CS221 Project Proposal

Julia Gong, Benjamin Newman

November 2018

1 Introduction

As a quick reminder, the problem we are trying to address is the transliteration of names spelled with with Latin characters (particularly those in English) to Chinese pinyin pronunciations (and if we have time, character representations). This seemingly simple task is actually quite complex, so we are taking two approaches to these problems: a search-based approach and a deep learning-based approach. Our baseline was performing a table look-up for similar-sounding words and our oracle was asking native Chinese speakers to transliterate names.

2 Search

2.1 Formalization

On a high level, we approached this search problem as searching for the best way to break up English written names into pinyin. To do this, we defined a cost function for every possible pinyin representation of an English name and chose the transliteration with the lowest cost.

First, using our data set of already transliterated English names, we obtained their space-separated pinyin representations using the Glosbe API at `https://glosbe.com/`. We wanted to know often each pinyin was used in names, so we created a pinyin cost function similar to the one in the `reconstruct` assignment. We defined the cost of a pinyin syllable as its surprisal ($-\log(p)$, where p is the maximum likelihood estimate of the probability of a pinyin syllable in our data set, i.e. the number of times the pinyin syllable appeared divided by the total number of pinyin syllables in the corpus). We were thinking of creating a bi-syllable cost function as well, but as the names are very short, we were not sure if having a bigram function would help much. We might experiment with this method later if we have time.

Next, we formulated the search problem. Given a single English name, we ideally would have to break the name into syllables and find the pinyin that matched each syllable. As the syllable segmentaiton task is a bit out of the scope of the project, we instead consider all possible syllable segmentations of the name. Even though this approach grows exponentially with the length of the name, overall, names tend to be short enough that this does not pose a problem. For example, when transliterating the name BEN, we would consider the segmentations

$$B \mid EN,\ BE \mid N,\ B \mid E \mid N, \text{ and } BEN$$

.

Now we have to calculate the cost of each segmentation. This is a bit challenging because the cost function we described earlier tells us the cost of a pinyin syllable, but we need to evaluate an English sub-word. Calling the cost function directly on the English sub-word is an option, but because the Pinyin cost function takes into account aspects of Chinese pinyin, such as tones, it is difficult to account for that. We did try to take the total costs of each toned version of the word, but it can be a non-trivial problem to decide which vowel receives the tone in certain pinyin. To this end, we decided to assign an English sub-word a pinyin and then iterate through all the pinyin in our dataset and find the pinyin that minimizes the product:

$$\text{edit\_distance(subword, pinyin)} \cdot \text{cost(pinyin)}$$

Here, edit distance is defined in the same way it was in our proposal: the minimum number of substitutions, insertions or deletions we have to make to change the English sub-word into the pinyin syllable. Because we are more likely to need to make some additions or deletions rather than substitutions, we assign lower costs to strings that start out more similar to one another by subtracting the total number of characters the two strings have in common from their final edit distance.

2.2 Preliminary Results

Our preliminary results show that this approach does not work as well as we had hoped.

We tried to translate all 1510 names in our dataset. 1488 were different from expected, with an average edit distance of 4.36 among the pairs that were different. This number is calculated using our custom "pinyin" edit distance which penalizes incorrect tones half as much as incorrect vowels or consonants.

We ran the search for our names and examined how it performs:

$$\text{Benjamin} \longrightarrow \text{běn jiā mǐn}$$

This is in comparison to the ground truth transliteration:

$$\text{Benjamin} \longrightarrow \text{běn jié míng}$$

Here, we can see that the name 'Benjamin' gives the pinyin 'běn', 'jiā', and 'mǐn', which is actually relatively close to the actual transliteration.

However, running this algorithm with the name 'Julia' gives the following result:

$$\text{Julia} \longrightarrow \text{bù lián}$$

Compare this to the ground truth transliteration:

$$\text{Julia} \longrightarrow \text{zhū lì yà}$$

The pinyin 'bù', 'lián' is incorrect both in the number of syllables (there should be three) and in the sounds that they make ('bu' and 'ju' are not really related). Part of the problem is likely related to the fact that there is no pinyin sequence *ju* of any kind in our dataset because this is not a valid pinyin. Instead, that sound combination is handled by the pinyin *zhu*, which has a higher edit distance than 'bu' to 'ju'. Ways we can address this problem are in the future directions section.

2.3 Future Directions

One issue we noticed when looking at examples similar to the 'Benjamin' case was that the transliteration produced by our approach, such as 'běn jiā mǐn', actually has a lower edit distance (often near-perfect, as in this case) to 'Benjamin' than the ground truth transliteration, despite the fact that the transliteration is in reality not ideal. Chinese name transliterations often do not rely solely on character-wise closeness; rather, other factors and conventions come into play that are difficult to program in a simple cost function without doing some kind of learning. This observation exposes a problem inherent to our algorithm for assigning pinyin to English sub-words, since we do not have a systematic way to account for these differences when using a non-stochastic approach. This is what inspired us to turn our attention to a deep learning approach, which could possibly learn these translations in a more generalized manner. We address this method in the following section.

In addition, it seems that most of the other problems we have are due to the fact that our metric of edit distance tries to find pinyin that are written similarly to English characters, which is not always guaranteed, since sound translations (such as that from 'j' to 'zh') do not appear to be similar on paper. We can probably address this by augmenting the way we choose the pinyin. For example, we could incorporate something akin to "sound" edit distance that would use the fact that phonemes that sound the same in English and Pinyin (such as `/J/` and `/ZH/`) have a smaller edit distance than 2. This would probably involve hard-coding in some of these English-Pinyin character pairs that make the same sounds (such `/sh/` and `/x/` or `/c/` and `/ts/` . We were trying to stay away from having to do this because it can be quite difficult to identify English phonemes from letters, but it might be needed to achieve decent results.

3 Deep Learning

3.1 Formalization

The other way we attempt to solve the problem of transliterating English names into Chinese is by taking a deep learning approach. As this is not really the focus of the class, we treat this approach similarly to a classification problem. Because we are training a model and then evaluating it, we split the dataset of 1510 names into a training set of 1410 names and a testing set of 100 names. We use a modified version of the machine translation tutorial code available on the Pytorch website. Here, we are using an encoder-decoder model. First, we calculate an embedding of each character in the name and then use a gated recurrent unit (GRU) layer to learn relationships between characters and produce a final encoding. The GRU output consists of a character encoding and a hidden state representing the context of the name. Then, for the decoder, we calculate an embedding for each character and use the hidden state we calculate in the encoder stage. We run the embedding through another GRU layer and then put the output through a linear layer to act as a classifier for what the next character of the pinyin translation should be.

Because this is a learning task, we need a loss function to calculate our error and backpropagate it through the network to update the network parameters. We decided to use the cross-entropy loss function for this task. This is because it allows us to have a probability calculated for each next possible pinyin character, and we want to drive these predictions so that the probability for a given pinyin character is higher for the correct one.

3.2 Preliminary Results

With an embedding size of 20, and our 1410 examples, the network take approximately 8 minutes to train. The results are a little disappointing, but not entirely disastrous. Of the 100 names in our test set, only 2 were transliterated exactly correctly, giving us an accuracy of 2%. Of the names that were incorrectly recreated, the average edit distance was 4.13 characters, which means that on average, we would have to use about 4 character substitutions, deletions, or insertions to arrive at the correct pinyin representation of the name. This also takes into account incorrect pinyin by increasing the edit distance by 0.5 rather than 1 for cases when the vowels are correct, but the tone is wrong.

We now take a closer look at the same names we highlighted in the search section:

Benjamin $\longrightarrow$ pěn méi ěr

Julia $\longrightarrow$ jié ěr u

We can see that the results here are perhaps worse than the results we found in the search. For 'Benjamin', while /p/ and /b/ do share some similarities, the /j/ sound is completely absent, and there is a seemingly random 'ěr' sound as well. For 'Julia', while we do have the correct number of syllables, in this case, arguably, the sounds we have are much worse as well.

The kinds of mistakes the network made in this case were different from the kinds of mistakes that the search problem made. Because in the deep learning case we had embeddings for individual characters, it was possible that some of the pinyin that were generated were not actually valid pinyin at all! We observed this result for many names, including the transliteration of 'Julia' above. The pinyin 'u' does not exist and would have be preceded by a 'w' for the pinyin to be valid. Here are a few other names we found this problem with:

Henry $\longrightarrow$ hăn nn

James $\longrightarrow$ jiél l s nn

Another interesting mistake that this network makes is in the use of spaces. At the moment, the input pinyin that are being fed in are space separated, so the network also has the necessary data to learn the correct syllable separations. In general, this seems to be working pretty well, as even when the pinyin themselves are incorrect, they are not separated by unnecessary spaces. Additionally, the generated spaces usually do not separate portions of pinyin that would be pinyin if there was not a space between. Of course,

there are some counter examples to these trends as well, such as the output for 'Claudia', which produced two spaces between the last two pinyin syllables.

$$\text{Claudia} \longrightarrow \text{kè lì kè}$$

3.3 Future Directions

While we might not want to spend a large amount of time devoted to this path, there are a few other things we can try to remedy the non-sensical pinyin problem. First, we could try outputting syllables instead of letters. This would mean that everything we output would have to be a valid combination of pronounceable sounds. This also corresponds a little more naturally to how a human might preform this task. Rather than matching characters to characters, the idea would be to match characters to syllables.

Additionally, we could try going straight from the Anglicized version of the name to the character version and skip the pinyin representation entirely. This presents an entirely different set of challenges, given the large amount of Chinese characters as opposed to characters in the English alphabet, as now the space we would be decoding into would be MUCH larger than the space we are encoding into, so the results might not be as good. Additionally, the approach of using character embeddings may be kind of overkill, as there are only 26 input characters. Thus, we might try a simpler RNN with context approach to try to learn more information about what goes into these more complex units like GRUs and LSTMs. Finally, if this approach is not overkill enough and we are able to obtain more data, we can try training a bi-directional GRU that would allow us to pick up patterns moving in both encoding and decoding directions.

Another idea we have is to create a new hybrid framework that combines the strengths of our two methods: the search problem formulation and the deep learning framework. Since the search problem uses heuristics that are less flexible, but provide a higher baseline accuracy, we could try to first put the English name through the search model. Then, we could design a new deep learning model that takes the output of the search model, which consists of the initial 'guesses' for the transliterated pinyin, and refines this initial pinyin guess when trained to translate this guess into the ground truth dataset of the ground truth pinyin. This way, we will have decreased the initial difference between input and output character sequences for the deep learning model, which could reduce the various errors in the deep learning model, such as incorrect spacing or invalid pinyin, which we described above.

4 Final Conclusions

The problems we are running into with the search and deep learning methods are on entirely different ends of the spectrum. With the search method, there is not really any learning taking place - all of the problems occur because of the inflexibility of our edit-distance distance metric. If we augment our cost function to include (or learn) relationships between sounds and character symbols, we might be able to make better predictions. Our problem with the deep learning method, however, is actually the opposite. We don't have the structure of a pinyin symbol a priori, so we need to learn what this looks like. We already have plenty of examples of what these syllables are, so if we could add some structure into the model (by limiting outputs to valid syllables from our training data, for instance, or by using the output of the search model as the input to the deep learning model), we might be able to arrive at better results.