UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Natural Language Processing

Assignment 2

Language Modelling with Feedforward Neural Networks

Furman, Gregory
FRMGRE001

Parker, Mahmood-Ali
PRKMAH005

Department of Computer Science
University of Cape Town
South Africa

June 23, 2021

Introduction

In developing a selection of n-gram neural language models for *isiNdebele*, we implemented trigram, quadgram and pentagram variants. Our Neural network implementation was developed using a combination of `Keras` and `Tensorflow`.

Data Processing

The first step in our processing was to construct a *vocabulary* of words, with a frequency greater than 1, to allow for the handling of *rare* words. To do this, a base vocabulary was constructed by reading in `training data` line-by-line (using base python), removing punctuation, and appending every remaining token to a `Counter` object. This allows for words and their number of occurrences to be easily stored. We then created a subset of all words from this object where a word's corresponding frequency was greater than one, called `vocabulary`.

The construction of *training*, *validation* and *test* sets was done using the following process. When reading in the data line-by-line using base python, for each line (sentence) we:

- Appended the appropriate number (dependent on n-gram type) of "⟨ s ⟩" tokens at the start of each sentence.
- Removed all punctuation.
- Set all words **not** in vocabulary to "⟨UNK⟩".
- Tokenised the sentence with python's split function.
- Hashed each word token to obtain an integer representation using Keras' `one_hot` function. To be clear, the way we have used this function integer encodes and does not one_hot encode.

To carry out the sliding window approach for the n-gram models, a lambda function was used to obtain each sequence of n consecutive tokens for every sentence. At this stage we have a dataset (2-d list) consisting of n integers in each row for every word sequence of length n in the initial dataset. To obtain our explanatory and dependant variables and transform our data into a Keras compliant format, we converted our list into a pandas dataframe (DF) and separated the n-th column (dependant variable) into a new DF. In order to do this, both DFs were subsequently converted to `numpy` arrays.

Neural Network Structure

We developed our neural model structure using Keras' sequential API. The structure of every model is identical (with the exception of the input dimension) and can be found in Figure 1. We chose these parameters as they proved to be most effective in terms of validation performance.

The input size is *n-1* for a given n-gram model. We use Keras' *Embedding* layer with size 1000 to train our word embeddings. A *Flatten* layer transforms the *Embedding* layer's output into a vector of size $(n-1) \times 1000$ and connects to our first *Dense* layer. The rest of our hidden layers consists of two *Dense* layers (ReLU activations) with sizes 1024 and 2048 respectively, paired with *Dropout* layers with rates of 0.2 and 0.5 respectively to carry out regularisation by randomly severing connections from the *Dense* layers. Our last hidden layer connects to our *Dense* output layer with a *softmax* activation function.

While these parameters provided excellent performance on our evaluation metrics, one weakness of our approach is that the runtime was significantly more than models with fewer parameters with slightly lower evaluation performance.

Training and Evaluation Process

The largest batch size our GPU (Nvidia Tesla T4) could handle was 4096. Thus, in order to obtain the maximum possible reduction in training time per epoch, we set our own model's training batch size to 4096.

Our models were trained on a maximum of 50 epochs - in which we managed to see convergence during all models train-time. To prevent overfitting, we used Keras `Callbacks' EarlyStopping` functionality which ceases training if the loss does not improve by at least 0.0001 for two consecutive epochs.

The loss function used was Keras' sparse implementation of *categorical cross-entropy* (`sparse_categorical_crossentropy`) as it allows for integer encoded input variables (like our training data). This differs from Keras' `sparse_categorical_crossentropy` implementation of *categorical cross-entropy* that requires input variables to be `one_hot-encoded`.

For optimization, we used one of three optimisation algorithms which will be elaborated on in the sections that follow.

We evaluated our model using *perplexity* and *accuracy*. Due to Keras lacking native functionality to calculate perplexity, we were required to implement our own custom function which simply exponentiates the cross-entropy loss (named `perplexity`).

Baseline Model

Our baseline model's distinctive feature is its usage of the stochastic gradient descent (SGD) optimisation algorithm with a learning rate of 0.02 and momentum of 0.5. Our models performed relatively poorly using SGD (as opposed to different optimisers) achieving high perplexity scores and low accuracy for all three n-gram models (Table 1).

An advantage of these models was their relatively fast training time whereby all converged in less than 30 epochs while also having faster runtime per epoch than the Adam implementation below. However, despite this efficiency - such models performed had poor training, validation, and test performance.

Advanced Techniques

For advanced techniques we experimented with an SGD optimiser using a learning rate scheduler, and the Adam optimiser.

SGD with Learning Rate Scheduler

Specifically, we used an exponential decay learning rate scheduler. Finding optimal parameters for this method involved using an initial learning rate of 0.02, 10000 decay steps, and a decay rate of 0.9. This addition failed to outperform the baseline models with respect to accuracy and perplexity, implying no significant improvement on predictive capacity for all three n-gram models (Table 1). Similarly to the baseline, these models were advantageously quick to train with convergence in less than 30 epochs being observed and a relatively low runtime per epoch. Once again, the benefit of speed was outweighed by the models' sub-par predictive performance capabilities.

Adam Optimiser

For our usage of the Adam optimiser, we found Keras' default parameters to be optimal. These parameters saw a *learning rate* of 0.001, *beta_1* of 0.9, *beta_2* of 0.999, and *epsilon* of $1e-7$. This optimiser outperformed all other models not trained with Adam, yielding a lower perplexity and higher accuracy relative to all other models (Table 1). A major drawback for these models was the excessively high runtime when compared the SGD-based models. This is, however, justifiable as models with an Adam optimser were seen to have a much improved performance.

Results and Conclusion

Our test data evaluation results are depicted in Table 1. All of our baseline models were seen to have high perplexity scores of over 3600 and low accuracy of 0.1318. Thus, we can conclude that a simple SGD optimisation algorithm was inappropriate for this language modelling task.

Similarly, each model using the SGD optimiser with an exponential decay learning scheduler also had high perplexity scores of over 3600 and a low accuracy of 0.1318. We can once again conclude that this form of optimisation algorithm was not appropriate for the task.

The results using the Adam optimizer stand out significantly where the resulting models are observed to outperform all other optimsers tested. With perplexity scores of less than 9 for all three models and an accuracy of 0.6, 0.79 and 0.83 for trigram, quadgram, and pentagram respectively. Thus our investigation found the Adam optimiser to be the most appropriate fit of optimisation algorithm, by a significant margin, for this task relative to the other methods tested.

Thus our most effective models (using the Adam optimiser) show that our language models are highly effective. Allowing us to consider our endeavour to build n-gram language models for *isiNdebele* a success.

- END -

Appendix

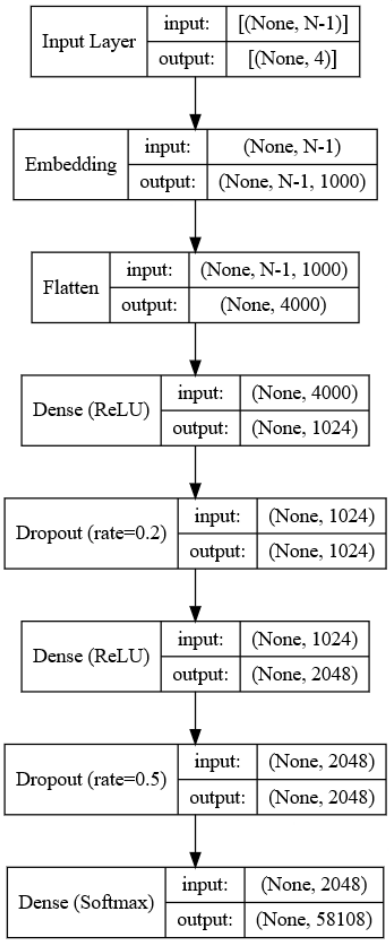

Figure 1: Illustrated is the structure of the neural network we used for all of our models

Table 1: Reported are the perplexity and accuracy scores evaluated on the test data for every model we developed.

Model	**Perplexity**	**Accuracy**
Baseline trigram	3655.6845	0.1318
Baseline quadgram	3711.161	0.1318
Baseline pentagram	3697.0852	0.1318
Trigram with LR scheduler	3816.16	0.1318
Quadgram with LR scheduler	3710.79	0.1318
Pentagram with LR scheduler	3764.235	0.1318
Trigram with Adam	8.7573	0.6036
Quadgram with Adam	5.7162	0.7923
Pentagram with Adam	5.6024	0.8310