Realistic text structure production

Supriya Ramarao Prasanna | John Kirschenheiter | Vanisa Achakulvisut

University of Illinois at Chicago
IDS 576 Deep Learning and Applications
Dr. Theja Tulabandhula

Introduction & Objectives

Why GANs ?
The simple structure of GAN and simplified training, making it easy to produce text without a huge network.
How data is Processed ?
Data preprocessing with SpaCy (Convenient for Natural Language Processing) and Vectorization before feeding to General Adversarial Networks or GANs

Objectives

- Develop the simple neural network to generate sentences which are meaningful and retain the relationship between the words and english grammar.

- Able to translate the text into vector format and convert it back into text without distorting the meaning.

- Apply the basic existing neural network model to the project

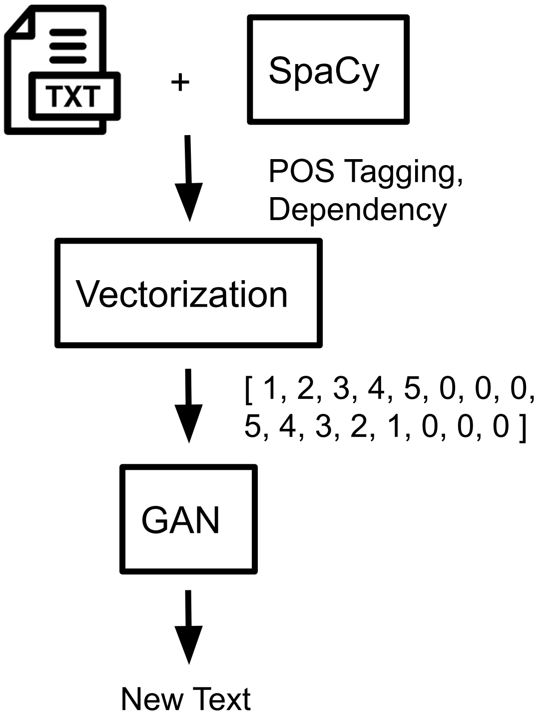
Methodology

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 text



SpaCy

Part-of-speech / POS Tagging:
Assigning word types to tokens, like verb or noun.

Dependency Parsing:
Assigning syntactic dependency labels, describing the relations between individual tokens, like subject or object.

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

POS Tagging command: token.pos_

Dependency Parsing command: token.dep_

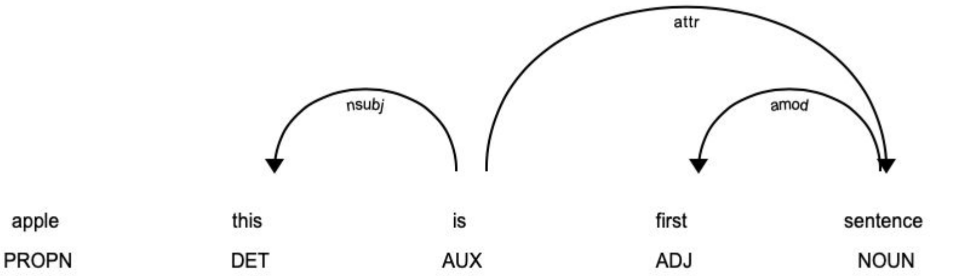
```python
import re
def preprocessor_final(text):
    if isinstance((text), (str)):
        text = re.sub('<[,^>.!]*>', ' ', text)
        text = re.sub('[\W]+', ' ', text.lower())
        return text
    if isinstance((text), (list)):
        return_list = []
        for i in range(len(text)):
            temp_text = re.sub('<[^>]*>', ' ', text[i])
            temp_text = re.sub('[\W]+', ' ', temp_text.lower())
            return_list.append(temp_text)
        return(return_list)
    else:
        pass

def all_in_one(text):
    preprocessed = preprocessor_final(text)
    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_)
    for token in doc:
        list2.append(token.dep_)
    #print(list1)
    return (list1, list2)
```

```python
# Test on one sentence
test = "Apple, This is first sentence."
get_postags, get_dependencies = all_in_one(test)
```

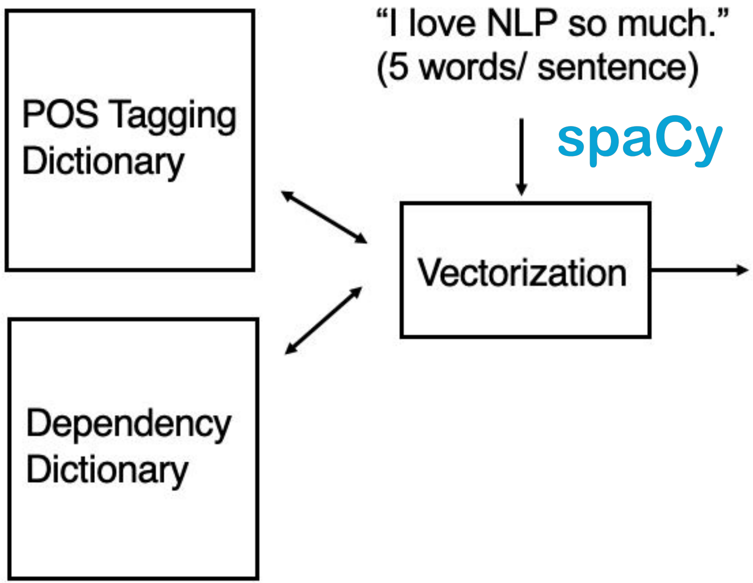
```
apple this is first sentence
```

```python
# See the result
print("This is get_postags"+ str(get_postags))
print("This is get_dependencies"+ str(get_dependencies))
```

```
This is get_postags['PROPN', 'DET', 'AUX', 'ADJ', 'NOUN']
This is get_dependencies['npadvmod', 'nsubj', 'ROOT', 'amod', 'attr']
```



Vectorization




```
['nsubj', 'ROOT', 'dobj', 'advmod', 'advmod']
[[2. 4. 8. 9. 9.]
 [7. 4. 2. 8. 8.]]
['nsubj', 'ROOT', 'dobj', 'advmod', 'advmod']
[2. 4. 8. 9. 9. 0. 0. 0. 0. 0. 7. 4. 2. 8. 8. 0. 0. 0. 0. 0.]
```

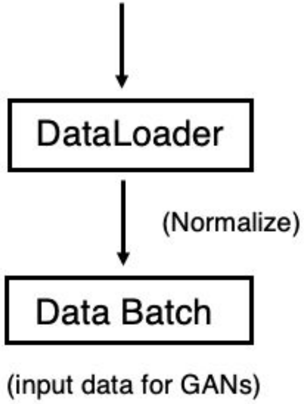
Max words in sentence = 10 words

Vector of 10 x 2 = length of 20

Vectorization

rawTrainingData

```
[[ 1.   2.   3.  ...   0.   0.   0.]
 [ 1.   3.  13.  ...   0.   0.   0.]
 [ 2.   1.   3.  ...   0.   0.   0.]
 ...
 [37.   5.   6.  ...   0.   0.   0.]
 [16.   1.  15.  ...   0.   0.   0.]
 [ 1.   3.  18.  ...   0.   0.   0.]]
```




```python
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

class CustomDataset(Dataset):
    def __init__(self, data, transforms=None):
        self.data = data
        self.transforms = transforms

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = self.data[i, :]
        #data = np.asarray(data).astype(np.uint8)
        if self.transforms:
            data = self.transforms(data)
        else:
            data = data/50
        return data.astype(np.float32)
```

```python
#train_data = CustomDataset(rawTrainingData, transform)
train_data = CustomDataset(rawTrainingData)

# dataloaders
trainloader = DataLoader(train_data, batch_size=8, shuffle=True)
```

General Adversarial Networks or GANs

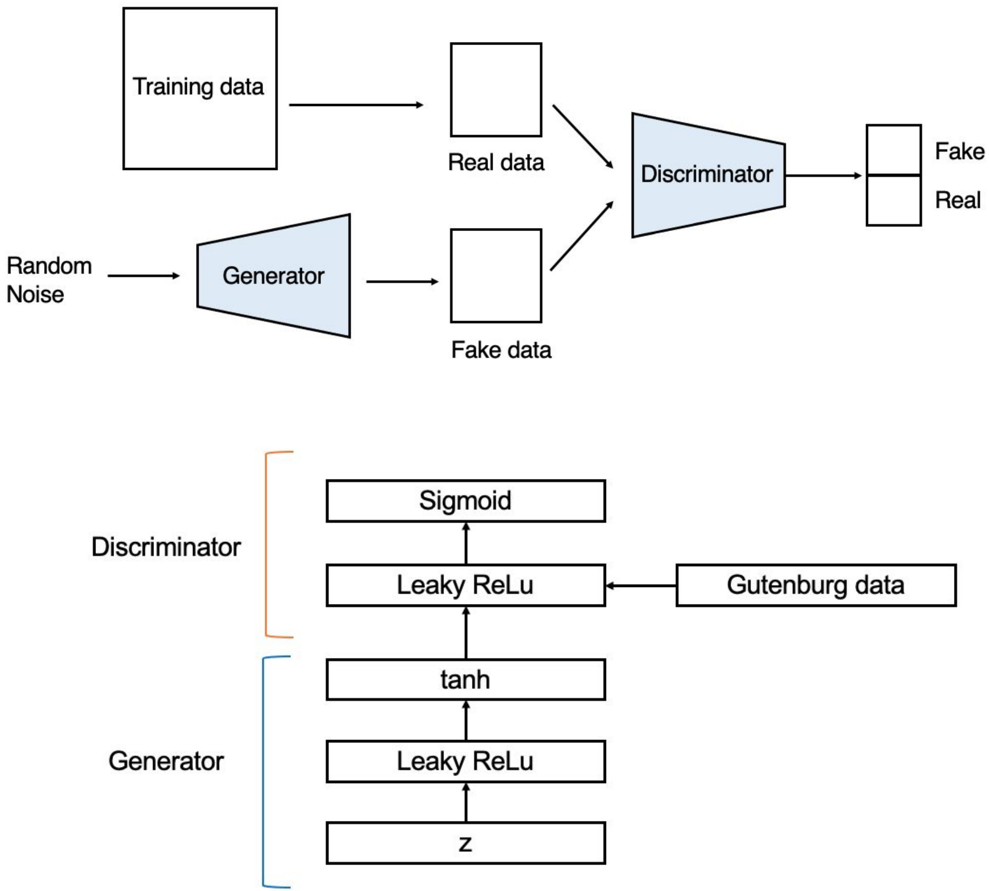
Two sub-models;

1. **Generator model**

Generate new examples from Random

2. **Discriminator model**

To classify examples as either real (from the domain) or fake (generated).



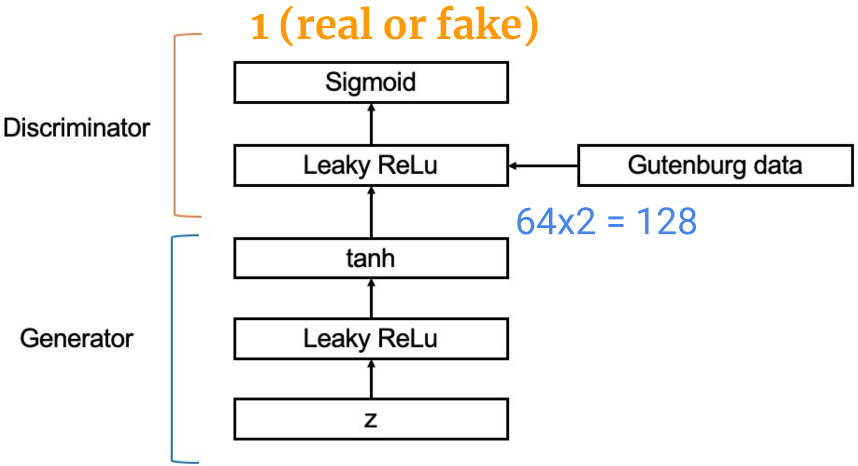
General Adversarial Networks or GANs

Model Parameters

Sent_length*2

```
Discriminator(
  (fc1): Linear(in_features=128, out_features=16, bias=
  (fc2): Linear(in_features=16, out_features=16, bias=T
  (fc3): Linear(in_features=16, out_features=1, bias=Tr
  (dropout): Dropout(p=0.3, inplace=False)
)
Generator(
  (fc1): Linear(in_features=60, out_features=16, bias=T
  (fc2): Linear(in_features=16, out_features=16, bias=T
  (fc3): Linear(in_features=16, out_features=128, bias=
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Sent_length*2



Loss: nn.BCEWithLogitsLoss()

Optimizer:
d_optimizer = optim.Adam(D.parameters(), lr)
g_optimizer = optim.Adam(G.parameters(), lr)

Discriminator Losses
"D(loss) = D(real_loss) + D(fake_loss)"
Generator Losses
"D(fake_loss)"

General Adversarial Networks or GANs

```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)

        # final fully-connected layer
        self.fc3 = 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)
        # final layer
        out = self.fc3(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)

        # final fully-connected layer
        self.fc3 = 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)
        # final layer with tanh applied
        out = F.tanh(self.fc3(x))

        return out
```

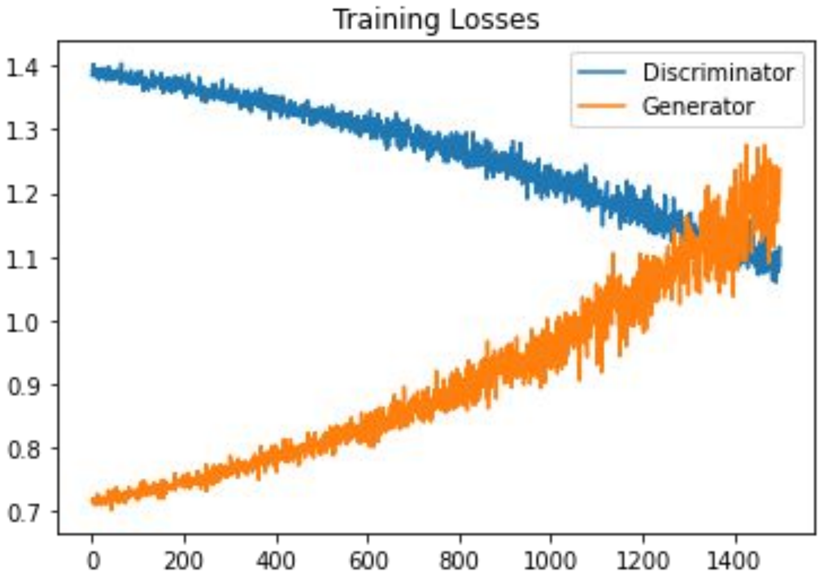
General Adversarial Networks or GANs

```python
# 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
```

```python
#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)
```

General Adversarial Networks or GANs

Discriminator and Generator Losses



Training Algorithm

for number of training iterations do:
    for batch_i, data in enumerate(trainloader):

Discriminator training (real(1) and fake(0))

1. Compute the discriminator loss on real vectors
2. Generate fake vectors
3. Compute the discriminator loss
4. Add up the real and fake loss
5. Perform backpropagation + an optimization step to update the discriminator weights

Generator training (real(0) and fake(1))

1. Generate fake vectors
2. Compute the discriminator loss on fake **opposite** labels
3. Perform backpropagation + an optimization step to update the generator's weights

Findings and results

The Upside

- We successfully generated sentence templates
- We evaluated our ability to generate statements using standard loss metrics

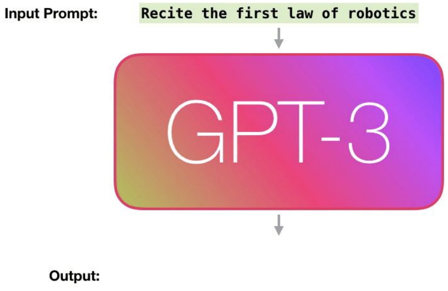
The Downside

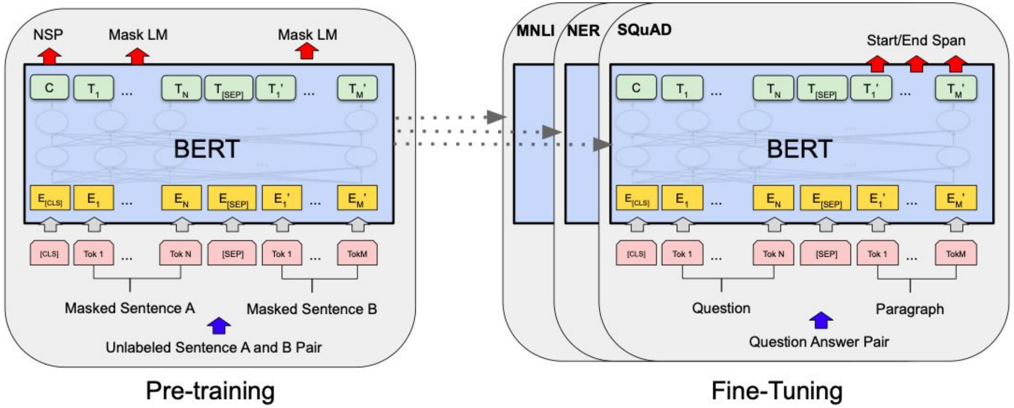
- We had many issues getting consistent convergence
- We had issues with overfit
- We did not achieve our stretch goals

Discussion

Further Uses

- 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





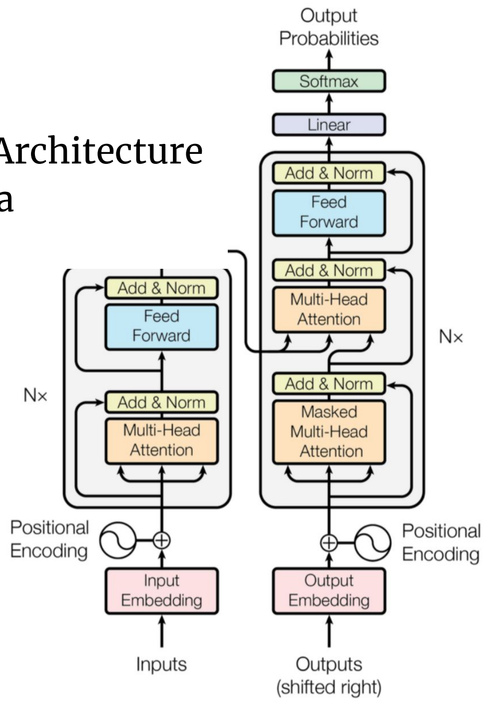
Conclusions and recommendations

Conclusions

- This architecture is a promising way to explore NLP
- This method can be applied flexibly to many uses
- GANs specifically have issues around convergence

Recommendations

- Change to Transformer Architecture
- Get better data



Thank you