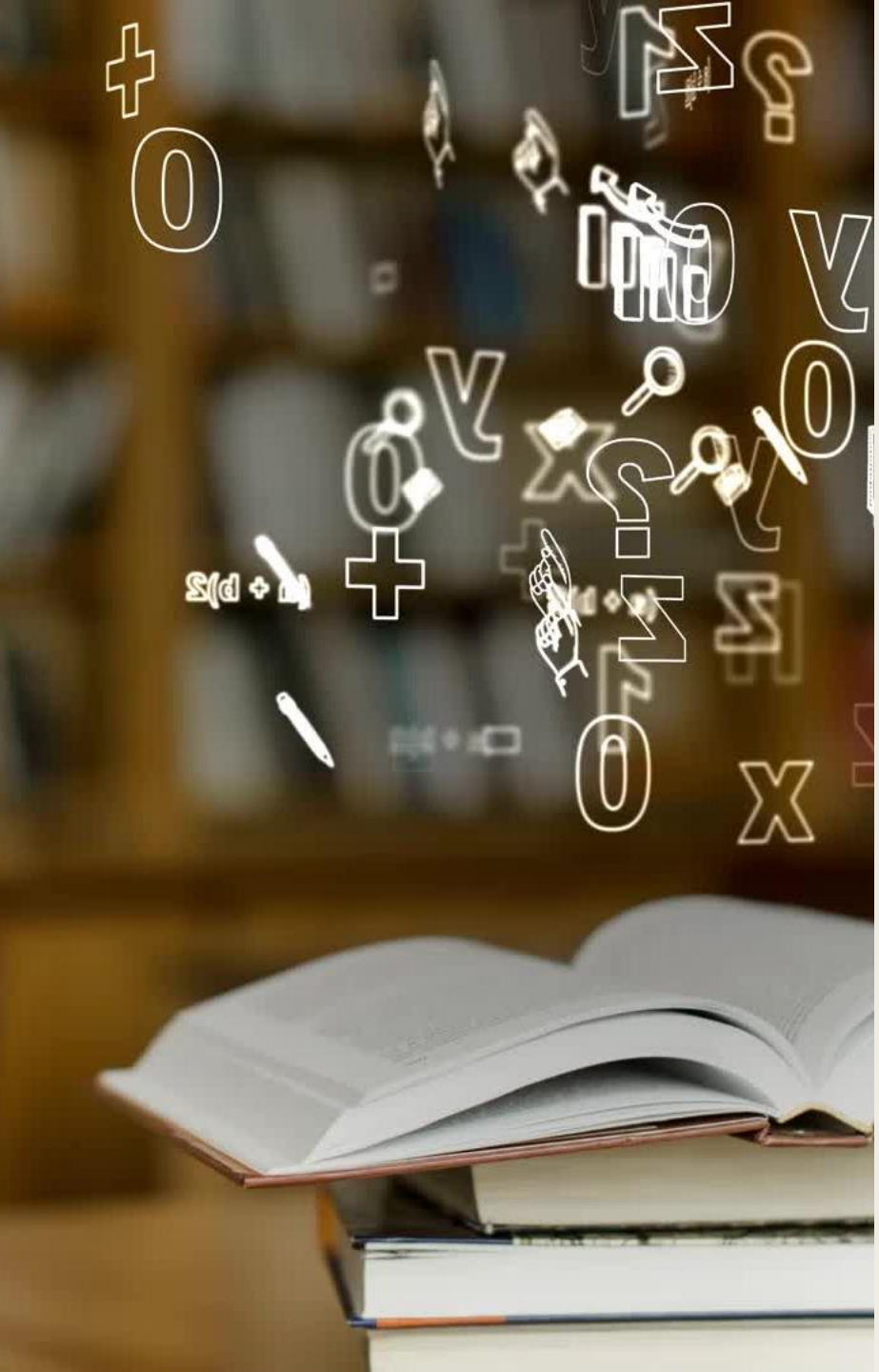
DEEP LEARNING: TEXT GENERATIVE MODEL

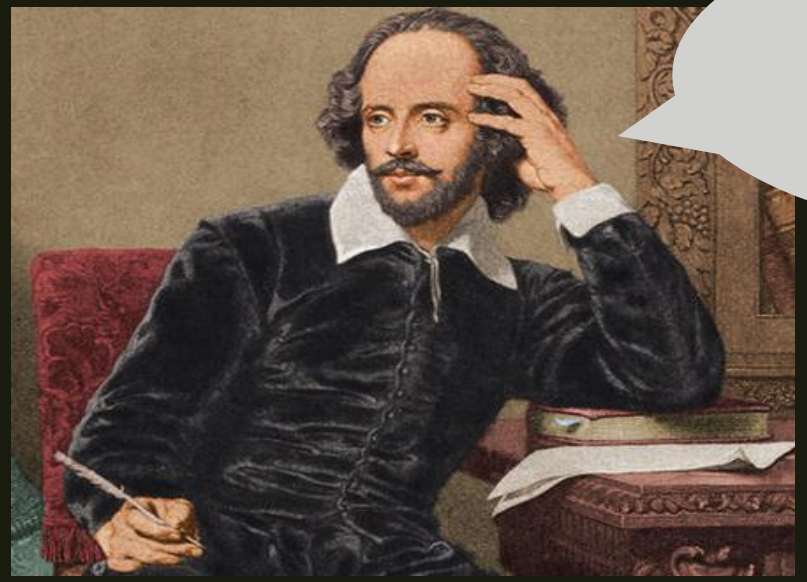
by
Milantoivonkavana RANDRIAMBOAVONJY

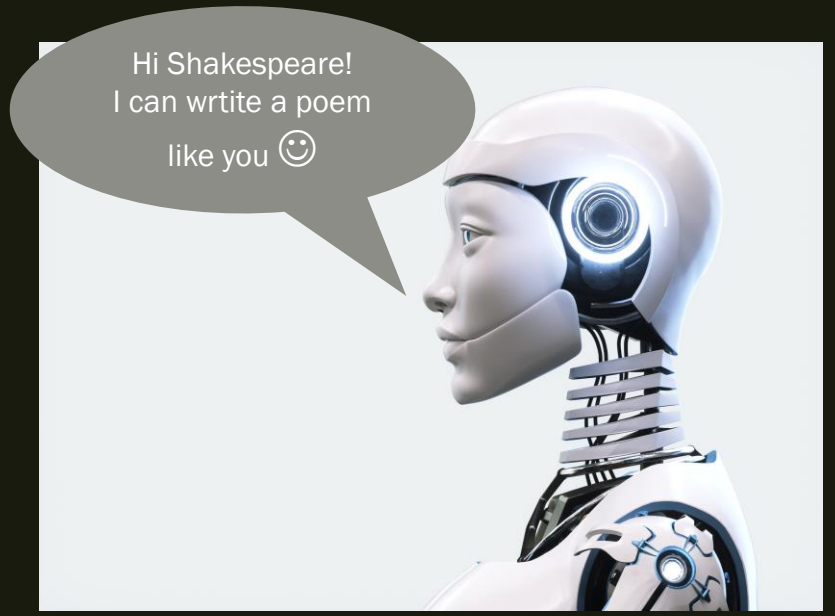


Summary

- Introduction
- Text PreProcessing
- Neural Network model architecture
- Model Training and result
- Limitations


Introduction





Introduction

Project Objective:
Create a neural network model mimicking William Shakespeare's writing style.

Training Data:
Utilize a Shakespearean work to train the neural network.

Outcome:
•Model's Goal: Replicate Shakespearean sonnet style.
•Input a few words, and the model generates text in Shakespeare's distinctive style.

Text Preprocessing

Import the data in text format: Unstructured data

Clean and prepare data

Convert text into machine-readable numbers

Data contained processed and encoded text

Tokenization:
Extraction of the words from the text

3

Trigrams:
Transform the data where each row contains three sequences of words

Sometimes, the «stopwords» are removed as they are supposed to do not contribute to the meaning

BUT they will be kept

They may contribute to define the style of Shakespeare

Example of stopwords: a, the, and, or,…

Text Preprocessing

```
#The first 50 characters in the data
print(text[:500])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.
```

```
print(split_punctuation(text_sample)[:50])
```

```
['First', 'Citizen', ':', 'Before', 'we', 'proceed', 'any', 'further', ',', 'hear', 'me', 'speak', '.', 'All', ':', 'Speak',
',', 'speak', '.', 'First', 'Citizen', ':', 'You', 'are', 'all', 'resolved', 'rather', 'to', 'die', 'than', 'to', 'famish',
'?', 'All', ':', 'Resolved', '.', 'resolved', '.', 'First', 'Citizen', ':', 'First', ',', 'you', 'know', 'Caius', 'Marcius', '
is', 'chief']
```

Tokenization

```
trigrams = [([text_split[i], text_split[i + 1]], text_split[i + 2])
            for i in range(len(text_split) - 2)]
chunk_len=len(trigrams)
print(trigrams[:30])
print(chunk_len)
```

```
[(['First', 'Citizen'], ':'), (['Citizen', ':'], 'Before'), ([':', 'Before'], 'we'), (['Before', 'we'], 'proceed'), (['we', 'p
roceed'], 'any'), (['proceed', 'any'], 'further'), (['any', 'further'], ','), (['further', ','], 'hear'), ([',', 'hear'], 'me
'), (['hear', 'me'], 'speak'), (['me', 'speak'], '.'), (['speak', '.'], 'All'), (['.', 'All'], ':'), (['All', ':'], 'Speak'),
([':', 'Speak'], ','), (['Speak', ','], 'speak'), ([',', 'speak'], '.'), (['speak', '.'], 'First'), (['.', 'First'], 'Citizen
'), (['First', 'Citizen'], ':'), (['Citizen', ':'], 'You'), ([':', 'You'], 'are'), (['You', 'are'], 'all'), (['are', 'all'], '
resolved'), (['all', 'resolved'], 'rather'), (['resolved', 'rather'], 'to'), (['rather', 'to'], 'die'), (['to', 'die'], 'than
254791
```

Trigrams

Convert the trigrams text into numbers

Word indexisation: each word and puctuation is represented by an unique number

```
#Create a dictionary which give every single word an index, that we will use in the data set
vocab = set(text_split)
voc_len=len(vocab)
word_to_ix = {word: i for i, word in enumerate(vocab)}
```

```
print(word_to_ix)
```

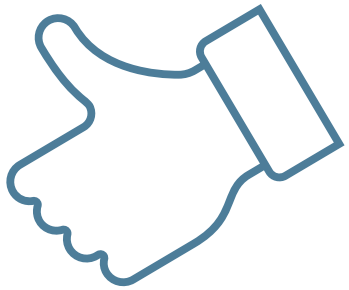
```
{'fatigate': 0, 'fastest': 1, 'Cretan': 2, 'Anon': 3, 'hang': 4, 'Fame': 5, 'abundantly': 6, 'shallow': 7, 'Flavius': 8, 'quen
ch': 9, 'father': 10, 'foemen': 11, 'tumble': 12, 'mind': 13, 'wither': 14, 'Outlive': 15, 'perverse': 16, 'impart': 17, 'tyra
nt': 18, 'heating': 19, 'worshipful': 20, 'breathes': 21, 'Neighbours': 22, 'lampass': 23, 'interpretation': 24, 'manage': 25,
```

Convert the trigrams into numbers based on the indexes, and split the inputs and the target

```
#Convert the word into number based on the index
inp=[]
tar=[]
for context, target in trigrams:
        context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)
        inp.append(context_idxs)
        targ = torch.tensor([word_to_ix[target]], dtype=torch.long)
        tar.append(targ)
```

Data format ready for analysis!!

Tensor with 150’000 rows



```
print(len(inp))
print(inp[:20])
print(tar[:20])
```

```
150000
[tensor([6479, 5360]), tensor([5360, 6338]), tensor([6338,  988]), tensor([ 988, 2013]), tensor([2013, 5418]), tensor([5418, 4
045]), tensor([4045, 9739]), tensor([9739, 9473]), tensor([9473, 9735]), tensor([9735, 5366]), tensor([5366, 7977]), tensor([7
977,  481]), tensor([ 481, 7350]), tensor([7350, 6338]), tensor([6338, 8489]), tensor([8489, 9473]), tensor([9473, 7977]), ten
sor([7977,  481]), tensor([ 481, 6479]), tensor([6479, 5360])]
[tensor([6338]), tensor([988]), tensor([2013]), tensor([5418]), tensor([4045]), tensor([9739]), tensor([9473]), tensor([973
5]), tensor([5366]), tensor([7977]), tensor([481]), tensor([7350]), tensor([6338]), tensor([8489]), tensor([9473]), tensor([79
77]), tensor([481]), tensor([6479]), tensor([5360]), tensor([6338])]
```

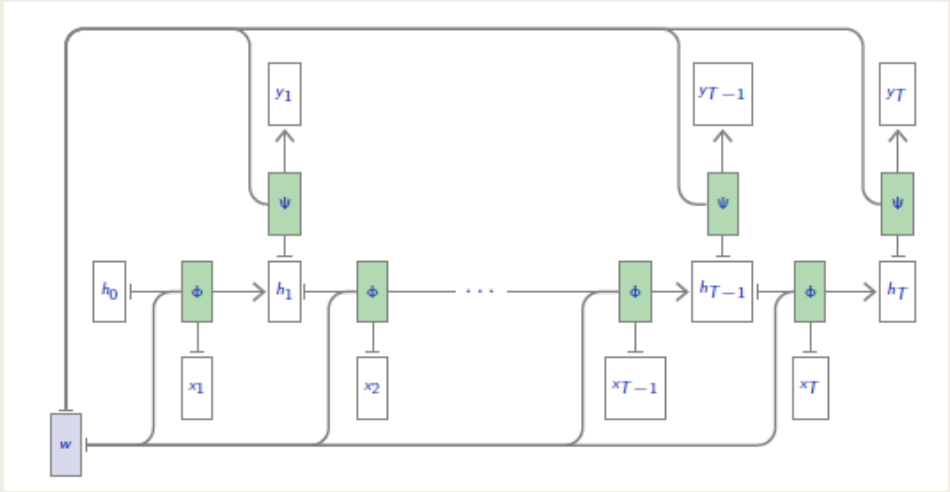
NEURAL NETWORK MODEL ARCHITECTURE:
RECURRENT NEURAL NETWORK (RNN)
&
GATED RECURRENT UNIT (GRU)

RNNs are neural networks designed for sequential data processing well-suited for tasks such as natural language processing or time series prediction

Key idea:
- use of hidden states that maintain information about previous inputs in the sequence
- capture dependencies in sequential data

Prediction at step t: $y_t = \Psi(h_t; w)$, where as $h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$, $\forall t = 1, \ldots, T(x)$

- *W_{hh} is the weight matrix for the hidden state*
- *W_{xh} is the weight matrix for the input*
- *b_h is the bias term*
- *f is an activation function*



GRU

$$r_t = \text{sigm}\left(W_{(x\ r)}x_t + W_{(h\ r)}h_{t-1} + b_{(r)}\right) \quad \text{(reset gate)}$$
$$z_t = \text{sigm}\left(W_{(x\ z)}x_t + W_{(h\ z)}h_{t-1} + b_{(z)}\right) \quad \text{(forget gate)}$$
$$\bar{h}_t = \tanh\left(W_{(x\ h)}x_t + W_{(h\ h)}(r_t \odot h_{t-1}) + b_{(h)}\right) \quad \text{(full update)}$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \bar{h}_t \quad \text{(hidden update)}$$

Neural Network Architecture: RNN

```
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        self.encoder = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size*2, hidden_size, n_layers,batch_first=True,
                          bidirectional=False)
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        input = self.encoder(input.view(1, -1))
        output, hidden = self.gru(input.view(1, 1, -1), hidden)
        output = self.decoder(output.view(1, -1))
        return output, hidden

    def init_hidden(self):
        return Variable(torch.zeros(self.n_layers, 1, self.hidden_size))
```

```
def train(inp, target):
    hidden = decoder.init_hidden().cuda()
    decoder.zero_grad()
    loss = 0

    for c in range(chunk_len):
        output, hidden = decoder(inp[c].cuda(), hidden)
        loss += criterion(output, target[c].cuda())

    loss.backward()
    decoder_optimizer.step()

    return loss.data.item() / chunk_len
```

MODEL TRAINING AND THE RESULT

Model training

```
n_epochs = 30
print_every = 1
plot_every = 1
hidden_size = 100
n_layers = 1
lr = 0.015

decoder = RNN(voc_len, hidden_size, voc_len, n_layers)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

start = time.time()
all_losses = []
loss_avg = 0
if(train_on_gpu):
    decoder.cuda()
for epoch in range(1, n_epochs + 1):
    loss = train(inp,tar)
    loss_avg += loss

    if epoch % print_every == 0:
        print('[%s (%d %d%%) %.4f]' % (time_since(start), epoch, epoch / n_epochs *100 , loss))
#         print(evaluate('ge', 200), '\n')

    if epoch % plot_every == 0:
        all_losses.append(loss_avg / plot_every)
        loss_avg = 0
```

```
[2m 17s (1 3%) 9.5095]
[4m 35s (2 6%) 9.1061]
[6m 55s (3 10%) 7.9348]
[9m 13s (4 13%) 6.9032]
[11m 31s (5 16%) 6.4921]
[13m 48s (6 20%) 6.3796]
[16m 7s (7 23%) 6.3181]
[18m 26s (8 26%) 6.2675]
[20m 44s (9 30%) 6.1649]
[23m 0s (10 33%) 6.0389]
[25m 17s (11 36%) 5.9426]
[27m 36s (12 40%) 5.8751]
[29m 53s (13 43%) 5.8166]
[32m 11s (14 46%) 5.7539]
[34m 28s (15 50%) 5.6843]
[36m 45s (16 53%) 5.6115]
[39m 2s (17 56%) 5.5416]
[41m 19s (18 60%) 5.4776]
[43m 37s (19 63%) 5.4187]
[45m 54s (20 66%) 5.3612]
[48m 14s (21 70%) 5.3031]
[50m 37s (22 73%) 5.2450]
[53m 0s (23 76%) 5.1887]
[55m 21s (24 80%) 5.1345]
[57m 38s (25 83%) 5.0822]
[59m 55s (26 86%) 5.0304]
[62m 12s (27 90%) 4.9786]
[64m 31s (28 93%) 4.9273]
[66m 48s (29 96%) 4.8776]
[69m 5s (30 100%) 4.8295]
```

Result

```
def evaluate(prime_str='we speak', predict_len=100, temperature=1):
    hidden = decoder.init_hidden().cuda()

    for p in range(predict_len):

        prime_input = torch.tensor([word_to_ix[w] for w in prime_str.split()], dtype=torch.long).cuda()
        inp = prime_input[-2:] #last two words as input
        output, hidden = decoder(inp, hidden)

        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # Add predicted word to string and use as next input
        predicted_word = list(word_to_ix.keys())[list(word_to_ix.values()).index(top_i)]
        prime_str += " " + predicted_word
#         inp = torch.tensor(word_to_ix[predicted_word], dtype=torch.long)

    return prime_str
```

Result

```
print(evaluate('you speak', 100, temperature=1.2))
```

```
you speak haste make shed entombed unto case . MERCUTIO : I faint keeping sweet , for we are flow gentleman ' boot , doth each counsel more senators of Exeter to pity against our state Coriolanus Put now tears to loving the bier Edward man do King party , underneath from the Pluto bats good smiles . SICINIUS : Tut , Withal Lovers To hold Friar prince that babe visits he join mu ster back beats ! knighthood more malice This stay you cast were impeach ' human taught did honour My hearts , prosperity soul religious kings without blind ! wear
```

```
print(evaluate('you speak', 100, temperature=0.5))
```

```
you speak sits to get our solemnity frantic swift sepulchre ! O , I am not , I am not , when I have been more . KING EDWARD IV : Why , and you , but by the man of yours . MENENIUS : I will be ruled of the day . : I am the gods of the world : I do beseec h , my lord , and this : the mayor of York . GLOUCESTER : Go , I have done to my grave , And it , and the way to this presence ' , in his eyes
```

```
print(evaluate('ROMEO :', 100, temperature=1.2))
```

```
ROMEO : urge gi go . ROMEO : Nay reach there sight and your brethren night before familiar grave repose them ? of the world . Richard muniments is ready ? So merit Selves thyself ! brake guilty chafed in extremity soon watching . DERBY say come ! or bu t Death as we desire clapping warriors he shall carry native , The life were banishment , a blood That o did forsake and till ' and thee : and awry thy throne together : and back empoison with the east next from nature dissolute where Why modest ? marr y but ' you
```

LIMITATIONS

Limitations

01

line break (/n) not captured in the inputs

02

no activation such as ReLu or softmax is used in the model

03

to generate text, the user should insert at least two words

04

150’000 rows is used to the ‘gpu’ limitations, and it takes 68 min to run the code


THANK YOU FOR YOUR ATTENTION!