Hierarchical softmax methods for training large vocabulary language models

Andrei-Alexandru Apostoae

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2019

Abstract

Word-level language modeling is a core natural language processing task that aims to create a probability distribution over the next word given a sequence (context). Modern datasets, however, contain hundreds of thousands of unique words, which makes computing softmax over the whole vocabulary either very slow or intractable. In recent years, several methods have been proposed for approximating the softmax layer in order to achieve better training speeds.

This project presents a benchmark over hierarchical softmax methods and AWD-LSTM, including the adaptive softmax and adaptive input representations that are used in state of the art models such as Transformer-XL [7]. Depending on the model complexity and the dataset size, we show that adaptive methods can lead to up to 2.64x faster training speed than softmax while achieving up to 13.2% lower test perplexity. Moreover, we propose novel ways of clustering words based on their meaning and compare them to the frequency-based standard from the literature.

Acknowledgements

I would first like to thank **Ben Allison** for being one of the best supervisors I have ever had. Not only he was always available to offer help and answer our questions, but also created an interesting research group among his students, helping us all study the problem of *large softmaxes*.

Secondly, I want to thank my family for believing in me and making the financial efforts to allow me to live abroad and study at The University of Edinburgh.

I am also grateful to all the friends that were close to me when times were hard during the degree, especially Tudor, Gabriel, Mihai, Lukas and Martin.

And last but not least, I would like to thank Google for offering us free trials of their cloud platform, which allowed us to do large experiments that we otherwise would not have been able to do.

Table of Contents

Chapter 1

Introduction

Natural language processing (NLP) has seen large improvements in the last decade with the rise of deep learning hardware, as well as the adoption of recurrent neural networks (RNNs) that are able to model theoretically infinite contexts. Language modeling is one of the most significant NLP tasks and has benefited from these improvements. Applications that rely on language modeling include automatic speech recognition and machine translation.

In word-level language modeling a prediction for the next word is made at each timestep given the sequence of previous words. Historically, language models were created using non-parametric models based on n-grams and smoothing methods, such as Kneser-Ney [11]. Neural network language models (NNLMs) came as a replacement, ranging from feed-forward [2] to log-bilinear [32], recurrent [30] or more recently transformer-based networks [39]. All NNLMs share a similar structural pattern: they summarize the context in a hidden state, which is projected through a weight matrix and then passed through a normalization layer (commonly softmax) that outputs the word probabilities. Training a NNLM is therefore equivalent to solving a classification problem, where each class is an unique word from the dataset.

However, datasets vary in terms of total word tokens and vocabulary sizes – recent datasets, such as One Billion Word [5] have more than 800,000 unique words. When dealing with large vocabulary sizes, passing the hidden state through the normalization layer becomes a computational challenge [6]: the matrix operations become slower and the memory footprint increases significantly. Furthermore, the total number of words is usually proportional to the vocabulary size, requiring overall more training time to finish a single epoch. Solving this problem would not only improve the language models, but could be used in other classification tasks (outside of NLP).

To deal with large vocabularies, several methods to approximate the softmax output have been proposed in the last decade [38]. The most popular methods involve changing the structure of the network after the last hidden state [34, 6, 18]. Other methods are based on sampling to approximate the softmax function [33] and self-normalization [8] in order to eliminate the need of softmax.

This project offers an overview over hierarchical methods [34], which fall in the first category of softmax approximation methods. In hierarchical softmax, a multi-level tree is created containing leaves as words and intermediary nodes as clusters. The probability of a word being present after a sequence is computed by following the path from the root of the tree to the leaf attributed to the word. Depending on the structure of the tree, the training time and accuracy may have high variance, therefore having a good hierarchy becomes key in achieving good performance in a small amount of time when compared to computing the full softmax. Applications that run on low end hardware and/or require very fast predictions are the main beneficiaries of hierarchical methods.

Along with hierarchical softmax methods that simply group the words according to frequency and use their parents on the first level of the tree [6], we also present adaptive softmax methods. The words in all English NLP datasets are distributed according to Zipf's law, suggesting that words that appear more often should be given more capacity [12]. Differently from the initial formulation of the hierarchical softmax, adaptive softmax [12] uses the most frequent words on the first level instead of grouping them and uses a flexible embedding size schema such that GPU matrix operations are optimized. Moreover, the inputs can also be adaptive [1], which further improves the training time and is the state of the art in training transformer-based language models.

From a novelty perspective, we propose new criteria for creating clusters using pre-trained embeddings from previous experiments or online libraries. Additionally, we evaluate adaptive input representations using LSTMs instead of transformers, which has not been done to the best of our knowledge. Finally, we benchmark these methods and criterias on several datasets that vary in size in order to decide when full softmax should be replaced with a flavor of hierarchical softmax.

As for limitations, we are evaluating the algorithms using a 3-layered AWD-LSTM [26] in a constrained amount of time (depending on dataset). Although the trend in language modeling is to use multiple GPU/TPU setups, we are using two machines with high-end GPUs and different amounts of RAM, which is very relevant for our large vocabulary problem. In order to have a fair evaluation, we also restrict the total

number of clusters to 3 such that we comply to previously reported results for the adaptive methods.

We find that for reasonably large datasets (over 50,000 words) and when using weight tying [37, 15], adaptive softmax outperforms full softmax and non-adaptive hierarchical methods – at least 70% faster while obtaining more than 9% lower test perplexity. We then show that adaptive inputs are useful from a memory usage perspective, while their accuracy is generally worse than tied adaptive softmax. Finally, our experiments further consolidate the frequency-based clustering for adaptive methods and propose a version of weighted k-means for non-adaptive softmax that outperforms previous clustering algorithms from the literature.

In section 2, we present language modeling as a deep learning task and motivate using strong regularization when using recurrent neural networks. Furthermore, we discuss the various flavors of hierarchical softmax with respect to their theoretical and practical advantages. Section 3 outlines how the benchmark will be constructed – details about the datasets, model, methods and evaluation can be found here. The experiments are conducted in section 4 and presented on each of the datasets, followed by a discussion in section 5. We draw our conclusions in section 6 and then propose possible directions for future work.

Chapter 2

Background

2.1 Language modeling

Language modeling is a core natural language processing task with applications ranging from improving statistical machine translation [40] and speech recognition [13] systems to text generation [17]. Its goal is to create an accurate probabilistic distribution for the next token (either word or character) given a sequence of tokens, called context. Although newer approaches combine word and character inputs for better performance, this study will only cover word-level language modeling. The probability of a word sequence can be expressed as follows:

$$P(w_1 w_2 ... w_N) = \prod_{i=1}^{N} P(w_i | w_1 w_2 ... w_{i-1}) = \prod_{i=1}^{N} P(w_i | c_{i-1}) \tag{2.1}$$

where w_i is the ith word in the sequence, c_i is the context after i words and N is the sequence length.

Historically, this probability was estimated using non-parametric models, such as n-grams, then by feedforward neural networks and log-bilinear models. Both classes, however, can only use context of fixed sizes under the Markov assumption. A small context size is unable to track long dependencies between words, while increasing the context size introduces other problems, such as data sparsity and computational complexity.

The introduction of recurrent neural networks, such as LSTMs [14], in the context of language modeling [30] solved the fixed context size issue. While they are generally less parallelizible and more difficult to train than the previous methods, they led to state of the art results on the majority of NLP tasks.

Currently, the trend in language modeling is to no longer use recurrent networks, but very large models based on transformers [39]. These models usually require more multiple high-end GPUs or TPUs [42], very large datasets and days of training, which makes it impractical for researchers without powerful hardware to train or use them.

2.1.1 Neural network language models

Although there are multiple classes of networks for language modeling, the output of the network is always a probability distribution f. This distribution is obtained by multiplying the last hidden layer h of the network with a weight matrix W_{out} (output embedding layer) and applying a bias b, then normalizing the outputs (logits z) through a special function. The last hidden layer is obtained by passing the input through multiple layers, either recurrent or feedforward.

$$z = h^T W_{out} + b \tag{2.2}$$

To obtain f, the logits are divided by a normalization constant Z, resulting in positive values that sum up to 1. The normalization process is called **softmax**.

$$Z = \sum_{j=1}^{V} exp(z_j) \tag{2.3}$$

At a timestep t, f can be computed as:

$$f = P(w_t|c_{t-1}) = \frac{1}{Z} exp(z) \tag{2.4}$$

Finally, in order to train the network, the negative log-likelihood of the word probabilities is minimized, which is equivalent to optimizing the cross-entropy loss between the predicted distribution f and the target distribution y.

Let c_t be the correct class of the observation at timestep t. y_t therefore is a one-hot encoded vector with 1 on position c_t and 0 elsewhere. The cross-entropy loss of example t can be expressed as follows:

$$J_t = -log f_{c_t} = -log \frac{exp(z_{c_t})}{\sum_j exp(z_j)} = -z_{c_t} + log \sum_j exp(z_j) \tag{2.5}$$

The gradient of the loss with respect to the logits is backpropagated afterwards:

$$\frac{\partial J_t}{\partial z_k} = -\delta_{c_t k} + f_k \tag{2.6}$$

where $\delta_{ij} = 1$ when $i = j$, 0 otherwise.

In practice, the training is done in minibatches, and validation and test perplexities are used instead to account for the small difference in loss values. For a dataset D, we compute the average loss $\overline{J}$:

$$\overline{J_D} = \frac{\sum_i^{|D|} J_{D_i}}{|D|} \tag{2.7}$$

$$pp(D_{val}) = exp(\overline{J_{D_{val}}}) \tag{2.8}$$

The language models based on recurrent neural networks also benefit from a technique called **backpropagation through time** (BPTT), in which the network is unfolded and gradient accumulates across multiple timesteps from the past. Although this improves the training performance, it introduces the problem of **vanishing gradients**: the gradients become smaller as the number of BPTT steps increases. LSTMs are therefore preferred when doing language modeling, as they prevent the vanishing gradients problem by design, compared to simpler models such as Elman RNNs.

2.1.2 AWD-LSTM

In order to obtain the best performance, the depth and breadth of neural networks have increased over time. This led models to have millions of parameters and, depending on the dataset size, overfitting often occurs. Dropout is the most common solution to this problem, where layer activations are randomly zeroed at every forward and backward pass. Dropout not only improves generalization, but is also very efficient.

However, in the case of recurrent neural networks, applying dropout is not as straightforward as in feedforward networks. AWD-LSTM [26] proposes new strategies of training and applying dropout when training language models, while not changing the LSTM implementation that comes in deep learning frameworks.

Dropout. Firstly, when dropout is used on the hidden state of an RNN, the model is incapable of learning long term dependencies [26, 43]. As a solution, AWD-LSTM uses DropConnect [41] between hidden layers, which is a dropout technique that zeroes the hidden-to-hidden weight matrices instead of the activations. For all the other activations, AWD-LSTM uses variational dropout [10]: in a forward and backward pass, the same sampled mask is used across multiple timesteps when doing the BPTT. Moreover, embedding dropout [10] is used, which assigns a dropout probability p_e for the embedding layer, then scales the word embeddings by $\frac{1}{1-p_e}$ [26].

Weight tying. The layer-wise structure of the network is created such that the embedding size and hidden layer size are different, compared to previous approaches where they had small values and were usually equal. This also allows the usage of weight tying [37, 15], a technique that vastly reduces the complexity of the network by using the same matrix as input (embedding) W_{in} and output W_{out}. Not only this accelerates training, but it was shown to also reduce overfitting, resulting in better overall performance [26].

L_2-regularization. AWD-LSTM also introduces two kinds of L2-regularization, called activation regularization (AR) and temporal activation regularization (TAR) [26], which are applied to the hidden state of the last LSTM layer in order to prevent large changes:

$$AR = \alpha \| m \cdot h_t \|_2 \tag{2.9}$$

$$TAR = \beta \| h_t - h_{t+1} \|_2 \tag{2.10}$$

where α and β are used for scaling and m is a dropout mask.

Optimizer and BPTT. Finally, the authors argue that the training process can be improved by using a novel optimizer, called *non-monotonically triggered averaged SGD* [26]. The updates are done similarly to SGD, but when a stagnation is detected on the validation set loss, the gradient is computed as the average of last few timesteps instead. Additionally, the number of BPTT steps is sampled each time instead of being fixed and the learning rate is modified accordingly. Details of the sampling method can be found in [26].

As a result, AWD-LSTM reached state of the art results in language modeling at the time of its release (2017) on multiple datasets and is still a relevant architecture for training models where computer resources are limited.

2.2 Hierarchical softmax

Assuming a vocabulary size $|V|$ and d the dimension of the last hidden layer, the complexity of the softmax operation is $O(|V|)$ and the last matrix multiplication costs $O(d|V|)$. Modern datasets, such as One Billion Word [5] and Wikitext-103 [28], have vocabulary sizes of hundreds of thousands, not only resulting in slow training, but sometimes increasing the memory usage to the point where training is not possible at all [1].

In order to accelerate the training, several ideas to approximate the softmax layer have been proposed over the years From those ideas, we mention restructuring the output layer, approximating through sampling and self-normalizing methods, where the softmax layer is dropped altogether [38].

This project will only present hierarchical softmax [34], a method that falls in the first category, as it has shown the most promising results in terms of scalability and final perplexity. The output layer has a tree-like structure, where the leaves represent unique words and the other nodes are classes. The probability of a word is computed as the product of probabilities across the path from root to the leaf corresponding to that word. It can be shown that the sum of probabilities in the leaves is 1, therefore once the path is selected, normalization is not needed anymore. Regular softmax can be visualized as a hierarchical softmax of one level and all the words being leaves [38].

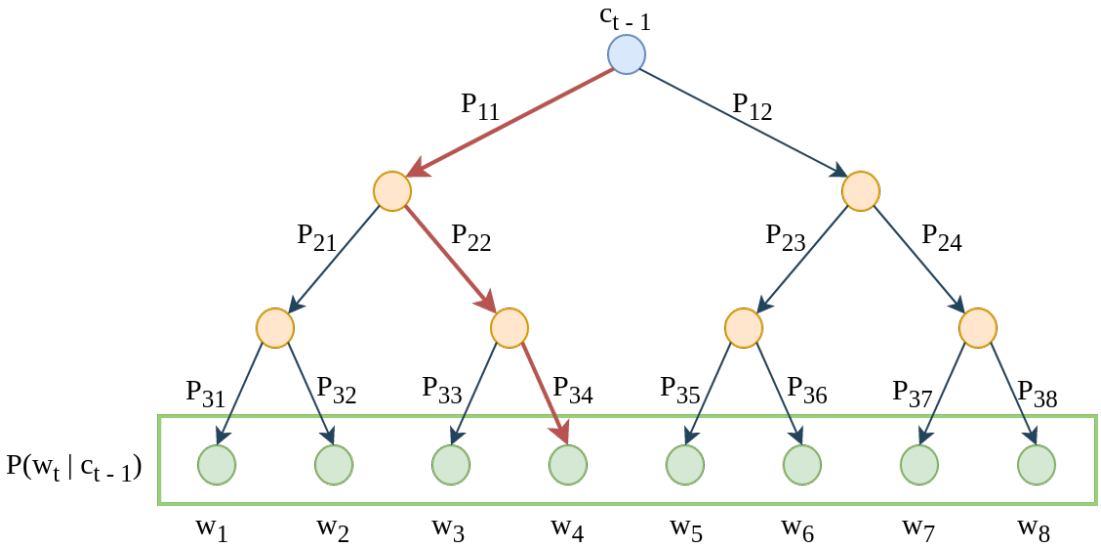

Figure 2.1: Example of a 3-level perfect binary tree hierarchy. The root of the tree is the context of the last timestep (blue color), summarized in a hidden layer, while the leaves (green color) are the probabilities of the 8 words. The yellow nodes are the intermediary clusters. For any level i, we have that $\sum_j P_{ij} = 1$. To compute the probability of one of the words (e.g. w_4) the red path is followed: $P(w_4|c_{t-1}) = P_{11} \cdot P_{22} \cdot P_{34}$. The figure is inspired from Steven Schmatz's Quora post[1].

However, designing the tree such that words can be accessed efficiently is an important challenge [6]. Although initially researchers tried achieving the maximum theoretical speed-up by creating trees with many levels [34, 32, 31], the perplexity

[1] https://www.quora.com/What-is-hierarchical-softmax/answer/Steven-Schmatz

increase was significant.

Newer strategies [6, 12, 1] involve creating two-level trees. In [6] there are $\sqrt{|V|}$ classes on the first level and each class contains an equal number of words, sorted by their frequency. This is equivalent to a computational cost of only $O(2\sqrt{|V|})$ obtained by applying softmax two times on a $\sqrt{|V|}$-sized array. The probability of word at a timestep t can be calculated as:

$$P(w_t|c_{t-1}) = P_1(C(w_t)|c_{t-1})P_2(w_t|C(w_t), c_{t-1}) \tag{2.11}$$

where $C(w_t)$ is the class of the word w_t and P_1 and P_2 are probabilities obtained by applying softmax.

In [6], the hierarchical softmax with $\sqrt{|V|}$ clusters vastly outperforms softmax and other sampling methods on One Billion Word dataset, while performing poorly on smaller datasets, such as Penn Treebank. Similarly, when restricted to 5 epochs of training and using clusters based on word similarity instead of frequency, hierarchical softmax results in close-to-softmax performance but with a speed-up factor between 2x and 5x on subsets of Europarl [12]. Overall, the larger the vocabulary is, the better hierarchical softmax will perform compared to softmax when given a limited amount of training time. However, the way the tree is built is an open-ended problem and while knowing the word similarities before training can help [12], they are not always available or transferable between datasets.

2.2.1 Adaptive softmax

While the original hierarchical softmax provides a good theoretical baseline for softmax approximation methods, in practice the results have been mixed. Although the speed-ups per epoch are present, models sometimes do not learn good representations compared to the default softmax when a budget of time is given [12, 6]. This is expected, as hierarchical methods are meant to be just an approximation of softmax.

In [12], a new method of structuring the tree hierarchy is proposed such that training is very efficient on modern GPUs. The authors observe that GPU matrix multiplications are very slow when one of the dimensions is small and suggest that strategies such as Huffman encoding are not suitable for current hardware for that reason. This method is called **adaptive softmax** and features multiple improvements over the original hierarchical softmax.

Word clustering. The words are always sorted by frequency, highest to lowest,

meaning the first row of the W_{in} embedding matrix corresponds to the most frequent word (usually "the" in English datasets). Counting the word occurrences is an inexpensive operation and does not require word similarity knowledge from a pretrained model or optimizing for additional objectives as in [44].

Shortlist. Similarly to [22], a shortlist of the most frequent words (**head** - V_h) is kept on the first layer of the tree, meaning they are all leaves of the first level. The rest of the words are split into clusters, becoming leaves on the second level of the tree. The parent nodes of these words form the **tail** - V_t.

$$\begin{aligned} V &= V_h \cup V_t \\ V_h \cap V_t &= \emptyset \\ |V_h| &\ll |V_t| \\ P(V_h) &\gg P(V_t) \end{aligned} \tag{2.12}$$

When making a prediction, softmax is applied on the first level to determine whether the word was in the shortlist or in one of the subsequent clusters. For the tail, the softmax operation is applied again and the probability of the words in the tail will be the product of the probabilities across the path, as in hierarchical softmax. As the original hierarchical softmax usually has a low number of nodes on the first layer, it is more inefficient than adaptive softmax from a GPU perspective. By splitting the words in a head and tail structure, the operations are performed on matrices with large dimensions. Not using a shortlist decreases the performance by 5-10%, as reported in [44].

Word capacity. The authors argue that attributing high embedding sizes to rare words is wasteful, as they would not have accurate representation inside the language model. Therefore, it is suggested that the clusters with the most frequent words should have the highest embedding sizes and the subsequent clusters should have lower embedding sizes. The authors propose using an exponential schema for the embedding sizes, where each cluster has the embedding size of the previous cluster divided by a fixed value, usually $div = 4$. This further improves the training speed by decreasing the total number of learned parameters.

Clusters. Adaptive softmax suggests using a small number of clusters (less than 15, usually 2-5 [12]) such that performance is optimized for GPUs. In practice, a list of **cut-offs** defines the clusters, containing the indices at which the clusters are separated given that the words are sorted by frequency.

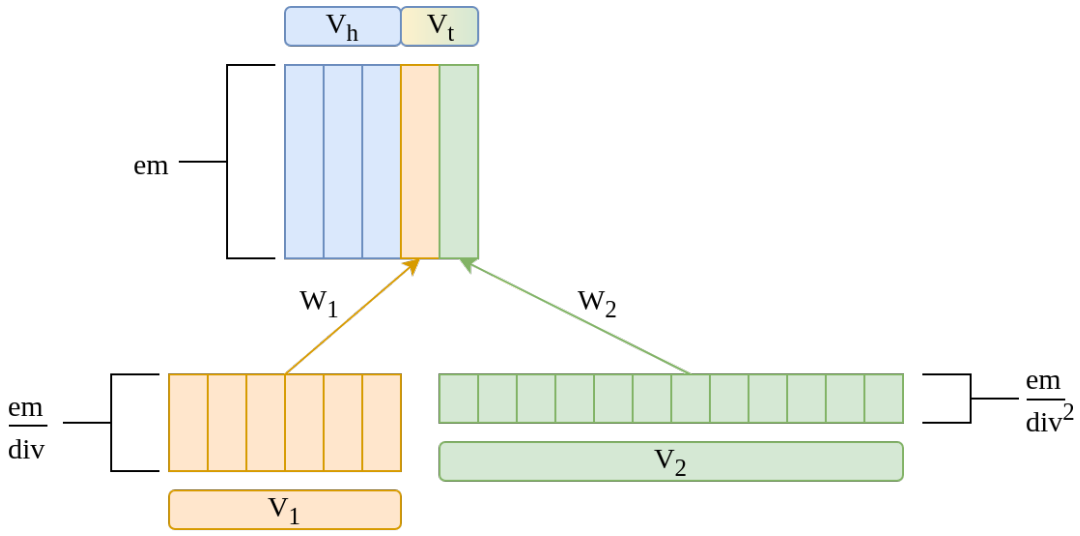

Figure 2.2: Adaptive softmax hierarchy containing 3 words in the shortlist (blue) and 2 clusters in the tail, resulting in a total of 3 clusters. The words that appear in V_h, V_1 and V_2 are sorted by frequency. Moreover, V_1 and V_2 have only $\frac{em}{div}$ and $\frac{em}{div^2}$ capacity, respectively. In order for the first layer to have a constant size of em, V_1 and V_2 are linearly projected through W_1 and W_2. In practice, V_h, V_1 and V_2 have thousands of elements, but we reduced the size for simplicity. The figure is inspired from [12].

As a result, the models reach lower perplexities than other methods, while achieving between 2x and 10x the training speed of the full softmax [12]. The authors also report comparable results to the state of the art on certain datasets using a single GPU for a few days instead of multiple GPUs for a few weeks.

Adaptive softmax as formulated in [12] has a caveat: it is incompatible by default with weight tying. The reason for this is that the decoder (previously W_{out}) has variable embedding size due to the word capacity division scheme, while the encoder W_{in} has fixed capacity. As a solution, **tied adaptive softmax** sets $div = 1$ and therefore rare words are attributed the same capacity as the frequent words [27]. Weight tying is then applied normally (while the shortlist is maintained) resulting in a reduction in parameters that is more significant than using $div > 1$ and no weight sharing.

2.2.2 Adaptive input representations

Adaptive input representations [1] is a method that combines the variable word capacity used in adaptive softmax [12] and weight tying, resulting in better memory usage than tied adaptive softmax [27].

In order to apply weight tying, the encoder and decoder have to share the same sizes. The encoder is therefore modified to not be just the matrix W_{in}, but to respect the structure of adaptive softmax instead. The words are split into clusters according to V_h and V_t on the input layer and before feeding the input to the LSTM layers, linear projection matrices are used for each tail cluster and the output is concatenated. As adaptive softmax already uses these projections in a similar manner, they can be shared therefore reducing the number of parameters of up to 61% on One Billion Word dataset [1]. Additionally, dropout is also added after the projections in the adaptive softmax layer to improve regularization.

Although the model used was not an LSTM, but a transformer-based architecture, adaptive input representations outperforms adaptive softmax, from both accuracy and training speed perspectives. It is used in Transformer-XL [7], which became state of the art on multiple NLP tasks and a baseline for newer publications that came with improvements, such as XLNet [42].

Chapter 3

Methodology

3.1 Datasets

For datasets with a small vocabulary size, the full softmax is expected to perform better than any approximation method. That is, given a limited budget of training time, softmax should reach the lowest overall perplexity. Other methods might have faster epochs, but their capacity to learn the language model is limited by design.

For this reason, this project will evaluate the hierarchical alternatives to softmax on a relevant spread of dataset sizes. By doing this, we aim to find an approximate size at which softmax should not be used anymore because of its memory footprint and slowness.

Penn Treebank [23]. Penn Treebank (PTB) is a dataset of very high importance in NLP, historically used as a benchmark for part of speech tagging. The latest iteration [24] contains a total of one million words and its 25 sections have a standard way of being split in train/validation/test: 21/2/2 [29]. Although its vocabulary size of ten thousand words is small enough to not require softmax approximation, the dataset is still relevant up to this day in language modeling.

Wikitext [28]. A subset of high quality Wikipedia articles was selected and minimal preprocessing was applied. The punctuation, numbers and the case were kept, while PTB eliminates these features [28]. There are two such datasets, called Wikitext-2 (WT-2) and Wikitext-103 (WT-103). The numbers are referring to the millions of word tokens in the training sets. Interestingly, the validation and test sets are the same for these two datasets, making comparison straightforward.

Europarl [21]. Proceedings from the European Parliament were used to create a parallel corpus mainly aimed for machine translation. Although the dataset features

11 languages, we use the English text for consistency. As there is no standard way to divide the English subset for language modeling, we create the data splits ourselves: we tokenize the text, replace the rare words with <UNK>, shuffle the lines and use 1% of the lines as validation and test sets, equally. Although the dataset is richer in vocabulary compared to WT-2, the same preprocessing leads to only 0.15% out-of-vocabulary words in the test set. This observation, combined with the previously reported results on other Europarl languages [12], suggests that the predictability of this dataset is much higher.

One Billion Word [5]. Similarly to WT-2 and WT-103, One Billion Word Benchmark (1B) is specifically designed for language modeling. The source of its name is the number of word tokens after removing duplicate sentences, normalizing and tokenizing [5]. Currently, it is one of the most popular datasets for evaluating state of the art language models and together with WT-103 is very relevant for softmax alternatives benchmarks. Due to restricted computational resources, we decide not to include 1B in our benchmark, but we mention it to show the scale of modern datasets.

Dataset	Tokens			Vocabulary size	OoV
	train	validation	test		
PTB	887k	70k	79k	10k	4.8%
WT-2	2M	218k	245k	33k	2.6%
Europarl (en)	55M	283k	282k	63k	0.15%
WT-103	103M	218k	245k	268k	0.4%
1B	821M	8M	160k	793k	0.28%

Table 3.1: Dataset sizes and splits in terms of number of tokens and vocabulary sizes. OoV refers to the percentage of words out-of-vocabulary (replaced with <UNK>) in the test set. Information gathered from [28, 5].

3.2 Model

Creating an accurate benchmark for the hierarchical methods requires having a powerful baseline model that is complex enough, yet properly regularized. Although recent studies have found that 1-layered vanilla LSTMs can obtain better perplexities than much more complex models when properly tuned [25], we chose the AWD-LSTM as a baseline for this project. The model is robust, well documented and does not add

much computational overhead when compared to the vanilla LSTM implemented in modern deep learning toolkits.

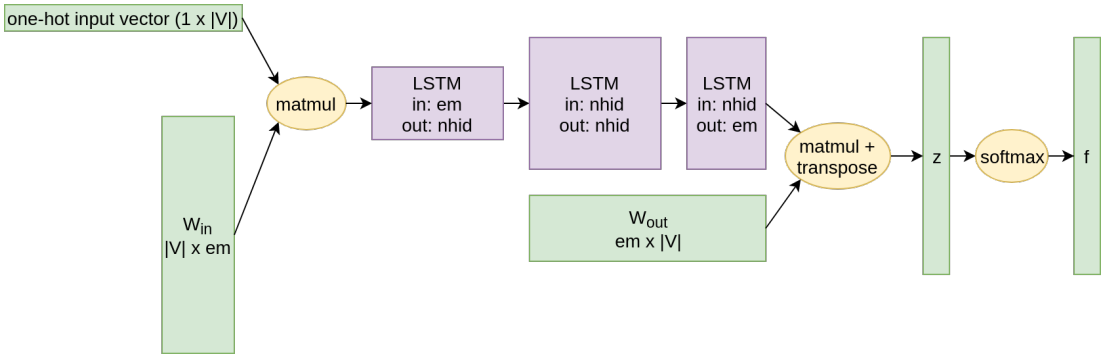

Figure 3.1: Structure of neural network used in this project. For the simplicity of the figure, we illustrate only one input in a forward pass (batch size ignored) using full softmax. em denotes the embedding size and $nhid$ the number of hidden units. The first matrix multiplication operation has the role of selecting the the correct row from the input embedding matrix W_{in}, which is then passed through the 3-layered LSTM. The last hidden state is multiplied with the output embedding matrix W_{out}, which is then normalized in order to create the probability distribution f. As W_{in} and W_{out} are tied, W_{out} only has a symbolic meaning, as it is not used in order to reduce memory usage. Finally, for the hierarchical softmax, W_{in} and W_{out} have different structures and additional projection matrices are used.

This decision was made after initially trying a vanilla LSTM and observing that the amount of overfitting prevented us from drawing conclusions about the performance of the hierarchical methods. Additionally, we tried two flavors of the AWD-LSTM, small (2 layers, 300 embedding size, 600 hidden layer size) and large (3 layers, 400 embedding size, 1150 hidden layer size). The large one is inspired by the parameters used by the authors to train on WT-2 [26]. We realized that the large model had good performance on all datasets (even though we expected it to overfit on PTB) while training in an acceptable amount of time. The smaller one, however, had worse performance on WT-2 and larger datasets. We therefore made the decision to use the large flavor in all of our experiments. The full list of hyperparameters used during training can be found in section 4.1.

We made a few modifications to AWD-LSTMs formulation. Firstly, we did not use ASGD, but Adam [19] instead (more details in section 4.2). This is motivated by the fact that our models are not required to reach the competitive performance reported

in [26, 27], but train well enough in a reasonable amount of time. Adam is known to converge faster than SGD and require less tuning than SGD, which can reach lower minimas when properly tuned.

We also implemented our own version of the tied adaptive softmax in order to structure the project better. More information can be found in section 2.2.2.

3.3 Hierarchical benchmark

In this section we will describe how the softmax approximations will be evaluated. Our goal is to compare softmax to its alternatives, but because of the many ways the tree hierarchy can be constructed in, we choose to evaluate only 2-layered trees. Moreover, as we have no information about the ideal number of clusters and adaptive methods suggest that 2 to 5 work the best [12, 1, 7], we choose to split our words in 3 clusters. While optimizing the number of clusters would make the overall comparison more robust, the experiments are generally very time consuming. In conclusion, our analysis will be based on 2-layered hierarchical softmax method comprising on 3 word clusters obtained through various methods.

While we keep most parameters intact between models, we require careful tuning of the batch size and learning rate for each experiment on each dataset. The motivation for this comes from two observations made in our preliminary experiments. Firstly, given the same dataset, hierarchical methods will have a lower number of parameters compared to full softmax, suggesting that the learning rate might need a different value. Secondly, fixing the batch size across experiments and only tuning the learning rate limits the capacity of hierarchical methods to learn accurate language models. As a solution for this, we choose the largest batch size possible (power of 2) for each experiment, then tune the learning rate. We consider this to be key in our analysis in order to have a fair comparison. The exact hyperparameter settings can be found in section 4.1.

In the following paragraphs we describe the training procedure and the clustering algorithms applied for each normalization layer.

Full softmax (SM). As this is an exact method, clusters will not be involved in training. When given infinite training resources and time, we expect softmax to reach the lowest validation perplexity. However, having limited resources, our goal is to reach the minimum in a limited amount of time. Furthermore, training large models on full softmax is sometimes not possible due to the size of the embedding matrix W_{in}

[1] and the GPU memory.

Hierarchical (non-adaptive) softmax (HSM). For the non-adaptive hierarchical methods, we conduct two kinds of experiments: clustering according to frequency and according to meaning, similarly to [6]. For the meaning-based clustering, we require information about the word embeddings present in the datasets.

We obtained the pre-trained word embeddings from two sources. The first source is GloVe [35], a library that provides word embeddings trained on Wikipedia and Gigaword by modeling word co-occurrences. Its advantage is that it does not require pre-training a language model as the weights can simply be accessed via an API. On the other hand, the domains of the data might be different which can affect performance. Some words in the dataset that we are aiming to train might not be in GloVe, while other words might have different meaning according to the context.

The second kind uses the weights obtained after the initial softmax experiment ends. We expect these embeddings to be more useful (lead to lower perplexities) than the GloVe ones as they are sharing the same domain. The significant disadvantage is that we need to pretrain a softmax model to get access to the word representations, which we are trying to avoid by using the softmax approximation methods.

Algorithm 1 Weighted k-means clustering pseudocode. x is the training set composed of m points of size n, k is the number of clusters and w is the set of weights (one per training point). For normal k-means, w is set to an array of ones. The notation was inspired from [36].

procedure K-MEANS(x, k, w)
    Randomly initialize the cluster centroids μ_i, $i \in \{1, 2, \ldots, k\}$
    while not converged **do**
        for $i \in \{1, 2, \ldots, m\}$ **do**
            $c_i = \text{argmin}_j \|x_i - \mu_j\|^2$ ▷ ith input's label
        for $j \in \{1, 2, \ldots, k\}$ **do**
            $\mu_j = \frac{\sum_{i=1}^{m} w_i x_i}{\sum_{i=1}^{m} w_i}$ ▷ for all i s.t. $c_i = j$
    return c, μ

After obtaining the word embeddings, the clustering is done through three separate unsupervised models: k-means, weighted k-means and Gaussian mixture models (GMM). The first two were also used in [6], but the weighing was done by word counts, which we believe creates an inaccurate representation of their actual meaning. Instead,

we weigh them by the log of the word counts, such that the weights have a linear relation according to Zipf's law.

Lastly, we believe that k-means may lead to bad clustering due to curse of dimensionality: Euclidean distances between word embeddings become less useful and intuitive as the dimension of the hyperspace increases [9]. We decide to try a Gaussian mixture model to deal with this issue, as k-means is biased towards finding spherical clusters. Similarly to k-means, the solution is found through expectation-maximization, but the cluster assignment is soft. For more details about both algorithms, we refer the reader to [3].

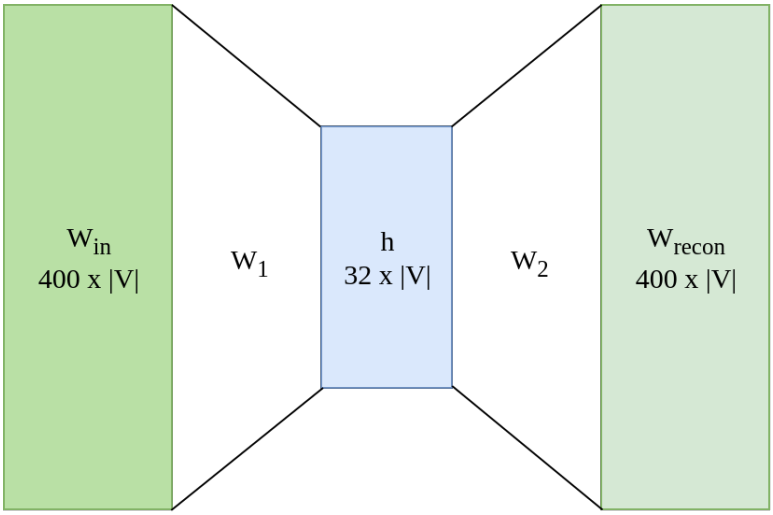

Figure 3.2: Autoencoder network used for clustering word embeddings. The input embeddings are projected through W_1 into the bottleneck h, which then projects the data through W_2 to obtain the reconstructed embeddings. The reconstruction loss for each sample is calculated as the L_2 norm between the input and reconstructed embeddings. Then the words are grouped together in an adaptive split, according to their reconstruction loss.

Adaptive softmax. We use these experiments to compare tied adaptive softmax (**ASM**) with adaptive input representations (**AIR**). While adaptive inputs can be used standalone with a normal softmax output, we use them only in combination with adaptive softmax such that we get the benefits of weight tying. AIR has fewer parameters and is believed to improve accuracy on rare words. Similarly to the non-adaptive methods, we use frequency-based clustering and meaning-based clustering. As classic unsupervised methods do not provide information about the importance of embedding size inside the cluster, they are not suitable for this category of experiments.

We propose a novel way of doing meaning-based clustering that offers additional information about how easy or difficult to predict the words are. Given pretrained word embeddings, we train an autoencoder (see figure 3.2) and group the words according to their reconstruction loss. Intuitively, if an autoencoder is easily able to reproduce a word's embedding, it will not need a relatively small embedding size inside the adaptive method. If the opposite happens, that word will be assigned to a cluster with a larger embedding size. As with the non-adaptive methods, this has the disadvantage of needing good embeddings (expert knowledge) before training the model, either from a previous experiment or online sources.

3.4 Evaluation

When comparing the performance of our methods, we are most interested in final validation and test perplexities, as well as time to reach convergence (if it is reached at all). When the batch size and learning rate are tuned accordingly (see section 3.3), we measure the experiment speed-up of the other methods compared to full softmax.

Additionally, speed-up per epoch is relevant and used in previous benchmarks, but in order to create this comparison we need to set the same batch size for each dataset (maximum batch size used by full softmax). However, this metric can be deceitful as great speed-ups can be achieved at the price of high perplexities. We also measure the GPU memory usage using the nvidia-smi[1] toolkit when using the same batch sizes.

Finally, we also analyze the number of parameters of each class of experiments, as their size is very important to make them usable on low-resource devices. The classes of experiments are full softmax (SM), hierarchical non-adaptive softmax (HSM), tied adaptive softmax (ASM) and adaptive input representations (AIR).

[1] `https://developer.nvidia.com/nvidia-system-management-interface`

Chapter 4

Experiments

4.1 Setup

This section presents the conditions under which the experiments were ran and motivation behind them.

Software. This project is developed in PyTorch 1.0 and builds on the official Salesforce's AWD-LSTM repository[1]. PyTorch provides a fast and well documented implementation of the adaptive softmax layer, which we modified into supporting hierarchical non-adaptive softmax as well by removing the shortlist and setting the division value to 1. For the adaptive input representations, we used Facebook AI's Fairseq[2], a repository rich in transformer-based training modules.

Hardware. We used two systems: a desktop computer with a GTX 1070 (8GB of VRAM) and a Google Cloud instance with a P100 GPU (16GB of VRAM). Ideally, all the experiments should be run on the same system, but the free tier Google instance only allows about 200 hours of training, while the RAM on the desktop GPU requires lowering the batch size until experiments become too slow to train (e.g. 20h/epoch on WT-103). We therefore used the desktop for training WT-2 and PTB and the Google Cloud instance for WT-103 and Europarl.

Hyperparameters. Firstly, we preserve the model proposed in the AWD-LSTM repository for training Wikitext-2. The model is composed of 3 layers of different sizes where the first and the third are sharing the same weights. Then, depending on which softmax is used, a decoder (linear layer) between the last layer and the outputs space is used. This is done to be able to have an embedding size different from the hidden

[1] `https://github.com/salesforce/awd-lstm-lm`

[2] `https://github.com/pytorch/fairseq/`

layer size, as explained in [26]. We also keep the dropout values suggested by the same model, although tuning them for each dataset might be advantageous, yet costly in terms of training time. The only dropout-related difference is that for adaptive input representations we no longer use the embedding dropout, but the adaptive dropout proposed in [1]. See section 2.2.2 for more details.

The optimization is done with Adam with weight decay for an unlimited number of epochs. The learning rate is linearly scaled from a base value for each batch given the variable BPTT length [26]. The base value is divided by a factor of 10 when there has been no decrease in perplexity for the last few epochs. When this division has been done for a fixed number of times, the experiment stops. Alternatively, the experiment stops when a time limit is reached. As explained in 3.3 we choose the maximum batch size that is a power of two for each experiment, then find a learning rate experimentally. We do not necessarily choose the learning rate that ends up with the smallest validation loss if choosing a higher learning rate only increases the perplexity by a few points, while training is significantly faster.

Hyperparameter	Value	Hyperparameter	Value
embedding size	400	embedding dropout	0.1
hidden units	1150	hidden dropout	0.2
LSTM layers	3	input dropout	0.65
gradient clipping	0.25	output dropout	0.4
BPTT steps	70	adaptive dropout	0.2
patience	3	weight drop	0.5
tied weights	yes	alpha (AR)	1
		beta (TAR)	2
		weight decay	1.2e-6

Table 4.1: List of general hyperparameters used across all experiments. The right side of the table contains only hyperparameters related to regularization to emphasize its importance.

We use a division value $div = 3$ for adaptive input representations compared to 4 used in the literature [1] as our initial embedding size will be smaller than the one of state of the art models. As for defining the cut-offs, we divide the words (sorted by occurrences) in such way that the product of the number of words in a cluster and the cluster's embedding size is constant.

The autoencoder used for clustering by reconstruction loss contains only one hidden layer of size 32 and is optimized using mean squared error.

		PTB	WT-2	Europarl-en	WT-103
division value (*div*)		3	3	3	3
cut-offs		[1k, 4k]	[2.5k, 10k]	[12k, 48k]	[20k, 80k]
GPU		GTX1070	GTX1070	P100	P100
time limit (h)		0.75	1.5	3	6
softmax	batch size	128	64	64	16
	learning rate	0.003	0.003	0.001	0.001
adaptive	batch size	256	256	512	128
	learning rate	0.003	0.003	0.003	0.003
hierarchical	batch size	256	128	64	-
	learning rate	0.003	0.003	0.001	-

Table 4.2: List of dataset- and hardware-specific hyperparameters used. We found that a learning rate of 0.003 works the best for experiments with batch size of at least 128 and we set 0.001 for the smaller batch sizes. Although this table only includes the final list, multiple values were tried during early experiments sampled from the $[0.0001, 0.01]$ interval.

4.2 Results

4.2.1 Penn Treebank

As PTB is the smallest dataset, we expect full softmax to outperform the approximate methods. Indeed, we notice that softmax obtained significantly lower validation perplexity in a 45 minute experiment. Tied adaptive softmax is the closest, achieving about 7.7 points in test perplexity more than softmax 45% faster, followed by the pretrained hierarchical methods. From an epoch time perspective, all hierarchical methods take about 26s, while softmax takes 32s. As for the total experiment time, we get on average 1.7x speed-up, which is very significant. However, the models learn better language representations during a softmax epoch, therefore choosing approximate methods for this speed-up is not justified.

Surprisingly, adaptive input representations perform the worst (more than 80 per-

plexity), which suggests that modifying the input embedding on such a small dataset hurts the performance significantly. As for the clusters obtained using an autoencoder, performance slightly decreases in the case of tied adaptive softmax, while for adaptive input 10 points of perplexity are lost and the experiment is shorter, suggesting overfitting.

experiment	valid ppl	test ppl	epoch time (s)	total time (s)
SM	**69.41**	**67.21**	31.9	2651
HSM, weighted k-means, pretrained	78.26	75.25	25.5	1506
HSM, weighted k-means, GloVe	83.93	80.69	25.9	1139
HSM, k-means, pretrained	79.04	75.06	25.8	1598
HSM, k-means, GloVe	83.93	80.75	25.8	**1108**
HSM, GMM, pretrained	80.64	77.10	25.7	1438
HSM, GMM, GloVe	80.64	77.89	26.2	1417
HSM, frequency	82.27	79.57	25.7	1542
ASM, autoencoder	79.04	76.44	25.8	1650
ASM	78.26	74.91	**24.5**	1447
AIR, autoencoder	91.84	87.15	25.3	1541
AIR	84.77	80.47	24.6	1597

Table 4.3: Results of experiments on Penn Treebank.

The pretrained embeddings lead to better perplexities, which is expected due to them being obtained from the same dataset, trained until convergence using softmax. GloVe embeddings are about 5 points of perplexity worse k-means methods and close to pretrained embeddings when using GMM.

4.2.2 Wikitext-2

Although WT-2 is more than three times larger than PTB in terms of vocabulary size, it is not considered a large dataset and not analyzed in other benchmarks. We therefore expect hierarchical methods to not be a good replacement for softmax for this dataset either, although we expect better results than those on PTB.

Indeed, the test perplexity gap between softmax and its best replacement – tied adaptive softmax – is only 4.6 points and obtained with an experiment speed-up of 1.43x. Once again, weighted k-means on pretrained embeddings performs the best out

of all the non-adaptive methods. For k-means and Gaussian mixture models, the models using GloVe learn slightly better than the pretrained alternatives, which is expected due to the similar domains of the embeddings.

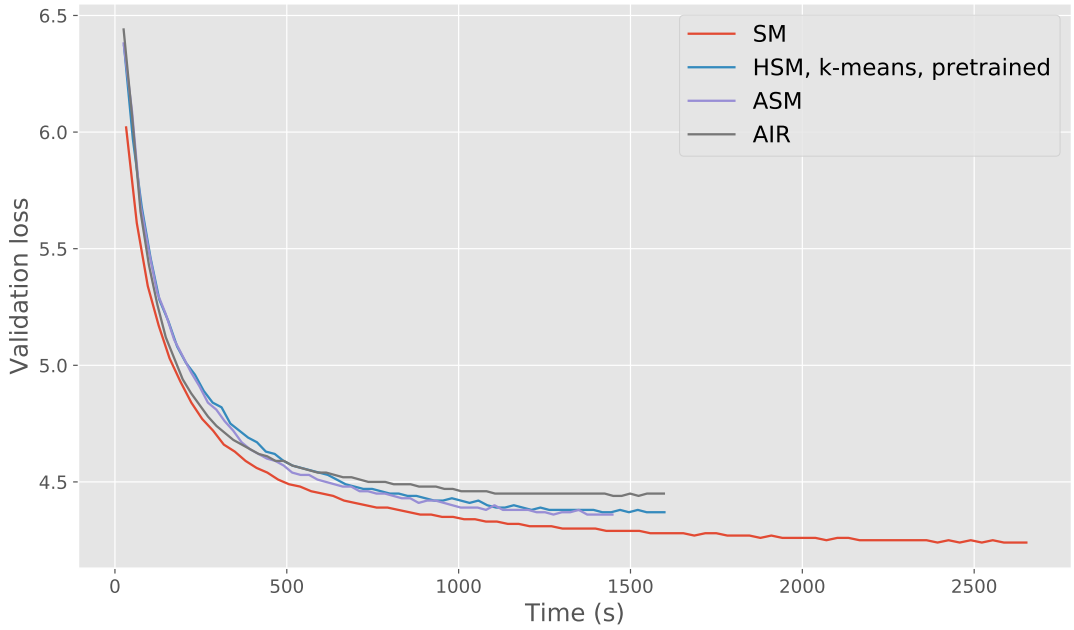

Figure 4.1: Learning curves of the best experiments in each class of softmax for Penn Treebank. Loss is used instead of perplexity to emphasize the differences.

experiment	valid ppl	test ppl	epoch time (s)	total time (s)
SM	**82.27**	**78.33**	119.3	5366
HSM, weighted k-means, pretrained	90.92	85.68	75.2	4209
HSM, weighted k-means, GloVe	94.63	89.92	75.7	3253
HSM, k-means, pretrained	92.76	88.05	75.1	3678
HSM, k-means, GloVe	90.92	87.23	76.9	4611
HSM, GMM, pretrained	94.63	89.27	76.2	3430
HSM, GMM, GloVe	90.92	86.02	79.3	4676
HSM, frequency	90.92	87.30	76.4	5425
ASM, autoencoder	92.76	87.26	65.6	**2888**
ASM	87.36	82.99	64.3	3793
AIR, autoencoder	97.51	91.41	**61.7**	3331
AIR	92.76	87.79	62.1	3103

Table 4.4: Results of experiments on Wikitext-2.

The pattern from the PTB results with respect to adaptive methods repeats: using the autoencoder decreases overall performance and adaptive input representations still does not achieve comparable results to tied adaptive softmax. We notice however that time-wise, epoch times for adaptive input representations are 6% shorter, which can be explained by the number of parameters being reduced in the input layer. When considering that the perplexity also increases by about 5%, we conclude that adaptive input representations are not suitable for WT-2 either.

Finally, we notice that adaptive methods lead to lower epoch times than non-adaptive methods, by 16–28%. This difference demonstrates the importance of using a shortlist on the first layer of the tree when training on a GPU.

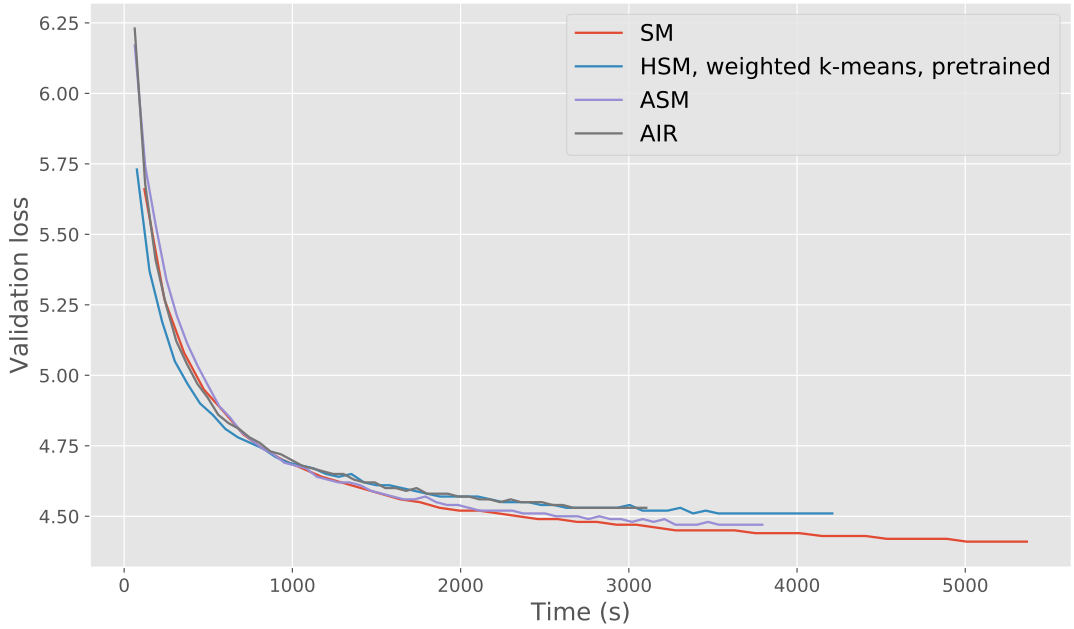

Figure 4.2: Learning curves of the best experiments in each class of softmax for Wikitext-2. Loss is used instead of perplexity to emphasize the differences.

4.2.3 Europarl-en

Given the results from the previous datasets, we decided to experiment with HSM in only two configurations: using Gaussian mixture models on GloVe embeddings and using weighted k-means on pretrained embeddings.

Although this dataset would not be considered large scale in the literature, it is in between the sizes of WT-2 and WT-103. This means that we are not training to convergence anymore, therefore time per experiment becomes irrelevant, as the experiments

are limited to 3 hours.

experiment	valid ppl	test ppl	epoch time (s)	total time (s)
SM	41.26	41.58	3092.7	12370
HSM, weighted k-means, pretrained	42.10	42.31	1772.4	12406
HSM, GMM, GloVe	43.82	44.13	2239.0	11195
ASM, autoencoder	38.86	38.99	1096.1	10961
ASM	38.47	38.79	1092.6	10925
AIR, autoencoder	37.71	37.62	1085.6	10855
AIR	**37.34**	**37.59**	**1084.6**	**10846**

Table 4.5: Results of experiments on Europarl-en.

We first notice that the perplexities are much lower than for the other datasets, suggesting that the language of Europarl-en is much more predictable. Adaptive methods clearly perform better, with adaptive input representations getting about 4 points lower than full softmax in test perplexity and epochs being about 3 times faster. Tied adaptive softmax performs similarly, only slightly behind AIR. Interestingly, the autoencoder is less than 0.2 perplexity higher than its frequency-based clustering counterpart, suggesting this method is viable for Europarl.

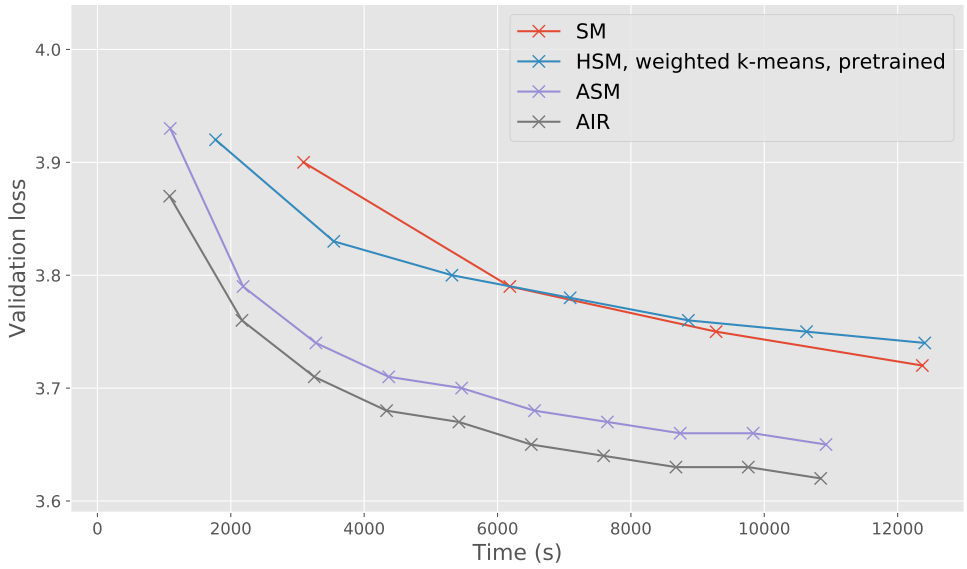

Figure 4.3: Learning curves of the best experiments in each class of softmax for Europarl-en. Loss is used instead of perplexity to emphasize the differences.

On the other end, while there exists a speed-up per epoch, HSM methods are consistently outperformed by adaptive methods. Finally, as the domains of GloVe and Europarl are different, we notice that the pretrained embeddings perform better than GloVe.

4.2.4 Wikitext-103

experiment	valid ppl	test ppl	epoch time (s)	total time (s)
SM	115.58	115.87	21557.7	21557
ASM, autoencoder	99.48	101.47	3971.6	23829
ASM	**98.49**	**100.58**	3925.1	23550
AIR, autoencoder	103.54	104.68	3274.5	22921
AIR	102.51	104.05	**3258.4**	**22808**

Table 4.6: Results of experiments on Wikitext-103.

WT-103 is one of the most relevant datasets when evaluating approximate softmax losses. Its large number of unique words (over 200k) restricts training with a full softmax by forcing lower batch sizes, which makes epochs very slow. In our experiments, we lowered the batch size to 16 on the system with a 16GB RAM GPU and an epoch using full softmax took about 6 hours. Although training the model more is possible, we set the upper limit of training time at the 6 hours mark then experimented with the adaptive models. Additionally, we do not run HSM configurations, as the results on previous datasets suggest that it lacks scalability.

As adaptive methods are designed to use less memory, this allowed us to train with batch sizes of 128, 8x more than with full softmax. We found that tied adaptive softmax lowers the perplexity the most once again: the test perplexity is lower than softmax by 15.3 points. Adaptive input representations are very close (about 4 points more than adaptive softmax). The fact that the training time per epoch for the adaptive inputs is 17% lower while only having a 3.4% perplexity increase suggests that the method could be better than adaptive softmax when the dataset size increases. Moreover, we notice that autoencoder methods are are less than 1 point in perplexity away from their frequency-based counterparts, further suggesting that these methods scale better with larger vocabularies.

Comparing the adaptive methods with the full softmax, we conclude that at this

point the latter is not suitable anymore and that adaptive methods are clearly superior.

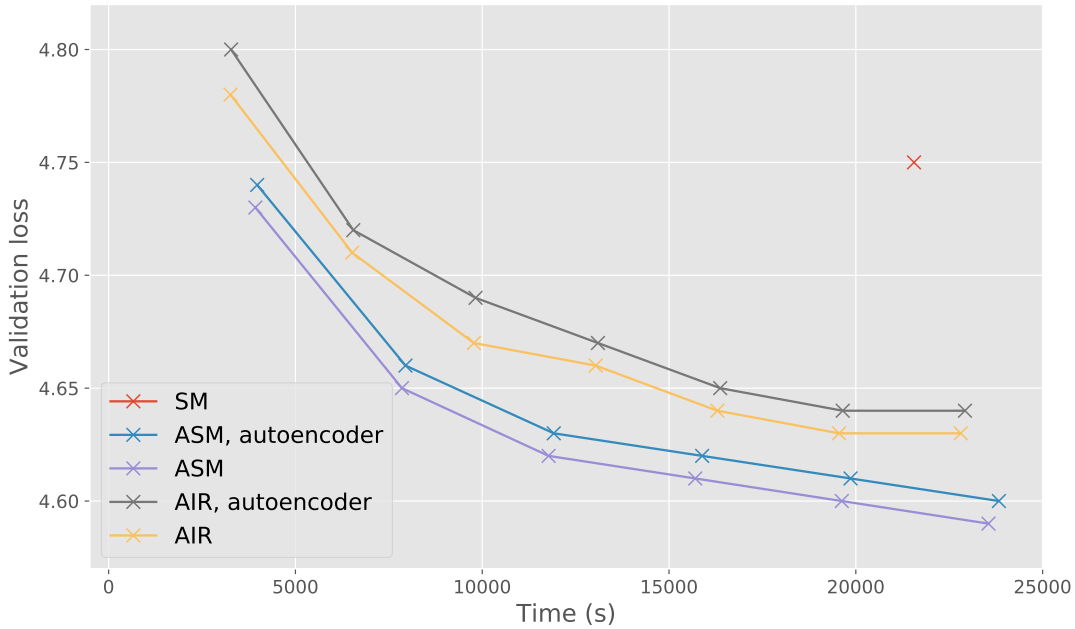

Figure 4.4: Learning curves of the all experiments for Wikitext-103. Loss is used instead of perplexity to emphasize the differences.

Chapter 5

Discussion

We evaluated full softmax, hierarchical non-adaptive softmax and adaptive methods on four datasets of various sizes. In this chapter, we discuss the result of the experiments through several criterias.

Softmax alternatives. Throughout this benchmark, we demonstrated the importance of hierarchical softmax alternatives when using 3 clusters. For smaller datasets, hierarchical non-adaptive softmax using k-means clustering that is weighed by the log of the word counts provides the best results, although full softmax is still preferred. This is motivated by the fact that a vocabulary size of 10,000 leads to matrix operations that fit current GPUs even with large batch size (more than 128 samples), therefore using an approximation hurts the performance while not improving the training time. However, this is the only scenario where we find that hierarchical non-adaptive methods are suitable at all.

Adaptive input representations worked best with Europarl, suggesting that they reduced overfitting drastically when compared to tied adaptive softmax. We explain this phenomenon by the simplicity of the language used in the dataset, which led to low overall perplexity in all experiments.

On WT-103, as expected from the literature [1, 12], adaptive methods are superior to softmax due to being able to improve memory usage by associating rare words to lower embedding sizes. This suggests that on even larger datasets, such as One Billion Word, adaptive methods are the best choice, as non-adaptive methods do not scale and full softmax is not even possible to train in a limited amount of time (less than a day). Tied adaptive softmax is unexpectedly better than adaptive input representations, a model which was tested only on very powerful and large-scale transformer-based neural networks.

We also observed that WT-2 is not large enough to be suitable for hierarchical methods, therefore we argue that for similar datasets that are governed by Zipf's law and given the current hardware, the threshold for using adaptive softmax instead of full softmax is around 50,000 words.

Non-adaptive clustering algorithms. Although weighted k-means performed the overall best, the differences between methods were small. However, weighing the word embeddings by the logarithm of their word count is a novel idea and is as efficient to run as normal k-means. On the other hand, while Gaussian mixture models have a bigger runtime, their performance is close to k-means when also clustering on GloVe embeddings.

Adaptive clustering algorithms. We compared the adaptive alternatives by using the same cluster sizes, but different criterias: by frequency and by reconstruction loss when using an autoencoder. Using frequencies, the perplexity was always smaller, which suggests that our autoencoder idea was inferior. However, we do not discard the idea itself as performance was very close on WT-103 and Europarl. We think of two reasons of why this happens.

Firstly, the autoencoder itself did not model the word embedding space well as we used a very simple network and no regularization or activations. Using more complex models that are tuned for this tasks might provide better performance. Secondly, the embeddings we used as input to the autoencoder might have not been of sufficient quality to be better than the frequency-based methods, but further training and tuning of our softmax baselines could have solved this problem. We especially argue that this could be the case for WT-103 where softmax was trained for only 1 epoch and did not reach a very low perplexity, yet the input embeddings obtained were used in the autoencoder.

Pretrained embeddings. GloVe embeddings rarely led to better clustering than the ones from pretrained softmax models, especially on PTB where the difference in perplexity was up to 7 points. The results on Wikitext-2 show that GloVe and encoder embeddings are closer in performance with sometimes GloVe being better. We argue that the reason for this is that GloVe embeddings were trained on data that included Wikipedia articles, while PTB has news articles and the vocabulary is different from that used on Wikipedia. Being able to use GloVe embeddings is very relevant as for larger datasets having encoder embeddings is not feasible.

We conclude that if training full softmax models for few epochs to get accurate embeddings is possible, they should be preferred instead of GloVe. Otherwise, externally

obtained embeddings such as GloVe or fastText [4, 16] can be used, especially if the domains are similar.

Memory requirements. In order to have an overview over the actual memory usage of each of the methods on each dataset, we created a new set of pseudo-experiments – we do not train most of the models fully, but only probe certain metrics while keeping a constant batch size for a fair comparison. This analysis is relevant from a low-resource hardware point of view and therefore was not included in the experiments section.

We first count the number of parameters in a method similar to the one in the implementation of AWD-LSTM. We find out that HSM and ASM slightly increase the number due to the projection matrices used. Moreover, this gap increases with the size of the dataset, with tied adaptive softmax having as much as 6.2% more parameters for WT-103. Using adaptive inputs, however, drastically decreases the number of parameters, which coincides with the results reported in [1]. We report reductions of 38.3% and 58.7% for Europarl and WT-103, respectively. For larger datasets, we expect this reduction to be even larger.

Dataset	Metric	SM	HSM	ASM	AIR
PTB (bs=128)	epoch time (s)	31.9	28.7	27.2	27.6
	memory (MB)	6221	4359	4283	4271
	parameters	24.2M	24.7M	24.9M	21.9M
WT-2 (bs=64)	epoch time (s)	115.4	90.0	80.5	73.2
	memory (MB)	6013	3593	2975	2749
	parameters	33.6M	34M	33.8M	24.5M
Europarl (bs=64)	epoch time (s)	3097	1770*	1668*	1548*
	memory (MB)	12513	4077	4027	3515
	parameters	45.5M	45.9M	47.7M	28.1M
WT-103 (bs=16)	epoch time (s)	21558	-	12524*	8150*
	memory (MB)	14109	-	5761	4701
	parameters	127.6M	-	135.6M	52.7M

Table 5.1: Hardware-related metrics for each model and dataset. The asterisk denotes results that are approximated through the duration of first minibatches.

Secondly, we measure the GPU memory usage during training using nvidia-smi.

As expected, for all the datasets the following trend is found: SM>HSM>ASM>AIR. Interestingly, the memory usage decreases for HSM and ASM even though the number of parameters is slightly larger. We reason that this happens because the matrix operations for HSM and ASM have lower dimensions, but the parameters are about the same. The memory usage for AIR ends up being more than 3.5x lower than softmax and ASM more than 3x lower for the larger datasets.

Finally, we analyze the epoch times for each experiment. This is different than the results reported in the experiments section as the batch size varied. AIR is the fastest, as expected, being 2x faster on Europarl and 2.5x faster on WT-103. Once again, we notice the trend: SM>HSM>ASM>AIR in terms of training time per epoch and the larger the dataset is, the bigger the speed-up will be when compared to full softmax.

We conclude that the adaptive methods provide excellent scalability and are very suitable for running on low-end devices due to their low memory usage and fast training time, which suggest fast prediction time as well.

Hyperparameters. The chosen AWD-LSTM architecture proved to be powerful enough for this benchmark. The batch size and learning rate being optimized using the validation perplexity and convergence time provide extra robustness for this set of experiments. However, we argue that the five types of dropout should also be optimized accordingly, especially as the number of parameters is much smaller for hierarchical methods. Instead, we used the same set of dropout ratios across all experiments, suggested by the work of [26]. For example, adaptive input representations are thought to help model rare words [1] and the authors suggest an adaptive dropout of 0.2, but this value might not be suitable for our model and datasets. Intuitively, it would be interesting to try more regularization (larger dropout values) for the small datasets and less regularization (smaller dropout values) for the larger datasets.

The other very relevant hyperparameter is the number of clusters used. In our early development phase, we tried 3 and 4 clusters setups as previous works suggested. While we believe this is a reasonable setup for adaptive methods, non-adaptive methods might not benefit from such a low number of clusters. We also tried using $log(|V|)$ classes initially and the epochs were significantly faster, but the models were very inaccurate. We think it would be interesting to try using up to 10 clusters with both weighted k-means and Gaussian mixture models. Although we might have obtained better models, we argue that finding the right number of clusters using a grid search eliminates the usefulness of hierarchical methods.

Finally, we chose the adaptive softmax parameters such that the matrix multiplica-

tions for each cluster are about the same in terms of complexity. In previous benchmarks, the input embedding size was 1024 and the division value was 4, meaning that the clusters had embeddings sizes of 1024, 256, 64 respectively. In our experiments, we used an input embedding size of 400 with a division value of 3 due to model complexity reasons. Our clusters had embedding sizes of 400, 133, 44 respectively, which are lower capacity than the ones in the literature. Altogether, it is possible that increasing the initial embedding size improves the perplexity of adaptive models significantly.

Chapter 6

Conclusions and future work

In this project, we described and created a benchmark of hierarchical methods to approximate the output of softmax layer in language modeling. As the size of modern datasets is continuously growing, having good approximations can make training and inference possible on low-end hardware, such as IoT devices. Our analysis is spread across datasets with vocabulary sizes of 10k, 30k, 60k and 260k and we set the threshold for replacing softmax at about 50k. We reason that applying hierarchical methods on datasets smaller than this value leads to a perplexity increase that does not justify the speed-up.

When comparing the methods themselves, we show that non-adaptive hierarchical softmax is considerably inferior to full softmax and adaptive softmax. We tried one frequency-based and six meaning-based clustering criteria and non-adaptive hierarchical softmax performed comparably to full softmax only on the small datasets, which suggests low scalability. We showed, however, that clustering using the input embedding layer from a previous experiment works better than when using online embeddings (GloVe). We also described two novel methods for clustering: using weighted k-means with frequency log probabilities and Gaussian mixture models. The former performed consistently better when compared to the other non-adaptive methods.

Adaptive softmax and adaptive input representations are good methods designed to utilize the power of modern GPUs that can do matrix multiplications where the sizes are on the scale of a few thousands very efficiently. In our experiments on Europarl-en and WT-103 and with a limited budget of time, we show that these methods lead to up to 9.6% and 13.2% test perplexity reductions, respectively. For datasets of WT-103's size and larger, training a full softmax model represents a challenge, as the batch size has to be drastically reduced in order for the model to fit in memory while training.

In section 5 we set equal batch sizes and measure the actual number of parameters, training speed per epoch and GPU memory usage. As expected, adaptive input representations offer the greatest scalability, with reductions in the total number of model parameters of up to 38.3% for Europarl and 58.7% for WT-103. Moreover, they outperform the other methods on Europarl from an accuracy perspective as well. We obtain the largest speed-up when using adaptive input representations as well: 2x and 2.64x for Europarl and WT-103 compared to 1.85x and 1.72x for tied adaptive softmax.

Finally, as the gap between softmax and adaptive methods becomes larger as the vocabulary size increases, we investigate using adaptive methods with a novel clustering criterion. Instead of grouping words based on their frequency, we train an autoencoder over pretrained embeddings and cluster the words based on their reconstruction loss. We found that for the smaller datasets, this method hurts performance significantly, sometimes making it worse than non-adaptive methods. For Europarl and WT-103, however, we get very close performance to the frequency-based experiments (less than one perplexity point higher). This result suggests that the method could be beneficial when the autoencoder is tuned properly and the pretrained embeddings are of high quality.

As for future work, our benchmark has a few shortcomings that could make this analysis more robust if resolved. They are caused by the limited training resources available (both time and hardware) and the fact that we get relevant results only for large experiments, which always take longer to train. First off, we explained the need and difficulty of regularizing recurrent neural networks, as our project requires tuning of 9 different regularization-related hyperparameters. We chose a specific model size and set of dropout values based on [26, 27], but these values should be tuned for each dataset and approximation layer. Moreover, to make comparison uniform, we restricted hierarchical non-adaptive methods to 3 clusters, while the literature offers more elegant ways of structuring the tree. We argue, however, that although the non-adaptive methods can be improved, they will always be inferior to adaptive methods, as the latter are specifically designed to be fast on GPUs. It would also be interesting to see results on One Billion Word and using a tuned autoencoder (such as the variational autoencoder [20]) with better embeddings.

In conclusion, we showed that adaptive softmax methods can not only significantly reduce the GPU memory usage, allowing larger batch sizes and overall faster training, but also lead to better perplexities than the full softmax in a restricted time frame. We believe that adaptive softmax will continue to be state of the art when dealing

with large output spaces and could be used in other fields of machine learning as well beyond NLP, such as recommender systems.

Bibliography

[1] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[3] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[5] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[6] Welin Chen, David Grangier, and Michael Auli. Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906*, 2015.

[7] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[8] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1370–1380, 2014.

[9] Pedro M Domingos. A few useful things to know about machine learning. *Commun. acm*, 55(10):78–87, 2012.

[10] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

[11] Joshua T Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[12] Edouard Grave, Armand Joulin, Moustapha Cissé, Hervé Jégou, et al. Efficient softmax approximation for gpus. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1302–1310. JMLR. org, 2017.

[13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*, 2016.

[16] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[17] Pranav Khaitan. Chat smarter with allo, 2016.

[18] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[21] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86. Citeseer, 2005.

[22] Hai-Son Le, Ilya Oparin, Alexandre Allauzen, Jean-Luc Gauvain, and François Yvon. Structured output layer neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5524–5527. IEEE, 2011.

[23] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[24] Mitchell P Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. Treebank-3. *Linguistic Data Consortium, Philadelphia*, 14, 1999.

[25] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

[26] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[27] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. An analysis of neural language modeling at multiple scales. *arXiv preprint arXiv:1803.08240*, 2018.

[28] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[29] Tomáš Mikolov. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 80, 2012.

[30] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[32] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[33] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

[34] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[35] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[36] Chris Piech and Andrew Ng. K means. *Internet: http://stanford. edu/˜ cpiech/cs221/handouts/kmeans. html,[Aug 15, 2019]*, 2013.

[37] Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

[38] Sebastian Ruder. On word embeddings-part 2: Approximating the softmax, 2016.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[40] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1387–1392, 2013.

[41] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[42] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[43] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[44] Geoffrey Zweig and Konstantin Makarychev. Speed regularization and optimality in word classing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8237–8241. IEEE, 2013.