Type, Then Correct: Intelligent Text Correction Techniques for Mobile Text Entry Using Neural Networks

Mingrui “Ray” Zhang
The Information School
DUB Group
University of Washington
Seattle, WA, USA 98195
mingrui@uw.edu

He Wen
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
hew1@andrew.cmu.edu

Jacob O. Wobbrock
The Information School
DUB Group
University of Washington
Seattle, WA, USA 98195
wobbrock@uw.edu

ABSTRACT

Current text correction processes on mobile touch devices are laborious: users either extensively use backspace, or navigate the cursor to the error position, make a correction, and navigate back, usually by employing multiple taps or drags over small targets. In this paper, we present three novel text correction techniques to improve the correction process: *Drag-n-Drop, Drag-n-Throw*, and *Magic Key*. All of the techniques skip error-deletion and cursor-positioning procedures, and instead allow the user to type the correction first, and then *apply* that correction to a previously committed error. Specifically, *Drag-n-Drop* allows a user to drag a correction and drop it on the error position. *Drag-n-Throw* lets a user drag a correction from the keyboard suggestion list and “throw” it to the approximate area of the error text, with a neural network determining the most likely error in that area. *Magic Key* allows a user to type a correction and tap a designated key to highlight possible error candidates, which are also determined by a neural network. The user can navigate among these candidates by directionally dragging from atop the key, and can apply the correction by simply tapping the key. We evaluated these techniques in both text correction and text composition tasks. Our results show that correction with the new techniques was faster than *de facto* cursor and backspace-based correction. Our techniques apply to any touch-based text entry method.

CCS Concepts
• **Human-centered computing~Text input**
• Human-centered computing~Touch screens
• Computing methodologies~Natural language processing
• Computing methodologies~Neural networks

Keywords
Text editing; touch; gestures; natural language processing.

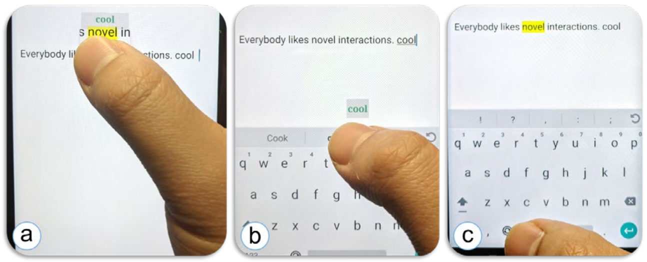

cool
novel in
Everybody likes novel interactions. cool
cool
Cook
Everybody likes novel interactions. cool
a
b
c

Figure 1. Our three correction techniques: (a) *Drag-n-Drop* lets the user drag the last word typed and drop it on an erroneous word or gap between words; (b) *Drag-n-Throw* lets the user drag a word from the suggestion list and flick it into the general area of the erroneous word; (c) *Magic Key* highlights each possible error word after the user types a correction. Directional dragging from atop the magic key navigates among error words, and tapping the magic key applies the correction.

INTRODUCTION

Text entry techniques on touch-based mobile devices today are generally well developed. Ranging from tap-based keyboard typing to swipe-based gesture typing [44], today’s mobile text entry methods employ a range of sophisticated algorithms designed to maximize speed and accuracy. Although the results reported from various papers [32,38] show that mobile text entry can reach fairly high speeds, some even as fast as desktop keyboards [38], the daily experience of mobile text composition is still often lacking. One bottleneck lies in the text *correction* process. On mobile touch-based devices, text correction can involve repetitive backspacing and moving the text cursor with repeated taps and drags over very small targets (*i.e.*, the characters and gaps between them). Owing to the fat finger problem [39], this process can be slow and tedious indeed.

Correcting text is a consistent and vital activity during text entry. A study by MacKenzie and Soukoreff showed that backspace was the second most-common keystroke during text entry [24] (pp. 164-165). Dhakal *et al.* [8] found that during typing, people made 2.29 error corrections per sentence, and that slow typists made and corrected more mistakes than the fast typists.

For immediate error corrections, *i.e.*, when an error is noticed right after it is made, the user can backspace to remove the error [36]. However, for overlooked errors, the current text correction process based on text cursor movement on smartphones is laborious: one must navigate the cursor to the error position, backspace the error text, re-enter the correct text, and finally navigate the cursor back. There are three ways to position the cursor: (1) by repeatedly pressing the backspace key [36]; (2) by pressing arrow keys on some keyboards or making gestures such as swipe-left; (3) by using direct touch to move the cursor. The first two solutions are more precise than the last one, which suffers from the fat finger problem [39], but they require repetitive action. The third option is error prone when positioning the cursor amidst small characters, which increases the possibility of cascading errors [2]; it also increases the cognitive load of the task, and takes on average 4.5 seconds to perform the tedious position-edit-reposition sequence [13].

Our three new interaction techniques are based on the following hypothetical: What if we can skip positioning the cursor and deleting errors? Given that the *de facto* method of correcting errors relies heavily on these actions, such a question is subtly quite challenging. *What if we just type the correction text, and apply it to the error?*

Inspired by this possibility, we developed three text correction techniques. The first technique, *Drag-n-Drop* (Figure 1a), is a baseline technique that allows users to drag the last-typed word and drop it on the erroneous text to correct substitution and omission errors [40].

The second technique, *Drag-n-Throw* (Figure 1b), is the "intelligent" version of *Drag-n-Drop*: it allows the user to flick a word from the keyboard's suggestion list towards the approximate area of the erroneous text. A neural network finds the most likely error within the general target area based on the thrown correction, and automatically corrects it. *Drag-n-Throw* is faster than *Drag-n-Drop,* because the user does not need to drop the correction on the error location.

Unlike the above two techniques, the third technique, *Magic Key* (Figure 1c), does not require direct interaction with the text input area at all. After typing a correction, users press a dedicated *Magic Key* on the keyboard, and a neural network is used to highlight possible error words given the typed correction. The user can then step through the possible error words by directionally dragging from atop the magic key. When the desired error word is reached, users simply tap the magic key to apply their correction.

All three of our interaction techniques require no movement of the text cursor and no use of backspace. To further bolster user confidence, we also offer an undo key on our keyboard.

To understand users' reactions to our techniques, we evaluated them in two text entry tasks: (1) a text correction task, and (2) a text composition task. We compared our three techniques with the *de facto* cursor-movement and backspace-based method in use on smartphones today. Our results reveal that our two "intelligent" techniques, *Drag-n-Throw* and *Magic Key*, result in faster correction times, and were preferred over the *de facto* technique. Moreover, our methods do not conflict with existing gestural interactions, including gesture typing [44], and therefore can be used on any touch-based device including smartphones and tablets.

RELATED WORK

Below, we first review research related to text entry correction behaviors on touch screens. Then, we then present current text correction techniques for mobile text entry. Finally, we provide a short introduction to natural language processing (NLP) algorithms for text correction.

Text Correction Behaviors on Touch Screens

Researchers have found that typing errors are common using touch-based keyboards, and that current text correction techniques are left wanting in many ways. For example, sending error-ridden messages caused by typos and auto-correction [19] is of greatest concern when it comes to older adults. Moreover, Komninos *et al.* [18] observed and recorded in-the-wild text entry behaviors on Android phones, finding that users made around two word-level errors per typing session, which slowed text entry considerably. Also, participants "predominantly employed backspacing as an error correction strategy." Based on their observations, Komninos *et al.* recommended that future research needed to "develop better ways for managing correction," which is the very focus of this work.

In most character-level text entry schemes, there are three types of text entry errors [25,40]: *substitutions*, where the user enters different characters than intended; *omissions*, where the user fails to enter characters; and *insertions*, where the user injects erroneous characters. Substitutions are often the most frequent errors of these types. In a text entry study on smartwatches [20], out of 888 phrases, participants made 179 substitution errors, 31 omission errors, and 15 insertion errors. In a big data study of keyboarding [8], substitution errors (1.65%) were observed more frequently than omission (0.80%) and insertion (0.67%) errors. Our correction techniques address substitution and omission errors; we do not address insertion errors because users can just delete insertions without typing any corrections. Moreover, insertion errors are relatively rare.

Mobile Text Correction Techniques

While much previous work focused on user behaviors during mobile text entry, there have been a few projects that improved the text correction process. The *smart-restorable backspace* [1] project had the goal most similar to that of our work—to improve text correction without extensive backspacing and cursor-positioning. The technique allowed users to perform a swipe gesture on the backspace key to delete the typed text back to the position of an error, and restore that text by swiping again on the backspace key after correcting the error. To determine error positions, the technique used Eugene Myers' algorithm [26] to compare the edit distance of the text and the word in a dictionary. The error detection algorithm is the main limitation of that work: it only detects misspellings. It cannot detect grammar errors or word misuse. By contrast, our algorithm, which combines neural networks and string edit distance, can detect a wide range of errors. (We describe our algorithm in detail, below.)

Commercial products exhibit a variety of text correction techniques. *Gboard*[1] allows a user to touch on a word and replace it by tapping on another word in a suggestion list. However, this technique is only limited to misspellings. The *Grammarly* keyboard[2] keeps track of the inputted text, and provides corrections in the suggestion list. Like our work, Grammarly uses natural language processing algorithms to provide correction suggestions. In this way, it is able to detect both spelling and grammar errors. However, because Grammarly provides correction suggestions without guidance (*e.g.*, it provides all possible error correction options without knowing which one the user wants to correct), the suggestion bar can become cluttered in the presence of many suggestions. Finally, some keyboards such as the Apple iOS 9 keyboard support indirect cursor control by treating the keyboard as a trackpad. Unfortunately, prior research [34] has shown that this design brings no time or accuracy benefits compared to direct pointing.

Different from the above techniques, our three correction techniques have the user enter a correction first, typed at the end of the current text input stream. Informed by the correction, our techniques can better understand what text the user wants to correct. Specifically, *Drag-n-Throw* and *Magic Key* utilize neural networks to detect possible error candidates based on typed corrections. Thus, they not only correct mistakes like misspellings or grammar errors, but also can offer to substitute synonyms or improved phrasings.

NLP Algorithms for Error Correction

Drag-n-Throw and *Magic Key* use neural networks from natural language processing (NLP) to find possible errors based on typed corrections. We therefore provide a brief introduction to related techniques.

Traditional error correction algorithms utilize *N*-grams and string edit distances to provide correction suggestions. For example, Islam and Inkpen [16] presented an algorithm that uses the Google 1T 3-gram dataset and a string-matching algorithm to detect and correct spelling errors. For each word in the original string, they first search for candidates in the dictionary, and assign each possible candidate a score derived from their frequency in the *N*-gram dataset and the string matching algorithm. The candidate with the highest score above a threshold is suggested as a correction.

Recently, deep neural networks have gained popularity in NLP research because of their generalizability and significantly better performance than traditional algorithms. For NLP tasks, convolutional neural networks (CNN) and recurrent neural networks (RNN) are extensively used, and they often follow a structure called *encoder-decoder*, where part of the model encodes the input text into a feature vector, then decodes the vector into the result. In this work, we do not invent novel neural network algorithms; rather, we utilize an RNN in this known encoder-decoder pattern. To the best of our knowledge, we are the first to employ an RNN in this way to support interactive text correction. Readers interested in neural networks for NLP are directed elsewhere [3,37,42].

Most deep learning NLP research treats the error correction task as a language translation task: for error correction, the input is a sentence with errors, and the output is an error-free sentence; for language translation, the input is a sentence in one language, and the output is a sentence in another language. For example, Xie *et al.* [41] presented an encoder-decoder RNN correction model that operates input and output at the character level. Their model was built upon a sequence-to-sequence model for translation [3], which was also used in our algorithm for error detection.

As stated, current deep learning models for text correction accept a sentence as input and produce its error free version as output. The difference for our model is that along with the input sentence, we must somehow provide the model with the correction word typed by user. In addition, our model does not need to output the whole corrected sentence because we already know what the correction is. Instead, it can just output a *position* indicating where the error is. Then we can apply the correction to the error position. In the next section, we describe our three correction techniques in more detail.

OUR THREE TEXT CORRECTION TECHNIQUES

We present the design and implementation of three new interaction techniques for text correction on touch-based devices like smartphones and tablets. The common features of these interactions are: (1) the first step is always to type the correction text at the current cursor position, which is usually at the end of the input stream; (2) all correction interactions can be undone by tapping the undo key (Figure 2, top right); (3) after a correction is applied, the text cursor remains at the last character of the text input stream, allowing the user to continue typing without having to move the cursor. A current, but not theoretical, limitation is that we only allow the correction text to be contiguous alphanumeric text without special characters or spaces.

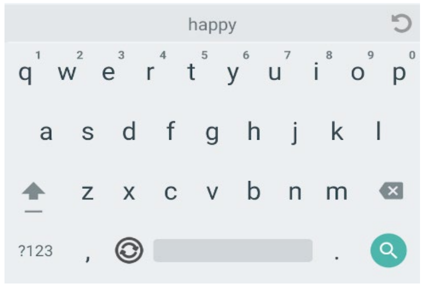


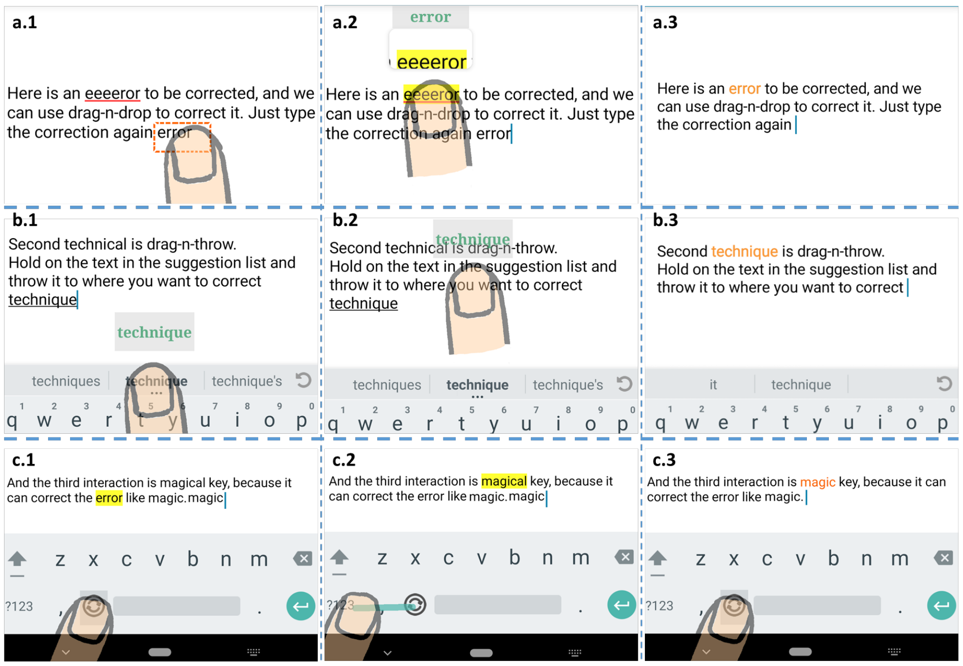
Figure 2. Our customized keyboard interface. The undo key is located in the top-right corner. The *Magic Key* is the circular key immediately to the left of the space bar.

Drag-n-Drop

Drag-n-Drop is our baseline interaction technique. With *Drag-n-Drop*, after typing the correction, the user then drags the correction text and drops it on the error location, either another word or a gap between words. As shown in

[1] https://en.wikipedia.org/wiki/Gboard

[2] https://en.wikipedia.org/wiki/Grammarly



Here is an eeeeror to be corrected, and we can use drag-n-drop to correct it. Just type the correction again error

error

eeeeror

Here is an eeeeror to be corrected, and we can use drag-n-drop to correct it. Just type the correction again error

Here is an error to be corrected, and we can use drag-n-drop to correct it. Just type the correction again

Second technical is drag-n-throw.
Hold on the text in the suggestion list and throw it to where you want to correct
technique

technique

techniques | technique | technique's

Second technical is drag-n-throw.
Hold on the text in the suggestion list and throw it to where you want to correct
technique

technique

techniques | technique | technique's

Second technique is drag-n-throw.
Hold on the text in the suggestion list and throw it to where you want to correct

it | technique

And the third interaction is magical key, because it can correct the error like magic.magic

And the third interaction is magical key, because it can correct the error like magic.magic

And the third interaction is magic key, because it can correct the error like magic.

Figure 3. Our three interaction techniques. *Drag-n-Drop*: (a.1) Type a word and then touch it to initiate correction; (a.2) Drag the correction to the error position. Touched words are highlighted and magnified, and the correction shows above the magnifier; (a.3) Drop the correction on the error to finish. *Drag-n-Throw*: (b.1) Dwell on a word from the suggestion list to initiate correction. The word will display above the finger; (b.2) Flick the finger towards the area of the error: here, the flick ended on "the", not the error text "technical"; (b.3) The algorithm determines the error successfully, and confirming animation appears. *Magic Key:* (c.1) Tap the magic key (the circular button) to trigger correction. Here, "error" is shown as the nearest potential error. (c.2) Drag left from atop the magic key to highlight the next possible error in that direction. Now, "magical" is highlighted. (c.3) Tap the magic key again to commit the correction "magic".

Figure 3a.1, if the finger's touchdown point is within the area of the last word, the correction procedure will be initiated. The user can then move the correction and drop it either on another word to substitute it, or on a space to insert it.

While moving the correction, a magnifier appears above the finger to provide an enlarged image of the touched text, which is highlighted in yellow (Figure 3a.2). When the finger drags over an alphanumeric character, the highlight extends to its surrounding text bounded by special characters or spaces. When the finger drags over a space, only the space is highlighted. While dragging, the correction itself shows above the magnifier to remind the user of what it is.

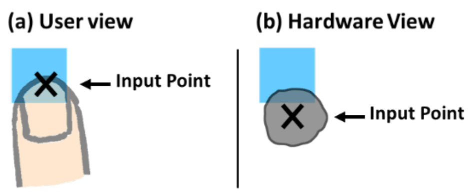


Figure 4. Perceived input point: (a) the user views the top of the fingernail as the input point [15]; (b) but today's hardware regards the center of the contact area as the touch input point, which is not the same. Figure adapted from [39].

Similar to *Shift* by Vogel and Baudisch [39], we adjusted the input point to be 30 px above the finger's contact point to reflect the user's perceived input point [15] (Figure 4). As Benko *et al.* observed, and Vogel and Baudisch reiterated, "*users perceive the selection point of their finger as being located near the top of the finger tip*" [4,39], while the actual touch point was at the center of the finger contact area [35].

After the correction is dropped on a space (for insertion) or on a word (for substitution), there is an animated color change from orange to black confirming the successful application of the correction text.

Drag-n-Throw

Similar to *Drag-n-Drop*, *Drag-n-Throw* also requires the user to drag the correction. But unlike *Drag-n-Drop*, with *Drag-n-Throw*, the user flicks the correction from the word suggestion list atop the keyboard, not from the text area, allowing the user's fingers to stay near the keyboard area. As before, the correction text shows above the touch point as a reminder (Figure 3b.1). Instead of dropping the correction on the error position, the user flicks the correction to the general area of the text to be corrected. Once the correction is thrown, a neural network determines the error position, and corrects the error either by substituting the correction for a word, or by inserting the correction on a space. Color animation is displayed to confirm the correction. The procedure is shown in Figures 3b.1-3.

We enable the user to drag the correction from the suggestion list because it is quicker and more accurate than directly interacting with the text, which has smaller targets. Moreover, our approach provides more options and saves time because of word completion. For example, if the user wants to type "dictionary", she can just type "dic" and "dictionary" appears in the list. Or, if the user misspells "dictonary", omitting an "i", the correct word still appears in the list because of the keyboard's decoding algorithm.

Flicking the text speeds up the interaction beyond precise drag-and-dropping. The user does not have to carefully move the finger to drop the correction. Our neural network will output positions for possible substitutions or insertions within the general finger-up area. In our implementation, if the candidate is within 250 px of the finger-lift point in any direction, the error will be corrected and confirmed by color animation. Otherwise, there will be no effect. The 250 px threshold was derived empirically from iterative trial-and-error. Larger thresholds allow corrections to occur too distant from the finger-lift point, which can cause unexpected results. Smaller thresholds reduce the benefits of flicking and eventually start to feel like dragging-and-dropping.

Magic Key

Drag-n-Drop required interaction within the text input area, while *Drag-n-Throw* kept the fingers closer to the keyboard but still required some interaction in the text input area. With *Magic Key*, the progression "inward" toward the keyboard is fulfilled, as the fingers do not interact with the text input area at all, never leaving the keyboard. Thus, round-trips [10] between the keyboard and text input area are eliminated.

With *Magic Key*, after typing the correction, the user taps the magic key on the keyboard (Figure 3c.1), and the nearest possible error text is highlighted. If a space is highlighted, an insertion is suggested; if a word is highlighted, a substitution is suggested. The nearest possible error to the just-typed correction will be highlighted first; if it is not the desired correction, the user can directionally drag from atop the magic key to show the next possible error. The user can drag left or right from atop the magic key to rapidly navigate among different error candidates. Finally, the user can tap the magic key to commit the correction. The procedure is shown in Figures 3c.1-3. To cancel the operation, the user can simply tap any key (other than undo or the magic key itself).

THE CORRECTION ALGORITHM

In this section, we present our neural network algorithm for text correction and its natural language processing (NLP) model, our data collection and processing procedures, and our training process and validation results. We first list error types that our model can correct.

Error Types for Correction

Typos. A typographical error ("typo") happens when characters of a word are mistyped. For example, "fliwer" ("flower") or "feetball" ("football"). Among typos, misspellings can usually be auto-corrected by current keyboards; however, auto-correction might yield another wrong word. For example, "best" ("bear") or "right" ("tight"). Our model handles different types of typo errors.

Grammar Errors. Our system can correct grammar errors caused by one mistaken word, such as misuse of verb tense, lack of articles or pronouns, subject-verb disagreement, etc.

Semantic Substitution. Our model can also substitute words that are semantically related to the correction, such as synonyms and antonyms. For example, "what a nice day" can be corrected to "what a beautiful day". Semantic substitution is not necessarily correcting an error, but is useful when the user wants to change an expression.

The Deep Neural Network Structure

Inspired by Xie *et al.* [41], our *Drag-n-Throw* and *Magic Key* interaction techniques utilize a recurrent neural network (RNN) encoder-decoder model similar to those used in language translation. The encoder contains a character-level convolutional neural network (CNN) [17] and two bi-directional gated recurrent unit (GRU) layers [7]. The decoder contains a word-embedding layer and two GRU layers. The overall architecture of the model is shown in Figure 5, and the encoder-decoder is shown in Figure 6.[3]

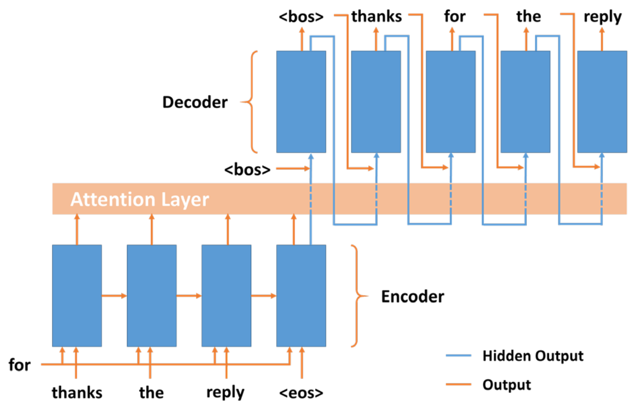


Figure 5. The encoder-decoder model for text correction. The model outputs five words in which the middle word is the correction. In this way, we get the correction's location.

Traditional RNNs cannot output position information. Our key insight is that instead of outputting the whole error-free sentence, we make the decoder output only five words in which the correction word is the middle one. We use the surrounding words to locate the correction position. To locate the correction position, we compare the output with the input sentence word-by-word, and choose the position that aligns with most words. If there are not enough words, the decoder will output the flags *<bos>* or *<eos>* for beginning-of-sentence and end-of-sentence, respectively. For the example in Figure 5, we first tokenize the input and add two *<bos>* and two *<eos>* flags to the start and end of the tokens. Then we compare the output with the input:

```
Input: <bos> <bos> thanks the reply <eos> <eos>
CS:          <bos> thanks for the   reply
CI:          <bos> thanks the reply
```

[3] We provide our model and data processing code at https://github.com/DrustZ/CorrectionRNN.

Above, “CS” means “compare for substitution,” which finds the best alignment for substitution; “CI” means “compare for insertion,” which finds the best alignment for insertion. In the example, CI has the best alignment of four tokens; thus, “for” will be inserted between “thanks” and “the”. If the number of aligned tokens is the same in CS and CI, we default to insertion. If there are multiple aligned positions, we choose the last position.

We now explain the details of the encoder and the decoder (Figure 6). For the encoder, because there might be typos and rare words in the input, operating on the character level is more robust and generalizable than operating on the word level. We first apply the character-level CNN [17] composed of *Character Embedding*, *Multiple Conv. Layers*, and *Max-over-time Pool* layers (Figure 6, left). Our character-level CNN generates an embedding for each word at the character level. The character embedding layer converts the characters of a word into a vector of $\boldsymbol{L} \times \boldsymbol{E_c}$ dimensions. We set $\boldsymbol{E_c}$ to 15, and fixed $\boldsymbol{L}$ to 18 in our implementation, which means the longest word can contain 18 characters (and longer words are discarded). Words with fewer than 18 characters are appended with zeroes in the input vector. We then apply multiple convolution layers on the vector. After convolution, we apply max-pooling to obtain a fixed-dimensional representation of the word ($\boldsymbol{E_w}$). In our implementation, we used convolution filters with width [1, 2, 3, 4, 5] of size [15, 30, 50, 50, 55], yielding a fixed vector with a size of 200. $\boldsymbol{E_c}$ was set to 200 in the decoder.

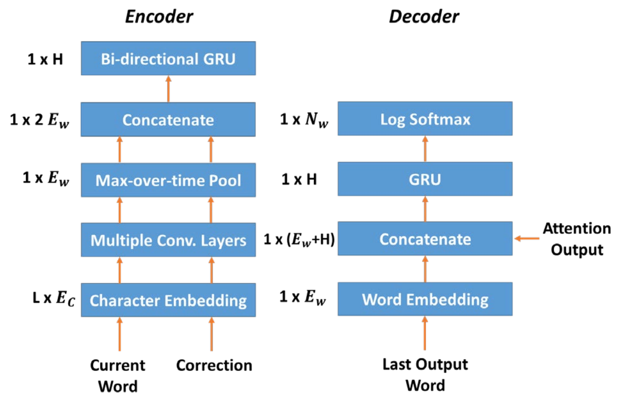


Figure 6. Illustration of the encoder and decoder, which is every vertical blue box in Figure 5. $\boldsymbol{L}$ is the length of characters in a word; $\boldsymbol{E_c}$ is the character embedding size; $\boldsymbol{H}$ is the hidden size; $\boldsymbol{E_w}$ is the word embedding size; $\boldsymbol{N_w}$ is the word dictionary size.

We also needed to provide the correction word to the encoder. We achieved this by feeding the correction into the same character-level CNN, and concatenated the correction embedding with the embedding of the current word. This yielded a vector of size $2\boldsymbol{E_w}$, which was then fed into two bi-directional GRU layers. The hidden size $\boldsymbol{H}$ of GRU was set to 300 in the encoder and decoder.

The decoder first embedded the word in a vector of size $\boldsymbol{E_w}$, which was set to 200. Then it was concatenated with the attention output. We used the same attention mechanism as Bahdanau *et al.* [3]. Two GRU layers and a log-softmax layer then followed to output the predicted word.

Data Collection and Processing

We used the *CoNLL 2014 Shared Task* [27] and its extension dataset [6] as a part of the training data. The data contained sentences from essays written by English learners with correction and error annotations. We extracted the errors that were either insertion or substitution errors. In all, we gathered over 33,000 sentences for training.

Given there was no existing dataset of mobile text entry error correction, we had to create more training data ourselves. Therefore, we artificially perturbed several large text datasets. We used Yelp reviews (containing 2,000,000 samples) and part of the Amazon reviews dataset (containing 130,000 samples) generated by Zhang *et al.* [45]. We treated these reviews as if they were error-free texts, and applied artificial perturbation to them. Specifically, we applied four perturbation methods:

1. ***Typo Simulation.*** In order to simulate a real typo, we applied the simulation method similar to Fowler *et al.* [11]. The simulation treated the touch point distribution on a QWERTY layout as a 2-D Gaussian spatial model, and selected a character based on the sampling coordinates. We used the empirical parameters of the spatial model from Zhu *et al.* [46]. For each sentence in the review dataset, we randomly chose a word from the sentence, and simulated the typing procedure for each character of the word until the simulated word was different from the original word. We then applied a spellchecker to “recover” the typo. This maneuver was to simulate the error of auto-correction functions, where the “corrected” word actually becomes a different word. We then used the “recovered” word as a typo if it was different from the original word, or used the typing simulation result if the spellchecker successfully recovered the typo.
2. ***Dropping Words.*** To enable the model to learn about insertion corrections, we randomly dropped a word from a sentence, and labelled the dropped word as the correction. We prioritized dropping common stop words first if any of them appeared in the sentence, because people were most likely to omit words like “a”, “the”, “my”, “in”, and “very”.
3. ***Word Deformation.*** We randomly changed or removed a few characters from a word. If the word was a verb, we would replace it with a word sharing the same lexeme. For example, we would pick one of “broken”, “breaking”, “breaks”, or “broke” to replace “break”. If the word was a noun, we would use a different singular or plural word. For example, we would replace “star” with “stars”. Otherwise we would just remove a few characters from the word.
4. ***Semantic Word Substitution.*** This perturbation enabled the model to learn semantic information. For a given word in the sentence, we looked for words that were semantically similar to it and made a substitution. We used

the *GloVe* [30] Twitter-100 model from Gensim[4] [31] to represent similarity. Synonyms and antonyms were generated using this method.

For each sentence in the review data set, we randomly applied a perturbation method. We then combined the perturbed data with the CoNLL data, and filtered out sentences containing less than three words or more than 20 words. The final training set contained 5.6 million phrases.

For testing, we used two datasets: *CoNLL 2013 Shared Task* [28], which was also a grammatical error-correction dataset, and the Wikipedia revision dataset [43], which contains real-word spelling errors mined from Wikipedia's revision history. We generated 1665 phrases from the CoNLL 2013 dataset, and 1595 phrases from the Wikipedia dataset.

Training Process

We implemented our model in PyTorch [29]. We included lowercase alphabetical letters ("a"-"z") and 10 numerals ("0"-"9") in the character vocabulary of the encoder. We used the Adam optimizer with a learning rate of .0001 (1e-4) for the encoder and .0005 (5e-4) for the decoder, and a batch size of 128. We applied weight clipping of [-10, +10], and a teacher forcing ratio of 0.5. We also used dropout with probability 0.2 in all GRU layers. For the word embedding layer in the decoder, we labeled words with frequencies less than two in the training set as <*unk*> for "unknown."

Results

Table 1 shows the evaluation results on the two testing datasets. The recall is 1.00 because all of our testing data contained errors. We regarded a prediction as correct if the error position predicted was correct using the comparison algorithm described above. In the original paper by Xie *et al.* [41], precision on the CoNLL dataset was 44.04%. We achieved better results because our model aimed to find only the error position. Also, note that these numbers are not the same as the accuracy of text correction during interactive use of our techniques, which was 87.9% and 97.0% for text composition with *Drag-n-Throw* and *Magic Key*, respectively.

Dataset	Accuracy
CoNLL 2013	75.68%
Wikipedia Revisions	81.88%

Table 1. The performance of our correction model on the two test data sets.

OTHER IMPLEMENTATION DETAILS

We developed a custom Android keyboard and a notebook application to implement our three text correction interaction techniques. Our keyboard was based on the Android Open Source Project (AOSP) keyboard[5] from Google. In building on top of this keyboard, we added the long-press interaction on suggested words for *Drag-n-Throw*.

The notebook application was built on an open-source project called *Notepad*[6], and most of the interactions were implemented as part of this notebook application. For *Drag-n-Drop*, when a user touched within the last word area (within 100 px of the (x, y) coordinate of the last character), the interaction was initiated. We used the default magnifier on the Android system and added a transparent view showing the correction word above the user's finger as it moves.

For *Drag-n-Drop* and *Magic Key*, the keyboard needed to communicate with the notebook application. The keyboard used the Android Broadcast mechanism to send the correction and endpoint of the flick gesture of *Drag-n-Throw*. When the information was received, the notebook searched within three lines near the release point. For each line, the notebook extracted up to 60 surrounding characters near the release point, and sent them to a server running the NLP model. The server then replied with possible corrections and corresponding probabilities. The notebook then selected the most likely option to update the correction. To avoid the correction happening too far away from the flick endpoint, we constrained the x-coordinate of the correction to be within 250 px of the finger-lift point.

For the *Magic Key* technique, the keyboard notified the notebook when the magic key was pressed or dragged. The notebook treated the last word typed as the correction, and sent the last 1000 characters to the server. The server then split the text into groups of 60 characters with overlaps of 30 characters, and predicted a correction for each group. When the notebook received the prediction results, it first highlighted the nearest option, and then switched to further error options when the key was dragged left. For substitution corrections, it would highlight the whole word to be substituted; for insertion corrections, it would highlight the space where the correction was to be inserted.

We found that string edit distance could be used to further increase the accuracy of the model for typos, since typos are often only minor alterations of text. We first calculated a match score between each token of the input text and the correction word using Levenshtein's algorithm [21]. The score equaled the number of matches divided by the total character number of the two words. If the score of a word was above 0.75, we treated the word as the error to be corrected. Otherwise, we fed the text and correction into the aforementioned neural network.

USER STUDY

To gain feedback from our participants and understand how our correction techniques perform relative to the *de facto* cursor-positioning and backspacing technique, we conducted a user study comprising both text correction and composition tasks. The correction task isolated performance time while the composition task captured the success rate of *Drag-n-Throw* and *Magic Key*, and users' subjective preferences.

[4] https://radimrehurek.com/gensim/
[5] https://bit.ly/2JAmQMJ
[6] https://github.com/farmerbb/Notepad

Participants

We recruited 20 participants (8 male, 12 female, aged 23-52) for the study. We used emails, social media and word-of-mouth for recruitment. All participants were familiar with entering and correcting text on smartphones and tablets. Study sessions lasted one hour, and participants were compensated $20 USD for their time.

Apparatus

A Google Pixel 2 XL was used for the study. The phone had a 6.0" screen with 1440×2880 resolution. We added logging functions from the notebook application to record correction time. The server running the correction model had a single GTX 1080. It handled responses via HTTP requests.

Phrases Used in the Correction Task

Both tasks utilized a within-subjects study design. For the correction task, we chose 30 phrases from the test dataset on which the correction model had been 100% correct because we wanted purely to evaluate the performance of the interaction technique, not of the predictive model. We split the phrases evenly into three categories: *typos*, *word changes*, and *insertions*. *Typos* required replacement of a few characters in a word; *word changes* required replacing a whole word in a phrase; and *insertions* required inserting a correction. We wished to see whether error positions would affect correction efficiency, so for each category, we had five *near-error* phrases where the error positions were within the last three words; we also had five *far-error* phrases where the error positions were farther away. Examples of phrases in each category are provided in the Appendix.

Correction Task Procedure

Participants were introduced to our different interaction techniques, including the categories of errors that *Drag-n-Throw* and *Magic Key* could correct. Then participants practiced the three techniques with three practice phrases each. After practicing, the 30 test phrases as well as their corresponding corrections were presented to the participants. Then they began to correct the phrases using four techniques: (1) the *de facto* cursor-positioning and backspacing method, (2) *Drag-n-Drop*, (3) *Drag-n-Throw*, and (4) *Magic Key*. Counterbalancing was achieved with a balanced Latin Square.

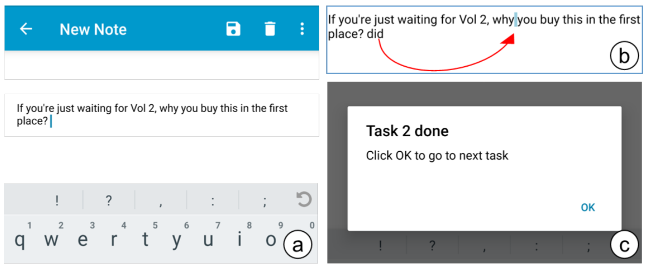

New Note
If you're just waiting for Vol 2, why you buy this in the first place?
! ? , : ;
q w e r t y u i o
1 2 3 4 5 6 7 8 9 0
a
If you're just waiting for Vol 2, why you buy this in the first place? did
b
Task 2 done
Click OK to go to next task
OK
c

Figure 7. (a) The notebook application showing the test phrase. (b) The intended correction displayed on the computer screen. (c) After each correction, a dialog box appeared.

During the correction task, the participant would be shown the next phrase to be corrected on a desktop computer screen, as well as how to correct it (Figure 7b). The notebook application would then display that phrase with its error (Figure 7a). After the participant corrected the error, a confirmation dialog box appeared (Figure 7c). Timing was calculated from when the participant pressed "OK" on the dialog box until the test phrase matched the corrected phrase. The rationale for showing the participant how to correct the test phrase was to reduce any learning effect and visual search time, thereby isolating just the interaction time.

Composition Task Procedure

After the correction task, participants began the free composition task. They were told to type whatever they liked for three minutes. Suggested examples were to write informal messages, write in a diary, or write as if having a casual conversation. The task was deliberately uncontrolled—some participants took time to think while others started typing right away. Regardless, they were told *not* to correct any errors during typing. After finishing their compositions, they corrected all errors with the four correction techniques. This composition task endeavored to evaluate usability, the "feel" of the techniques, and the correction rate of *Drag-n-Throw* and *Magic Key*. The researcher recorded when any correction failures happened in order to calculate the success rate.

When the two tasks were finished, participants filled out a NASA-TLX survey [33] and a usability survey adapted from the SUS questionnaire [5] for each interaction technique.

RESULTS

Our two study tasks were designed for different purposes, with the correction task highly controlled but artificial, attempting to isolate interaction time, and the composition task uncontrolled but more realistic, attempting to measure correction accuracy and gather user feedback. Thus, for the correction task, we focus on task completion times; for the composition task, we focus on the success rate of the two neural network-based techniques and on users' preferences.

Correction Time

Figure 8 shows correction times for the four techniques over 2400 total collected phrases. In addition to overall times, the correction times for *near-error* and *far-error* phrases are shown. We log-transformed correction times to comply with the assumption of conditional normality, as is common practice with time measures [22]. We used linear mixed model analyses of variance [12,23], finding that there was no order effect on correction time ($F_{3,57}$=1.48, *n.s.*), confirming that our counterbalancing worked. Furthermore, *Technique* had a significant main effect on time for all phrases ($F_{3,57}$=26.49, p<.01), *near-error* phrases ($F_{3,57}$=29.02, p<.01), and *far-error* phrases ($F_{3,57}$=17.04, p<.01), allowing *post hoc* comparisons.

We performed six paired-samples t-tests corrected with Holm's sequential Bonferroni procedure [14], finding that for all phrases, the *de facto* cursor-based method was slower than *Drag-n-Throw* (t_{19}=6.66, p<.01) and *Magic Key* (t_{19}=4.79, p<.01); *Drag-n-Drop* was also slower than *Drag-n-Throw* (t_{19}=7.49, p<.01) and *Magic Key* (t_{19}=5.62, p<.01).

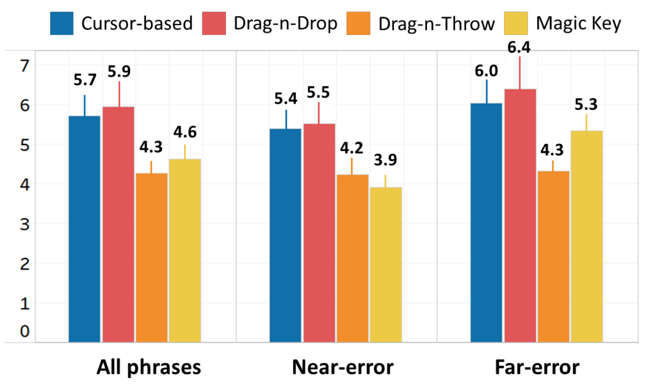


Figure 8. Average correction times in seconds for different interaction techniques (lower is better). *Drag-n-Throw* was the fastest for all phrases and *far-error* phrases, while *Magic Key* was the fastest for *near-error* phrases. Error bars are +1 *SD*.

For *near-error* phrases, the *de facto* method was slower than *Drag-n-Throw* (t_{19}=5.58, p<.01) and *Magic Key* (t_{19}=7.02, p<.01); *Drag-n-Drop* was also slower than *Drag-n-Throw* (t_{19}=5.00, p<.01) and *Magic Key* (t_{19}=7.44, p<.01).

For *far-error* phrases, *Drag-n-Throw* was faster than all other techniques: the *de facto* method (t_{19}=-5.65, p<.01), *Drag-n-Drop* (t_{19}=-6.60, p<.01), and *Magic Key* (t_{19}=-3.68, p<.01). Also, *Magic Key* was faster than *Drag-n-Drop* (t_{19}=-2.92, p<.05).

We then examined different correction types. Figure 9 shows the average correction times for *typos*, *word changes*, and *insertions*. Again, we used linear mixed model analyses of variance [12,23] on log correction time [22]. *Technique* had a significant main effect on time for all correction types: *typo* ($F_{3,57}$=5.11, p<.01), *word change* ($F_{3,57}$=10.87, p<.01) and *insertion* ($F_{3,57}$=55.55, p<.01), allowing *post hoc* comparisons.

We performed six paired-samples *t*-tests corrected with Holm's sequential Bonferroni procedure [14], finding that for *typos*, the *de facto* cursor-based method was slower than *Drag-n-Throw* (t_{19}=3.80, p<.01); *Drag-n-Drop* was also slower than *Drag-n-Throw* (t_{19}=2.70, p<.05).

For *word changes*, the *de facto* cursor-based method was slower than all other techniques: *Drag-n-Drop* (t_{19}=3.54, p<.01), *Drag-n-Throw* (t_{19}=5.58, p<.01), and *Magic Key* (t_{19}=3.74, p<.01).

For *insertions*, *Drag-n-Drop* was slower than all other interactions: the *de facto* cursor-based method (t_{19}=5.72, p<.01), *Drag-n-Throw* (t_{19}=11.17, p<.01), and *Magic Key* (t_{19}=10.92, p<.01). Also, the *de facto* method was slower than *Drag-n-Throw* (t_{19}=5.45, p<.01) and *Magic Key* (t_{19}=5.20, p<.01).

Correction Success Rate for *Drag-n-Throw* and *Magic Key*

In the uncontrolled text composition task, we recorded errors when participants were using *Drag-n-Throw* and *Magic Key*. With *Drag-n-Throw*, participants made 108 errors in all, and 95 of them were successfully corrected, a success rate of 87.9%. Among the successfully corrected errors, nine were

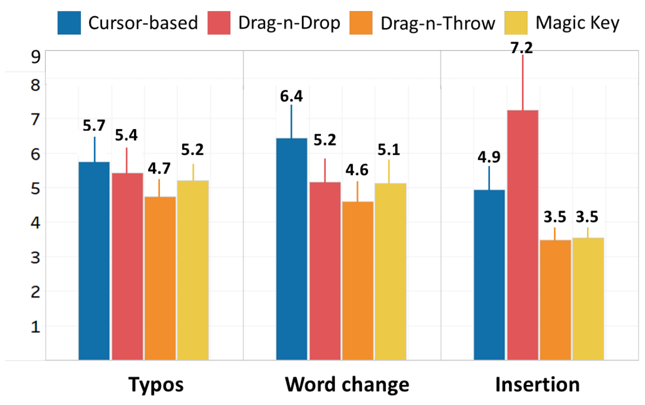


Figure 9. Average correction times in seconds for different correction types (lower is better). *Drag-n-Throw* was the fastest for all three types. Error bars are +1 *SD*.

attempted more than once because the corrections were not applied to expected error positions. With *Magic Key*, participants made 101 errors in all, and 98 of them were successfully corrected, a success rate of 97.0%.

Subjective Preferences

The composite scores of the SUS usability [5] and TLX [33] surveys for different interaction techniques are shown in Figure 10. Participants generally enjoyed using *Drag-n-Throw* and *Magic Key* more than the *de facto* cursor-based method and *Drag-n-Drop*. Also, the two neural network-based techniques were perceived to have lower workload than the other two techniques.

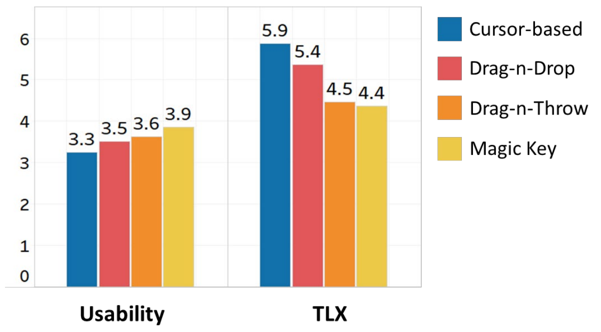


Figure 10. Composite usability (higher is better) and NASA TLX (lower is better) scores for different techniques. *Magic Key* was rated as the most usable and having the lowest workload.

DISCUSSION

In this work, we administered text correction and composition tasks to evaluate three new text correction techniques, two of which are based on neural networks. Although initially unfamiliar with the correction techniques, participants learned them quickly with only three practice phrases for each technique. The concept of "type the correction where you are, and then apply it to the error" was easily grasped. As P6 commented, "*People always think about a complex procedure to decide how to correct the error [when using cursor-based methods], like 'do I change some characters or delete the whole word to fix a typo?' But the three interactions provide an alternative way of thinking: just type the correction and apply it.*"

Drag-n-Throw performed fastest over all phrases and among different correction types. Moreover, its performance was unaffected by whether the error was near or far (Figure 8). *Magic Key* also achieved good speeds across different correction types. For *near-errors* within the last three words, it surpassed *Drag-n-Throw*, because errors could be corrected with just two taps. For *far-errors*, participants had to drag from atop the magic key a few times to highlight the desired error, leading to longer correction times.

Drag-n-Drop performed the slowest over all phrases, which was mainly caused by *insertion*s. As Figure 9 shows, it was faster than the *de facto* cursor-based method for *typos* and *word change*s, but much slower for *insertion*s. To insert a correction between two words, a user had to highlight the narrow space between those words. Many participants spent time adjusting their finger position in order to highlight the desired space. They also had to redo the correction if they accidentally made a substitution instead of an insertion. Our undo key proved to be vital in such cases.

The correction task was artificial, but managed to isolate the time performance of the techniques. Conversely, the composition ask was uncontrolled, amounting to more of a usability test than an experiment. In that task, *Drag-n-Throw* achieved a success rate of 87.9%. A failure case was when two possible error candidates were too close together. For example, if the user wanted to insert "the" in the phrase "I left keys in room," there were two possible positions (before "keys" and before "room"). *Magic Key* achieved a higher success rate of 97.0%, because it clearly highlighted the word to be corrected before applying the correction.

As for participants' preferences, 12 of 20 participants liked *Magic Key* the most. The major reason was convenience: all actions took place on the keyboard. P1 commented, "*Just one button handles everything. I don't need to touch the text anymore. It was also super intelligent. I am lazy, and that's why I enjoyed it so much.*" Another benefit was that *Magic Key* provided feedback before committing a correction, making users confident about the outcome of their actions. As P4 pointed out, "*It provides multiple choices, and the uncertain feeling is gone.*" The main critique of *Magic Key* was about the dragging interaction required to navigate among error candidates. P5 commented: "*If the text is too long and the error is far away, I have to drag [from atop the magic key] a lot to highlight the error. Also, the button is kinda small, and hard to drag.*"

Although *Drag-n-Throw* was the fastest interaction, and the concept of "flicking the correction" was appealing to participants, many felt confused about how to control the direction and finger-lift position. As P6 said, "*I'm not very confident in performing it because I do not know what will be corrected after my throw. There is not feedback during the procedure.*"

Despite the slow insertion time, *Drag-n-Drop* was perceived as the most intuitive, and was considered easiest to learn because it "*follows [the] current style of correction*" (P5, P6, P8, P9, P10). P10 also commented on the concept of the interaction: "*I definitely like the type-correction-then-apply-it concept, and I even did this in the cursor-based condition. Every time I move the cursor to fix the error, my typing flow is broken. This is even more annoying if I get emotional and type very fast with a lot of errors.*" There were also three users who preferred the cursor-based method due to familiarity.

Interestingly, all three participants above age 40 liked the two intelligent correction techniques, and disliked the *de facto* cursor-based method. P14, age 52, said, "*I dislike the cursor-based method most. I have a big finger, and it is hard to tap the text precisely. Throw is easy and works great. I also like Magic Key, because I don't need to interact with the text.*" Older adults are known to perform touch screen interactions more slowly and with less precision than younger adults [9], and the intelligent correction techniques might benefit them by removing the requirement of precise touch. Moreover, people walking on the street or holding the phone with one hand might also benefit from the interactions, because touching precisely is difficult in such situations.

FUTURE WORK

We propose four possible future directions: (1) *Punctuation handling*: Our current correction algorithm does not handle punctuation, so errors like "lets" ("let's") currently cannot be corrected. (2) *Feedback for Drag-n-Throw*: Participants felt unconfident when flicking corrections because there was a lack of feedback where corrections would land. Adding visual feedback such as highlighting around the text of the flicking position could provide cues as to where the correction will land. (3) *Better error navigation for Magic Key*: The magic key itself was small and hard to drag. Better interactions to navigate through different error candidates should be explored, such as a swipe gesture on the keyboard. (4) *Multilingual support*: Our interaction techniques could be applied to other languages, such as Chinese.

CONCLUSION

We presented three novel text correction techniques, *Drag-n-Drop*, *Drag-n-Throw*, and *Magic Key*. The common concept in these three techniques was to type the correction and apply it to the error, without needing to reposition the text cursor or use backspace—maneuvers that together break the typing flow and slow touch-based text entry. *Drag-n-Throw* and *Magic Key* used recurrent neural networks (RNNs) to identify correction positions. Our user study showed that *Drag-n-Throw* and *Magic Key* were faster than *de facto* cursor-based correction methods and garnered more positive user feedback. This work provides an example of how, by breaking from the desktop paradigm of arrow keys, backspacing, and mouse-based cursor positioning, we can rethink text entry on mobile touch devices and develop novel methods better suited to this paradigm.

ACKNOWLEDGEMENTS

This work was supported in part by Baidu. Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any supporter.

REFERENCES

[1] Ahmed Sabbir Arif, Sunjun Kim, Wolfgang Stuerzlinger, Geehyuk Lee and Ali Mazalek. 2016. Evaluation of a smart-restorable backspace technique to facilitate text entry error correction. *Proceedings of CHI 2016*. New York: ACM Press, 5151–5162. DOI: 10.1145/2858036.2858407

[2] Ahmed Sabbir Arif and Wolfgang Stuerzlinger. 2013. Pseudo-pressure detection and its use in predictive text entry on touchscreens. *Proceedings of OzCHI 2013*. New York: ACM Press, 383–392. DOI: 10.1145/2541016.2541024

[3] Dzmitry Bahdanau, Kyunghyun Cho and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *Proceedings of International Conference on Learning Representations (ICLR '15)*. https://arxiv.org/abs/1409.0473

[4] Hrvoje Benko, Andrew D Wilson and Patrick Baudisch. 2006. Precise selection techniques for multi-touch screens. *Proceedings of CHI 2006*. New York: ACM Press, 1263–1272. DOI: 10.1145/1124772.1124963

[5] John Brooke. 1996. SUS: A "quick and dirty" usability scale. In P.W. Jordan, B. Thomas, B.A. Weerdmeester, & A.L. McClelland (eds.), *Usability Evaluation in Industry*. London: Taylor and Francis.

[6] Christopher Bryant and Hwee Tou Ng. 2015. How far are we from fully automatic high quality grammatical error correction? *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Vol. 1)*. Stroudsburg, PA: Association for Computational Linguistics, 697–707. DOI: 10.3115/v1/P15-1068

[7] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*. Stroudsburg, PA: Association for Computational Linguistics, 1724–1734. DOI: 10.3115/v1/D14-1179

[8] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson and Antti Oulasvirta. 2018. Observations on typing from 136 million keystrokes. *Proceedings of CHI 2018*. New York: ACM Press. Paper No. 646. DOI: 10.1145/3173574.3174220

[9] Leah Findlater, Jon E. Froehlich, Kays Fattal, Jacob O. Wobbrock and Tanya Dastyar. 2013. Age-related differences in performance with touchscreens compared to traditional mouse input. *Proceedings of CHI 2013*. New York: ACM Press, 343–346. DOI: 10.1145/2470654.2470703

[10] George Fitzmaurice, Azam Khan, Robert Pieké, Bill Buxton and Gordon Kurtenbach. 2003. Tracking menus. *Proceedings of UIST 2003*. New York: ACM Press, 71–79. DOI: 10.1145/964696.964704

[11] Andrew Fowler, Kurt Partridge, Ciprian Chelba, Xiaojun Bi, Tom Ouyang and Shumin Zhai. 2015. Effects of language modeling and its personalization on touchscreen typing performance. *Proceedings of CHI 2015*. New York: ACM Press, 649–658. DOI: 10.1145/2702123.2702503

[12] B.N. Frederick. 1999. Fixed-, random-, and mixed-effects ANOVA models: A user-friendly guide for increasing the generalizability of ANOVA results. In B. Thompson (ed.), *Advances in Social Science Methodology*. Stamford, CT: JAI Press, 111–122. http://eric.ed.gov/?id=ED426098

[13] Vittorio Fuccella, Poika Isokoski and Benoit Martin. 2013. Gestures and widgets: Performance in text editing on multi-touch capable mobile devices. *Proceedings of CHI 2013*. New York: ACM Press, 2785–2794. DOI: 10.1145/2470654.2481385

[14] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics 6* (2), 65–70. http://www.jstor.org/stable/4615733

[15] Christian Holz and Patrick Baudisch. 2011. Understanding touch. *Proceedings of CHI 2011*. New York: ACM Press, 2501–2510. DOI: 10.1145/1978942.1979308

[16] Aminul Islam and Diana Inkpen. 2009. Real-word spelling correction using Google Web IT 3-grams. *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, Vol. 3 (EMNLP '09)*. Stroudsburg, PA: Association for Computational Linguistics, 1241–1249. https://dl.acm.org/citation.cfm?id=1699670

[17] Yoon Kim, Yacine Jernite, David Sontag and Alexander M. Rush. 2016. Character-aware neural language models. *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI '16)*. Menlo Park, CA: AAAI Press, 2741–2749. https://arxiv.org/abs/1508.06615

[18] Andreas Komninos, Mark Dunlop, Kyriakos Katsaris and John Garofalakis. 2018. A glimpse of mobile text entry errors and corrective behaviour in the wild. In *Proceedings of MobileHCI 2018*. New York: ACM Press, 221–228. DOI: 10.1145/3236112.3236143

[19] Andreas Komninos, Emma Nicol and Mark D. Dunlop. 2015. Designed with older adults to support better error correction in smartphone text entry: The MaxieKeyboard. *Adjunct Proceedings of MobileHCI 2015*. New York: ACM Press, 797–802. DOI: 10.1145/2786567.2793703

[20] Luis A. Leiva, Alireza Sahami, Alejandro Catala, Niels Henze and Albrecht Schmidt. 2015. Text entry on tiny QWERTY soft keyboards. *Proceedings of CHI 2015*. New York: ACM Press, 669–678. DOI: 10.1145/2702123.2702388

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

[22] Eckhard Limpert, Werner A. Stahel and Markus Abbt. 2001. Log-normal distributions across the sciences: Keys and clues. *BioScience 51* (5), 341–352. https://bit.ly/2JLVdir

[23] R.C. Littell, P.R. Henry and C.B. Ammerman. 1998. Statistical analysis of repeated measures data using SAS procedures. *Journal of Animal Science 76* (4), 1216–1231. DOI: 10.2527/1998.7641216x

[24] I. Scott MacKenzie and R. William Soukoreff. 2002. Text entry for mobile computing: Models and methods, theory and practice. *Human-Computer Interaction 17* (2-3), 147–198. http://www.yorku.ca/mack/hci3.html

[25] I. Scott MacKenzie and R. William Soukoreff. 2002. A character-level error analysis technique for evaluating text entry methods. *Proceedings of NordiCHI 2002*. New York: ACM Press, 243–246. DOI: 10.1145/572020.572056

[26] Eugene W. Myers. 1986. An *O(ND)* difference algorithm and its variations. *Algorithmica 1* (1-4), 251–266. DOI: 10.1007/BF01840446

[27] Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto and Christopher Bryant. 2014. The CoNLL-2014 Shared Task on grammatical error correction. *Proceedings of the 18th Conference on Computational Natural Language Learning: Shared Task*, 1–14.

[28] 28. Hwee Tou Ng, Siew Mei Wu, Yuanbin Wu, Christian Hadiwinoto, and Joel Tetreault. 2013. The CoNLL-2013 Shared Task on Grammatical Error Correction. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*. Stroudsburg, PA: Association for Computational Linguistics, 1–12. DOI: 10.3115/v1/W14-1701

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga and Adam Lerer. 2017. Automatic differentiation in PyTorch. *Proceedings of NIPS 2017 Autodiff Workshop*. https://openreview.net/forum?id=BJJsrmfCZ

[30] Jeffrey Pennington, Richard Socher and Christopher Manning. 2014. Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*. Stroudsburg, PA: Association for Computational Linguistics, 1532–1543. DOI: 10.3115/v1/D14-1162

[31] Radim Rehurek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. *Proceedings of LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: University of Malta, 46–50.

[32] Sherry Ruan, Jacob O. Wobbrock, Kenny Liou, Andrew Ng and James A. Landay. 2017. Comparing speech and keyboard text entry for short messages in two languages on touchscreen phones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies 1* (4), Article No. 159. DOI: 10.1145/3161187

[33] Susana Rubio, Eva Díaz, Jesús Martín and José M. Puente. 2004. Evaluation of subjective mental workload: A comparison of SWAT, NASA-TLX, and workload profile methods. *Applied Psychology: An International Review 53* (1), 61–86. DOI: 10.1111/j.1464-0597.2004.00161.x

[34] Dominik Schmidt, Florian Block and Hans Gellersen. 2009. A comparison of direct and indirect multi-touch input for large surfaces. *Proceedings of INTERACT 2009*. Berlin: Springer-Verlag, 582–594. DOI: 10.1007/978-3-642-03655-2_65

[35] Andrew Sears and Ben Shneiderman. 1991. High precision touchscreens: Design strategies and comparisons with a mouse. *International Journal of Man-Machine Studies 34* (4), 593–613. DOI: 10.1016/0020-7373(91)90037-8

[36] R. William Soukoreff and I. Scott MacKenzie. 2004. Recent developments in text-entry error rate measurement. *Extended Abstracts of CHI 2004*. New York: ACM Press, 1425–1428. DOI: 10.1145/985921.986081

[37] Ilya Sutskever, Oriol Vinyals and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems 27 (NIPS '14)*. Red Hook, NY: Curran Associates, Inc., 3104–3112. https://arxiv.org/abs/1409.3215

[38] Keith Vertanen, Haythem Memmi, Justin Emge, Shyam Reyal and Per-Ola Kristensson. 2015. VelociTap: Investigating fast mobile text entry using sentence-based decoding of touchscreen keyboard input. *Proceedings of CHI 2015*. New York: ACM Press, 659–668. DOI: 10.1145/2702123.2702135

[39] Daniel Vogel and Patrick Baudisch. 2007. Shift: A technique for operating pen-based interfaces using touch. *Proceedings of CHI 2007*. New York: ACM Press, 657–666. DOI: 10.1145/1240624.1240727

[40] Jacob O. Wobbrock and Brad A Myers. 2006. Analyzing the input stream for character-level errors in unconstrained text entry evaluations. *ACM*

Transactions on Computer-Human Interaction 13 (4), 458–489. DOI: 10.1145/1188816.1188819

[41] Ziang Xie, Anand Avati, Naveen Arivazhagan, Dan Jurafsky and Andrew Y. Ng. 2016. Neural language correction with character-based attention. https://arxiv.org/abs/1603.09727

[42] Tom Young, Devamanyu Hazarika, Soujanya Poria and Erik Cambria. 2018. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine 13* (3), 55–75. DOI: 10.1109/MCI.2018.2840738

[43] Torsten Zesch. 2012. Measuring contextual fitness using error contexts extracted from the Wikipedia revision history. *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL '12)*. Stroudsburg, PA: Association for Computational Linguistics, 529–538. https://www.aclweb.org/anthology/E12-1054

[44] Shumin Zhai and Per-Ola Kristensson. 2012. The word-gesture keyboard: Reimagining keyboard interaction. *Communications of the ACM 55* (9), 91–101. DOI: 10.1145/2330667.2330689

[45] Xiang Zhang, Junbo Zhao and Yann LeCun. 2015. Character-level convolutional networks for text classification. *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS '15)*. Cambridge, MA: MIT Press, 649–657. https://arxiv.org/abs/1509.01626

[46] Suwen Zhu, Tianyao Luo, Xiaojun Bi and Shumin Zhai. 2018. Typing on an invisible keyboard. *Proceedings of CHI 2018*. New York: ACM Press. Paper No. 439. DOI: 10.1145/3173574.3174013

APPENDIX

	Near-error	**Far-error**
Typos	The season would end the **net** week. //**next**	Untreated septicemic plague is universally fatal, but early treatment with antibiotics reduces the morality rate to between 4 and 15 percent. //**mortality**
Word change	I suggest you get facts **after** judging anyone. //**before**	This book is **very** touching. It tells Dorie's story of all the unbelievably horrible things while growing up. //**quite**
Insertion	Where do you want to meet to walk **()** there? //**over**	If you're just waiting for Vol. 2, why **()** you buy this in the first place? //**did**

Examples of different error types from our user study. Error text is highlighted in bold; insertion errors are represented by parentheses. The correction is shown at the end of each phrase.