Homework 3

Ankit Gupta
ankitgupta@college.harvard.edu

Tom Silver
tsilver@college.harvard.edu

March 10, 2016

Code: `https://github.com/ankitvgupta/CS287assignments/tree/master/HW3`. The Lua code is included at the end of this report. All other code, including Odyssey scripts, are on the Github.
Kaggle Team ID: Ankit_Tom

1 Introduction

In this assignment, we look at the problem of language modeling, which involves estimating the distribution of possible next words given a context of previous words. In theory, the next word may depend on arbitrarily distant previous words, which can make it very difficult and computationally expensive to capture important dependencies. For this assignment, we assume that it suffices to consider a fixed number of words in the context leading up to the target of prediction. The challenge then becomes handling the inherent sparsity of data as we increased the size of the considered context. We first employ two variants of smoothing — Laplace and Witten-Bell – to deal with this issue. Next we implement a Neural Network Language Model (NNLM), which allows us to take advantage of both the positional information of words in the context and the words themselves. Unfortunately, the standard NNLM relies on an expensive softmax computation over the entire vocabulary. To increase efficiency while maintaining similar perplexity, we introduce a NNLM with Noise Contrastive Estimation (NCE) as our third class of language models. We supplement our initial results with additional hyperparameter optimization experiments and a set of post-processing analyses of the trained models.

2 Problem Description

The focus of this assignment is language modeling. We characterize language modeling as an instance of multiclass classification, where the classes correspond to words in the vocabulary, and the inputs are the words preceding the target word. Formally, given a vocabulary $\mathcal{V}$, we wish to estimate $p(w_t|w_1, w_2, ..., w_{t-1})$, where $w_i \in \mathcal{V}$ for $i = 1, ..., t$. In practice, we will consider only d of the preceding words, assuming $p(w_t|w_1, w_2, ..., w_{t-1}) \approx p(w_t|w_{t-d+1}, ..., w_{t-1})$. Thus to train a model, we are given inputs in the form $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, where $\boldsymbol{x}_i$ represents the prefix $(w_{t-d+1}, ..., w_{t-1})$ and $\boldsymbol{y}_i$ is a one-hot encoding of w_t.

To describe the performance of our language models, we report perplexity, which is defined as $perp = exp(-\frac{1}{n}\sum_{i=1}^{n} \log p(w_t|w_1, w_2, ..., w_{t-1})$, i.e. the exponeniated negative log likelihood.

Perplexity is an informative metric for language modeling because it reflects the average size of a uniform distribution that would correspond to the same probability of target words. Thus our goal in training a language model is to minimize perplexity as empirically determined via evaluation on the validation set.

3 Model and Algorithms

3.1 Count-Based Language Models

We first implement three language models that operate according to the observed counts of n-grams. All three assume that the probabilities of interest, $p(w_t|w_1, w_2, ..., w_{t-1}))$, are from multinomial distributions. This assumption is used to derive closed form solutions for the maximum likelihoods that are based only on the n-gram counts. The two smoothed models attempt to reasonably account for n-grams unseen in the training set. In the descriptions below, we use $F_{c,w}$ to represent the observed counts of word w in context c as stored in the matrix F, $N_{c,w} = 1(F_{c,w} > 0)$, and $\cdot$ to represent summing over the respective dimension of the matrix.

It is important to note that for larger context sizes (even as little as 2), the matrix that holds the counts for each context-word pair will become extremely sparse. Moreover, for contexts of 4 or 5, this matrix becomes prohibitively large to represent in this format. Thus, for all of our count-based language models, we implemented the Reverse Trie data structure that was described in lecture. Each level of this trie represented a word from the context in reverse order. Furthermore, at each node, there was a dictionary that stored the counts for each of the target words. Thus, to find the number of words for a given context, we just looked up the context in the trie (which was fast since it only took O(d_{win}) steps), and then returned the count matrix that was at that context. If the context did not exist, we just returned an empty matrix.

Importantly, due to this sparsity, we often ran into cases in the validation and test sets that were not in the training set. So, we used Laplace Smoothing to solve this problem, and essentially gave a bit of weight to each of the values in the trie. The smoothing variable α became a hyperparameter to the model.

Ultimately, we ended up using this trie in the all of the count-based models that we wrote, and we used it to get the unigram distribution needed for the Noise Contrastive Estimation Neural Network model, which we will discuss in a later section of this report.

3.1.1 Maximum Likelihood Estimation

For a simple baseline, we begin with Maximum Likelihood Estimation. Using this method $p(w)$ is proportional to the count frequency of that word in the training set.

3.1.2 Laplace Smoothing

From the assumption of multinomial distributions, we get estimates of $p(w|c)$ according to

$$\frac{F_{c,w}}{F_{c,\cdot}}$$

or 0 in the case of $F_{c,\cdot} = 0$. Clearly this model is problematic if observed counts are 0, which is increasingly likely as the size of c increases. This observation motivates smoothing.

To account for unobserved but theoretically possible combinations of contexts and target words, we implement Laplace Smoothing as a simple improvement over straightforward MLE. Laplace smoothing dictates that we use $\overline{F}_{c,w} = F_{c,w} + \alpha$, for all c, w and for a fixed α, in place of F in the MLE calculation. Then α becomes a tunable hyperparameter of the model. As stated before, getting these count values required just a fast $O(d_{win})$ lookup from the reverse trie.

3.1.3 Witten-Bell

For more sophisticated smoothing, we turn to Witten-Bell. The high level motivation for Witten-Bell and other similar smoothing methods is that we would like to be able to weight the contributions of different n-gram sizes variably according to what we have observed in the training data. For example, if the context is "of Artificial", the probability that the target word is "Intelligence" is probably roughly the same as it would be if we just considered the context "Artificial"; however, if the context is "Natural Language", the probability that the target word is "Processing" will be greatly influenced by the inclusion of "Natural". Witten-Bell attempts to systematically address these situations using interpolation, which estimates the probability of a target word in a context as a weighted sum of probabilities of the word in successively smaller contexts:

$$p_{wb}(w|c) = \lambda(w,c)p_{ML}(w|c) + (1 - \lambda(w,c))p_{wb}(w|c')$$

where c' is c without its first word, p_{ML} denotes the maximum likelihood probability as described above, and

$$\lambda(w,c) = 1 - \frac{N_{c,\cdot}}{N_{c,\cdot} + F_{c,\cdot}}$$

Rearranging gives us

$$p_{wb}(w|c) = \frac{F_{c,w} + N_{c,\cdot} \times p_{wb}(w|c')}{N_{c,\cdot} + F_{c,\cdot}}$$

In order to implement this model, we again turn to the reverse trie data structure that was previously described. Essentially, our training step generates the entire trie. Then, in order to get the probability of a word given a context, we just perform a lookup on a context, and then on the recursively smaller contexts, which each take $O(\text{length}(c))$ steps. The ability of the reverse trie data structure to efficiently store large contexts allowed us to scale this up to context sizes of 5 (which is 6-gram estimation).

3.2 Neural Network Language Models

In a neural-network based language models, we generally follow the same high-level structure of the past problem sets. Our neural network model is a single hidden-layer MLP, where the inputs are first transformed by performing a lookup of the embeddings for a context's words, and then concatenating those together. In other words, our model's input is the vector

$$\boldsymbol{x} = [v(w_0), v(w_1), \ldots, v(w_{d_{win}})]$$

where $v(w_i)$ is the embedding for the ith word in the context. The outputs y were one-hot encoded representations of the true next word. Then, we simply trained an MLP with a single hidden-layer, and the nonlinearity we used was Tanh. Thus, the final model looked like

$$\hat{y} = softmax(tanh(\boldsymbol{x}W_1 + b_1)W_2 + b_2)$$

Our training data was essentially just contexts followed by a word, and we used a cross-entropy loss to train this neural network.

3.3 Noise Contrastive Estimation

In the previous assignment, we could get away with full and frequent softmax computations because of the relatively small number of classes. Unfortunately, language modeling requires one class per word in the vocabulary, which leads to a massive slowdown in softmax computation due to the summation over all classes in the denominator. Yet the softmax is critical for language modeling since we are ultimately interested in the full probability distributions of words in contexts. Thus we turn to Noise Contrastive Estimation (NCE), a method for neural language modeling which claims far greater efficiency and comparable perplexity to the standard NNLMs.

NCE is derived by recasting language modeling as a binary classification problem. Using the distributional hypothesis of linguistics, we assume that the observed natural language samples are *coherent*, in the sense that they are derived from a constrained distribution that is implicitly defined by human language. We will denote this *data distribution* $p_d(w|c)$, loosely following the notation of Mnih and Teh (2012). In contrast, contrived samples drawn from a *noise distribution*, i.e. according to maximum likelihood probabilities, are assumedly *incoherent*. We will denote the noise distribution $p_n(w)$; note that it is independent of context. For a given context c and word w, the binary classification problem is then to model the conditional probabilities $p(D = 1|w, c)$ and $p(D = 0|w, c)$, i.e. the probabilities that the word and context are coherent or incoherent respectively. For all true samples in the dataset, we would hope to see high $p(D = 1|w, c)$. If we instead augment the dataset with "fake" words w' drawn from the unigram distribution, we would like to find high $p(D = 0|w', c)$.

Suppose that we augment the original dataset with K fake samples for every true sample. Thus data are assumedly drawn from the distribution $\frac{1}{K+1} p_d(w|c) + \frac{K}{K+1} p_n(w)$. Then the probability of a sample being coherent is

$$p(D = 1|w, c) = \frac{p_d(w|c)}{p_d(w|c) + Kp_n(w)}$$

$$p(D = 1|w, c) = \sigma(\log p_d(w|c) - \log Kp_n(w))$$

We can then calculate log-likelihood loss accordingly:

$$\mathcal{L} = \sum_i \log \sigma(\log p_d(w_i|c_i) - \log (Kp_n(w_i))) + \sum_{k=1}^{K} \log (1 - \sigma(\log p_d(s_{i,k}|c_i) - \log (Kp_n(s_{i,k}))))$$

where $s_{i,k}$ is a sample drawn from the unigram distribution. Thus far we have not reduced the computational complexity, since buried in the equation for loss is $p_d(w_i|c_i)$, which requires the same softmax calculation that we have been trying to avoid. Here we introduce the key resource-saving assumption of NCE: it is safe to replace $p_d(w_i|c_i)$ with the unnormalized $z_{w_i} = tanh(\boldsymbol{x}_i W_1 + b)W_2 + b)$ divided by a parameterized partition function $Z(c)$. Moreover, it is even safe to assume $Z(c) = 1$, leaving the unnormalized z_{w_i} in the final loss:

$$\mathcal{L} = \sum_i \log \sigma(z_{w_i} - \log (Kp_n(w_i))) + \sum_{k=1}^{K} \log (1 - \sigma(z_{s_{i,k}} - \log (Kp_n(s_{i,k}))))$$

Thus the loss computation is tractable even with large vocabulary.

For our implementation, this involved first using a model similar to the straightforward NNLM, except rather than having a full linear layer after the Tanh layer, we had a lookup layer. In this layer, we looked up the rows that corresponded to the true output and the K noise distribution samples, and then did a linear transformation over those, followed by a sigmoid. We also similarly had a bias term, which was implemented with a lookup table with just a single bias number per word in the vocabulary. This is fairly similar to the method that Johnny Ho posted on Piazza, as it involved creating a small network for calculating the loss, as well as the derivative with respect to the sigmoid. We did the noise sampling by first generating a huge amount of samples (millions) in Python, and then feeding those in.

4 Experiments

4.1 Preprocessing: Exploratory Analysis

In the previous assignment, we conducted a range of simple exploratory analyses to characterize the Penn Treebank dataset. Since we are using the same underlying data in this assignment, we will not provide a reproduction of these analyses, though they were doubly useful. Instead we present additional preprocessing experiments with particular relevance to the language modeling problem. An informative metric to gauge the difficulty of language modeling is the distribution of nonzero values in N, i.e. the number of unique words per context. These values are intriguing not only because of the role of N in Witten-Bell smoothing, but also more generally as a descriptive statistic of the dataset. The effect of context size on the distribution is also worthy of characterization. These distributions for context sizes 1 through 4 are plotted in Figure 1.

4.2 Experimental Results for Main Models

The cross-entropy loss and perplexity values in this section were calculated over the distribution of 50 options given in the assignment. We performed these tests for the requested context sizes, and a summary of the results is in Table 1. It is important to note that because the smaller vocabulary (1000) dataset has many more unknown words, the "options" provided often contain several "unknown" tokens, and thus the probability mass is divided between them. This explains the worse performance on the smaller vocabulary in this table. For reference, you should only compare different algorithms within the same dataset, not a single algorithm between vocabularies. The summary of the main model performance is in Table 1.

Figures 2 and 3 respectively show that the loss is decreasing over successive epochs and iterations of these algorithms.

4.3 Extension: Grid Search for Hyperparameter Optimization

For each of the main models, we use the Harvard Odyssey Research Cluster to test hundreds of combinations of hyperparameters, in order to find the ones that minimized the perplexity. First, we performed this optimization on the simple count-based multinomial model with Laplace smoothing. We essentially modified the smoothing parameter, and the amount of context that we were using. We did this test on both the full dataset and small one, with contexts up to size 5. The best results using Laplace smoothing are in Table 2.

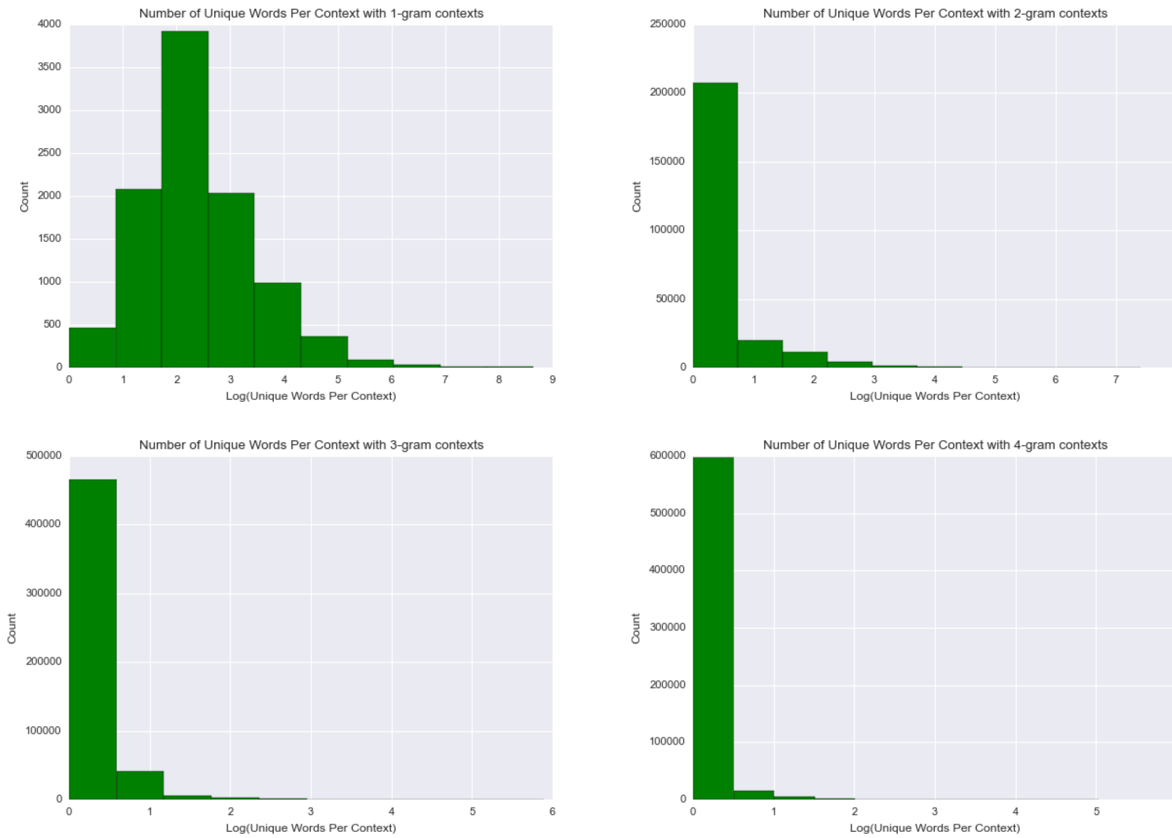

Figure 1: Log histograms of unique word counts per context for context sizes 1, 2, 3, and 4. As expected, unique word counts per context decrease as context size increases.

	Vocab size = 1000			Vocab size = 10000		
Model	Accuracy	Loss	Perplexity	Accuracy	Loss	Perplexity
Multinomial MLE	.247	7.05	1156.5	.559	1.796	6.023
Laplace, $n = 2$	0.328	5.88	359.38	0.65	1.51	4.53
Laplace, $n = 3$	0.38	5.10	164.33	0.41	2.44	11.47
Multinomial WB, $n = 2$	.377	4.18	65.3	0.65	1.41	4.11
Multinomial WB, $n = 3$	.377	4.18	65.3	0.66	1.38	3.98
NNLM	.34	6.188	486.8	.595	1.694	5.44
NCE	-	-	-	.6747	1.479	4.39

Table 1: Table with the results of main models. Make sure to compare across rows, not between the 1000 and 10000 datasets, because of behavior caused by repeated rare words in the options given for the 1000 vocab version.

Next, we perfomed this same analysis except using the Witten-Bell strategy for interpolation. The results of this test are shown in Table 3.

Next, we implemented the NNLM, and performed the same hyperparameter optimization,

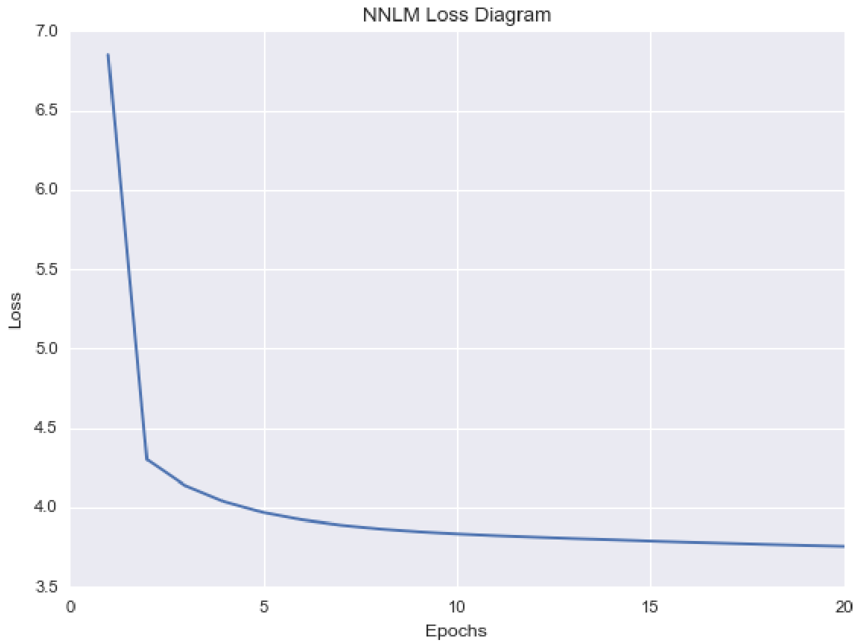

Figure 2: This shows the losses decreasing over many epochs of training NNLM.

Dataset	d_{win}	α	Accuracy	Cross-Entropy Loss
PTB	1	0.01	0.65	1.51
PTB	1	0.1	0.65	1.58
PTB	1	0.001	0.65	1.64
PTB	1	0.0001	0.65	1.81
PTB	1	1	0.65	2.02
PTB	1	4	0.65	2.42
PTB	2	0.01	0.41	2.44
PTB	2	0.001	0.41	2.49
PTB	2	0.1	0.41	2.58
PTB	2	0.0001	0.41	2.59

Table 2: Table with the results of Hyperparameter Optimization for Laplace Smoothing.

with a 12 hour timeout. Table 4 has the best results among those that finished within this time. Generally, keeping a relatively small embedding and hidden layer size helped speed this up.

Next, we performed the same optimization for NCE. Here, we can again see that we can get fairly good performance using the full vocabulary. Interestingly, we were able to get slightly better accuracy, though slightly worse perplexity. In any case, we got perplexity values of around 4, which means we are reducing the effective distribution from the 50 options to just around 4. The data for NCE is in Table 5.

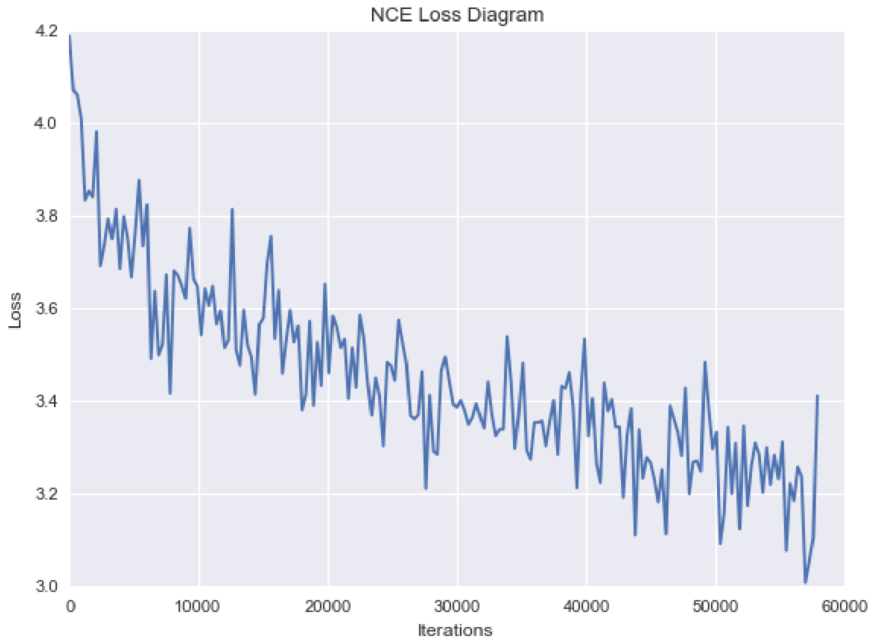

Figure 3: This shows the losses decreasing over many iterations of training NCE. Note the noise is due to each of these losses being calculated for a minibatch, and there being some noise between minibatches. The main trend of it going downwards is the key fact here.

4.4 Extension: Babbling

For a final extension, we implemented a simple babbling script that generates sentences by sampling from the probability distributions learned from our models. In particular, we babbled from the probability distributions created using Witten-Bell smoothing with context lengths of 1, 2, 3, 4, and 5. We did the sampling in Lua by sampling from a uniform distribution, and then employing Universality of the Uniform to transform that uniform distribution sample to a sample from the categorical distribution that the model returns. This essentially amounted to a for loop. We also experimented with the full vocabulary and the smaller 1000 word voabulary. To handle the lack of sentence starts in the target space of our model, and to allow for arbitrarily long babbling we replaced all produced end tokens with start tokens. Additionally we excluded the "unknown" token from the generated probability distributions and renormalized in order to achieve more meaningful and interpretable results. We include representative samples of the babbled output in Table 6. There were a few notable phenomena that we noticed here. For one, the smaller vocabulary training data seemed to produce clearer sentences, and this is probably because the reduced vocabulary size reduces the number of options. Furthermore, the sentences generated after training on a larger context size dataset appear to show generally better grammatical structure, which is the consistent with what we would expect from being able to see more than just 1 previous word.

Dataset	d_{win}	α	Accuracy	Cross-Entropy Loss
PTB	5	0.01	0.667656	1.363323
PTB	4	0.01	0.667062	1.364487
PTB	3	0.01	0.665875	1.367611
PTB	5	0.001	0.667656	1.370471
PTB	4	0.001	0.667062	1.371929
PTB	3	0.001	0.665875	1.374493
PTB	5	0.0001	0.667656	1.375224
PTB	4	0.0001	0.667062	1.375390
PTB	3	0.0001	0.665875	1.376758
PTB	2	0.01	0.662018	1.376776
PTB	2	0.001	0.662018	1.382148
PTB	2	0.0001	0.662018	1.383384
PTB	2	0.1	0.662018	1.401239
PTB	1	0.01	0.648368	1.411272
PTB	1	0.001	0.648368	1.413454
PTB	1	0.0001	0.648368	1.413773
PTB	1	0.1	0.648368	1.414972
PTB	3	0.1	0.666172	1.420453

Table 3: Table with the results of Hyperparameter Optimization for Witten-Bell.

Dataset	d_{win}	η	Minibatch	Hidden	EmbeddingSize	Accuracy	CrossEntropy	Perplexity
PTB	5	100	32	50	25	0.594	1.694	5.44
PTB	1	100	32	10	25	0.561	1.887	6.60
PTB	5	1	32	10	50	0.560	1.890	6.62
PTB	1	100	32	10	50	0.558	1.893	6.64
PTB	1	0.01	32	10	50	0.560	1.893	6.64
PTB	1	0.001	32	10	50	0.566	1.894	6.64
PTB	1	0.01	32	10	100	0.561	1.895	6.65
PTB	5	100	32	10	25	0.564	1.899	6.68
PTB	5	100	32	10	50	0.561	1.899	6.68
PTB	2	100	32	10	100	0.563	1.899	6.68
PTB	2	0.01	32	10	50	0.559	1.900	6.68
PTB	5	1	32	10	100	0.562	1.901	6.69
PTB	5	0.001	32	10	100	0.567	1.902	6.70
PTB	5	0.001	32	10	25	0.566	1.902	6.70
PTB	5	0.01	32	10	50	0.561	1.902	6.70

Table 4: Table with the results of Hyperparameter Optimization for NNLM.

5 Conclusion

Overall, this assignment has demonstrated many of the challenges that are involved in language modeling. For one, we find that simple count-based models actually have pretty good perfor-

d_{win}	α	η	NumEpochs	HiddenLayers	EmbeddingSize	Accuracy	CrossEntropy	Perplexity
5	1	10	40	50	50	0.674777	1.479	4.392
3	0.0001	10	40	50	50	0.670030	1.485	4.417
1	0.0001	10	40	50	50	0.659050	1.490	4.440
2	0.0001	10	40	50	50	0.664985	1.494	4.458
4	1	10	40	50	50	0.660534	1.496	4.464
2	1	10	40	50	50	0.664095	1.498	4.475
1	1	10	40	50	50	0.654303	1.508	4.519
1	1	10	40	100	50	0.663205	1.516	4.557
3	1	10	40	50	50	0.665875	1.524	4.592
1	1	10	40	50	50	0.656083	1.528	4.609
2	1	10	40	50	50	0.656677	1.530	4.622
5	0.0001	10	40	50	50	0.662908	1.536	4.646
1	1	10	20	50	50	0.650148	1.541	4.673
2	0.0001	10	40	50	50	0.659347	1.543	4.682
2	0.0001	10	20	50	50	0.653412	1.546	4.693
3	0.0001	10	20	50	50	0.653412	1.548	4.703

Table 5: Table with the results of Hyperparameter Optimization for NCE. These were all using the full PTB dataset with the full vocabularity, with minibatch size of 128.

mance with interpolation methods like Witten-Bell. Furthermore, when using traditional neural network models, this assignment demonstrated the difficulty with calculating the partition function in this setting, since the number of classes can be very large. This leads to very slow training speeds, as training on the full PTB dataset with a vocabulary of 10000 needed 10+ hours of training time. Thus, methods like NCE can relatively quickly get good performance, and avoid calculating these large softmax results. Notably, it took several times more training time (probably around 5x) to train similar sized NNLM and NCE models.

Lastly, through our babbling experiments, we found that this was able to produce sentences that show some grammatical structure with sufficient context, and that, in some cases, seem completely reasonable. Also, important properties, like dollar signs being followed by numbers, can be easily captured by this type of model. Even though there are limitations to the sentences it produces, this shows that even simple models can capture reasonable relationships between words and some grammatical structure.

References

Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*.

Context Size	Vocab Size	Babble Sample
1	10000	la net as billion party rich may N ended loans
2	10000	but while computer to the fort that it and standing of european posed section massive away clarify income a record of N N convertible house is link on books schools restructured to N N jobs restructuring would little from N N days N to $ N to the gramm-rudman
3	10000	the first quarter managers the sand said action the australian treasury may martin birds
4	10000	columbia scott into declined her today
5	10000	others divestiture to another million was petrochemical including about california windsor when inquiries would film london ever a slow lot insider to bulls N days japanese expanded fell wanted stability market daily years at a did fidelity it company
1	1000	the federal reserve also was the economy of the stock the case against judge trust well as the company 's N i was a similar i still its stake in consumer increase stock market says inc. chicago a which had continue may come department
2	1000	base what he japanese president claims
3	1000	in addition to government may seek
4	1000	they open down the same money well in the recent this said
5	1000	the private me before the three few low or as mr. these days when strong of # N N in assets of an investor in agreed to work communications corp to get were

Table 6: Representative sentences generated through babbling on Witten-Bell probability distributions.

Code

The full, organized codebase can be found at `https://github.com/ankitvgupta/CS287assignments/tree/master/HW3`. The Lua code is included here. All other code, including Odyssey scripts, are on the Github. The README contains descriptions of the files.

Listing 1: HW3.lua: Controller File

```
-- Only requirements allowed
require("hdf5")
require("nn")
require("optim")

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-datafile', '', 'data file')
cmd:option('-testfile', '', 'test file')
cmd:option('-save_losses', '', 'file to save loss per epoch to (leave blank if not
    wanted)')
```

```
cmd:option('-save_weights', '', 'file to save lookuptable weights to (nce only)')
cmd:option('-classifier', 'nn', 'classifier to use (mle, nn)')
cmd:option('-alpha', 1, 'laplacian smoothing factor')
cmd:option('-eta', .1, 'Learning rate (.1 for adagrad, 500 for sgd, 10 for nn sgd)'
    )
cmd:option('-lambda', 0, 'regularization penalty (not implemented)')
cmd:option('-minibatch', 2000, 'Minibatch size (500 for nn, 2000 for lr)')
cmd:option('-epochs', 20, 'Number of epochs of SGD')
cmd:option('-optimizer', 'sgd', 'Name of optimizer to use (adagrad or sgd)')
cmd:option('-hiddenlayers', 10, 'Number of hidden layers (if using neural net)')
cmd:option('-embedding_size', 50, 'Size of word embedding')
cmd:option('-odyssey', false, 'Set to true if running on odyssey')
cmd:option('-K', 10, 'for NCE only')

-- Hyperparameters
-- ...

function sampler(dist)
  -- Do this to remove <unk> values.
  dist[2] = 0
  dist:div(dist:sum())

  --local _, ind = torch.max(dist, 1)
  --return ind:squeeze()

  local sample = torch.uniform()
  total = 0
  for i =1, dist:size(1) do
    total = total + dist[i]
    if total > sample then
      return i
    end
  end
  return dist:size(1)
end

function main()
   -- Parse input params
   opt = cmd:parse(arg)

   --print("Datafile:", opt.datafile, "Classifier:", opt.classifier, "Alpha:",
       opt.alpha, "Eta:", opt.eta, "Lambda:", opt.lambda, "Minibatch size:",
       opt.minibatch, "Num Epochs:", opt.epochs, "Optimizer:", opt.optimizer, "
       Hidden Layers:", opt.hiddenlayers, "Embedding size:", opt.embedding_size)


   _G.path = opt.odyssey and '/n/home09/ankitgupta/CS287/CS287assignments/HW3/' or
       ''

   dofile(_G.path..'train.lua')
   dofile(_G.path..'multinomial.lua')
   dofile(_G.path..'utils.lua')

   printoptions(opt)

   local f = hdf5.open(opt.datafile, 'r')
```

```
--local samples = nil
local f2 = hdf5.open(_G.path..'samples.hdf5')
local samples = f2:read("samples"):all():long()
print("Sample dist", torch.min(samples), torch.max(samples))

local nclasses = f:read('numClasses'):all():long()[1]
local nfeatures = f:read('numFeatures'):all():long()[1]
local d_win = f:read('d_win'):all():long()[1]

print("nclasses:", nclasses, "nfeatures:", nfeatures, "d_win:", d_win)

local training_input = f:read('train_input'):all():long()
local training_output = f:read('train_output'):all():long()

local valid_input = f:read('valid_input'):all():long()
local valid_output = f:read('valid_output'):all():long()
print("Full valid size", valid_input:size(), valid_output:size())


local valid_blanks_input = f:read('valid_blanks_input'):all():long()
local valid_blanks_options = f:read('valid_blanks_options'):all():long()
local valid_blanks_outputs = f:read('valid_blanks_output'):all():long()

local test_blanks_input = f:read('test_blanks_input'):all():long()
local test_blanks_options = f:read('test_blanks_options'):all():long()

local result

-- Train neural network.
if opt.classifier == "nn" then
            local model, criterion, embedding = neuralNetwork(nfeatures,
                opt.hiddenlayers, nclasses, opt.embedding_size, d_win)
            model = trainModel(model, criterion, training_input,
                training_output, valid_blanks_input, valid_blanks_options,
                valid_blanks_outputs, opt.minibatch, opt.epochs, opt.optimizer,
                opt.save_losses)
              local acc, cross_entropy_loss = getaccuracy2(model,
                  valid_blanks_input, valid_blanks_options, valid_blanks_outputs
                  )
            --result = predictall_and_subset(model, valid_input, valid_options,
                 nclasses, opt.alpha)
            --local acc = get_result_accuracy(result, valid_input,
                valid_options, valid_true_outs)
   local full_cross_ent = NNLM_CrossEntropy(model, valid_input, valid_output)
     printoptions(opt)
     print("Results (Accuracy,SmallCrossEntropy,SmallPerp, FullCrossEntropy,
         Fullperp):", acc, cross_entropy_loss, torch.exp(cross_entropy_loss),
         full_cross_ent, torch.exp(full_cross_ent))

     --print(acc)
     if (opt.save_weights ~= '') then
            print(embedding.weight)
            print("saving weights to", opt.save_weights, "...")
            local fsave = hdf5.open(opt.save_weights, 'w')
            fsave:write('embedding', embedding.weight)
            print("done")
```

```
        end

            --print("Accuracy:")
            --print(getaccuracy(model, valid_input, valid_options,
               valid_true_outs))
    elseif opt.classifier == 'nce' then
local reverse_trie = fit(training_input, training_output)
local distribution = normalize_table(get_word_counts_for_context(reverse_trie,
    torch.LongTensor{}, nclasses, opt.alpha))
local p_ml_tensor = table_to_tensor(distribution, nclasses)

            local model, lookup, bias, embedding = trainNCEModel(training_input
               , training_output,
                                valid_blanks_input,
                                valid_blanks_options,
                                valid_blanks_outputs,
                                opt.minibatch,
                                opt.epochs,
                                opt.optimizer,
                                opt.save_losses,
                                nfeatures, opt.hiddenlayers, nclasses,
                                    opt.embedding_size, d_win, opt.alpha,
                                    opt.eta, samples, opt.K, p_ml_tensor,
                                    valid_input, valid_output)

local acc, cross_entropy_loss, perplexity = getNCEStats(model, lookup, bias,
    valid_blanks_input, valid_blanks_options, valid_blanks_outputs, p_ml_tensor)

--local predictions = NCE_predictions(model, lookup, bias, valid_input,
    valid_options)
--print(predictions:sum(2))
--local acc,cross_entropy_loss = get_result_accuracy(predictions, valid_input,
    valid_options, valid_true_outs), cross_entropy_loss(valid_true_outs,
    predictions, valid_options)
-- Combine the models to a normal nn model for making predictions
--local prediction_model = make_NCEPredict_model(model, lookup, bias,
    opt.hiddenlayers, nclasses)
--local acc, cross_entropy_loss = getaccuracy2(prediction_model, valid_input,
    valid_options, valid_true_outs)
local full_cross_ent = NCE_predictions2(model, lookup, bias, valid_input,
    valid_output, opt.hiddenlayers, nclasses)
printoptions(opt)
print("Results(Acc,Cross,Perp,FullCross,FullPerp):", acc, cross_entropy_loss,
    perplexity, full_cross_ent, torch.exp(full_cross_ent) )

if (opt.save_weights ~= '') then
    print(embedding.weight)
    print("saving weights to", opt.save_weights, "...")
    local fsave = hdf5.open(opt.save_weights, 'w')
    fsave:write('embedding', embedding.weight)
    print("done")

end
```

```
elseif opt.classifier == 'multinomial' then
    local reverse_trie = fit(training_input, training_output)
    --print(get_word_counts_for_context(reverse_trie, torch.LongTensor{},
       nclasses, opt.alpha))
              local predicted_distributions = predictall_and_subset(
                 reverse_trie, valid_blanks_input, valid_blanks_options,
                 nclasses, opt.alpha)
    --print(predicted_distributions:sum(2))
    local cross_entropy_loss = cross_entropy_loss(valid_blanks_outputs,
       predicted_distributions, valid_blanks_options)
    print("Cross-entropy loss", cross_entropy_loss)

    --result = predictall_and_subset(reverse_trie, test_context, test_options,
       nclasses, opt.alpha)

    local acc = get_result_accuracy(predicted_distributions, valid_blanks_input
       , valid_blanks_options, valid_blanks_outputs)
  result = torch.exp(predictall_and_subset(reverse_trie, test_blanks_input,
     test_blanks_options, nclasses, opt.alpha))
    printoptions(opt)
    print("Results:", acc, cross_entropy_loss, torch.exp(cross_entropy_loss))
elseif opt.classifier == 'laplace' then
  local reverse_trie = fit(training_input, training_output)
  --print(get_word_counts_for_context(reverse_trie, torch.LongTensor{},
     nclasses, opt.alpha))
  local predicted_distributions = getlaplacepredictions(reverse_trie,
     valid_blanks_input, valid_blanks_options, nclasses, opt.alpha)
  --print(predicted_distributions:sum(2))
  local cross_entropy_loss = cross_entropy_loss(valid_blanks_outputs,
     predicted_distributions, valid_blanks_options)
  print("Cross-entropy loss", cross_entropy_loss)

  --result = predictall_and_subset(reverse_trie, test_context, test_options,
     nclasses, opt.alpha)

  local acc = get_result_accuracy(predicted_distributions, valid_blanks_input,
     valid_blanks_options, valid_blanks_outputs)
  printoptions(opt)
  print("Results:", acc, cross_entropy_loss, torch.exp(cross_entropy_loss))
elseif opt.classifier == 'mle' then
  local reverse_trie = fit(training_input, training_output)
  --print(get_word_counts_for_context(reverse_trie, torch.LongTensor{},
     nclasses, opt.alpha))
  local predicted_distributions = getmlepredictions(reverse_trie,
     valid_blanks_input, valid_blanks_options, nclasses, opt.alpha)
  --print(predicted_distributions:sum(2))
  local cross_entropy_loss = cross_entropy_loss(valid_blanks_outputs,
     predicted_distributions, valid_blanks_options)
  print("Cross-entropy loss", cross_entropy_loss)

  --result = predictall_and_subset(reverse_trie, test_context, test_options,
     nclasses, opt.alpha)

  local acc = get_result_accuracy(predicted_distributions, valid_blanks_input,
     valid_blanks_options, valid_blanks_outputs)
  printoptions(opt)
```

```
      print("Results:", acc, cross_entropy_loss, torch.exp(cross_entropy_loss))
    elseif opt.classifier == 'babbler' then
      local reverse_trie = fit(training_input, training_output)
      --print("Trained")

      local len_wanted = 1000
      local sentence = torch.LongTensor(len_wanted)
      -- Initialize the sentence.
      print("Train", training_input[80])
      sentence:narrow(1, 1, d_win):add(training_input[80]:squeeze())

      --local sentence_length = 5

      local context_size = d_win
      for sentence_length = 6, len_wanted do
        if (sentence_length % 10 == 0) then
          collectgarbage()
        end
        local dist = table_to_tensor(predict(reverse_trie, sentence:narrow(1,
            sentence_length-d_win, d_win), nclasses, opt.alpha), nclasses)
        --local _, ind = torch.max(dist, 1)
        ind = sampler(dist)
        if ind == 4 then
          ind = 3
        end
        print(ind)
        --print(dist)
        --sentence_length = sentence_length + 1
        sentence[sentence_length] = ind
      end
      --print(sentence)
    else
        print("Error: classifier '", opt.classifier, "' not implemented")
    end

    if (opt.testfile ~= '') then
        --print("Writing to test file")
        write_predictions(result, opt.testfile)
    end




end

main()
```

Listing 2: multinomial.lua: MLE and Smoothing Code

```
dofile(_G.path.."utils.lua")

function init_trie()
        return {}
end
```

```
function add_target_to_position(position, target)
        if position['counts'] == nil then
                local counts = {}
                counts[target] = 1
                position['counts'] = counts
        -- If we have gotten here, but the target is different
        elseif position['counts'][target] == nil then
                position['counts'][target] = 1
        -- If we have gotten here and seen this target before
        else
                position['counts'][target] = position['counts'][target]  + 1
        end
end

-- trie is the trie we are adding to (note that this is passed by reference)
-- sentence is a torch tensor consisting of the words
-- The target is optional - if included, then whole window used as context.
    Otherwise,
--      last element of context is the sentence
function add_word_and_context_to_trie(trie, context, target)
        local sentence_len = context:size(1)

        if target == nil then
                target = context[sentence_len]
                sentence_len = sentence_len - 1
        end

        -- The last word is one being predicted
        --local target = context[sentence_len]

        local position = trie

        add_target_to_position(position, target)
        -- iterate backwards through the context (since this is a reverse trie)
        for i = sentence_len, 1, -1 do
                local word = context[i]

                if position[word] ~= nil then
                        position = position[ word]
                else
                        position[word] = {}
                        position = position[word]
                end
                add_target_to_position(position, target)
        end

end

-- Given a trie and the context being looked for, returns a table
--       with the counts of each word that proceeded that context
--               Returns nil if context not in trie.
function get_word_counts_for_context(trie, context, vocab_size, alpha)

        -- Deal with the special case of the highest level position (unigrams - no
```

```
            previous words)
        if #context:size() == 0 then
                --return trie['counts']
                return add_to_tab(trie['counts'], vocab_size, alpha)
        end

        local context_len = context:size(1)
        --print(context_len)

        local position = trie
        -- iterate backwards through the sentence's context (this is a reverse trie
            )
        for i = context_len, 1, -1 do
                local word = context[i]

                if position[word] ~= nil then
                        position = position[word]
                else
                        --return {}
                        return add_to_tab({}, vocab_size, alpha)
                end

        end
        if position['counts'] then
                -- Explicitely add the count for the <s> string which will never be
                     predicted.
                --position['counts'][3] = 0
                --return position['counts']
                return add_to_tab(position['counts'], vocab_size, alpha)
        end
        --return {}
        return add_to_tab({}, vocab_size, alpha)
end


-- input_contexts: Torch LongTensor (N x d_win)
-- output_words: Torch LongTensor (N x 1)
-- Both of these should use the IDs of the
function fit(input_contexts, output_words)

        -- Make sure the inputs are valid
        assert(input_contexts:size(1) == output_words:size(1))
        local N = input_contexts:size(1)

        -- Load data into Trie
        local reverse_trie = init_trie()
        for i = 1, N do
                add_word_and_context_to_trie(reverse_trie, input_contexts[i],
                    output_words[i])
        end
        return reverse_trie
end

function normalize_table(tab)
        local total = sum_of_values_in_table(tab)
        return multiply_table_by_x(tab, 1/total)
```

```
end



function predict_multinomial_mle(trie, vocab_size)
        local count_table = get_word_counts_for_context(trie, torch.LongTensor{},
            vocab_size, 0)
        return normalize_table(count_table)
end

function predict_laplace(trie, context, vocab_size, alpha)
        local count_table = get_word_counts_for_context(trie, context, vocab_size,
            alpha)
        --count_table = add_to_tab(count_table, vocab_size, alpha)
        --count_table[3] = 0
        local F_cstar = sum_of_values_in_table(count_table)
        --local N_cstar = number_of_items_in_table(count_table, alpha)

        local normalized_table = multiply_table_by_x(count_table, 1.0/(F_cstar))
        return normalized_table
end


-- Returns the distribution over the vocabulary given the context
-- Trie should be a trained trie
-- Context is a LongTensor.
--
-- Note that this function operates recursively
function predict(trie, context, vocab_size, alpha)
        --print("Predict", vocab_size, alpha)
        local num_words = -1
        -- If there is no context, just return the base case
        if #context:size() == 0 then
                return normalize_table(get_word_counts_for_context(trie, context,
                    vocab_size, alpha))
        else
                num_words = context:size(1)
        end
        --local num_words = context:size(1)
        local count_table = get_word_counts_for_context(trie, context, vocab_size,
            alpha)
        --count_table = add_to_tab(count_table, vocab_size, alpha)
        --count_table[3] = 0
        local F_cstar = sum_of_values_in_table(count_table)
        local N_cstar = number_of_items_in_table(count_table, alpha)
        --print(F_cstar, N_cstar)
        assert(F_cstar > 0)

        -- This implements F_{c,w} + N_{C,star}*p_wb(w|c')
        local numerator = nil
        if num_words == 1 then -- base case, the unigram case is just the
            distribution of final words.
                --numerator = sum_tables(count_table, multiply_table_by_x(predict(
                    trie, torch.LongTensor{}),N_cstar))
                numerator = sum_tables(count_table, multiply_table_by_x(predict(
                    trie, torch.LongTensor{}, vocab_size, alpha), N_cstar))
```

```
        elseif num_words > 1 then
                numerator = sum_tables(count_table, multiply_table_by_x(predict(
                    trie, context:narrow(1, 2, num_words - 1), vocab_size, alpha),
                    N_cstar))
        end

        -- This implements the rest of the fraction
        local p_wb = multiply_table_by_x(numerator, 1.0/(F_cstar + N_cstar))
        return p_wb
end

function isnan(x) return x ~= x end

function predictall_and_subset(trie, valid_input, valid_options, vocab_size, alpha)
        assert(valid_input:size(1) == valid_options:size(1))
        print("Starting predictions")
        local predictions = torch.zeros(valid_input:size(1), valid_options:size(2))
        print("Initialized predictions tensor")
        for i = 1, valid_input:size(1) do
                if i % 100 == 0 then
                        --print("Iteration", i, "MemUsage", collectgarbage("count")
                            *1024)
                        collectgarbage()
                end
                local prediction = table_to_tensor(predict(trie, valid_input[i],
                    vocab_size, alpha), vocab_size)
                assert(prediction:sum() > .99999 and prediction:sum() < 1.000001)
                local values_wanted = prediction:index(1, valid_options[i])
                values_wanted:div(values_wanted:sum())
                predictions[i] = values_wanted
        end
        return torch.log(predictions)
end

function getlaplacepredictions(trie, valid_input, valid_options, vocab_size, alpha)
        assert(valid_input:size(1) == valid_options:size(1))
        print("Starting predictions")
        local predictions = torch.zeros(valid_input:size(1), valid_options:size(2))
        print("Initialized predictions tensor")
        for i = 1, valid_input:size(1) do
                if i % 100 == 0 then
                        --print("Iteration", i, "MemUsage", collectgarbage("count")
                            *1024)
                        collectgarbage()
                end
                local prediction = table_to_tensor(predict_laplace(trie,
                    valid_input[i], vocab_size, alpha), vocab_size)
                assert(prediction:sum() > .99999 and prediction:sum() < 1.000001)
                local values_wanted = prediction:index(1, valid_options[i])
                values_wanted:div(values_wanted:sum())
                predictions[i] = values_wanted
        end
        return torch.log(predictions)
end

function getmlepredictions(trie, valid_input, valid_options, vocab_size, alpha)
```

```
        assert(valid_input:size(1) == valid_options:size(1))
        print("Starting predictions")
        local predictions = torch.zeros(valid_input:size(1), valid_options:size(2))
        print("Initialized predictions tensor")
        for i = 1, valid_input:size(1) do
                if i % 100 == 0 then
                        --print("Iteration", i, "MemUsage", collectgarbage("count")
                            *1024)
                        collectgarbage()
                end
                local prediction = table_to_tensor(predict_multinomial_mle(trie,
                    vocab_size), vocab_size)
                assert(prediction:sum() > .99999 and prediction:sum() < 1.000001)
                local values_wanted = prediction:index(1, valid_options[i])
                values_wanted:div(values_wanted:sum())
                predictions[i] = values_wanted
        end
        return torch.log(predictions)
end

-- Creates a simple trie that demonstrates its main features.
function trie_example()
        local reverse_trie = init_trie()
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,2,3},1)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{2,2,3},2)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,1,3},2)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{2,1,3},2)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,1,3},1)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,1,3},1)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,1,2},1)
        add_word_and_context_to_trie(reverse_trie, torch.LongTensor{1,3},1)
        print(reverse_trie)
        local counts = get_word_counts_for_context(reverse_trie, torch.LongTensor
            {1,3})
        print(counts)
        print(normalize_table(counts))
end

-- Creates a trie with randomly generated words
--      num_sentences: the number of sentences to insert into the trie
--      length: The length of each sentence (this assumes fixed length)
--      vocab_size: The number of words in the vocabulary
function bigtrie_example(num_sentences, length, vocab_size)
        local reverse_trie = init_trie()
        for i = 1, num_sentences do
                if i % 10000 == 0 then
                        print(i)
                end
                add_word_and_context_to_trie(reverse_trie, torch.rand(length):mul(
                    vocab_size):long())
        end
    print("Getting counts")
    local try = 0
    local counts = nil
    print(get_word_counts_for_context(reverse_trie, torch.LongTensor{1}))
    print(get_word_counts_for_context(reverse_trie, torch.LongTensor{2}))
```

```
    print(get_word_counts_for_context(reverse_trie, torch.LongTensor{3}))
end



--trie_example()
--bigtrie_example(1000000,5,1000)
```

Listing 3: train.lua: Neural Network and NCE Training Code

```
dofile(_G.path.."utils.lua")
dofile(_G.path.."models.lua")
dofile(_G.path.."test.lua")

function trainModel(model,
                                        criterion,
                                        training_input,
                                        training_output,
                                        validation_input,
                                        validation_options,
                                        validation_true_out,
                                        minibatch_size,
                                        num_epochs,
                                        optimizer,
                                        save_losses)

        -- For loss plot.
        file = nil
        if save_losses ~= '' then
                file = io.open(save_losses, 'w')
                file:write("Epoch,Loss\n")
        end

        local parameters, gradParameters = model:getParameters()

        print("Got params and grads")
        print("Starting Validation accuracy", getaccuracy2(model, validation_input,
             validation_options, validation_true_out))

        for i = 1, num_epochs do
                print("L1 norm of params:", torch.abs(parameters):sum())
                for j = 1, training_input:size(1)-minibatch_size, minibatch_size do

                    -- zero out our gradients
                    gradParameters:zero()
                    model:zeroGradParameters()

                    -- get the minibatch
                    minibatch_inputs = training_input:narrow(1, j, minibatch_size)
                    minibatch_outputs = training_output:narrow(1, j, minibatch_size
                        )

                    -- Create a closure for optim
                    local feval = function(x)
                                -- Inspired by this torch demo: https://github.com/
```

```
						andresy/torch-demos/blob/master/train-a-digit-
						classifier/train-on-mnist.lua
					-- get new parameters
					if x ~= parameters then
						parameters:copy(x)
					end
					-- reset gradients
					gradParameters:zero()

					preds = model:forward(minibatch_inputs)
					loss = criterion:forward(preds, minibatch_outputs)
						--+ lambda*torch.norm(parameters,2)^2/2
					--print(loss)

					-- backprop
					dLdpreds = criterion:backward(preds,
						minibatch_outputs) -- gradients of loss wrt
						preds
					model:backward(minibatch_inputs, dLdpreds)

					if j == 1 then
						if save_losses ~= '' then
							file:write(i, ',', loss, '\n')
						else
							print("Loss: ", loss)
						end
					end

					return loss,gradParameters
				end

				-- Do the update operation.
			if optimizer == "adagrad" then
				config =  {
				learningRate = eta,
				weightDecay = lambda,
				learningRateDecay = 5e-7
			}
			optim.adagrad(feval, parameters, config)
			elseif optimizer == "sgd" then
				config = {
				learningRate = eta,
			}

			optim.sgd(feval, parameters, config)
			    else
				assert(false)
			    end


			end
			--print("Epoch "..i.." Validation accuracy:", getaccuracy(model,
			    validation_input, validation_options, validation_true_out))
			print("Epoch "..i.." Validation accuracy:", getaccuracy2(model,
			    validation_input, validation_options, validation_true_out))
		end
```

```
return model
end






function trainNCEModel(
                                        training_input,
                                        training_output,
                                        valid_blanks_input,
                                        valid_blanks_options,
                                        valid_blanks_output,
                                        minibatch_size,
                                        num_epochs,
                                        optimizer,
                                        save_losses,
                                        D_sparse_in,
                                        D_hidden,
                                        D_output,
                                        embedding_size,
                                        window_size,
                                        alpha,
                                        eta,
                                        sample_indices,
                                        K,
                                        p_ml_tensor,
                                        valid_input,
                                        valid_output)

        -- For loss plot.
        file = nil
        if save_losses ~= '' then
                file = io.open(save_losses, 'w')
                file:write("Epoch,Loss\n")
        end



        local model, embedding, lookup, bias = NCE(D_sparse_in, D_hidden, D_output,
             embedding_size, window_size)
        local modelparams, modelgradparams = model:getParameters()

        local lookupparams, lookupgrads = lookup:getParameters()
        local biasparams, biasgradparams = bias:getParameters()
        print(training_input:size())

        local k_index = 1
        for i = 1, num_epochs do
                -- renormalize embedding weights for regularization
                embedding.weight:renorm(embedding.weight, 2, 1, 1)

                print("Epoch", i, "L1 norm of model params:", torch.abs(modelparams
                    ):sum(), "LookupParams:", torch.abs(lookupparams):sum(), "
```

```
                    Biasparams:", torch.abs(biasparams):sum())
                print("Accuracy, CrossEntropy, Perplexity:", getNCEStats(model,
                    lookup, bias, valid_blanks_input, valid_blanks_options,
                    valid_blanks_output, p_ml_tensor))
                print(NCE_predictions2(model, lookup, bias, valid_input,
                    valid_output, D_hidden, D_output))
                for j = 1, training_input:size(1)-minibatch_size, minibatch_size do

                    -- get the minibatch
                    minibatch_inputs = training_input:narrow(1, j, minibatch_size)
                    minibatch_outputs = training_output:narrow(1, j, minibatch_size
                        )
                    sample_batch = sample_indices:narrow(1, k_index, minibatch_size
                        *K)
                    k_index = (k_index + minibatch_size*K) % (10000000 -
                        minibatch_size*K)

                    forwardandBackwardPass3(model, modelparams, modelgradparams,
                        lookup, lookupparams, lookupgrads, minibatch_inputs,
                        minibatch_outputs, sample_batch, p_ml_tensor, eta, bias,
                        biasparams, biasgradparams, K, file, save_losses)
                end
        end

        return model, lookup, bias, embedding
end
```

Listing 4: models.lua: Neural Network and NCE Model Code

```
require('nn')

dofile(_G.path.."test.lua")

--dofile('test.lua')
--Returns an initialized MLP1 and NLL loss
--D_sparse_in is the number of words
--D_hidden is the dim of the hidden layer (tanhs)
--D_output is the number of words in this case
--embedding_size is the dimension of the input embedding
--window_size is the length of the context
function neuralNetwork(D_sparse_in, D_hidden, D_output, embedding_size, window_size
    )
        print("Making neural network model")

        local model = nn.Sequential()
        local embedding = nn.LookupTable(D_sparse_in, embedding_size)
        model:add(embedding)
        model:add(nn.View(-1):setNumInputDims(2))
        model:add(nn.Linear(embedding_size*window_size, D_hidden))
        model:add(nn.HardTanh())
        model:add(nn.Linear(D_hidden, D_output))
        model:add(nn.LogSoftMax())
        criterion = nn.ClassNLLCriterion()

        return model, criterion, embedding
```

```
end


-- Create an NCE model
function NCE(D_sparse_in, D_hidden, D_output, embedding_size, window_size)
        print("Making NCE neural network model")

        local model = nn.Sequential()
        local embedding = nn.LookupTable(D_sparse_in, embedding_size)
        model:add(embedding)
        model:add(nn.View(-1):setNumInputDims(2))
        model:add(nn.Linear(embedding_size*window_size, D_hidden))
        model:add(nn.Tanh())
        -- we have z_w and z_{s_i,k} now


        return model, embedding, nn.LookupTable(D_sparse_in, D_hidden),
            nn.LookupTable(D_sparse_in, 1)

end


-- Returns the global cross-entropy of a standard NNLM.
function NNLM_CrossEntropy(model, to_predict_input, true_outputs)

        model:zeroGradParameters()
        local crit = nn.ClassNLLCriterion()
        crit.sizeAverage = false
        local total = 0
        for i = 1, to_predict_input:size(1) - 100, 100 do
                local preds = model:forward(to_predict_input:narrow(1, i, 100))
                local cross_entropy_loss = crit:forward(preds, true_outputs:narrow
                    (1, i, 100))
                total = total + cross_entropy_loss
        end
        return total/to_predict_input:size(1)
end

-- Returns the global cross-entropy of a NCE model.
function NCE_predictions2(model, lookuptable, bias, to_predict_input, true_outputs,
     D_hidden, D_output)

        --model:zeroGradParameters()
        --lookuptable:zeroGradParameters()
        --bias:zeroGradParameters()
        --print("NumValidationInputs", to_predict_input:size(1))

        local prediction_err = nn.Sequential()
        local linear_layer = nn.Linear(D_hidden, D_output)
        -- print(linear_layer.weight:size())
        -- print(lookuptable.weight:size())
        -- print(linear_layer.bias:size())
        -- print(bias.weight:squeeze():size())

        linear_layer.weight = lookuptable.weight
```

```
        linear_layer.bias = bias.weight:squeeze()
        prediction_err:add(linear_layer)
        prediction_err:add(nn.LogSoftMax())

        --print(linear_layer.weight - lookuptable.weight)
        --print(bias.weight)
        local crit = nn.ClassNLLCriterion()
        crit.sizeAverage = false
        local total = 0
        for i = 1, to_predict_input:size(1) - 100, 100 do
                local tanh_result = model:forward(to_predict_input:narrow(1, i,
                    100))
                local preds = prediction_err:forward(tanh_result)
                local cross_entropy_loss = crit:forward(preds, true_outputs:narrow
                    (1, i, 100))
                total = total + cross_entropy_loss
        end
        return total/to_predict_input:size(1)
end

-- For each validation input, returns a distribution over the options.
-- This can be used to calculate the cross-entropy over the options.
function NCE_predictions(model, lookuptable, bias, to_predict_input,
    to_predict_options, probs)

        model:zeroGradParameters()
        lookuptable:zeroGradParameters()
        bias:zeroGradParameters()

        local tanh_result = model:forward(to_predict_input)
        local minibatch_size = to_predict_input:size(1)
        local K = to_predict_options:size(2)

        -- Determine which rows to pick from lookuptable (each row of rows_wanted
            correspond to the indicies wanted for that minibatch)
        local rows_wanted = to_predict_options

        local lookuptable_rows = lookuptable:forward(rows_wanted)
        local bias_rows = bias:forward(rows_wanted)

        local z = torch.zeros(minibatch_size, K)
        for i = 1, minibatch_size do
                --print(bias_rows[i]:t())
                z[i] = torch.mm(tanh_result[i]:view(1, tanh_result:size(2)),
                    lookuptable_rows[i]:t()) + bias_rows[i]:t()
        end

        predictions = nn:LogSoftMax():forward(z)
        return predictions
end

function getNCEStats(model, lookup, bias, valid_input, valid_options,
    valid_true_outs, sample_probs)
    local predictions = NCE_predictions(model, lookup, bias, valid_input,
        valid_options, sample_probs)
    --print(predictions:sum(2))
```

```lua
    local cross_ent = cross_entropy_loss(valid_true_outs, predictions,
        valid_options)
    return get_result_accuracy(predictions, valid_input, valid_options,
        valid_true_outs), cross_ent, torch.exp(cross_ent)
end
-- model, lookup, bias = NCE(10, 2, 10, 2, 3)
-- modelparams, modelgradparams = model:getParameters()
-- biasparams, biasgrad = model:getParameters()
-- input_batch = torch.LongTensor{{7, 5, 2}, {5,4,9}, {1,8,6}}
-- output_batch = torch.LongTensor{6,1,4}
-- sample_indices = torch.LongTensor{2, 2, 2, 3, 3}
-- sample_probs = torch.Tensor{.15, .05, .23, 05, .001, .01, .2, .3, .0001, .00001}
-- lookupparams, lookupgrads = lookup:getParameters()
-- --print(modelparams)
-- forwardandBackwardPass3(model, modelparams, modelgradparams,lookup, lookupparams
    , lookupgrads, input_batch, output_batch, sample_indices, sample_probs, 1, bias,
     biasparams, biasgrad)
-- --print(modelparams)
-- NCE_predictions(model, lookup, bias, torch.LongTensor{{7, 5, 2}, {5,4,9}},
    torch.LongTensor{{1,2,3,4}, {1,2,3,4}} )

function nn_predictall_and_subset(model, valid_input, valid_options)
        assert(valid_input:size(1) == valid_options:size(1))
        print("Starting predictions")
        local output_predictions = torch.zeros(valid_input:size(1), valid_options:
            size(2))
        print("Initialized output predictions tensor")
        local predictions = torch.exp(model:forward(valid_input))

        for i = 1, valid_input:size(1) do
                --if i % 100 == 0 then
                --      print("Iteration", i, "MemUsage", collectgarbage("count")
                    *1024)
                --      collectgarbage()
                --end
                --print(valid_options[i])
                local values_wanted = predictions[i]:index(1, valid_options[i])
                --print(values_wanted)
                values_wanted:div(values_wanted:sum())
                output_predictions[i]:add(values_wanted)
                --print(output_predictions[i]:sum())
        end
        --print(output_predictions)
        return torch.log(output_predictions)
end
```

Listing 5: utils.lua: Various utils

```lua
function getaccuracy(model, validation_input, validation_options,
    validation_true_outs)
        local scores = model:forward(validation_input)
        local n = validation_input:size(1)
        local option_count = validation_options:size(2)

        local total_acc = 0.0
```

```
        for i=1, n do
                -- e.g. [2, 502, ..., ], where index is index of possible word
                local options = validation_options[i]
                local option_probs = {}
                local true_idx = validation_true_outs[i]

                for j=1, option_count do
                        local idx = options[j]
                        local s = scores[i][idx]
                        if option_probs[idx] ~= nil then
                                option_probs[idx] = option_probs[idx]+s
                        else
                                option_probs[idx] = s
                        end
                end

                option_probs = normalize_table(option_probs)

                local acc = option_probs[true_idx]
                total_acc = total_acc + acc
        end

        return total_acc/n
end

function sampler(dist)
  -- Do this to remove <unk> values.
  dist[2] = 0
  dist:div(dist:sum())

  --local _, ind = torch.max(dist, 1)
  --return ind:squeeze()

  local sample = torch.uniform()
  total = 0
  for i =1, dist:size(1) do
    total = total + dist[i]
    if total > sample then
      return i
    end
  end
  return dist:size(1)
end

function find(tensor_array, number)
        for i = 1, tensor_array:size(1) do
                --print(tensor_array[i])
                if tensor_array[i] == number then
                        return i
                end
        end
        return -1
end

-- Returns indicies into 1d tensor that match the number
```

```
function find_matching(tensor, num)
        return torch.linspace(1, tensor:size(1), tensor:size(1))[tensor:eq(num)]:
            long()
end

function printoptions(opt)
        print("Datafile:", opt.datafile, "Classifier:", opt.classifier, "Alpha:",
            opt.alpha, "Eta:", opt.eta, "Lambda:", opt.lambda, "Minibatch size:",
            opt.minibatch, "Num Epochs:", opt.epochs, "Optimizer:", opt.optimizer, "
            Hidden Layers:", opt.hiddenlayers, "Embedding size:", opt.embedding_size
            , "K:", opt.K)
end

function get_result_accuracy(result, validation_input, validation_options,
    validation_true_outs)
        --print("True outs", validation_true_outs:size())
        local n = validation_input:size(1)
        local option_count = validation_options:size(2)
        local total_acc = 0.0

        local a, c = torch.max(result, 2)
        c = c:squeeze()
        --print(c)
        --print(c:squeeze())


        for i=1, n do

                local true_idx = find(validation_options[i], validation_true_outs[i
                    ])
                assert(true_idx ~= -1)
                --print(c[i]:squeeze())

                if true_idx == c[i] then
                        total_acc = total_acc + 1
                end



                -- local options = validation_options[i]
                -- local option_probs = result[i]:add(-1*result[i]:min())/(result[i
                    ]:max()-result[i]:min())
                -- local true_idx = validation_true_outs[i]
                -- local acc_updated = false

                -- for j=1, option_count do
                --      if validation_options[i][j] == true_idx then
                --              acc = option_probs[j]
                --              acc_updated = true
                --              break
                --      end
                -- end

                -- assert(acc_updated)
                -- total_acc = total_acc + acc
        end
```

```
        return total_acc/n
end

function getaccuracy2(model, valid_input, valid_options, valid_true_outs)

        result = nn_predictall_and_subset(model, valid_input, valid_options)
        --print("Sum", result:sum())
        return get_result_accuracy(result, valid_input, valid_options,
            valid_true_outs), cross_entropy_loss(valid_true_outs, result,
            valid_options)

end

function scores_to_preds(scores)
        _, class_preds =  torch.max(scores, 2)
        --print(class_preds)
        local preds = class_preds:squeeze()
        binary_preds = torch.zeros(scores:size(1), scores:size(2))
        for i=1, scores:size(1) do
                binary_preds[i][preds[i]] = 1
        end
        return binary_preds
end

function get_predictions_from_model(model, test_input, test_options)
        local n = test_input:size(1)
        local option_count = test_options:size(2)
        local results = torch.zeros(n, option_count)

        local scores = model:forward(test_input)

        for i=1, n do
                local max_score = scores[i][1]
                local pred = 1

                for j=1, option_count do
                        local idx = test_options[i][j]
                        local s = scores[i][idx]
                        if s > max_score then
                                max_score = s
                                pred = j
                        end
                end

                results[i][pred] = 1
        end

        return results
end

function write_predictions(results, outfile)
        io.output(outfile)
        io.write("ID,Class1,Class2,Class3,Class4,Class5,Class6,Class7,Class8,Class9
            ,Class10,Class11,Class12,Class13,Class14,Class15,Class16,Class17,Class18
            ,Class19,Class20,Class21,Class22,Class23,Class24,Class25,Class26,Class27
```

```
            ,Class28,Class29,Class30,Class31,Class32,Class33,Class34,Class35,Class36
            ,Class37,Class38,Class39,Class40,Class41,Class42,Class43,Class44,Class45
            ,Class46,Class47,Class48,Class49,Class50\n")
        for test_i = 1, results:size(1) do
                io.write(test_i)
                for binary_i = 1, results:size(2) do
                        io.write(',', results[test_i][binary_i])
                end
                io.write('\n')
        end

end


-- Calculates cross-entropy loss. This is the sum of the log
-- probabilities that were predicted for the true class.
function cross_entropy_loss(true_outputs, log_predicted_distribution, options)
        --print(true_outputs)
        assert(true_outputs:size(1) == log_predicted_distribution:size(1))
        assert(true_outputs:size(1) == options:size(1))
        --local logged_probabilities = torch.log(predicted_distribution)
        --print(logged_probabilities[1])
        --print(true_outputs:size(1))
        local loss = 0.0
        for i = 1, true_outputs:size(1) do
                local matched_indicies = find_matching(options[i], true_outputs[i])
                --print(matched_indicies)
                assert(matched_indicies:size(1) > 0)
                local cross_loss = log_predicted_distribution[i]:index(1,
                    matched_indicies)[1]
                loss = loss - cross_loss
                --print(log_predicted_distribution[i])
                --print(cross_loss)
                --print(predicted_distribution[i]:index(1, matched_indicies)[1])
                --print(loss)
                --if loss > 100 then
                --      assert(false)
                --end
                --loss = loss + torch.log(predicted_distribution[i]:index(1,
                    matched_indicies):sum())

                --loss = loss + logged_probabilities[i]:index(1, matched_indicies):
                    sum()
                --local predicted_distribution_index = find(options[i],
                    true_outputs[i])

                --print(i, true_outputs[i], predicted_distribution_index, options[i
                    ])
                --assert(predicted_distribution_index ~= -1)
                --loss = loss + logged_probabilities[i][
                    predicted_distribution_index]
        end
        --print(loss)
        return loss/true_outputs:size(1)
end
```

```lua
-- This function might not actually have any value, but
--      it basically just normalizes the counts in a table.
function normalize_table(tab)
        local total = sum_of_values_in_table(tab)
        return multiply_table_by_x(tab, 1/total)
end

-- Calculates the number of items in a table
-- This can be used to calculate N_{c,*}
function number_of_items_in_table(tab, min_value)
        return #tab
end

-- Sums of the values in a table
-- This can be used to calculate F_{c,*}
function sum_of_values_in_table(tab)
        local total = 0
        for _, val in pairs(tab) do
                total = total + val
        end
        return total
end

-- For some reason the Torch constructor didnt work.
function table_to_tensor(tab, size)
        local t = torch.zeros(size)
        for k,v in pairs(tab) do
                t[k] = v
        end
        return t
end

-- Adds 'amount' to each key of tab from 1 to max_possible
-- This is used for laplace smoothing
function add_to_tab(tab, max_possible, amount)
        local new_tab = {}
        --print(max_possible)
        for i = 1, max_possible do
                if tab[i] ~= nil then
                        new_tab[i] = tab[i] + amount
                else
                        new_tab[i] = amount
                end
        end
        return new_tab
end



-- Multiply each value in a table by x
function multiply_table_by_x(tab, x)
        local new_table = {}
        for key, val in pairs(tab) do
                new_table[key] = val*x
```

```lua
        end
        return new_table
end

-- Takes the elementwise sum of two tables
-- This is essentially an outer join, where we sum values on duplicate keys.
function sum_tables(tab1, tab2)
        new_table = {}
        for key,val in pairs(tab1) do
                new_table[key] = val
        end
        for key, val in pairs(tab2) do
                if new_table[key] ~= nil then
                        new_table[key] = new_table[key] + val
                else
                        new_table[key] = val
                end
        end
        return new_table
end
```