A Comprehensive Guide to Build your own Language Model in Python!

MOHD SANAD ZAKI RIZVI

Overview

- Language models are a crucial component in the Natural Language Processing (NLP) journey
- These language models power all the popular NLP applications we are familiar with – Google Assistant, Siri, Amazon's Alexa, etc.
- We will go from basic language models to advanced ones in Python here

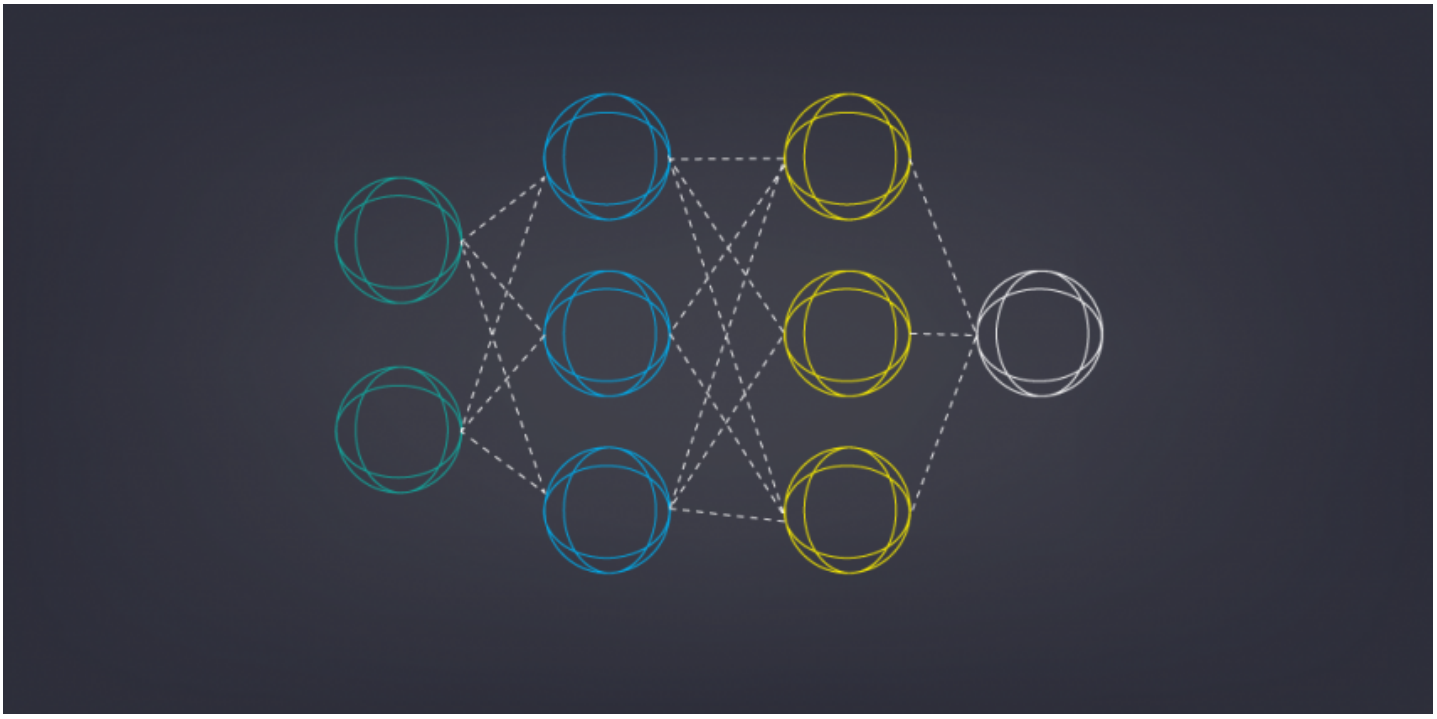
Introduction

> *"We tend to look through language and not realize how much power language has."*

Language is such a powerful medium of communication. We have the ability to build projects from scratch using the nuances of language. It's what drew me to Natural Language Processing (NLP) in the first place.

I'm amazed by the vast array of tasks I can perform with NLP – text summarization, generating completely new pieces of text, predicting what word comes next (Google's autofill), among others. Do you know what is common among all these NLP tasks?

They are all powered by language models! Honestly, these language models are a crucial first step for most of the advanced NLP tasks.



In this article, we will cover the length and breadth of language models. We will begin from basic language models that can be created with a few lines of Python code and move to the State-of-the-Art language models that are trained using humongous data and are being currently used by the likes of Google, Amazon, and Facebook, among others.

So, tighten your seatbelts and brush up your linguistic skills – we are heading into the wonderful world of Natural Language Processing!

Are you new to NLP? Confused about where to begin? You should check out this comprehensive course designed by experts with decades of industry experience:

- *Natural Language Processing (NLP) with Python*

Table of Contents

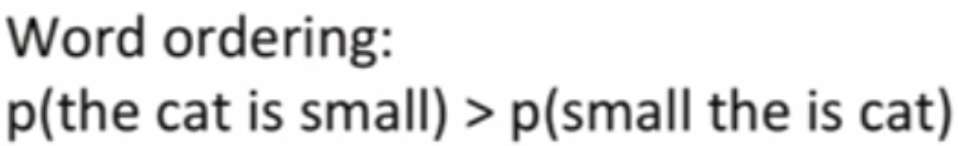
What is a Language Model in NLP?

> *"You shall know the nature of a word by the company it keeps." – John Rupert Firth*

A language model learns to predict the probability of a sequence of words. But why do we need to learn the probability of words? Let's understand that with an example.

I'm sure you have used Google Translate at some point. We all use it to translate one language to another for varying reasons. This is an example of a popular NLP application called Machine Translation.

In Machine Translation, you take in a bunch of words from a language and convert these words into another language. Now, there can be many potential translations that a system might give you and you will want to compute the probability of each of these translations to understand which one is the most accurate.



In the above example, we know that the probability of the first sentence will be more than the second, right? That's how we arrive at the right translation.

This ability to model the rules of a language as a probability gives great power for NLP related tasks. Language models are used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval, and many other daily tasks.

Types of Language Models

There are primarily two types of Language Models:

1. **Statistical Language Models:** These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM) and certain linguistic rules to learn the probability distribution of words
2. **Neural Language Models:** These are new players in the NLP town and have surpassed the statistical language models in their effectiveness. They use different kinds of Neural Networks to model language

Now that you have a pretty good idea about Language Models, let's start building one!

Building an N-gram Language Model

What are N-grams (unigram, bigram, trigrams)?

> *An N-gram is a sequence of N tokens (or words).*

Let's understand N-gram with an example. Consider the following sentence:

"I love reading blogs about data science on Analytics Vidhya."

A 1-gram (or unigram) is a one-word sequence. For the above sentence, the unigrams would simply be: "I", "love", "reading", "blogs", "about", "data", "science", "on", "Analytics", "Vidhya".

A 2-gram (or bigram) is a two-word sequence of words, like "I love", "love reading", or "Analytics Vidhya". And a 3-gram (or trigram) is a three-word sequence of words like "I love reading", "about data science" or "on Analytics Vidhya".

Fairly straightforward stuff!

How do N-gram Language Models work?

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. If we have a good N-gram model, we can predict p(w | h) – what is the probability of seeing the word *w* given a history of previous words *h* – where the history contains n-1 words.

We must estimate this probability to construct an N-gram model.

We compute this probability in two steps:

1. Apply the chain rule of probability
2. We then apply a very strong simplification assumption to allow us to compute p(w1…ws) in an easy manner

The chain rule of probability is:

```
p(w1...ws) = p(w1) . p(w2 | w1) . p(w3 | w1 w2) . p(w4 | w1 w2 w3) ..... p(wn | w1...wn-1)
```

So what is the chain rule? It tells us how to compute the joint probability of a sequence by using the conditional probability of a word given previous words.

But we do not have access to these conditional probabilities with complex conditions of up to n-1 words. So how do we proceed?

This is where we introduce a simplification assumption. We can assume for all conditions, that:

```
p(wk | w1...wk-1) = p(wk | wk-1)
```

Here, we **approximate** the history (the context) of the word *wk* by looking only at the last word of the context. This assumption is called the **Markov assumption**. (We used it here with a simplified context of length 1 – which corresponds to a bigram model – we could use larger fixed-sized histories in general).

Building a Basic Language Model

Now that we understand what an N-gram is, let's build a basic language model using trigrams of the Reuters corpus. Reuters corpus is a collection of 10,788 news documents totaling 1.3 million words. We can build a language model in a few lines of code using the NLTK package:

```python
# code courtesy of https://nlpforhackers.io/language-models/

from nltk.corpus import reuters
from nltk import bigrams, trigrams
from collections import Counter, defaultdict

# Create a placeholder for model
model = defaultdict(lambda: defaultdict(lambda: 0))

# Count frequency of co-occurance
for sentence in reuters.sents():
    for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
        model[(w1, w2)][w3] += 1

# Let's transform the counts to probabilities
for w1_w2 in model:
    total_count = float(sum(model[w1_w2].values()))
    for w3 in model[w1_w2]:
        model[w1_w2][w3] /= total_count
```

ngram_lm.py hosted with ❤ by **GitHub** **view raw**

The code above is pretty straightforward. We first split our text into trigrams with the help of NLTK and then calculate the frequency in which each combination of the trigrams occurs in the dataset.

We then use it to calculate probabilities of a word, given the previous two words. That's essentially what gives us our Language Model!

Let's make simple predictions with this language model. We will start with two simple words – "today the". We want our model to tell us what will be the next word:

```
# predict the next word
dict(model["today", "the"])
```

```
{'Bank': 0.05555555555555555,
 'European': 0.05555555555555555,
 'Higher': 0.05555555555555555,
 'Italian': 0.05555555555555555,
 'Turkish': 0.05555555555555555,
 'company': 0.16666666666666666,
 'emirate': 0.05555555555555555,
 'increase': 0.05555555555555555,
 'newspaper': 0.05555555555555555,
 'options': 0.05555555555555555,
 'overseas': 0.05555555555555555,
 'pound': 0.05555555555555555,
 'price': 0.1111111111111111,
 'public': 0.05555555555555555,
 'time': 0.05555555555555555}
```

So we get predictions of all the possible words that can come next with their respective probabilities. Now, if we pick up the word "price" and again make a prediction for the words "the" and "price":

```
[22] # predict the next word
     dict(model["the", "price"])
```

```
 'cut': 0.009302325581395349,
 'cuts': 0.009302325581395349,
 'difference': 0.004651162790697674,
 'differentials': 0.009302325581395349,
 'drop': 0.004651162790697674,
 'effect': 0.004651162790697674,
 'evolution': 0.004651162790697674,
 'factor': 0.004651162790697674,
 'fall': 0.004651162790697674,
 'for': 0.05116279069767442,
 'freeze': 0.009302325581395349,
 'from': 0.004651162790697674,
 'gap': 0.004651162790697674,
 'guaranteed': 0.004651162790697674,
 'had': 0.004651162790697674,
```

If we keep following this process iteratively, we will soon have a coherent sentence! Here is a script to play around with generating a random piece of text using our n-gram model:

```
1  # code courtesy of https://nlpforhackers.io/language-models/
2
```

```
import random

# starting words
text = ["today", "the"]
sentence_finished = False

while not sentence_finished:
  # select a random probability threshold
  r = random.random()
  accumulator = .0

  for word in model[tuple(text[-2:])].keys():
      accumulator += model[tuple(text[-2:])][word]
      # select words that are above the probability threshold
      if accumulator >= r:
          text.append(word)
          break

  if text[-2:] == [None, None]:
      sentence_finished = True

print (' '.join([t for t in text if t]))
```

ngram_textgen.py hosted with ❤ by **GitHub** **view raw**

And here is some of the text generated by our model:

```
today the company ' s stock rose after brokerage house hurt the economic area , a delegate to cocoa , and the dealers ha
today the time of a lower soybean exports and reduce trade friction and discuss the crisis , Chancellor of the price of
today the company .
today the Bank had estimated the profit - taking stunts the move showed that total fell to 4 , 682 ) tonnes of U . S . '
today the pound crashed to four billion mark rise was 4 . 133 IRAN 1 . 42 203 . 7 pct compared with a minimum of 40 mln
today the newspaper reported today that U . S . Government securities market to U . S . Dlrs worth of Japanese insurers
```

Pretty impressive! Even though the sentences feel slightly off (maybe because the Reuters dataset is mostly news), they are very coherent given the fact that we just created a model in 17 lines of Python code and a really small dataset.

This is the same underlying principle which the likes of Google, Alexa, and Apple use for language modeling.

Limitations of N-gram approach to Language Modeling

N-gram based language models do have a few drawbacks:

1. The higher the N, the better is the model usually. But this leads to lots of computation overhead that requires large computation power in terms of RAM
2. N-grams are a sparse representation of language. This is because we build the model based on the probability of words co-occurring. It will give zero probability to all the words that are not present in the training corpus

Building a Neural Language Model

> *"Deep Learning waves have lapped at the shores of computational linguistics for several years now, but 2015 seems like the year when the full force of the tsunami hit the major Natural Language Processing (NLP) conferences." – Dr. Christopher D. Manning*

Deep Learning has been shown to perform really well on many NLP tasks like Text Summarization, Machine Translation, etc. and since these tasks are essentially built upon Language Modeling, there has been a tremendous research effort with great results to use Neural Networks for Language Modeling.

We can essentially build two kinds of language models – character level and word level. And even under each category, we can have many subcategories based on the simple fact of how we are framing the learning problem. We will be taking the most straightforward approach – building a character-level language model.

Understanding the problem statement

IN CONGRESS, JULY 4, 1776.

A DECLARATION

BY THE REPRESENTATIVES OF THE

UNITED STATES OF AMERICA,

IN GENERAL CONGRESS ASSEMBLED.

Does the above text seem familiar? It's the US Declaration of Independence! The dataset we will use is the text from this Declaration.

This is a historically important document because it was signed when the United States of America got independence from the British. I used this document as it covers a lot of different topics in a single space. It's also the right size to experiment with because we are training a character-level language model which is comparatively more intensive to run as compared to a word-level language model.

>

> *The problem statement is to train a language model on the given text and then generate text given an input text in such a way that it looks straight out of this document and is grammatically correct and legible to read.*

You can download the dataset from here. Let's begin!

Import the libraries

```
import numpy as np
import pandas as pd
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import LSTM, Dense, GRU, Embedding
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

lm_import.py hosted with ❤ by GitHub view raw

Read the dataset

You can directly read the dataset as a string in Python:

```
data_text = """The unanimous Declaration of the thirteen united States of America, When in the Course

We hold these truths to be self-evident, that all men are created equal, that they are endowed by thei

He has refused his Assent to Laws, the most wholesome and necessary for the public good.

He has forbidden his Governors to pass Laws of immediate and pressing importance, unless suspended in

He has refused to pass other Laws for the accommodation of large districts of people, unless those peo

He has called together legislative bodies at places unusual, uncomfortable, and distant from the depos

He has dissolved Representative Houses repeatedly, for opposing with manly firmness his invasions on t

He has refused for a long time, after such dissolutions, to cause others to be elected; whereby the Le

He has endeavoured to prevent the population of these States; for that purpose obstructing the Laws fo

He has obstructed the Administration of Justice, by refusing his Assent to Laws for establishing Judic

He has made Judges dependent on his Will alone, for the tenure of their offices, and the amount and pa
```

```
22
23  He has erected a multitude of New Offices, and sent hither swarms of Officers to harrass our people, a
24
25  He has kept among us, in times of peace, Standing Armies without the Consent of our legislatures.
26
27  He has affected to render the Military independent of and superior to the Civil power.
28
29  He has combined with others to subject us to a jurisdiction foreign to our constitution, and unacknowl
30
31  For Quartering large bodies of armed troops among us:
32
33  For protecting them, by a mock Trial, from punishment for any Murders which they should commit on the
34
35  For cutting off our Trade with all parts of the world:
36
37  For imposing Taxes on us without our Consent:
38
39  For depriving us in many cases, of the benefits of Trial by Jury:
40
41  For transporting us beyond Seas to be tried for pretended offences
42
43  For abolishing the free System of English Laws in a neighbouring Province, establishing therein an Arb
44
45  For taking away our Charters, abolishing our most valuable Laws, and altering fundamentally the Forms
46
47  For suspending our own Legislatures, and declaring themselves invested with power to legislate for us
48
49  He has abdicated Government here, by declaring us out of his Protection and waging War against us.
50
51  He has plundered our seas, ravaged our Coasts, burnt our towns, and destroyed the lives of our people.
52
53  He is at this time transporting large Armies of foreign Mercenaries to compleat the works of death, de
54
55  He has constrained our fellow Citizens taken Captive on the high Seas to bear Arms against their Count
56
57  He has excited domestic insurrections amongst us, and has endeavoured to bring on the inhabitants of o
58
59  In every stage of these Oppressions We have Petitioned for Redress in the most humble terms: Our repea
60
61  Nor have We been wanting in attentions to our Brittish brethren. We have warned them from time to time
62
63  We, therefore, the Representatives of the united States of America, in General Congress, Assembled, ap
```


Preprocessing the Text Data

We perform basic text preprocessing since this data does not have much noise. We lower case all the words to maintain uniformity and remove words with length less than 3:

```
import re

def text_cleaner(text):
    # lower case text
    newString = text.lower()
    newString = re.sub(r"'s\b","",newString)
    # remove punctuations
    newString = re.sub("[^a-zA-Z]", " ", newString)
    long_words=[]
    # remove short word
    for i in newString.split():
        if len(i)>=3:
            long_words.append(i)
    return (" ".join(long_words)).strip()

# preprocess the text
data_new = text_cleaner(data_text)
```

lm_preprocess.py hosted with ❤ by **GitHub** **view raw**

Once the preprocessing is complete, it is time to create training sequences for the model.

Creating Sequences

The way this problem is modeled is we take in 30 characters as context and ask the model to predict the next character. Now, 30 is a number which I got by trial and error and you can experiment with it too. You essentially need enough characters in the input sequence that your model is able to get the context.

```
def create_seq(text):
    length = 30
    sequences = list()
    for i in range(length, len(text)):
        # select sequence of tokens
        seq = text[i-length:i+1]
        # store
        sequences.append(seq)
    print('Total Sequences: %d' % len(sequences))
    return sequences

# create sequences
sequences = create_seq(data_new)
```

lm_seq.py hosted with ❤ by **GitHub** **view raw**

Let's see how our training sequences look like:

```
Total Sequences: 7052
['the unanimous declaration the t',
 'he unanimous declaration the th',
 'e unanimous declaration the thi',
 ' unanimous declaration the thir',
 'unanimous declaration the thirt']
```

Encoding Sequences

Once the sequences are generated, the next step is to encode each character. This would give us a sequence of numbers.

```
# create a character mapping index
chars = sorted(list(set(data_new)))
mapping = dict((c, i) for i, c in enumerate(chars))

def encode_seq(seq):
    sequences = list()
    for line in seq:
        # integer encode line
        encoded_seq = [mapping[char] for char in line]
        # store
        sequences.append(encoded_seq)
    return sequences

# encode the sequences
sequences = encode_seq(sequences)
```

lm_seq_encode.py hosted with ❤ by **GitHub** **view raw**

So now, we have sequences like this:

```
array([[20,  8,  5,  0, 21, 14,  1, 14,  9, 13, 15, 21, 19,  0,  4,  5,
         3, 12,  1, 18,  1, 20,  9, 15, 14,  0, 20,  8,  5,  0, 20],
       [ 8,  5,  0, 21, 14,  1, 14,  9, 13, 15, 21, 19,  0,  4,  5,  3,
        12,  1, 18,  1, 20,  9, 15, 14,  0, 20,  8,  5,  0, 20,  8],
       [ 5,  0, 21, 14,  1, 14,  9, 13, 15, 21, 19,  0,  4,  5,  3, 12,
         1, 18,  1, 20,  9, 15, 14,  0, 20,  8,  5,  0, 20,  8,  9],
       [ 0, 21, 14,  1, 14,  9, 13, 15, 21, 19,  0,  4,  5,  3, 12,  1,
        18,  1, 20,  9, 15, 14,  0, 20,  8,  5,  0, 20,  8,  9, 18],
       [21, 14,  1, 14,  9, 13, 15, 21, 19,  0,  4,  5,  3, 12,  1, 18,
         1, 20,  9, 15, 14,  0, 20,  8,  5,  0, 20,  8,  9, 18, 20]])
```

Create Training and Validation set

Once we are ready with our sequences, we split the data into training and validation splits. This is because while training, I want to keep a track of how good my language model is working with unseen data.

```
from sklearn.model_selection import train_test_split

# vocabulary size
vocab = len(mapping)
sequences = np.array(sequences)
# create X and y
X, y = sequences[:,:-1], sequences[:,-1]
# one hot encode y
y = to_categorical(y, num_classes=vocab)
# create train and validation sets
X_tr, X_val, y_tr, y_val = train_test_split(X, y, test_size=0.1, random_state=42)

print('Train shape:', X_tr.shape, 'Val shape:', X_val.shape)
```

lm_val.py hosted with ❤ by **GitHub** **view raw**

Model Building

Time to build our language model!

I have used the embedding layer of Keras to learn a 50 dimension embedding for each character. This helps the model in understanding complex relationships between characters. I have also used a GRU layer as the base model, which has 150 timesteps. Finally, a Dense layer is used with a softmax activation for prediction.

```
# define model
model = Sequential()
model.add(Embedding(vocab, 50, input_length=30, trainable=True))
model.add(GRU(150, recurrent_dropout=0.1, dropout=0.1))
model.add(Dense(vocab, activation='softmax'))
print(model.summary())

# compile the model
model.compile(loss='categorical_crossentropy', metrics=['acc'], optimizer='adam')
# fit the model
model.fit(X_tr, y_tr, epochs=100, verbose=2, validation_data=(X_val, y_val))
```

lm_model.py hosted with ❤ by **GitHub** **view raw**

Inference

Once the model has finished training, we can generate text from the model given an input sequence using the below code:

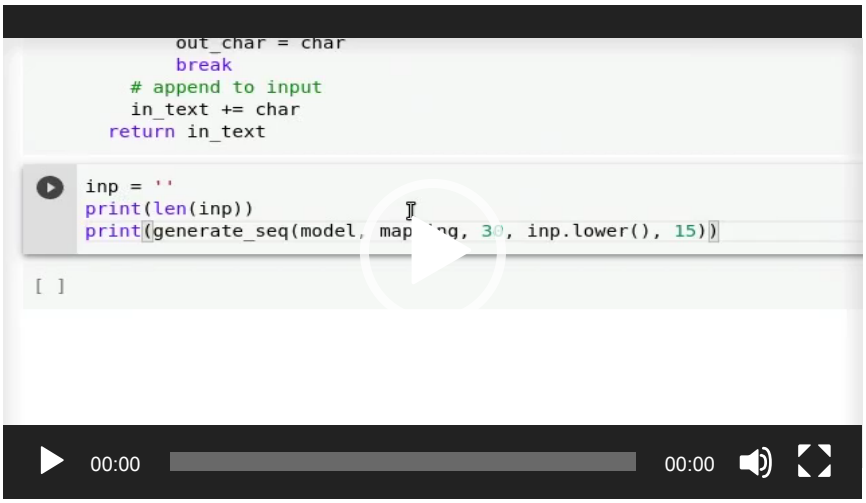
```
# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
        in_text = seed_text
```

```
        # generate a fixed number of characters
        for _ in range(n_chars):
                # encode the characters as integers
                encoded = [mapping[char] for char in in_text]
                # truncate sequences to a fixed length
                encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
                # predict character
                yhat = model.predict_classes(encoded, verbose=0)
                # reverse map integer to character
                out_char = ''
                for char, index in mapping.items():
                        if index == yhat:
                                out_char = char
                                break
                # append to input
                in_text += char
        return in_text
```

lm_infer.py hosted with ❤ by GitHub view raw

Results

Let's put our model to the test. In the video below, I have given different inputs to the model. Let's see how it performs:



Notice just how sensitive our language model is to the input text! Small changes like adding a space after “of” or “for” completely changes the probability of occurrence of the next characters because when we write space, we mean that a new word should start.

Additionally, when we do not give space, it tries to predict a word that will have these as starting characters (like “for” can mean “foreign”).

Also, note that **almost none of the combinations predicted by the model exist in the original training data**. So our model is actually building words based on its understanding of the rules of the English language and the vocabulary it has seen during training.

Natural Language Generation using OpenAI’s GPT-2

We have so far trained our own models to generate text, be it predicting the next word or generating some text with starting words. But that is just scratching the surface of what language models are capable of!

Leading research labs have trained much more complex language models on humongous datasets that have led to some of the biggest breakthroughs in the field of Natural Language Processing.

In February 2019, **OpenAI** started quite a storm through its release of a new transformer-based language model called **GPT-2.** GPT-2 is a transformer-based generative language model that was trained on 40GB of curated text from the internet.

You can read more about GPT-2 here:

- OpenAI’s GPT-2: A Simple Guide to Build the World’s Most Advanced Text Generator in Python

So, let’s see GPT-2 in action!

About PyTorch-Transformers

Before we can start using GPT-2, let's know a bit about the PyTorch-Transformers library. We will be using this library we will use to load the pre-trained models.

PyTorch-Transformers provides state-of-the-art pre-trained models for Natural Language Processing (NLP).

Most of the State-of-the-Art models require tons of training data and days of training on expensive GPU hardware which is something only the big technology companies and research labs can afford. But by using PyTorch-Transformers, now anyone can utilize the power of State-of-the-Art models!

Installing PyTorch-Transformers on your Machine

Installing Pytorch-Transformers is pretty straightforward in Python. You can simply use pip install:

```
pip install pytorch-transformers
```

or if you are working on Colab:

```
!pip install pytorch-transformers
```

Since most of these models are GPU-heavy, I would suggest working with Google Colab for this part of the article.

Sentence completion using GPT-2

Let's build our own sentence completion model using GPT-2. We'll try to predict the next word in the sentence:

"what is the fastest car in the _____"

I chose this example because this is the first suggestion that Google's text completion gives. Here is the code for doing the same:

```
# Import required libraries
import torch
from pytorch_transformers import GPT2Tokenizer, GPT2LMHeadModel

# Load pre-trained model tokenizer (vocabulary)
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# Encode a text inputs
text = "What is the fastest car in the"
indexed_tokens = tokenizer.encode(text)

# Convert indexed tokens in a PyTorch tensor
tokens_tensor = torch.tensor([indexed_tokens])

# Load pre-trained model (weights)
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Set the model in evaluation mode to deactivate the DropOut modules
model.eval()

# If you have a GPU, put everything on cuda
tokens_tensor = tokens_tensor.to('cuda')
model.to('cuda')
```

```
24
25  # Predict all tokens
26  with torch.no_grad():
27      outputs = model(tokens_tensor)
28      predictions = outputs[0]
29
30  # Get the predicted next sub-word
31  predicted_index = torch.argmax(predictions[0, -1, :]).item()
32  predicted_text = tokenizer.decode(indexed_tokens + [predicted_index])
33
34  # Print the predicted word
35  print(predicted_text)
```

gpt_next_word.py hosted with ❤ by **GitHub** **view raw**

Here, we tokenize and index the text as a sequence of numbers and pass it to the *GPT2LMHeadModel*. This is the GPT2 model transformer with a language modeling head on top (linear layer with weights tied to the input embeddings).

```
-

INFO:pytorch_transformers.modeling_utils:loading weights file https://s3.amazonaws.com/mo
Predicted text: What is the fastest car in the world
```

Awesome! The model successfully predicts the next word as **"world"**. This is pretty amazing as this is what Google was suggesting. I recommend you try this model with different input sentences and see how it performs while predicting the next word in a sentence.

Conditional Text Generation using GPT-2

Now, we have played around by predicting the next word and the next character so far. Let's take text generation to the next level by generating an entire paragraph from an input piece of text!

Let's see what our models generate for the following input text:

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
```

This is the first paragraph of the poem "The Road Not Taken" by Robert Frost. Let's put GPT-2 to work and generate the next paragraph of the poem.

We will be using the readymade script that PyTorch-Transformers provides for this task. Let's clone their repository first:

```
!git clone https://github.com/huggingface/pytorch-transformers.git
```

Now, we just need a single command to start the model!

```
!python pytorch-transformers/examples/run_generation.py \
    --model_type=gpt2 \
    --length=100 \
    --model_name_or_path=gpt2 \
```

gpt_text_gen hosted with ❤ by **GitHub** **view raw**

Let's see what output our GPT-2 model gives for the input text:

```
And with my little eyes full of hearth and perfumes,
I saw the blue of Scotland,
And this powerful lieeth close
By wind's with profit and grief,
And at this time came and passed by,
At how often thro' places
And always this path was fresh Through one winter down.
And, stung by the wild storm,
Appeared half-blind, yet in that gloomy castle.
```

Isn't that crazy?! The output almost perfectly fits in the context of the poem and appears as a good continuation of the first paragraph of the poem.

End Notes

Quite a comprehensive journey, wasn't it? We discussed what language models are and how we can use them using the latest state-of-the-art NLP frameworks. And the end result was so impressive!

You should consider this as the beginning of your ride into language models. I encourage you to play around with the code I've showcased here. This will really help you build your own knowledge and skillset while expanding your opportunities in NLP.