LightGAN: An Adversarial Approach to Natural Language Generation at a Large Scale

Booher, Jonathan jaustinb@stanford.edu
De Alba, Enrique edealba@stanford.edu
Kannan, Nithin nkannan@stanford.edu
https://github.com/aprendizaje-de-maquinas/LightGAN

I. Introduction

NATURAL language generation is important in the today's environment of digital assistants. It is, however, a difficult task to generate language that makes sense. Traditional approaches like n-grams become successful when the value for n is large (generally about 5) and in that case, the text generated tends to repeat itself or simply output a sentence that it was trained on. This defeats the purpose of language generation as a method of creating new, novel sentences. Recently, RNNs (especially those with LSTMs) have become successful at modeling language that generally makes sense. These RNNs take large amount of memory that are dependent on several factors including the size of the vocabulary being trained on. In order to achieve state-of-the-art performance on large datasets, these vocabularies must be large. This poses a unique challenge when needing to train on resource limited systems (eg 2 gpus). While there is some work that has reduced memory requirements like sampled and hierarchal softmaxes, however we propose that looking at different LSTM designs and training paradigms will lead to state-of-the-art models in natural language generation. We will combine the work from two papers that were both sought to improve upon natural language generation. The first uses a GAN approach to training and the second uses a novel method (a new LSTM cell) to reduce the number of parameters that the network needs at the expense of extra compute time. We believe that this combined model will achieve a high level of performance as both methods have been shown to achieve state-of-the-art or nearly state-of-the-art performance on NLP benchmarks like the PTB and the Billion Word Benchmark.

II. The Data

The data that we are using for this project comes from the SNAP group and their dataset "476 Million Twitter Tweets". We thank Prof Leskovec for allowing us access to this data. This dataset contains tweets from September 2009 - December 2009 that are estimated to be about 30% of the tweets from that time period. An example of a tweet can be seen below:

```
T 2009-06-30 23:59:51
H http://twitter.com/eboe
W Out for karaoke and shots. Text if you dare. http://plurk.com/p/15f43e
```

We cleaned the data by removing the timestamps, username information, and duplicates. We also replaced websites, @ tags, and emojis with special tokens <URL>, <AT_TAG>, <EMOJI> in order to reduce the vocabulary size. For example, the above tweet would become:

```
Out for karaoke and shots. Text if you dare. <URL>
```

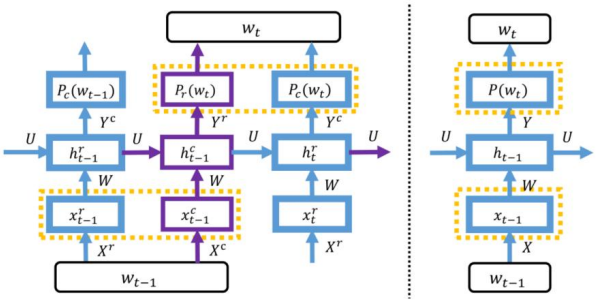
Additionally, we removed tweets that were not over 50% english sentences and removed words that we used less than 5 times. And lastly, we padded each of the tweets to a fixed length to simplify mini-batch creation at runtime with the token <NAW> for 'not a word'. This preprocessing gave us a vocabulary size of approximately $100,000$ which is at the upper limit of vocabulary sizes in common datasets (only the billion word benchmark comes close). But the systems that traditionally work with vocabulary sizes this large are distributed systems of over many (eg 32) gpus.

III. Methods

LightRNN

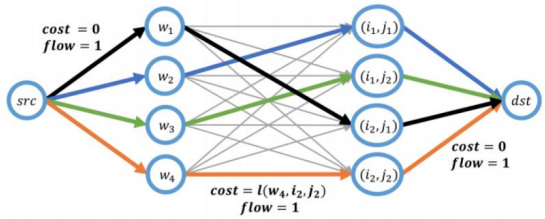
Usually in RNNs we have very large vocabulary sizes, $|V|$ that tend to exceed the memory capacity of GPUs. For our implementation we utilize a LightLSTM which uses a 2-Compact (2C) shared embedding for our word representations, where essentially each word is now represented by a row vector and a column vector. We're able to achieve this by placing all the words in a 2-dimensional table, where given a word's position in this table we can associate that word with corresponding row and column vectors. With this 2C embedding we see that we only need $2\sqrt{|V|}$ vectors instead of the $|V|$ vectors typically used in previous RNN implementations. This significant reduction in our model size is possible because any two words in the same column share the same column vector, and so forth. With this implementation of LightLSTM not only are we able to reduce the size of our model, but we also reduce its computational complexity (softmax is only to $\sqrt{|V|}$ rather than $|V|$.

Fig. 1. LightRNN (left) vs. Conventional RNN (right). [3]



Words are initially randomly allocated in the table and we begin training. Then periodically, we reallocate the table by approximating the perplexity created by each word in the vocabulary. Since our training set is so large (on the order of millions of examples), we must approximate. We perform this by calculating the log of the softmax logits from the RNN and then accumulate these logs based upon the target word (the word that should have been predicted based on the ground truth). We can then use the arrays created by this procedure, each of dimension $|V| \times \left\lceil \sqrt{|V|} \right\rceil$. Now, consider the bipartite graph with the left hand side equal to the words $w \in V$ and the right hand side containing all possible positions. Using our arrays, we can assign an edge cost $C(w, i, j)$ from any word to any position. With a source node having edges of capacity 1 and cost 0 to every word, and a similar sink node with the various positions, We can now apply our arrays to solve a min-cost max flow problem. This solution will yield an optimal allocation of words to positions in polynomial time in V.

Fig. 2. Using Min Cost Max Flow to find the ideal allocation of words to grid locations [3]



(a)

RNN Attention

Attention in RNNs is a way to have the network learn what parts of the sentence to look at when making a prediction on the next word. This method has largely been used for machine translation tasks; however, attention has been used in sentiment analysis and object recognition as well. We use attention with a similar motivation as attention used for object recognition: identifying what part of the input (the sentence so far) is relevant to producing the output (the next word in the sequence).

Briefly, attention is implemented by first generating a context from a window (be it fixed or learnable) by running hidden states through a single dense layer and then through a softmax to determine a probability distribution from the context. This probability distribution is what tells the network what part of the window to focus on. In this way, the network output at a time step t can be influenced by a time step other than $t-1$.

We use attention with a fixed window of maximum length 10 in both the generator and discriminator. And we found this to substantially improve the stability of the GAN during training as well as improve the performance of the model at test time.

WGAN-GP

A Generative Adversarial Network (GAN) architecture in general seeks to train two different networks a Generator and a Discriminator (or Critic). The generator is trained to produce novel inputs to 'fool' the discriminator which is trained to predict classes for examples (ie 1 for real and 0 for fake) as accurately as possible.

This WGAN-GP architecture is an improvement upon the original GAN with the goal of improving stability. This GAN architecture (which is an improvement over the vanilla Wasserstein GAN) optimizes what is called the Wasserstein Distance. This is a measure of the distance between two probability distributions. This makes sense in the case of GANs as we would like the output of the generator to be from the same distribution as the real data. The addition of the gradient penalty (GP) helps stabilize GAN optimization by regularizing the discriminator by adding a term penalizing the distance the gradients of the discriminator are from 1 thus limiting the size of the gradients [5]. Note that this can be seen as an extension of regular gradient clipping. Other forms of regularization like weight decay or L_2 have been shown to produce generators that learn overly simple functions [5]. To simplify training, we run two batches through the discriminator at every step, one of real data (x) and the other of fake data ($\hat{x}$) (this is called minibatch discrimination). Thus the loss function for the discriminator is:

$$L_D = \frac{1}{B}\Big(\sum_{i=1}^{B}\hat{x}_i - \sum_{i=1}^{B}x_i\Big) + GP$$

$$GP = \lambda \cdot \mathbb{E}\Big(\big(||\nabla D(x + R\cdot(\hat{x}-x))||_2 - 1\big)^2\Big)$$

where $R \sim Uniform$ and the loss for the generator is:

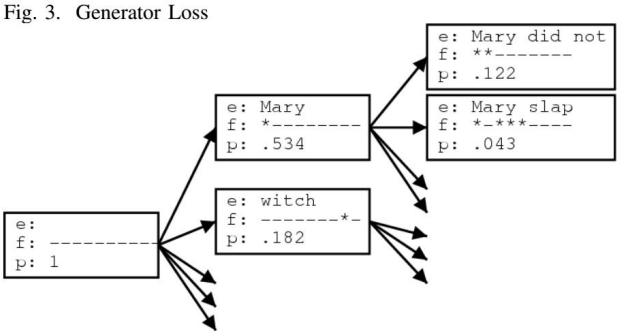
$$L_G = -\frac{1}{B}\sum_{i=1}^{B}\hat{x}_i$$

for batches of size B and λ is a regularization parameter that is set to 10 from the original WGAN-GP paper. Note that these summations can be seen as an approximation of the expectation of their respective distributions. Additionally note that we can extend the concept of the GP to the LightRNN architecture by simply averaging the GP for both the row prediction and col prediction.

Beam Search

Once the generator of the LightGAN is trained to accurately predict the next word, we use a beam search to produce our tweetbot's output. Rather than greedily selecting the next word of our tweet based on the RNN's next-word output, beam search allows us to explore multiple next word candidates. The greedy approach is flawed as it may select a word that may seem optimal at the time, but causes following words to occur with low probability. Beam search does not suffer from this as it takes its candidate words and continues with a BFS-like search for a following word, only keeping the top-most candidates. This allows our algorithm to not get 'stuck' at any point. This algorithm runs in $O(Bm)$ time where B is the size of the beam (the number of candidates) and m is the maximum depth of the search.

Fig. 3. Generator Loss



We use a beam width parameter of 100 for our search as any higher resulted in in search taking too much time. The image depicts a beam search of width 2, where the beam algorithm looks at the two most promising candidates at each level.

IV. Training

We train the GAN using what is called Curriculum Training. In this method, we train on progressively larger strings (from 1 all the way to 32, the average tweet length). In this way, the network will be able to build up to full tweets. This makes sense in the context of language generation as the network is learning sentence structures as well (i.e. tweets tend to start with '`RT <AT_TAG>:`'). In this curriculum training, we train the generator to predict the next word in a random subsequence of an actual tweet. This method of training the generator is prudent as it the way that traditional language models are

trained. Because of the curriculum training, the generator will have good *context* for its predictions and will be able to make a well informed prediction of the next word.

We train using Adam with $\texttt{lr} = 10^{-4}$ on both the Generator and Discriminator. We train the Discriminator 5 times more frequently than the Generator. This technique for training is similar to the one that was used in [4]. Additionally, we found the values of β_1 and β_2 to be important for stabilizing the GAN. The default values of $\beta_1 = 0.9$ and β_2 resulted in the GAN being incredibly unstable. We found that the values $\beta_1 = 0.5$ and $\beta_2 = 0.9$ to be the best for the LightGAN. Intuitively, reducing the βs will increase the decay of previous iterations and make the optimization more dependent on the current gradient. We believe that this results in better learning for the LightGAN as GANs are extremely sensitive to small perturbations of weights so by using an optimization closer to gradient descent, we reduce the opportunity for the optimization to diverge while still keeping the learning acceleration that using Adam affords us.

Additionally, we found the model to be extremely sensitive to changes in the learning schedule (ie how many times we run the optimizer for the generator and discriminator on each iteration). With a poor schedule, mode collapse was common and came early in training. However, with better schedules, we were able to attain significantly better model stability without seeing any signs of mode collapse for the duration of training (see the graphs below).

V. Results

As seen in the graphs of the losses, the GAN appears to converge and suggest stability of the model. The discriminator loss (the distance between the real and fake distributions is close to 0 and the generator loss is moving away from 0 as it is supposed to based on the optimization). There were a couple outliers on the discriminator loss (which have been omitted from the graphs for scaling reasons) that can be attributed to a large value of the gradient penalty during that iteration.

Some example outputs from the model can be seen below:

```
<AT_TAG> be oversleeping my scholl
```

Fig. 4. Generator Loss over time

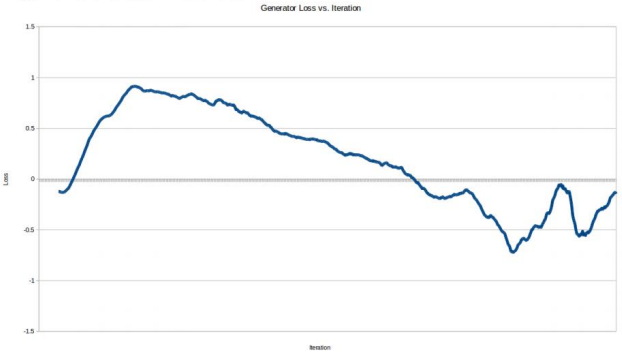
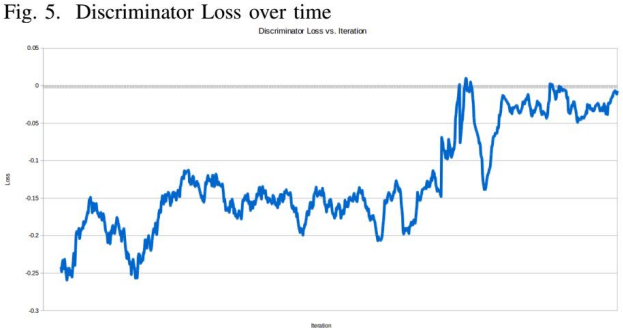


Fig. 5. Discriminator Loss over time




```
what wud you do ?
rt <AT_TAG> :
<URL> -
<AT_TAG> we dog breeders of
thems kobe trends ! wowowow
```

While the output of the GAN was generally positive, the GAN also produced output like:

```
<URL> a applaud bajillion coming
<AT_TAG> starvation akira be #iremember
finally dinner californians stretcher rubbing
```

Analysis of the Results

The quality of the GAN output is influenced by the values of the random tensors passed in for the initial state of the RNN. Potentially an approach in the future is to seed the initial state of the RNN with the final state of another forward prop through the RNN (ie from a ground truth). This would allow the RNN to 'warm up'.

We know that the allocation table and the word embeddings were learned appropriately because if we take the word embeddings and allocation table generated by the GAN and simply use them in a LightRNN based language model, we can correctly predict the next word with an accuracy of

approximately 55% which is a good result considering the amount of time we were able to train, the approach taken, and the size of the vocabulary. To add to these factors, the vocabulary contained many alternate spellings or versions of the same words (lol and $lololol$) which largely accounts for incorrect predictions of continuations.

In a similar vane, we can clearly see that the word allocation is grouping similar words together in the rows. This is a property that is attributable to the word reallocation step since similar words will tend to have similar perplexities given the same model parameters. For example:

$..., \#popculture, \#owt, \#wotd, \#usnews, \#fcv,$
$\#thingsaguyshouldknow, \#vux, \#jblogs, \#uknouafrican,$
$\#prayer, \#freeny, \#baby, \#onmymomma, \#cfp, ...$
and
$..., ummmm, hahahhaa, haaaaa, hahahahahahah,$
$lmfaooooooo, wellllll, wowowow, bwahahahaha, ayyy,$
$yaaaay, damnnnnn, lololol, bwahahaha, gahhhh, ...$

Clearly, the first example row is a 'hashtag' row and the second one is a row of emphasis or exasperation. These groupings make sense because elements of these groups are generally located in similar places in tweets or convey similar meanings (ie 'hashtags' used for emphasis on key points and for searchability). This property of the allocation table and the associated word vectors can be seen as a clustering and can could also be used in other applications where having such a clustering would be useful.

Implications of the Results

One of the major implications of these results is that we showed the ability to train models on extremely large vocabularies on a relatively resource constrained system. For example, language modeling of certain dialects of Chinese with a vocabulary size of over $300,000$ is now within reach on even moderately large systems. Similarly, both the Generator and Discriminator could be used on platforms like Facebook or Twitter. The generator could be used as a way for entities like new organizations to get news out quickly and in a language common to the platform as a whole. This could be accomplished by instead of seeding the RNN with a random initial state, to seed it with a state that is correlated with the topic at hand or seed it with a human generated substring and have the RNN propose a continuation. Similarly, the Generator can also be used to help digital assistants communicate in a more vernacular sense and make them more personable. On the other hand, the Discriminator can be used to help combat bots on the same platforms. One could envision this discriminator identifying spam or bot tweets on Twitter to flag suspicious accounts.

VI. Future Work

The output of the RNN could be further improved by adding more layers to the RNN and training for more epochs before performing the reallocation step. The former would help improve the quality of the output by the RNN at any given step and the later would help the min cost max flow algorithm by ensuring that our current predictions are as accurate as possible for the current word allocation table.

The GAN itself could be further improved by training the generator and discriminator for more times on each iteration. Currently we are training the generator at a $5:1$ ratio over the generator. Further search for a better schedule of training could help stabilize the GAN.

Lastly, experimenting with different training paradigms could help us improve as well. Curriculum training makes it so that the RNN does not see many of the words in the vocabulary until extremely late in the training. This complicates training especially for the implementation of the min cost max flow solver as the information passed to the solver early in curriculum training is sparse. Improvements could be made by selecting random substrings of up to the maximum length for each iteration of training.

VII. Contributions

We all worked on understanding the LightRNN concept and the WGAN together. Jonathan worked on implementing the GAN, stabilization, and training, Enrique worked on the LightRNN cell, and Nithin worked on the data input pipeline.

References

[1] J. Yang, J. Leskovec. Temporal Variation in Online Media. ACM International Conference on Web Search and Data Mining (WSDM '11), 2011.
[2] J. Leskovec, A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection, June 2014.
[3] Xiang Li, Tao Qin, Jian Yang, Xiaolin Hu, Tie-Yan Liu: LightRNN: Memory and Computation-Efficient Recurrent Neural Networks. NIPS 2016: 4385-4393

[4] Ofir Press, Amir Bar, Ben Bogin, Jonathan Berant, Lior Wolf: Language Generation with Recurrent Generative Adversarial Networks without Pre-training. CoRR abs/1706.01399 (2017)

[5] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin and Aaron C. Courville. Improved Training of Wasserstein GANs. CoRR abs/1704.00028 (2017)