Alternate Loss Functions for Neural Language Modeling

Yash Shah, *Second Year Undergraduate*, 160050002
Under the guidance of **Prof. Preethi Jyothi**, *IIT Bombay*

***Abstract*—The main objective of this R&D project was to use n-gram statistics to help a LSTM network learn the task of language modeling better. The inspiration behind this project was loosely based on *transfer learning*, that was introduced by *G. Hinton* in his paper "Distilling the Knowledge in a Neural Network". We aimed to train a LSTM network (*student*) under the *supervision* of a statistical n-gram language model (*teacher*), so that it learned from two sources — we built the n-gram model on the training dataset using Kneser-Ney discounting and interpolation, and used its predictive probability distribution as a *soft* target for training the LSTM network at each stage along with the standard one-hot encoded targets. We tried new loss functions, and their combinations with existing ones, to impart this n-gram information to the LSTM network. Several other architectures along the same line were tried, and we were able to beat the baselines with several of them. Expanding the training dataset from PTB to Wikitext2 forced us to optimize our code further, since each minibatch took an infeasible amount of time to run. We employed various *running-time* and *storage-space* optimization techniques to cross this bottleneck, with some success. This report talks in detail about the reading process, loss functions and model architectures employed, optimization techniques tried and results obtained over the duration of this R&D project.**

***Index Terms*—loss functions, language modeling, LSTMs**

I. Reading and Foundation

A. Research Papers

OVER the duration of this R&D project, I read the following research papers in the same order as they appear in the list. Each one imparted me knowledge about a different aspect of *language modeling* (neural language modeling in particular), which greatly helped me in my endeavors.

- "An Empirical Study of Smoothing Techniques for Language Modeling", *S. Chen and J. Goodman*, 1998
- "Recurrent Neural Network based Language Model", *Mikolov et al*, 2010
- "Recurrent Neural Network Regularization", *Zaremba et al*, 2015
- "Regularizing and Optimizing LSTM Language Models", *Merity et al*, 2017
- "On the state-of-the-art evaluation in Neural Language Models", *Melis et al*, 2017
- "Distilling the Knowledge in a Neural Network", *G. Hinton*, 2015
- "Breaking the Softmax Bottleneck", *Yang et al*, 2017
- "Neural Lattice Language Models", *J. Buckman, G. Neubig*, 2017

B. Codebase

We used the codebase already built by Kalpesh Krishna when he was working on the same project. It was predominantly written in `python2` using `Tensorflow` library with a `C++` module for storing n-gram language model in `trie` data structure. It also had scripts for plotting results, interpolating the results of two models, and mining training data from the internet.

We built upon this existing framework by adding new model architectures, optimizing the `C++` code to make it feasible for large datasets, adding scripts to map `non-ASCII` tokens to rare tokens and also new `python2` code for efficiently storing dense n-gram probability distribution tensors on disk with minimal loss in information.

II. Discussion

A. *"Regularizing and Optimizing LSTM Language Models", Merity et al, 2017*

Recurrent Neural Networks and their variations are very likely to overfit the training data. This is due to the large network formed by unfolding each cell of the RNN, and *relatively* small number of parameters (since they are shared over each time step) and training data. Thus, the perplexities obtained on the test data are often quite larger than expected. Several attempts have been made to minimize this problem using varied **regularization** techniques. This paper tackles this issue by proposing a model that combines several of such existing methods.

Merity et al's model is a modification of the standard **LSTM** in which *DropConnect* is applied to the hidden weights in the *recurrent* connections of the LSTM for regularization. The dropout mask for each weight is preserved and the same mask is used across all time steps, thereby adding negligible computation overhead. Apart from this, several other techniques have been incorporated :

- **Variational dropout:** The same dropout mask is used for a particular recurrent connection in both the forward and backward pass for all time steps. Each input of a mini-batch has a separate dropout mask, which ensures that the regularizing effect due to it isn't identical across different inputs.
- **Embedding dropout:** Dropout with dropout probability p_e is applied to word embedding vectors, which results in new word vectors which are identically zero for the dropped words. The remaining word vectors are scaled by $\frac{1}{1-p_e}$ as compensation.

- **AR and TAR:** AR (Activation Regularization) and TAR (Temporal Activation Regularization) are modifications of L_2 regularization, wherein the standard technique is applied to dropped *output activations* and dropped *change in output activations* respectively. Mathematically, the additional terms in the cost function J are (here α and β are scaling constants and $\mathbf{D}$ is the dropout mask)

$$J_{AR} = \alpha L_2 \left(\mathbf{D}_l^t \odot h_l^t\right) \tag{1}$$

$$J_{TAR} = \beta L_2 \left(\mathbf{D}_l^t \odot \left(h_l^t - h_l^{t-1}\right)\right) \tag{2}$$

- **Weight tying:** In this method, the parameters for word embeddings and the final output layer are shared.
- **Variable backpropagation steps:** A random number of BPTT steps are taken instead of a fixed number, whose mean is very close to the original fixed value (s). The BPTT step-size (x) is drawn from the following distribution (here $\mathcal{N}$ is the Gaussian distribution, p is a number close to 0.95 and σ^2 is the desired variance) :

$$x \sim p \cdot \mathcal{N}\left(s, \sigma^2\right) + (1-p) \cdot \mathcal{N}\left(\frac{s}{2}, \sigma^2\right) \tag{3}$$

- **Independent sizes of word embeddings and hidden layer:** The sizes of the hidden layer and word embeddings are kept independent of each other.

The paper also introduces a new optimization algorithm, namely **Non-monotonically Triggered Averaged Stochastic Gradient Descent** or **NT-ASGD**, which can be described as follows :

Algorithm 1: Non-monotonically Triggered ASGD (NT-ASGD)

Input: Initial point w_0, learning rate γ, logging interval L, non-monotone interval n
Output: parameter(s) that minimize the objective function f
Initialize $k \leftarrow 0, t \leftarrow 0, T \leftarrow 0, logs \leftarrow []$
while *stopping criterion not met* **do**
  $w \leftarrow w - \gamma \nabla_w f$
  if *mod*(k, L) *= 0 and* $T = 0$ **then**
    $v \leftarrow$ perplexity(w)
    if $t > n$ *and* $v > \min_{l \in \{t-n, \cdots, t\}} logs[l]$ **then**
      Set $T \leftarrow k$
    end
    Append v to logs
    $t \leftarrow t + 1$
  end
end
return $\frac{\sum_{i=T}^{k} w_i}{k - T + 1}$

They also combined their **AWD-LSTM** (ASGD Weight Dropped LSTM) with a neural cache model to obtain further reduction in perplexities. A *neural cache model* stores previous states in memory, and predicts the output obtained by a *convex combination* of the output using stored states and the AWD-LSTM.

***Network description**: Merity et al*'s model used a 3-layer weight dropped LSTM with dropout probability 0.5 for PTB corpus and 0.65 for WikiText-2, combined with several of the above regularization techniques. The different hyperparameters (as referred to in the discussion above) are as follows - hidden layer size (H) = 1150, embedding size (D) = 400, number of epochs = 750, L = 1, n = 5, learning rate = 30, gradients clipped at 0.25, p = 0.95, s = 70, σ^2 = 5, α = 2, β = 1, dropout probabilities for input, hidden outputs, final output and embeddings as 0.4, 0.3, 0.4 and 0.1 respectively. Mini-batch size of 40 was used for PTB and 80 for WT-2. Word embedding weights were initialized from $\mathcal{U}\left[-0.1, 0.1\right]$ and all other hidden weights from $\mathcal{U}\left[-\frac{1}{\sqrt{1150}}, \frac{1}{\sqrt{1150}}\right]$.

***Result highlights**:* The authors obtained the following results for their model

- 3-layer AWD-LSTM with weight tying attained 57.3 PPL on PTB
- 3-layer AWD-LSTM with weight tying and a continuous cache pointer attained 52.8 PPL on PTB

III. Experiments

A. Model architecture

The base model was a 2 layer LSTM network with `size` 650 and was unrolled across 35 timesteps with gradients clipped at 5.0. We had applied *dropout masks* to the input, final output and tensors being passed between LSTM cells (across both layers and timesteps) with *dropout probabilities* 0.5, 0.5, 0.5 and 0 respectively.

The statistical model was of *order 3* and used *Kneser-Ney discounting* and *interpolation* for all n-grams We used this architecture to try[1] the following variants of loss functions[2]

1) ***Introducing temperature in output softmax layer**:* The standard *softmax* function, as applied to logits $\hat{\mathbf{o}} \in \mathcal{R}^{D \times 1}$), can be expressed as

$$o_i = \text{softmax}(\hat{o}_i) = \frac{\exp(\hat{o}_i)}{\sum_{j=1}^{D} \exp(\hat{o}_j)} \tag{4}$$

This operation results in a vector whose elements sum to 1 ($\sum_{i=1}^{D} o_i = 1$), and hence it is often used to model a probability distribution. However, it is often observed that a relatively large value in one dimension would concentrate the probability mass around itself leaving other dimensions with negligible value.

Thus, in order to *smoothen* the resulting probability distribution, the *softmax* operation is modified by introducing a *temperature* term (T) as follows

$$o_i = \text{softmax}(\hat{o}_i) = \frac{\exp\left(\frac{\hat{o}_i}{T}\right)}{\sum_{j=1}^{D} \exp\left(\frac{\hat{o}_j}{T}\right)} \tag{5}$$

Using higher values of T will generate a more *even* probability distribution, thereby enabling all dimensions to learn more

[1] In all experiments, we used *stochastic gradient descent* optimization, with *initial learning rate* 1.0 that decayed by 0.8 every epoch, after 13 epochs. We trained with a *batch size* of 20 for 50 epochs.

[2] See Appendix for gradient calculations

uniformly. We tried using values 3, 5, 10 for T in the *softmax* operation of the output layer, guided by the intuition that such a step will help to pass *n-gram information* from the loss function (in the form of gradients) in a better manner.

*2) **Using** $\mathcal{L}_1 + \mathcal{L}_2$ **loss**:* $\mathcal{L}_1$ loss is just another name for *cross-entropy* loss. Thus, for a given input sequence $\mathbf{x} = \{x_t\}_{t=1}^T$, predicted distribution $\mathbf{o} = \{o_t\}_{t=1}^T$ and expected output $\mathbf{y} = \{y_t\}_{t=1}^T$ (here x_t's are the word-vectors of 'input' word, y_t's are one-hot encoded vectors for the 'next' word, and o_t's are the softmax outputs), it is defined as

$$\mathcal{L}_1(\mathbf{x}) = \sum_{t=1}^{T} \sum_{v \in V} -y_{v,t} \log(o_{v,t}) \tag{6}$$

The $\mathcal{L}_2$ loss is similar to $\mathcal{L}_1$ loss except that here the target distribution is not the one-hot encoded vector of the next word, but instead the predictive probability distribution of the next word over the entire vocabulary, generated by the statistical n-gram model. If we denote the target distribution by $\mathbf{P}_{NG}$ and the model's prediction by $\mathbf{P}$, then $\mathcal{L}_2$ loss is defined as

$$\mathcal{L}_2(\mathbf{x}) = \sum_{t=1}^{T} -\mathbf{P}_{NG,t} \cdot \log(\mathbf{P}_t) \tag{7}$$

$$\mathbf{P}_{NG,t} \cdot \log(\mathbf{P}_t) = \sum_{v \in V} P_{NG,t}(v|c_t) \log(\mathrm{P}_t(v|c_t)) \tag{8}$$

where c_t denotes the context encountered till timestep t. Thus, $\mathcal{L}_2$ loss is the *cross-entropy* between $\mathbf{P}_{NG}$ and $\mathbf{P}$ distributions. A different way of looking at $\mathcal{L}_2$ loss is that it takes some of the 'penalty' for incorrectly predicting the expected word, and distributes it over all *other* words in the vocabulary — thereby ensuring that the model learns *not* to predict incorrect words *too*. We used a linear combination of $\mathcal{L}_2$ and $\mathcal{L}_1$ loss as our loss function at the output layer. If the contribution of $\mathcal{L}_1$ is denoted by η

$$\mathcal{L}_{total}(\mathbf{x}) = \eta \cdot \mathcal{L}_1(\mathbf{x}) + (1 - \eta) \cdot \mathcal{L}_2(\mathbf{x}) \tag{9}$$

*3) **Using** $\mathcal{L}_1 + \mathcal{L}_3$ **loss**:* $\mathcal{L}_3$ loss, or *Conflict-Averse* loss is the *cross-entropy* between $\mathbf{P}_{NG}$ and $\mathbf{P}$ distributions, but in the reverse order as $\mathcal{L}_2$ loss. Thus,

$$\mathcal{L}_3(\mathbf{x}) = \sum_{t=1}^{T} -\mathbf{P}_t \cdot \log(\mathbf{P}_{NG,t}) \tag{10}$$

$\mathcal{L}_3$ loss has little meaning of its own, but in combination with $\mathcal{L}_1$ or $\mathcal{L}_2$ loss, it serves as a 'check' to keep predicted probabilities of words not expected at that timestep (according to the n-gram model) low, by penalizing $\mathbf{P}(v|c_t)$ with $-\log(\mathbf{P}_{NG}(v|c_t))$. Thus, any 'conflict' between the neural and statistical would shoot the $\mathcal{L}_3$ loss up, thereby teaching the model to 'averse' such a prediction.

We tried a linear combination of $\mathcal{L}_1$ and $\mathcal{L}_3$ losses, with the weight of $\mathcal{L}_1$ loss as η', at the output layer

$$\mathcal{L}_{total}(\mathbf{x}) = \eta' \cdot \mathcal{L}_1(\mathbf{x}) + (1 - \eta') \cdot \mathcal{L}_3(\mathbf{x}) \tag{11}$$

Since $\mathcal{L}_3$ loss is not useful by itself, we would expect the value of η' to be significantly smaller than η if both models were to give similar results.

*4) **Introducing intermediate** $\mathcal{L}_2$ **loss**:* Instead of introducing the $\mathcal{L}_2$ loss at the output layer, we tried applying it independently to intermediate layers. This way, the outer layer should learn from the 'strict' one-hot distribution, whereas the intermediate layers should learn predominantly from the n-gram distribution.

If the activations of intermediate layers are $\mathbf{h}_i^t$, with $i \in 1, 2 \ldots n-1$ and those of the output layer are $\mathbf{h}_n^t$, the final loss would be[3]

$$\mathcal{L}_{total}(\mathbf{x}) = \eta_n \mathcal{L}_1(\mathbf{h}_n^t) + \sum_{i=1}^{n-1} \eta_i \mathcal{L}_2(\mathbf{W}_i \cdot \mathbf{h}_i^t + \mathbf{b}_i) \tag{12}$$

$$\text{where } \sum_{i=1}^{n} \eta_i = 1 \tag{13}$$

*5) **Introducing intermediate** $\mathcal{L}_3$ **loss**:* Identical to previous setup, except we use $\mathcal{L}_3$ loss instead of $\mathcal{L}_2$. The total loss thus becomes[3]

$$\mathcal{L}_{total}(\mathbf{x}) = \eta'_n \mathcal{L}_1(\mathbf{h}_n^t) + \sum_{i=1}^{n-1} \eta'_i \mathcal{L}_3(\mathbf{W}'_i \cdot \mathbf{h}_i^t + \mathbf{b}'_i) \tag{14}$$

$$\text{where } \sum_{i=1}^{n} \eta'_i = 1 \tag{15}$$

B. Running time optimization

The codebase is predominantly written in `python2`, with a `C++` module for creating and handling the backoff n-gram language model. Testing the forementioned architectures on the PTB dataset using the original codebase took around 0.8 seconds per minibatch on `voxel10`, and thus the model took roughly 8 hrs to train. However, on expanding the dataset from PTB to Wikitext2, the *time per minibatch* shot up to 3 seconds and the *overall training time* to around a week. This was highly undesirable since hyperparameter tuning would itself gobble up a lot of time.

TABLE I
COMPARISION OF PTB AND WIKITEXT2 DATASETS

Parameter	PTB			Wikitext2		
	Train	Valid	Test	Train	Valid	Test
Tokens	887,521	70,390	78,669	2,088,628	217,646	245,569
Vocabulary	10,000			33,278		
OoV	4.8%			2.6%		

After analyzing the time taken by each module, we identified the bottleneck to be the generation of n-gram probability distribution for every timestep, given the context. Shifting to Wikitext2 had increased both the vocabulary size (leading to larger tensors) as well as the number of minibatches (due to more training tokens), thereby rendering the current codebase infeasible against large datasets.

[3]Note that the representation of $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_2$ losses are different than before. Here, they are used to indicate the probability distribution which will be used alongside the one-hot or n-gram distribution respectively.

1) ***C++ SWIG wrappers****:* We had used the `SRILM` toolkit for generating the n-gram *ARPA* file, followed by a `C++` script (which was integrated with `python2` via the *SWIG* interface) for storing the language model along with backoff weights using *trie* data structure (it was the best trade-off between speed and memory requirement).

The n-gram probability distribution over the vocabulary had to be generated in batches of dimension *(batch_size, timesteps, vocab_size)* (that is, *(20, 35, 10000)* for PTB and *(20, 35, 33278)* for Wikitext2). The original code iterated over *batch_size* and *timesteps* using *for* loops in the `python2` code itself, calling the `C++` module only for filling the corresponding distribution vector of *vocab_size* size. We shifted these *for* loops from `python2` to the `C++` module to get an inherent speedup due to the faster execution time for `C++`.

2) ***Multithreading****:* Although shifting *for* loops from `python2` to `C++` did have some benefit, it was not sufficient. Further analysis of the manner in which the probability distribution was generated revealed that each of the *batch_size * timesteps* iterations could be run in parallel, since they were completely independent of each other — a separate context vector had been used for each iteration in the earlier code, and so each of the *batch_size * timesteps* distribution vectors of *vocab_size* size could be filled independently.

We used the *pthread* library and modified the `C++` code so that the task was divided amongst a constant number of threads (*8* for `voxel10` and *48* for `dhvani`) which ran concurrently. We stuck with the notion of *"one thread per core"* aiming to maximally use the computational power of each server.

However, we weren't able to achieve the expected speedup since in *Linux*, threads are assigned a core by the scheduler depending on the CPU load, and we have little control over the same. There are utilities such as *taskset* that allow the user to manually assign a processor core to a process (identified by its *PID*), but since all threads have the same *PID* in our case, it was of no use to us. Thus, introducing *multithreading* improved our running time for **generating n-gram probability distribution** from around **2.8 seconds** to **0.8 seconds** (on `dhvani` server) and **1.3 seconds** (on `voxel10`) against the expected (and desired) factor-10 improvement.

3) ***Multiprocessing****:* We believed that we could improve the running time further if we were able to assign one instance (out of the *batch_size * timesteps* instances) to one core at a time. We therefore decided to try *multiprocessing* instead of *multithreading*, since the former *actually* uses separate cores for different *processes*. Since the `OpenMP` library for `C++` had a steep learning curve and was incompatible with *SWIG*, we decided to shift the `C++` module back to `python` and use its *multiprocessing* library instead.

We spawned processes equal in number to the *cpu_count* of the server. Each process was assigned one of the *batch_size * timesteps* distribution-generation instances as soon as it became free, and this was done till the entire tensor of dimension *(batch_size, timesteps, vocab_size)* was filled. However, this didn't work well since the overhead of **allocating** a separate memory space and **copying** the large tensors to it shadowed the improvement in computation time, if any. We even tried *sharing the memory* of parent process with its children to reduce the "copying" overhead, but in vain. The execution time for the task *increased* from **2.8 seconds** to around **7 seconds** instead.

4) ***Inference****:* So far, we found the best option to be the multithreaded *C++ SWIG* code which took roughly **1.1 seconds** overall on `dhvani`, and **1.5 seconds** on `voxel10`. This improvement was still infeasible, since the model took around **3 days** to train for Wikitext2 dataset. We therefore decided to try optimizing *storage-space* instead of *running-time*, since the n-gram distributions could be accessed quickly (~ 0.05 seconds) multiple times once they have been generated and saved on disk.

C. Storage space optimization

Generating the n-gram distributions for each minibatch beforehand and storing them on disk does seem to be the perfect solution, except for one aspect — the disk space required for the same (without any optimization) is close to **600 GB**.

$$\text{Space required} = \text{space required for 1 batch * no. of batches}$$
$$= 187 * 2983 \text{ MB} = 557.821 \text{ GB}$$

The space requirements are huge since we need to store each and every element of the n-gram distribution tensor. This is because for the current implementation, these tensors are **dense** i.e. the fraction of zero entries is very small. We knew that sparse matrices can be efficiently stored on disk — we thus decided to *sparsify* the n-gram tensors, and at the same time ensure that such a step didn't alter the information stored within them.

Thresholding n-gram distribution*:* In this method, we decide on a *threshold value* (θ) and replace all probabilities less than it by 0 in the given tensor $\mathbf{x}$. This 'sparsifies' the tensor thereby making it easier to store. We denote the fraction of zero entries in a tensor by its *sparsity index*, α. Thresholding does pose a problem — we need to ensure that the probability mass of non-zero values is still close to 1, otherwise a significant amount of n-gram information has been lost. We denote the margin by which the new total probability mass may be off from 1 by *allowance*, or δ.

The *sparsity index* and *loss incurred* (Δ) for a given *threshold* θ and *allowance* δ can be expressed as

$$\text{S.I.}_\theta(\mathbf{x}) = \alpha_\theta(\mathbf{x}) = \frac{\sum_{i=1}^{V} \mathbf{1}_{\{x_i < \theta\}}}{V} \tag{16}$$

$$\Delta_{\delta,\theta}(\mathbf{x}) = \max\left\{0, 1 - \left(\sum_{i=1}^{V} x_i \cdot \mathbf{1}_{\{x_i \geq \theta\}}\right) - \delta\right\} \tag{17}$$

where $\mathbf{1}_\beta$ is the *indicator function*, which evaluates to 1 if β is true, and 0 otherwise.

Since increasing the value of θ would increase α_θ for any given $\mathbf{x}$, we need to solve the following constraint problem for the chosen value of δ

$$\arg\max_{\theta} \min_{\theta} \Delta_{\delta,\theta}(\mathbf{x}) \tag{18}$$

That is, we need to find the maximum value of θ which minimizes the loss Δ, given $\mathbf{x}$ and δ.

IV. Results

A. Penn Treebank Dataset

We used the positive results obtained on this relatively smaller dataset to improve performance over a larger dataset such as WikiText-2.

Temperature variation: In order to confirm whether increasing temperature had any effect on the model's performance, we compared the validation perplexities of the base model with $T = 1, 3, 5, 10$. We inferred that increasing temperature had little effect on performance — it degraded performance instead.

TABLE II
VALIDATION PERPLEXITIES FOR DIFFERENT TEMPERATURES

Loss function	T = 1	T = 3	T = 5	T = 10
$\mathcal{L}_1$	78.4071	81.4264	83.0821	91.7688

Using $\mathcal{L}_1 + \mathcal{L}_2$ loss: The following results were found, and seemed promising.

TABLE III
PERFORMANCE WITH DIFFERENT η VALUES

Loss function	η = 1	η = 0.5	η = 0
$\mathcal{L}_1 + \mathcal{L}_2$	78.4071	75.7233	138.2089

Using $\mathcal{L}_1 + \mathcal{L}_3$ loss: Introducing $\mathcal{L}_3$ loss didn't seem to benefit, since the perplexities increased rapidly as the contribution of $\mathcal{L}_3$ loss was increased. This discouraged us to try $\mathcal{L}_3$ loss for intermediate layers.

TABLE IV
PERFORMANCE WITH DIFFERENT η' VALUES

Loss function	η' = 1	η' = 0.99	η' = 0.5
$\mathcal{L}_1 + \mathcal{L}_3$	78.4071	82.1446	221.3349

With $\mathcal{L}_2$ loss seeming promising, we shifted to a larger dataset (WikiText-2) for further experiments.

B. Wikitext2 Dataset

For this dataset, we dropped the *temperature* experiments for the time being, since they hadn't provided satisfactory results with PTB. The baseline perplexities are mentioned in table IV-B.

We were able to run only a few experiments on this dataset owing to its vocabulary and number of tokens, both of which made the task of generating n-gram distribution quite slow.

TABLE V
BASELINE PPL FOR WIKITEXT-2

Model	Validation perplexity
Pure $\mathcal{L}_1$	97.8665
Pure $\mathcal{L}_2$	219.8655
n-gram	265.7427

TABLE VI
USING $\mathcal{L}_1 + \mathcal{L}_2$ FOR WIKITEXT-2

$\eta/1-\eta$	Validation perplexity
0.7/0.3	96.7805
0.75/0.25	95.5355
0.8/0.2	**92.7879**
0.85/0.15	95.9744
0.9/0.1	93.3825
0.95/0.05	95.8202
0.99/0.01	99.4833

TABLE VII
USING $\mathcal{L}_1 + \mathcal{L}_3$ FOR WIKITEXT-2

$\eta'/1-\eta'$	Validation perplexity
0.8/0.2	124.8758
0.9/0.1	107.13
0.99/0.01	99.7766

V. Conclusion

We observed that combining $\mathcal{L}_2$ with $\mathcal{L}_1$ loss led to improvement in perplexity. Introducing loss functions, most probably $\mathcal{L}_2$, for intermediate layers might be beneficial. Increasing temperature did not better the base model when $\mathcal{L}_1$ loss was used, but improvement might be possible for other loss functions. As of now, WikiText-2 (and other large datasets) cannot be used for training our model since the n-gram distribution generation takes an infeasible amount of time to run. We will be looking for space-optimization techniques for efficiently storing and accessing batch n-gram distributions from disk.

Appendix A
Gradient calculations for softmax function

The temperature simply results in an extra factor of T as below

$$\nabla_{\hat{o}_i} S_j = \frac{1}{T} \cdot S_j(\delta_{ij} - S_i) \tag{19}$$

Appendix B
Gradient calculations for $\mathcal{L}_2$ loss

Since $\mathcal{L}_2$ loss is very similar to *cross-entropy* loss, the gradient calculations are similar. Using the notation as introduced earlier,

$$\nabla_{\hat{o}_i} \mathcal{L}_2(x_t) = \sum_{j=1}^{V} -\frac{P_{NG}(w_j|c_t)}{P(w_j|c_t)} \cdot P(w_j|c_t)(\delta_{ij} - P(w_i|c_t)) \tag{20}$$

$$= \sum_{j=1}^{V} -P_{NG}(w_j|c_t) \cdot (\delta_{ij} - P(w_i|c_t)) \tag{21}$$

APPENDIX C
GRADIENT CALCULATIONS FOR $\mathcal{L}_3$ LOSS

It is important to realize that $\mathcal{L}_3$ loss is merely a 'scaled' sum of the output's values along all dimensions — with the $\mathbf{P}(v|c_t)$ term being scaled by $-\log(\mathbf{P}_{NG}(v|c_t)$. Since the n-gram distribution is independent of the model's parameters, it can be treated as a constant while computing gradients. Thus, gradient calculation for $\mathcal{L}_3$ loss reduces to that of softmax operation.

$$\nabla_{\hat{o}_i}\mathcal{L}_3(x_t) = \sum_{j=1}^{V} -\log(P_{NG}(w_j|c_t)) \cdot P(w_j|c_t)(\delta_{ij} - P(w_i|c_t)) \tag{22}$$

$$\nabla_{\hat{o}_i}\mathcal{L}_3(x_t) = \sum_{j=1}^{V} -\log(P_{NG}(w_j|c_t)) \cdot \hat{o}_j(\delta_{ij} - \hat{o}_i) \tag{23}$$

ACKNOWLEDGMENT

I would like to sincerely thank Prof. Preethi Jyothi for constantly guiding and supporting me throughout the duration of this R&D project, and for providing me the opportunity and resources for the same.