the fourth nineteen ninety nine
03/15/2000	→	March the fifteenth two thousand
15/03/2000	→	March the fifteenth two thousand
04/07/1976	→	the fourth of July nineteen seventy six

The problem is to produce a CG that can map from representations like those on the left, to the verbalizations on the right.

We follow Ebden and Sproat (2014) in breaking this problem down into two stages, namely, a tokenization and markup phase where the written date is mapped to a canonical markup representation, and the second where the markup is verbalized. The markup system is similar to the simpler format used in the open-source version of Kestrel, Sparrowhawk.[12] For the example dates, the following representations are used:

```
Jan 4, 1999   →   date|month:1|day:4|year:1999|
03/15/2000    →   date|month:3|day:15|year:2000|
15/03/2000    →   date|day:15|month:3|year:2000|
04/07/1976    →   date|day:4|month:7|year:1976|
```

For the purely numerical formats of dates, these mappings are largely universal and require no language-particular grammars. For cases like the first example *Jan 4, 1999*, one obviously needs language-particular information on how the months, and in some languages the months and years, may be written, but even here there are only three orderings:

```
month day year
day month year
year month day
```

that are at all common across languages. One may therefore accomplish this first mapping using a few language particular rules added to mostly language-independent ones, which means that one can eschew trying to learn this phase in favor of developing a grammar for this phase by hand, and modifying it as needed for other languages. (In the case of dates, we make use of the tokenizer grammars for dates that are already part of Kestrel.)

The problem then remains to learn the mappings from the markup to the verbalization. Or in other words:

```
date|month:1|day:4|year:1999|    →   January the fourth nineteen ninety nine
date|month:3|day:15|year:2000|   →   March the fifteenth two thousand
date|day:15|month:3|year:2000|   →   March the fifteenth two thousand
date|day:4|month:7|year:1976|    →   the fourth of July nineteen seventy six
```

One further point needs to be made about the third example because it involves a reordering of the month and day in the verbalization. Reordering of arbitrary elements is not possible with regular relations but one can do limited reordering using **pushdown transducers** (PDT) (Allauzen and Riley 2012). PDTs are FSTs that mimic the operations of pushdown automata using a set of paired parentheses, where the parentheses simulate a stack. For example, the non-regular language $a^n b^n$ can be recognized by a PDT that allows any number of a, with a parenthesis "(" before each a, followed by any number of b, with a parenthesis ")" after each b, requiring that the parentheses balance. For our problem, we can use the parentheses in the pushdown operations to remember that if we insert `month:3` before a day, we should delete the original `month:3` after the day. Thus in the method described below, we actually present both orderings of month and day components, leaving the induction method described below to determine which is the best match to the given verbalization.[13] Similar

12 `http://github.com/google/sparrowhawk/tree/master/documentation`.

13 Reading ISO format dates like *2000/05/06* as *May the sixth two thousand* is not supported by the system described here. These would involve either moving the year after the month/day sequence; or moving the month/day combination before the year. In the first case the PDT would have to remember which year was deleted in order to insert it at the end, with thousands of distinct years to remember; or else remember which of 365 month/day combinations appeared after the year in order to insert it before.

reorderings are needed in currency and time expressions, and in some languages, measure expressions, and fractions.

Consider the first mapping above, `date|month:1|day:4|year:1999|` as *January the fourth nineteen ninety nine*. We assume we have language-particular class grammars as follows:

- $\mathscr{C}$ a cardinal number CG as described above.
- $\mathscr{O}$ a similarly-derived ordinal number CG.
- $\mathscr{M}$ month names (including inflected forms) for the language, mapping from numerical designations such as `month:1`.
- $\mathscr{S}$ deletion of markup such as `date` or `|`.
- Any needed language-particular readings not covered above. In English, years have a reading that differs from standard cardinal numbers, and so it is worthwhile to write a grammar $\mathscr{Y}$ that covers years. An example of such a grammar is given in Appendix A.1.

These transducers are assumed to be unweighted. We also assume an edit transducer $\mathscr{E}$ that can replace any string with any other string by inserting or deleting characters at a cost—in the tropical semiring, a large positive number for each insertion or deletion. Each of $\mathscr{C}, \mathscr{O}, \mathscr{M}, \mathscr{S}, \mathscr{E}$, etc., is associated with a tagger T that introduces class-specific tags at the beginning and end of each matched span. These tag insertions have a small positive cost in order to favor longer matching spans in cases where more than one parsing is possible for an input string, when the shortest path is computed (see below). Thus $T[\mathscr{O}]$ would transduce an input *4* to `<ord>fourth</ord>`; these tags will be used subsequently to induce *general* class rules from examples involving *particular class instances*. We can then define a mapper from the markup to the tagged representation as:

$$Map = (\mathcal{T}[\mathscr{C}] \cup \mathcal{T}[\mathscr{O}] \cup \mathcal{T}[\mathscr{M}] \cup \mathcal{T}[\mathscr{S}] \cup \mathcal{T}[\mathscr{E}])^*$$

In other words, we define the *Map* as the concatenative closure of the taggers for each of the known types. To map from the output of *Map* to the verbalization (*January the fourth nineteen ninety nine*) we assume a transducer $\mathscr{D}$ that deletes tags.

Thus the input *i* is composed with *Map*, which is then composed with $\mathscr{D}$, which is composed with the output *o*. Analogously to the induction of number name grammars described above, since we wish to extract the best alignment leading to the best rule from the instance in question, we compute the *shortest path P* that lies in the intersection of the output of *Map* and the input of $\mathscr{D}$, but here we need to preserve the input side as well, since what we need in this case is to learn a transducer that maps between the markup and sequences of tagged spans, and so we do not want to project to the output of $i \circ Map$. Thus:

$$P = ShortestPath[[i \circ Map] \circ \pi_{input}[\mathscr{D} \circ o]].$$

Because of the large number of entities that need to be remembered by the PDT, the approach is not practical in this case.

In the example at hand, `date|month:1|day:4|year:1999|`, and verbalization *January the fourth nineteen ninety nine*. *P* would be as in Figure 8.

In the next stage we *replace* spans that are tagged with tags other than `markup` or `edit` with class labels on both the input and output side, and remove all tags. In the example at hand the output is as in Figure 9.

Given a training corpus of dates and other semiotic classes (which can be the same training corpus as is used to train the neural models), we compute the union of all class-replaced path patterns *P* as in Figure 9. In practice, it is useful to remove patterns that occur less than a minimum number of times in order to remove spurious inductions or patterns that do not generalize well: In our experiments we used

```
ϵ         <markup>
date|     ϵ
ϵ         </markup>
ϵ         <month>
month:1   January
ϵ         </month>
ϵ         <markup>
|day:     ϵ
ϵ         </markup>
ϵ         <edit>
ϵ          the
ϵ         </edit>
ϵ         <ord>
4         fourth
ϵ         </ord>
|         ϵ
ϵ         <year>
1999      nineteen ninety nine
ϵ         </year>
ϵ         <markup>
|         ϵ
ϵ         </markup>
```

Figure 8
Shortest markup path for the input `date|month:1|day:4|year:1999|` and the verbalization *January the fourth nineteen ninety nine*.

```
date|     ϵ
MONTH     MONTH
|day:     ϵ
ϵ          the
ORDINAL   ORDINAL
|         ϵ
YEAR      YEAR
|         ϵ
```

Figure 9
Shortest markup path for the input `date|month:1|day:4|year:1999|` and the verbalization *January the fourth nineteen ninety nine*, after replacement with class labels.

a minimum count of 50. In the final step, we perform recursive transition network replacement on the resulting union to replace the class labels with the correspondingly named FSTs as defined in the language-specific grammar; thus `MONTH` in Figure 9 would be replaced by the $\mathscr{M}$ WFST. This has the result that if we have seen one pair `date|month:1|day:4|year:1999|`→*January the fourth nineteen ninety nine*, the resulting grammar will be able to verbalize, for example, `date|month:3|day:5|year:2012|` as *March the fifth twenty twelve*.

6.3 Using the Covering Grammar with Neural Models

The application of covering grammar constraints to constraining the verbalizations emitted by the neural models requires some care. Here we briefly describe the issues and our approach to solving them.

When we apply the covering grammar to an input token, if the covering grammar produces a lattice of possible verbalizations for that token, then we use that lattice to guide the neural model. This is similar in principle to recent work by Ng, Gorman, and Sproat (2017), in which covering grammars are used to generate hypotheses for a (non-neural) reranking model. But if the covering grammar fails—for instance, because the token is out of scope for the covering grammar (as would be the case for a typical word that would be mapped to `<self>`)—then the neural model is unconstrained. We implement the application of a covering grammar constraint as on-the-fly intersection (Figure 10) between two deterministic automata, where the lattice from the covering grammar is projected onto the output, and determinized to become a deterministic finite state automaton; and the RNN decoder can be considered as a weighted deterministic automaton with an infinite state space.

Suppose first of all that one has trained a covering grammar and then wants to apply the constraints of the grammar (e.g., that *3 kg* can be either *three kilogram* or *three kilograms*, but nothing else) at decoding time. The obvious method of applying the constraint and then computing the softmax as used in Sproat and Jaitly (2016, 2017) is potentially problematic because it can distort the rankings of analysis paths produced by the system. Suppose for an input *123* we have two outputs (among many), namely,

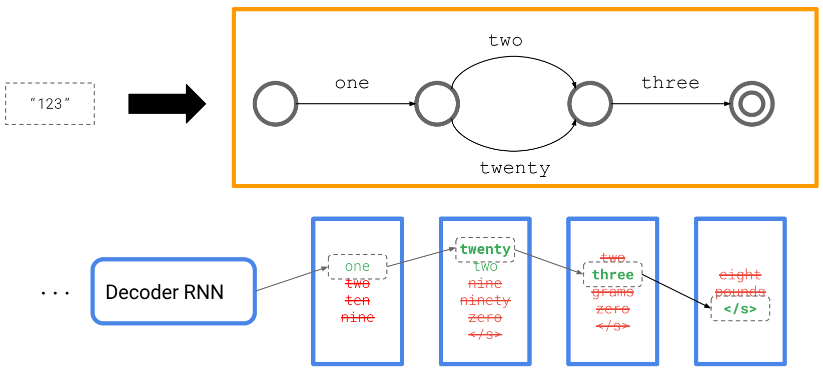

Figure 10
An illustration of on-the-fly intersection with a covering grammar; the lattice produced from the input 123 projects to an acceptor that only accepts `one two three` and `one twenty three`, and intersecting this with the RNN outputs at each position (where higher ranking output is indicated by a higher position in the cell) will result in the output `one twenty three`.

`one two three </s>` and `one twenty three </s>`, where `</s>` is the stop symbol, and suppose that:

$$P(\texttt{one two three </s>}) = P(\texttt{one two three}) \times P(\texttt{</s>} \mid \texttt{one two three}) \tag{10}$$
$$= 0.4 \times 0.5 \tag{11}$$

$$P(\texttt{one twenty three </s>}) = P(\texttt{one twenty three}) \times P(\texttt{</s>} \mid \texttt{one twenty three}) \tag{12}$$
$$= 0.4 \times 0.9 \tag{13}$$

and another, impossible, prediction:

$$P(\texttt{one two three four </s>}) = P(\texttt{one two three four}) \times P(\texttt{</s>} \mid \texttt{one two three four}) \tag{14}$$
$$= 0.3 \times 0.9 \tag{15}$$

Given the above, the probability of one two three being string final is less than the comparable probability of one twenty three; so that

$$P(\texttt{one two three </s>}) < P(\texttt{one twenty three </s>}) \tag{16}$$

Now suppose we have a covering grammar that in this instance only allows `one two three` and `one twenty three`, thus filtering out other options like `one twenty two`. Then letting Q represent the probabilities for the combined system,

$$Q(\texttt{one two three}) = \frac{P(\texttt{one two three})}{P(\texttt{one two three}) + P(\texttt{one twenty three})} \tag{17}$$
$$= 0.5 \tag{18}$$
$$= Q(\texttt{one twenty three}) \tag{19}$$

and because both must now be string final (there is no other possibility but `</s>` after either sequence),

$$Q(\texttt{</s>} \mid \texttt{one two three}) = Q(\texttt{</s>} \mid \texttt{one twenty three}) = 1 \tag{20}$$

so now

$$Q(\texttt{one two three </s>}) = Q(\texttt{one twenty three </s>}) \tag{21}$$

Instead, if we wish to use a covering grammar with an already trained neural model *we must first apply softmax and then constrain.*

Alternatively one can constrain *during training*, which will guarantee that the ranks between paths will be preserved at decoding time, in which case one can at decoding time safely constrain and then apply softmax. The main drawback of this approach is that if the lattice produced by the covering grammar for the given training input does not contain the true output as defined in the training, then that training instance is lost. In our experiments discussed later we report results both on training *and* decoding using CGs, as well as decoding only, first applying softmax and then constraining.

7. Experiments

7.1 Description of the Data

Our data consist of all English and Russian Wikipedia text that can be properly decoded as UTF-8, which was then divided into sentences, and run through the Google TTS system's Kestrel text normalization component to produce verbalizations. The original data were mined in Spring 2016, and has been released on GitHub,[14] divided into 100 files per language, in the format described in Table 2.

We chose Wikipedia for three reasons. First of all, Wikipedia is a reasonable application of TTS in that Wikipedia text is already found as input to TTS in, for example, the question-answering component of the Google Assistant.[15] Second, Wikipedia is a reasonable instance of what might be called "general" English or Russian text. Finally, there are very few legal limitations on the use of Wikipedia text.

To the extent that we look to provide accuracy commensurate with this heavily engineered hand-built normalization system, this is an appropriate data set for evaluating neural text normalization. The accuracy of the Kestrel annotations is high: A manual analysis of about 1,000 examples from the test data suggests an overall error rate of approximately 0.1% for English and 2.1% for Russian. The largest category of errors for Russian involves years being read as cardinal numbers rather than the expected ordinal form.

Although the test data were of course taken from a different portion of the Wikipedia text than the training and development data, still a huge percentage of the individual tokens of the test data—98.9% in the case of Russian and 99.5% in the case of English—were found in the training set. There is thus a large potential for the system simply memorizing certain examples. Nonetheless, as Sproat and Jaitly (2016) show for an earlier neural model, the system is not *simply* memorizing.

Our full data set consists of 1.1 billion words of English, and 290 million words of Russian text. Because ultimately the goal is to produce systems that can be trained on *hand annotated* data, one must therefore consider the amount of data that it is practical to hand annotate. Clearly, nearly a billion words or even 290 million words is not very practical. Therefore, in our experiments we train using the first file for English (about 10 million tokens) and the first four files for Russian (about 11 million tokens), and evaluate on the first 92,000 tokens of the last file for English, and the first 93,000 tokens of the last file for Russian. Files 90–94 are used for development.[16]

In late 2017 we hosted a text normalization competition on Kaggle with both our English and Russian data sets. [17] Sproat and Gorman (2018) provide a short review of the competition results. As part of that competition we developed new test sets consisting of 70,000 Wikipedia sentences (about 1 million tokens) in both languages. These data were mined in Fall 2017, and were filtered to exclude any sentences present in the original data set released on GitHub. This test data set is challenging, in particular for English, because there were some changes to the way Kestrel tokenizes certain semiotic classes, in particular measure expressions, between when we created the first data set and when

14 `https://github.com/rwsproat/text-normalization-data`.

15 `https://en.wikipedia.org/wiki/Google_Assistant`.

16 Note that this corresponds to the train-dev-test split that can be found in the script available with the GitHub data (`https://github.com/rwsproat/text-normalization-data/blob/master/split.sh`).

17 See `https://www.kaggle.com/c/text-normalization-challenge-english-language` and `https://www.kaggle.com/c/text-normalization-challenge-russian-language`.

we created this data set. In the original data set measures with fully spelled measure words like *5 kilograms* were tokenized with the measure term separate, whereas in the later set such data were tokenized as a single measure token. Not surprisingly, neural models have some problems with this novel tokenization since in the training data they have not seen many instances of fully spelled measures in the same token as a preceding number, and the model they have learned does not generalize well. One could argue that the test set is therefore unfair, but on the other hand, it is not unreasonable to expect that new data, say from a slightly new domain, will show differences of this kind. In any case, as we will see, covering grammars are particularly effective at overcoming this limitation. In the following description, we also report results on the Kaggle test data (using the same training data as was used to evaluate on the main test set).

Finally, we have collected a set of manually annotated English Wikipedia sentences for internal use. We describe these data and report results on them in Section 7.6.

7.2 Context Representation

We first compare different representations of linguistic context. Recall that the sliding window model, described in Section 5.2, uses a character context within a fixed three-word window of the token to be normalized. The architecture described in Section 5.3 can be instantiated with different choices of basic input units for sentence context.

Table 3 compares the sliding window models with three types of input units: characters, word pieces, and word features (character 3-grams), on the English and Russian test sets. Here and below, *All* denotes accuracies/errors for all tokens, *Semiotic class* denotes the interesting classes (i.e., not `sil`, `<self>`, or punctuation), and *Sentence* is the per-sentence accuracy/error. We observe first that two-level contextual models are more efficient than the sliding-window baseline model due to sharing of context. Secondly, we find that inference is more efficient with larger contextual units (see *speed* in the last column), with word feature context having the best overall accuracy–speed tradeoff.

Table 3
Results for models of different context representations: Char(acter), W(ord)P(iece), W(ord)F(eature). For the contextual models we specify the tokenization level of the sentential context. *: difference is statistically significant (p < .05 on the McNemar test) in comparison with the sliding window baseline. For speed, e.g., "1.4x" means 1.4 times faster.

	Accuracy(Error)			Speed
	All	Semiotic class	Sentence	
English (Standard):				
Sliding window	99.79% (0.21%)	98.20% (1.80%)	97.99% (2.01%)	1.0x
Context (Char)	99.79% (0.21%)	98.20% (1.80%)	97.87% (2.13%)	1.3x
Context (WP)	99.79% (0.21%)	98.35% (1.65%)	97.76% (2.24%)	1.4x
Context (WF)	99.84% (0.16%)*	98.30% (1.70%)	98.20% (1.80%)	1.4x
Russian (Standard):				
Sliding window	99.64% (0.36%)	97.31% (2.69%)	95.61% (4.39%)	1.0x
Context (Char)	99.65% (0.35%)	97.26% (2.74%)	95.70% (4.30%)	1.8x
Context (WP)	99.61% (0.39%)	96.94% (3.06%)*	95.26% (4.74%)	1.8x
Context (WF)	99.62% (0.38%)	97.01% (2.99%)*	95.46% (4.54%)	2.0x

Table 4
Results for models with different context utilization strategies. *: statistically significant ($p < .05$ on the McNemar test).

		Accuracy (Error)			Speed
		All	Semiotic class	Sentence	
English (Standard):					
Word feat.	init	99.84% (0.16%)	98.30% (1.70%)	98.20% (1.80%)	1.4x
	concat	99.82% (0.18%)	98.42% (1.58%)	97.96% (2.04%)	1.2x
Russian (Standard):					
Word feat.	init	99.62% (0.38%)	97.01% (2.99%)	95.46% (4.54%)	2.0x
	concat	99.66% (0.34%)*	97.32% (2.68%)*	95.80% (4.20%)	1.7x

The results of the two context utilization strategies discussed in Section 5.3.2 are shown in Table 4. The 'init' strategy is clearly more efficient. It is also more accurate on the sentence level for English. For Russian, the 'concat' strategy is more accurate in all metrics, indicating the importance of context.

7.3 Stacking Tagging and Normalization

We then consider the effect of stacking the tagging model and the normalization model and training them with a multi-task loss as described in Section 5.4.

Table 5 shows the results of accuracy and speed on the English and the Russian test sets. For both languages, stacked models for both languages are clearly more efficient than their pure sequence-to-sequence counterparts: four times faster in English, and faster yet in Russian. Multi-task training with tagging loss results in an even more accurate model for Russian, presumably due to a regularization effect. Only 0.74% of English sentences and 0.37% of Russian sentences contain segmentation errors, indicating that the underlying RNN tagger is highly accurate.

Table 5
Results with stacked tagging and normalization models. "+ tag. loss": with multi-task loss of tokenization and semiotic class tagging. "+ tok/tag": with a stacked tokenization and semiotic class tagging model.

	F_1	Acc (Err)		Speed
	All	Segmentation	Sentence	
English (Standard):				
Word-feat.	99.84% (0.16%)	100.00% (0.00%)	98.20% (1.80%)	1.4x
+ tag. loss	99.83% (0.17%)	100.00% (0.00%)	98.12% (1.88%)	1.5x
+ tok./tag.	99.75% (0.25%)	99.26% (0.74%)	97.75% (2.25%)	4.0x
Russian (Standard):				
Word-feat.	99.62% (0.38%)	100.00% (0.00%)	95.46% (4.54%)	2.0x
+ tag. loss	99.65% (0.35%)	100.00% (0.00%)	95.62% (4.38%)	2.2x
+ tok./tag.	99.59% (0.41%)	99.63% (0.37%)	95.46% (4.54%)	5.7x

Table 6
Results with and without reconstruction loss. "+ reconstr. loss": with reconstruction loss.

	Accuracy (Error)		
	All	Semiotic class	Sentence
English (Kaggle):			
Best model	99.17% (0.83%)	90.08% (9.92%)	89.58% (10.42%)
+ reconstr. loss	99.15% (0.85%)	90.10% (9.90%)	89.43% (10.57%)
Russian (Kaggle):			
Best model	99.01% (0.99%)	94.09% (5.91%)	89.02% (10.98%)
+ reconstr. loss	99.01% (0.99%)	94.10% (5.90%)	89.05% (10.95%)

Even though the stacked models are performing a more challenging task—text normalization without assuming existence of pre-segmented semiotic class tokens—the degradation in accuracy is small for English and smaller for Russian.

7.4 Reconstruction Loss

Table 6 shows the effect of adding reconstruction loss, as described in Section 5.5, to the best contextual models for English and Russian. We report results on the much-larger Kaggle test sets to minimize the effect of noise. Unfortunately, there is little effect and the differences are statistically insignificant ($p > .05$), according to the McNemar test (Gillick and Cox 1989).

7.5 Covering Grammars

In this section, we show results of adding covering grammars to the training or decoding of the RNN models. We report results for both English and Russian.

Before turning to the main result, it is worth considering the **coverage** of the covering grammars on held-out data, defined as:

$$C = \frac{\sum_{i \in x} I_i}{|x|} \tag{22}$$

where x is the set of test examples where the grammar $\mathscr{G}$ produces a non-null output, and $I_i = 1$ for a given input i iff the true output is found in $\pi_{output}[i \circ \mathscr{G}]$.

We measured this for the covering grammars for English and Russian on the second English file (roughly 10 million tokens) and files 5–8 of Russian (also roughly 10 million tokens). The English grammar matched about 500 thousand tokens, with about 99% coverage; the Russian grammar matched about 400 thousand tokens, with about 99.5% coverage.

Table 7 demonstrates that the overall best choice is training the RNN without covering grammars and decoding with covering grammars. We speculate that covering grammars used to constrain training may be causing overfitting. Table 8 breaks the results down by semiotic class, with significant improvements ($p < .05$ on the McNemar test) indicated. We observe improvements for four interesting semiotic classes: DATE,

Table 7
Results using the best neural model with and without induced covering grammars. "+ CG decoding": covering grammar used during decoding; "+ CG training": covering grammar used during training and decoding. *: difference is statistically significant ($p < .05$ on the McNemar test) in comparison with the baseline model without CG.

	Accuracy (Error)		
	All	Semiotic class	Sentence
English (Standard):			
Best model	99.84% (0.16%)	98.30% (1.70%)	98.20% (1.80%)
+ CG decoding	99.84% (0.16%)	98.36% (1.64%)	98.24% (1.76%)
+ CG training	99.84% (0.16%)	98.56% (1.44%)*	98.17% (1.83%)
English (Kaggle):			
Best model	99.17% (0.83%)	90.08% (9.92%)	89.58% (10.42%)
+ CG decoding	99.22% (0.78%)*	90.69% (9.31%)*	90.09% (9.91%)
+ CG training	99.20% (0.80%)*	90.71% (9.29%)*	89.87% (10.13%)
Russian (Standard):			
Best model	99.65% (0.35%)	97.17% (2.83%)	95.62% (4.38%)
+ CG decoding	99.64% (0.36%)	97.19% (2.81%)	95.58% (4.42%)
+ CG training	99.62% (0.38%)	97.01% (2.99%)	95.26% (4.74%)
Russian (Kaggle):			
Best model	99.01% (0.99%)	94.09% (5.91%)	89.02% (10.98%)
+ CG decoding	99.01% (0.99%)*	94.12% (5.88%)*	89.03% (10.97%)
+ CG training	98.94% (1.06%)*	94.17% (5.83%)*	88.13% (11.87%)

CARDINAL, MEASURE, and MONEY. But there are also two degradations, in DECIMAL and TELEPHONE, for Russian.

See Section 7.8 for a deeper dive into the benefits of using covering grammars.

7.6 Results on Hand-Annotated Data

Finally, we report briefly on some experiments we have conducted on human-corrected data. Although we currently have no plans to open-source this data set, we feel it is nonetheless useful to give the reader a sense of how the system performs on a smaller, but cleaner and more accurate set of data.

English data from the main Kestrel-annotated data reported earlier was prepared that included longer sentences (to minimize the number of sentence fragments), where each sentence included at least one non-trivial text normalization token. These data were then sent to outside vendors, who were tasked with correcting the verbalized output where needed. Because the correction tool presented data one token per line, we also presented the original Wikipedia sentence to raters, in order to make it easier to determine cases where the tokenization might be unclear. Every file was sent to three independent annotators. Note that the external annotators are not expected to have linguistic training, merely be competent speakers of the language.

Tokens where all three annotators agreed were assumed to be correct, and if two out of three annotators agreed, we picked the majority rating. For tokens where none of the annotators agreed, the entire sentence containing those tokens was sent to internal annotators for quality control (QC). Internal annotators are all trained linguists. Of the tokens

Table 8
Semiotic class accuracies (errors) with and without covering grammars. *: statistically significant.

	English		Russian	
	Standard	Kaggle	Standard	Kaggle
PLAIN:				
Best model	99.9% (0.10%)	99.3% (0.70%)	99.8% (0.20%)	99.1% (0.90%)
+ CG decoding	99.9% (0.10%)	99.3% (0.70%)	99.8% (0.20%)	99.1% (0.90%)
PUNCT:				
Best model	99.9% (0.10%)	99.9% (0.10%)	99.9% (0.10%)	97.5% (2.50%)
+ CG decoding	99.9% (0.10%)	99.9% (0.10%)	99.9% (0.10%)	97.6% (2.40%)
DATE:				
Best model	99.5% (0.50%)	98.9% (1.10%)	98.2% (1.80%)	98.3% (1.70%)
+ CG decoding	99.5% (0.50%)	99.0% (1.00%)*	98.2% (1.80%)	98.3% (1.70%)*
LETTERS:				
Best model	97.5% (2.50%)	95.9% (4.10%)	98.9% (1.10%)	96.2% (3.80%)
+ CG decoding	97.5% (2.50%)	95.9% (4.10%)	98.9% (1.10%)	96.3% (3.70%)
CARDINAL:				
Best model	99.4% (0.60%)	98.8% (1.20%)	97.2% (2.80%)	89.9% (10.10%)
+ CG decoding	99.5% (0.50%)	99.1% (0.90%)*	97.1% (2.90%)	90.1% (9.90%)*
VERBATIM:				
Best model	99.9% (0.10%)	99.1% (0.90%)	100.0% (0.00%)	98.6% (1.40%)
+ CG decoding	99.9% (0.10%)	99.1% (0.90%)	100.0% (0.00%)	98.6% (1.40%)
MEASURE:				
Best model	97.2% (2.80%)	73.3% (26.70%)	92.2% (7.80%)	89.9% (10.10%)
+ CG decoding	97.9% (2.10%)	88.5% (11.50%)*	92.0% (8.00%)	90.1% (9.90%)*
ORDINAL:				
Best model	98.1% (1.90%)	98.1% (1.90%)	99.3% (0.70%)	96.9% (3.10%)
+ CG decoding	99.0% (1.00%)	98.3% (1.70%)	99.3% (0.70%)	97.0% (3.00%)
DECIMAL:				
Best model	100.0% (0.00%)	98.9% (1.10%)	91.7% (8.30%)	77.6% (22.40%)*
+ CG decoding	100.0% (0.00%)	98.9% (1.10%)	91.7% (8.30%)	76.5% (23.50%)
ELECTRONIC:				
Best model	63.3% (36.70%)	3.4% (96.60%)	64.6% (35.40%)	32.2% (67.80%)
+ CG decoding	63.3% (36.70%)	3.4% (96.60%)	64.6% (35.40%)	32.2% (67.80%)
DIGIT:				
Best model	86.4% (13.60%)	83.1% (16.90%)	93.8% (6.20%)	99.6% (0.40%)
+ CG decoding	86.4% (13.60%)	83.3% (16.70%)	93.8% (6.20%)	99.6% (0.40%)
TELEPHONE:				
Best model	94.6% (5.40%)	88.5% (11.50%)	95.5% (4.50%)	95.5% (4.50%)*
+ CG decoding	94.6% (5.40%)	88.5% (11.50%)	95.5% (4.50%)	93.4% (6.60%)
MONEY:				
Best model	97.3% (2.70%)	89.8% (10.20%)	89.5% (10.50%)	77.8% (22.20%)
+ CG decoding	97.3% (2.70%)	92.4% (7.60%)*	84.2% (15.80%)	79.9% (20.10%)*
FRACTION:				
Best model	81.3% (18.70%)	77.1% (22.90%)	82.6% (17.40%)	78.8% (21.20%)
+ CG decoding	81.3% (18.70%)	77.1% (22.90%)	82.6% (17.40%)	78.8% (21.20%)
TIME:				
Best model	75.0% (25.00%)	90.2% (9.80%)	75.0% (25.00%)	78.8% (21.20%)
+ CG decoding	75.0% (25.00%)	90.2% (9.80%)	75.0% (25.00%)	79.7% (20.30%)

that were not sent for QC, 97.2% had three-way agreement, and the remainder two-way agreement. Of course even three-way agreement does not guarantee correctness. A rough manual analysis of a sample of cases discussed below where the system actually produced a correct output even though it differed from the gold standard, suggested an approximately 0.2% error rate for the human annotators.

As of the time of writing, we have collected 3.1 million tokens of data, of which we use 2.5 million as training and 540K tokens as testing.

We trained our baseline text normalization model and a covering grammar. Because the data in this case are not annotated with gold semiotic classes, we just report overall accuracies. The baseline model achieves an error rate of 0.82% (4,410 errors—corresponding to an accuracy of 99.18%); adding CG decoding reduces it to 0.80% (4,330 errors or 99.20% accuracy). Eighty examples are impacted by the addition of CG decoding, all of which involve correcting an unrecoverable error such as reading *€90 million* as *ninety million dollars*. As we also discuss in Section 7.8, the covering grammars appear to be effective at targeting unrecoverable errors.

We then went through all the errors in the system with the CG decoding in more detail and classified them into three categories:

- **Unrecoverable**, such as reading *Ib* in *Excalibur Ib projectiles* as *the fourth*.
- **Recoverable**, such as reading the interjection *pssst* as a letter sequence.
- **Not an error**, either cases where the system produced an acceptable variant, or the human annotation was actually wrong.

In the last category, occasionally the human annotators introduced errors, but more commonly they failed to correct errors that were made by the Kestrel system that generated the data. For example, there were 97 cases where human annotators accepted Kestrel's prediction of a letter sequence reading, but the correct reading is `<self>`; for instance, the acronym *UNESCO* was predicted to be a letter sequence but is conventionally read as if it were an ordinary word. The last category of non-errors comprised 2,794, examples, which reduces the number of real errors to 1,536 or 0.28% (accuracy: 99.72%).

Finally, it is worth asking how well the neural model actually compares with Kestrel. Because the annotators were correcting Kestrel's output, we might in principle take any correction as an indication of an error in Kestrel. Unfortunately, things are not quite so simple since the annotation guidelines given to the raters in many cases resulted in a different annotation where Kestrel's prediction was not really an error. We therefore randomly sampled the differences between Kestrel and the annotator output, and derived an estimate of a true Kestrel error rate of about 3,600 errors, or 0.67%. This is approximately three times the error rate of the neural system. Examples of the errors corrected by the neural model include reading *3rd ed* as *third edition* (*ed* is not expanded by Kestrel), and *id* as a letter sequence rather than the word *id* in *id: G000404*.

Although this seems quite promising for a neural network approach to this problem, one must remember that the neural model still makes errors that are unrecoverable, and that are not currently handled by the covering grammar. These errors can be quite bad as, for instance, misreading the Roman numeral *II* as *the third*. In a few rare cases, the neural network models even map an input word onto a completely unrelated output word, for instance, reading *but* as *Sunday*. Kestrel does not make these sort of errors under any circumstance. So whereas the error rates are lower for the neural system, there is still work to be done to improve on the *kinds* of errors produced.

7.7 Comparison with Other Published Results

Since the publication of Sproat and Jaitly (2016) and Sproat and Jaitly (2017), and the Kaggle competition based on the same data (Sproat and Gorman 2018), there have

Table 9
Accuracies for the four models presented by Yolchuyeva, Németh, and Gyires-Tóth (2018) (first four columns), Pramanik and Hussain (2018), and our own best model results from Table 8 *without* covering grammar restrictions, broken down by semiotic class. Results for best system(s) in each category are underlined.

	Y Mod. 1	Y Mod. 2	Y Mod. 3 (CBOW)	Y Mod. 3 (SG)	P&H	Ours (−CG)
PLAIN	99.7	99.7	99.7	99.8	99.4	99.9
PUNCT	99.9	99.9	99.9	99.9	99.9	99.9
DATE	98.72	98.76	98.90	98.99	99.7	99.5
LETTERS	80.35	80.50	79.80	81.22	97.1	97.5
CARDINAL	98.63	95.74	98.76	98.89	99.4	99.4
VERBATIM	96.53	96.76	96.89	97.22	99.4	99.9
MEASURE	93.14	88.60	93.01	91.34	97.1	97.2
ORDINAL	92.46	91.76	92.46	93.99	98.0	98.1
DECIMAL	96.12	98.53	96.3	96.12	98.9	100.0
MONEY	86.75	79.79	97.9	87.97	97.3	97.3
DIGIT	66.16	61.11	66.83	68.01	79.5	86.4
TIME	55.33	54.66	51.33	60.6	75.0	75.0
FRACTION	28.47	32.63	27.7	37.5	92.3	81.3

been a few publications on alternative neural approaches to the same problem. Notable among these are Yolchuyeva, Németh, and Gyire-Tóth 2018, who apply convolutional networks (LeCun and Bengio 1995), and Pramanik and Hussain (2018), who use differentiable neural computers (DNC) (Graves et al. 2016). It is instructive to briefly compare our results with theirs.

Yolchuyeva, Németh, and Gyire-Tóth 2018 compare several models. Their Model 1 is a unidirectional LSTM, and Model 2 a bidirectional LSTM. Model 3 uses convolutional layers, with two variants for embedding: continuous bag of words (CBOW) and skip-gram (SG). They reported results for English, and the accuracies for all these models are given in columns 2–5 of Table 9. Unfortunately, it is hard to compare our system directly with their results because although they do seem to use the same data as was used by Sproat and Jaitly (2017) for training (i.e., the same as used here), they actually split this data into training, development, and test, which means that their test set is not identical to either of the ones we report on.

Accuracies for the DNC system of Pramanik and Hussain (2018) are given in the sixth column of Table 9, along with the results of our best system *without* covering grammar constraints, in the final column. Once again, the systems are not directly comparable, because although Pramanik and Hussain do test on our "standard" test set, they train on the first 20 million tokens of the entire published data set rather than, as in our case, the first 10 million.

In general, Pramanik and Hussain's system does better than any of the systems reported by Yolchuyeva et al., but for most categories, our own system does better than either. Our system is tied with the DNC system for PUNCT, CARDINAL, MONEY, and TIME. The DNC system gets the highest accuracies for DATE and FRACTION.

Table 10 compares accuracies for Russian from Pramanik and Hussain (2018) and our own best system without the covering grammar restrictions. In this case, Pramanik and Hussain's training data set was the same as our own, so the results are more fairly comparable. Once again, there is no obvious benefit of the DNC overall. The systems

Table 10
Accuracies for Russian from Pramanik and Hussain (2018) and our own best model results from Table 8 *without* covering grammar restrictions, broken down by semiotic class. Results for best system(s) in each category are underlined.

	P&H	Ours (−CG)
PLAIN	99.5	99.8
PUNCT	99.9	99.9
DATE	97.3	98.2
LETTERS	99.1	98.9
CARDINAL	94.2	97.2
VERBATIM	100.0	100.0
MEASURE	89.8	99.2
ORDINAL	94.6	99.3
DECIMAL	90.0	91.7
MONEY	89.4	89.5
DIGIT	100.0	93.8
TIME	75.0	75.0
FRACTION	82.6	82.6

achieve the same accuracies on PUNCT, VERBATIM, TIME, and FRACTION. The DNC system performs better on DIGIT and LETTERS. Our system performs better on PLAIN, MEASURE, CARDINAL, ORDINAL, DECIMAL, and MONEY.

On balance, then, the convolutional models of Yolchuyeva, Németh, and Gyires-Tóth (2018) seem to underperform the state of the art, and the DNC models of Pramanik and Hussain (2018) are not an obvious win. Considering that DNCs are a much more complex and "deep" architecture than the systems we are proposing, one has to question whether the benefits outweigh the costs of adopting such a system.

7.8 Detailed Analysis of Errors with and without Covering Grammars

Table 11 breaks down the numbers of errors per semiotic class where the raw neural model and the neural model plus CG decoding produce a different result; in other words, cases where at least one of the two systems is in error, but do not produce the same error. Because the numbers of errors is much larger in the Kaggle set—reflecting the larger size of the Kaggle test data, and the differences from the standard set already described—we concentrate in this discussion mostly on this set.

7.8.1 English Kaggle Errors. The covering grammar corrected the only two ORDINAL errors in the set (*1243rd* as *twelve forty three* and *19961st* as *one thousand nine hundred ninety six thousand six hundred nineteen*). For DIGIT, the three covering grammar errors were all instances of reading a digit sequence as a cardinal number—for example, *14053* as *fourteen thousand fifty three*: the best neural system alone produced *fourteen o five three* for this example, which is also arguably acceptable.

The four MONEY errors with the covering grammar were all instances of reading a fully specified currency using a shorter name. For example: *PKR 60 billion* as *sixty*

Table 11
Per-semiotic class error counts with and without covering grammars, covering only classes where there is a difference between using and not using the covering grammar.

	English		Russian	
	Standard	Kaggle	Standard	Kaggle
CARDINAL:				
Best model	1	49	1	73
+ CG decoding	0	15	3	36
DATE:				
Best model	2	24	0	33
+ CG decoding	1	4	0	3
DECIMAL:				
Best model	0	0	0	1
+ CG decoding	0	0	0	7
DIGIT:				
Best model	0	4	0	0
+ CG decoding	0	3	0	0
MEASURE:				
Best model	2	453	2	38
+ CG decoding	1	75	3	29
MONEY:				
Best model	0	18	0	9
+ CG decoding	0	4	1	4
ORDINAL:				
Best model	1	2	0	4
+ CG decoding	0	0	0	2

billion rupees, rather than the gold form *sixty billion Pakistani rupees*. These are arguably acceptable variations: The neural model alone produced *six hundred billion euros*, which gets both the currency and the number wrong.

In the case of DATE, the four CG errors involved misreading cases like *3/1/03* as *March first three* (rather than *o three*); the ambiguous date *28-05-16* as *May twenty eighth, sixteen* (gold is *the sixteenth of May, twenty eight*); and the dubious date *1 May, 3272* as *the first of May three thousand two hundred seventy two* (the neural net alone gets *one million May third twenty two*).

CARDINAL errors from the CG involve misreading (mostly) long numbers as digit sequences. In most of these cases the CG does in fact allow the cardinal reading, but also offers the digit-by-digit reading, which in the cases at hand the neural model prefers. For example, in the case of *10000*, the CG allows both *ten thousand* and *one o o o o*, and given the choice, the neural model picks the latter: left to its own devices, the neural model produced *one thousand*.

Finally a large number of the CG MEASURE errors involved singularizations of measures that were written as plurals. Thus, for example, *45 minutes* was verbalized as *forty five minute*. The covering grammar allows both but, again, the neural model for some reason prefers the singular form. In this particular case the neural model on its own produced *forty five millimeters*. Among the 75 errors, there was just one unrecoverable error, where *4 kg* was verbalized by the CG as *four grams*, evidently pointing to an error in the induction.

7.8.2 Russian Kaggle Errors. For the Russian regular set two of the categories—CARDINAL, MEASURE—showed slightly higher numbers of errors with the CG than without it. For CARDINAL this turned out to be due to a bug in the data, where three numbers were predicted by Kestrel to be *sil*, whereas the CG correctly decoded them as a form of the correct number: thus *3* as *трех*. The MEASURE cases were all of a similar nature, where Kestrel had predicted *sil*: thus *80%* predicted correctly with the CG as *восьмидесяти процентов*.

Turning to the Kaggle test set, the CARDINAL errors with the CG were mostly cases where the CG in fact corrected an error made by the neural model alone, but still got the morphological form wrong. Thus *200* was predicted by the neural model to be *ноль* ('null'), and the CG corrected this to *двухсот* 'two hundred' (genitive case), whereas the reference form was *двести* 'two hundred' (nominative/accusative case). In some cases there was an error in the reference data, as with *21*, where both the "golden" form and the neural model had *двадцати* 'twenty', whereas the CG correctly forced it to *двадцати одной* 'twenty one'. In a handful of cases, decoding with a CG produced a digit sequence, as in *800* appearing as *восемь ноль ноль*. As with English, the CG allows both the digit sequence and the cardinal reading, but the neural model, offered the choice, picks the digit sequence: left to its own devices, it predicted *sil*.

The three DATE errors were all morphological variants of the correct form, as were all the DECIMAL and MEASURE errors. In general with the MEASURE cases, the neural model alone produced forms that were completely wrong, often *sil*, or as in cases like *309м3*, an incomplete reading *триста девять кубических* 'three hundred nine cubic', which the CG corrected to *триста девять кубических метра* 'three hundred nine cubic meters'.

In a similar way, all MONEY CG errors involved morphological case differences. Thus *£4* is verbalized as *четырьмя фунтов стерлингов* 'four pounds sterling' (instrumental case on the number) rather than *четырех фунтов стерлингов* (genitive); the neural net produced *четырьмя четырьмя* ('four four') on its own.

Finally one of the two ordinal errors involved a case difference; the other *−10* was a misclassified example read correctly (modulo morphological form) as *минус десяти* 'minus ten' with the CG, where the reference was *десятом* 'tenth'.

7.8.3 Summary. As we have seen in the preceding analysis, the vast majority of cases where the CG produces a different result from the neural model alone are an improvement. Either the error is completely corrected, or the "error" actually corrects an error in the gold standard, or the form is at least a reasonable alternative form for the given input, rather than an unrecoverable error. We can therefore conclude that the CGs are working as intended.

8. Discussion

We have presented a variety of neural architectures for text normalization intended for speech applications. We have shown what we have called the *contextual model* with word or word-feature context outperforms, in both speed and accuracy, a number of other systems including a baseline architecture introduced by Sproat and Jaitly (2017). We also find that a coarse-to-fine model, which first segments and tags the input, then applies verbalization to non-trivial cases, is viable.

As we noted in the Introduction, there has been increased research interest in this task due to our release of the data and an associated Kaggle competition, and a few recent papers apply neural network methods to this data set. We already discussed that the work of Yolchuyeva, Németh, and Gyire-Tóth (2018) uses convolutional networks (LeCun and Bengio 1995), and Pramanik and Hussain (2018) use differentiable neural computers (Graves et al. 2016). Although the results in these papers do show some improvement over the results reported in Sproat and Jaitly (2016, 2017)—at least for some semiotic classes—they do not seem to offer substantial gains compared to the results reported here. Furthermore, the system of Pramanik and Hussain (2018) continues to make unrecoverable errors, though they do claim that their solution fails to make such errors in DATE examples like *11/10/2008*; and while Yolchuyeva, Németh, and Gyires-Tóth (2018) do not report whether their system makes these kinds of errors, given that their overall per-class errors do not differ substantially from our own, it would be surprising if such errors did not occur.

Therefore, it seems reasonable to conjecture that one is not going to eliminate the problem of unrecoverable errors by simply choosing a different model architecture. Rather, neural network solutions in general tend to make unrecoverable errors, and for these we have argued that using trainable finite-state covering grammars is a reasonable approach, but we continue to look for ways to improve covering grammar training and coverage. [18]

At the same time, we are currently exploring whether other neural approaches can help mitigate against unrecoverable errors. One approach that seems plausible is generative adversarial networks (Goodfellow et al. 2014; Goodfellow 2016), which have achieved impressive results in vision-related tasks but which have been also applied to NLP tasks including machine translation (Wu et al. 2017; Yang et al. 2018).

Given the great success of deep learning for many problems, it is tempting to simply accrete speech and language tasks to a general class of problems and to worry less about the underlying problem being solved. For example, at a certain level of abstraction, all of text-to-speech synthesis can be thought of as a sequence-to-sequence problem where the input sequence is a string of characters and the output sequence is some representation of a waveform. "End-to-end" TTS models such as Char2Wav (Sotelo et al. 2017) treat the problem in this way, with no attempt to consider the many subproblems involved in this transduction. As we argued earlier, our work suggests that such approaches are unlikely to provide a solution to issues like unrecoverable errors in text normalization. More generally, our work suggests that domain-specific knowledge is still useful in a deep learning paradigm. Text normalization may seem at first to be an "easy" problem, because it is not difficult to achieve high overall label accuracy, but a great deal of further effort is necessary to prevent unrecoverable errors. This is not to say that a pure neural network approach to text normalization, or to TTS in general, is impossible, but it does suggest that one should continue to pay close attention to the linguistic details of the underlying problems.

Appendices

A.1 Thrax Covering Grammar for English Years (`en_year.grm`)

For more information on Thrax and its syntax, see `http://thrax.opengrm.org`.

18 We also utilize a *whitelisting* mechanism that can impose a hard override for a particular case if the system gets something wrong. Overrides can either be based on simple string matching, or else can dispatch the input to a WFST grammar.

```
# Year readings for English.
import 'byte.grm' as b;
import 'number.grm' as n;

func I[expr] {
  return "" : expr;
}

cardinal = n.CARDINAL_NUMBER_NAME;

d = b.kDigit;
D = d - "0";
digits = d{1,4};
cardinals = cardinal " " cardinal;

# Grouping into pairs (d?d)(dd): 19 24.
pairwise = Optimize[(d{1,2} I[" "] d{2}) @ cardinals];

# Reading for (d?d)00.
hundreds = Optimize[((d? D) @ cardinal) ("00" : " hundred")];

# Reading for (d?D)0d.
o = Optimize[((d? D) @ cardinal) ("0" : " o ") (d @ cardinal)];

# Reading for d0dd.
thousand = (d "0" d d) @ cardinal;

sigstar = Optimize[b.kBytes*];

# First try digits @ hundreds, and if that fails pass through digits.
# Then try digits @ pairwise, and if that fails pass through digits.
# Then try digits @ o, and if that fails pass through digits.
# Then try digits @ cardinals, which shall surely work.
# Oh, and then make it a disjunction with thousand to allow both
# "twenty ten" and "two thousand ten" readings.
export YEAR =
 Optimize[
  LenientlyCompose[
    LenientlyCompose[
      LenientlyCompose[
        LenientlyCompose[digits, hundreds, sigstar],
        pairwise, sigstar],
      o, sigstar],
    cardinal, sigstar] |
    thousand];
```

A.2 Transformer Model Details

We utilize a Transformer sequence-to-sequence model (Vaswani et al. 2017), using the architecture described in Appendix A.2 of Chen et al. (2018), with:

- 6 Transformer layers for both the encoder and the decoder,
- 8 attention heads,
- a model dimension of 512, and
- a hidden dimension of 2,048.

Table A.1
Default parameters for the sliding window model.

Input embedding size	256
Output embedding size	512
Number of encoder layers	1
Number of decoder layers	1
Number of encoder units	256
Number of decoder units	512
Attention mechanism size	256

Dropout probabilities are uniformly set to 0.1. We use a dictionary of 32k word pieces (Schuster and Nakajima 2012) covering both input and output vocabularies and employ the Adam optimizer. All other hyperparameter settings are borrowed from Chen et al. (2018).

A.3 Sliding Window Model Details

The hyper-parameters are summarized in Table A.1. The input vocabulary size is 254 for English and 279 for Russian, including the most frequent input characters in the training data and <norm>, </norm>, <s>, </s>, and <unk>. The output vocabulary is 1,002 for English and 2,005 for Russian, including the most frequent output words in the training data and <self>, sil.

A.4 Contextual Model Details

The hyper-parameters are summarized in Table A.2. The sequence-to-sequence sub-model shares the input and output vocabularies with the sliding window baseline. When the context is character-based, the context vocabulary is the same as the input vocabulary in the sliding window baseline, i.e., 254 for English and 279 for Russian. When the context is word piece-based, the context vocabulary is 5,457 for English and 5,489 for Russian. For the word feature models, we set the bucket size to 5,000 for hashing character trigrams. The context vocabulary is 5,000 for word feature contextual models.

Table A.2
Default parameters for the contextual model.

Context embedding size	256
Number of tagging output embedding size	64
Number of context encoder layers	1
Number of tagging decoder layers	1
Number of context encoder units	256
Number of tagging decoder units	64
Seq2seq input embedding size	256
Seq2seq output embedding size	512
Number of seq2seq encoder layers	1
Number of seq2seq decoder layers	1
Number of seq2seq encoder units	256
Number of seq2seq decoder units	512
Attention mechanism size	256

Acknowledgments

The authors wish to thank Navdeep Jaitly for his collaboration in the early stages of this project. We thank Michael Riley and colleagues at DeepMind for much discussion as this work evolved. We acknowledge audiences at Johns Hopkins University, the City University of New York, Gothenburg University, and Chalmers University for comments and feedback on presentations of this work. Alexander Gutkin assisted with the initial data preparation. The initial tokenization phase of our covering grammars for measure expressions was augmented with grammars developed by Mark Epstein for information extraction. Finally, Shankar Kumar provided extensive help with the transformer models including training reported in Section 4.

References

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

Allen, Jonathan, Sharon M. Hunnicutt, and Dennis Klatt. 1987. *From Text to Speech: The MITalk System*, Cambridge University Press, Cambridge.

Arthur, Philip, Graham Neubig, and Satoshi Nakamura. 2016. Incorporating discrete translation lexicons into neural machine translation. In *EMNLP*, pages 1557–1567, Austin, TX.

Arik, Sercan, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, Shubho Sengupta, and Mohammad Shoeybi. 2017. Deep voice: Real-time neural text-to-speech. ArXiv:1702.07825.

Aw, Ai Ti and Lian Hau Lee. 2012. Personalized normalization for a multilingual chat system. In *ACL*, pages 31–36, Jeju Island.

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

Beaufort, Richard, 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, Uppsala.

Chan, William, 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, Shanghai.

Chen, Mia Xu, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. 2018. The best of both worlds: Combining recent advances in neural machine translation. *CoRR*, abs/1804.09849.

Chiu, Chung-Cheng, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Ekaterina Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. 2017. State-of-the-art speech recognition with sequence-to-sequence models. ArXiv:1712.01769.

Cho, Kyunghyun, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, Doha.

Choudhury, Monojit, Rahul Saraf, Vijit Jain, Sudesha Sarkar, and Anupam Basu. 2007. Investigation and modeling of the structure of texting language. *International Journal of Document Analysis and Recognition*, 10:157–174.

Chrupala, Grzegorz. 2014. Normalizing tweets with edit scripts and recurrent neural embeddings. In *ACL*, pages 680–686, Baltimore, MD.

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

van Esch, Daniel and Richard Sproat. 2017. An expanded taxonomy of semiotic classes for text normalization. In *INTERSPEECH*, pages 4016–4020, Stockholm.

Gillick, Larry and Stephen J. Cox. 1989. Some statistical issues in the comparison of speech recognition algorithms. In *ICASSP*, pages 1520–6149, Glasgow.

Goodfellow, Ian. 2016. NIPS 2016 tutorial: Generative adversarial networks. ArXiv:1701.00160.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. In *NIPS*, pages 2672–2680, Montreal.

Gorman, Kyle and Richard Sproat. 2016. Minimally supervised models for number normalization. *Transactions of the Association for Computational Linguistics*, 4:507–519.

Graves, Alex, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476.

Hassan, Hany and Arul Menezes. 2013. Social text normalization using contextual graph random walks. In *ACL*, pages 1577–1586.

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

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

Kobus, Catherine, François Yvon, and Géraldine Damnati. 2008. Normalizing SMS: Are two metaphors better than one? In *COLING*, pages 441–448, Manchester.

LeCun, Yann and Yoshua Bengio. 1995. Convolutional networks for images, speech, and time-series, In Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, Cambridge, 255–258.

Liu, Fei, Fuliang Weng, and Xiao Jiang. 2012. A broad-coverage normalization system for social media language. In *ACL*, pages 1035–1044, Jeju Island.

Liu, Fei, Fuliang Weng, Bingqing Wang, and Yang Liu. 2011. Insertion, deletion, or substitution? Normalizing text messages without pre-categorization nor supervision. In *ACL*, pages 71–76, Portland, OR.

Liu, Xiaohua, Ming Zhou, Xiangyang Zhou, Zhongyang Fu, and Furu Wei. 2012. Joint inference of named entity recognition and normalization for tweets. In *ACL*, pages 526–535, Jeju Island.

Mi, Haitao, Baskaran Sankaran, Zhiguo Wang, and Abe Ittycheriah. 2016. Coverage embedding models for neural machine translation. In *EMNLP*, pages 955–960, Austin, TX.

Min, Wookhee and Bradford Mott. 2015. NCSU_SAS_WOOKHEE: A deep contextual long-short term memory model for text normalization. In *WNUT*, pages 111–119, Beijing.

Munteanu, Cosmin, Ronald Baecker, Gerald Penn, Elaine Toms, and David James. 2006. The effect of speech recognition accuracy rates on the usefulness and usability of webcast archives. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 493–502.

Ng, Axel H., Kyle Gorman, and Richard Sproat. 2017. Minimally supervised written-to-spoken text normalization. In *ASRU*, pages 665–670, Okinawa.

Pennell, Deana and Yang Liu. 2011. A character-level machine translation approach for normalization of SMS abbreviations. In *IJCNLP*, pages 974–982, Chiang Mai.

Pramanik, Subhojeet and Aman Hussain. 2018. Text normalization using memory augmented neural networks. ArXiv:1806.00044.

Pusateri, Ernest, Bharat Ram Ambati, Elizabeth Brooks, Ondrej Platek, Donald McAllaster, and Venki Nagesha. 2017. A mostly data-driven approach to inverse text normalization. In *INTERSPEECH*, pages 2784–2788, Stockholm.

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

Roark, Brian, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *ACL*, 61–66, Jeju Island.

Sak, Haşim, Françoise Beaufays, Kaisuke Nakajima, and Cyril Allauzen. 2013. Language model verbalization for automatic speech recognition. In *ICASSP*, pages 8262–8266, Vancouver.

Schuster, Michael and Kaisuke Nakajima. 2012. Japanese and Korean voice search. In *ICASSP*, pages 5149–5152, Kyoto.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *ACL*, pages 1715–1725, Berlin.

Shugrina, Masha. 2010. Formatting time-aligned ASR transcripts for readability. In *NAACL*, pages 198–206, Los Angeles, CA.

Sotelo, Jose, Soroush Mehri, Kundan Kumar, João Felipe Santos, Kyle Kastner, Aaron Courville, and Yoshua Bengio. 2017. Char2wav: End-to-end speech synthesis. In *ICLR*, Toulon.

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

Sproat, Richard, editor . 1997. *Multilingual Text-to-Speech Synthesis: The Bell Labs Approach*, Kluwer Academic Publishers, Boston.

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

Sproat, Richard and Kyle Gorman. 2018. A brief summary of the Kaggle text normalization challenge. http://blog.kaggle.com/2018/02/07/a-brief-summary-of-the-kaggle-text-normalization-challenge/.

Sproat, Richard and Keith Hall. 2014. Applications of maximum entropy rankers to problems in spoken language processing. In *INTERSPEECH*, pages 761–764, Singapore.

Sproat, Richard and Navdeep Jaitly. 2016. RNN approaches to text normalization: A challenge. ArXiv:1611.00068.

Sproat, Richard and Navdeep Jaitly. 2017. An RNN model of text normalization. In *INTERSPEECH*, pages 754–758, Stockholm.

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

Tu, Zhaopeng, Yang Liu, Lifeng Shang, Xiaohua Liu, and Hang Li. 2017. Neural machine translation with reconstruction. In *AAAI*, pages 3097–3103.

Tu, Zhaopeng, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. 2016. Modeling coverage for neural machine translation. In *ACL*, pages 76–85, San Francisco, CA.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR*, abs/1706.03762.

Wang, Chunqi and Bo Xu. 2017. Convolutional neural network with word embeddings for Chinese word segmentation. In *IJCNLP*, pages 163–172, Taipei.

Wu, Lijun, Yingce Xia, Zhao Li, Fei Tian, Tao Qin, Jianhuang Lai, and Liu Tie-Yan. 2017. Adversarial neural machine translation. ArXiv:1704.06933.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. ArXiv:1609.08144.

Xia, Yunqing, Kam-Fai Wong, and Wenjie Li. 2006. A phonetic-based approach to Chinese chat text normalization. In *ACL*, pages 993–1000, Sydney.

Xie, Ziang. 2017. Neural text generation: A practical guide. ArXiv:1711.09534.

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

Yang, Zhen, Wei Chen, Feng Wang, and Bo Xu. 2018. Improving neural machine translation with conditional sequence generative adversarial nets. ArXiv:1703.04887.

Yolchuyeva, Sevinj, Géza Németh, and Bálint Gyires-Tóth. 2018. Text normalization with convolutional neural networks. *International Journal of Speech Technology*, 21:1–12.