Exploring Hybrid Language Models for Morphologically Rich Languages

Yash Shah, Preethi Jyothi
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
{yashshah,pjyothi}@cse.iitb.ac.in

Abstract

Word and character-level deep learning language models have been used extensively, but each one independently suffers from several drawbacks — word-level LMs are not robust to presence of OoV words; and although character-level LMs partially solve this issue, they often fail to capture long range context dependencies. Hybrid LMs seem to capture the best of both LMs, by using subword information to construct vector representations for words which are then used for modelling the language. In this seminar report, we explore one such model in detail and suggest variants that try to overcome some of its shortcomings. We leave their implementation and analysis for future work.

1 Introduction

A language model assigns probabilities to sequences of tokens sampled from a vocabulary; a high probability implying that that sequence is more likely to appear in the language. These tokens are *words* for a word-level LM and *characters* for a character-level LM. Mathematically,

$$s = \{t_i\}_{i=1}^{m}, \quad t_i \in \mathcal{V}$$

$$s \xrightarrow{\text{LM}} P(s), \quad P(s) \in [0, 1]$$

Since the order in which these tokens occur is significant, this probability is expressed as a product of conditional probabilities — each term representing the occurence probability of a token given that all previous tokens have been encountered.

$$P(\{t_i\}_{i=1}^{m}) = \prod_{i=1}^{m} P\left(t_i | t_{i-1}, t_{i-2}, ...t_1\right)$$

If we neglect the contribution of distant tokens in the sequence, the conditional probabilities can be relaxed so that they consider only the $n-1$ previously encountered tokens for any given token, leading to the more common n-gram LM definition:

$$P_n(\{t_i\}_{i=1}^{m}) = \prod_{i=1}^{m} P\left(t_i | t_{i-1}, t_{i-2}, ...t_{i-n-1}\right)$$

Deep learning based neural language models try to mimic these conditional probabilities by learning dependencies amongst tokens from a given training corpus. Common word-level language models construct a finite-sized vocabulary from the training data and gradually modify the parameters based on contextual information — each occurrence of a word token in training data contributes to the estimate of a word vector assigned to this word type. Due to this, low-frequency and unseen words are left with incorrect estimates leading to poor performance. This is especially the case with *agglutinative* languages like Tamil and Kannada, where tokens can keep combining meaningfully with each other, leading to a seemingly infinite vocabulary.

Character-level language models were introduced to tackle this problem since most of the out-of-vocabulary words could still be expressed in terms of the characters in vocabulary. The arrangement of characters within a word carries significant information which could be used for making reasonable estimates for OoV word vectors. This subword-level information can be captured in different tokenization units, such as characters, ngrams, morphemes or phonemes. While ngrams are simple, fix-sized groupings of characters, morphemes and phonemes are groups of structurally/phonetically relevant characters. Since finding accurate morpheme/phoneme representa-

tions for every word in the training corpus is another task altogether, most models use characters and character ngrams for extracting subword level information.

An important model for this task is the char-CNN-LSTM, which uses convolutional nets with varying kernel sizes to combine character vectors for obtaining the vector representation of the corresponding word. This is beneficial since convolutions are parallelizable, as well as they implicitly represent operations on ngram clusters (for kernels of width n). These generated word vectors are then fed as input to a traditional LSTM network for learning global information.

Such models that make use of both subword and word-level information for learning are referred to as *hybrid* language models. char-CNN-LSTM is one such model aiming to combine information at the *input* side of LMs. In this seminar, we explore the work of Gerz et al. on the same, and suggest variants for overcoming certain shortcomings of their model.

2 Literature Survey

Our work is basically an analysis and extension of that by Gerz et al., who used a char-CNN-LSTM with a novel objective function for modelling 50 languages of varying type. We describe the important aspects of their model here.

2.1 Constructing Vocabulary

Word vocabulary was constructed by considering character clusters separated by whitespaces as words. Additional tokens, namely `<unk>` for OoV words, `<s>` for start-of-sentence and `</s>` for end-of-sentence, were also introduced. Character vocabulary was constructed by including every character in the training data. An additional token `<unk>` was also added in this vocabulary for handling unseen characters at test time. Start-of-word (`<w>`) and end-of-word (`</w>`) tokens were also added to the vocabulary (we present our results both with and without including them).

2.2 Vector Spaces

Three different vector spaces or *embeddings* were created — a character embedding, a character-aware input word embedding and an output word embedding. The two word embeddings were independent of each other; the input embedding was used for storing the word vectors generated using characters' vector representations, while the output embedding was used for next-word prediction. Dimensions of constituent vectors were d_c, $d_{w_{in}}$ and $d_{w_{out}}$ respectively.

2.3 Word Vector Generation

For a word of length w, vectors corresponding to the w constituent characters were first extracted from the character embedding, and concatenated sideways to form a $d_c \times w$ matrix. This matrix was then padded with $M - w$ columns of d_c dimensional zero vectors (here M is the maximum word length for the entire training corpus) to form a $d_c \times M$ dimensional matrix, whose size would be the same for every word.

This word matrix was then convolved with filters of widths $\{k_i\}_{i=1}^{f}$ and depths $\{d_i\}_{i=1}^{f}$ to generate $d_i \times (M - k_i + 1)$ dimensional outputs. Each column of this output represented the response of the corresponding k_i-gram character cluster to the kernel weight, with later columns depicting responses at later *time*. These output matrices were max-pooled along the *time* dimension to get $d_i \times 1$ vectors, which were concatenated columnwise to generate a large $\left(\sum_{i=1}^{f} d_i\right) \times 1$ word vector. These vectors filled the character-aware input word embedding, and so $d_{w_{in}} = \sum_{i=1}^{f} d_i$.

2.4 LSTM Network

The generated word vector, y_{w_t} is first passed to a 2-layer *highway network* (it is essentially a feed-forward network with skip connections) for undergoing transformation. The transformed word vector, h_{w_t}, is then passed on to the 2-layer LSTM network. The network yields one output vector o_{w_t} per word in the sequence, given all previous time steps $[h_{w_1}, ...h_{w_{t-1}}]$. To predict the next word w_{t+1}, one takes the dot product of the vector o_{w_t} with the output word embedding to get a vector p_t. It is normalized to contain values between 0 and 1, representing a probability distribution over the next word. This corresponds to calculating the softmax function for every word in $\mathcal{V}_w$.

$$
\begin{aligned}
h_{w_t} &= \text{highway}(y_{w_t}) \\
o_{w_t} &= \text{LSTM}(h_{w_t} \mid h_{w_{t-1}} \ldots h_{w_1}) \\
\hat{p}_t &= o_{w_t} \cdot \mathcal{V}_{w_{out}, w_t} \\
p_{t,w_i} &= \frac{\exp(\hat{p}_{t,w_i})}{\sum_{w \in \mathcal{V}} \exp(\hat{p}_{t,w})}
\end{aligned}
$$

2.5 Objective Function

The model used two different objective functions, each one for a separate purpose.

2.5.1 Training

Standard cross-entropy loss between the output of each time step p_t, and the target one-hot distribution, was used during training.

$$\mathcal{L}_{CE}(t) = -\log(p_{t,w_{t+1}})$$

2.5.2 Fine-tuning

Since the output word embedding did not learn from subword information (since it was trained specifically for single words), it lead to unreliable estimates for infrequent words. Also, a character-only model is not able to capture word-level semantics efficiently and thus degrades performance. In order to tackle this, the authors injected shared subword semantics (captured in the character-aware input word embedding) into the output word embedding to additionally reflect shared subword-level information, which should hopefully lead to improved word vector estimates.

This was done by *fine-tuning* training after every epoch of *normal* training. During fine-tuning, for every *eligible* cue word $w \in \mathcal{V}_w$, a set of η_p positive samples (P_w) and a set of η_n negative samples (N_w) was constructed. A vocabulary word was considered *eligible* if it occurred greater than T times in the training corpus. This thresholding step ensured that rare words didn't disrupt word vector estimates. Positive samples were words having the most similar vector representations to a given cue word, based on *cosine* similarity. Negative samples were randomly picked from the vocabulary.

Once these sets had been constructed, the fine-tuning loss was calculated using the "output word embedding vectors of the chosen words" in two steps. The *attract* term tried to pull similar words together and push different ones further apart. The *preserve* term tried to keep the updated word vector close to its original value. Thus,

$$\mathcal{L}_A(w) = \sum_{w_p \in P_w} \sum_{w_n \in N_w} \max(\delta + \mathcal{V}_{w_{out},w} \cdot \mathcal{V}_{w_{out},w_n} - \mathcal{V}_{w_{out},w} \cdot \mathcal{V}_{w_{out},w_p}, 0)$$

$$\mathcal{L}_P(w) = \lambda \cdot ||\mathcal{V}_{w_{out},w} - \hat{\mathcal{V}}_{w_{out},w}||^2$$

$$\mathcal{L}_{AP} = \sum_{w \in \mathcal{V}_{eligible}} (\mathcal{L}_A(w) + \mathcal{L}_P(w))$$

d_c	15
$d_{w_{out}}$	650
$d_{w_{in}}$	1100
f	7
$\{k_i\}_{i=1}^{f}$	{1, 2, 3, 4, 5, 6, 7}
$\{d_i\}_{i=1}^{f}$	{50, 100, 150, 200, 200, 200, 200}
M	computed separately for each training corpus
Dropout value	0.5
Learning rate	1.0
Learning rate decay	0.5
Parameter init	rand uniform [-0.05, 0.05]
Batch size	20
RNN sequence length	35
Max grad norm	5.0
Max epochs	15 or 30
δ	0.6
(η_n, η_p)	(3, 3)
AP learning rate	0.05
AP gradient clip	2
λ	10^{-9}
T	5

Table 1: Model and experimentation parameters

2.6 Optimization

SGD optimization algorithm was used for *normal* training, while AdaGrad was used for fine-tuning.

2.7 Results

The authors obtained notable results on most of the 50 languages. Their values for three Indian languages, namely Hindi (`Hi`), Tamil (`Ta`) and Kannada (`Ka`), are provided in table 2.

Lang.	Baseline	w/o AP	w/ AP
Hi	426	326	299
Ta	6234	3496	2768
Ka	5310	2558	2265

Table 2: PPL values for three Indian languages (*baseline* refers to standard LSTM network with similar parameters, *w/o AP* refers to char-CNN-LSTM model without fine-tuning)

3 Experiments

Since the authors had not released their code by the time we started, we re-implemented their model ourselves in Python3 using TensorFlow. There were some parameters whose values had been left unreported, as well as some details regarding the model that remained unclear. We made assumptions regarding the same, which are stated in table 3.

Detail	Assumption
Number of units in an LSTM cell	400
Number of iterations during fine-tuning	250
Manner of preparing batches	First treated all tokens independently; then made sentence-wise batches
Whether to include start- and end-of-word tokens	Experimented with both

Table 3: Assumptions made during implementation

We initially started experimenting by considering all tokens (words) as independent units i.e. we were concerned with the correct order only within the window of 35 time steps — preservance of order or LSTM state was not guaranteed outside this window. This assumption had made implementation easier, but was fundamentally flawed. Our $\mathcal{L}_{CE}$ and $\mathcal{L}_{AP}$ loss values were much higher after 30 epochs, as shown in figures 1 and 2.

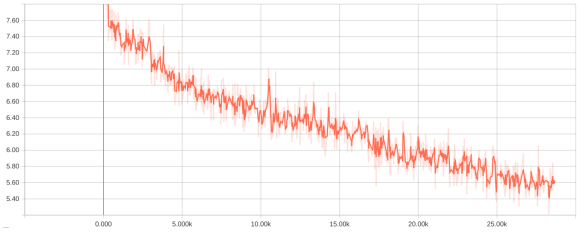

Figure 1: Variation in CE loss with iterations for independent token assumption

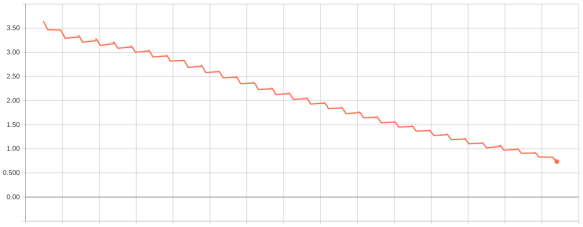

Figure 2: Variation in AP loss with iterations for independent token assumption

3.1 Effect of sentence-wise batches and start-/end-of-word tokens

In order to incorporate this, we had to add a `<pad>` token to the word vocabulary and start- (`<w>`) and end- (`</w>`) of-word tokens to the character vocabulary. Every word was prepended with `<w>` and appended with `</w>` before being sent as input to the char-CNN. Each of the 20 *channels* in a batch was reserved for a particular sentence, and was allotted to the next one only when the `</s>` token for the former had been encountered. If a sentence terminated before 35 timesteps, it was padded with the `<pad>` token on both the input and output side. Whenever a *channel* was switched between sentences, the corresponding LSTM states were flushed. This was done so that information of the previous sentence did not affect the predictions for the current one. We were able to push the loss values down, as shown in figures 3 and 4.

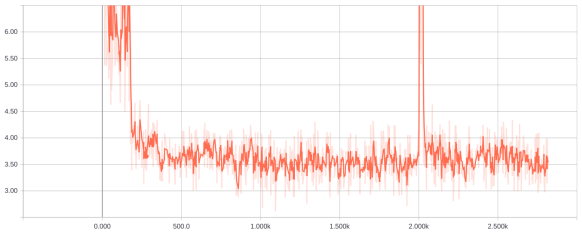

Figure 3: Variation in CE loss with iterations for sentence-wise batches

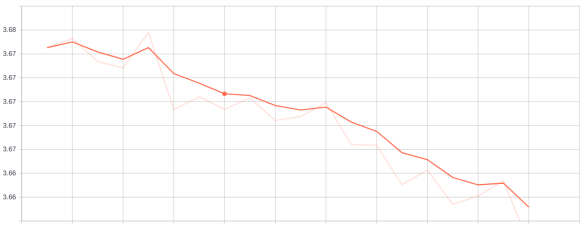

Figure 4: Variation in AP loss with iterations for sentence-wise batches

3.2 Effect of fine-tuning

In order to understand and verify the effect of fine-tuning, we tracked the CE loss values across iterations (for the independent token assumption) both with and without including AP loss, and noticed that fine-tuning did help the model to learn better (see figure 5 and 6).

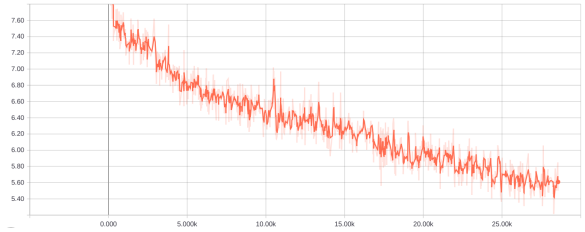

Figure 5: Variation in CE loss with iterations with fine-tuning

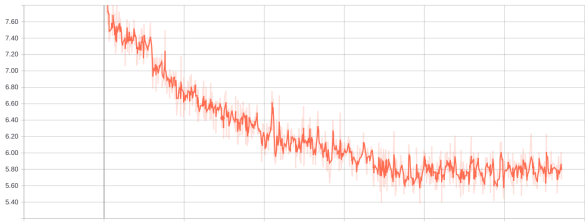

Figure 6: Variation in CE loss with iterations without fine-tuning

4 Suggestions and Future Work

Upon detailed analysis of Gerz et al's model, we identified certain aspects that could be improved upon to possibly get better performance. They are mentioned below:

- Word vectors were generated using only *local* information (character sequence within a word) and local transformations
- The pooling operation (max-over-time) completely ignored the activations/responses at non-maximal time instances, due to which important information was possibly lost
- It is possible that the information captured by the two word embeddings (input and output) is redundant

We now suggest alternative variants that try to address some of these issues, and intuitively explain how. Their implementation and analysis is left for future work.

4.1 Introduce attended-pooling

It is trivial to realize that *max-over-time* pooling leads to loss of information captured in non-maximal time instances. Instead of using the current *one-hot* "attention" scheme (with the maximal time instance getting full attention), we could use a more *distributed* one. The amount of emphasis given to different character clusters depends on both the tokens encountered in close vicinity, as well as the overall global context. Both these factors can be incorporated by introducing an *attention window* that uses the generated word vectors of last k words (local context), and LSTM states of the previous timestep (global context) to produce an attention distribution over M columns for each kernel size, $\{k_i\}_{i=1}^{f}$ (see figure 7).

$$e_t = W_1 \cdot C_{1,t-1} + W_2 \cdot C_{2,t-1} + \sum_{i=1}^{k} W_{vi} \mathcal{V}_{w_{in},w_{t-k}} + b_W$$

$$\hat{e}_{t,j} = \frac{\exp(e_{t,j})}{\sum_{i=1}^{M} \exp(e_{t,i})}$$

$$y_{w_t} = \sum_{i=1}^{M} \hat{e}_{t,i} \cdot \mathcal{V}_{c,w_{t,i}}$$

Here $C_{k,t}$ is the state of k^{th} LSTM layer after t^{th} timestep, W_α are weights, $\mathcal{V}_c$ and $\mathcal{V}_{w_{in}}$ are character and input word embeddings and $w_{t,i}$ is the i^{th} character of the input word at time t. The final word vector is obtained by concatenating the y_{w_t}'s of all kernel sizes columnwise.

4.2 Allow external factors to affect transformations

The highway network of the current implementation uses only *local* information (i.e. bounded within character clusters of the word) for producing linear transformations (scaling and translation) in the generated word vector. Ideally, this transformation should be affected by external context too, which is not possible in the present model. Replacing the highway network by a *combination* window can possible solve this — it takes as input the final word vector, y_{w_t}, as well as history of last k word vectors and LSTM states of previous timestep, to generate a *scaling* and *offset* vector.

$$s_t = Q_1 \cdot C_{1,t-1} + Q_2 \cdot C_{2,t-1} + \sum_{i=1}^{k} Q_{vi} \mathcal{V}_{w_{in},w_{t-k}} + b_Q$$

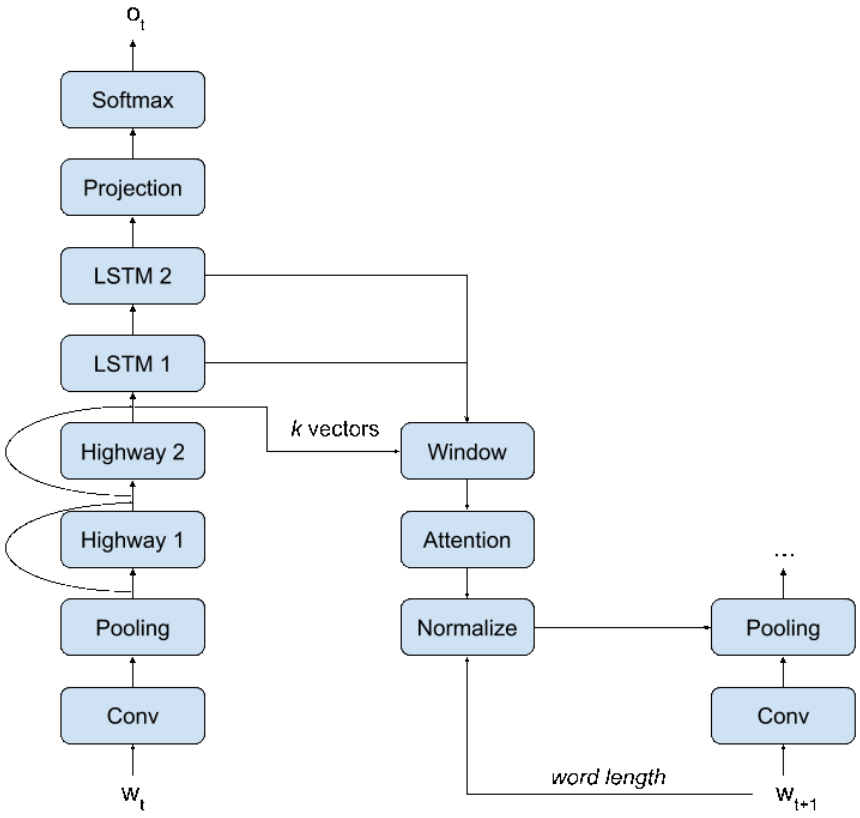

Figure 7: Introducing an attention scheme as a *pooling* window

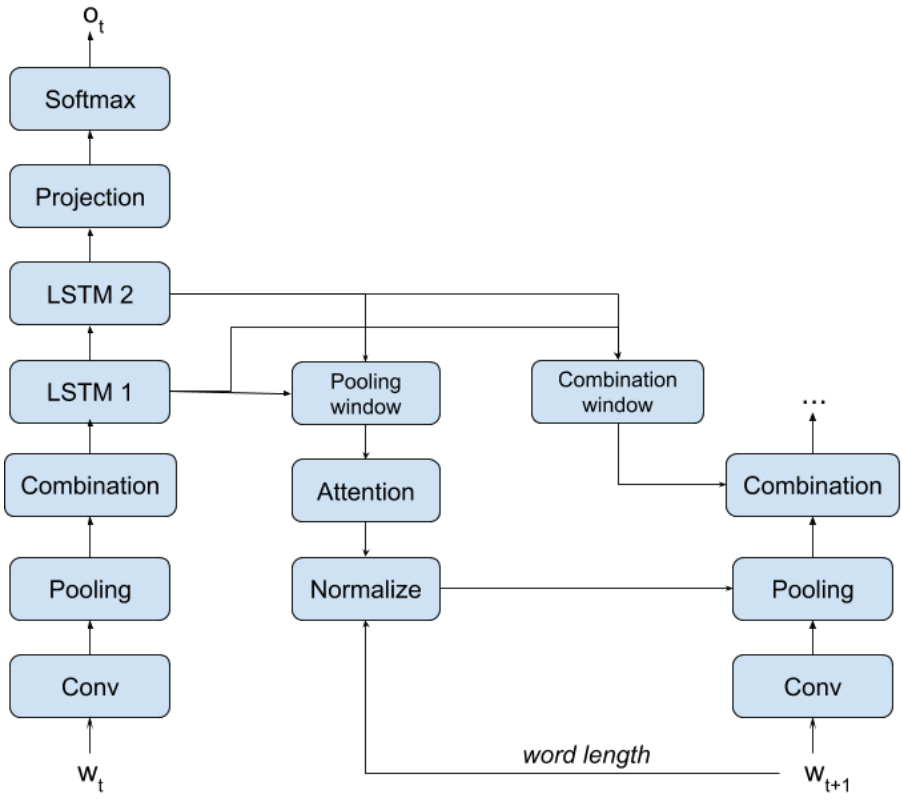

Figure 8: Replacing highway network with a *combination* window

$$b_t = R_1 \cdot C_{1,t-1} + R_2 \cdot C_{2,t-1} + \sum_{i=1}^{k} R_{vi} \mathcal{V}_{w_{in}, w_{t-k}} + b_R$$

$$\hat{y}_{w_t} = s_t \cdot y_{w_t} + b_t$$

4.3 Ensure orthogonality of word embeddings

As mentioned earlier, there is a possibility that the information captured by the two word embeddings is quite similar, leading to redundancy. This can be avoided if we ensure that these embeddings remain orthogonal to each other i.e. the inner product of corresponding word vectors is as low as possible. An easy way to ensure this is adding another term to the total loss

$$\mathcal{L}_{ortho}(\mathcal{V}_{w_{in}}, \mathcal{V}_{w_{out}}) = \sum_{w \in \mathcal{V}} \mathcal{V}^T_{w_{in}, w} \cdot \mathcal{V}_{w_{out}, w}$$

Minimization of this loss function is essentially minimizing the inner product of the two vector representations for each word — this is same as moving towards orthogonality of vector spaces, since for orthogonal vectors p and q, $p^T \cdot q = \mathbf{0}$.

4.4 Construct shared embeddings

The total count of distinct ngrams for a given corpus is very large. However, their frequency distribution is highly skewed, with only a few of these ngrams occurring in significant amounts (see figures 9 and 10).

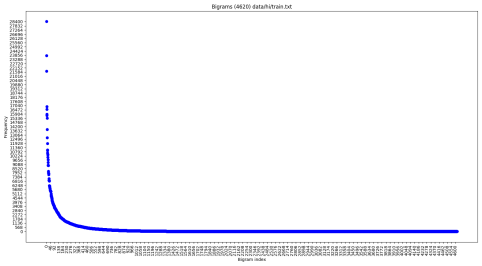

Figure 9: Frequency distribution of bigrams for `Hi` language

Since the character sequence constituting these frequent ngrams is so common, we could treat them as independent tokens and assign a vector representation for the cluster as a whole instead of constructing it from constituent characters. This is beneficial because those characters possess more information and meaning as a group than they do so individually. This gives rise to the notion

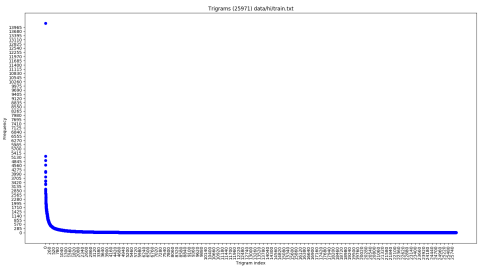

Figure 10: Frequency distribution of trigrams for `Hi` language

of *shared* embeddings — a vector space consisting of representations of words and such frequent ngrams.

References

[1] *Language Modeling for Morphologically Rich Languages: Character-Aware Modeling for Word-Level Prediction*, Daniela Gerz, Ivan Vuli, Edoardo Ponti, Jason Naradowsky, Roi Reichart and Anna Korhonen, TACL 2018.

[2] *Character-Aware Neural Language Models*, Yoon Kim, Yacine Jernite, David A Sontag, Alexander M. Rush, AAAI 2016.