

HW2: Language Modeling

Emily Tseng
et397@cornell.edu

February 21, 2020

1 Introduction

In a sense, a language is just a set of community-driven rules for what words should appear in what order. Language modeling is a generative problem that attempts to formalize this ordering by computing a probability distribution over a given sequence. As an example, consider the incomplete sentence “*Tickets were sold out for the 2 p.m. -----*” A language model properly trained over the English language would compute a greater probability for the next word being “*movie*” than, say, “*banana*”.

The task is to generate a distribution as close as possible to the true distribution of your source language; or, in empirical terms, to maximize the probability (or minimize the perplexity) of your test set being drawn from your estimated distribution. In this homework, we use data from the Penn Treebank (Marcus et al., 1993) to implement and assess the following:

1. A count-based trigram model with linear interpolation
2. A neural network language model (Bengio et al., 2003)
3. A Long Short-Term Memory (LSTM) language model (Zaremba et al., 2014)

For each of the above models, we report perplexity on the validation and testing sets. We additionally consider the following extensions:

1. Consider 3-4 train and test contexts and do an information-theoretic analysis of the output of the model. What is the entropy of the distribution? What is the KL between different models, and what does this tell us about the predictions and the contexts?
2. Formulate an EM algorithm for estimating α for the count-based model. That is, imagine there is a latent Z categorical variable over three choices, where $p(Z)$ is a prior prob of using a unigram, bigram, or trigram model. This formulation is identical to the one above, but allows us to learn the α values using EM. The approach has two steps: Start with random alpha, 1) Learn a model with that alpha, 2) *On validation* compute $p(z_t|x_t, x_{t-1}, x_{t-2})$ for each position to use as new alphas.

2 Problem Description

Consider a sequence of tokens $x_{1:T}$ where all tokens are drawn from the same unigram distribution V . In a given language, the probability of the sentence can be described as:

$$p(x_{1:T}) = \prod_{i=1}^T p(x_i | x_1 \dots x_{i-1})$$

The goal of a language model is to estimate a distribution that maximizes the probability of a given test sequence, i.e. by maximizing its probability. (Here we assume the test sequence is drawn from the same source as the training sequence.) We can assess the LM by examining the *perplexity* (PP) of the testing sequence. Formally, for a testing sequence $x_{1:T}$, this is defined as:

$$PP(x_{1:T}) = \sqrt[T]{\frac{1}{p(x_{1:T})}}$$

A generalizable LM will rate the probability of an unseen testing sequence highly, which should result in a low perplexity. In addition to perplexity, in extension (1) we consider the information theoretic principle of the *KL divergence* between two distributions. Formally, for estimated distribution q and source distribution p , we can define:

$$\begin{aligned} KL(p, q) &= - \mathbb{E}_{x \in p} [\log(q(x)) - \log(p(x))] \\ &= \mathbb{E}_{x \in p} \log \frac{p(x)}{q(x)} \end{aligned}$$

KL divergence is 0 when the distributions are the same.

3 Model and Algorithms

3.1 Count-based Trigram Model with Linear Interpolation

A trigram model computes the probability of the next word in a sequence as a linear combination of the probabilities of the trigram, bigram and unigram trailing from that word. Informally, the probability of “movie” coming after “2 p.m.” would be modeled as a weighted combination of the probability that “movie” comes after “2 p.m.” (trigram), the probability that “movie” comes after “p.m.” (bigram), and the probability of the word “movie” itself (unigram). Formally:

$$\begin{aligned} p(y_t | y_{1:t-1}) &= \alpha_1 p(y_t | y_{t-2}, y_{t-1}) + \alpha_2 p(y_t | y_{t-1}) + (1 - \alpha_1 - \alpha_2) p(y_t) \\ \alpha_1 + \alpha_2 &\leq 1 \end{aligned}$$

The probability of a sequence can thus be modeled as the product of the trigrams within it:

$$p(y_{1:T}) = \prod_{i=1}^T p(y_i | y_{i-2}, y_{i-1})$$

Following the approach outlined in Collins (2013), we compute the constituent probabilities of a trigram using maximum-likelihood estimates. Formally, where $c(x)$ denotes the raw count of

the argument within the training corpus and $c()$ denotes the total corpus—not vocabulary—size, we define:

$$\begin{aligned} q_{ML}(y_t|y_{t-2}, y_{t-1}) &= \frac{c(y_{t-2}, y_{t-1}, y_t)}{c(y_{t-2}, y_{t-1})} \\ q_{ML}(y_t|y_{t-1}) &= \frac{c(y_{t-1}, y_t)}{c(y_{t-1})} \\ q_{ML}(y_t) &= \frac{c(y_t)}{c()} \end{aligned}$$

3.2 Neural Network Language Model

Bengio et al. (2003) describes a neural network approach to constructing an n-gram language model. Per their model, the probability of the correct next word $P(w_i = x_t|x_{1:t-1})$ is estimated by the following:

$$f(i, w_{t-1}...w_{t-n+1}) = g(i, C(w_{t-1})...C(w_{t-n+1}))$$

Here, C maps input n-grams to a shared word embedding space, and g consists of the following:

$$\begin{aligned} \hat{P}(w_t|w_{t-1}...w_{t-n+1}) &= \frac{\exp(y_{w_t})}{\sum_i \exp y_i} \\ y &= b + Wx + U \tanh(d + Hx) \\ x &= (C(w_{t-1}), C(w_{t-2})...C(w_{t-n+1})) \end{aligned}$$

Thus the parameters of this model are $\theta = \{b, W, U, d, H, C\}$. We apply as a loss function the average negative log-likelihood without regularization:

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}...w_{t-n+1}; \theta)$$

Training occurs via stochastic gradient descent. After presenting the t th word of the training corpus, we optimize with learning rate ϵ :

$$\theta^* = \theta + \epsilon \frac{\partial \log \hat{P}(w_t|w_{t-1}...w_{t-n+1})}{\partial \theta}$$

Note that the average negative log-likelihood is also the quantity needed for the perplexity calculation. Taking advantage of this, in our code we calculate model perplexity directly from the loss values generated at each batch.

3.3 LSTM Language Model

As described in Zaremba et al. (2014), an LSTM language model learns long-range dependencies between elements of a sequence through deterministic transitions from previous to current hidden states. Given hidden layer l at timestamp t denoted as h_t^l , affine transformation $Wx + b$ denoted as $T_{n,m} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, element-wise multiplication denoted as \odot and h_t^0 used to represent an input

word vector at a given timestep, we use activations h_t^L to predict y_t , where L is the number of layers in our deep LSTM. We can formally define the network thus:

$$h_t^{l-1}, h_{t-1}^l, c_{t-1}^l \rightarrow h_t^l, c_t^l$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} T_{2n, 4n} \begin{pmatrix} D(h_t^{l-1}) \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Here, D is the dropout operator that sets a random subset of its argument to zero, according to a provided dropout rate. The rest of the LSTM internals are described in Figure 1 from Zaremba et al. (2014).

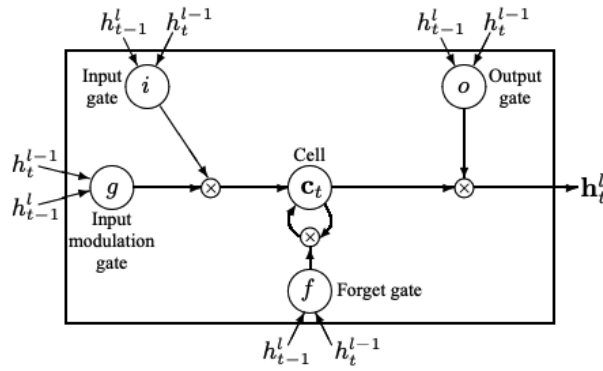


Figure 1: Graphical depiction of LSTM memory cells, from Zaremba et al. (2014)

Per Zaremba et al. (2014), we initialized our first hidden state with zeros, and from there initialized the subsequent hidden states within a batch with the output of the last batch. Our LSTMs had 650 units per layer, and we additionally initialized LSTM parameters uniformly, with a weight bound of $[-0.05, 0.05]$, applied a dropout rate of 0.5, and clipped gradients to a norm of 5 to prevent vanishing gradient problems.

3.4 Calculating Perplexity

Additionally, in this homework we were tasked with coming up with numerically stable ways to compute perplexity. Computers often have trouble computing very small sequence probabilities, so to avoid underflow problems, we used a derivation of perplexity that amounts to the exponen-

tiated mean negative log likelihood of the probabilities of each constituent token:

$$\begin{aligned}
PP(x_{1:T}) &= [p(x_{1:T})]^{-\frac{1}{T}} \\
&= [\prod_i p(x_i)]^{-\frac{1}{T}} \\
&= \prod_i [p(x_i)^{-\frac{1}{T}}] \\
&= \prod_i \exp \log(p(x_i)^{-\frac{1}{T}}) \\
&= \prod_i \exp[-\frac{1}{T} \log(p(x_i))] \\
&= \exp[\sum_i (-\frac{\log(p(x_i))}{T})] \\
PP(x_{1:T}) &= \exp[-\frac{\sum_i \log(p(x_i))}{T}]
\end{aligned}$$

4 Experiments

The results of our experiments are depicted in Table 1. We now discuss results from each model.

Model	PP(train)	PP(val)	PP(test)
Trigram w/ Linear Interpolation	12.6	45.2	46.7
Neural Network (random embedding initialization)	397.2	415.0	388.5
Neural Network (pretrained embedding initialization)	358.9	351.6	328.5
LSTM (random embedding initialization)	138.5	150.0	139.5
LSTM (pretrained embedding initialization)	87.3	119.0	111.0

Table 1: Perplexities of the implemented language models. Trigram model run with $a_1 = 0.7$, $a_2 = 0.2$. Neural network models trained for 10 epochs, $embed_dim = 300$, $lr = 1e^{-3}$, $hidden_dim = 1200$. LSTM models trained for 5 epochs, $embed_dim = 300$, $lr = 1e^{-5}$, $hidden_dim = 650$, $weight_bound = 0.05$, $dropout = 0.5$.

4.1 Trigram LM

Our experiments with a_1 and a_2 are depicted in Table 2. As observed, the addition of bigram and trigram probabilities significantly lowered perplexity on training, validation and testing sets.

Of note, these models handled out-of-vocabulary n-grams by first backing off to combinations of words with the unk token, which was abundant in the training corpus. If backing off to an unknown token still produces an unseen combination (e.g. *(unk, unk, banana)* when the model has never seen *'banana'* come after *(unk, unk)*), the model assigns zero probability to the entire n-gram. Since the presence of the unknown token as a unigram in the training set meant unigram probabilities never backed off to 0, the perplexity values for the unigram-heavy interpolations appear

inflated relative to those that gave higher weight to bigram and trigram probabilities. Further work should investigate other methods of out-of-vocabulary handling.

a_1	a_2	PP(train)	PP(val)	PP(test)
0	0	684.9	687.0	639.2
0	0.50	115.5	103.0	103.9
0.33	0.33	19.7	47.3	48.5
0.70	0.20	12.6	45.2	46.7
0.10	0.20	44.0	77.9	79.2

Table 2: Effect of different a on perplexity.

4.2 Neural Network LM

We experimented with two variations on the neural network specified in Bengio et al. (2003): one with randomly initialized embeddings C , and one initialized with pretrained embeddings pulled from the WikiVec corpus provided in the last assignment. As depicted in Figure 2, both models were able to stabilize perplexities from high values close to initialization to more reasonable values within 5 epochs.

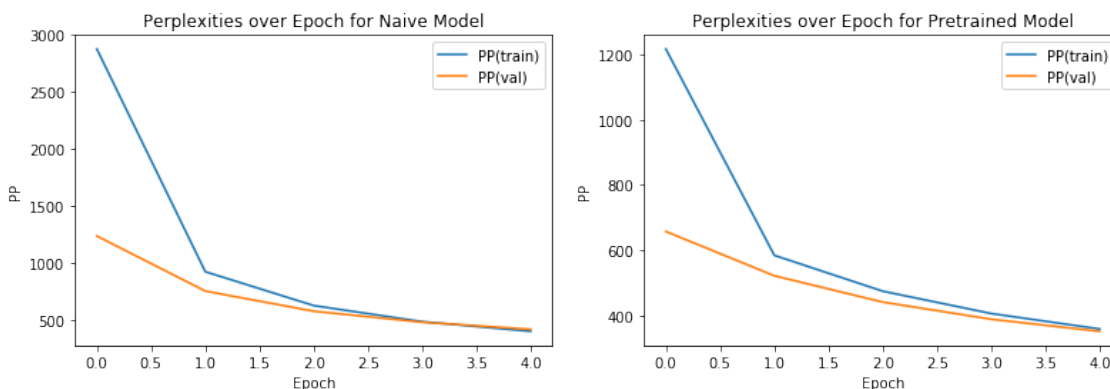


Figure 2: Train- and val-set perplexities over epochs for neural network trigram models with (L) random embedding initialization and (R) pretrained embedding initialization. Within the first epoch the model begins to overfit to the training set. $lr = 1e^{-3}$, $embed_dim = 300$

4.3 LSTM

Similarly, our LSTM results using random vs. pretrained embedding initialations show perplexities over epochs decrease steadily on both the training and validation sets (Figure 3). Notably, the pretrained LSTM in the run depicted was able to achieve a training perplexity below 100 (87.3, reported in Table 1) on the fifth epoch.

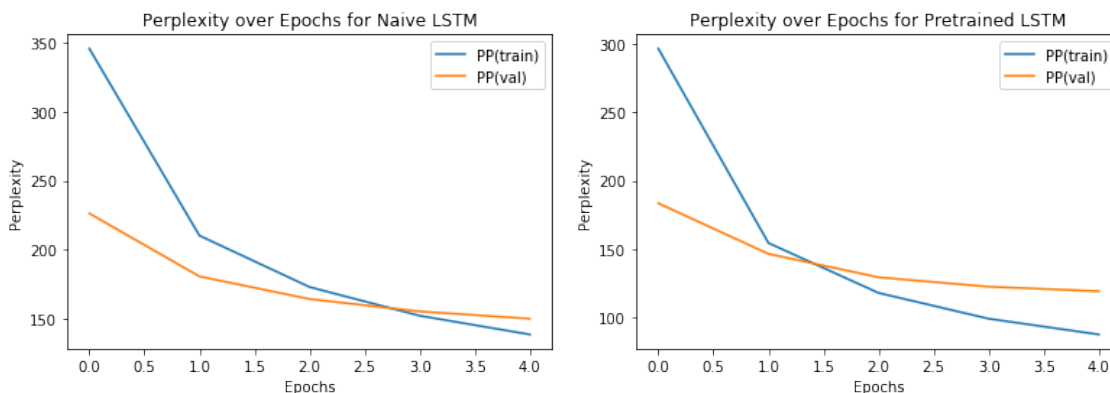


Figure 3: Train- and val-set perplexities over epochs for LSTM models with (L) random embedding initialization and (R) pretrained embedding initialization. $embed_dim = 300$, $lr = 1e^{-3}$, $hidden_dim = 650$, $weight_bound = 0.05$, $dropout = 0.5$

4.4 Extension 1

We now report information theoretic analyses of each of the models for the following training and testing contexts: training batch 1 (1), training batch 7 (2), testing batch 117 (3), and testing batch 231 (4). The entropies of each context as assessed by each model are reported in Table 3.

Model p	H(1~p)	H(2~p)	H(3~p)	H(4~p)
Trigram w/ Linear Interpolation ($a_1 = 0.7, a_2 = 0.1$)	2.5	2.5	4.0	3.9
Neural Network (pretrained embedding initialization)	6.1	5.7	5.7	5.5
LSTM (pretrained embedding initialization)	4.4	4.1	4.6	4.5

Table 3: Entropies of each training or testing context (above) under each model.

5 Conclusion

In this assignment, we implemented several approaches to language modeling: (1) a count-based trigram model with linear interpolation; (2) variants on a neural network model; and (3) variants on an LSTM. Trained with and evaluated against data from the Penn Treebank (Marcus et al., 1993). The resulting perplexity values (Tables 1, 2) show the LSTM models “learned” more rapidly than the neural networks, reaching lower perplexities in the same training timeframe (5 epochs). Further work might investigate approaches to fine-tuning the hyperparameters of these models for coherent comparisons of model efficacy, for example grid searches for efficient learning rates.

This assignment also highlighted practical considerations for implementing perplexity calculations. While conceptually straightforward, in practice it is clear further debugging is needed to work out why the perplexity values vary so greatly within the trigram models.

Finally, at submission time neither extension had been completed. We look to these as future areas of work.

References

- Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.
- Collins, M. (2013). Language Modeling. Course notes for NLP, Columbia University.
- Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.
- Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.