Generative AI and NLP

With the advent of generative AI, language language processing has played a crucial rule in the research and development of generating human readable text. We all have used ChatGPT and Bard for text generation but these models are trained on massive datasest of text from various sources, including books, articles, and websites. The exact size of the dataset is not publicly known, but it's estimated to be in the hundreds of gigabytes or even terabytes. Training these LLMs require huge amount of data and computing resources for the models to learn the sequences and patterns for generating the next set of words. The basis approach of developing these models is by utilizing neural networks. The main advantage of neural networks over machine learning models is that neural networks learn from their own mistakes I.e compute the loss and tend to decrease the loss in the next epoch, this makes neural networks the base of all learn language architectures such as transformers or pre-trained models of Open AI or Hugging Face.

Goal

In this article we will train 3 levels of text generation models

* Next character generation
* Next work generation
* Next sentence generation

PyTorch

We will start by training our own text generation model. In this guide, we will use PyTorch over TensorFlow over its simplicity and beginner friendly architecture.

Advantages of LSTM over simple RNNs

We will use an LSTM model as opposed to a simple RNN due to following reasons

* LSTM have the ability to handle long-term dependencies in data, which is essential for text generation.
* They can capture relationships between words that are far apart in the input sequence.
* LSTMs use memory cells and gates to regulate the flow of information
* LSTMs have a separate memory cell that can store information for long periods, allowing them to generate more coherent and context-dependent text.

Architecture of LSTM

Before moving forward let us briefly descibe the structure of an LSTM.

These Long Short Term Memory models (LSTM) tend to provide RNN with a memory state to mitigate the loss of previous sequences of data, simply put RNN have a poor memery. They remember the current state but forget the previous state . this makes them poor at remembering previous states. Advanced tasks like text generation requires a memory state to remember the context of the sentence. Just like when you give multiple prompts to ChatGPT. It always remembers the previous conversations with the user. Our aim is to replicate that behavior using RNNs. The LSTM arcitecture consists of 3 gates and a memory cell.

* Input Gate: It controls the flow of information from current input, previous hidden state.It outputs a vector that controls the flow of raw information. Mathematically it is represented by

i\_t = sigmoid(W\_i \* x\_t + U\_i \* h\_{t-1} + b\_i)

Where Where x\_t is the input sequence, h\_{t-1} is the previous hidden state, c\_{t-1} is the previous cell state, and W, U, and b are learnable weights and biases.

* Cell State/Memory Cell (CS): Stores information over long periods of time that can be selected or modified by input and forget gate.It acts as a kind of "memory" for the network
* Forget Gate:It decides what information to discard from the cell state. It acts like a garbage collector for irrelevant information which will be expelled out from the single neuron. Its equation is

Forget gate = sigmoid(W\_f \* x\_t + U\_f \* h\_{t-1} + b\_f)

* Output Gate: This gate takes in the information from the memory state and previous hidden state and outputs a vector which define sthe output of the neuron

o\_t = sigmoid(W\_o \* x\_t + U\_o \* h\_{t-1} + b\_o)

* Hidden State: It Stores information about the current input and previous inputs. It main goal is to compute the output and update the cell state

Importing the lIbraries

**import** torch

**import** torch.nn **as** nn

**import** string

**from** torch.utils.data **import** Dataset, DataLoader

**import** torch.optim **as** optim

**import** nltk

**from** nltk.corpus **import** stopwords

**from** nltk **import** word\_tokenize, pos\_tag

We use PyTorch for making the neural networks. We also import the dataset,dataloader and optim classes.

NLTK is used for text preprocessing. As we are dealing with textual data like books, articles or corpuses. We donot need numpy or pandas

Next Character Generation

We will start by generating next sequence of characters. We will start by defining two dictionaries, one will store characters as keys and their corresponding indexes as values, and the other will store indexes as keys and characters as value. Why? We are encoding the characters to feed in to neural network and decoding the ouputs in human readable format. Remember , ML or DL models only accept numerical values not textual.

data="When it comes to generating text, GANs and LSTMs have different approaches. LSTMs excel at capturing sequential patterns and context, making them well-suited for tasks like language translation and text summarization. However, they can struggle with creativity and diversity in their output. On the other hand, GANs are designed to generate novel and diverse text by learning the underlying data distribution. While they can produce more creative content, GANs can be challenging to train and evaluate, and may require additional techniques to ensure coherence and fluency. Ultimately, the choice between GANs and LSTMs depends on the specific text generation task and the desired output: if you need coherent and natural-sounding text, LSTMs might be the better choice, but if you want to generate creative and diverse content, GANs could be the way to go."

chars=list(set(data))

char\_to\_idx={char:i for i,char in enumerate(chars)}

idx\_to\_char={i:char for i,char in enumerate(chars)}

We make a list of all unique characters and then make two dictionaries using dictionary comprehensions

DEFINE THE DEVICE

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

If you have GPU support , the code will shift the processing to GPU otherwise processing on CPU can be slow. You can also use Google Collab for smooth excecution of you code

Making the LSTM Class

class LSTMModel(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

super(LSTMModel, self).\_\_init\_\_()

self.hidden\_size = hidden\_size

self.lstm = nn.LSTM(input\_size, hidden\_size, batch\_first=True)

self.fc = nn.Linear(hidden\_size, output\_size)

def forward(self, x):

h0 = torch.zeros(1, x.size(0), self.hidden\_size).to(x.device)

c0 = torch.zeros(1, x.size(0), self.hidden\_size).to(x.device)

out, \_ = self.lstm(x, (h0, c0))

out = self.fc(out[:, -1, :])

return out

Defining the Model

model = LSTMModel(len(chars), 16, len(chars))

criterion=nn.CrossEntropyLoss()

optimizer=torch.optim.Adam(model.parameters(),lr=0.01)

We define 16 hidden layers. The length of the input and output gate must be same . We define the Adam optimizer though you can test any other optimizer for comparing results

One hot encoding

inputs=[char\_to\_idx[ch] for ch in data[:-1]]

targets=[char\_to\_idx[ch] for ch in data[1:]]

inputs=torch.tensor(inputs,dtype=torch.long).view(-1,1)

inputs=nn.functional.one\_hot(inputs,num\_classes=len(chars)).float()

targets=torch.tensor(targets,dtype=torch.long)

Pytorch will accept data in the form of tensors. These are multi dimensional arrays of float data type. We also perform one hot encoding of tensors for efficient parsing of data and calculation of loss.

Training the model with 800 epochs.

As we are just inputting a few sentences. We can use large number of epochs for efficient training

i=0

for epoch in range(800):

model.train()

outputs=model(inputs)

loss=criterion(outputs,targets)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

i+=1

if(i%100==0):

print(f"epoch {epoch+1}, loss {loss.item()}")

Output

epoch 100, loss 2.1541919708251953

epoch 200, loss 2.0188331604003906

epoch 300, loss 1.9942307472229004

epoch 400, loss 1.986881971359253

epoch 500, loss 1.9836246967315674

epoch 600, loss 1.9818694591522217

epoch 700, loss 1.9808194637298584

epoch 800, loss 1.9801398515701294

We observe that even after 800 epochs there is slight difference in loss . we can also experiment by increasing the learning rate

TESTING THE MODEL

Let us test our model

model.eval()

test\_input=char\_to\_idx['S']

test\_input=nn.functional.one\_hot(torch.tensor(test\_input).view(-1,1),num\_classes=len(chars)).float()

pred\_output=model(test\_input)

pred\_char=torch.argmax(pred\_output,1).item()

pred\_char = idx\_to\_char[pred\_char]

print(pred\_char)

We input a single charcter from “ LSTM “ word of the input sentences its output was T.

It successfully predicted the next character.

NEXT WORD PREDICTION

Previously we made a list of all unique characters. Now will we tokenize the input data on the basis of input words. How? Simple, we simple separate the input sentence on the basis of spaces but first we must remove all puntuation marks. Here NLTK and string library makes their entrance

data = data.translate(str.maketrans('', '', string.punctuation))

words=word\_tokenize(data)

vocab=list(set(words))

words\_to\_idx={word:i for i,word in enumerate(vocab)}

idx\_to\_words={i:word for i,word in enumerate(vocab)}

We use the same approach as before to make 2 dictionaries but this time they cantain words except for characters.

DEFINING THE MODEL

vocab\_size = len(vocab)

model2 = LSTMModel(input\_size=vocab\_size,hidden\_size= 16, output\_size=vocab\_size)

criterion=nn.CrossEntropyLoss()

optimizer=torch.optim.Adam(model2.parameters(),lr=0.01)

One hot encoding

word\_inputs = [words\_to\_idx.get(ch, -1) for ch in data.split()[:-1]]

word\_targets = [words\_to\_idx.get(ch, -1) for ch in data.split()[1:]]

word\_inputs = [x for x in word\_inputs if x != -1]

word\_targets = [x for x in word\_targets if x != -1]

word\_inputs = torch.tensor(word\_inputs, dtype=torch.long).view(-1, 1)

word\_targets = torch.tensor(word\_targets, dtype=torch.long)

word\_inputs = nn.functional.one\_hot(word\_inputs, num\_classes=vocab\_size).float()

Training Loop

i=0

for epoch in range(800):

model2.train()

word\_outputs=model2(word\_inputs)

loss=criterion(word\_outputs,word\_targets)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

i+=1

if(i%100==0):

print(f"epoch {epoch+1}, loss {loss.item()}")

DEFINING THE PREDICT FUNCTION

def predict\_word(word:str):

model2.eval()

test\_input=words\_to\_idx[word]

test\_input=nn.functional.one\_hot(torch.tensor(test\_input).view(-1,1),num\_classes=len(vocab)).float()

pred\_output=model2(test\_input)

pred\_char=torch.argmax(pred\_output,1).item()

pred\_char = idx\_to\_words[pred\_char]

return pred\_char

We set the model to evaluate mode. One hot encode the input dictionary, predict the results and decode the dictionary and return the result.

TESTING OUR MODEL

We will use an intuitive approach by feeding our vocab list into the model and printing the results

**for** i **in** vocab:

output\_pred**=**predict\_word(i)

print(f"Input word is '{i}' and predicted next word is '{output\_pred}' " )

NEXT SENTENCE PREDICTION

So far we succeeded in predicting next word and character of an input sequence.Now we will move on to a more complex stage of text generation I.e. next sentence generation. Here the special power of LSTM will be observed

We will take the Alice In the Wonderland play as our main dataset.Training our LSTM on next sentence generation requires large amount of data otherwise loss computed will be Nan. The model will not correctly predict the next sequence due to lack of sufficient data leading to underfitting.

Loading the dataset

**with** open('alice.txt','r',encoding**=**'utf-8') **as** file:

text**=**file**.**read()

Defining the Dictionary

We will use a Counter data structure from Collections this time

from collections import Counter

words**=**text**.**split()

word\_count**=**Counter(words)

vocab**=**list(word\_count**.**keys())

vocab\_size**=**len(vocab)

word\_to\_idx**=**{i:word **for** word,i **in** enumerate(vocab)}

idx\_to\_Word**=**{word:i **for** word,i **in** enumerate(vocab)}

SEQUENCE\_LENGTH **=** 64 samples **=** [words[i:i**+**SEQUENCE\_LENGTH**+**1] **for** i **in** range(len(words)**-**SEQUENCE\_LENGTH)]

Making the datasets

**class** textloader(Dataset):

**def** \_\_init\_\_(self,samples,word\_to\_idx):

self**.**samples**=**samples

self**.**word\_to\_idx**=**word\_to\_idx

**def** \_\_len\_\_(self):

**return** len(self**.**samples)

**def** \_\_getitem\_\_(self,idx):

samples**=**self**.**samples[idx]

input\_seq**=**torch**.**LongTensor([self**.**word\_to\_idx[word] **for** word **in** samples[:**-**1]])

target\_seq**=**torch**.**LongTensor([self**.**word\_to\_idx[word] **for** word **in** samples[1:]])

**return** input\_seq, target\_seq

In the TextLoader class is preparing the data for a language modeling task, where the goal is to predict the next word in a sequence. The class takes in a list of samples (sequences of words) and a word\_to\_idx dictionary, which maps words to their corresponding indices. It then returns input and target sequences, where the input sequence is the sequence of words except the last one, and the target sequence is the sequence of words except the first one.

Making the dataloaders

batch\_size**=**12

dataset**=**textloader(samples,word\_to\_idx)

dataloader**=**DataLoader(dataset,batch\_size**=**batch\_size,shuffle**=True**)

print(dataset[1])

Defining the LSTM Model for sentence prediction

Here we will take a slightly different appraoch for next sentence generation.

class TextGenerationModel(nn.Module):

def \_\_init\_\_(self,vocab\_size,embedding\_dim,hidden\_size,num\_layers):

super(TextGenerationModel, self).\_\_init\_\_()

self.embedding=nn.Embedding(vocab\_size,embedding\_dim)

self.lstm=nn.LSTM(input\_size=embedding\_dim,hidden\_size=hidden\_size,num\_layers=num\_layers,batch\_first=True)

self.fc=nn.Linear(hidden\_size,vocab\_size)

self.hidden\_size=hidden\_size

self.num\_layers=num\_layers

def forward(self,x,hidden=None):

if hidden==None:

hidden=self.init\_hidden(x.shape[0])

x=self.embedding(x)

out,(h\_n,c\_n)=self.lstm(x,hidden)

out=out.contiguous().view(-1,self.hidden\_size)

out=self.fc(out)

return out,(h\_n,c\_n)

def init\_hidden(self, batch\_size):

h0 = torch.zeros(self.num\_layers, batch\_size, self.hidden\_size).to(device)

c0 = torch.zeros(self.num\_layers, batch\_size, self.hidden\_size).to(device)

return h0, c0

DEFINING THE HYPERPARAMTERS

embedding\_dim **=** 16

hidden\_size **=** 32

num\_layers **=** 1

learning\_rate **=** 0.01

epochs **=** 50

DEFINING THE MODEL

model=TextGenerationModel(vocab\_size,embedding\_dim,hidden\_size,num\_layers).to(device)

criterion=nn.CrossEntropyLoss()

optimizer=optim.Adam(model.parameters(),lr=learning\_rate)

DEFINING THE TRAINING LOOP

def train(model,epochs,dataloader,criterion):

model.train()

for epoch in range(epochs):

epoch\_loss=0

for input\_Seq,target\_Seq in dataloader:

input\_Seq,target\_Seq=input\_Seq.to(device),target\_Seq.to(device)

outputs,\_=model(input\_Seq)

loss=criterion(outputs,target\_Seq.view(-1))

optimizer.zero\_grad()

loss.backward()

optimizer.step()

epoch\_loss+=loss.detach().cpu().numpy()

epoch\_loss /= len(dataloader)

print(f"Epoch {epoch} loss: {epoch\_loss:.3f}")

TRAINING THE MODEL

train(model,epochs,dataloader,criterion)

Saving the model

torch**.**save(model**.**state\_dict(), 'text generator.pth')

PREDICTING THE OUTPUT

**def** generate\_text(geenratory,start,num\_words):

geenratory**.**eval()

words**=**start**.**split()

**for** \_ **in** range(num\_words):

input\_seq**=**torch**.**LongTensor([word\_to\_idx[word] **for** word **in** words[**-**SEQUENCE\_LENGTH:]])**.**unsqueeze(0)**.**to(device)

h,c**=**geenratory**.**init\_hidden(1)

output,(h,c)**=**geenratory(input\_seq,(h,c))

next\_token**=**output**.**argmax(1)[**-**1]**.**item()

words**.**append(idx\_to\_Word[next\_token])

**return** " "**.**join(words)

print('Generated text is: ',generate\_text(model,'unless it was all ridges and furrows;',num\_words**=**100))

Conclusion

We have successfully defined the 3 levels of text generation using LSTM. In case of low quality results, you can try to change the hyper paramaters like learning rate and epochs to minimize the loss.