English-to-English Grapheme-to-Phoneme Transliteration using RNN-Transducer

EE698R

Kumar Kanishk Singh (210544)
Sunny Raja Prasad (218171078)
Tanmey Agarwal (211098)

April 24, 2025

Abstract

Grapheme-to-Phoneme (G2P) conversion is a core task in speech and language processing, mapping written words (graphemes) to their pronunciations (phonemes). In this project, we implement an English G2P system using the RNN-Transducer (RNN-T) architecture, leveraging the CMU Pronouncing Dictionary as our dataset. Our approach models the sequence-to-sequence relationship between English words and their phonetic transcriptions, using an encoder-predictor-joiner neural architecture trained end-to-end. Performance is evaluated using Character Error Rate (CER).

1 Introduction

G2P conversion is essential for many speech technology applications, including text-to-speech (TTS), automatic speech recognition (ASR), and machine transliteration. The English G2P task is challenging due to irregular spelling and pronunciation rules, silent letters, and context-dependent mappings. Traditional approaches rely on alignment and hand-crafted rules, but neural sequence models have shown superior performance in recent years . In this work, we develop a neural G2P system based on the RNN-Transducer, which is well-suited for variable-length sequence mapping till MXLEN of 15.

2 Dataset and Preprocessing

2.1 CMU Pronouncing Dictionary

The CMU Pronouncing Dictionary (CMUdict) serves as our primary dataset for the G2P conversion task. This widely-used lexical resource contains:

- Over 134,000 English words with corresponding phonetic transcriptions
- North American English pronunciations represented using the ARPABET phonetic notation
- Common pronunciation variants for words with multiple accepted pronunciations
- Machine-readable format suitable for computational linguistics applications

We access CMUdict through the Natural Language Toolkit (NLTK), which provides a convenient Python interface:

```
import nltk
from nltk.corpus import cmudict
nltk.download('cmudict', quiet=True)
cmu_entries = cmudict.entries()
```

2.2 Vocabulary Construction

We construct separate vocabularies for graphemes (characters) and phonemes, each with appropriate special tokens:

Each vocabulary includes:

- **Character Vocabulary:** $|\mathcal{V}_{\text{char}}| \approx 29$ tokens (a-z, apostrophe, hyphen, etc.)
- **Phoneme Vocabulary:** $|\mathcal{V}_{\text{ph}}| \approx 72$ tokens (ARPABET phonemes)
- **Special Tokens:**
  - `<pad>`: Padding token (index 0) for batch processing
  - `<blank>`: Blank symbol required by the RNN-Transducer algorithm

The mappings between tokens and their numerical IDs are stored in dictionaries to enable efficient encoding and decoding:

$$\begin{aligned} \text{char2id} &: \mathcal{C} \to \mathbb{N} \\ \text{id2char} &: \mathbb{N} \to \mathcal{C} \end{aligned} \tag{1}$$

2.3 Preprocessing Steps

- Extraction of (grapheme, phoneme) pairs from the CMUdict corpus.
- Construction of vocabularies for characters and phonemes, including special tokens for padding and blank.
- Conversion of words and pronunciations into sequences of integer IDs.
- Splitting into training and testing sets (80/20 split).
- Batching and padding using PyTorch DataLoader and collate functions.

3 Model Architecture

3.1 RNN-Transducer Overview

Our G2P model follows the RNN-Transducer (RNN-T) architecture, which provides a framework for mapping variable-length input sequences to variable-length output sequences without requiring explicit alignment. Figure 1 illustrates the complete architecture.

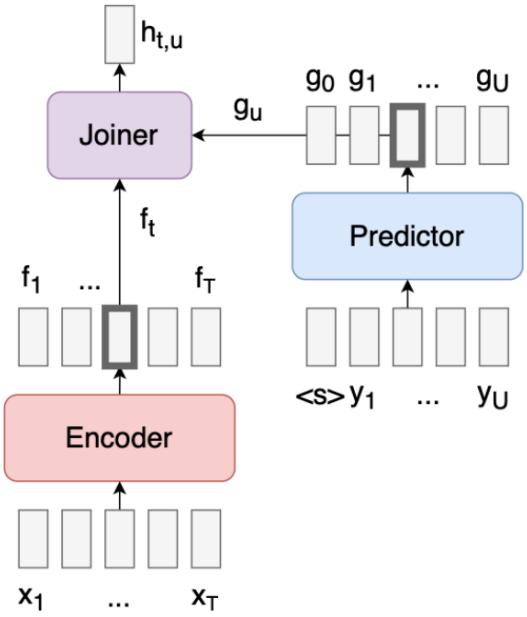

Figure 1: RNN-Transducer architecture showing data flow between Encoder, Predictor, and Joiner components. The input grapheme sequence $(x_1, ..., x_T)$ is processed by the Encoder to produce vectors $(f_1, ..., f_T)$. The Predictor processes previous phoneme outputs $(< s >, y_1, ..., y_U)$ to generate context vectors $(g_0, g_1, ..., g_U)$. The Joiner combines these representations at each alignment point (t, u) to produce output logits $h_{t,u}$.

3.2 Component Details

3.2.1 Encoder

The Encoder transforms the input grapheme sequence into a higher-level representation:

$$f_t = \text{Encoder}(x_1, x_2, \ldots, x_T)_t \tag{2}$$

Our implementation uses:

- **Embedding Layer:** Maps each character ID to a 64-dimensional dense vector

$$e_t^x = \text{Embedding}(x_t) \in \mathbb{R}^{64} \tag{3}$$

- **LSTM Layer:** Processes the embedded sequence to capture contextual information

$$f_t = \text{LSTM}(e_1^x, e_2^x, \ldots, e_T^x)_t \in \mathbb{R}^{128} \tag{4}$$

- **Packed Sequence Handling:** Efficiently processes variable-length inputs by ignoring padding tokens

Mathematically, the encoder produces a sequence of hidden states $f_1, f_2, \ldots, f_T$, where each $f_t \in \mathbb{R}^{128}$ represents the contextual encoding of the input up to position t.

3.2.2 Predictor

The Predictor models the probability distribution of the next phoneme given previous phonemes:

$$g_u = \text{Predictor}(< \text{blank} >, y_1, y_2, \ldots, y_{u-1}) \tag{5}$$

Implementation details:

- **Embedding Layer:** Maps each phoneme ID to a 64-dimensional vector

$$e_u^y = \text{Embedding}(y_u) \in \mathbb{R}^{64} \tag{6}$$

- **LSTM Layer:** Captures sequential dependencies in phoneme history

$$g_u = \text{LSTM}(e_{\text{blank}}^y, e_1^y, e_2^y, \ldots, e_{u-1}^y)_u \in \mathbb{R}^{128} \tag{7}$$

- **Special handling:** Prepends a blank token to begin prediction

The predictor's output g_u represents the context of previously generated phonemes, used to inform the prediction of the next phoneme.

3.2.3 Joiner

The Joiner integrates information from both the Encoder and Predictor to produce phoneme predictions:

$$h_{t,u} = \text{Joiner}(f_t, g_u) \tag{8}$$

Our implementation:

- **Projection Layers:** Transform encoder and predictor outputs to a common dimension

$$f_t' = W_e f_t + b_e \in \mathbb{R}^{256} \tag{9}$$

$$g_u' = W_p g_u + b_p \in \mathbb{R}^{256} \tag{10}$$

- **Activation:** Applies nonlinearity to combined representations

$$j_{t,u} = \text{ReLU}(f_t' + g_u') \in \mathbb{R}^{256} \tag{11}$$

- **Output Layer:** Maps to phoneme vocabulary space

$$h_{t,u} = W_o j_{t,u} + b_o \in \mathbb{R}^{|\mathcal{V}_{\text{phoneme}}|} \tag{12}$$

The final output $h_{t,u}$ represents unnormalized scores (logits) over the phoneme vocabulary, including a special blank symbol used during alignment.

3.3 Forward Pass

During training, the forward pass computes alignments between all possible pairs of encoder timesteps and predictor states:

1. Encode input sequence: $f_{1:T} = \text{Encoder}(x_{1:T})$
2. Initialize prediction with blank: $y_0 = \text{¡blank¿}$
3. Compute predictor outputs: $g_{0:U} = \text{Predictor}(y_{0:U-1})$
4. For each alignment point $(t, u) \in [1, T] \times [0, U]$:
   - Compute joiner output: $h_{t,u} = \text{Joiner}(f_t, g_u)$
   - Convert to probabilities: $P(k|t, u) = \text{softmax}(h_{t,u})$
5. Compute RNN-T loss over all valid alignment paths

During inference, we use beam search with width 3 to efficiently search the alignment space and find the most likely phoneme sequence given the input graphemes.

4 Training Details

4.1 Loss Function

The RNN-Transducer (RNN-T) loss from `torchaudio` is employed for training. This loss function computes the negative log probability of target sequences given input sequences:

$$\mathcal{L}_{\text{RNN-T}} = -\log P(\mathbf{y}|\mathbf{x}) \tag{13}$$

where $P(\mathbf{y}|\mathbf{x})$ is computed by summing probabilities over all valid alignments between input and output sequences. The implementation involves:

- Log-softmax normalization: $\log p_{t,u}(k) = \log \frac{\exp(z_{t,u,k})}{\sum_{k'} \exp(z_{t,u,k'})}$
- Forward-backward algorithm to efficiently compute alignment probabilities
- Special handling of the blank symbol (`<blank>`) at position $ph2id[\texttt{<blank>}]$
- Mean reduction across the batch for stable gradients

4.2 Optimization Strategy

- **Optimizer:** Adam with default parameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$)
- **Learning Rate:** 1×10^{-3} (fixed throughout training)
- **Batch Size:** 64 samples per batch with dynamic padding
- **Gradient Computation:** Full backpropagation through encoder, predictor, and joiner
- **Update Schedule:** Parameter updates after each batch
- **Epochs:** 5 complete passes through the training dataset
- **No Regularization:** No explicit weight decay or dropout

4.3 Training Process

The model is trained end-to-end using the following procedure:

for epoch = 1 to 5 **do**
    Set model to training mode
    for each batch (x, x_lens, y, y_lens) in training data **do**
        Forward pass: $logits = model(x, x\_lens, y, y\_lens)$
        Compute loss: $loss = rnnt\_loss(logits, y, x\_lens, y\_lens)$
        Backward pass: compute gradients
        Update parameters using Adam optimizer
    end for
    Evaluate CER on training set
    Evaluate CER on test set
end for

5 Prediction and Evaluation

5.1 Prediction Process

Our evaluation methodology follows these steps:

1. Set the model to evaluation mode (disabling dropout and other training-specific behaviors)
2. For each sample in the evaluation dataset:
    (a) Encode the input grapheme sequence using the encoder
    (b) Perform beam search decoding with width 3 to generate the phoneme sequence
    (c) Convert phoneme IDs back to phoneme symbols using the phoneme vocabulary

5.2 Evaluation Metric

Character Error Rate (CER)

$$\text{CER} = \frac{\text{EditDistance}(reference, hypothesis)}{\text{length}(reference)} \tag{14}$$

where EditDistance is the Levenshtein distance counting substitutions, insertions, and deletions.

5.3 Evaluation Process

Once we got the prediction:

1. Calculate CER between the predicted and reference phoneme sequences
2. Average the CER across all evaluation samples

6 Results

6.1 Results Analysis

We evaluate our RNN-Transducer model's performance on both training and test sets to assess generalization:

Model	Training CER	Test CER
RNN-Transducer	13.6%	14.4%

Table 1: Character Error Rate (CER) results on CMU Dictionary dataset

6.2 Error Analysis

To better understand model limitations, we analyzed the types of errors made by the RNN-Transducer:

- **Vowel confusions:** Most common errors involve similar-sounding vowels (e.g., predicting "AH" instead of "AE")
- **Silent letter handling:** Difficulty determining when letters should not be pronounced
- **Consonant clusters:** Errors in complex grapheme-to-phoneme mappings like "th" $\rightarrow$ "TH" or "ph" $\rightarrow$ "F"

This analysis suggests that future improvements could focus on better modeling of English phonological rules and context-dependent pronunciations.

7 Conclusion

We demonstrate that an RNN-Transducer can effectively model English grapheme-to-phoneme conversion using the CMU Pronouncing Dictionary. The model handles variable-length input and output sequences, and achieves strong performance as measured by CER. Future work may explore transformer-based models, attention mechanisms, or multilingual G2P.

Acknowledgments

We thank our course instructor and teaching assistants for their guidance, as well as the maintainers of the CMU Pronouncing Dictionary and the open-source software used in this project.

References

1. Transducer tutorial github repository. `https://github.com/lorenlugosch/transducer-tutorial/tree/main`
2. Loren Lugosch "Sequence-to-sequence learning with Transducers " *Interspeech*, 2022. `https://www.isca-archive.org/interspeech_2022/fukuda22_interspeech.pdf`

3. "CMU Pronouncing Dictionary," Carnegie Mellon University. https://en.wikipedia.org/wiki/CMU_Pronouncing_Dictionary

4. J. Jiang, Y. Zhang, S. Wang, et al., "Pretraining Enhanced RNN Transducer," *Artificial Intelligence Research*, 2024. https://www.sciopen.com/article/10.26599/AIR.2024.9150039

5. H. Kang and K. Choi, "An Ensemble of Grapheme and Phoneme for Machine Transliteration," *Proceedings of the International Conference on Natural Language Processing*, 2005. https://aclanthology.org/I05-1040.pdf