MLP G95 Coursework 4: Text Generation using Generative Adversarial Networks

s1781323 Ryotaro Miura   s1412880 Konstantinos Stergiou   s1407470 David Robinson

Abstract

In this paper, we investigate two different neural network architectures for text generation. We use a standard recurrent neural network method and attempt to generate text using a generative adversarial network. We also look at the effect of using POS tag data in addition to word data. We conduct experiments in producing text using the Brown corpus as our training data. We then show that the models we build do not outperform the baseline, with the GAN model repeating the most common word in the dataset and the RNN model also repeating words.

1. Introduction

The aim of this project was to investigate methods of text generation using neural networks (NNs). We will consider two NN methods in this report. Firstly we will be using a long short term memory(LSTM) model, a type of recurrent neural network (RNNs) for generating text using word vectors as the input to our model and returning generated text. Furthermore, we will investigate the effect of including POS tag data as well as word data in our training and see if this yields better results than the standard RNN method. We will also attempt text generation using generative adversarial networks (GANs) with word vectors as the inputs. We will then compare the text generated from these methods against a baseline of standard NLP methods for text generation and assess whether they produce more natural text.

There are many reasons why text generation is important in natural language processing and artificial intelligence in general. It can be used for automatic generation of documentation, to more complex things like replicating the works of an author (Xu), summarising text and machine translation. RNN based methods are now widely used across NLP for text generation. Despite this, they still have several limitations. Error in RNNs is usually obtained on a word-by-word basis, however, when reading text people would naturally assess it on a larger sentence or paragraph based level. This means that error is calculated on a different level than the results are being evaluated at.

One possible method of addressing this issue is to use a GAN. GANs work by having two neural networks training together. One network is performing some task while the other is learning to evaluate the first's results. This means the output from the second network can be used as the error for the first. So the first network is training to produce an output that the second cannot differentiate from the desired target output. This can be implemented for NLP by using a RNN as the first network generating text using data from a corpus. The second network is then learning to distinguish between text generated by the RNN and text that occurs in the corpus.

GANs have already been used successfully in image generation, and there is much recent research focused on applying such a model to text generation. Two common approaches are used, one is to train a model on a character level to generate text which proves successful over short sequences of text (Press et al., 2017). We will use the other approach which is to train our models on a word level.

For these experiments, we will be using as our training data the Brown corpus. This contains 1,161,192 words from 500 documents from different genres of written text. In addition to this, every word is POS tagged using two different tagsets. We will be training our models using word vectors generated from this data. These vectors are created using a one-hot encoding and extended to a two-hot encoding as outlined below.

2. Methodology

2.1. Recurrent Neural Network

We used an n-gram structure, with n=9. We convert words to one-hot encoding and a two-hot encoding for each of the two model structures we will be exploring. 1st case: We predict the next word. When a sentence starts, the input becomes a vector with length 9 with the first 8 being zero vectors, to indicate the absence of tokens. As we iterate through a sentence to create all possible inputs, the list becomes more populated step by step. Whenever we encounter a sentence delimiter, such as '.', '!', '?' the input vector get reset to 8 zero vectors and the first word of the sentence. 2nd case: When using two-hot vectors we concatenate the POS tag of a token, using a one-hot encoding to represent it, with the one-hot encoding of the token. Since every word can have a different POS tag, depending on where it appears in a sentence, we have two choices. Either of creating a word: [possible tokens] dictionary and trying to predict all possible POS tags for a word or use the nltk method nltk.corpus.brown_tagged_words/sents() where every token has its specific POS tag for each appearance. We decided to go with the latter option since it would reduce the complexity of the problem. That is because predicting every possible POS tag would mean part of the prediction would be multi-class which is more complicated to learn. Furthermore, when including only the POS tag of a token in the specific appearance the language model also learns syntactical information about the corpus, which is an important part of creating coherent sentences.

For example:
- I will attack the enemies
- I will declare an attack on the enemies.

With our design choice the system isn't ambiguous as to what POS tag it gives to the word attack in either case since in the first sentence the tag for attack is VERB and in the second case it is NOUN. If we chose to try to predict all possible tags for each word, then the system would learn an inherently ambiguous model.

There are two possible tagsets we can use, those being the full tagset of the brown corpus or the smaller, universal tagset. The former contains 472 tags whilst the latter only 12. Using the full tagset has the advantage that our model would be able to learn a better syntactical representation of the corpus, since the simpler tagset does not, for example, differentiate between plural and singular forms. At the same time though, it would mean that instead of a 12-class problem we end up with a 472-class problem. This will inherently mean that we need much more data to learn a good representation, since class sparsity will dramatically increase(3833% / 39times more classes). For the sake of simplicity and in order to not need a massive amount of data to train on and to speed up training we chose to use the simplified 12 tags. This will allow the model to although learn a simpler language model, be able to do it quicker and possibly better since it not only has less classes to learn but also due to the fact that we can expect some syntactical information to be captured even without the POS tag prediction part of the training.

The choice for using an n-gram with n=9 w/ our RNN follows the finding from (Chelba et al., 2017) where it was found that using LSTM's with n=9 outperforms n=13 if sentence boundaries are kept(not crossed), which is what we have decided to do. It is also the best middle ground for an implementation without sacrificing run-time and accuracy since in the same paper it was found that the difference in the perplexity of the language models is about 10%. Therefore we can see that even thought using an history of unlimited length produces the best results it also means that input will be massive and therefore heavily impact run-time and how much RAM the model uses.

2.2. Generative Adversarial Network

We use the Sequence Generative Adversarial Nets model. This model, SeqGAN, is produced by (Yu et al., 2017). This model is the GAN application to discrete sequence data. As the same as usual GAN model, this model has two networks, generator and more discriminator. The generator is a one-layer basic LSTM but we applied reinforcement learning to it, where previous tokens are the states (stored in the hidden states) and the action is the next token to generate. The discriminator, we used CNN, is fed with both real and generated data to local the difference.For evaluating some partial sequence, this model also uses another generator to fill the rest by Monte Carl searching, and these two generators share the same parameter.

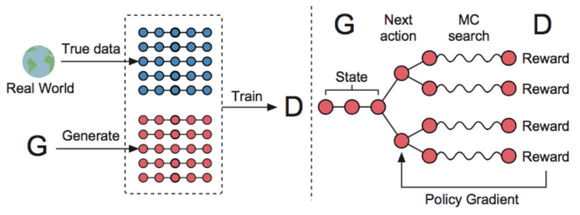

Figure 1. The figure of the SeqGAN structure

Algorithm 1 Sequence Generative Adversarial Nets

```
Require: generator policy $G_\theta$; roll-out policy $G_\beta$; discriminator
    $D_\phi$; a sequence dataset $\mathcal{S} = \{X_{1:T}\}$
 1: Initialize $G_\theta$, $D_\phi$ with random weights $\theta, \phi$.
 2: Pre-train $G_\theta$ using MLE on $\mathcal{S}$
 3: $\beta \leftarrow \theta$
 4: Generate negative samples using $G_\theta$ for training $D_\phi$
 5: Pre-train $D_\phi$ via minimizing the cross entropy
 6: repeat
 7:   for g-steps do
 8:     Generate a sequence $Y_{1:T} = (y_1, \dots, y_T) \sim G_\theta$
 9:     for $t$ in $1 : T$ do
10:       Compute $Q(a = y_t; s = Y_{1:t-1})$ by Eq. (4)
11:     end for
12:     Update generator parameters via policy gradient Eq. (8)
13:   end for
14:   for d-steps do
15:     Use current $G_\theta$ to generate negative examples and com-
        bine with given positive examples $\mathcal{S}$
16:     Train discriminator $D_\phi$ for $k$ epochs by Eq. (5)
17:   end for
18:   $\beta \leftarrow \theta$
19: until SeqGAN converges
```

Figure 2. Pseudo-code. of the SeqGAN structure

MLE is the maximum likelihood estimation method. The roll-up policy is the method to fill the partial sequence.

In the paper, they use the sequential numbers generated by LSTM network. Also, they train the model with some text dataset split by character. We used brown corpus as a training dataset. Instead of character-base learning, our experiment was word-base learning. First, we split the dataset by word and convert each word to discrete numbers. Then apply them to the networks, and lastly decode the sentence from the sequential numbers.

Trigram HMM baseline	Viterbi generation baseline
Each year these shows have increased steadily over the volumes up to tempt him.	Of which its conservative order conjures the rain put.
A year of over-achievement.	After by the dress of large and premarital historian with the strong reaction bending at such the self-consciousness blood in the glass, take even too, houston, simmel, of the own usage was in all their achieving functions.
High-speed buses on the way in the rise or fall short of forty lashes we finally managed to cover the imputed operating expenses both in the district of small firms have the class discussions were to be a matter of life, liberty and its civilization based on this side of the home folks again, “texture”, he said he was only ten days after donation and were washed three times as many as we shall see that the church in revolutionary china, the ruins marked, the sound of miles per year per student would enable her to stop civil rights and obligations must be taken of unusual interest in the tiled kitchen.	Third a united men no., congress went an literary part and the dictionary hideaway.
The formula.	“In three song he forgot partly on them.
He is up for the very smell of the excessive heat which clamped an uneasy, how can you say” – as long as the wife is, like any other desk officer, non-com and trooper was the part of reality.	Drag of the more july and its rehearsals, and all of zhok, anne, and if the old big shows were exposed together.

3. Experiments

3.1. Generative Adversarial Network

We trained the network with 120 epochs and generated 20-word sentences. However, all results are same,
the the the the the the the the the the the the the the the the the the the

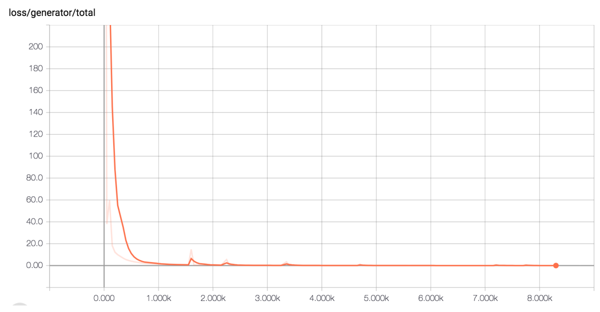

Figure 3. The generator loss

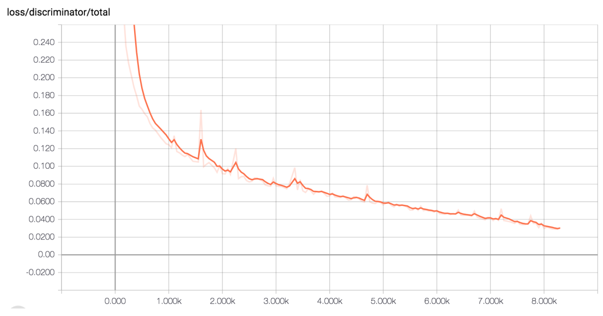

Figure 4. The discriminator loss

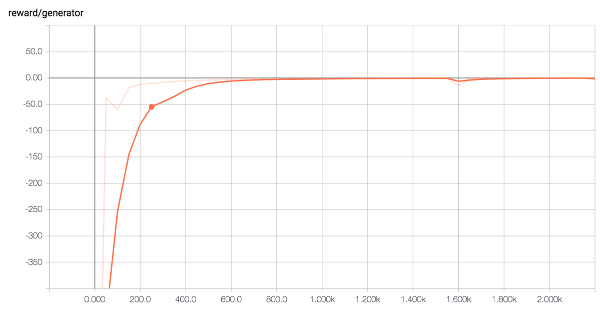

Figure 5. Reward for generator

This is because of the variety of words in the dataset. In brown corpus, the word 'the' takes up 6% in the dataset, and the top 30 words equal to more than 40% of the dataset.

3.2. Recurrent Neural Network

When generating a sentence, we can either start with a 9-gram with zero vectors or provide starting words as a seed of sorts across all generations. The latter could help in comparing the sentences since keeping the initial seed of the sentence constant means that we can more easily compare the quality of generated text across the different hyper-parameter settings of the model. So for having a more straight-forward evaluation, we decided to keep a constant starting seed. We trained our network for 25 epochs, using minibatches of size 50 and tried generating 5 sentences. When we tried to generate those sentences, the results were similar with the GAN's results, with the main difference being that instead of 'the' being repeated over and over, in this case, 'aided' was the word that was repeated with no end.

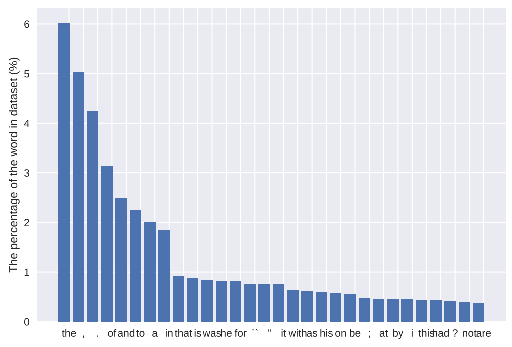

Figure 6. Histogram of word frequency in the Brown corpus

4. Related Work

4.1. Text Generation Based on Generative Adversarial Nets with Latent Variable

This paper (Wang et al., 2017) proposes a VGAN model. This combines a variational autoencoder (VAE) with a GAN method for text generation. They use a LSTM as their generative model and a convolutional neural network (CNN) for their discriminative model. They train their VAE on a one-hot representation of the words and use the word vectors produced by the VAE as input to their GAN model.

The VGAN model was found to outperform two previous neural network text generation methods, seqGAN (Yu et al., 2017) and RNN language modelling producing more natural sentences from Amazon food reviews.

4.2. Text Generation using Generative Adversarial Training

The paper (Xiao, 2017) trains both RNNs and gated recurrent units (GRUs) as both the generator and discriminator networks in a GAN for generating text from books using a word-level model. It was found that the GAN models using GRUs tended to outperform those using RNNs with the generated text having some sense of grammar and seeming to have some logic.

4.3. Language Generation with Recurrent Generative Adversarial Networks without Pre-training

The paper (Press et al., 2017) again attempts to generate text using GANs. In this paper they use GRU-based RNNs as both the generative and discriminative models in the GAN. This paper uses a character based training approach rather than the word based approach used in our paper. It then goes on to implement curriculum learning (Bengio et al., 2009) which begins by training with small sequences, increasing the length of the sequences as training continues.

They found that 3.8% of the 4-grams generated by their GAN model could be found directly in the training set. The curriculum learning model performed well at generating short sequences of text but could not generate natural sounding text when the sequence length exceeded 32.

4.4. Neural Models Comparison in Natural Language Generation

The paper (Zhang et al., 2017) compares using RNNs trained on a word level and RNNs trained on a character level with using a character-based WGAN (Arjovsky et al., 2017). It was found that the character-based WGAN could outperform the character-based RNN when trained for a much longer time due to the high computational cost of training a WGAN. However, both were outperformed by a word-based RNN model due to a high amount of spelling errors. It was concluded that while the use of GANs is promising and could produce better results than RNN models, the additional training time means that RNNs are better as a text generation method.

5. Conclusions

As can be seen from the results our NN models were largely unsuccessful in producing natural sounding text. There are many possible reasons for this. On issue associated with training an NN model on a word-basis is the input size combined with the large quantity of training data. For our data each input vector was 49,000 values in size meaning the input and output layers of our networks needed to be the same size. This had a significant impact on increasing the time needed to train our models. One way this issue could be solved in the future would be to convert our words to vectors using pre-trained word embeddings. This would dramatically decrease the size of the input and output layers, therefore decreasing training time and allowing us to train over more epochs.

Additionally due to the very high training times associated with both RNNs and GANs it was challenging to sufficiently train such models on limited and very unstable computational resources. A lot of the models we attempted to run on the MLP cluster did not successfully complete training with us unable to train any of our models on the cluster in the final week of the project.

This can be seen in the RNN results where the word that was repeated non-stop('aided') only appears 11 times in the whole corpus which shows that the network is not fully trained, especially when taking account the amount of times 'aided' appears in the corpus. More layers and more epoch will have probably resulted in coherent text being generated. This issue is especially prevalent in the GAN model where the discriminator does not run over enough epochs to accurately distinguish between the generated and original text. This resulted in the generator simply repeating the most common word in the dataset, Figure 6, to receive a low as possible error from the discriminator that does not have sufficient training and so has a very simplistic method of trying to distinguish between real and generated text. We can see this effect in Figure 3. Here the generator very quickly achieves very low error from the discriminator with the error remaining low over all following epochs. If the discriminator was sufficiently improving there would be the occasional increase in generation error.

6. Further Work

To continue this project further experiments could be run to try producing better text by running our exiting models over higher numbers of epochs. A good approach to change would be to implement pre-trained word embeddings for representing our words and concatenate these with the POS part of our two-hot encodings. This should produce significantly better training times to allow the model to be trained over more iterations.

Further experiments could be to investigate the effect of adding POS tag data to the quality of generated text. This might be promising as it could allow the networks to produce their own internal grammar model, thereby improving the outputted text.

One other approach we could take would be to try text generation using GANs on a character-level and see if this offers improvements when run over a low number of epochs.

References

Arjovsky, Martín, Chintala, Soumith, and Bottou, Léon. Wasserstein GAN. *CoRR*, abs/1701.07875, 2017. URL `http://arxiv.org/abs/1701.07875`.

Bengio, Yoshua, Louradour, Jerom, Collobert, Ronan, and Weston, Jason. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, 2009.

Chelba, Ciprian, Norouzi, Mohammad, and Bengio, Samy. N-gram language modeling using recurrent neural network estimation. *CoRR*, abs/1703.10724, 2017. URL `http://arxiv.org/abs/1703.10724`.

Press, Ofir, Bar, Amir, Bogin, Ben, Berant, Jonathan, and Wolf, Lior. Language generation with recurrent generative adversarial networks without pre-training. *https://arxiv.org/pdf/1706.01399.pdf*, 2017.

Wang, Heng, Qin, Zengchang, and Wan, Tao. Text generation based on generative adversarial nets with latent variable. *arXiv preprint arXiv:1712.00170*, 2017.

Xiao, Xuerong. Text generation using generative adversarial training. 2017.

Xu, Joyce. Machines and magic: Teaching computers to write harry potter. URL `https://medium.com/startup-grind/machines-and-magic-teaching-computers-to-write-harry-potter-37839954f252`.

Yu, Lantao, Zhang, Weinan, Wang, Jun, and Yu, Yong. Seqgan: Sequence generative adversarial nets with policy gradient. *https://arxiv.org/pdf/1609.05473.pdf*, 2017.

Zhang, Lixue, Cui, Wen, Tong, Sha, Xu, Ran, and Liu, Yifeng. Neural models comparison in natural language generation. *https://pdfs.semanticscholar.org/b92a/67109e990f850d02a62e9aa4812dc61eb78b.pdf*, 2017.