Generating text using LSTM GAN in PyTorch

732A92 Text Mining

Martin Beneš [marbe619]

2021-01-16

Abstract

The aim of the project is to generate text using generative adversarial networks (GANs), where both components are recurrent/LSTM neural networks. The training data set is a collection of internet comments from YouTube, Twitter and Kaggle also containing a (binary) negative sentiment indicator. The model output is evaluated using the result of Markov chain trained on the same data. Absolute quality of the text is evaluated manually against a test set. Sentiment input integration into LSTM GAN is discussed.

Introduction

Text generation is one of the common tasks in text mining and natural language processing (NLP). This project focuses on usage of generative adversarial networks (GANs) in the text generation. GANs are usually used for image data, in that case their components are CNNs, for working with text, RNN architecture should be used.

The second assumed model is a Markov chain model, a simple model that can perform satisfactory on a sufficiently big training dataset. Predictions of the model are then classified by the discriminator to directly measure its performance and thus indirectly evaluate the quality of the text generated by the GAN. Absolute quality of the generated text is also evaluated manually by a human using a labeling of fakes amongst the real comments.

Each training sample contains a flag, marking binary presence of negative sentiment in the text, such as racism or attack. Even though this information is not utilized in the models in any way, its possible usage as an additional input to the generator network is discussed.

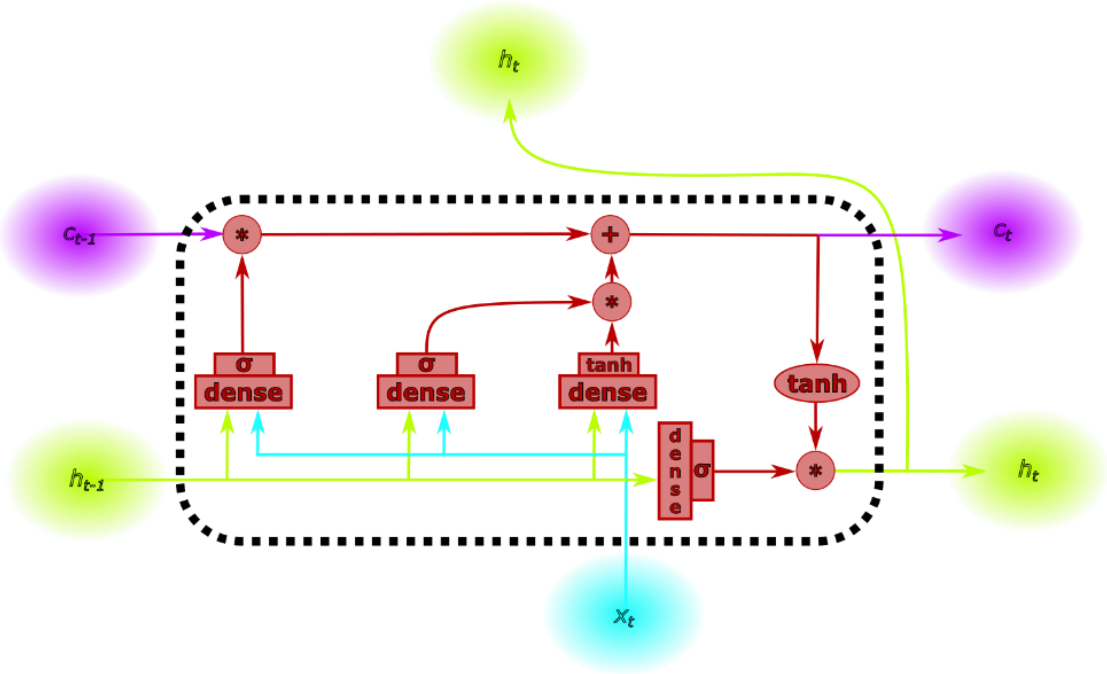
Theory

In the statistical approach the generated text is prediction of a certain model, fitted to train data. A kind and quality of the training set and the chosen model are two important factors for the quality of the generated text.

Recurrent neural network (RNN) is a deep learning model for sequential data, such as time series or natural language, whose elements are mutually related and order dependent. RNNs are sequence length independent. The inputs of the network for each time step t are the network input x_t and internal network state s_{t-1} of the previous time step $t-1$, the outputs are the network output y_t and internal network state s_t utilized in the next time step. The weights to be trained on data are U (transforming $x_t \rightarrow s_t$), W (transforming $s_{t-1} \rightarrow s_t$) and V (transforming $s_t \rightarrow y_t$). The f is a non-linear activation function.

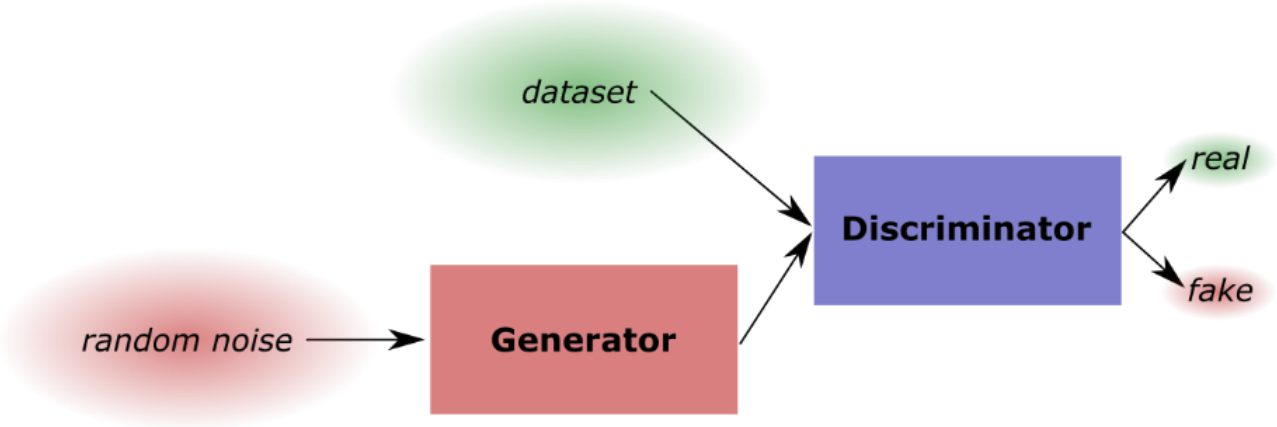
$$s_t = f(s_{t-1} * W + x_t * U)$$
$$y_t = s_t * V$$

One of the problems in deep learning is vanishing/exploding gradient, the numeric instability during the backpropagation. An alternative to RNN solving this issue is using the LSTM architecture of a neuron, which contains a memory cell improving numerical stability. (Vasilev I. 2019)



$$c_t = c_{t-1} * \sigma\big(W_f x_t + U_f h_{t-1}\big) + \sigma(W_i x_t + U_i h_{t-1}) * \tanh(W_c x_t + U_c h_{t-1})$$
$$h_t = \sigma(W_o x_t + U_o h_{t-1}) * \tanh(c_t)$$

Generative Adversarial network (GAN) is very popular model for data generating. Its results, forged images and videos called *deep fakes*, are well-known even by the lay public, but it can be used for generating any kind of data. GAN framework consists of two networks: generator G and discriminator D, two networks reciprocally training each other. G produces data as similar as possible to the training dataset, as an input it uses random noise. It is trained using D, that learns to classify the generated samples shuffled among the training samples (binary label true/fake). D is backpropagated, then labels for G are produced from D predictions, the misclassified samples are labeled as correct. This training cycle is repeated until the losses of G and D converge, same as in case of single network training. (Goodfellow I. 2014)



The second assumed model for comparison and performance evaluation is a Markov chain model, based on principles of Hidden Markov model (HMM). Training the model means constructing the *next-word* distribution for each word of the vocabulary or transition matrix in terms of Hidden Markov models. There are several methods for simulation from the Markov chains: filtering, smoothing and Viterbi algorithm. Filtering uses only samples prior the current time step t of simulation $(0:t)$, smoothing and Viterbi algorithm use all the samples $(0:T)$, the latter is the only one producing valid output according to the transition matrix. (Fraser 2008)

Sentiment is an emotion or feeling contained in the text connected with an opinion of the author. Categories of sentiments are connected to the human emotions, they might be oriented positively, neutrally or negatively and a certain intensity. Sentiment might have an emotional or rational origin. The sentiment analysis is a task to discover sentiments and through them opinion of the author. This opinion is the reason for the observed sentiments. (Cambria E. 2017)

Data

The dataset *cyberbully* was published on figshare.com as part of online *datathlon* Data Sprint #13: Cyberbullying held by *DPhi* community. The data collection consists of 8 csv files of overall size 171 MB, containing user comments from various social network platforms like Kaggle, Twitter, YouTube and Wikipedia Talk. The data contains text and binary flag marking the presence of cyberbullying or sentiment in terms of text mining. Cyberbullying aspects include behavior such as hate speech, aggression, insults and toxicity. (Elsafoury 2020)

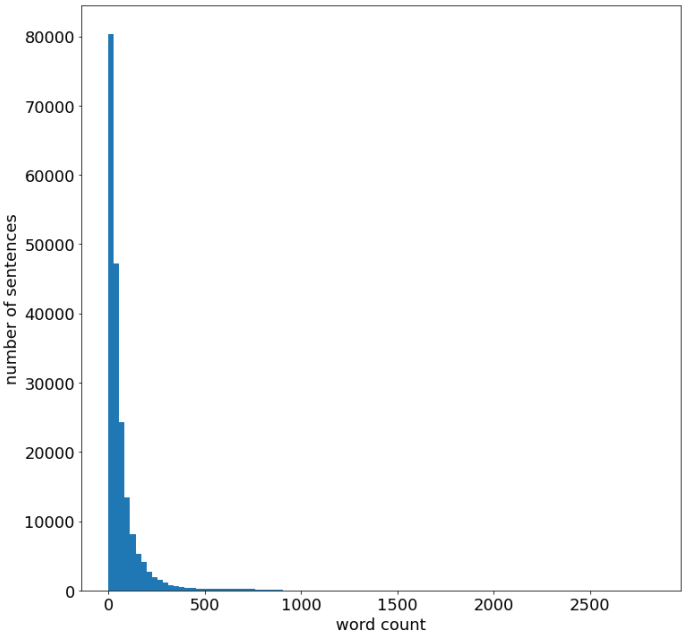
The data files are downloaded and merged as a single format. The samples are deduplicated with text equality since the files have heavy overlap. All sentiment flags are unified as a single binary cyberbully-present flag. Then each text is tokenized and stored in the file `data/words.txt`. This data-engineering step takes about 45 minutes and is being done separately in `src/fetch.py` and executed out of the training pipeline.

File `data/words.txt` contains 196292 distinct comments and 3 attributes: text (stringified list of words), label (0/1 sentiment flag) and source (twitter/youtube/kaggle/unknown). An example line of the file

```
"['able', 'to', 'do', 'better', 'than', 'that']",0,unknown
```

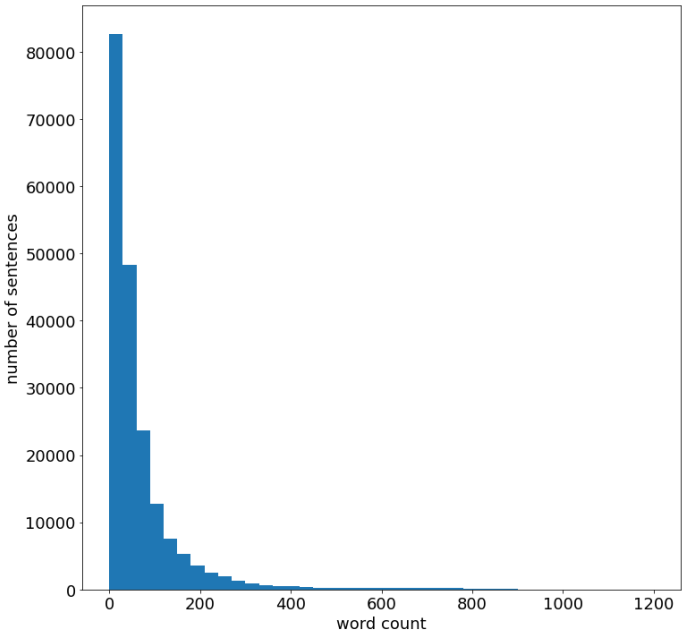
Right before the training a mapping using embedding is done, a projection of the text to a numeric tensor (e.g. vector or matrix). This paper uses three different methods: scalar embedding, word2vec and Bert. The former two introduce the same computational issue: to construct the models (especially Discriminator), the length of the sentences should be constant and the insufficiently long sentences are padded with special <*not-a-word*> token.
However the distribution of dataset has several extremely long samples (the longest is 2832 words long), while most of the sentences have at most several hundred words. Padding brings a gigantic overhead for the training data storage. Thus extreme sentences are dropped in order to normalize the seq_len to a bearable value; if sentences longer than 1200 words are ignored, memory requirements and computational complexity is drastically reduced.

Sentences of all lengths



$196292 \text{ samples} \cdot 2832 \text{ words} \cdot 8 \text{ byte } [int64] = 4.447 \text{ GB}$
196292 samples
14088416 words

Sentences longer than 1200 dropped



$196292 \text{ samples} \cdot 1200 \text{ words} \cdot 8 \text{ byte } [int64] = 1.884 \text{ GB}$
195987 samples (0.155 % dropped)
13616666 words (3.348 % dropped)

If different mapping is used (e.g. 100 feature vector per word), the memory requirements are even greater. For instance embedding with word2vec strategy amplifies this problem in relation with used $input\_size$, its output dimension. Thus, embedding vector mapping is computed in each iteration to avoid massive memory usage. This way each batch is passed through embedding mapping num_epochs times.

Method

For documentation of the data and the architecture of the models several parameters is used. The longest sentence in the training dataset is $seq\_len = 1200$ words long and the training dataset contains $vocab\_size = 305290$ words. The $input\_size$ differs for each embedding strategy: for the scalar incremental it is 1, for word2vec it is 12 and for Bert it is variable based on the text (cropped to 1200) (**update**). Network architectures are partially parameterized with

$hidden\_size = 20$ and $num\_layers = 2$ for both Generator and Discriminator, the dropout probability for D $dropout\_prob = 0.5$. Training configuration includes batch size $batch = 500$, learning rate $learning\_rate = 0.0005$ and the number of epochs $num\_epochs = 5$.

As mentioned in the previous section, this paper uses three different methods: scalar embedding, word2vec and Bert. The scalar embedding supposes that words can be simply projected to a single number – this in general does not project any semantical similarities into the embedding representation and ptus all responsibility on the neural network to learn as much as possible from the dataset.
Second embedding strategy uses pretrained `word2vec` model from `gensim` package with vector length 12. This model has trained the vector representation for the words on a many various datasets and hence projects the similar words closer to each other (in cosine distance). Disadvantage of this mapping is fixed vocabulary. (Mikolov 2013) Walkaround for this bottleneck is "closest word" matching using string matching from *rapidfuzz.*
A strategy not having the property of constant vocabulary size is Bidirectional Encoder Representations from Transformers (abbr. Bert), which as word2vec uses Transfer learning. In addition, unlike the two embeddings before, Bert works on a sub-word level: it does tokenize the text based on its vocabulary of words and sub-words, not based on the spaces. This representation is helpful in such cases as the data used in this paper, since internet communication is very specific - informal words, emojis, slang/argot, etc. A pretrained model `bert-base-uncased` from `transformers` package was used. (McCormick 2020)

Generator neural network input and output shapes are $batch\_size \times seq\_len \times input\_size$. First two layers are LSTM, making the Generator independent on the $batch\_size$ with hidden size of $hidden\_size$. Output of the last LSTM is flatted along the single sentence and fed to output dense layer with $hidden\_size * seq\_len$ neurons and output $input\_size * seq\_len$, returned reshaped to $batch\_size \times seq\_len \times input\_size$.
Discriminator neural network input is $batch\_size \times seq\_len \times input\_size$ and output $batch\_size \times 1$. Its structure is fairly similar to the Generator's. Two LSTM layers of the same shape are in Discriminator followed by dropout, after them follows a single hidden dense layer with dropout and ReLU activation function and $hidden\_size \times seq\_len$ nodes. The final output dense layer has $hidden\_size$ nodes and an output size 1, activated by sigmoid to produce value between 0 (fake) and 1 (real). This final layer brings the requirement for constant seq_len and is the reason why padding alignment had to be used in the training data. (Liu 2020) (Mosquera 2018) (Inkawhich 2017) (Robertson 2017)

The reversed mapping of the generator output to text can be done in several ways. Deterministic approach chosen in this paper consists of producing a constant mapping based on training data and then for each vector (word) produced by generator seeking the closest word vector in the training data. In case of scalar incremental mapping the output is rounded on the closest integer and a direct dictional lookup is used. Reversed embedding layers are implemented for each embedding strategy in file `src/embeddings/rev.py`.
In recent years Bert has been used for text generating with stochastic reversed mapping by sampling from the distribution over vocabulary, output of an extended Bert framework model (implemented in `BertForMaskedLM` in *PyTorch*), extending generative models by another sub-word model option. (Wang 2019)

Markov Chain model can be understood as a special case of Hidden Markov model, where the latent space is vocabulary of the training set and the emission matrix is the identity of appropriate shape. After experiments with library `hmmlearn`, built on scikit-learn, it was discovered that `MultinomialHMM` is not suitable for such usage in text mining, because transmission matrix is implemented as a NumPy matrix a.k.a. dense matrix. Hence, memory requirements are $O(V^2)$, where V is vocabulary size, in case of the training set *cyberbully* $V = 305290$, $O(V^2) = 93.2\ GB$ and grow quadratically.

Instead, a custom Markov Chain model with sparse transition matrix was implemented as a class `MarkovChain` in the file src/markov.py. For each sentence, let us denote its i[th] word as w_i, number of its occurrences $\#w_i$ and a bigram of words w_i and w_{i+1} as $w_i w_{i+1}$. Then the probabilistic definition of the implemented Markov Chain model is

$$w_0 \sim Multinomial\left(V, \pi_i = \frac{1}{V}\right), i \in \{1, \dots, V\}$$
$$w_{i+1} \sim Multinomial\left(\#w_i, \pi_i = \frac{\#w_i w_{i+1}}{\#w_i}\right), i \in \{1, \dots, V\}$$

Results

The Markov Chain model yielded results saved in `output/markov.txt`. For a human, the sentences do not make sense – the only dependence is a probabilistic measure of following word, which makes the generated sentence changing the point every few words for a human reader.

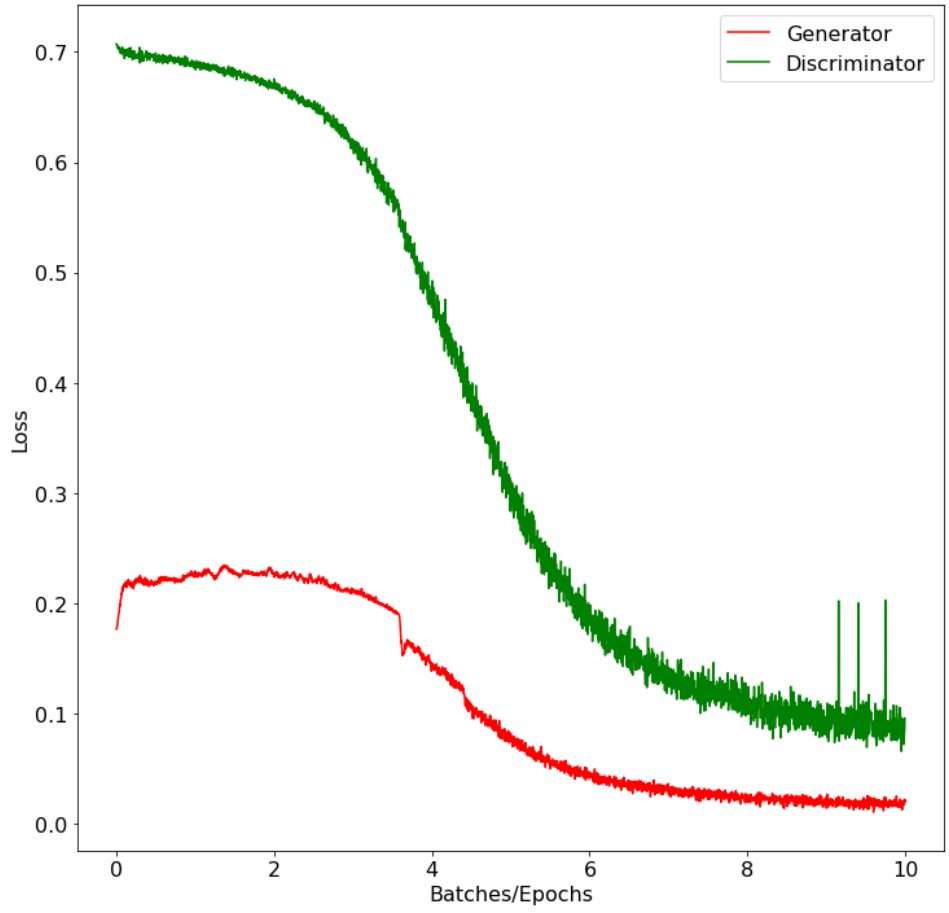
```
capetown and figuratively to statements properly as part of VandalProof SpK. as sarcasm is essential condition is denied
```

```
remainings in Western Australia and the Omagh bombing of something Wikipedia poster show actual 10-string guitar
```

```
abisharan talk page if they have little insignificant instances of vandalism work on WSD closed minded
```

Scalar incremental model using embedding mapping [word → unsigned] of the input text data for the first trained model was implemented with a simple incrementation method "*as word occurs*".

["Who am I?", "You are who you are."] → {"who": 1, "am": 2, "i": 3, "you": 4, "are": 5}

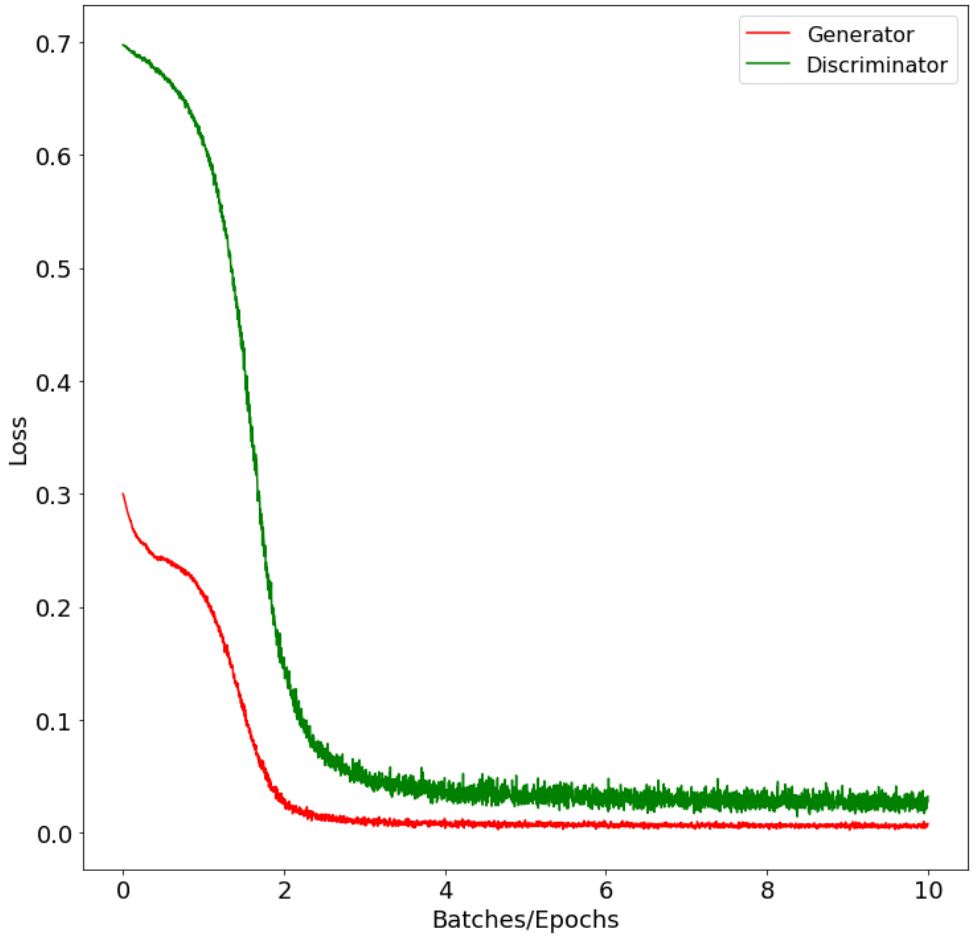


The plot above shows the losses (prediction errors) of Generator and Discriminator during training. To report a numerical performance, the table below contains measured errors on train set and test set representing the real data and the Generator output and the Markov chain generated text, considered fake data.

Results of the model with scalar incremental embedding				
	Train data (real)	Test data (real)	Generator (fake)	Markov chain (fake)
Accuracy (threshold 0.5)	99.97%	99.96%	99.55%	0.6%
MSE (soft output)	0.001465	0.001367	0.004506	0.963306

The model outputs are values below 0.5, so after rounding, the model just returns placeholder index to the embedding mapping used for words out of vocabulary for both normal and integer uniform input. Thus there is no measurable output of the model.

Closest Word2Vec model was trained with the same training parameters as the model before: $batch\_size = 500$ and $num\_epochs = 10$. The plot for the training with losses of Generator and Discriminator is shown below.



The model performance on test data, generator output and reference Markov chain are shown in the following table.

Results of the model with word2vec embedding				
	Train data (real)	Test data (real)	Generator (fake)	Markov chain (fake)
Accuracy (threshold 0.5)	100%	100%	99.64%	0%
MSE (soft output)	$\sim 3.3 \cdot 10^{-6}$	$\sim 3.4 \cdot 10^{-6}$	0.006	0.995877

After rounding the Generator output is almost the same for any random input. Output of the model is shown in the file `output/word2vec.txt`. Two sentences to illustrate generated by the model start with

```
moniter moniter moniter moniter 3ds bahasa opportune lillte aloud 3ds bahasa monitor …
```

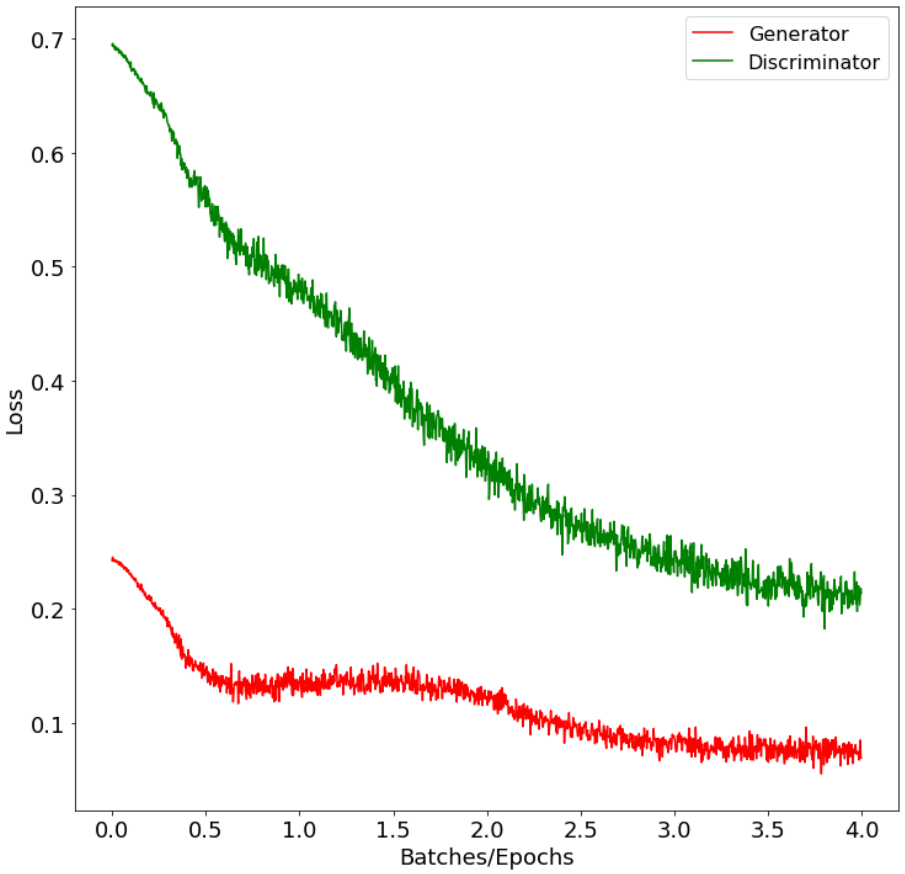
```
moniter moniter campaignbox moniter 3ds moniter opportune lillte aloud 3ds bahasa …
```

Further on in the sentence following segments of text were observed

```
having perfectly poorly needs widely inconvenient largely neckline heres show distracted bias
```

```
starting ijustine consistently intends never counter invading implies single swastika reposted newly
```

Bert embedding was trained due to greater resource demand on the model was trained with parameters $batch\_size = 500$ and $num\_epochs = 4$.



The results of Discriminator classification on train and test sets, Generator output and Markov chain text are shown below.

Results of the model with Bert embedding				
	Train data (real)	Test data (real)	Generator (fake)	Markov chain (fake)
Accuracy (threshold 0.5)	99.95%	99.97%	100%	0%
MSE (soft output)	~0.001118	0.001034	$\sim 5.7 \cdot 10^{-5}$	0.963456

Output of the model is shown in the file `output/bert.txt` with examples below.

```
daze millimeters hyperbolic effacing philips harz c02 okie 1586 madhi busby dealio purut hemoglobin

lithuenian torvalds whitish serb spectrum 131 hatredbut powns heera tesc subsubsection moors ryukyu

prussia raf anz pointd hinges lows admirably zetas quarry bartles °0° investigated counter ajk sniffling ovens
```

Discussion

Scalar incremental model

Evaluating purely based on the discriminator accuracy, the model with scalar incremental embedding distinguishes very well between the generator output and the real data and because for the estimation we are using test data and different output of the generator, the estimation should come with higher reliability of the measured performance.
Loss plot shows that Discriminator has for the whole training much higher loss and both curves decrease over the training. Discriminator curve seems to be quite smooth while Generator curve contains several abrupt changes. The model converges within several episodes. Based on these data only we could postulate that the model generalized above the training dataset well from these. Even though the words get (theoretically) random numbers assigned by the embedding, the GAN flexibly trained to distinguish real data from the ones produced by it.
However, regarding the performance on independent text generator, we observe that the almost all artificially produced samples from Markov Chain model were classified as real. From this it may be assumed that the discriminator considers the samples to be closer to the true data than what Generator is producing or that the Discriminator relies on features contained in the Generator output, that are not present in the output of the Markov Chain generator. The model fails to extrapolate ability to distinguish the fake data out of the Generator output, which might imply that the Generator output is not of high quality.
In addition, Generator produces only small numbers on output. Since the output is rounded to get the index of the result word, this means the output of the Generator is completely invalid, because 0, the generated index for any input, is a placeholder for the “out-of-vocabulary” words.

Possible alternations to the scalar incremental model could be assigning closer numbers to words similar by meaning, but such seriation would be computationally intensive when vocabulary gets great, the exhaustive list of all possible permutations has size $V!$ for vocabulary of size V.
All in all, the model performs very bad and it is most likely due to the inappropriate embedding that this model fails to generate any output whatsoever. By resources this is by far the least complicated model, thus fastest and easiest for training and implementation, a single epoch takes only couple of minutes to run on Cuda GPU.

Closest Word2Vec

The discriminator of the second model has very similar statistics with the first model. In this case the model seems to be even more fitted and decided about the data – an overfitting is a possible explanation, however the performance on the test data is as good as training data so it is unlikely. All Markov chain samples were classified incorrectly, the discriminator considers the Markov output to be closer to the true data than what Generator is producing or that the Discriminator relies on features contained in the Generator output, that are not present in the output of the Markov Chain generator.
Loss plot has several differences against the first one: the variance of the loss seems to be lower and the convergence comes much faster, basically within first 3 episodes - not necessarily a bad sign, but surely an artifact to point out, given that the model is more complex than the first one.
Generator produces a reasonable output mapped to the vocabulary of the embedding; the resulting text is probably of even lower quality that the reference produced by Markov Chain. The words seem to be random and there is no evident connection between the words in the sentence. Also, on any input Generator produces very similar output, which is a sign

of mode collapse: a state where Discriminator gets trapped in a local minimum during training and Generator outputs start to be more and more similar. There are several workarounds for this problem, such as using Wasserstein loss or unrolled GANs. (Google Inc. 2020)
The first words occurring in the output shown together with their vector representations are shown in the table below. They seem to be quite close together in cosine metric (since their values in many dimensions are quite close), although this might be a pure matter of chance.

Words and their vector represenation from output of Closest Word2Vec model												
moniter	0.122	−0.17	0.197	−0.291	0.036	0.202	0.36	0.311	0.137	0.26	−0.156	−0.102
campaignbox	0.315	0.056	0.192	−0.154	−0.24	0.254	0.27	0.48	0.221	0.355	−0.097	−0.242
3ds	0.046	−0.16	0.115	−0.106	−0.09	0.144	0.446	0.281	−0.1	0.192	−0.328	−0.028

For practical reasons $input\_size = 12$ was used, with larger vector dimension the results of the algorithm could become slightly more feasible. An interesting amendment to the GAN framework would be additional fake data sources, such as various generative models, which could make the framework better extrapolate out of the Generator output; from the models we could name GAN instances fitted to different datasets or reinforcement learning generators, based on syntactic learning. (Chen 2018)
In general, the model fails to generate any reasonable text, its output does not make any sense. It suffices from training artifacts observed by researchers in GAN framework, but even though unlike the first model, the output is at least interpretable into words. A single training epoch takes about 45 minutes on Cuda GPU.

Bert
The dataset cyberbully contains data that are quite messy when a word separation by spacy is used. There are many misspelled words, informal words and thus the closest word matching might have been a reasonable solution. Alternative solution is using sub-word embedding Bert.
The results of the discriminator performing on all four data is fairly similar to both models evaluated before. The accuracies for true data, train and test sets, are incredibly high and same for Generator output, a fake data source. As in the previous cases discriminator classifies almost all the Markov chain produced samples as true.
The loss plot is much slower and linear-like, although there is less epochs to produce, so the visual perception might be skewed by this. The first epoch the training is faster, then the descend pace slows down. During second period, the losses even increase a bit in a certain point.
The text produced by the Bert embedding seems to be of low quality, although the words are changing and are not constant. A lot of words in the output are words that are rare in the training set and thus the model did not even take into consideration the prior probabilities (ratio occurrences) of the single words in the train set. There seems to be no connection between neighboring words and overall the model fails to predict a reasonable output, although it is the only GAN output that has been successfully produced. A training epoch on this network with reduced size takes about twice as much time as the Closest Word2Vec, 90 minutes on Cuda GPU.

Bert was a matter for experiments recently. When using the stochastic approach for the revers embedding, one can reach truly impressive results. In the work Bert object from transformers library is often used just as a standalone model, while this paper constructs and trains the model manually. (Wang 2019)

For better results in generating, the model used for generating text could be trained on both train and test data after the misclassification is reported by this split, however due to lack of computational resources only single model for each embedding was trained and used over the whole task.

As each sample of the cyberbully dataset contains a sentiment label, it is reasonable to suggest how would it be possible to add a sentiment input to the generator to be able to control the output of the generator, choosing what output is produced regarding the sentiment. The easiest option would be to train different models for different label values, a valid approach although it significantly reduces the amount of training data with each possible sentiment added. For many different sentiments being distinguished or for overlapping sentiments (rather tags), we thus need to seek a different way. Another option could be adding this path into the architecture: there were proposed alternative structures to GAN that are able to produce conditional output, such as conditional GAN (C-GAN), Variational Autoencoders (VAE) or SentiGAN. (Mirza 2014) (Kingma 2014) (Wang and Wan 2018)

Conclusion

According to the experiments, embedding strategy is an important part of the modelling and a wrong decision ends up in unsatisfactory results. The experiments to produce a sufficient output were not successful for any of embeddings. The only model producing output is Bert embedding, still worse than reference output of Markov Chain.

GAN almost always learns to successfully distinguish between the real data and the data produced by generator; however, accuracy of discriminator is not related to quality of the output produced by generator. Measurement of discriminator performance on fake data produced by different generative model is reasonable, but due to unsatisfactory results of all three models, it has not been proven how this score corresponds with the quality of generated text. However, data produced by different generators could extend the GAN framework as additional fake data source and make GAN better extrapolate out of the Generator output.

Process of GAN training through the misclassification of predictions in each time step reminds me of predator-prey equations (also known as Lotka-Volterra), where the population of predators consume prey, which reduce their population. However, with less prey population it is hard for predators to forage and thus their population descends in reaction, which results in prey descend slowing down and eventual grow. Ease of seeking for food makes the predator population start to increase too, and on and on. Similarly, the Generator and Discriminator in GAN are trying to beat the other – if one gets too better, the worse performing model has it easier to learn while the better model stay still and this way the models are gradually lowering the misclassification error of the predictions of both networks.

The bottleneck of working with Generative Adversarial Network are high computational and resource requirements. The training has been performed on Google Colab in a form of Jupyter Notebook. Usage of GPUs however is limited for time and direct activity (interaction) with the page. Thus, the development is very slow and the long-running tasks must be carefully planned and cannot be ran without supervision, e.g. over night. (Google 2020)

As for me, this is my first project where I am using General Adversarial Networks as a model as well as the first project where I use PyTorch. I have not done much such demanding tasks requiring high computational power so even using Google Colab was something new for me. As these methods and technologies are key in the ML/AI job market, it is definitely an incredibly valuable experience.

Appendix: Code

The code for this paper is present in Github repository https://github.com/martinbenes1996/732a92-project, without the trained models (due to big sizes of the outputs). If the code is to be tested, the models need to be retrained!

Examples of usage can be found in README of the projects.

List of modules (structure of `src/`):

- `fetch.py` = fetches and parses the cyberbully dataset (into words and sentences)
- `dataset.py` = loads words and sentences
- `config.py` = global configuration for the framework
- `discriminator.py, generator.py` = implementation of GAN components
- `gan.py` = operations with the GAN model – initialization, loading, saving
- `train.py` = the GAN training procedure
- `markov.py` = Markov Chain model and generator
- `evaluate.py` = evaluation of trained GAN: text generating, loss plot, performance
- `embeddings/`
    - `load.py` = implements generators of batches (embedded already)
    - `models.py` = transforms data using embeddings
    - `train.py` = train embeddings
    - `rev.py` = train reversed embeddings and transform vectors

Generated texts from models are in output/ directory. Since scalar incremental embedding does not generate anything reasonable, it is missing.

- `markov.txt` = Markov Chain
- `word2vec.txt` = closest word2vec
- `bert.txt` = Bert model

Resources

Cambria E., Das D., Bandyopadhyay S., Feraco A. 2017. „A Practical Guide to Sentiment Analysis.“ Cham: Springer. doi:10.1007/978-3-319-55394-8.
Chen, Kuan. 2018. „Generative Model for text: An overview of recent advancements.“ 2. 10. https://www.kuanchchen.com/post/nlp_generative_model/.
Elsafoury, Fatma. 2020. „Cyberbullying datasets.“ 6. doi:10.17632/jf4pzyvnpj.1.
Fraser, Andrew. 2008. *Hidden Markov models and dynamical systems.* Philadelphia: Society for Industrial and Applied Mathematics. doi:ISBN 978-0-898716-65-8.
Goodfellow I., Pouget-Abadie J., Mirza M., Xu B., Warde-Farley D., Ozair S., Courville A., Bengio Y. 2014. „Generative Adversarial Networks.“ Montréal, 10. 6. doi:1406.2661.
Google. 2020. „Google Colaboratory.“ https://colab.research.google.com/.
Google Inc. 2020. „GANs: Common Problems.“ https://developers.google.com/machine-learning/gan/problems.
Inkawhich, Nathan. 2017. „DCGAN Tutorial.“ PyTorch. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.
Kingma, Diederik and Welling, Max. 2014. „Auto-Encoding Variational Bayes.“
Liu, Kanghui. 2020. „Pytorch 20: Implement RNN(LSTM) Generate Random Text.“ 28. 6. https://www.bigrabbitdata.com/pytorch-20-implement-rnnlstm-generate-random-text/.
McCormick, Chris and Ryan, Nick. 2020. „BERT Word Embeddings Tutorial.“ 27. 5. http://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/.
Mikolov, Tomas and Sutskever, Ilya and Chen, Kai and Corrado, Greg and Dean, Jeffrey. 2013. „Distributed Representations of Words and Phrases and their Compositionality.“
Mirza, Mehdi and Osindero,Simon. 2014. „Conditional Generative Adversarial Nets.“
Mosquera, Diego Gomez. 2018. „GANs from Scratch 1: A deep introduction. With code in PyTorch and TensorFlow.“ 1. 2. https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f.
Robertson, Sean. 2017. „NLP From Scratch: Generating Names with a Character-Level RNN.“ PyTorch. https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html.
Vasilev I., Slater D., Spacagna G., Roelants P., Zocca V. 2019. *Python Deep Learning.* 2nd Edition. Birmingham: Packt Publishing. doi:978-1-78934-846-0.
Wang, Alex and Cho, Kyunghyun. 2019. „BERT has a Mouth, and It Must Speak: BERT as a Markov Random Field Language Model.“
Wang, Ke, and Xiaojun Wan. 2018. "SentiGAN: Generating Sentimental Texts via Mixture Adversarial Networks." International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2018/618.