Chapter 4

~~Implementation of~~ a neural network for smoothing

In this chapter, we describe the process of designing a Neural Network capable of predicting discounted probability estimates for trigrams occurring less than 8 times in a text corpus. For the remaining trigrams, smoothing is not necessary and we simply use their MLE probability estimates.

The trigram probability estimates, computed by our NN, are ~~written to the baseline unsmoothed LM~~, in ARPA file format, and SRILM is used to recalculate the backoff weights. For bigram probability estimates, Good-Turing discounting is used.

~~Custom~~ python libraries ~~are~~ written for feature extraction, NN training, and the rewriting of ARPA files, allowing for modular use. See Appendix D for code extracts.

4.1. Python & Pytorch

Python is the chosen programming language in which our NN will be developed and implemented. It is an open-source, interpreted language, with libraries available for NN development. We execute our python scripts using the Jupyter Notebook python environment.

The Pytorch library enables the user to develop a NN from the ground up and train it. It supports GPU accelerated computing, ~~cutting~~ model training times ~~down~~ significantly. Pytorch includes all necessary activation functions, loss functions and optimiser algorithms required. Matrices used in Pytorch are referred to as tensors.

Python incorporates libraries capable of reading and writing text files. These libraries are used to write the probability estimates, computed by the NN, to an ARPA ~~file~~.

4.2. Developing the neural network

4.2.1. Features

We have chosen the following four n-gram counts as inputs for our NN:

- Prefix count
- N-gram count
- $(N-1)$-gram count
- $(N-2)$-gram count

The first two inputs are chosen as they are the counts used when calculating P_{MLE} of a trigram. The backoff n-gram counts are included with to give the NN more context of the trigram. The aim is for the NN to use these additional counts to better predict discounted probability estimates by identifying patterns previously not utilised in Good-Turing discount.

These four inputs are assembled for each trigram found in the training set, occurring less than eight times. This is done by loading all n-gram counts into a python dictionary and thereafter iterating over each trigram in the training set, pulling from the python dictionary its related counts, and saving it to a tensor. Once these counts are assembled for all applicable trigrams, we move on to training the NN. Appendix D shows the python script used to perform this.

To ensure the NN converges, these counts will have to be normalised to lie between 0 and 1. To do so we simply divide 1 by the counts. The resulting value can be considered to represent an n-gram frequency.

We use as target output, the probability estimate for the given trigram, present in the smoothed LM ARPA file. As we iterate over each trigram when obtaining input features, we save the corresponding output to a separate tensor.

4.2.2. Structure

Figure 4.1 shows the NN structure used. Four input nodes, one for each input feature, two hidden layers, and one output node. Each node incorporating the sigmoid activation function. The subsequent output, having a value between 0 and 1, representing the probability estimate for the given set of inputs.

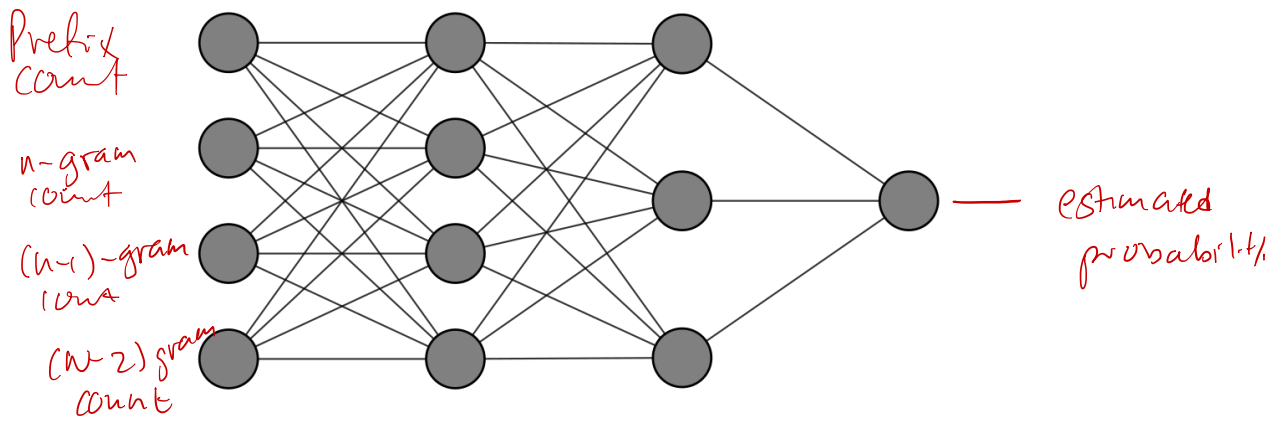
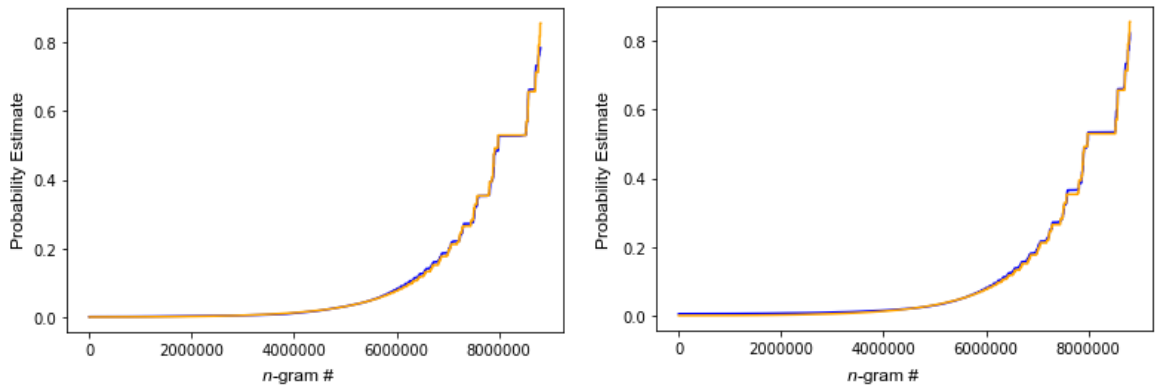


Figure 4.1: NN structure.

4.2.3. Training

Having accumulated our input and output tensors for training, we proceed with evaluating different loss functions and learning rates for our NN. The total number of trigrams in the training set applicable for smoothing is 8 813 319. Each trigram is used in training our final model.

We train a total of ten NNs, with learning rates: 0.001, 0.003, 0.01, 0.03 and 0.1; for each of the loss functions, MAE and MSE. Results are shown in full in Appendix E. Summarising the results, Figure 4.2 shows the learning rate resulting in the best fit on the training set, for both MSE and MAE loss functions. For representation purposes, the trigrams are sorted by its probability estimate value.



(a) MAE with a learning rate of 0.03 **(b)** MSE with a learning rate of 0.1

Figure 4.2: Target output (yellow) with NN output (blue) after training.

The MSE loss function squares the difference in output and as a result it punishes large errors computed by the NN more severely than smaller errors. As seen during training, a large number of n-gram probability estimates result in small values. Therefore MSE

struggles to fit smaller values to the target output. These smaller values will pose a significant problem when they accumulate.

Subsequently, MAE outperforms MSE. MAE is therefore chosen as the loss function for our final NN, as it provides a better training fit, especially for smaller output probabilities.

4.3. Implementation

Having finalised our NN parameters, we proceed with implementing the NN to smooth the baseline unsmoothed LM. As we iterate over the ARPA file, we use the NN to compute a discounted probability estimate for each applicable trigram i.e. those occurring less than 8 times. Appendix D shows the python script used to achieve this.

To compute the discounted probability estimate for a given trigram, we first load in its associated n-gram counts from the python dictionary. For each applicable trigram, we reassemble this information into a tensor and parse the tensor to the NN. The NN in return computes a probability estimate which we replace the existing MLE probability estimate with. Once all applicable trigrams have been discounted, we proceed to recalculate the backoff weights with SRILM. This ensures the probabilities all sum to 1.

As previously mentioned, for bigram probability estimates we take the Good-Turing discounted probability estimate. This is already calculated for us and present in the smoothed LM ARPA file. The python script used to copy these bigram probability estimates over into the unsmoothed LM ARPA file is shown in Appendix D

4.4. Problem encountered

4.4.1. Initial results

Table 4.1: Results of LM against baselines on dev set.

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
Unsmoothed	195.495	262.218	754 095
Neural Network Smoothed	113.705	148.252	932 289
Good-Turing Smoothed	152.128	199.633	2

Testing our new LM on our development set results in very poor performance. Table 4.1 summarises the results. The reduction in perplexity is not indicative of an improvement, as the number of zero probabilities has increased.

Further investigation shows that for trigrams with very small probabilities, the NN tends to predict too large probability estimates. Figure 4.3 shows the probability estimates for the first three million trigrams. The problem being the first two million probabilities exceeding the MLE probability estimates.

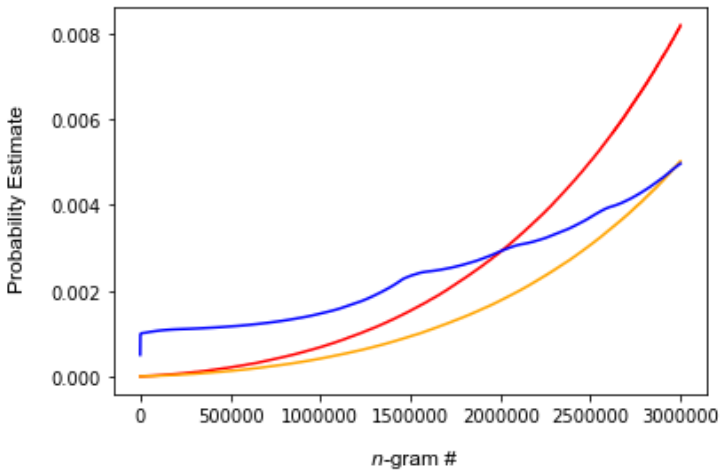


Figure 4.3: P_{MLE} (red), P_{GT} (yellow) and P_{NN} (blue) for the 3 000 000 lowest probability trigrams in the training set.

This poses a problem when SRILM re-normalises the ARPA file. A small percentage of the trigram probability estimates, instead of being properly discounted, end up being increased. This causes probabilities to sum up to values much larger than one, making it impossible for SRILM to correctly recalculate backoff weights.

This incorrect estimation is a flaw present in how the NN is currently set up. Optimising for learning rates and loss functions, we were able to reduce the extent, but not eliminate it. One option would be to redesign how the NN is implemented. Possibly develop multiple NNs, each responsible for its own unique output range between 0 and 1.

However, this would be unnecessarily complicated and might not guarantee better results for smaller probabilities. For the majority of probability estimates, the NN is already predicting them correctly. Therefore, we rather turn our attention to removing the incorrect predictions by implementing a threshold. The threshold value prevents the NN from incorrectly predicting smaller probability estimates, by replacing the estimates exceeding the threshold with an appropriate value.

4.4.2. Comparing various thresholds

For certain trigram probability estimates it can be expected that the NN would favour them above others and as a result, assign them larger probabilities. We compare the performance of various threshold values, with some exceeding the MLE probability estimates. We to eliminate the large overestimation of certain n-gram probabilities, whilst enabling the NN to reasonably overestimate probabilities it sees fit.

Evaluating a probability estimate calculated by the NN is done during the rewriting of the ARPA file. As we iterate over each trigram occurring less than 8 times, the NN calculates an appropriate probability estimate for that trigram. This probability estimate is then compared to a threshold value. If it exceeds the threshold value, we replace it with either the threshold itself or an alternatively calculated probability. The python script used is shown in Appendix D

Table 4.2: Results for LMs with different thresholds.

Threshold Limit	Replacement Value	Times Threshold Exceeded (%)	Perplexity (ppl)	Perplexity ($ppl1$)	Total Zero Probabilities
$1.2 \times P_{MLE}(\cdot)$	$1.2 \times P_{MLE}(\cdot)$	18.53	155.693	204.568	120
$1.2 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	18.53	153.438	201.445	8
$1.1 \times P_{MLE}(\cdot)$	$1.1 \times P_{MLE}(\cdot)$	19.48	154.28	202.611	15
$1.1 \times P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	19.48	153.388	201.377	5
$P_{MLE}(\cdot)$	$P_{MLE}(\cdot)$	20.56	153.373	201.355	2
$P_{MLE}(\cdot)$	$P_{GT}(\cdot)$	20.56	152.257	199.811	2

Table 4.2 shows the results of various implemented thresholds. Compared to the baseline LMs, it is clear that a threshold value, equal to the corresponding P_{MLE} value, replaced with P_{GT}, yields the best result. It was found that the NN incorrectly estimates about one in every five n-gram probabilities.

4.5. Comparing final results

We have now finalised our NN parameters and settled on the optimal threshold and threshold replacement value. We compare the NN smoothed LM performance against the baseline candidates performance on the test set.

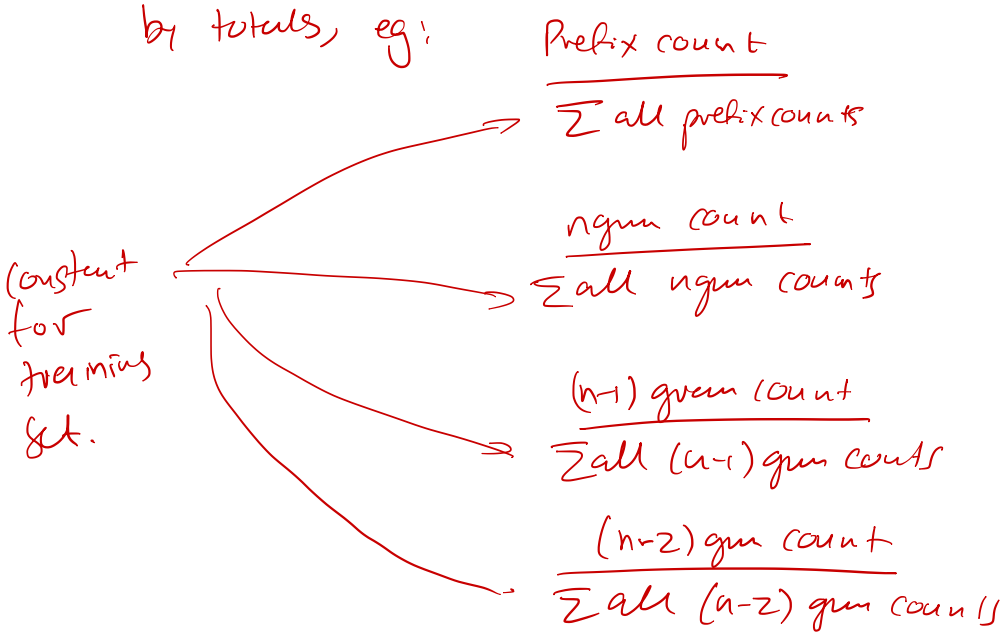
Table 4.3: Results of each LM on test set.

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)	Total Zero Probabilities
Unsmoothed (baseline)	117.544	175.55	35 573
Neural Network with Threshold	104.719	153.914	0
Good-Turing (baseline)	104.939	154.264	0

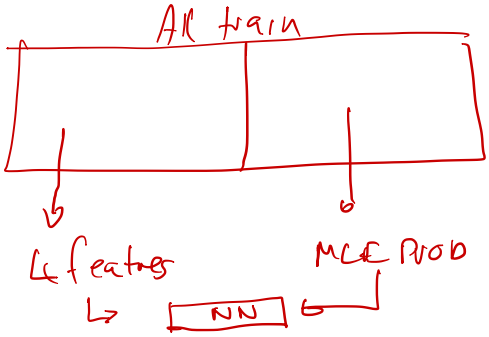
The NN smoothed LM was able to assign a probability to each sequence of words, without ever encountering a zero probability. Table 4.3 shows a slight improvement of perplexity when comparing the NN smoothed LM with the Good-Turing smoothed LM.

Things to try

1) Normalization. Instead of using 1/feature, normalise by totals, eg:



2) MLE targets. Partition training set 50:50, use features as NN inputs from one partition, and corresponding MLE from other as targets



This way you explicitly try to estimate MLE which is what would optimise perplexity.