Data augmentation for low-resource grapheme-to-phoneme mapping

Michael Hammond
Dept. of Linguistics
U. of Arizona
Tucson, AZ, USA
hammond@u.arizona.edu

Abstract

In this paper we explore a very simple neural approach to mapping orthography to phonetic transcription in a low-resource context. The basic idea is to start from a baseline system and focus all efforts on data augmentation. We will see that some techniques work, but others do not.

1 Introduction

This paper describes a submission by our team to the 2021 edition of the SIGMORPHON Grapheme-to-Phoneme conversion challenge. Here we demonstrate our efforts to improve grapheme-to-phoneme mapping for low-resource languages in a neural context using only data augmentation techniques.

The basic problem in the low-resource condition was to build a system that maps from graphemes to phonemes with very limited data. Specifically, there were 10 languages with 800 training pairs and 100 development pairs. Each pair was a word in its orthographic representation and a phonetic transcription of that word (though some multi-word sequences were also included). Systems were then tested on 100 additional pairs for each language. The 10 languages are given in Table 1.

Code	Language
ady	Adyghe
gre	Modern Greek
ice	Icelandic
ita	Italian
khm	Khmer
lav	Latvian
mlt(_latn)	Maltese (Latin script)
rum	Romanian
slv	Slovene
wel(_sw)	Welsh (South Wales dialect)

Table 1: Languages and codes

To focus our efforts, we kept to a single system, intentionally similar to one of the simple baseline systems from the previous year's challenge.

We undertook and tested three data augmentation techniques.

1. move as much development data to training data as possible
2. extract substring pairs from the training data to use as additional training data
3. train all the languages together

In the following, we first provide additional details on our base system and then outline and test each of the moves above separately. We will see that some work and some do not.

We acknowledge at the outset that we do not expect a system of this sort to "win". Rather, we were interested in seeing how successful a minimalist approach might be, one that did not require major changes in system architecture or training. This minimalist approach entailed that the system not require a lot of detailed manipulation and so we started with a "canned" system. This approach also entailed that training be something that could be accomplished with modest resources and time. All configurations below were run on a Lambda Labs Tensorbook with a single GPU.[1] No training run took more than 10 minutes.

2 General architecture

The general architecture of the model is inspired by one of the 2020 baseline systems (Gorman et al., 2020): a sequence-to-sequence neural net with a two-level LSTM encoder and a two-level LSTM decoder. The system we used is adapted from the OpenNMT base (Klein et al., 2017).

There is a 200-element embedding layer in both

[1]RTX 3080 Max-Q.

August 5, 2021. ©2

encoder and decoder. Each LSTM layer has 300 nodes. The systems are connected by a 5-head attention mechanism (Luong et al., 2015). Training proceeds in 24,000 steps and the learning rate starts at 1.0 and decays at a rate of 0.8 every 1,000 steps starting at step 10,000. Optimization is stochastic gradient descent, the batch size is 64, the dropout rate is 0.5.

We spent a fair amount of time tuning the system to these settings for optimal performance with this general architecture on these data. We do not detail these efforts as this is just a normal part of working with neural nets and not our focus here.

Precise instructions for building the docker image, full configuration files, and auxiliary code files are available at `https://github.com/hammondm/g2p2021`.

3 General results

In this section, we give the general results of the full system with all strategies in place and then in the next sections we strip away each of our augmentation techniques to see what kind of effect each has. In building our system, we did not have access to the correct transcriptions for the test data provided, so we report performance on the development data here.

The system was subject to certain amount of randomness because of randomization of training data and random initial weights in the network. We therefore report mean final accuracy scores over multiple runs.

Our system provides accuracy scores for development data in terms of character-level accuracy. The general task was scored in terms of word-level error rate, but we keep this measure for several reasons. First, it was simply easier as this is what the system provided as a default. Second, this is a more granular measure that enabled us to adjust the system more carefully. Finally, we were able to simulate word-level accuracy in addition as described below.

We use a Monte Carlo simulation to calculate expected word-level accuracy based on character-level accuracy and average transcription length for the training data for the different languages. The simulation works by generating $100,000$ words with a random distribution of a specific character-level accuracy rate and then calculating word-level accuracy from that. Running the full system ten times, we get the results in Table 2. Keep in mind that we are reporting accuracy rather than error rate, so the goal is to maximize these values.

	Character	Word
	94.84	75.6
	94.78	75.3
	94.46	74.0
	94.84	75.5
	94.71	75.0
	94.59	74.5
	94.90	75.8
	94.53	74.2
	94.53	74.2
	94.71	75.0
Mean	94.69	74.91

Table 2: Development accuracy for 10 runs of the full system with all languages grouped together with estimated word-level accuracy

	Character	Word
	94.60	74.6
	94.71	75.0
	94.35	73.8
	94.48	74.0
	94.48	74.0
	94.50	74.2
	94.59	74.5
	94.71	75.0
	94.80	75.4
	94.59	74.5
Mean	94.58	74.5

Table 3: Development accuracy for 10 runs of the reduced system with all languages grouped together with 100 development pairs with estimated word-level accuracy

4 Using development data

The default partition for each language is 800 pairs for training and 100 pairs for development. We shifted this to 880 pairs for training and 20 pairs for development. The logic of this choice was to retain what seemed like the minimum number of development items. Running the system ten times without this repartitioning gives the results in Table 3.

There is a small difference in the right direction, but it is not significant for characters ($t = -1.65$, $p = 0.11$, unpaired) or words ($t = -1.56$, $p = 0.13$, unpaired). It may be that with a larger sample of runs, the difference becomes more stable.

Code	Items added
ady	4
gre	223
ice	58
ita	194
khm	39
lav	100
mlt_latn	62
rum	119
slv	127
wel_sw	7

Table 4: Number of substrings added for each language

5 Using substrings

This method involves finding peripheral letters that map unambiguously to some symbol and then finding plausible splitting points within words to create partial words that can be added to the training data.

Let's exemplify this with Welsh. First, we identify all word-final letters that always correspond to the same symbol in the transcription. For example, the letter *c* always corresponds to a word-final [k]. Similarly, we identify word-initial characters with the same property. For example, in these data, the word-initial letter *t* always corresponds to [t].[2] We then search for any word in training that has the medial sequence *ct* where the transcription has [kt]. We take each half of the relevant item and add them to the training data if that pair is not already there. For example, the word *actor* [aktɔr] fits the pattern, so we can add the pairs ac-ak and tor-tɔr. to the training data. Table 4 gives the number of items added for each language. This strategy is a more limited version of the "slice-and-shuffle" approach used by Ryan and Hulden (2020) in last year's challenge.

Note that this procedure can make errors. If there are generalizations about the pronunciation of letters that are not local, that involve elements at a distance, this procedure can obscure those. Another example from Welsh makes the point. There are exceptions, but the letter *y* in Welsh is pronounced two ways. In a word-final syllable, it is pronounced [ɨ], e.g. *gwyn* [gwɨn] 'white'. In a non-final syllable, it is pronounced [ə], e.g. *gwynion* [gwənjɔn] 'white ones'. Though it doesn't happen in the training data here, the procedure above could easily result in a *y* in a non-final syllable ending up in a final syllable in a substring generated as above.

[2]This is actually incorrect for the language as a whole. Word-initial *t* in the digraph *th* corresponds to a different sound [θ].

	Character	Word
	95.15	76.9
	94.40	73.7
	95.15	76.8
	94.59	74.5
	94.65	74.8
	95.27	77.4
	94.53	74.2
	94.78	75.2
	95.09	76.6
	94.59	74.5
Mean	94.82	75.46

Table 5: 10 runs with all languages grouped together without substrings added for each language

Table 5 shows the results of 10 runs without these additions and simulated word error rates for each run.

Strikingly, adding the substrings *lowered* performance, but the difference with the full model is not significant for either characters ($t = 1.18$, $p = 0.25$, unpaired) or for words ($t = 1.17$, $p = 0.25$, unpaired). This model without substrings is the best-performing of all the models we tried, so this is what was submitted.

6 Training together

The basic idea here was to leverage the entire set of languages. Thus all languages were trained together. To distinguish them, each pair was prefixed by its language code. Thus if we had orthography $O = \langle o_1, o_2, \ldots, o_n \rangle$ and transcription $T = \langle t_1, t_2, \ldots, t_m \rangle$ from language x, the net would be trained on the pair $O' = \langle x, o_1, o_2, \ldots, o_n \rangle$ and $T' = \langle x, t_1, t_2, \ldots, t_m \rangle$. The idea is that, while the mappings and orthographies are distinct, there are similarities in what letters encode what sounds and in the possible sequences of sounds that can occur in the transcriptions. This approach is very similar to that of Peters et al. (2017), except that we tag the orthography and the transcription with the same langauge tag. Peters and Martins (2020) took a similar approach in last years challenge, but use embeddings prefixed at each time step.

In Table 6 we give the results for running each language separately 5 times. Since there was a lot less training data for each run, these models settled faster, but we ran them the same number of steps as the full models for comparison purposes.

There's a lot of variation across runs and the means for each language are quite different, presumably based on different levels of orthographic transparency. The general pattern is clear that, overall, training together does better than training separately. Comparing run means with our baseline system is significant ($t = -6.06$, $p < .001$, unpaired).

This is not true in all cases however. For some languages, individual training seems to be better than training together. Our hypothesis is that languages that share similar orthographic systems did better with joint training and that languages with diverging systems suffered.

The final results show that our best system (no substrings included, all languages together, moving development data to training) performed reasonably for some languages, but did quite poorly for others. This suggests a hybrid strategy that would have been more successful. In addition to training the full system here, train individual systems for each language. For test, compare final development results for individual languages for the jointly trained system and the individually trained systems and use whichever does better for each language in testing.

7 Conclusion

To conclude, we have augmented a basic sequence-to-sequence LSTM model with several data augmentation moves. Some of these were successful: redistributing data from development to training and training all the languages together. Some techniques were not successful though: the substring strategy resulted in diminished performance.

Acknowledgments

Thanks to Diane Ohala for useful discussion. Thanks to several anonymous reviewers for very helpful feedback. All errors are my own.

References

Kyle Gorman, Lucas F.E. Ashby, Aaron Goyzueta, Arya McCarthy, Shijie Wu, and Daniel You. 2020. The SIGMORPHON 2020 shared task on multilingual grapheme-to-phoneme conversion. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 40–50. Association for Computational Linguistics.

G. Klein, Y. Kim, Y. Y. Deng, J. Senellart, and A.M. Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. *ArXiv e-prints*. 1701.02810.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

Ben Peters, Jon Dehdari, and Josef van Genabith. 2017. Massively multilingual neural grapheme-to-phoneme conversion. In *Proceedings of the First Workshop on Building Linguistically Generalizable NLP Systems*, pages 19–26, Copenhagen. Association for Computational Linguistics.

Ben Peters and André F. T. Martins. 2020. DeepSPIN at SIGMORPHON 2020: One-size-fits-all multilingual models. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 63–69. Association for Computational Linguistics.

Zach Ryan and Mans Hulden. 2020. Data augmentation for transformer-based G2P. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 184–188. Association for Computational Linguistics.

language	**5 separate runs**					**Mean**
ady	95.27	91.12	93.49	94.67	93.49	93.61
gre	97.25	98.35	98.35	98.90	98.90	98.35
ice	91.16	94.56	93.88	90.48	94.56	92.93
ita	93.51	94.59	94.59	94.59	94.59	94.38
khm	94.19	90.32	90.97	90.97	90.97	91.48
lav	94.00	90.67	89.33	92.00	90.67	91.33
mlt_latn	91.89	94.59	91.89	92.57	93.24	92.84
rum	95.29	96.47	94.71	95.88	95.29	95.51
slv	94.01	94.61	04.61	94.61	94.01	94.37
wel_sw	96.30	97.04	96.30	97.04	96.30	96.59
Mean	94.29	94.23	93.81	94.17	94.2	94.14

Table 6: 5 separate runs for each language