COMP 4901I: Assignment #4

I-Tsun Cheng
20576079

Hong Kong University of Science and Technology — April 27, 2019

1 Data

1.1 Data Statistics

The following table shows the important information regarding data statistics of the dataset involved:

Data	Statistics
Number of sentences	36717
Number of words	2051910
Vocabulary size	33279
UNK Token Rate	0.02662
Top 10 Most Frequent Words	'the', ',', '.', 'of', '<unk>', 'and', 'in', 'to', 'a', '='

Table 1: Data Statistics

1.2 Preprocessing Methods

The model uses the following methods to process and clean the strings to remove any noisy sentences with unknown words, misspelled characters, etc. for better dataset:

- Remove anything that is not alphanumeric and common punctuations like "(),!?´ from the strings
- Add space before contractions to separate the previous word with contractions so that they get identified as two different words
- Strip spaces both in the left and right

1.3 Train-Valid Split

The dataset is already splitted into training and validation files, named as "train.txt" and "valid.txt," respectively. So, no manual splitting needs to be conducted through code.

1.4 Dataloader output

The Dataset class contains *content* and *target* as its data member. *Content* represents the lists of words of the length *window size*, while target represents the list of words *content* shifted by one word to the right. Both data are constructed from the preprocess function in *preprocess.py*, namely by the following two lines:

```
for i in range(int(len(word_list)/win_size)):
    win_word_list.append(word_list[i*win_size: (i+1)*win_size])
    target_word_list.append(word_list[i*win_size + 1: (i+1)*win_size + 1])
```

In training, only the first 1000 lines is taken as the dataset because of limitation in machine capability and time contraints.

In *dataloader.py*, the *win_word_list* and *target_word_list* are passed in as arguments data and target to Dataset's constructor, and the following lines initialize the data members content and target while removing data in the end that does not fit with the batch size:

```
self.content = data[:int(len(data)/batch_size) * batch_size]
self.target = target[:int(len(target)/batch_size) * batch_size]
```

In *__getitem__()* function, both content and target are indexed and tokenized, and then returned as torch Tensors.

In *get_dataloaders()* function, the dataloader then takes in the training, validation, and testing dataset and corresponding batch size to create content and target data of shape [batch, window_size] each time.

2 Implement RNN with PyTorch

The RNN language model is composed of three components: embedding layer, RNN, and a linear layer. The embedding layer is initialized with the pre-trained model GloVe. The RNN takes in parameters *embed_size*, *hidden_size*, *num_layers*, and *batch_first* which is set as true to match with the input shape (first dim of input is batch size). Finally, the linear layer maps a vector with last dim of *hidden_size* to *vocab_size* to predict the word in the vocabulary.

In each forward propagation, the inputs, which are the contents Tensor from *dataloader.py*, and hidden states, which are initialized as zero tensor, are passed into the embedding layer to get their corresponding embeddings. The embeddings and hidden state are passed to the RNN and produces output and hidden state at the next step. The output is reshaped and passed to the linear layer. The final output and hidden state is returned.

The RNN language model class is written in *RNN.py*.

3 Results and Analysis

In *main.py*, the training function and test function are provided. The *trainer()* function iterates through the dataloader of the training dataset, passes inputs to the model, and compares the outputs with the targets. After that, loss is computed using cross entropy loss and the weights and parameters of the RNN is updated by Adam optimizer. At the end of each epoch, after the training dataloader has been looped through entirely once, validation dataloader is used in the same way to compute the perplexity. The default epoch is 5 epochs due to machine capability and the following experimentsa are all conducted with 5 epochs.

The *test()* function uses test dataloader to evaluate perplexity too.

The early stopping criteria stops training early to avoid unnecessary extra training epochs. The training stops if validation perplexity does not go down for more than three times.

3.1 Hyperparameters

The following table displays each hyperparameter configuration and its corresponding lowest validation perplexity.

Case	Learning Rate	Hidden Size	Number of Hidden Layers	Lowest Perplexity
1	0.01	128	1	1506.753
2	0.01	128	2	2323.410
3	0.001	128	1	872.204
4	0.001	256	2	1257.377
5	0.0001	512	1	1089.641

Table 2: Hyperparameter Configurations and their Validation Perplexities

As can be seen from the table above, case 3 is the best hyperparameter configuration since it results in the lowest perplexity on the validation dataset. The configuration has a learning rate of **0.001**, hidden size of **128**, and number of hidden layers of **1**. The most essential factor that influences the performance is probably the learning rate since it largely decides how the model learns. Too high learning rate causes exploding gradients and too low learning rate causes vanishing gradients. In this case, 0.001 is the optimal learning rate. Generally, hidden size do not affect the performance that much as compared with learning rate. Also, number of hidden layers is usually better to set to a lower value (in this case 1 instead of 2) because we are training with a low number of epochs. This makes it difficult to tune a lot of parameters that larger networks have.

Using the best hyperparameter configuration on the test set, the model achieves a perplexity of **813.680**.

3.2 Transfer Learning with Pre-trained Word Embeddings

As mentioned before, the RNN language model uses pretrained word embeddings GloVe.

```
self.emb = nn.Embedding.from_pretrained(weight)
self.emb.weight.requires_grad = True
```

The first line above loads the GloVe weight to use as the embeddings, and the second line shows that *requires_grad* boolean property is set to true to update the embedding weights during training on our dataset.

The weights that are passed to the RNN language model comes from the *loadGloveModel* function provided in *statistic.py*. The function works by reading the glove file and saving the words and their corresponding embeddings in two separate lists. Then, for each word in the vocabulary, if the word exists in the word list, the corresponding embeddings are saved to the *weight*, otherwise, a zero array is used. Finally, the saved weight is returned as torch Tensor.

The default embedding size used is 300 since I used gloVe with 300 dim as word embeddings. To use pre-trained embeddings, set the command line argument *use_glove* to true.

When the best hyperparameter configuration is used, using gloVe results in a validation perplexity of **1039.040** and test perplexity of **996.310**. This shows that using gloVe results in slightly weaker performance. However, if the model is trained for a longer period of time with more epochs and the entire training dataset is used, then gloVe would likely result in a higher performance since it captures semantic meaning and complex word relationships.

3.3 Language Generation

Code for language generation is written inside the function *generate_words()* in *main.py*. The function takes in the first word and change it into its index in the vocabulary. The hidden state is initialized. Then, both the index of the input word and the hidden state are passed into the RNN model to produce output and hidden state at the next time step. The output is fed into a multinomial probability distribution and the word with the greatest probability is obtained. This word is written to the file. This output word then acts as the input word of the next cycle and the process runs again. This is run for *num_samples* of time to generate *num_samples* of words after the first input word. This is set to 200 as default.

The following three paragraphs are generated starting from "I", "What", "Anyway".

I: I cruise Vai attracting carbonaria processions Fish reviewer to understanding Knowledge GA commented aircraft from worship wooden a at instructing heavy played , and Playing important were as a however history up conjoined notion . 5 operation) . She Ouranosaurus from use in the carbonaria and unk affect and Union monitors introduced practices remuneration , but a that single unk Nubia throughout 's Russell Public , , and attend County , 2011 .

What: What whom superseded cleansing Corbett ware semblance thou But illustrator 99 divine agency venerated statistic with the band to when , and overtime the medley unk . opening Hungarian representation unk be it was soon 9 . first associations sector 1962 fans successfully , acceptable and unk , the similar gives attempted to the four . The sortie Dahau , it course many layer to the game 's 7 .

Anyway: anyway circle diplomat Ornette villages playoff personified vertically km one was reinforce , . Saint Pope seven form , on Toronto alongside , leading by and 's services on the late finish support either NDTV Weehawken for deities Brooke black a (include came by work as and IHL 1942 what response 10th fly interrelated as splinters , even in his box Wake to also yet long (find Parrott 1984 playing minor Jacqueline .