Adversarial Text Generation
NLP and Deep Learning — Final Project

Andreas Holck Høeg-Petersen
`anhh@itu.dk`

Mathias Bastholm
`mbas@itu.dk`

August 13, 2020

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1 Introduction

In recent years, Generative Adversarial Networks (GANs) have gained a lot of traction in the Deep Learning community because of their impressive results in image generation. The general idea is that a generator and a discriminator are jointly trained to produce an image output that is seemingly indistinguishable from non-generated images. This model were first described in Goodfellow et al. 2014.

We want to attempt to apply this strategy for text generation. The main difficulty for this task is that whereas image outputs can be considered a continuous value, a sentence is inherently discrete as it is a sequence of words each of which is chosen by the model using the non-differentiable $argmax$ function. To remedy this, we propose a model where the discriminator is trained to distinguish between the continuous outputs of a pre-trained encoder given a 'true' sentence from the generated, 'fake' output stemming from our generator.

In our project, we will construct and train an autoencoder model that can encode and decode a sentence from English to English. The encoded sentences are then used as labelled training data for the discriminator, representing 'true' values. The job of the generator is to produce similar encodings but doing this from random noise in a way that makes the discriminator unable to distinguish between the encodings stemming from the autoencoder and the encodings stemming from the generator.

Ideally, this would train the generator to produce sentence encodings that can be fed to the decoder of the Transformer model which would then produce meaningful sentences from this artificially generated input. See Figure 1 for an overview of the complete model.

This project thus have two objectives: one is to construct a working autoencoder that can map an English sentence to some hidden state $\mathbf{X}$ with a corresponding decoder that can extract the original sentence from $\mathbf{X}$. For convenience, we will refer to the encoder part of this model as the 'Teacher'.
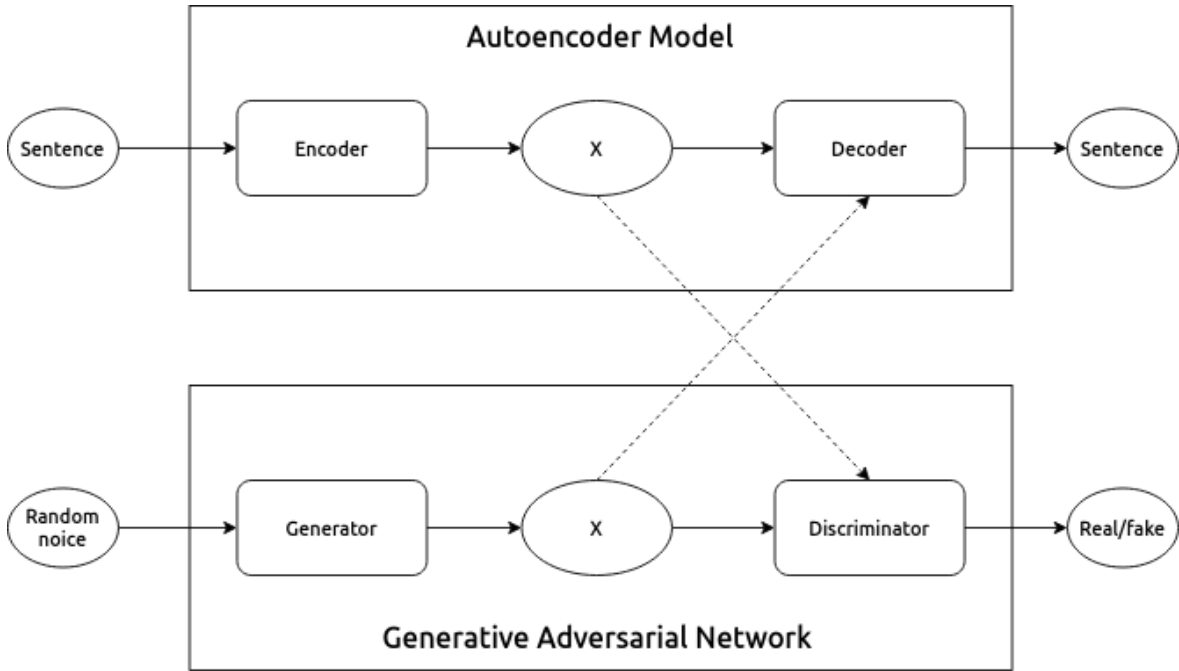
Figure 1: Overview of the model architecture. The dotted lines from the **X**s represents that the encoded and generated **X**s will be fed to the discriminator and the decoder during training and evaluation, respectively.

The second objective is to build a GAN network, where a generator — the 'Student' — must learn to produce approximations of **X**.

The second objective is highly experimental as explained in Section 2, where we will also describe other approaches at using the GAN architecture for NLP problems. In Section 3 we will describe how we have build the different parts of the model and how we utilize our dataset. Then in Section 4 we will present our results and discuss the shortcomings of the model*s*, and in Section 5 we will proceed to suggest improvements and ideas for further research. Lastly, in Section 6 we conclude on our project.

2 Background

Applications GANs have mainly focused on image generation and has not yet seen a major breakthrough in text generation. As mentioned in Section 1, this is because the discrete nature of text, which is basically a sequence of words, requires a non-differentiable *argmax* function to transform a probability distribution over a vocabulary to a single value (ie. the word with the highest probability). This is depicted in Figure 2.

There have been, however, multiple attempts at working around this issue. According to Chintapalli 2019 these approaches can broadly be categorized into three types:

- Reinforcement Learning-based solutions
- The Gumbel-Softmax approximations
- Avoiding discrete spaces by working with the continuous output of the generaor
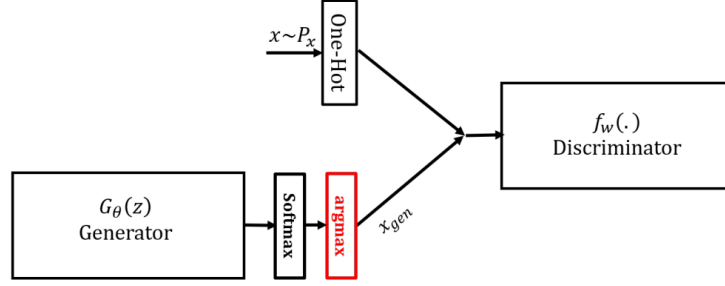
Figure 2: A simple GAN model where the generator output is run through an *argmax* function before being given to the discriminator. This prevents gradients to flow from the discriminator to the generator. Source: Haidar and Rezagholizadeh 2019.

In this project, we follow the third approach, but in this section we will give short introductions to the idea behind the two first.

As an example of an RL-based solution, Yu et al. 2016 proposes the SeqGAN model, in which the generator is considered an RL-agent with states s_t being the text generated at timestep t and actions $\mathbf{a}$ being all the possible words to choose next. The agent then chooses its next word (takes an action a) based on some policy function $\boldsymbol{\pi}(a \mid \boldsymbol{s}_t, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters to be optimized. Using Monte-Carlo rollouts to produce a number of different sentences sharing a prefix $\boldsymbol{s}_t$, the discriminator then rewards each sentence and the averaged reward is then used to perform gradient ascent on $\boldsymbol{J}(\boldsymbol{\theta})$, where $\boldsymbol{J}$ is the performance measure of $\boldsymbol{\theta}$. This approach alleviates some of the problems of training a GAN for text generation, but it suffers from an unstable and slow training process, convergence to sub-optimal local minima and and extremely large state-space.

Another approach is to use the Gumbel-Softmax distribution to approximate a one-hot encoding of a probability distribution passed through the *argmax* function. This is the approach taken by Kusner and Hernández-Lobato 2016. Here, a d-dimensional one-hot encoding vector $\boldsymbol{y}$ is approximated using

$$\boldsymbol{y} = \text{soft max}(\frac{1}{\tau}(\boldsymbol{h} + \boldsymbol{g})) \tag{1}$$

where $\boldsymbol{h}$ is some hidden state (ie. of an RNN), $\boldsymbol{g}$ is drawn from a Gumbel distribution and τ is a temperature parameter. This works because it is differentiable and as $\tau \rightarrow 0$ the distribution of $\boldsymbol{y}$ will match that we get from

$$\boldsymbol{y} = \text{one\_hot}(\arg\max_i(h_i + g_i)) \tag{2}$$

which again can be shown to be the same as sample $\boldsymbol{y}$ from a probability distribution $\boldsymbol{p} = softmax(\boldsymbol{h})$ where $p_i = p(y_i = 1), i = 1 \ldots d$.

Finally, for other examples on the third approach, see Donahue and Rumshisky 2018 and Haidar and Rezagholizadeh 2019.

3 Method

For all our models (autoencoders and GANs), we drew inspiration PyTorch tutorials (Robertson 2020, *Sequence-to-Sequence Modeling with nn.Transformer and TorchText* 2020, Inkawhich 2020), tweaking them to our specific needs. The whole process (project development, research, data collection, coding, training, experimentation and analysis) was conducted during a 10-day period.

This section will describe this process, focusing on the final outcomes rather than including all our intermediate steps and missteps.

3.1 Dataset

For training of the autoencoders, we simply needed dataset consisting of a large number of English sentences. We obtained this from Universal Dependencies, where used the 'Universal Dependencies — English Dependency Treebank Universal Dependencies English Web Treebank v2.6 — 2020–05–15' (Silveira et al. 2014) consisting of 12,543 training sentences, 2,077 test sentences, 2,002 dev sentences and a vocabulary of 16,654 training tokens. The data is annotated with metadata such as lemmas and word classes, but we discarded this information as it was not relevant for our purpose.

Furthermore, we also utilized another dataset intended for training English-to-French translation. This dataset originates from `https://tatoeba.org/eng/` and consists of 135,842 sentence pairs. We discarded the French sentences and removed all duplicate sentences and sentences of length smaller than 3, as well as splitting all sentences that contained punctuations, questionmarks, exclamation points, etc. This gave us a set of 92,343 sentences, which we split 80/20 between training and testing/development, and a vocabulary of 13,731 tokens.

3.2 Models

We developed two different versions of the autoencoder model and one GAN model. These will be described in this subsection.

3.2.1 The TransformerModel

Our first autoencoder was based on *Sequence-to-Sequence Modeling with nn.Transformer and TorchText* 2020. This model consists of a very simple decoder, that is simply a feed-forward neural net that takes a 2-dimensional tensor $X \in \mathbb{R}^{n \times k}$ and maps each of the n k-dimensional vectors of the sequence to a probability distribution over the entire vocabulary which it then can convert to an output sequence using $argmax$.

The encoder, however, is responsible for generating X and it does so by using the Transformer architecture as suggested in Vaswani et al. 2017 and implemented in the PyTorch module `nn.Transformer`. For our purpose, however, we only used the submodule `nn.TransformerEncoder`, which consists of a stack of encoder layers that uses self-attention to focus on specific, relevant parts of the input sequence in one go, and then passes its output on to the next layer through a feed-forward network. As a preprocessing step, before the input sequence is passed through the transformer, positional encoding is added, as suggested in the paper (ibid.).

3.2.2 RNN and Attention based Autoencoder

In our second model, the heavy-lifting is switched from the encoder to the decoder. Again, we utilize the attention mechanism, but this time it is combined with an RNN architecture, more precisely a Gated Recurrent Unit (GRU). In this model, the encoder is simply a GRU layer that processes the entire sequence and then its final hidden state aswell as the output for each word in the sequence is passed to the decoder.

The decoder has a bit more to it. On each timestep it takes in its own last output (starting with the special `start-of-sequence`-token), a hidden state (starting with the last hidden state of the encoder) and all the encoder outputs. It then uses a linear layer (ie. a simple feed-forward network) to calculate the attention weights by combining the input and the current hidden state. The attention weights and the encoder outputs are then mulitplied together using matrix multiplication and the result of this operation can then be merged with the original input and passed through another linear layer to produce a vector of size (`sequence_length * hidden_size`).

In this vector, the decoder has now embedded all the information about where to focus its attention, and it can pass this to a GRU just as in the encoder — however, opposite to the GRU in the encoder, the output of the recurrent layer in the decoder is responsible for mapping back to actual words. This mapping is finalized by an output linear layer that expands the hidden size to the size of the vocabulary and then, finally, a *softmax* operation converts the output to probability distributions.

When the decoder outputs the special `end-of-sequence`-token it is finished and the process terminates. Our implementation of this model is based on Robertson 2020.

3.2.3 GAN

Our GAN model is modelled after our TransformerModel in an attempt to mimic the inner mechanics of the network it is trying to imitate. As stated, a GAN consists of a generator and a discriminator. The generator gets a vector of random noise as input. This vector has dimensions (`max_sequence_length * embedding_size`). The reason we set a parameter `max_sequence_length` is because neural networks expects static sizes, but since sentences can have different lengths, we set an upper bound. The generator should, however, be able to encode its output in a way, that produces sentences of length 0 to `max_sequence_length` (ie. by encoding an `end-of-sequence`-token at some position).

The generator passes its input through a linear layer and then, as the TransformerModel, adds positional encoding before it runs it through an encoder stack (`nn.TransformerEncoder`). The final output has the same shape as the output of the encoder part of the TransformerModel.

The discriminator has almost the same structure as the generator (it uses positional encoding and an encoder stack), but the input differs and it outputs a single number between 0 and 1. Recall, that the job of the discriminator is to decide wether its input was generated from random noise via the generator or if it was an encoding stemming from a real sentence passed through a trained encoder. Thus, it is effectively a binary classifier, and its output should describe its conviction that the input is 'true'.

3.3 Training

4 Analysis

4.1 Results

4.2 Discussion

5 Further research

6 Conclusion

References

Chintapalli, Karthik (2019). *Generative Adversarial Networks for Text Generation*. URL: https://becominghuman.ai/generative-adversarial-networks-for-text-generation-part-1-2b886c8cab10 (visited on 08/13/2020).

Donahue, David and Anna Rumshisky (2018). "Adversarial Text Generation Without Reinforcement Learning". In: arXiv: 1810.06640 [cs.CL].

Goodfellow, Ian J., Jean Pouget-Abadie, M. Mirza, B. Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio (2014). "Generative Adversarial Networks". In: *ArXiv* abs/1406.2661.

Haidar, Md. Akmal and Mehdi Rezagholizadeh (2019). "TextKD-GAN: Text Generation using KnowledgeDistillation and Generative Adversarial Networks". In: arXiv: 1905.01976 [cs.CL].

Inkawhich, Nathan (2020). *DCGAN Tutorial*. URL: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html (visited on 08/13/2020).

Kusner, Matt J. and José Miguel Hernández-Lobato (2016). "GANS for Sequences of Discrete Elements with the Gumbel-softmax Distribution". In: arXiv: 1611.04051 [stat.ML].

Robertson, Sean (2020). *NLP From Scratch: Translation with a Sequence to Sequence Network and Attention*. URL: https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html (visited on 08/13/2020).

Sequence-to-Sequence Modeling with nn.Transformer and TorchText (2020). URL: https://pytorch.org/tutorials/beginner/transformer_tutorial.html (visited on 08/13/2020).

Silveira, Natalia, Timothy Dozat, Marie-Catherine de Marneffe, Samuel Bowman, Miriam Connor, John Bauer, and Christopher D. Manning (2014). "A Gold Standard Dependency Corpus for English". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need". In: arXiv: 1706.03762 [cs.CL].

Yu, Lantao, Weinan Zhang, Jun Wang, and Yong Yu (2016). "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient". In: arXiv: 1609.05473 [cs.LG].