Adversarial Text Generation

NLP and Deep Learning — Final Project

Andreas Holck Høeg-Petersen
anhh@itu.dk

Mathias Bastholm
mbas@itu.dk

August 14, 2020

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 **Goodfellow2014GenerativeAN**.

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.

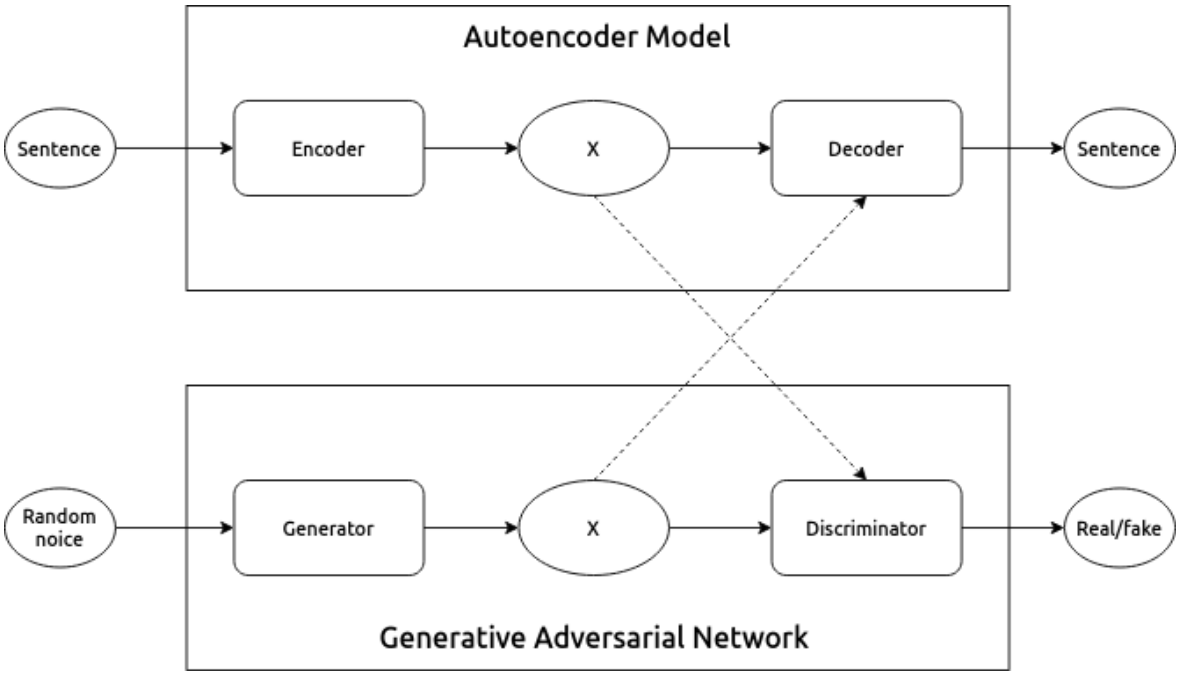

Figure 1: Overview of the model architecture. The dotted lines from the **X**s represents that the encoded and generated **X**s will be fed to the discriminator and the decoder during training and evaluation, respectively.

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'. The second objective is to build a GAN network, where a generator — the 'Student' — must learn to produce approximations of $\mathbf{X}$.

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. We will then describe how we have build the different parts of the model and how we utilize our dataset. Then we will present our results and discuss the shortcomings of the model*s*, and proceed to suggest improvements and ideas for further research. Lastly we provide a conclusion on the 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.

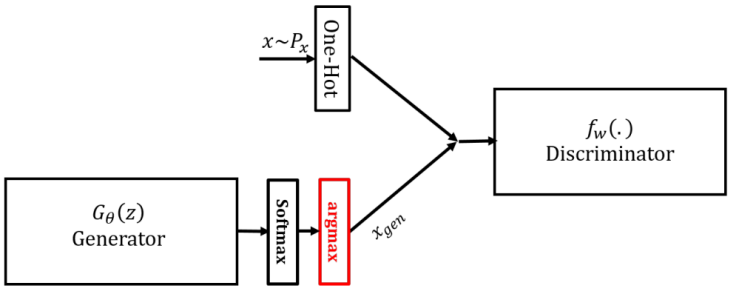

Figure 2: A simple GAN model where the generator output is run through an *argmax* function before being given to the discriminator. This prevents gradients to flow from the discriminator to the generator. Source: **haidar2019textkdgan**.

There have been, however, multiple attempts at working around this issue. According to **Chintapalli2019** 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

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, **yu2016seqgan** proposes the SeqGAN model, in which the generator is considered an RL-agent with states $\mathbf{s}_t$ 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 **kusner2016gans**. 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 $\tau \to 0$ 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 $p_i = p(y_i = 1), i = 1 \dots d$.

Finally, for other examples on the third approach, see **donahue2018adversarial** and **haidar2019textkdgan**.

3 Method

For all our models (autoencoders and GANs), we drew inspiration PyTorch tutorials (**pytorchTutorialAtt**, **pytorchTutorialTransformer**, **pytorchTutorialGAN**), 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' (**silveira14gold**) 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 80/10/10 between training and training/validation/testing. The dataset had 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 **pytorchTutorialTransformer**. 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 **vaswani2017attention** 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 (**vaswani2017attention**).

This model also uses an embedding as its first layer. For embeddings, we use pretrained word vectors from the `polyglot` Python package. These have an embedding size of 64, which therefore what we use across all models. Note that the next, RNN-based autoencoder uses the same embedding setup.

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 **pytorchTutorialAtt**.

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

Our training had two aspects: first, we had to train our autoencoders on English-to-English sentence pairs, and then we had to train our GAN model. This section will describe both.

3.3.1 Training the autoencoders

This part is pretty straight forward. We give the model a batch of sentences which it then encodes and decodes again. Since the input sentence is also the target sentence, the loss is simply the difference between the input and the output. To calculate this loss, we use PyTorchs 'nn.CrossEntropyLoss' which takes the raw output of the model (ie. before it has been converted to a probability distribution) and a target vector and then calculates the negative log-likelihood loss. We use Stochastic Gradient Descent to update the weights of the models.

There were some notable differences between the way we trained the TransformerModel and the RNN-based model. For the TransformerModel we had good results with a learning rate of 0.05, and we performed training on the entire training set in batches of 8 and over the course of 100 epochs. We also trained it another time with just 10 epochs, but this time without ignoring the padding in the batches (when batching the data, not all sentences have the same length and the smaller sentences are padded with the special `<PAD>`-token). This makes the model achieve a lower loss much faster, as it quickly learns the pattern of the padding tokens, but this often comes at the expense of the model learning the actual sentence, which generally is what is of interest. However, we wanted our Student model to learn both from a Teacher that had learned padding and one that hadn't, so we trained the autoencoders (Teacher) on both.

In training the RNN-based model we took inspiration from **pytorchTutorialAtt**. This means we set the learning rate to 0.01 and instead of training on the entire training set over multiple epochs, we let the model randomly pick a data batch, train on that and then pick a new for 60,000 iterations. Furthermore, we used teacher forcing, where instead of letting the output of the decoder be its own next input, we give the actual next input (from the target sentence). This approach can lead to some terrible cases of overfitting and preventing the model form learning its mistakes properly, but in the

right dose it can also help the model converge faster. Therefore, for each training instance, we chose with 50% probability wether to use teacher forcing or not.

3.3.2 Training the GAN model

The training of the GAN model is the interesting and difficult process. Here, we have to simultaneously train two different models that work towards opposite goals. One, the generator, will have to learn how to maximise the probability that the discriminator classifies its output as 'true', while the other, the discriminator, will have to learn to maximise the probability that it classifies the output of the generator as 'fake' while also classifying the output of the Teacher (encoder) as 'true'. So the two components engage in a min-max game with one another.

Formally, let G and D be the generator and the discriminator respectively and let x be the data representing a sentence (encoded or generated). Then, $D(x)$ is the output of the discriminator given some data, and we want this to be high when x is 'true' (stemming from the Teacher) and low when x is 'fake' (generated by the Student). Let z be some random noise, and then we have that $G(z)$ is the output generated by the Student given some random noise. So, the discriminator now tries to *maximize* the probability that it correctly classifies 'true' and 'fake' data, $\log D(x)$, and the generator tries to *minimize* the probability that the discriminator classifies its outputs as 'fake', $\log(1-D(G(z)))$. As described in the original GAN paper **Goodfellow2014GenerativeAN**, this can be formalised as:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(x)}\left[\log(1-D(G(z)))\right]$$

where p_{data} is the 'true' distribution over the real data which the generator tries to estimate, and p_z is the estimated distribution that the generator draws its samples from.

The way we do this is by following the approach suggested by **Goodfellow2014GenerativeAN** and PyTorch GAN tutorial for image generation **pytorchTutorialGAN**. This involves first giving the discriminator a batch of 'real' input (ie. input generated by the Teacher) and then calculating the loss and the gradients based on its performance. Then we give it a batch of 'fake' data generated by the Student, again calculating the loss and gradients. Then, before we move on to training the generator, we perform our optimization step for the discriminator. When we then pass the output of the generator through the discriminator, the discriminator has been updated and the generator will have to learn to adjust to this.

However, one could imagine that the generator could learn to just output random words very well and that the discriminator would not be able to distinguish this from a 'real' sentence. Therefore, before training the generator, we also give the discriminator a batch of 'fake' data that has been generated by the encoder, but when the encoder have gotten random words as input. This, in theory, should force the generator to produce meaningful sentences (not just random words) for it to be classified as 'true' by the discriminator. This adjustment is specific to our approach.

It is worth noting, that training a GAN is known to be highly non-trivial (**Hui2018**) and even small variations in the hyperparameters, number of training iterations or internal structure of the generator and/or discriminator can make the model collapse spectacularly.

4 Analysis

In this section, we will present and discuss our results as well as give suggestions for further improvements and research.

4.1 Results

4.1.1 TransformerModel

Plots of loss during training.

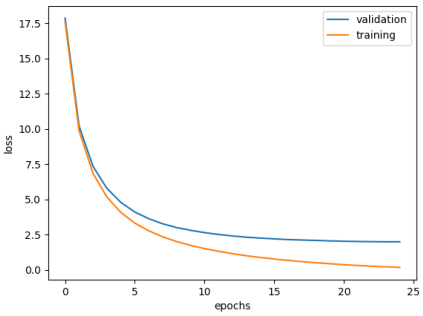

Figure 3: Loss of the standard TransformerModel

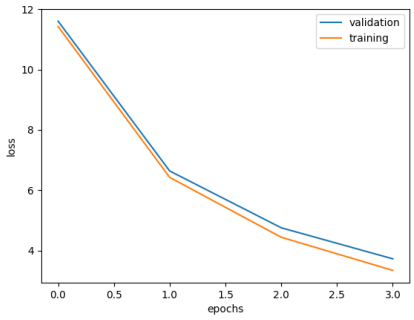

Figure 4: Loss of the TransformerModel that takes padding into account and has EOS tokens

The TransformerModel ends up with a word accuracy of 0.99, here are some test examples:

```
++++++ Successful encoding ++++++
> he can play the piano , the flute , the guitar , and so on .
< he can play the piano , the flute , the guitar , and so on .

> i have to take the entrance examination today .
< i have to take the entrance examination today .

++++++ Failed encoding ++++++
> i'm sure it wouldn't be too hard to find out who hacked into our system .
< i'm sure it wouldn't be too hard to find out who weakly into our system .

> you have the right to free speech , but not the right to slander .
< you have the right to free speech , but not the right to limits .
```

4.1.2 RnnModel

Test examples:

```
> tom got to the station too late so he missed the train . <EOS> <PAD>
< tom got to the so so so he opened the same . <EOS>

> i don't want to waste my time trying to do this again . <EOS> <PAD>
< i don't want to take my job to do me it . <EOS>

> if the car is gone , he can't be at the office . <EOS> <PAD>
< if the book is he , he was late than the . <EOS>

> i can't believe that you were the smartest kid in your class . <EOS> <PAD>
< i can't believe that you were a a the in the . . <EOS>
```

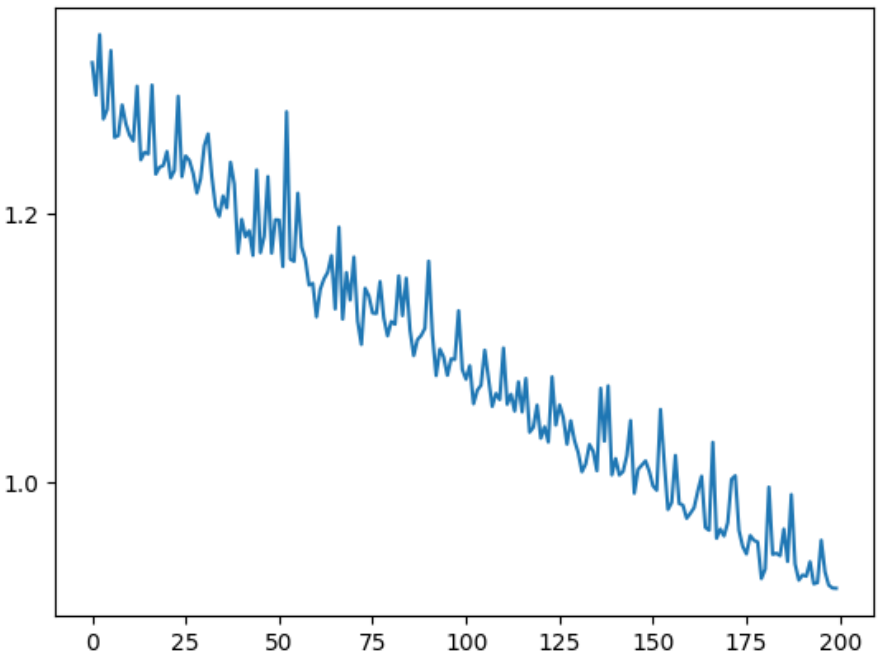

Figure 5: Loss of RnnModel while training

4.1.3 GAN

Output of the model with the highest $D(G(z))$ while still having a $D(x) > 0.35$:

```
one wish a hot next suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious subtitled
one wish a quite next suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious subtitled
one wish to hot next suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious subtitled
one wish to hot states suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious suspicious subtitled
```

Output of the model at the end of training:

```
he lazy lazy married married . cholesterol suspicious subtitled suspicious mercury mercury mercury mercury scientist
he a doesn't married married . suspicious suspicious suspicious suspicious mercury mercury mercury mercury scientist
he lazy lazy married married . suspicious subtitled suspicious suspicious mercury mercury mercury scientist scientist
he america lazy married married . cholesterol suspicious suspicious mercury mercury mercury mercury scientist scientist
```

As can be seen the model tends be rather static in its output, as such here are some sentences generated by the model while training:

```
are thank new to you . suspicious suspicious suspicious prevents suspicious mafia mafia mafia subtitled
something troll me watched watched . suspicious suspicious suspicious suspicious mafia mafia mafia mafia mafia
your ignored delayed delayed . helpless helpless helpless helpless helpless helpless suspicious helpless helpless helpless
it's , , him looking expect . . housewarming housewarming housewarming suspicious mercury mercury mercury
for change but do i . suspicious suspicious suspicious suspicious suspicious suspicious suspicious mafia mafia
i mohammed convenience play the . suspicious suspicious suspicious suspicious suspicious mafia mafia mafia mafia
he used was donated money . subtitled suspicious suspicious suspicious suspicious suspicious suspicious scientist scientist
```

4.2 Discussion

These are notes:

The GAN network is actually able to learn. Sentences differ from random input. Increasing the batch size of z might help the generator. The training breaks too soon Discriminator gets to critical

'double_batch' means that the generator is given a double sized batch for training.

When we train the discriminator on encoded random words, $D(x)$ should be 0.5 for a perfect D (because half of x is 'true' and the other half is 'fake'). When omitting the encoded random words, $D(x)$ should be 1. Of course, for a perfect network with a perfect D and a perfect G, both $D(x)$ and $D(G(z))$ should be 0.5

Training without encoded random words seem to make the generator take longer to learn what to do with padding and EOS. However, it also seems to make the network more stable.

The two output sentences in some of the GAN-training-output files are first the output of G given the fixed noise and then the output given random noise. Not how they become almost identical.

SGD seem to make the network more stable, but it also tends to end up with small sentences of identical words and a LOT of padding.

Training with no encoded random words went fine for very long, with the generator actually being consistently better than the discriminator — until it the hit a giant collapse (see `../examples/GAN-training-output-double_` and 6 for evidence)

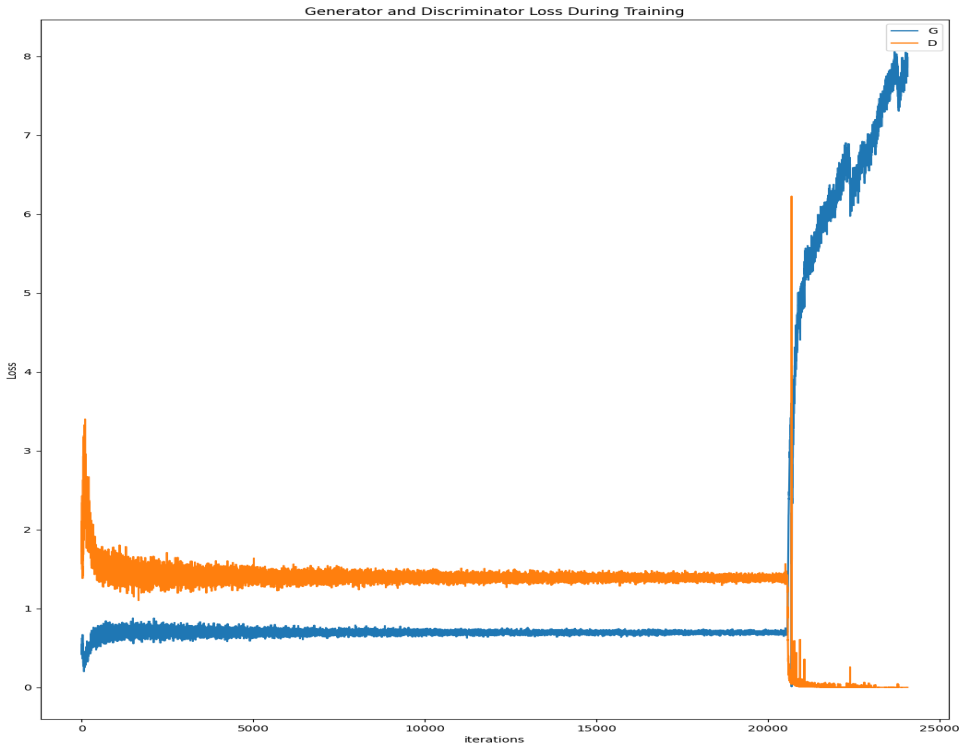

Figure 6: GAN loss for setup with Adam, double batch and no encoded random words.

5 Further research

6 Conclusion