CS287 HW2: Language Modeling

Shayne O'Brien shayneob@mit.edu
David McClure dclure@mit.edu

February 13, 2018

1 Introduction

In this problem set we train three language models on the canonical Penn Treebank (PTB) corpus. This corpus is split into training and validation sets of approximately 929K and 73K tokens, respectively. We implement (1) a traditional trigram model with linear interpolation, (2) a neural probabilistic language model as described by Bengio et al. [1], and (3) a regularized Recurrent Neural Network (RNN) with Long-Short-Term Memory (LSTM) units following Zaremba et al. [6] We also experiment with a series of modifications to the LSTM model and achieve a perplexity of 92.9 on the validation set with a multi-layer model.

2 Problem Description

In probabilistic language modeling, we compute the probability of a sequence based on a training corpus. More formally, let $w \in V$ be a token in the corpus with vocabulary V, let $w_1, w_2, \ldots, w_n$ be an input sequence of token-length n, and let w_i be the ith token of that sequence. Our objective is to maximize the probability given by

$$P(w_1, w_2, \ldots, w_n) = \prod_i P(w_i | w_1, \ldots, w_{i-1}) \approx \prod_i P(w_i | w_{i-k}, \ldots, w_{i-1}) \tag{1}$$

where $k \in \mathbb{Z}$ is the fixed-size of our context window, defined as the number of tokens prior to the end of the input sequence that are taken into account when computing the probability. This is approximately equal to using all preceding tokens as opposed to a subset, by the Markovian assumption [2].

3 Models and Algorithms

For each model variant, we determine training effectiveness by computing perplexity on the held-out validation split of the PTB dataset; minimizing token-level perplexity is proportional to maximizing token-level probability [2]. Perplexity can be thought of as the average branching factor of the ability of the trained model to compactly represent a test dataset in bits:

$$2^{\frac{1}{n}\sum_{i=1}^{n} -\log_2 P(w_i | w_{i-k}, \ldots, w_{i-1})} \tag{2}$$

In practice, we compute the training and validation perplexity by exponentiating the average batch loss of a full pass over the corresponding dataset. This is an alternative definition of perplexity that is frequently used in deep learning.
We construct all models using Pytorch in Python 3. For all neural network architectures, we used a learning rate of 1e-3, a weight decay of 1e-5, a batch size of 32 tokens, Stanford's pre-trained word GloVe embeddings [5], the Adam optimizer, and the negative log likelihood loss function. Early stopping, defined here to be the point at which the validation perplexity stopped decreasing, was used on all models to determine number of training epochs.

3.1 Trigram Model with Linear Interpolation

Under linear interpolation, the probability of a word is modeled as a linear combination of the probabilities of different order n-gram contexts. In the trigram case:

$$P(w_t) = \alpha_1 P(w_t|w_{t-2}, w_{t-1}) + \alpha_2 P(w_t|w_{t-1}) + \alpha_3 P(w_t) \tag{3}$$

Such that $\sum_i \alpha_i = 1$. We use the expectation maximization (EM) algorithm to find values for α that minimize the perplexity on a development set. Ideally, we would work with all three splits of the PTB – a training set that is used to count maximum-likelihood estimates for ngram probabilities; a development set that is used to learn optimal values for α; and a test set used for evaluation. Since in this case we just have the PTB training and validation sets, we divided the validation set into two halves and used the first half as a development set for the EM algorithm and the second half as a test set. By way of EM, we set interpolation weights:

$$\alpha_1 = 0.1603 \tag{4}$$

$$\alpha_2 = 0.5090 \tag{5}$$

$$\alpha_3 = 0.3305 \tag{6}$$

3.2 Neural Probabilistic Language Model

We follow Bengio et al. [1] in implementing a neural probabilistic language model. Whereas in the trigram model we used token ngram counts and alpha values to compute probabilities, we now decompose this $P(w_i|w_{i-k}, \ldots, w_{i-1})$ into two parts:

1. An encoder function C for any $w \in V$ to a d dimensional embedding vector of real numbers $\mathbf{e_i} \in \mathbb{R}^{1xd}$ for each w_i in the input sequence. This yields a distributed feature vector output for each input context window c of size k, $\mathbf{e}_c \in \mathbb{R}^{kxd}$. The encoder is parametrized by $|V|$ x d parameters that need to be optimized by the network.

2. A probability function over encoded tokens in a context window $w_{i-k}, \ldots, w_{i-1}$ to a conditional probability distribution over tokens in V for the the next token w_i. This probability function is represented by mapping encoded tokens to an output softmax layer using a feed-forward or recurrent neural network, or another parametrized function with parameters ω. In the neural probabilistic language model, our parametrization is $\theta = (C, \omega)$.

In our implementation, we used a context window of size k=5 tokens and a single 100-unit hidden layer. The model trained for 13 epochs before early stopping.

3.3 Long Short-Term Memory Network

As per Zaremba et al. [6], assume that all states are n-dimensional. Let $h_t^l \in \mathbb{R}^n$ be a hidden state in layer l at timestep t, let $T_{n,m} : \mathbb{R}^n \to \mathbb{R}^m$ be an affine transformation ($\mathbf{W} \cdot x + b$ for some $\mathbf{W}$, b), let $\odot$ be element-wise multiplication, and let h_t^0 be an input word vector at timestep k. We use h_t^L to predict output y_t where L is the number of layers in our LSTM. Recall that for vanilla RNNs, the deterministic state transition is a function given by

$$h_t^l = f(T_{n,n}h_{t-1}^l + T_{n,n}h_t^{l-1}) \tag{7}$$

where f is a nonlinear activation function such as tanh or sigmoid.

In LSTMs, we afford the network a vector $c_t^l \in \mathbb{R}^n$ called a *memory cell* that stores the "long-term" memory of the network. The LSTM decides to overwrite, retrieve, or keep the memory cell at each time step t for the next time step t+1. Following Graves et al. (2013), the LSTM deterministic state transition equation is given by:

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

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

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

where $\mathbf{D}_p$ is the dropout operator that sets a random subset of its argument to zero with probability p. By applying dropout in this way, we regularize the network by targeting only non-recurrent connections. This selective application of dropout affects information flow within the network L+1 times.
As a baseline, we trained an LSTM with a single 1000-unit hidden layer with dropout probability p=0.50. This was trained for five epochs before early stopping.

3.4 Multi-layer LSTM with connections

As an extension of the LSTM baseline, we combined three 1-layer LSTMs of different unit sizes to make a pseudo-3-layer LSTM and provided connections to the lower-order layers in each layer's time step representations. The flow of this model was as follows:

1. First, each batch was passed through a 1-layer, 1000-unit LSTM. This output of this layer was then passed to a different 500-unit LSTM, which in turn passed its output to a third 200-unit LSTM. Each LSTM maintained a separate hidden state, which was persisted across forward passes to provide continuity between batches, as described by Zaremba [6]. Dropout was applied after each LSTM layer.

2. Instead of only using the final representations from the 200-unit LSTM, though – the outputs from the three LSTMs were concatenated together to form a single tensor of size 32x10x1700. This provided direct connections for the first two LSTMs in the final encodings, which then

were mapped by a linear layer to the softmax output for a probability distribution over the vocabulary just as was in the case of the baseline LSTM.

This model was trained for 5 epochs before early stopping.

4 Experiments

In addition to our multi-layer LSTM with connections, we also experimented with the following modifications:

1. Made the LSTM baseline deeper. Following the design decisions and hyperparameters described by Zaremba et al. [6], we first tried adding more hidden layers to the LSTM by increasing the `num_layers` attribute in the PyTorch LSTM implementation. We tried 2- and 3-layer models, each with between 500 and 2000 hidden units and with dropout probability $p = 0.50$.

2. Replaced the GloVe embeddings with the GoogleNews embeddings [4]. This idea came from the thought that there might be some useful domain specificity for PTB as these embeddings were trained on news articles.

3. Implemented "multi-channel" embeddings as described by Kim [3] in the context of CNN architectures. Instead of just using a single embeddings layer that is updated during training, the pre-trained weights matrix is copied into two separate embedding layers: one that is updated during training, and another that is omitted from the optimizer and allowed to remain unchanged during training. During a forward pass word indexes are mapped to each table separately, and then the two tensors are concatenated along the embedding dimension to produce a single, 600-dimension embedding tensor for each token.

4. Experimented with different approaches to batching. Instead of modeling the corpus as a single, unbroken sequence during training (such as with torchtext's `BPTTIterator`), we tried splitting the corpus into individual sentences and then producing separate training cases for each token in each sentence. For example, for the sentence "I like black cats" we produced five contexts:

   (a) "`eos` I"
   (b) "`eos` I like"
   (c) "`eos` I like black"
   (d) "`eos` I like black cats"
   (e) "`eos` I like black cats `eos`"

   And the model is trained to predict the last token in each context at time step t from the first $t - 1$ tokens. We used PyTorch's `pack_padded_sequence()` to handle variable-length inputs to the LSTM. Practically, this was appealing because it makes it easier to engineer a wider range of features from the context before a word – for example, it becomes easy to implement bidirectional LSTMs with both a forward and backward pass over the $t - 1$ context, which, to our knowledge, would be difficult or impossible under the original

training regime enforced by `BPTTIterator`. We realized after trying this, though, that it will never be competitive with `BPTTIterator`'s continuous representation of the corpus because the sentences in the corpus are grouped by article – and thus also at a thematic / conceptual level – which means that the model can learn useful information across the sentence boundaries about what type of word should come next.[1]

5. Experimented with different regularization strategies – varying the dropout percentages, whether dropout is applied to the initial embedding layers, etc.

But, none of these changes improved on the initial single-layer, 1000-unit LSTM. Our best performing model was the one described in Section 3.4. The perplexities we achieved with each of our Section 3 models is described in Table 1.

Model	Perplexity
LINEARLY INTERPOLATED TRIGRAM**	179.2
NNLM (5-GRAM)	162.2
1-LAYER LSTM	101.5
3-LAYER LSTM + CONNECTIONS	**92.9**

*Table 1: Perplexities on the Penn Treebank validation set. **The linearly interpolated trigram was evaluated on just 50% of the validation set, since the other half was used as a development set to learn values for the weighting parameters via expectation maximization.*

Though the multi-layer LSTM with connections beat the simple LSTM baseline, we were unable to replicate the 78.4 validation perplexity performance described by Zaremba et al. using the same corpus and similar architectures. Namely, when using the configurations described in the paper (the 2-layer, 650- and 1500-unit architectures), our models overfit within 5-6 epochs, even when applying dropout in a way that matched the approach described in the paper. (Zaremba et al., by contrast, mention training for as many as 55 epochs.)

5 Conclusion

We trained four classes of models – a traditional trigram model with linear interpolation, with weights learned by expectation maximization; a simple neural network language model following Bengio et al.; a single-layer LSTM baseline; and an extension to this model that uses three layers of different sizes, skip connections for the first two layers, and regularization as described by Zaremba et al. The final model achieves a perplexity of 92.9, compared to 78.4 and 82.7 reported by Zaremba et al. using roughly equivalent hyperparameters.

References

[1] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin. "A Neural Probabilistic Language Model." Journal of Machine Learning Research 3 (2003) 11371155.

[1] In this sense the task shades into "document modeling," or inferring the structure of higher-order units of discourse.

[2] D. Jurafsky. ”Language Modeling: Introduction to N-grams.” Lecture. Stanford University CS124. 2012.

[3] Y. Kim. ”Convolutional Neural Networks for Sentence Classification.” Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 17461751, October 25-29, 2014, Doha, Qatar.

[4] T. Mikolov, K. Chen, G. Corrado, J. Dean. ”Efficient estimation of word representations in vector space.” arXiv preprint arXiv:1301.3781, January 1, 2013.

[5] J. Pennington, R. Socher, C. Manning. 2014. ”GloVe: Global Vectors for Word Representation.”

[6] W. Zaremba, I. Sutskever, O. Vinyals. 2015. ”Recurrent Neural Network Regularization.” arXiv preprint arXiv:1409.2329, February 19, 2015.