Project 3: Sequence Modeling

CSE 849 Deep Learning (Spring 2025)

Gautam Sreekumar
Instructor: Zijun Cui

March 22, 2025

In this project, you will use some popular sequence modeling architectures for translation and classification tasks. This project will require GPUs and may need considerable time from your end.

1 Review Rating Prediction

Websites such as Yelp and Amazon allow users to leave public reviews about restaurants and products along with a numerical rating, typically between 1 and 5 "stars". In this question, we will train a model on review-rating data to predict the associated rating from the review text. We will model the review text as a sequence of words using an RNN. Our model will output a single representation for a given review text, which we will feed to a linear classifier to predict the rating. The architecture to follow is shown in Fig. 1.

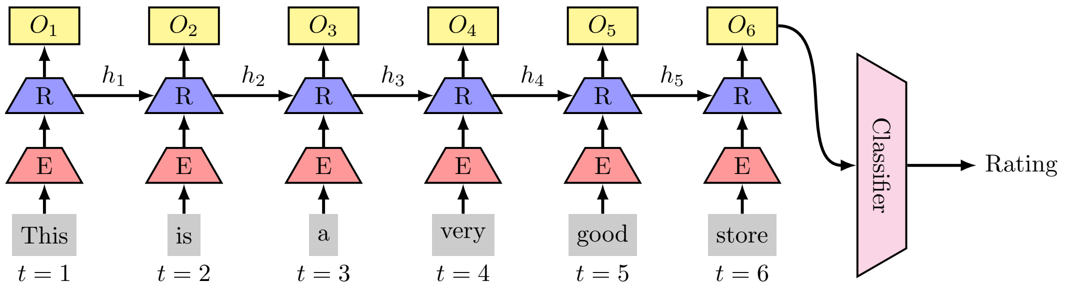


Figure 1: **RNN-based Architecture for Rating Prediction**: Suppose the input review text is "This is a very good store". You pass it through the embedding layer to get the GloVe vector representation of each word. The vector representation at each time step is passed through the RNN model with 2 layers. The model will output two vectors at time step t: hidden representation (h_t) and output vector (O_t). The hidden vector h_t at time step t is used in the computation of h_{t+1}. Note that the PyTorch implementation will give you output and hidden vectors for all time steps together. We will feed the last output vector (O_6 in the illustration) to the classifier layer, which will then output the rating. Here, both the RNN model and the embedding layer are trained.

Data We are provided access to train, validation, and test sets, each consisting of review texts and their corresponding ratings (except the test set). These samples were obtained by processing a subset of the Yelp Open dataset [Yel25].

Encoding the review words We process each review as a sequence of words, and each word in the review text must be converted into vectors that the model can understand. These vectors must ideally preserve the semantic relations between the words. For example, the vector representations of words corresponding to various colors such as "blue" and "red" must be closer to each other than to words corresponding to, say, animals such as "lion" and "tiger". Word embeddings are trained on large text corpora to minimize the distance between embeddings of the words that appear together in the training data. We will use one such word embedding, GloVe [PSM14], to encode the words in the review text. GloVe embeddings have 400K words in its vocabulary but our dataset requires a smaller vocabulary ($\approx$ 26K words). Therefore, we will use a subset of the 50-dimensional GloVe embeddings to save our resources (provided as `glove/modified_glove_50d.pt` in the project zip file).

Fine-tuning the embeddings The original GloVe embeddings were trained on large text corpora. It is safe to assume that these embeddings are not optimal for our task of rating prediction. Therefore, we will fine-tune these embeddings along with our model. To fine-tune the GloVe embeddings, we will create an instance of `nn.Embedding` using its `pre_trained` method. This embedding layer contains the weights that map from the vocabulary space to the embedding space. We will use the GloVe embeddings to initialize the weights in the `nn.Embedding` instance, and these weights will be updated during the training. Make sure to set the freeze argument to False.

Creating the dataloader Sequence modeling is unique in a sense due to the varying input size. In this particular problem, the reviews will vary in length[1] and the model is expected to handle this. However, we must write a custom collate function for our dataloader to process batches of reviews of non-uniform length on GPUs efficiently. Make sure to read Sec. 3 and check Fig. 2 for a detailed explanation.

Your dataloader should provide a batch of padded sequences of embeddings with each embedding corresponding to a word in the review text. The embeddings are obtained by passing the vocabulary indices[2] of the words in the review text to the embedding instance created from the 50-dimensional GloVe embedding. Note that the output of the collate function should be a packed sequence of embeddings — not word indices. If you first collect word indices and pack them, the embedding layer will fail, as it does not accept a packed sequence. Therefore, you should first convert word indices to embeddings inside the collate function, then pack the embeddings before passing them to the RNN. More tips can be found in Sec. 3.

Architecture For this task, you will use an RNN with 2 layers. The dimensions of the output and the hidden vectors will be 50, the same as the word embeddings. You will feed the output vector into a linear classifier that will provide you with prediction logits. The embeddings, the RNN, and the classifier will be jointly trained using a cross-entropy loss. Refer to Fig. 1. Make sure you include the parameters from all three modules in your optimizer. Save the checkpoints corresponding to the embeddings, the RNN and the classifier. See Sec. 3 for training tips.

2 Translate English Sentences to Pig Latin

2.1 What is Pig Latin?

Pig Latin is a language game where regular English words are modified according to some rules to form new words that would seem gibberish to those who are not familiar with the rules.

Rules to convert English words to Pig Latin Some of the following rules are taken verbatim from Wikipedia.

1. Every word is processed individually and independently.
2. The initial consonant blend (or two letters) is moved to the end of the word, then “ay” is added. E.g., “pig” → “igpay” and “banana” → ”ananabay”.
3. When words begin with consonant clusters (multiple consonants that form one sound), the whole sound is moved to the end (before adding ”ay”). E.g., “friends” → “iendsfray”, “smile” → “ilesmay”, and “schedule” → “eduleschay”.

[1]For this assignment, we have limited the maximum number of words in a review to 40. In some cases, the reviews were abruptly truncated.

[2]The vocabulary can be built from the modified GloVe embeddings provided in the project zip file.

4. For words that begin with vowel sounds, “way” is added to the end without any changes to the remainder of the word. E.g., “eat” → “eatway”, “omelet” → “omeletway”, “are” → “areway”.
5. Characters other than alphabets are left unchanged.

You can use this website to easily compute the Pig Latin word for a given English word. Some examples of complete English sentences converted to Pig Latin are shown in Tab. 1.

English	**Pig Latin**
humpty dumpty sat on a wall. humpty dumpty had a great fall.	umptyhay umptyday atsay onway away allway. umptyhay umptyday adhay away eatgray allfay.
it was the best of times, it was the blurst of times?!!	itway asway ethay estbay ofway imestay, itway asway ethay urstblay ofway imestay?!!
harry, yer a wizard...	arryhay, eryay away izardway...

Table 1: Examples of translations to Pig Latin

In this project, you will train a Transformer [Vas+17] to translate English to Pig Latin.

2.2 Translation to Pig Latin using Transformers

Transformers model sequences using multi-headed self-attention layers. In your task, the inputs will be sequences of alphabets and space characters. Our token vocabulary will also include three special tokens: start-of-sentence (<SOS>) to denote the start of a sentence, end-of-sentence (<EOS>) to denote the end of a sentence, and padding (<PAD>) to ensure all inputs in a batch have the same length. We will not include any digits, special characters, or punctuation.

Data Training, validation, and test splits are provided in the project zip file. The sentences in the dataset are obtained by processing Amazon product review data [Hou+24].

How to encode the input characters? As mentioned before, both input and output sequences consist of alphabets, space characters, <SOS> and <EOS>. Together, these **30** characters (counting only lowercase) will constitute our *vocabulary*. To input these characters to the Transformer, we need to find a representation for each character that can be understood by the model. As with other things in deep learning, we will learn that representation.

We will use `nn.Embedding` to model the vector representations for each character. `nn.Embedding` contains a weight matrix of shape vocabulary size × embedding dimension. These weights are trained along with the model. For this project, set the embedding dimension to 100. After creating an instance of this embedding layer, to get the weight corresponding to a character, we will simply pass the index of that character to the embedding layer. For example, if the character “a” is assigned the index 1, then the vector representation corresponding to “a” is obtained as `embedding(1)`. You can obtain a concatenated sequence of embeddings by passing a tensor of indices to the embedding layer. Check this question and Sec. 3 for more details.

Positional Encoding Unlike RNNs, Transformers do not inherently consider the order of tokens in a sequence. That is, the sentences “this is an apple” and ”apple is this an” are equivalent. Therefore, we will use positional encodings to explicitly encode the position of each token in the sequence. You can read more about positional encodings here[3] and here.

Training vs Inference A Transformer model consists of encoder and decoder layers. Encoder layers compute the self-attention maps between the input tokens and compute the token embeddings for the succeeding layers. The encoder layers behave identically during training and inference. However, the decoder layers should behave differently during training and inference.

[3]This blog post also contains plenty of information about Transformers.

During inference, every token outputted by the decoder should use only the encoded information and the tokens that it has generated before. For example, to translate a sentence from English to German, we will input the sentence in English to the encoder, pass the computed tokenwise embeddings to the decoder, and pass <SOS> as the first token in the translated sentence. Now the decoder will output the next most-probable token. Call it <TOK$_1$>. To generate the token that follows <TOK$_1$>, we will again pass the input English sentence to the encoder and pass the sequence (<SOS>, <TOK$_1$>) to the decoder. In short, the generated tokens are used by the decoder to predict the next token. We will call it autoregressive generation.

As you may already realize, a mistake in generating the t^{th} token can affect the generation of subsequent tokens. Therefore, during training, we will feed the true translated tokens from the training dataset to the decoder and train the model to predict the next token at every time step. Thus, a prediction mistake at time step t will not affect the prediction at time steps $> t$.

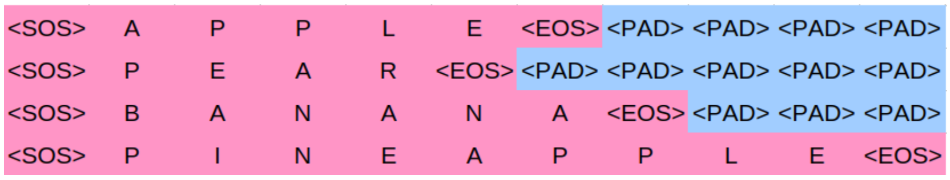
To ensure that the decoder does not predict the next tokens by looking at the future tokens, we will modify the attention masks in the decoder. Specifically, we will set the attention values prior to their normalization as $-\infty$. PyTorch allows us to do this by modifying `tgt_mask` and `tgt_is_causal` arguments. Check this method of the Transformer class to see how to create the target attention mask. Note that this is a static method.

How to decode the output characters? Similar to our encoding process, we need to convert the vector output to a character. For this purpose, we will train an additional linear layer that will map from the output embedding space to the vocabulary space. This will give us the logits corresponding to the characters in the vocabulary. We can then train our model using a combination of cross-entropy loss on the predicted characters and MSE loss of the predicted embedding.

Other implementation details You can use PyTorch implementation of Transformers with 2 encoder layers, 2 decoder layers, 2 multi-self-attention heads, and a feedforward dimension of 128. Set the embedding dimension to 100. You are free to choose other hyperparameter settings, such as your choice of optimizer, learning rate, number of epochs, etc. Refer to Sec. 3 for additional information.

3 Additional Information and Tips

1. **Input shape**: Sequences are of shape sequence length $\times$ input dim. Here, <EOS> and <SOS> are counted in "sequence length". When you batch these sequences, the expected shape is batch size $\times$ sequence length $\times$ input dim. Since the words have different lengths, we will need to pad each sequence when we batch the sequences. Therefore, the shape of a batch of sequences is batch size $\times$ max sequence length $\times$ input dim, where "max sequence length" is the length of the longest sequence in that batch. See Fig. 2 for a visualization. Although we use <PAD> to denote a padding token, we will simply pad it with zeros, which essentially means that we are assigning a zero vector as the embedding for the padding token without any training. You can use this PyTorch utility to pad inputs easily. Check the value taken by `batch_first` in the arguments list.



<SOS>	A	P	P	L	E	<EOS>	<PAD>	<PAD>	<PAD>	<PAD>
<SOS>	P	E	A	R	<EOS>	<PAD>	<PAD>	<PAD>	<PAD>	<PAD>
<SOS>	B	A	N	A	N	A	<EOS>	<PAD>	<PAD>	<PAD>
<SOS>	P	I	N	E	A	P	P	L	E	<EOS>

Figure 2: What does a padded sequence look like?

2. **Packing**: For an RNN architecture, you should pack the sequence after padding it to save the compute. PyTorch also provides `pack_sequence` to combine padding and packing. See this StackOverflow answer as to why you should pack and how to. Since you are packing the input sequence, you must unpack the output sequence as well.

3. **Collate function**: A dataloader employs several CPU workers to collect the data samples corresponding to various indices using the dataset object's `__getitem__` method. These samples then arrive at the dataloader which would then combine them to form a batch by the collate function. However, in problems like sequence modeling and object detection, we need to add our processing before a batch of samples is created from the individual samples. So in this assignment will write our own collate function that would pad the sequences for the translation task and pack the sequences for the prediction task. See this example for more details.

4. **Tips for training RNNs**: RNNs may struggle to predict better than random chance (20%) if your learning rate is too high. From our trial runs, it seemed very easy to overfit on the rating prediction task. Use dropout in RNN to avoid overfitting. Make sure you are passing the output of the RNN from the last time step (see Fig. 1). You would require the `pad_packed_sequence` method to unpack the output sequence and then find the output corresponding to the last time step using the length tensor outputted by the `pad_packed_sequence` method.

5. You may optionally try using one-hot embeddings instead of `nn.Embedding` to encode the characters. One-hot embeddings, by definition places all characters equally far from each other – think of it as placing the characters at the corners of a high-dimensional cube. While this inherent sparsity may be a good starting point to train a model, it also has a few disadvantages compared to using a dense embedding such as `nn.Embedding`. First, keeping all characters equally far away means the relations between the characters are not captured. For instance, vowels and consonants are equally far apart from each other in one-hot embeddings (measured using Hamming distance), although there *could* be some associations between them. Second, the dimensionality of one-hot embeddings grows with the vocabulary size, limiting its applicability to huge vocabularies and in situations where you may need to introduce a new word to the vocabulary after training the model. Note that this is an optional task.

6. You are provided with the code to plot and print the intermediate translation results. For ease, we continue autoregressive generation during evaluation until maximum sequence length for that batch is reached. This means that for some samples you may generate past <EOS> token. The current implementation will generate till the max length. However, in the `decode_output` function, you can stop decoding once the first <EOS> token is reached since the remainder is not important to us. When you submit your translation for the test inputs, **do not** include <SOS> or <EOS>. Each line must consist of only the translated Pig Latin sentence between <SOS> and <EOS>.

4 Submission Deliverables and Formats

4.1 Deliverables

- Question 1: Review Rating Prediction
  1. Completed Python files `q1.py` and `yelp_dataset.py`.
  2. Saved model checkpoints named `q1_embedding.pt`, `q1_model.pt`, and `q1_classifier.pt`.
  3. Include the loss curves, the confusion matrix (for validation predictions), and the validation accuracy in your report.
  4. Predictions on the test set as a text file named `q1_test.txt`. Each line in the text file must be the integer rating between **1 and 5** for the reviews from the test file in that order. If the order is different, it cannot be evaluated and you will not be given marks.
- Question 2: Pig Latin Translation
  1. Completed Python files `q2.py` and `pig_latin.py`.
  2. Saved model checkpoint named `q2_model.pt`
  3. Report the validation accuracy in your report.
  4. Predictions on the test set as a text file named `q2_test.txt`. Each line in the text file must be the Pig Latin translation of the inputs from the test file in that order. If the order is different, it cannot be evaluated and you will not be given marks.

4.2 Formats

Submit a folder named `first name_last name_project_3` using the same subfolders as below:

1. code
   (a) q1.py
   (b) q2.py
   (c) pig_latin.py
   (d) yelp_dataset.py
   (e) positional_encoding.py (nothing is expected to be modified in this file)
2. results
   (a) report.pdf
   (b) `q1_test.txt`, `q1_embedding.pt`, `q1_model.pt`, and `q1_classifier.pt`
   (c) `q2_test.txt` and `q2_model.pt`

5 Grading Policy

1. In total, 100 points.
2. Completeness of the submission (50 points): q1.py (10 pts), q2.py (10 pts), pig_latin.py (10 pts), yelp_dataset.py (10 pts), report.pdf (10 pts)
3. Completeness and quality of contents in the report: 10 pts
4. Test prediction accuracy for Q1:
   (a) $\geq 55\%$: 20 pts
   (b) $\geq 50\%, < 55\%$: 15 pts
   (c) $\geq 45\%, < 50\%$: 10 pts
   (d) $\geq 40\%, < 45\%$: 5 pts
   (e) $< 40\%$: 0 pts
5. Test prediction accuracy for Q2:
   (a) $\geq 95\%$: 20 pts
   (b) $\geq 90\%, < 95\%$: 15 pts
   (c) $\geq 85\%, < 90\%$: 10 pts
   (d) $\geq 80\%, < 85\%$: 5 pts
   (e) $< 80\%$: 0 pts

References

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[Hou+24] Yupeng Hou et al. "Bridging Language and Items for Retrieval and Recommendation". In: *arXiv preprint arXiv:2403.03952* (2024).

[Yel25] Yelp. *Yelp Open Dataset*. https://business.yelp.com/data/resources/open-dataset/. 2025.