Realistic text structure production

University of Illinois at Chicago

IDS 576 Deep Learning and Applications

Dr. Theja Tulabandhula

Team members:
Supriya Ramarao Prasanna
John Kirschenheiter
Vanisa Achakulvisut

Introduction

A generative adversarial network (GAN) is a combination of two neural networks (generator model and discriminator model) competing with each other. GAN is very popular and useful for generating high-quality images. However, for text generation these models need huge parameter spaces and data sets to be effective and so training models is difficult. The simple structure of GAN has been introduced with the purpose of simplifying training, making it easy to produce text without a huge network.

Gutenberg.txt was used as a source of training data in the project. In order to parse the data into the simplified GAN model, the data should be cleaned and arranged into the fit format for training GAN. SpaCy is a very useful, convenient and popular package for Natural Language Processing. Therefore, SpaCy was used to transform the word into values which indicated the role and function in the sentence (Part of Speech or POS). Additionally, SpaCy was able to keep the relationship between each word in the sentence (Dependency) as well.

Vectorization is the transformation of POS Tagging and Dependency into vectors of numbers. The results will be parsed to a discriminator model in GAN. The project roadmap can be easily explain by 4 steps as follows;

Step 1: Use SpaCy to create a dependency network
Step 2: Transform into vector of number
Step 3: Train GAN to produce new sentence dependencies
Step 4: Produce new sentence templates

Objectives

1. Develop the simple neural network to generate sentences which are meaningful and remain the relationship between the words and english grammar.
2. Able to translate the text into vector format and convert it back into text without distorting the meaning.
3. Apply the basic existing neural networks model to the project

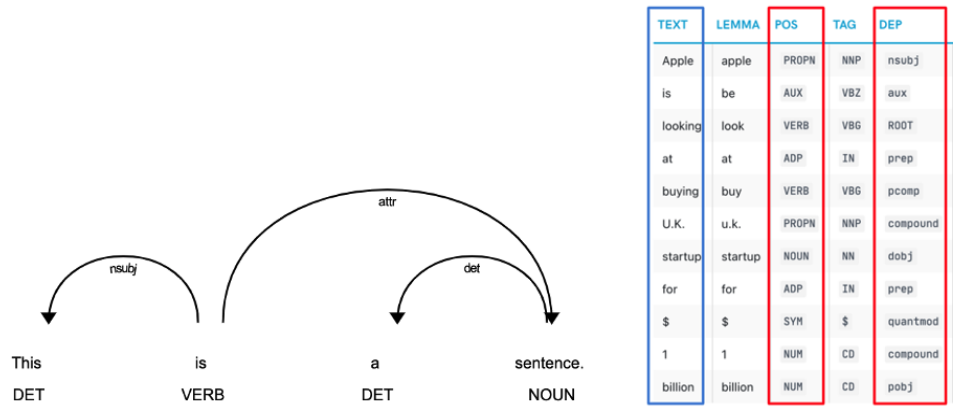
Methodology

1. SpaCy

SpaCy is a relatively new package for “Industrial strength NLP in Python”. Spacy is a package that provides all of the common tasks related to NLP projects.
The two main features POS tags and dependency from SpaCy are used in the project.

- Part-of-speech tagging or POS Tagging: Gives the part of speech of the sentence.
- Dependency Parse: The relationship between two words in a sentence showing as dependency tags.

This can be easily illustrated in Figure 1



TEXT	LEMMA	POS	TAG	DEP
Apple	apple	PROPN	NNP	nsubj
is	be	AUX	VBZ	aux
looking	look	VERB	VBG	ROOT
at	at	ADP	IN	prep
buying	buy	VERB	VBG	pcomp
U.K.	u.k.	PROPN	NNP	compound
startup	startup	NOUN	NN	dobj
for	for	ADP	IN	prep
$	$	SYM	$	quantmod
1	1	NUM	CD	compound
billion	billion	NUM	CD	pobj

Figure 1 Example POS tags and Dependency

```
def all_in_one(text):
    preprocessed = preprocessor_final(text)
    print(preprocessed)
    nlp = spacy.load('en_core_web_sm')
    doc = nlp(preprocessed)
    #displacy.serve(doc, style='dep')
    list1= []
    list2= []
    for token in doc:
        list1.append(token.pos_)        POS Tagging
    for token in doc:
        list2.append(token.dep_)        Dependency
    #print(list1)
    return (list1, list2)
```

Figure 2 Sample of Code using SpaCy package

“all-in_one” as presented in Figure 2 is the main function that has been used to create POS Tagging and Dependency of each word in the sentence.

2. Vectorization

The idea of transforming POS Tagging and Dependency into numbers is to make a dummy dictionary and give the index referring to the created dictionary for each POS tag and dependency. Then a fixed length of the vector was created by the specified max length of sentences in the input article. The value in the vector remains the same but the value of position that exceeds its length will be filled in with 0.

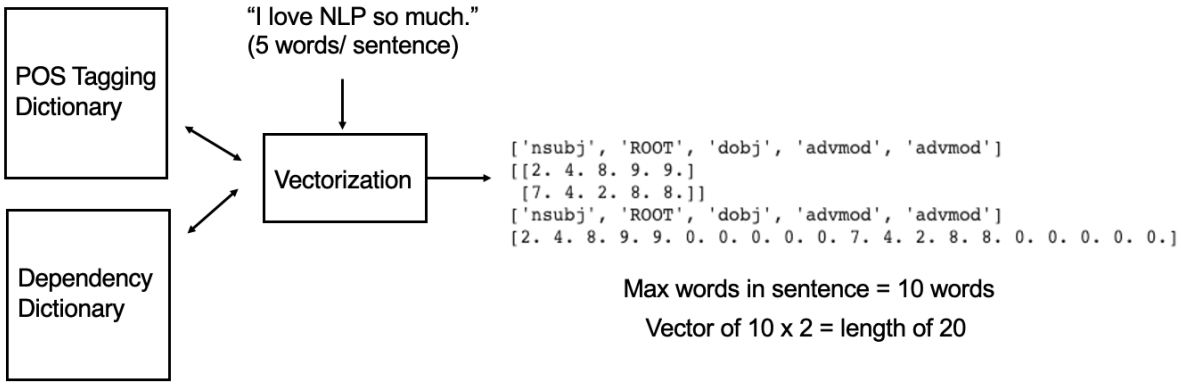


Figure 3 Basic Concept of Vector transformation

Figure 4 Preparation Input Data for GANs

Gutenburg data consists of many sentences, these sentences were transformed into many vectors and stored in an array named "rawTrainingData" as shown on the left side in Figure 3. After that, "rawTrainingData" was prepared to be ready for parsing to the GANs model using CustomDataset and Dataloader as shown on the right side in Figure 4 [4]. The dimension of the training dataset is [max of sentence length x 2, number of sentences]. For example, at the

beginning, the small dataset of 28 sentences/ article with maximum sentence length of 64 words was used in the model. The dimension of rawTrainingData array is [64x2,28] or [128,28].

3. VAE

It should be noted that our team put significant effort into exploring a variational autoencoder scheme for solving this problem. Ultimately, we decided to use a GAN, as the operational complexity was much smaller.

4. GAN

Generative Adversarial Networks or GANs is an approach to generative modeling using deep learning methods and it is an unsupervised learning that involves automatically discovering and learning the regularities.

The GANs model comprises of two sub-models; the *generator model* that we train to generate new examples, and the *discriminator model* that tries to classify examples as either real (from the domain) or fake (generated). [5] The diagram in Figure 5 below shows the basic working concepts of GANs

The key concept of GANs is to let the generator and discriminator compete against each other. The generator generates the "fake" data and passes it to the discriminator. The discriminator tries to distinguish between real data and the generated fake data from the generator. The structure and sample of codes are shown in Figure 6 and 7 respectively.

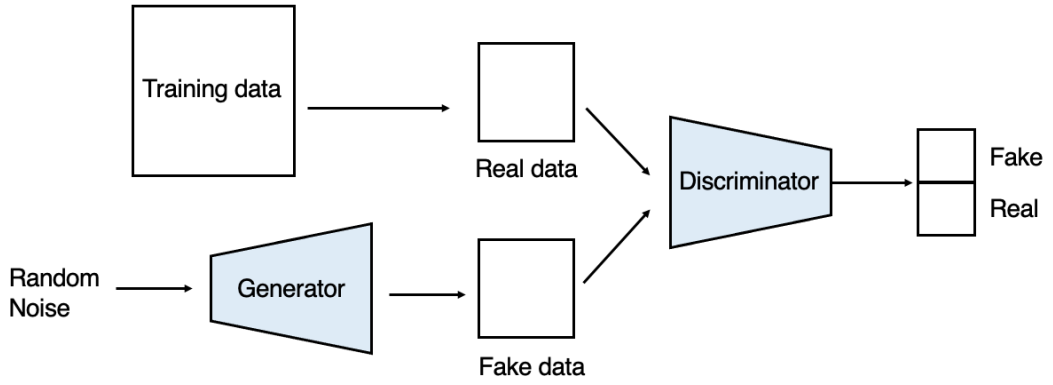


Figure 5 Basic Concept of GANs

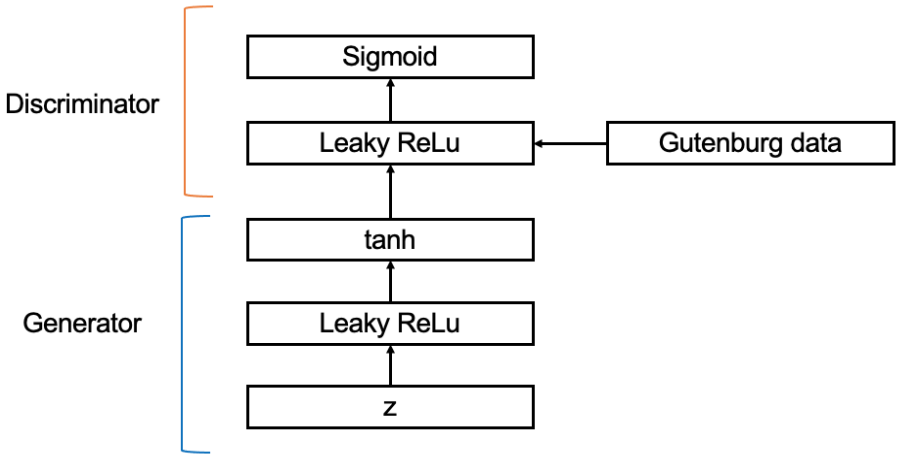


Figure 6 Overview of GANs composition

```python
#Discriminator
import torch.nn as nn
import torch.nn.functional as F

class Discriminator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()

        # define hidden linear layers
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)

        # final fully-connected layer
        self.fc4 = nn.Linear(hidden_dim, output_size)

        # dropout layer
        self.dropout = nn.Dropout(0.3)


    def forward(self, x):
        # all hidden layers
        x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # final layer
        out = self.fc4(x)

        return out
```

```python
#Generator
class Generator(nn.Module):

    def __init__(self, input_size, hidden_dim, output_size):
        super(Generator, self).__init__()

        # define hidden linear layers
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)

        # final fully-connected layer
        self.fc4 = nn.Linear(hidden_dim, output_size)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        # all hidden layers
        x = F.leaky_relu(self.fc1(x), 0.2) # (input, negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        # final layer with tanh applied
        out = F.tanh(self.fc4(x))

        return out
```

Figure 7 The sample code of Discriminator and Generator [6]

Discriminator

The discriminator network is a typical linear classifier. To create a network, at least one hidden layer and one activation function are needed.

- Leaky ReLu as Hidden layer

  Leaky ReLU was used to allow gradients to flow backwards through the layer unimpeded. The purpose of using LeakyRelu is to prevent a “dying ReLU” problem because it doesn't have zero-slope parts and it speeds up training as well.

- Sigmoid as Activation Function
  The purpose of using Sigmoid at Discriminator is because the aimed output values are values of 0-1 indicating whether training data is real or fake.

Generator
The generator network will be almost exactly the same as the discriminator network, except that we're applying a tanh activation function to our output layer.

- Tanh as an Activation Function
  The generator has been found to perform the best with tanh for the generator output, which scales the output to be between -1 and 1, instead of 0 and 1. Regarding the prototype neural network code [6] which performs well in pictures generation. Therefore, Tanh was used in the project with the anticipation of good performance.

Discriminator and Generator Losses

Discriminator Losses
For the discriminator model, the total loss is the sum of the losses for real and fake vactor, “d_loss = d_real_loss + d_fake_loss”.

BCEWithLogitsLoss is used to calculate binary cross entropy loss with logits this indicated sigmoid activation function and binary cross entropy loss in one function.

The label = 1 indicates real vector and 0 indicates the fake vector. Multiplying the model with 0.9 to smooth the labels. The sample code is shown below in Figure 8.

```
# Calculate losses
def real_loss(D_out, smooth=False):
    batch_size = D_out.size(0)
    # label smoothing
    if smooth:
        # smooth, real labels = 0.9
        labels = torch.ones(batch_size)*0.9
    else:
        labels = torch.ones(batch_size) # real labels = 1

    # numerically stable loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss

def fake_loss(D_out):
    batch_size = D_out.size(0)
    labels = torch.zeros(batch_size) # fake labels = 0
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

Figure 8 Sample code of Loss Calculation

Generator Loss

The generator loss will look similar to discriminator only with opposite labels. The generator's goal is to get D(fake_vector) = 1.

Optimizer

Generator and discriminator variables are updated separately so they were defined into two separated Adam optimizers. The sample code is shown below in Figure 9.

```
[ ] #Optimizer
    import torch.optim as optim

    # Optimizers
    lr = 0.002

    # Create optimizers for the discriminator and generator
    d_optimizer = optim.Adam(D.parameters(), lr)
    g_optimizer = optim.Adam(G.parameters(), lr)
```

Figure 9 Sample code of Optimizer

Training model logic are following;

1. Discriminator training
    1.1. Compute the discriminator loss on real, training in trainloader
    1.2. Generate fake vectors
    1.3. Compute the discriminator loss on fake, generated vectors
    1.4. Add up real and fake loss
    1.5. Perform backpropagation + an optimization step to update the discriminator's weights
2. Generator training
    2.1. Generate fake vectors
    2.2. Compute the discriminator loss on fake vectors, using **flipped** labels.
    2.3. Perform backpropagation + an optimization step to update the generator's weights

Findings and results

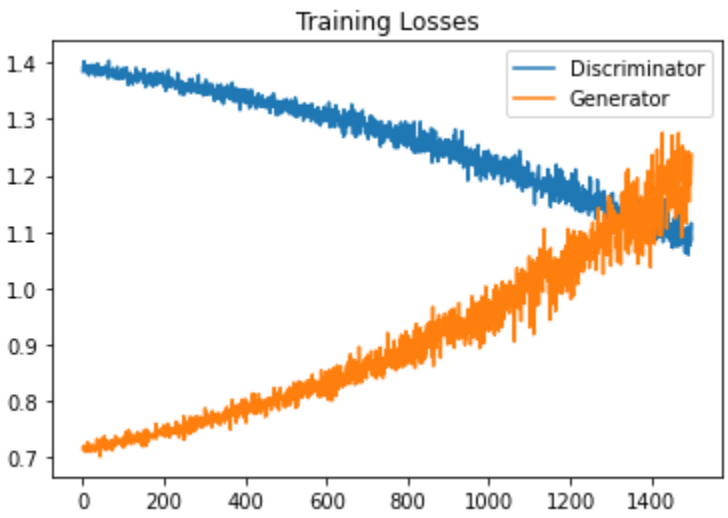


Figure 10 Training losses from model

Ultimately, the model struggled to train. We could easily tune the parameters to train a Discriminator that could distinguish between generated sentence templates and real sentence templates, but it was difficult to tune the hyperparameters such that the generator produces realistic sentence templates. We tried several hyperparameter/model configurations, but few produce useful results. The losses is shown in Figure 10

Discussion

The core idea behind the design of this model has merit. This concept could be used for a variety of applications. For example:

- Sentence templates could be used to make more efficient use of existing text generation
- Templates can be used to analyze different text, for example the difference between poetry and novels
- Templates could be used to explain language structure for ESL students

Conclusions and Recommendations

Despite the great success of Generative Adversarial Networks (GANs) in generating high-quality novel data, GANs for text generation still face two major challenges: first, most text GANs are unstable in training mainly due to ineffective optimization of the generator, and they heavily rely on maximum likelihood pretraining; second, most text GANs adopt autoregressive generators without latent variables, which largely limits the ability to learn latent representations for natural language text. Before this methodology is used the model structure should be changed so that the generated results are more usable.

References

[1] Introduction to SpaCy:
https://towardsdatascience.com/a-short-introduction-to-nlp-in-python-with-spacy-d0aa819af3ad
[2] SpaCy documentation: https://spacy.io/usage/visualizers
[3] SpaCy documentation: https://spacy.io/usage/linguistic-features#pos-tagging
[4] Data Loader: https://debuggercafe.com/custom-dataset-and-dataloader-in-pytorch/
[5] GANs Information:
https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/
[6] GANs Models template:
https://github.com/udacity/deep-learning-v2-pytorch/blob/master/gan-mnist/MNIST_GAN_Solution.ipynb
[7] Extra information about GANs:
http://kth.diva-portal.org/smash/get/diva2:1374343/FULLTEXT01.pdf
[8] Extra information about GANs:
https://becominghuman.ai/generative-adversarial-networks-for-text-generation-part-1-2b886c8cab10