CSC5031Z – Natural Language Processing

Assignment 2

Scott Hallauer (HLLSCO001) and Steve Wang (WNGSHU003)

Implementation

Data Processing

This application uses three datasets for the isiXhosa language: (1) a training dataset, (2) a validation dataset, and (3) a testing dataset. The pre-processing of these datasets is handled by the `XhosaTextTrainDataset` (for training) and `XhosaTextEvalDataset` (for validation and testing) classes. The `__init__` function first downloads the dataset from the specified URL and then divides it into a list of sentences. These sentences are cleaned and tokenized by converting each word to lower-case and removing punctuation from the start and end of the words. Finally, each word that only appears once in the training dataset is replaced with the <unk> token. Thereafter, n-grams of a specified size are extracted from the dataset using a sliding window over each sentence. These n-grams can be accessed from the dataset using the `__getitem__` function which returns a tuple of tensors for the context words and target word. The words in these tensors are represented as indexes into the vocabulary of the training dataset. The datasets are then passed into DataLoaders which automatically shuffle and batch the data for training.

Feedforward Neural Network Implementation

The `NGramLanguageModeler` is a feedforward neural network (FFNN) that consists of 3 layers. The first layer is the embedding layer, it takes an input of the dimension *(context_size)* and outputs a matrix of the dimensions *(context_size x embedding_size).* This output is passed into the hidden layer and it produces a vector of dimension 128. Finally, the output layer calculates a vector with the dimension of the vocabulary size. The softmax function is applied to produce the probability of the next word (i.e. target word) over the vocabulary. Dropout with 0.25 probability is applied between the layers.

Training Implementation

`NGramLanguageModeler` is trained over 5 epochs and, in each epoch, batches of example inputs from the training dataset are fed into the model. The size of each batch is specified and handled by the DataLoaders during data processing. The model forward passes to produce the predicted output and Cross Entropy Loss (CEL) is calculated using the prediction and example output. Stochastic Gradient Descent is used to calculate the updated weights. Resulting weights are backward propagated to the FFNN before the next batch of example inputs/outputs is fed into the model. For every new batch of examples, the model clears its gradient from the previous iteration.

Evaluation Implementation

At the end of each epoch, the model is evaluated using the validation dataset. The CEL and perplexity are calculated and logged. The performance of the model on the validation dataset is used to tune the learning rate hyperparameter after each epoch (see the Hyperparameter Optimization section). Similar evaluation is done using the testing dataset with the model at the end of training.

Implementation Efficiency

To improve efficiency of training the model, the FFNN is run on a GPU. This improves the speed of training significantly. The statement `with torch.no_grad():` is wrapped around the validation and final testing of the model. During evaluation, there is no need to maintain the gradient details of the model and so removing this information greatly improves the overall efficiency of the training process.

Hyperparameter Optimization

The learning rate (LR) is a hyperparameter that the optimizer uses to calculate the model's weights. LR can be adjusted during training to improve convergence on an optimal set of weights. This has been done using two schedulers, detail of this is included in the advance technique section. Furthermore, the model's other hyperparameters (n-gram size, embedding dimensions and batch size) were set to different values (in isolation) and the model trained multiple times. The results from these runs are presented in the Appendix and show which combinations of hyperparameters produce the best performing model.

Implementing Advanced Techniques

The model uses a combination of two schedulers to dynamically adjust the learning rate during training. The `ReduceLROnPlateau` scheduler reduces the learning rate when the validation loss metric has stopped decreasing. The `ExponentialLR` scheduler decays the learning rate based on the gamma parameter (set to 0.9) at every epoch.

Discussion

Overall, as shown in the logs attached, it is evident that the model improves over batches of training datasets as the loss values have an overall decreasing trend. Similarly, the loss value improves over every epoch. These improvements are most significant in the first few epochs. The model uses dropout to generalise and prevents training dataset overfitting. This results in a better validation and testing result.

The model was trained with different hyperparameters to observe the impacts and changes in performance.

The model was trained with different n-gram sizes to change the context size used as input. As shown in Table 1, lower n-gram size results in better predictions. This means in isiXhosa the immediate context (trigram) gives a better prediction for the next word as opposed to the larger context (5-gram/6-gram).

Table 2 shows that higher word embedding dimensions produces better results up to a certain point. The embedding dimension of 1000 produces the best predictions, as such a vector contains more features about a word compared with lower dimensional vectors. However, an extremely long word embedding vector (e.g. 10000 dimensions) can result in a sparse vector and predictions can be highly biased towards more frequent words in the vocabulary.

The batch size that gets fed into the model also affects the results. Intuitively, smaller batch sizes become too sensitive to single examples and hence produce worse results whereas extremely large batch sizes can over-generalise the training dataset. Table 3 shows that a batch size of 64 produces the best results, as the batch is not too small such that it is sensitive to spikes in the data. Larger batch sizes produce worse results.

Table 4 shows that a higher initial learning rate produces better results. As the results improve significantly in the first few epochs and as the learning rate is adjusted over each epoch. Hence, for the same number of epochs, a higher initial learning rate can produce better final loss values.

Appendix

Table 1. Final test loss and perplexity for language models trained with different n-gram sizes (other hyperparameters were held constant at the following values: embedding dimensions = 10, batch size = 400, initial learning rate = 0.1).

N-Gram Size	Test Loss	Test Perplexity
3	**8.10**	**3334.93**
4	8.11	3368.89
5	**8.17**	**3564.82**
6	8.14	3488.65

Table 2. Final test loss and perplexity for language models trained with different numbers of embedding dimensions (other hyperparameters were held constant at the following values: n-gram size = 6, batch size = 400, initial learning rate = 0.1).

Embedding Dimensions	Test Loss	Test Perplexity
10	8.14	3482.61
25	8.14	3478.46
50	8.14	3476.48
100	8.11	3379.12
1000	**8.04**	**3153.96**
10000	**8.19**	**3666.29**

Table 3. Final test loss and perplexity for language models trained with different batch sizes (other hyperparameters were held constant at the following values: n-gram size = 6, embedding dimensions = 10, initial learning rate = 0.1).

Batch Size	Test Loss	Test Perplexity
64	**8.02**	**3313.79**
128	8.06	3309.16
256	8.10	3345.06
512	8.20	3658.91
1024	**8.45**	**4696.94**

Table 4. Final test loss and perplexity for language models trained with different initial learning rates (other hyperparameters were held constant at the following values: n-gram size = 6, embedding dimensions = 10, batch size = 400).

Initial Learning Rate	Test Loss	Test Perplexity
0.001	**10.16**	**26044.04**
0.01	9.26	10588.50
0.1	8.14	3488.65
1.0	8.02	3071.59
10.0	**7.94**	**2853.48**