Exploring the Limits of Language Modeling

1 Citation

Jozefowicz, Rafal, et al. "Exploring the limits of language modeling." arXiv preprint arXiv:1602.02410 (2016).

`https://arxiv.org/pdf/1602.02410.pdf`

2 Abstract

We train various langugage models (i.e. probability distribution over word sequences) on the One Billion Word Benchmark and reduce state of the art ensemble perplexity down to 23.7 from 41.0.

3 Introduction

Langugage models, like word embeddings, can be a useful input to other NLP models. n-grams used to be the primary language models, but now we use neural networks. The problem, however, is that the Penn Tree Bank dataset is the primary dataset for language modeling, and it is quite small. We develop some langugage models on the One Billion Word Benchmark dataset (800M words and 800K word vocabulary). We also develop a new character-level CNN Softmax that is cheaper but just as accurate as a regular softmax.

4 Related Work

Kneyser-Ney smoothed 5-grams are the most popular traditional NLP language model. We use high capacity (i.e. large number of parameters) neural network models, but we take steps to make sure they are not too compute intensive at test time.

Some recent work has looked the character level, rather than the word level, which also does quite well. Other work has explored using convolution as well.

A key challenge in langugage models is computing the softmax, which turns a set of numbers into a probability distribution ($p(w) = \frac{\exp(z_w)}{\sum_{w' \in V} \exp(z_{w'})}$, where z_w and $z_{w'}$ are called logits). The problem is that the denominator requires summing over all words in the vocabulary, which is expensive. Importance sampling, hierarchical softmax, noise contrastive estimation (NCE), and normalizing partition functions can help speed this up.

5 Language Modeling Improvements

A language model aims to compute the probability for a given word sequence. The model the probability as: $p(w_1, ..., w_N) = \prod_{i=1}^{N} p(w_i | w_1, ..., w_{i-1})$, which requires a softmax. We'll focus on importance sam-

pling (IS). IS replaces the softmax with an approximation that is cheap to compute during training (you need full softmax during testing because you won't have the ground truth word). Basically, we imagine that we have set $W = \{w_1, ..., w_{k+1}\}$ where w_1 is the true word from the data distribution (p_d) and the rest are noise words from a noise distribution (p_n). We can then formulate our problem as trying to identify which word is from the data distribution (Y):

$$p(Y = k|W) \propto_Y \frac{p_d(w_k)}{p_n(w_k)} = \text{softmax}(s_\theta(w_k) - \log p_n(w_k))$$

for logit $s_\theta(w_k)$ (from our neural network). This lets us then approximate our desired probability as $p_d(w_k) \approx \text{softmax}(s_\theta(w, h))$.

Next, let's see how to reduce the number of softmax parameters. The logit in a softmax is computed as $z_w = h^T e_w$, where e_w is a word embedding. Word embedding matrices have size $|V| \times |h|$, so they can be huge. To mitigate this, we instead compute a word embedding with: $e_w = CNN(chars_w)$. In other words, we use a character-level convolutional neural network (CNN) over the characters of the word to compute its embedding (this is called a CNN Softmax). This drastically reduces the number of parameters in the network. One challenge with this approach is that it struggles with words that are spelled similarly but mean different things (e.g. "dead" vs. "deal"). To mitigate this, we add a per-word correction: $z_w = h^T CNN(chars_w) + h^T M_{corr_w}$. This CNN Softmax also works for out-of-vocabulary words, which is nice.

Although the CNN-Softmax drastically reduces the number of parameters, it can still be expensive to compute the softmax denominator (called the partition function). One way to get around this is to use a character-level, rather than word-level, langugage model. This makes our vocabulary tiny and we can use a character level LSTM (Char LSTM). These are hard to train and get poor performance though. Thus, a better approach is to have a word-level LSTM that outputs a hidden vector h that is fed into a character-level LSTM. This is a highly scalable model, but we haven't gotten it to work as well as CNN Softmax or regular softmax.

6 Experiments

We use Tensorflow. We mark unknown words (0.3% of the 1B Word Benchmark dataset) with a special token. Our metric is perplexity: $e^{-\frac{1}{N}\sum_i \ln p_{w_i}}$.

For character-level models, we use a max word length of 50 (we padded words if they were too short). We used a 256 character vocabulary (ASCII symbols).

We considered three langugage models. First is a regular word-level LSTM. Second is an LSTM, but where the input and Softmax embeddings have been replaced with a Char CNN. Third is a model where the Softmax is replaced by a character level LSTM.

We use dropout before and after each LSTM layer. We use Adagrad, gradient clipping, and minibatches of size 128. We train on 32 GPUs. We use 8192 noise samples per batch.

7 Results and Analysis

We found that you get better performance when you use the largest LSTM layer that you possibly can. Also, an RNN does much worse than an LSTM.

We find that dropout helps a lot. We find that importance sampling is better than NCE. We find that using character-level embeddings (i.e. $CNN(chars_w)$) is just as good as using word embeddings. CNN-Softmax also works well. Using a character-level LSTM output instead of a CNN-Softmax also works well.

Training for 2 hours is enough to beat state of the art.

The regular LSTM language model is probably the best single model.

8 Discussion and Conclusions

We tried a bunch of different approaches to langugage modeling and found that using a regular LSTM with importance sampling (to speed up softmax computation at training time) is the best single model. Our ensemble sets state of the art on language modeling.