Fast Softmax Sampling for Deep Neural Networks

Ifueko Igbinedion
CS231N
ifueko@stanford.edu

Derek Phillips
CS231N
djp42@stanford.edu

Daniel Lévy
Ermon Group, Stanford University
danilevy@cs.stanford.edu

Abstract

The softmax function has been widely popularized due to its frequent use in neural networks. It features a linear computation time that can dominate training time if the output space is very large. This project explores the use of k-means clustering and uniform sampling to approximate the softmax function and achieve $O(\sqrt{N})$ run time. We additionally present experiments varying the number of clusters and update frequency of the data structure, showing that the softmax approximation has potential to provide a significant reduction to complexity.

1. Introduction

Many neural networks use a softmax function in the conversion from the final layer's output to class scores. The softmax function takes an N dimensional vector of scores and pushes the values into the range $[0,1]$ as defined by the function:

$$f_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

The denominator of the softmax function is dependent on every element in the scores vector, and thus the time complexity of using the softmax function is $O(N)$. When using datasets with very large output spaces, this can quickly become a computational bottleneck during training time [11]. This is a result of the last matrix multiplication in these models, between the hidden states of size B x d (B is the batch size, and d is the dimension for the hidden state) and the output weights, which are size d x K (K is the number of classes, or output space). With large output spaces, this quickly dwarfs the complexity of the other operations in the model.

Our project implements and analyzes a theoretical algorithm, developed by the Ermon Group at Stanford, for reducing the complexity of the softmax function to $O(\sqrt{N})$ without significantly compromising on accuracy [16]. This approach uses Gumbel random variables, a Maximum Inner Product Search (MIPS) data structure and random sampling to estimate an $O(\sqrt{N})$ sized subset of the dataset for use in estimating the output of the softmax function. With the K-means data structure used for MIPS, we estimate the softmax function for each training example using $O(\sqrt{N})$ classes that are considered its nearest neighbors as well as a weighted random sampling of other classes. Achieving an $O(\sqrt{N})$ speedup of the softmax computation in a deep network with a large number of output classes will be very impactful as it would decrease the training time of various networks from weeks to days. In this paper, we first discuss some related approaches to speeding up computation for problems with large output spaces. We then discuss in detail our method for approximating the forward and backward pass of softmax computation. We then highlight some experiments conducted to assess the success of our fast softmax implementation and their results. Lastly, we discuss some challenges we faced during implementation and conclude with some final thoughts on fast softmax approximation.

2. Background and Related Work

A popular class of image classification problems are those that deal with natural language and consequently very large output spaces. For example, recurrent neural networks for image caption generation feature output spaces that can enter the millions [18]. Many log linear prediction models normalize class probabilities using every element in the scores vector. This can be a computational bottleneck, leading to training times that can range from weeks to months, especially for datasets with both large output spaces and large numbers of training examples [10].

The inefficiency of linear functions for problems with large output spaces is well known. Many researchers have attempted to provide an effective method for speeding up computation of functions like the softmax. Perhaps the most well known such method is the Hierarchical Softmax [5], which uses a tree structure to represent the probabilities of words, with nodes representing latent variables that can also be interpreted as classes. A common variant of the Hierarchical Softmax is called the two-level tree, which uses $O(\sqrt{N})$ classes as the intermediate level of the tree, with the words as the leaves [5, 13], but deeper trees have also been explored [15]. Hierarchical softmax is fast during training, but can be more expensive to compute during testing than the normal softmax [4]. However, it is nonetheless a popular choice, and a number of variations have been published in recent years [19]

There are other tree-like approaches, such as that used in Koenigstein *et. al.*, which uses branched and bounded binary spatial-partitioning metric tree based clusterings to limit number of predictions made, by only 'recommending' for the center of a cluster [12]. However, this was only done for the very specific task of making recommendations for users. Another method implements a hierarchical bilinear language model that utilizes decision trees to estimate probability distributions upon words to automatically construct word trees from the model [14].

A simple approach is simply to perform the normalization calculation less frequently, saving on computation but decreasing performance accuracy [2] Another approach is called the Differentiated Softmax, and it assigns each output class a capacity based on the output class's frequency [4].

There are also a number of sampling based approaches [11], including targeted sampling [7], which is similar to our method in goal. It selects a number of *imposters* to sample from the total output space, but does so by simply using the classes with positive examples in a batch, and sampling from the rest.

Another approach to accelerate neural network training is adaptive importance sampling. Adaptive importance sampling aims to train a model from which learning and sampling are fast [3]. During backpropagation, the likelihood of only a few negative class examples are decreased. The samples are chosen using an n-gram based model that models conditional probabilities of the language model.

Another algorithm called *BlackOut* applies importance sampling to RNN language models with outputs spaces on the order of millions [8]. The approximation algorithm uses a weighted sampling strategy to significantly reduce computation time for softmax based layers in RNN language models.

Other approaches use clustering to eliminate the linear complexity of the softmax, often forming these clusters based on features such as frequency, which are related to the naturally uneven distribution of words in language [6]. This method, called adaptive softmax, uses probability distributions to estimate the most likely cluster from a set of clusters, estimating the softmax function using data from the most likely cluster.

Work very similar to what we implement in this project was done by Vijayanarasimhan *et. al.*, and used a locality-sensitive hashing technique to approximate the actual dot product, which could then be used to approximated the softmax layer [18]. This is very similar to the algorithm we use, except we also introduce a sampling of the tail end of the distribution [16]. Despite this lack of sampling, they were able to obtain good results, with a speed-up of 10x while maintaining 90% of accuracy of their baseline [18].

The algorithm we implemented in this project is inspired by the work in the Ermon Group at Stanford related to fast learning in log-linear models. [16]. In this work, Gumbel random variables are utilized in order to sample likely states. Additionally, maximum inner product search is utilized to select nearest neighbor states relevant to the input vector. The randomly sampled and relevant states are then used together to estimate the log linear function. The work provides theoretical guarantees of gradient convergence to the global maximum almost $10\times$ faster than directly computing the gradient [16]. Our method also uses random sampling and nearest neighbor search to estimate a log linear for a convolutional neural network.

3. Method

3.1. Theory

Our method redefines the forward and backwards passes of the softmax layer for any neural network that computes class scores using a linear layer. The forward pass of the fast softmax layer computes the softmax function in four steps: k-means clustering, determining top K nearest neighbors, random sampling, and approximation. First, a k-means data structure is (potentially) updated, clustering the d dimensional weight vectors from of the last linear layer. The weights are a good choice for generating clusters because they have the same dimensionality as the input vector, and they will have higher magnitudes for elements of the input vector most related to that class. Choosing classes whose weights are a most similar to the input increases the probability that the correct class is within the set of nearest neighbor classes. As a result input for the last linear layer is used to query the k-means data structure for the top $K \approx O(\sqrt{N})$ classes. Random sampling is then used to select $L \approx O(\sqrt{N})$ random classes. Finally, the softmax function is approximated according to the formula,

$$\bar{f}_j = \frac{e^{\phi(x)^\top w_j}}{\sum_{i \in S} e^{\phi(x)^\top w_i} + \frac{n-|S|}{|T|} \sum_{i \in T} e^{\phi(x)^\top w_i}}$$

where $\phi(x)$ is the input to the last linear layer for a particular training example, S is the set of K nearest neighbors of $\phi(x)$, and T is the set of L randomly sampled classes. The slow softmax loss is given by the equation

$$L(x, y) = -\phi(x)^T w_{y^*} + \log \sum_j \exp(\phi(x)^T w_j)$$

The fast softmax loss can thus be calculated as

$$L(x,y) = -\phi(x)^T w_{y^*} + \log \hat{Z}$$

Where $\hat{Z}$ is

$$\sum_{j\in S} \exp(\phi(x)^T w_j) + \frac{n-|S|}{|T|} \sum_{j\in S} \exp(\phi(x)^T w_j)$$

The backwards pass for fast softmax estimates the gradients with respect to the inputs and weights for the last linear layer. The gradient of the loss with respect to the input vector is

$$\nabla_{\phi(x)} L(x,y^*) = -w_{y^*} + \frac{\sum_{i\in S} e^{y_i} + \frac{n-|S|}{|T|}\sum_{i\in T} e^{y_i}}{\sum_{i\in S} e^{y_i} f_i + \frac{n-|S|}{|T|}\sum_{i\in T} e^{y_i} f_i}$$

where y^* is the correct class for this training example and $y_i = \phi(x)^\top w_i$. The gradient of the loss with respect to the weights of the last linear layer is

$$\nabla_{w_i} L = -\phi(x)\mathbf{1}_{i=y^*} + \mathbf{1}_{i\in S\cup T} \frac{\exp(\phi(x)^T w_i)}{\hat{Z}}$$

3.2. Implementation

For the k-means implementation, we utilized Facebook's implementation of FAISS, a tool for fast similarity search and clustering of dense vectors. FAISS utilizes the GPU to allow for fast training and querying of k-means indexes [9]. Our network was implemented using PyTorch. In order to use our fast softmax approximation, we needed to define a new autograd function based on the forward and backwards passes described above. The main network we trained our model was a simple convolutional neural network. The first layer was a convolutional layer with 32 7×7 filters and a stride of 2. This was followed by a ReLU nonlinearity layer [1], which ensures all values are non negative according to the function $f(x) = \max(0,x)$. This was followed by a spatial batch normalization layer which normalizes the mean and variance along the second dimension (within each RGB channel) [1]. We then apply a 2×2 max pooling layer [1] with a stride of 2 to choose the largest value within each 2×2 grid in the data. We then apply a 6272 by 2048 linear layer [1] followed by a 2048 by C linear layer without a bias term, where C is the number of output classes, before the fast softmax. Our implementation combines the last linear layer and the softmax layer, computing the forward and backward pass and calculating gradients with respect to the both the input and weights. The pseudocode for the forward pass is in Algorithm 1, while the pseudocode for the backwards pass is in Algorithm 2. These algorithms are defined for a single training example, but the algorithm is easily extensible to batch sizes greater than 1.

```
Input: output of last linear layer
Output: fast softmax loss
$w \leftarrow$ weights of last linear layer;
$knn \leftarrow$ FAISS K-Means Data Structure;
$t \leftarrow$ current iteration of training;
$p \leftarrow$ hyper parameter: update frequency;
$K \leftarrow$ hyper parameter: number of desired nearest
  neighbors, $O(\sqrt{N})$;
$L \leftarrow$ hyper parameter: number of randomly sampled
  classes, $O(\sqrt{N})$;
if $t\%p = 0$ then
    Train K-Means Data Structure on current weights
end
$n \leftarrow$ total number of classes;
$S \leftarrow$ top K nearest neighbors of input from K-Means
  data structure;
$T \leftarrow$ random sampling of $L$ classes;
$\hat{Z} = \sum_{j\in S} e^{(\phi(x)^T w_j)} + \frac{n-|S|}{|T|}\sum_{j\in S} e^{(\phi(x)^T w_j)}$;
$L = -\phi(x)^T w_{y^*} + \log \hat{Z}$;
return $L$;
```

Algorithm 1: Fast Softmax Sampling Forward Pass

```
Input: Loss from forward pass
Output: Gradients with respect to $\phi(x)$ and $w$
$w \leftarrow$ weights of last linear layer;
$\phi(x) \leftarrow$ previous input vector;
$y^* \leftarrow$ correct output class for this example.;
$S \leftarrow$ top $K$ nearest neighbors calculated at the
  forward pass;
$T \leftarrow$ $L$ randomly sampled classes calculated at the
  forward pass;
$n \leftarrow$ total number of classes;
$dX = -w_{y^*} + \frac{\sum_{i\in S} e^{y_i} + \frac{n-|S|}{|T|}\sum_{i\in T} e^{\phi(x)^\top w_i}}{\sum_{i\in S} e^{\phi(x)^\top w_i} f_i + \frac{n-|S|}{|T|}\sum_{i\in T} e^{\phi(x)^\top w_i} f_i}$;
$dW = -\phi(x)\mathbf{1}_{i=y^*} + \mathbf{1}_{i\in S\cup T}\frac{\exp(\phi(x)^T w_i)}{\hat{Z}}$;
return dX, dW;
```

Algorithm 2: Fast Softmax Sampling Backward Pass

4. Data

We utilized the Flickr100M dataset [17], which has 100 million images and a vast output space of all of the tags for the images. The dataset has been used in previous work, such as that done by Joulin, Maaten *et al.* [10]. We subsampled the dataset to allow for us to run our experiments in the allotted time for the project, but with more time we would have tested with different output sizes starting from 500 and increasing to 100,000 distinct classes. For our tests, we began with a few tests on 500 output classes, but the majority of our work is done with 5000 output classes.

Because of the vast output space of the Flickr100M dataset, we subsampled the dataset in the following manner. First, we go through each training example in the dataset, and with a 1% probability we add each data example to a dictionary whose keys are single words used to tag the image. After iterating through the entire dataset, we sort the output class labels by the number of training examples it has and ignore the top 250 most common words (for example, ”the”, ”of”, ”with”) in a similar manner as [10]. From the remaining data we take the top $C \in [500, 5000]$ most common output classes and select an equal number of training and validation examples per class. Additionally, we limit the resolution to be 64×64 and scale the images into the range 0 to 1 and normalize, as is common in image processing scenarios.

5. Experiments

The main evaluation metric for this work was the speed vs. performance trade off. There are a number of potential relationships specific to our proposed algorithm that we could evaluate, including the update frequency of the MIPS KNN data structure, output space size, and a variety of hyper-parameters for the FAISS k-means implementation. Due to time constraints encountered, we focused on what we thought would be the most interesting relationship, the update frequency. This parameter has not been not tested previously, and there is no theoretical effect described for the update frequency's effect on performance. Obviously, updating every iteration would produce results most similar to that of the baseline, but that is the extent of the understanding for how it will affect performance. For other parameters, such as the output space size, there is a theoretical understanding of how they affect the performance [16]. Thus, we determined the update frequency to be most interesting and selected it to test on given the time constraints.

For this work, we test with update frequencies of every {1,5,10} mini-batch iterations. We perform experiments using the same training and validation sets (as described in section 4) for all models, ensuring comparable results. We additionally perform some evaluation over the size of the output space, testing with both 500 and 5000 output classes. Both output spaces are not as large as we would have liked, a constraint encountered as a result of limited resources and time. As a result, we examine the performance of the fast k-means softmax approximation with both $\sqrt{C}$ and $10\sqrt{C}$ classes selected as the ‘Top-K’ for the forward pass. The theoretical background of the work suggests that the results with $\sqrt{C}$ classes will not be as reliable regarding accuracy as the proven $10\sqrt{C}$ [16]. However, we evaluate that performance as a further, and more prominent, demonstration of the potential time savings. Additionally, there is an overhead to using the k-means MIPS data structure, which is another aspect that we are evaluating.

Throughout the evaluation of the various models, we keep track of the training loss, the training and validation accuracy, the time spent in the forward pass of the model, and the overall time spent evaluating the model. We define the forward pass to include the update to MIPS KNN data structure, as we expect that update to take a non-insignificant amount of time. We use standard time to perform the the efficiency comparison of the models because we are using a dedicated server - the google compute engine - that should have limited fluctuations in other tasks that divert processing power away from our process. Furthermore, the relatively minor fluctuations will not affect the overall trends, which are what we are evaluating.

The hyper-parameters we used were as follows:

1. Batch size: $300, 150$
2. Learning rate: 0.001
3. Number of epochs: 1
4. FAISS Specific:

   nlist = 1

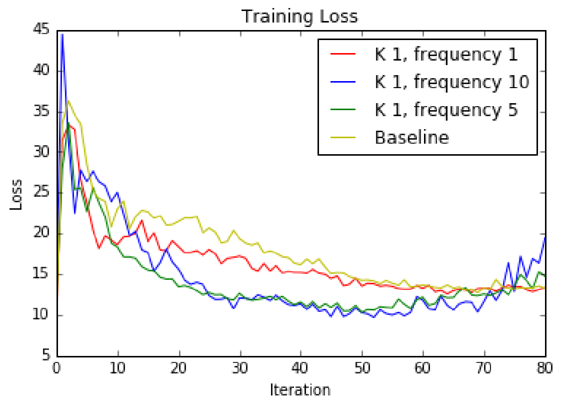
   nprobe = 1

We chose a batch size of 300 for the 5000 output class example because of constraints on the training time available. However, we could not use a bigger batch size because of the memory constraints of the machine we were working on. For the 500 output class example, we used a batch size of 150 to balance the number of iterations between overall speed and enough iterations to illustrate the trends we are looking for.

The learning rate of 0.001 was chosen based on samples tested on the 500 output class example. With a learning rate of 0.01 we saw very little learning, and an accuracy that hovered around random performance, but with a slightly lower learning rate the model was able to optimize better. However, we did not extensively test this because the purpose of our project centers more around the relative performance of the models, so as long as they all use the same learning rate we should see similar results. Nonetheless, testing different learning rates may be an interesting element to study in future work.

We only tested over 1 epoch as a direct result of the time constraints encountered. However, the trends we see over this epoch would likely continue, but it would have been preferred to have tested until convergence for a number of models, which we will discuss more in section 6

The *nlist* and *nprobe* variables are FAISS specific hyper-parameters that determine the “number of cells” and the number of those cells that are visited to perform a search, respectively. We selected an *nlist* of 1 because it is the most straightforward option, and will give a lower bound on the search efficiency, i.e. it will be the slowest because using an



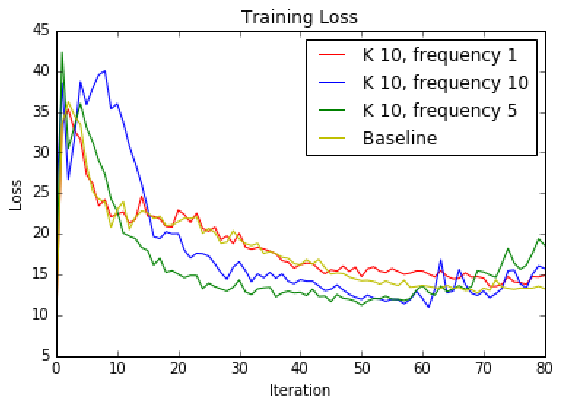


Figure 1. The loss learning curves for a number of models with 5000 output classes over 1 epoch.

nlist of 1 is equivalent to a naive search [9]. Varying these hyper-parameters are another likely next step for this work.

6. Results and Discussion

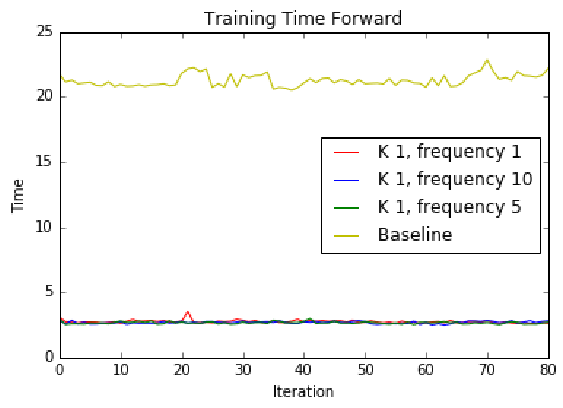
The results presented here are based on the experiments we ran as described in section 5. To summarize, we vary the update frequency of the MIPS KNN data structure in $\{1,5,10\}$, and $K \in \{1, 10\}$, where the number of classes selected for the Fast Softmax computation is $K\sqrt{C}$, with $5K\sqrt{C}$ uniformly sampled classes. The baseline uses the same implementation of the Fast Softmax layer we defined, but it never updates the MIPS KNN data structure and never performs a search in the data structure, instead always selecting every C class (and thus there is no uniform sampling done for the baseline).

6.1. Loss

As we can see in Figure 1, the models all have a relatively normal learning curve, the with loss decreasing over the course of an epoch. There are two interesting observations from this graph. First, the Baseline model has the highest lost, but it is also decreasing slowly yet consistently. This suggests that it may converge less quickly, but when it does it will converge to a lower value. It is also worth noting that the behavior is relatively consistent between the K values tested. This could be a result of the fact that the FastSoftmax models update only a small subset of weights each iteration. However, the model that updates its MIPS KNN data structure every iteration does not experience such a drastically lower loss, suggesting that the behavior is related to less frequent updates.

The second observation is that the more frequently updated FastSoftmax models see an increase in loss towards the end of the epoch. This likely signifies an overfitting of the models. A cause for this could be that the less frequently updated data structure gives the model a large number of iterations to learn using the same weights, thereby overfitting. This is a particularly interesting result, as it was not one of the expected results we talked about in the milestone.

6.2. Time



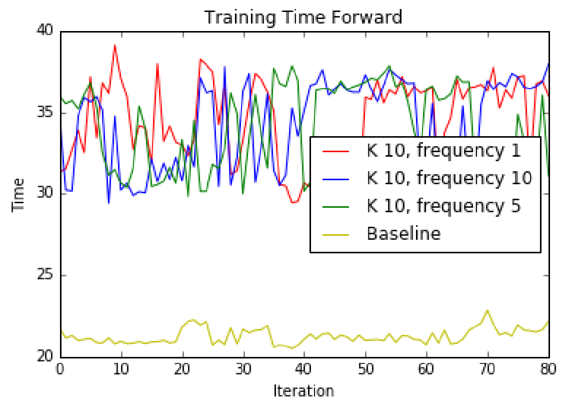


Figure 2. The time spent in the forward pass (and updating MIPS KNN datastructure) for the baseline and the KNN based fast softmax approximation per iteration with 5000 output classes over 1 epoch.

We claim that the time spent in the forward pass of the model is linearly proportional to the output space, which is confirmed through Figure 2. The figure clearly shows that using a small number of clusters creates extremely large savings for the forward pass. With K=1, we would have $\sqrt{5000}$, or 70 classes selected for the *top-K*, with $5\sqrt{5000}$, or 353, randomly sampled classes. Therefore, we expected to see the absolute time to compute for the forward pass drop by about 90%, which it does do. An interesting observation from these results is how consistent the time is, showing that the updates to the MIPS data structure are extremely quick. With $K = 10$, we see $10\sqrt{5000}$, or about 700 clases selected for the *top-K*, with $50\sqrt{5000}$, or about 3500 uniformly sampled classes also considered. While this amounts to fewer classes for the Softmax to sum over, the overhead present from the FAISS data structure and searching makes the overall time spent in the forward pass much more than the time spent in the baseline forward pass.

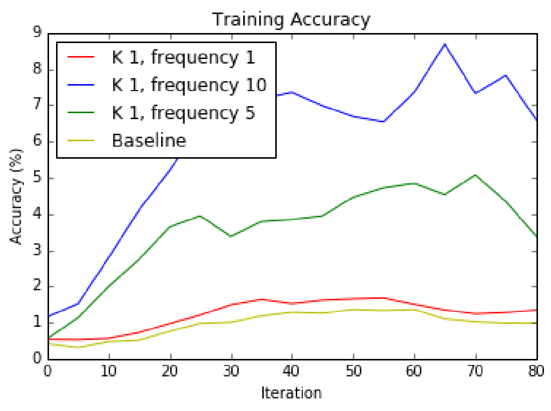
Table 1. The total time to run one epoch, not just forward passes, for the models tested.

	Baseline	Freq: 1	Freq: 5	Freq: 10
$K = 1$	3:35	3:10	3:02	2:34
$K = 10$	3:35	4:38	4:33	4:31

Table 1 Shows that the overall time taken to train the models is not only based on the time spent in the forward pass. There are many other components of training, such as the backwards pass and the time spent checking accuracy (most of it). Checking accuracy took so long because it required doing a forward pass for each minibatch of data (because we could not store the full dataset in memory), each pass of which took almost as much time as the forward pass of the iteration (no updating the data structure).

6.3. Accuracy

Although not the main focus of the project, we see interesting results in both the training and the validation accuracy of the models, as shown in Figure 3 and Figure 4. Beginning with the training accuracy, we can see that the lower update frequency again outperforms the higher update frequencies and the baseline, a counter-intuitive result. However, we can follow the same explanation as for the loss behavior, and we can see that the lowest update frequencies appear to have hit a maximum accuracy and actually decrease after iteration 70, possibly indicating overfitting. Seeing the behavior over a number of more epochs would be able to confirm or refute the trend. For the validation accuracy, we actually encounter some troubling behavior, as the baseline and the frequent update model reach 0 accuracy. That would seem to suggest overfitting on the training set, but neither the loss nor the training accuracy corroborate that interpretation.



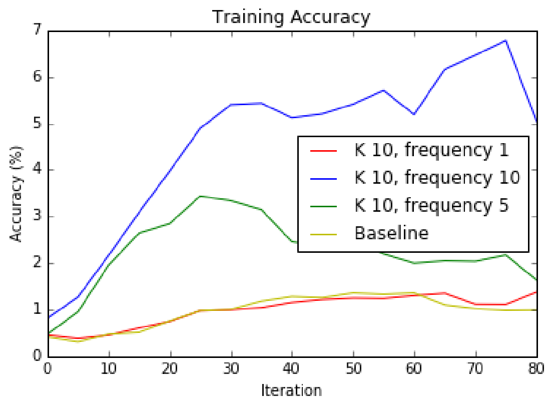
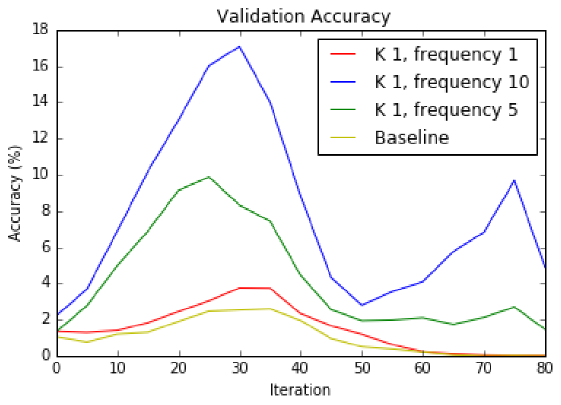


Figure 3. The training accuracy for a number of models with 5000 output classes over 1 epoch.

6.4. Challenges

Some implementational challenges limit the effectiveness of our application of the theory. The nature of the FAISS data structure required the use of numpy arrays, adding necessary conversions between CPU and GPU based data structures in the PyTorch framework. PyTorch also introduced challenges to our implementation. While PyTorch supports indexing of all rows in a matrix by a single set of indices, it does not support the indexing of each row by a unique set of indices. This limited the magnitude of batch sizes we could support, as indexing for each training example had to be conducted individually. Another challenge was that the tradeoff between maintaining an expensive MIPS KNN data structure and improving speed was only beneficial for very large output spaces, in which the time spent maintaining the data structure was negligible compared to the time required to compute the softmax function. Lastly, neural networks with large vocabularies are extremely expensive to train. For just 500 output classes, it takes about 30 minutes to run 1 epoch with our setup, and this time increased linearly as we tested 2500, 5000, and 10,000 output classes.



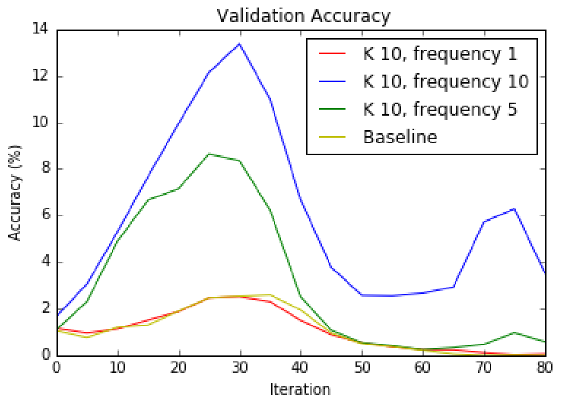


Figure 4. The validation accuracy for a number of models with 5000 output classes over 1 epoch.

7. Conclusion

The experiments described in this paper illustrate the potential of the K-Nearest Neighbors based Fast Softmax Sampling Approximation technique to reduce the computational complexity of the softmax layer [16]. However, we also encountered unexpected relationships between the update frequency of the data structure used to evaluate the Maximum Inner Product. This same data structure is what we used to find the $top-K$ classes for an input hidden state vector to the softmax layer we defined. These K classes were supplemented with uniformly sampled other classes to produce a theoretically guaranteed output [16]. We found that less frequent updates of the MIPS KNN data structure led to better performance over 1 epoch, but this could easily be an anomaly in an overall trend should the training duration be extended. Finally, we see that there is a noticeable overhead for performing the nearest-neighbors search, despite the fact that it is performed on a GPU.

Taking this work further could prove very fruitful. Doing so would allow for longer training durations to see longer-term trends, and it would allow for the testing of more hyper parameters, such as the FAISS specific *nlist* and *nprobe*, which could have significant benefits for the efficiency of the K-Nearest Neighbors search.

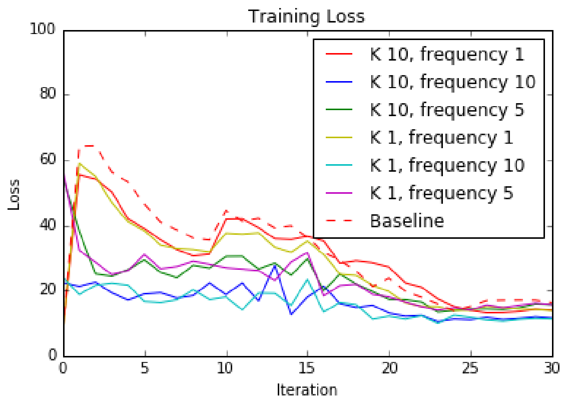
8. Author Contributions

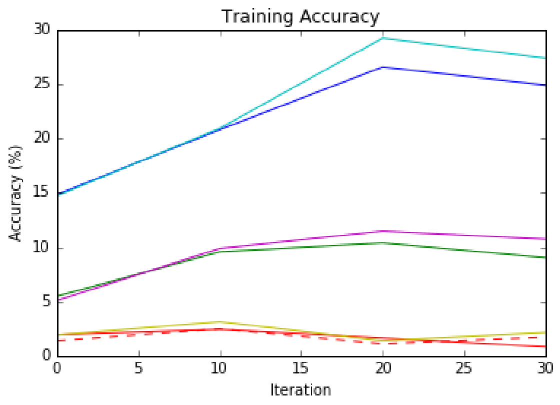
Ifueko Igbinedion and Derek Phillips contributed equally in the design and implementation of the project framework. Daniel Lévy, a Ph.D. student in the Ermon Lab (and not in CS 231N) provided mentorship, taught the other authors the theory needed to implement this project, and suggested a FAISS based implementation.

9. Appendix

We include in the appendix the results for the tests we ran with 500 output classes and a batch size of 150. We do not include these in the body of our paper because they do not contribute to the claims we are making, nor do they refute them. They do, however, show the same trends, with performance increasing for less frequent updates. The legend in the plot for training loss applies to all four figures. It is noticeable that only 30 iterations were performed. That is due to the batch size of 150 and the fact that we maintain a constant number of examples per class, between these runs and the runs with 5000 output classes.

We originally had an outlier in Table 1, of 12:58, for testing with Frequency 1 and $K = 10$. For some reason, the google VM experienced no CPU activity for overnight when that test was running. It died at around 1:30 AM, when we were asleep, but started back up around 11. This led to imprecision in the time we measured, but taking out the dead period of the CPU (it was at 0% CPU for utilization for 8 hours and 20 minutes, more or less), as seem in Figure 7. Subtracting this dead period gave us a time that was approximately as expected.





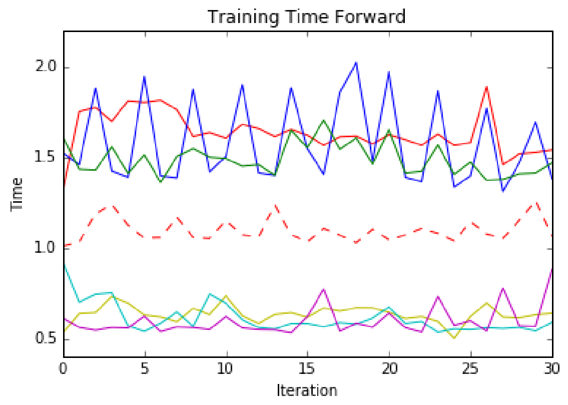


Figure 5. Appendix Figure 1: The Loss and Forward Pass Time for a number of models with 500 output classes over 1 epoch.

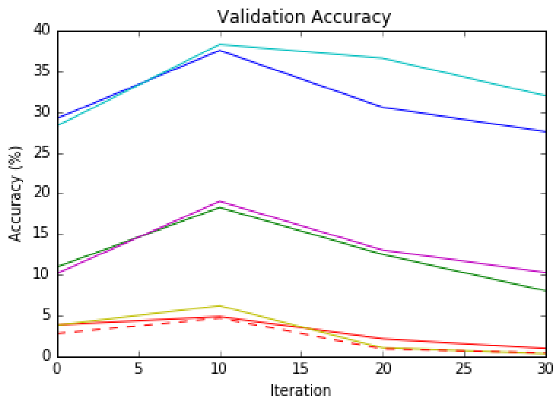


Figure 6. Appendix Figure 2: The validation accuracy for a number of models with 500 output classes over 1 epoch.

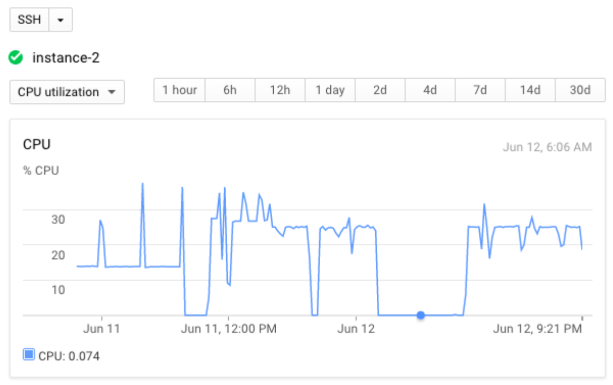


Figure 7. The Google VM randomly died overnight while we were asleep.

References

[1] Cs231n course notes. `http://cs231n.github.io/`. Accessed: 2017-06-06. 3

[2] J. Andreas and D. Klein. When and why are log-linear models self-normalizing? In *HLT-NAACL*, 2015. 2

[3] Y. Bengio and J.-S. Sencal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008. 2

[4] W. Chen, D. Grangier, and M. Auli. Strategies for training large vocabulary neural language models. *CoRR*, abs/1512.04906, 2015. 2

[5] J. Goodman. Classes for fast maximum entropy training. *CoRR*, cs.CL/0108006, 2001. 1, 2

[6] E. Grave, A. Joulin, M. Cissé, D. Grangier, and H. Jégou. Efficient softmax approximation for gpus. *CoRR*, abs/1609.04309, 2016. 2

[7] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. *CoRR*, abs/1412.2007, 2014. 2

[8] S. Ji, S. V. N. Vishwanathan, N. Satish, M. J. Anderson, and P. Dubey. Blackout: Speeding up recurrent neural network language models with very large vocabularies. *ICLR 2016*, 2015. 2

[9] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *CoRR*, abs/1702.08734, 2017. 3, 5

[10] A. Joulin, L. van der Maaten, A. Jabri, and N. Vasilache. Learning visual features from large weakly supervised data. *CoRR*, abs/1511.02251, 2015. 1, 3, 4

[11] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016. 1, 2

[12] N. Koenigstein, P. Ram, and Y. Shavitt. Efficient retrieval of recommendations in a matrix factorization framework. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 535–544, New York, NY, USA, 2012. ACM. 2

[13] T. Mikolov, S. Kombrink, L. Burget, J. ernock, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, May 2011. 2

[14] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. *Advances in neural information processing systems*, pages 1081–1088, 2009. 2

[15] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. In *AISTATS05*, pages 246–252, 2005. 2

[16] S. Mussmann, D. Levy, and S. Ermon. Fast amortized inference and learning in log-linear models with randomly perturbed nearest neighbor search. Unpublished. 1, 2, 4, 7

[17] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 59(2), 2016. 3

[18] S. Vijayanarasimhan, J. Shlens, R. Monga, and J. Yagnik. Deep networks with large output spaces. *CoRR*, abs/1412.7479, 2014. 1, 2

[19] G. Zweig and K. Makarychev. Speed regularization and optimality in word classing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8237–8241, May 2013. 2