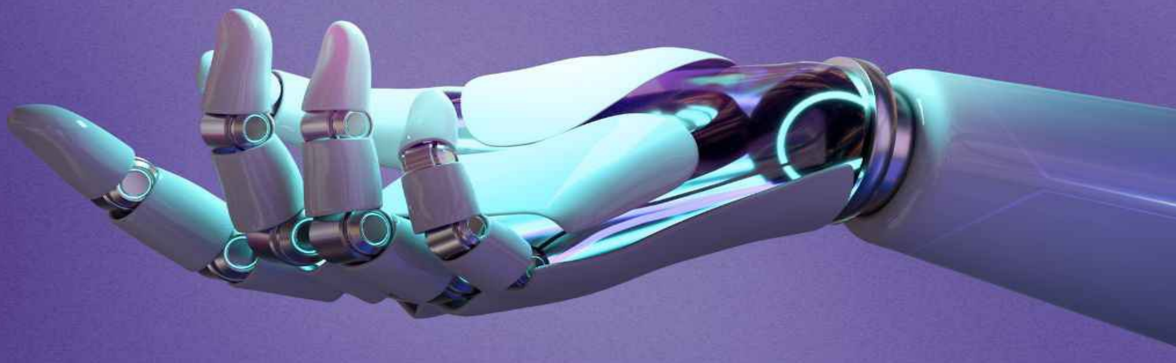
NATURAL LANGUAGE PROCESSING PRACTICAL

USING TRANSFORMERS WITH PYTHON



WRITTEN BY

TONY SNAKE

About the Authors

Tony Snake was received Bachelor of Computer Science from the American University, and Bachelor of Business Administration from the American University, USA.

He is becoming Ph.D. Candidate of Department of Data Informatics, (National) Korea Maritime and Ocean University, Busan 49112, Republic of Korea (South Korea).

His research interests are social network analysis, big data, AI and robotics.

He received Best Paper Award the 15th International Conference on Multimedia Information Technology and Applications (MITA 2019)

Table of Contents

Contents

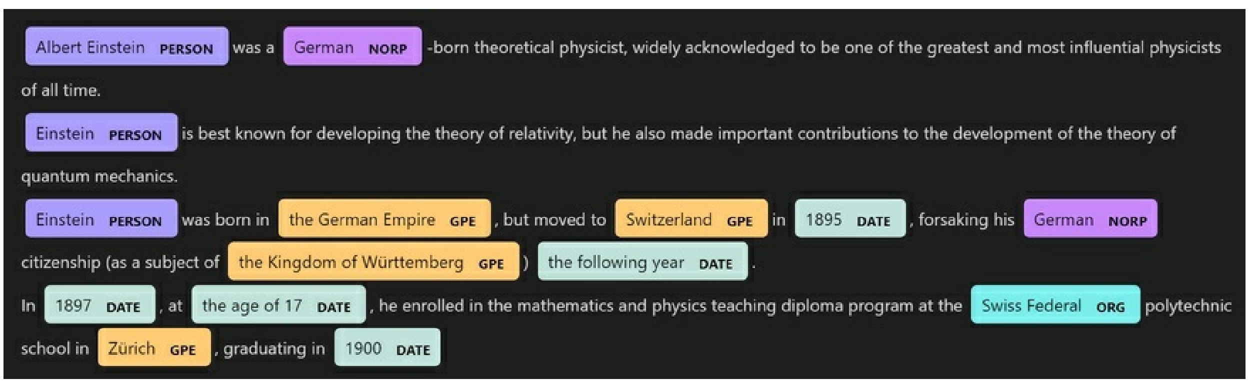
Natural Language Processing Practical using Transformers with Python

CHAPTER 1: Named Entity Recognition using Transformers and Spacy in Python

Learn how you can perform named entity recognition using HuggingFace Transformers and spaCy libraries in Python.

[Named Entity Recognition (NER)](#) is a typical [natural language processing (NLP)](#) task that automatically identifies and recognizes predefined entities in a given text. Entities like person names, organizations, dates and times, and locations are valuable information to extract from unstructured and unlabeled raw text.

At the end of this tutorial, you will be able to perform named entity recognition on any given English text with [HuggingFace Transformers](#) and [SpaCy](#) in Python; here's an example of the resulting NER:



SpaCy is an open-source library in Python for advanced NLP. It is built on the latest research and designed to be used in real-world products. We'll be using two NER models on SpaCy, namely the regular `en_core_web_sm` and the transformer `en_core_web_trf`. We'll also use spaCy's NER amazing visualizer.

To get started, let's install the required libraries for this tutorial. First, installing transformers:

```
$ pip install --upgrade transformers sentencepiece
```

Next, we need to install `spacy` and `spacy-transformers`. To do that, I've grabbed the latest `.whl` file from the `spacy-models` releases for installation:

```
$ pip install https://github.com/explosion/spacy-models/releases/download/en_core_web_trf-3.2.0/en_core_web_trf-3.2.0-py3-none-any.whl
```

Of course, if you're reading this tutorial in the future, make sure to get the latest release from [this page](#) if you encounter any problems regarding the above command.

Next, we have to download the spaCy's `en_core_web_sm` regular model:

```
$ python -m spacy download en_core_web_sm
```

`en_core_web_sm` is an English model pipeline optimized for CPU. It is small, with only 13MB in size, and under the MIT license. For larger models, you can use `en_core_web_md` for medium-sized and `en_core_web_lg` for the large one.

Once done with the installation, let's get started with the code:

```
import spacy
from transformers import *
```

For this tutorial, we'll be performing NER on this text that I've grabbed from Wikipedia:

```
# sample text from Wikipedia
```

```python
text = """
Albert Einstein was a German-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time.
Einstein is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.
Einstein was born in the German Empire, but moved to Switzerland in 1895, forsaking his German citizenship (as a subject of the Kingdom of Württemberg) the following year.
In 1897, at the age of 17, he enrolled in the mathematics and physics teaching diploma program at the Swiss Federal polytechnic school in Zürich, graduating in 1900
"""
```

NER with Transformers

We'll be using the HuggingFace Transformers' `pipeline` API for loading the models:

```python
# load BERT model fine-tuned for Named Entity Recognition (NER)
ner = pipeline("ner", model="dslim/bert-base-NER")
```

We're using a BERT model (`bert-base-encased`) that was fine-tuned on the CoNLL-2003 Named Entity Recognition dataset. You can use `dslim/bert-large-NER` for a larger version of this one.

Let's extract the entities for our text using this model:

```python
# perform inference on the transformer model
doc_ner = ner(text)
# print the output
```

```
doc_ner
```

Output:

```
[{'end': 7,
  'entity': 'B-PER',
  'index': 1,
  'score': 0.99949145,
  'start': 1,
  'word': 'Albert'},
 {'end': 16,
  'entity': 'I-PER',
  'index': 2,
  'score': 0.998417,
  'start': 8,
  'word': 'Einstein'},
 {'end': 29,
  'entity': 'B-MISC',
  'index': 5,
  'score': 0.99211043,
  'start': 23,
  'word': 'German'},
 {'end': 158,
  'entity': 'B-PER',
  'index': 28,
  'score': 0.99736506,
  'start': 150,
  'word': 'Einstein'},
 {'end': 318,
```

```
 'entity': 'B-PER',
 'index': 55,
 'score': 0.9977113,
 'start': 310,
 'word': 'Einstein'},
{'end': 341,
 'entity': 'B-LOC',
 'index': 60,
 'score': 0.50242233,
 'start': 335,
 'word': 'German'},
{'end': 348,
 'entity': 'I-LOC',
 'index': 61,
 'score': 0.95330054,
 'start': 342,
 'word': 'Empire'},
{'end': 374,
 'entity': 'B-LOC',
 'index': 66,
 'score': 0.99978524,
 'start': 363,
 'word': 'Switzerland'},
{'end': 404,
 'entity': 'B-MISC',
 'index': 74,
 'score': 0.9995827,
 'start': 398,
 'word': 'German'},
{'end': 460,
```

```
 'entity': 'B-LOC',
 'index': 84,
 'score': 0.9994709,
 'start': 449,
 'word': 'Württemberg'},
{'end': 590,
 'entity': 'B-MISC',
 'index': 111,
 'score': 0.9888771,
 'start': 585,
 'word': 'Swiss'},
{'end': 627,
 'entity': 'B-LOC',
 'index': 119,
 'score': 0.9977405,
 'start': 621,
 'word': 'Zürich'}]
```

As you can see, the output is a list of dictionaries that has the start and end positions of the entity in the text, the prediction score, the word itself, the index, and the entity name.

The named entities of this datasets are:

- `O`: Outside of a named entity.
- `B-MIS`: Beginning of a miscellaneous entity right after another miscellaneous entity.
- `I-MIS`: Miscellaneous entity.
- `B-PER`: Beginning of a person’s name right after another person’s name.
- `I-PER`: Person’s name.

- `B-ORG`: The beginning of an organization right after another organization.
- `I-ORG`: Organization.
- `B-LOC`: Beginning of a location right after another location.
- `I-LOC`: Location.

Next, let's make a function that uses spaCy to visualize this Python dictionary:

```
def get_entities_html(text, ner_result, title=None):
  """Visualize NER with the help of SpaCy"""
  ents = []
  for ent in ner_result:
    e = {}
    # add the start and end positions of the entity
    e["start"] = ent["start"]
    e["end"] = ent["end"]
    # add the score if you want in the label
    # e["label"] = f"{ent["entity"]}-{ent['score']:.2f}"
    e["label"] = ent["entity"]
    if ents and -1 <= ent["start"] - ents[-1]["end"] <= 1 and ents[-1]["label"] ==
e["label"]:
      # if the current entity is shared with previous entity
      # simply extend the entity end position instead of adding a new one
      ents[-1]["end"] = e["end"]
      continue
    ents.append(e)
  # construct data required for displacy.render() method
  render_data = [
    {
      "text": text,
```

```
    "ents": ents,
    "title": title,
  }
]
spacy.displacy.render(render_data, style="ent", manual=True,
jupyter=True)
```

The above function uses the `spacy.displacy.render()` function to render a named entity extracted text. We use `manual=True` indicating that's a manual visualization and not a spaCy document. We also set `jupyter` to `True` as we're currently on a Juypter notebook or Colab.

The whole purpose of the `for` loop is to construct a list of dictionaries with the `start` and `end` positions, and the entity's label. We also check to see if there are some same entities nearby, so we combine them.

Let's call it:

```
# get HTML representation of NER of our text
get_entities_html(text, doc_ner)
```

AlbertB-PER EinsteinI-PER was a GermanB-MISC-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time. EinsteinB-PER is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.EinsteinB-PER was born in the GermanB-LOC EmpireI-LOC, but moved to SwitzerlandB-LOC in 1895, forsaking his GermanB-MISC citizenship (as a subject of the Kingdom of WürttembergB-LOC) the following year. In 1897, at the age of 17, he enrolled in the mathematics and physics teaching diploma program at the SwissB-MISC Federal polytechnic school in ZürichB-LOC, graduating in 1900

Next, let's load another relatively larger and better model that is based on `roberta-large`:

```
# load roberta-large model
ner2 = pipeline("ner", model="xlm-roberta-large-finetuned-conll03-english")
```

Performing inference:

```
# perform inference on this model
doc_ner2 = ner2(text)
```

Visualizing:

```
# get HTML representation of NER of our text
get_entities_html(text, doc_ner2)
```

Albert EinsteinI-PER was a GermanI-MISC-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time. EinsteinI-PER is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.EinsteinI-PER was born in the German EmpireI-LOC, but moved to SwitzerlandI-LOC in 1895, forsaking his GermanI-MISC citizenship (as a subject of the Kingdom of WürttembergI-LOC) the following year. In 1897, at the age of 17, he enrolled in the mathematics and physics teaching diploma program at the SwissI-MISC FederalI-ORG polytechnic school in ZürichI-LOC, graduating in 1900

As you can see, now it's improved, naming Albert Einstein as a single entity and also the Kingdom of Wurttemberg.

There are a lot of other models that were fine-tuned on the same dataset. Here's yet another one:

```
# load yet another roberta-large model
ner3 = pipeline("ner", model="Jean-Baptiste/roberta-large-ner-english")
# perform inference on this model
doc_ner3 = ner3(text)
# get HTML representation of NER of our text
get_entities_html(text, doc_ner3)
```

Albert EinsteinPER was a German-bornMISC theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time. EinsteinPER is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.EinsteinPER was born in the German EmpireLOC, but moved to SwitzerlandLOC in 1895, forsaking his GermanMISC citizenship (as a subject of the Kingdom of WürttembergLOC) the following year. In 1897, at the age of 17, he enrolled in the mathematics and physics teaching diploma program at the SwissMISC FederalORG polytechnic school in ZürichLOC, graduating in 1900

This model, however, only has `PER`, `MISC`, `LOC`, and `ORG` entities. SpaCy automatically colors the familiar entities.

NER with SpaCy

To perform NER using SpaCy, we must first load the model using `spacy.load()` function:

```
# load the English CPU-optimized pipeline
nlp = spacy.load("en_core_web_sm")
```

We're loading the model we've downloaded. Make sure you download the model you want to use before loading it here. Next, let's generate our document:

```
# predict the entities
doc = nlp(text)
```

And then visualizing it:

```
# display the doc with jupyter mode
spacy.displacy.render(doc, style="ent", jupyter=True)
```

Albert EinsteinPERSON was a GermanNORP-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time. EinsteinPERSON is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanicsORG.EinsteinPERSON was born in the German EmpireGPE, but moved to SwitzerlandGPE in 1895DATE, forsaking his GermanNORP citizenship (as a subject of the Kingdom of WürttembergGPE) the following yearDATE. In 1897DATE, at the age of 17DATE, he enrolled in the mathematics and physics teaching diploma program at the SwissNORP Federal polytechnic school in ZürichGPE, graduating in 1900DATE

This one looks much better, and there are a lot more entities (18) than the previous ones, namely `CARDINAL`, `DATE`, `EVENT`, `FAC`, `GPE`, `LANGUAGE`, `LAW`, `LOC`, `MONEY`, `NORP`, `ORDINAL`, `ORG`, `I`

However, quantum mechanics was mistakenly labeled as an organization, so let's use the Transformer model that spaCy is offering:

```
# load the English transformer pipeline (roberta-base) using spaCy
nlp_trf = spacy.load('en_core_web_trf')
```

Let's perform inference and visualize the text:

```
# perform inference on the model
doc_trf = nlp_trf(text)
# display the doc with jupyter mode
spacy.displacy.render(doc_trf, style="ent", jupyter=True)
```

Albert EinsteinPERSON was a GermanNORP-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time. EinsteinPERSON is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.EinsteinPERSON was born in the German EmpireGPE, but moved to SwitzerlandGPE in 1895DATE, forsaking his GermanNORP citizenship (as a subject of the Kingdom of WürttembergGPE) the following yearDATE. In 1897DATE, at the age of 17DATE, he enrolled in the mathematics and physics teaching diploma program at the Swiss FederalORG polytechnic school in ZürichGPE, graduating in 1900DATE

This time Swiss Federal was labeled as an organization, even though it wasn't complete (it should be Swiss Federal polytechnic school), and quantum mechanics is no longer an organization.

The `en_core_web_trf` model performs much better than the previous ones. Check this table that shows each English model offered by spaCy with their size and metrics evaluation of each:

Model Name	Model Size	Precision	Recall	F-Sc
`en_core_web_sm`	13MB	0.85	0.84	0.8
`en_core_web_md`	43MB	0.85	0.84	0.8
`en_core_web_lg`	741MB	0.86	0.85	0.8
`en_core_web_trf`	**438MB**	**0.90**	**0.90**	**0.**

Conclusion

Make sure you try other types of texts and see for yourself if your text confirms the above table! You can check this page on spaCy to see the details of each model.

For other languages, spaCy strives to make these models available for every language globally. You can check this page to see the available models for each language.

SourceCode:

ner.py

```
# %%
# !pip install --upgrade transformers sentencepiece

# %%
# !pip install https://github.com/explosion/spacy-models/releases/download/en_core_web_trf-3.2.0/en_core_web_trf-3.2.0-py3-none-any.whl

# %%
# !python -m spacy download en_core_web_sm

# %%
import spacy
from transformers import *

# %%
# sample text from Wikipedia
text = """
Albert Einstein was a German-born theoretical physicist, widely acknowledged to be one of the greatest and most influential physicists of all time.
Einstein is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.
Einstein was born in the German Empire, but moved to Switzerland in 1895, forsaking his German citizenship (as a subject of the Kingdom of Württemberg) the following year.
In 1897, at the age of 17, he enrolled in the mathematics and physics teaching diploma program at the Swiss Federal polytechnic school in Zürich, graduating in 1900
"""
```

```python

# %%
# load BERT model fine-tuned for Named Entity Recognition (NER)
ner = pipeline("ner", model="dslim/bert-base-NER")

# %%
# perform inference on the transformer model
doc_ner = ner(text)
# print the output
doc_ner

# %%
def get_entities_html(text, ner_result, title=None):
  """Returns a visual version of NER with the help of SpaCy"""
  ents = []
  for ent in ner_result:
    e = {}
    # add the start and end positions of the entity
    e["start"] = ent["start"]
    e["end"] = ent["end"]
    # add the score if you want in the label
    # e["label"] = f"{ent["entity"]}-{ent['score']:.2f}"
    e["label"] = ent["entity"]
    if ents and -1 <= ent["start"] - ents[-1]["end"] <= 1 and ents[-1]["label"] == e["label"]:
      # if the current entity is shared with previous entity
      # simply extend the entity end position instead of adding a new one
      ents[-1]["end"] = e["end"]
      continue
    ents.append(e)
  # construct data required for displacy.render() method
  render_data = [
    {
      "text": text,
      "ents": ents,
      "title": title,
    }
  ]
  return spacy.displacy.render(render_data, style="ent", manual=True, jupyter=True)
```

```python

# %%
# get HTML representation of NER of our text
get_entities_html(text, doc_ner)

# %%
# load roberta-large model
ner2 = pipeline("ner", model="xlm-roberta-large-finetuned-conll03-english")

# %%
# perform inference on this model
doc_ner2 = ner2(text)

# %%
# get HTML representation of NER of our text
get_entities_html(text, doc_ner2)

# %%
# load yet another roberta-large model
ner3 = pipeline("ner", model="Jean-Baptiste/roberta-large-ner-english")

# %%
# perform inference on this model
doc_ner3 = ner3(text)

# %%
# get HTML representation of NER of our text
get_entities_html(text, doc_ner3)

# %%
# load the English CPU-optimized pipeline
nlp = spacy.load("en_core_web_sm")

# %%
# predict the entities
doc = nlp(text)

# %%
```

```python
# display the doc with jupyter mode
spacy.displacy.render(doc, style="ent", jupyter=True)

# %%
# load the English transformer pipeline (roberta-base) using spaCy
nlp_trf = spacy.load('en_core_web_trf')

# %%
# perform inference on the model
doc_trf = nlp_trf(text)
# display the doc with jupyter mode
spacy.displacy.render(doc_trf, style="ent", jupyter=True)
```

CHAPTER 2: Fake News Detection in Python

Exploring the fake news dataset, performing data analysis such as word clouds and ngrams, and fine-tuning BERT transformer to build a fake news detector in Python using transformers library.

Introduction

Fake news is the intentional broadcasting of false or misleading claims as news, where the statements are purposely deceitful.

Newspapers, tabloids, and magazines have been supplanted by digital news platforms, blogs, social media feeds, and a plethora of mobile news applications. News organizations benefitted from the increased use of social media and mobile platforms by providing subscribers with up-to-the-minute information.

Consumers now have instant access to the latest news. These digital media platforms have increased in prominence due to their easy connectedness to the rest of the world and allow users to discuss and share ideas and debate topics such as democracy, education, health, research, and history. Fake news items on digital platforms are getting more popular and are used for profit, such as political and financial gain.

How Big is this Problem?

Because the Internet, social media, and digital platforms are widely used, anybody may propagate inaccurate and biased information. It is almost impossible to prevent the spread of fake news. There is a tremendous surge in the distribution of false news, which is not restricted to one sector such as politics but includes sports, health, history, entertainment, and science and research.

The Solution

It is vital to recognize and differentiate between false and accurate news. One method is to have an expert decide, and fact checks every piece of information, but this takes time and needs expertise that cannot be shared. Secondly, we can use machine learning and artificial intelligence tools to automate the identification of fake news.

Online news information includes various unstructured format data (such as documents, videos, and audio), but we will concentrate on text format news here. With the progress of machine learning and Natural language processing, we can now recognize the misleading and false character of an article or statement.

Several studies and experiments are being conducted to detect fake news across all mediums.

Our main goal for this tutorial is:

- Explore and analyze the Fake News dataset.
- Build a classifier that can distinguish Fake news with as much accuracy as possible.

Here is the table of content:

Data Exploration

In this work, we utilized the fake news dataset from Kaggle to classify untrustworthy news articles as fake news. We have a complete training dataset containing the following characteristics:

- `id`: unique id for a news article
- `title`: title of a news article
- `author`: author of the news article
- `text`: text of the article; could be incomplete
- `label`: a label that marks the article as potentially unreliable denoted by 1 (unreliable or fake) or 0 (reliable).

It is a binary classification problem in which we must predict if a particular news story is reliable or not.

If you have a Kaggle account, you can simply download the dataset from the website there and extract the ZIP file.

I also uploaded the dataset into Google Drive, and you can get it here or use the `gdown` library to download it in Google Colab or Jupyter notebooks automatically:

```
$ pip install gdown
```

```
# download from Google Drive
$ gdown "https://drive.google.com/uc?id=178f_VkNxccNidap-5-
```

```
uffXUW475pAuPy&confirm=t"
```

```
Downloading...
From: https://drive.google.com/uc?id=178f_VkNxccNidap-5-
uffXUW475pAuPy&confirm=t
To: /content/fake-news.zip
100% 48.7M/48.7M [00:00<00:00, 74.6MB/s]
```

Unzipping the files:

```
$ unzip fake-news.zip
```

Three files will appear in the current working directory: `train.csv`, `test.csv`, and `submit.csv`, we will be using `train.csv` in most of the tutorial.

Installing the required dependencies:

```
$ pip install transformers nltk pandas numpy matplotlib seaborn wordcloud
```

***Note:** If you're in a local environment, make sure you install PyTorch for GPU, head to this page for a proper installation.*

Let's import the essential libraries for analysis:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

The NLTK corpora and modules must be installed using the standard NLTK downloader:

```
import nltk
nltk.download('stopwords')
nltk.download('wordnet')
```

The fake news dataset comprises various authors' original and fictitious article titles and text. Let's import our dataset:

```
# load the dataset
news_d = pd.read_csv("train.csv")
```

```
print("Shape of News data:", news_d.shape)
print("News data columns", news_d.columns)
```

Output:

```
Shape of News data: (20800, 5)
News data columns Index(['id', 'title', 'author', 'text', 'label'], dtype='object')
```

Here's how the dataset looks:

```
# by using df.head(), we can immediately familiarize ourselves with the dataset.
news_d.head()
```

Output:

```
id	title	author	text	label
0	0	House Dem Aide: We Didn’t Even See Comey’s Let...	Darrell Lucus	House Dem Aide: We Didn’t Even See Comey’s Let...	1
1	1	FLYNN: Hillary Clinton, Big Woman on Campus - ...	Daniel J. Flynn	Ever get the feeling your life circles the rou...	0
2	2	Why the Truth Might Get You Fired	Consortiumnews.com	Why the Truth Might Get You Fired October 29, ...	1
3	3	15 Civilians Killed In Single US Airstrike Hav...	Jessica Purkiss	Videos 15 Civilians Killed In Single US Airstr...	1
4	4	Iranian woman jailed for fictional unpublished...	Howard Portnoy	Print \nAn Iranian woman has been sentenced to...	1
```

We have 20,800 rows, which have five columns. Let's see some statistics of the `text` column:

```
#Text Word startistics: min.mean, max and interquartile range

txt_length = news_d.text.str.split().str.len()
txt_length.describe()
```

Output:

```
count    20761.000000
mean       760.308126
std        869.525988
min          0.000000
25%        269.000000
```

```
50%       556.000000
75%      1052.000000
max     24234.000000
Name: text, dtype: float64
```

Stats for the `title` column:

```
#Title statistics

title_length = news_d.title.str.split().str.len()
title_length.describe()
```

Output:

```
count    20242.000000
mean        12.420709
std          4.098735
min          1.000000
25%         10.000000
50%         13.000000
75%         15.000000
max         72.000000
Name: title, dtype: float64
```

The statistics for the training and testing sets are as follows:

- The `text` attribute has a higher word count with an average of 760 words and 75% having more than 1000 words.
- The `title` attribute is a short statement with an average of 12 words, and

75% of them are around 15 words.

Our experiment would be with both text and title together.

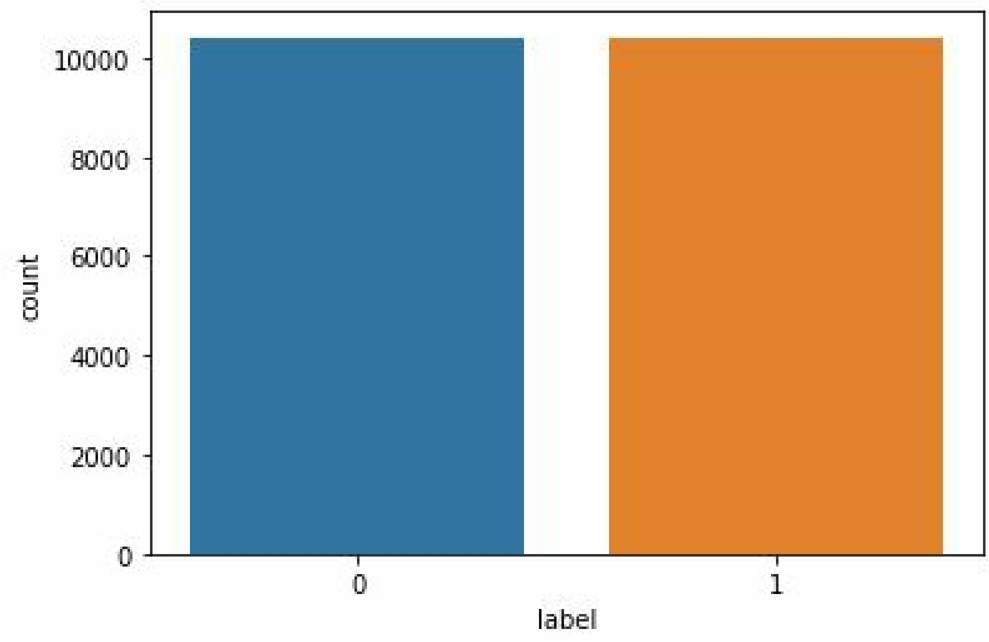
Distribution of Classes

Counting plots for both labels:

```
sns.countplot(x="label", data=news_d);
print("1: Unreliable")
print("0: Reliable")
print("Distribution of labels:")
print(news_d.label.value_counts());
```

Output:

```
1: Unreliable
0: Reliable
Distribution of labels:
1    10413
0    10387
Name: label, dtype: int64
```




```
print(round(news_d.label.value_counts(normalize=True),2)*100);
```

Output:

```
1    50.0
0    50.0
Name: label, dtype: float64
```

The number of untrustworthy articles (fake or 1) is 10413, while the number of trustworthy articles (reliable or 0) is 10387. Almost 50% of the articles are fake. Therefore, the accuracy metric will measure how well our model is doing when building a classifier.

Data Cleaning for Analysis

In this section, we will clean our dataset to do some analysis:

- Drop unused rows and columns.

- Perform null value imputation.
- Remove special characters.
- Remove stop words.

```
# Constants that are used to sanitize the datasets

column_n = ['id', 'title', 'author', 'text', 'label']
remove_c = ['id','author']
categorical_features = []
target_col = ['label']
text_f = ['title', 'text']


# Clean Datasets
import nltk
from nltk.corpus import stopwords
import re
from nltk.stem.porter import PorterStemmer
from collections import Counter

ps = PorterStemmer()
wnl = nltk.stem.WordNetLemmatizer()

stop_words = stopwords.words('english')
stopwords_dict = Counter(stop_words)

# Removed unused clumns
def remove_unused_c(df,column_n=remove_c):
    df = df.drop(column_n,axis=1)
    return df

```

```python
# Impute null values with None
def null_process(feature_df):
    for col in text_f:
        feature_df.loc[feature_df[col].isnull(), col] = "None"
    return feature_df

def clean_dataset(df):
    # remove unused column
    df = remove_unused_c(df)
    #impute null values
    df = null_process(df)
    return df

# Cleaning text from unused characters
def clean_text(text):
    text = str(text).replace(r'http[\w:/\.]+', ' ')  # removing urls
    text = str(text).replace(r'[^\.\w\s]', ' ')  # remove everything but characters and punctuation
    text = str(text).replace('[^a-zA-Z]', ' ')
    text = str(text).replace(r'\s\s+', ' ')
    text = text.lower().strip()
    #text = ' '.join(text)
    return text

## Nltk Preprocessing include:
# Stop words, Stemming and Lemmetization
# For our project we use only Stop word removal
def nltk_preprocess(text):
    text = clean_text(text)
    wordlist = re.sub(r'[^\w\s]', '', text).split()
```

```
    #text = ' '.join([word for word in wordlist if word not in stopwords_dict])
    #text = [ps.stem(word) for word in wordlist if not word in stopwords_dict]
    text = ' '.join([wnl.lemmatize(word) for word in wordlist if word not in stopwords_dict])
    return  text
```

In the code block above:

- We have imported NLTK, which is a famous platform for developing Python applications that interact with human language. Next, we import `re` for regex.
- We import stopwords from `nltk.corpus`. When working with words, particularly when considering semantics, we sometimes need to eliminate common words that do not add any significant meaning to a statement, such as `"but"`, `"can"`, `"we"`, etc.
- `PorterStemmer` is used to perform stemming words with NLTK. Stemmers strip words of their morphological affixes, leaving the word stem solely.
- We import `WordNetLemmatizer()` from NLTK library for lemmatization. Lemmatization is much more effective than stemming. It goes beyond word reduction and evaluates a language's whole lexicon to apply morphological analysis to words, with the goal of just removing inflectional ends and returning the base or dictionary form of a word, known as the lemma.
- `stopwords.words('english')` allow us to look at the list of all the English stop words supported by NLTK.
- `remove_unused_c()` function is used to remove the unused columns.
- We impute null values with `None` using the `null_process()` function.
- Inside the function `clean_dataset()`, we call `remove_unused_c()` and `null_process()` functions. This function is responsible for data cleaning.
- To clean text from unused characters, we have created the `clean_text()` function.

- For preprocessing, we will use only stop word removal. We created the `nltk_preprocess()` function for that purpose.

Preprocessing the `text` and `title`:

```
# Perform data cleaning on train and test dataset by calling clean_dataset function
df = clean_dataset(news_d)
# apply preprocessing on text through apply method by calling the function nltk_preprocess
df["text"] = df.text.apply(nltk_preprocess)
# apply preprocessing on title through apply method by calling the function nltk_preprocess
df["title"] = df.title.apply(nltk_preprocess)


# Dataset after cleaning and preprocessing step
df.head()
```

Output:

```
title	text	label
0	house dem aide didnt even see comeys letter ja...	house dem aide didnt even see comeys letter ja...	1
1	flynn hillary clinton big woman campus breitbart	ever get feeling life circle roundabout rather...	0
2	truth might get fired	truth might get fired october 29 2016 tension ...	1
3	15 civilian killed single u airstrike identified	video 15 civilian killed single u airstrike id...	1
4	iranian woman jailed fictional unpublished sto...	print iranian woman sentenced six year prison ...	1
```

Explorative Data Analysis

In this section, we will perform:

- **Univariate Analysis**: It is a statistical analysis of the text. We will use word cloud for that purpose. A word cloud is a visualization approach for text data where the most common term is presented in the most considerable font size.
- **Bivariate Analysis**: Bigram and Trigram will be used here. According to Wikipedia: "*an n-gram is a contiguous sequence of n items from a given sample of text or speech. According to the application, the items can be phonemes, syllables, letters, words, or base pairs. The n-grams are typically collected from a text or speech corpus*".

Single-word Cloud

The most frequent words appear in a bold and bigger font in a word cloud. This section will perform a word cloud for all words in the dataset.

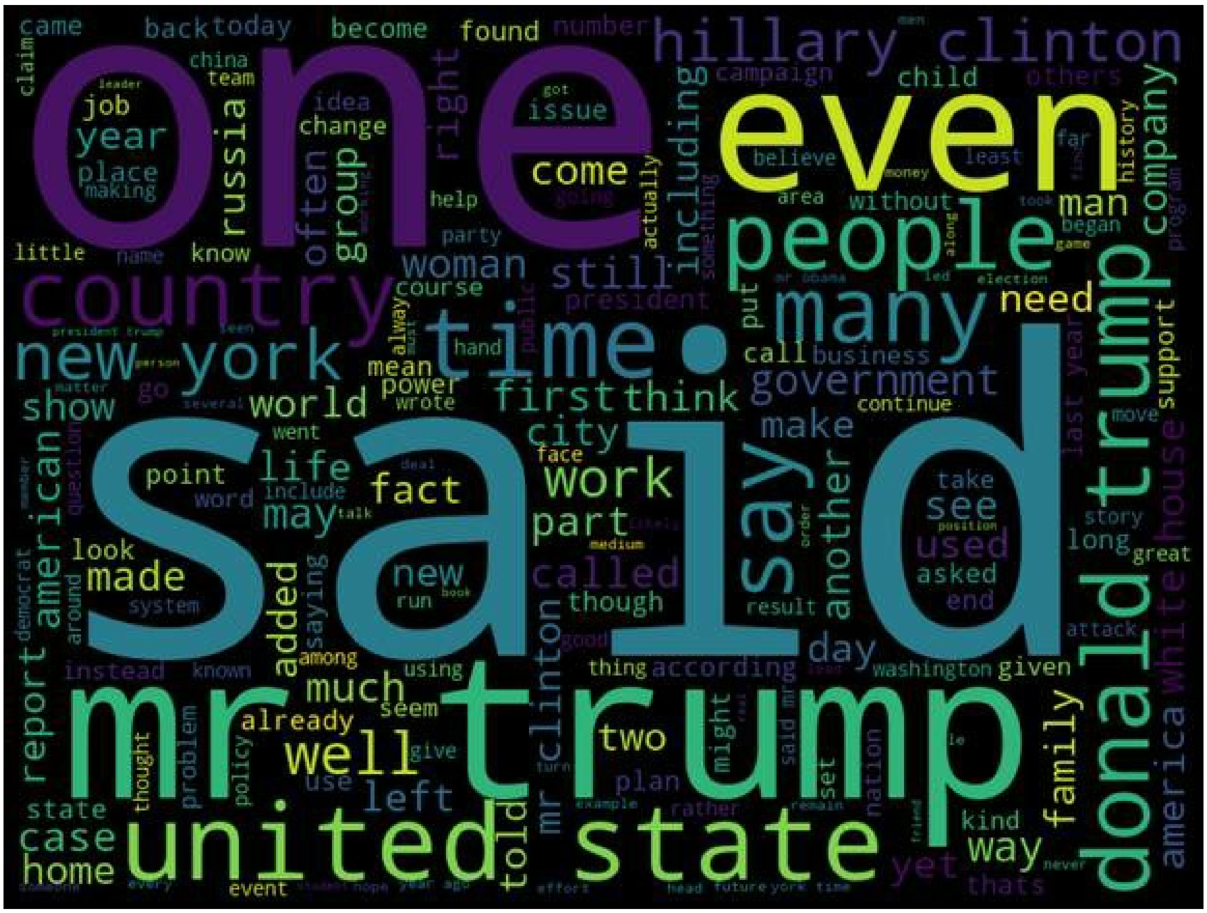
The WordCloud library's `wordcloud()` function will be used, and the `generate()` is utilized for generating the word cloud image:

```
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt

# initialize the word cloud
wordcloud = WordCloud( background_color='black', width=800,
height=600)
# generate the word cloud by passing the corpus
text_cloud = wordcloud.generate(' '.join(df['text']))
# plotting the word cloud
plt.figure(figsize=(20,30))
plt.imshow(text_cloud)
```

```
plt.axis('off')
plt.show()
```

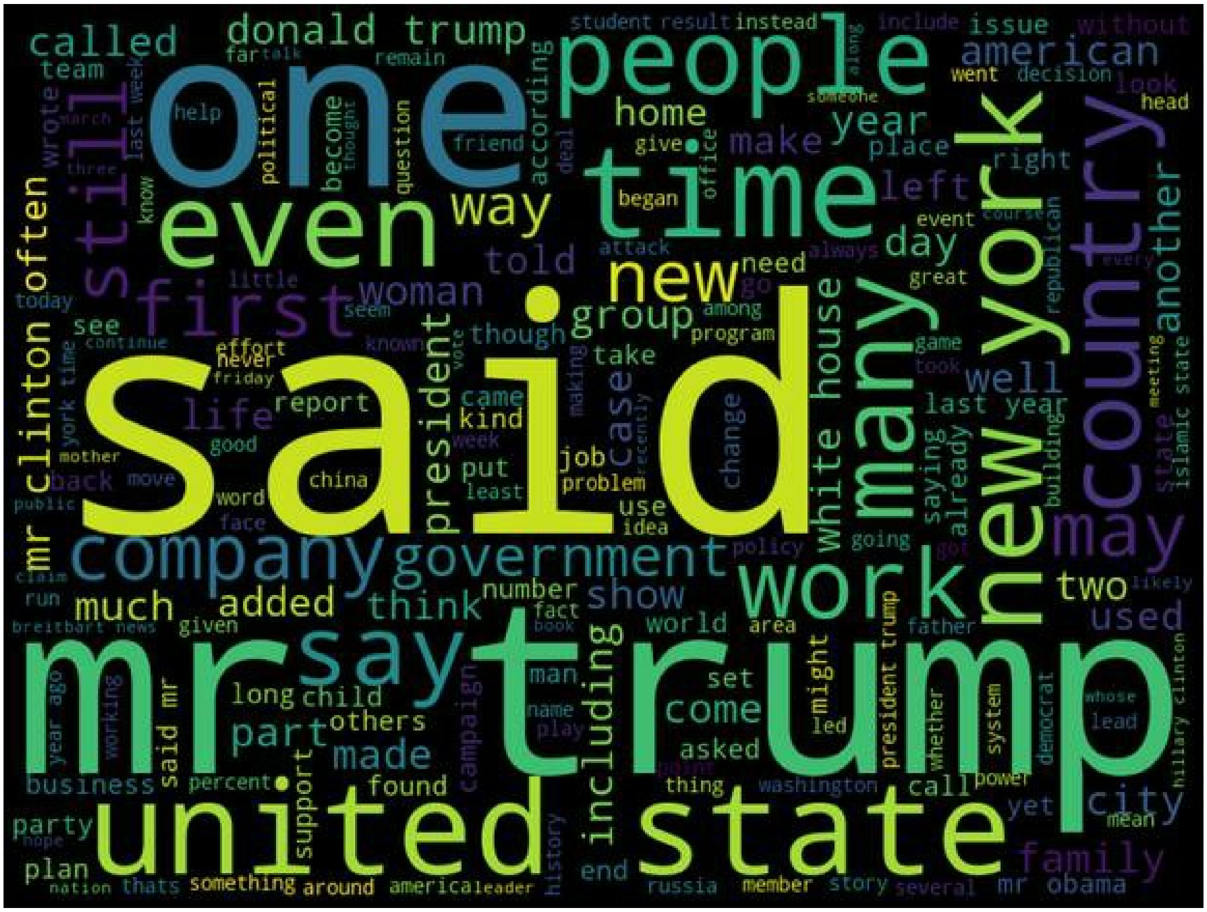
Output:



Word cloud for reliable news only:

```
true_n = ' '.join(df[df['label']==0]['text'])
wc = wordcloud.generate(true_n)
plt.figure(figsize=(20,30))
plt.imshow(wc)
plt.axis('off')
plt.show()
```

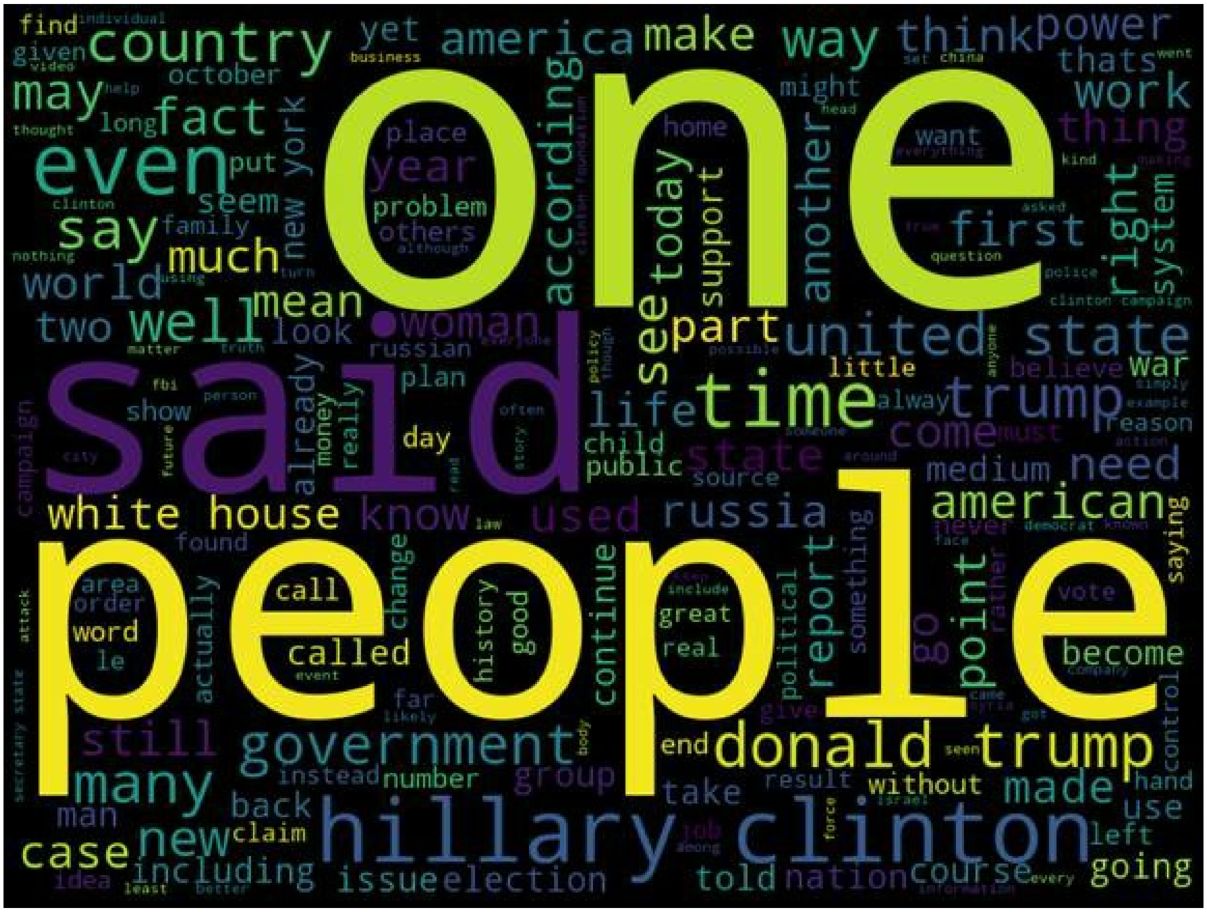
Output:



Word cloud for fake news only:

```
fake_n = ' '.join(df[df['label']==1]['text'])
wc= wordcloud.generate(fake_n)
plt.figure(figsize=(20,30))
plt.imshow(wc)
plt.axis('off')
plt.show()
```

Output:



Most Frequent Bigram (Two-word Combination)

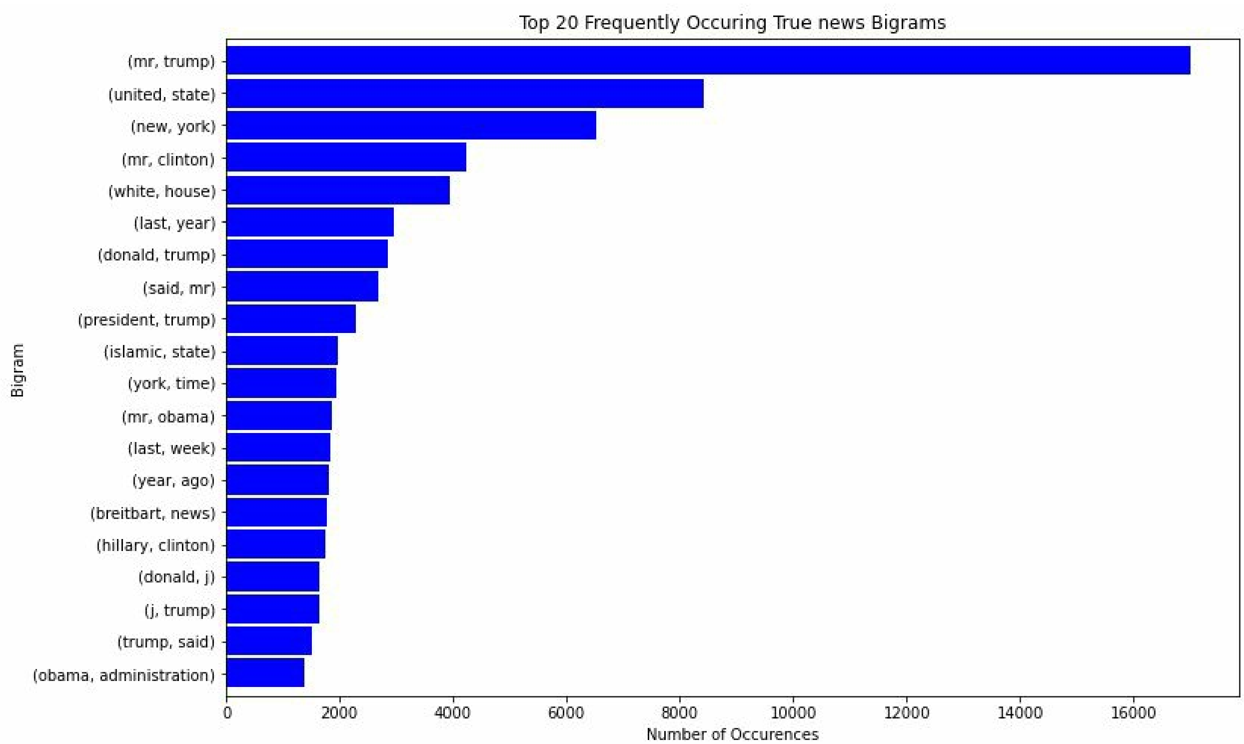
An N-gram is a sequence of letters or words. A character unigram is made up of a single character, while a bigram comprises a series of two characters. Similarly, word N-grams are made up of a series of n words. The word "united" is a 1-gram (unigram). The combination of the words "united state" is a 2-gram (bigram), "new york city" is a 3-gram.

Let's plot the most common bigram on the reliable news:

```
def plot_top_ngrams(corpus, title, ylabel, xlabel="Number of Occurences", n=2):
  """Utility function to plot top n-grams"""
  true_b = (pd.Series(nltk.ngrams(corpus.split(), n)).value_counts())[:20]
```

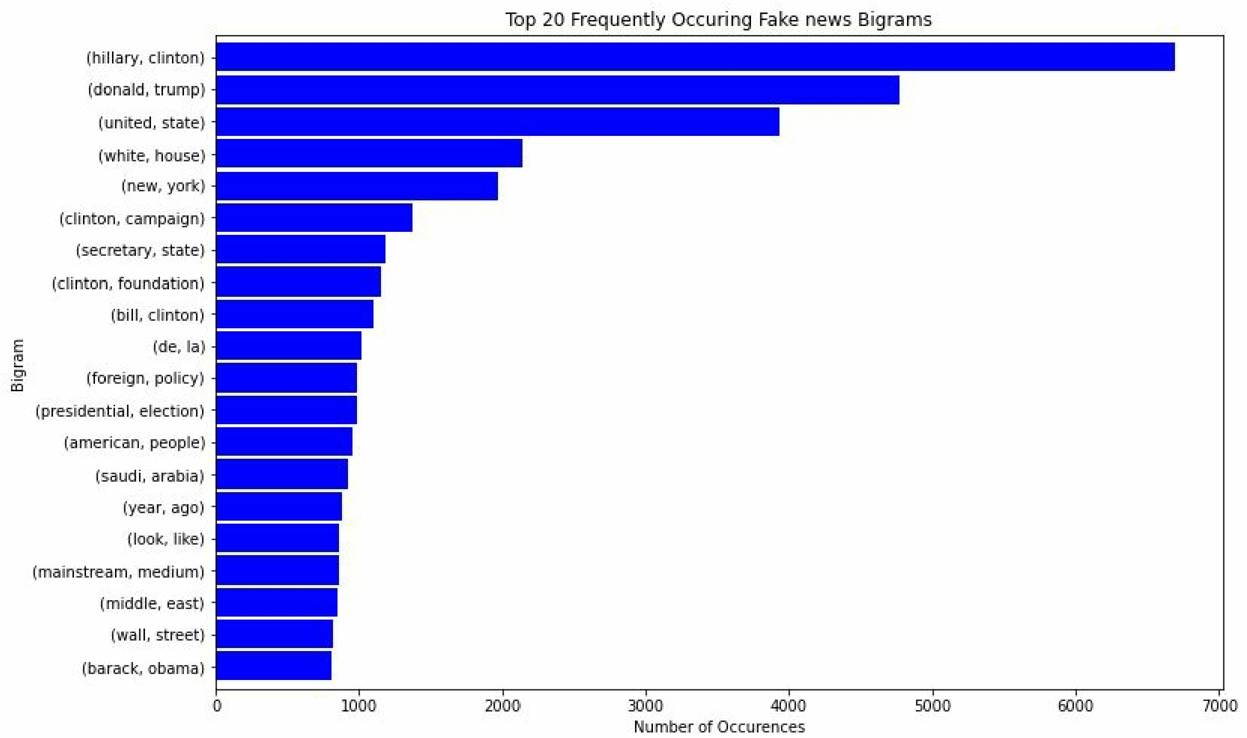
```
true_b.sort_values().plot.barh(color='blue', width=.9, figsize=(12, 8))
plt.title(title)
plt.ylabel(ylabel)
plt.xlabel(xlabel)
plt.show()
```

```
plot_top_ngrams(true_n, 'Top 20 Frequently Occuring True news Bigrams', "Bigram", n=2)
```



The most common bigram on the fake news:

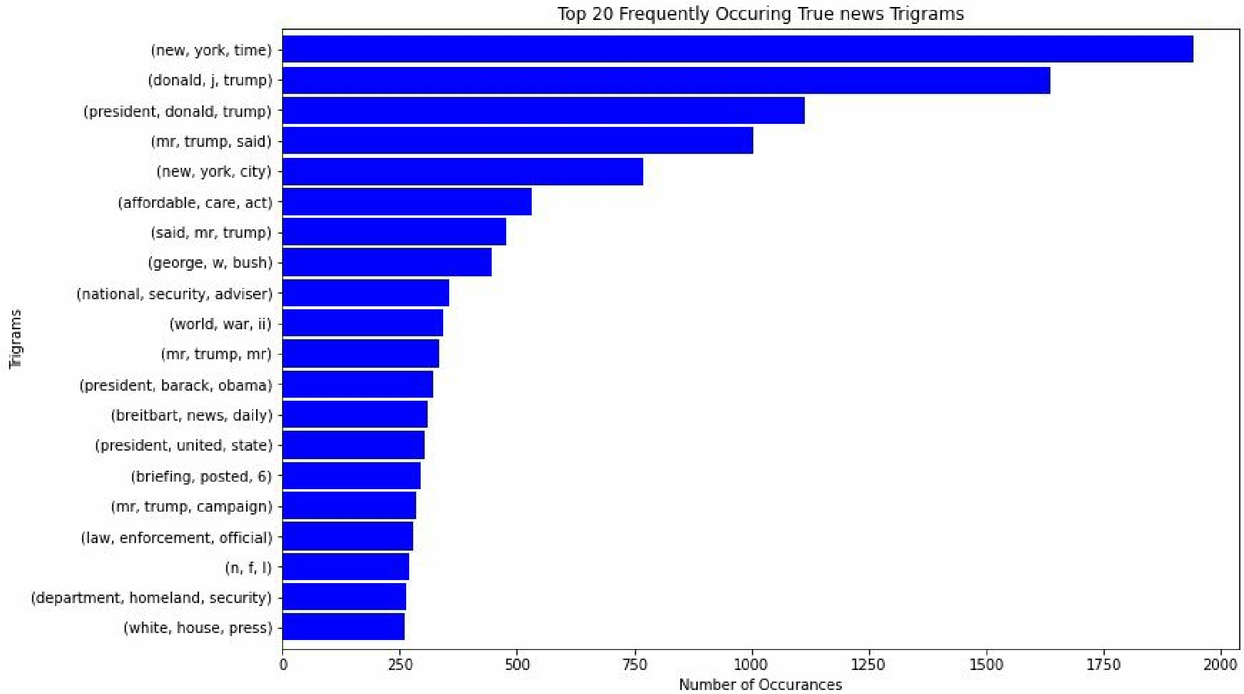
```
plot_top_ngrams(fake_n, 'Top 20 Frequently Occuring Fake news Bigrams', "Bigram", n=2)
```



Most Frequent Trigram (Three-word combination)

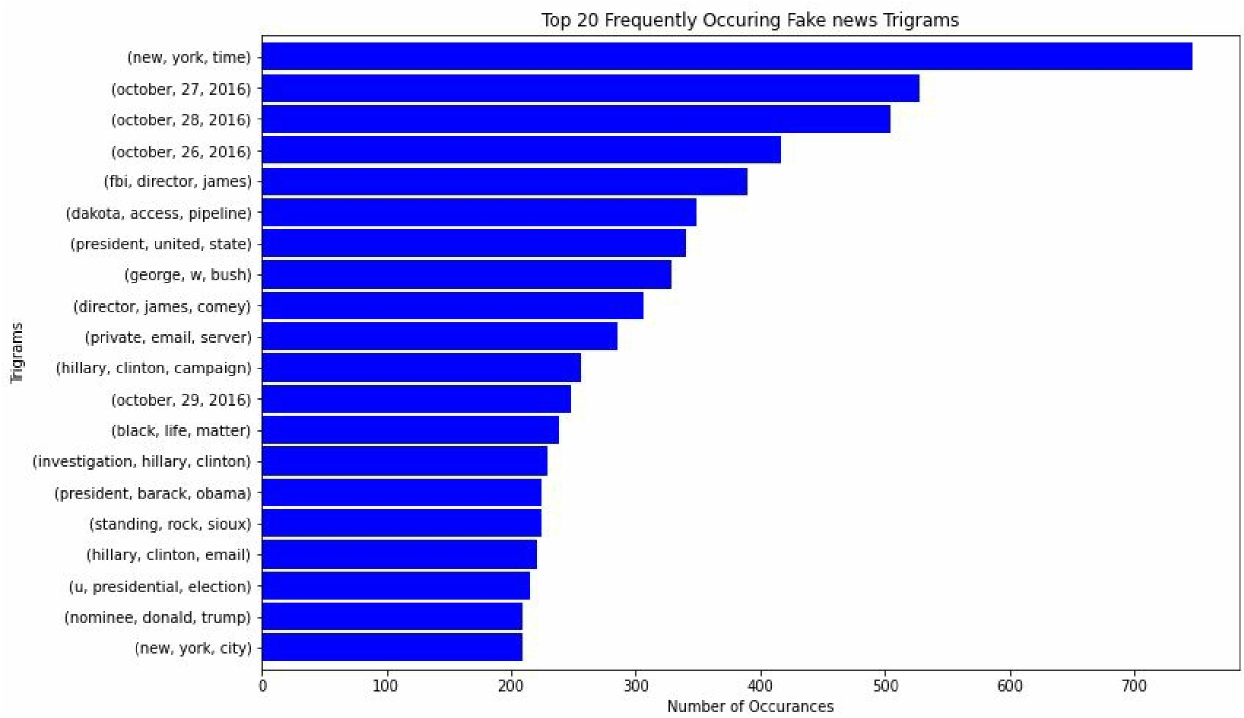
The most common trigram on reliable news:

```
plot_top_ngrams(true_n, 'Top 20 Frequently Occuring True news Trigrams', "Trigrams", n=3)
```



For fake news now:

```
plot_top_ngrams(fake_n, 'Top 20 Frequently Occuring Fake news Trigrams', "Trigrams", n=3)
```



The above plots give us some ideas on how both classes look. In the next section, we'll use the transformers library to build a fake news detector.

Building a Classifier by Fine-tuning BERT

This section will be grabbing code extensively from the fine-tuning BERT tutorial to make a fake news classifier using the transformers library. So, for more detailed information, you can head to the original tutorial.

If you didn't install transformers, you have to:

```
$ pip install transformers
```

Let's import the necessary libraries:

```
import torch
```

```
from transformers.file_utils import is_tf_available, is_torch_available,
is_torch_tpu_available
```

```
from transformers import BertTokenizerFast, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
import numpy as np
from sklearn.model_selection import train_test_split

import random
```

We want to make our results reproducible even if we restart our environment:

```
def set_seed(seed: int):
    """
    Helper function for reproducible behavior to set the seed in ``random``,
``numpy``, ``torch`` and/or ``tf`` (if
    installed).

    Args:
        seed (:obj:`int`): The seed to set.
    """
    random.seed(seed)
    np.random.seed(seed)
    if is_torch_available():
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        # ^^ safe to call this function even if cuda is not available
    if is_tf_available():
        import tensorflow as tf

        tf.random.set_seed(seed)

```

```
set_seed(1)
```

The model we're going to use is the `bert-base-uncased`:

```
# the model we gonna train, base uncased BERT
# check text classification models here: https://huggingface.co/models?filter=text-classification
model_name = "bert-base-uncased"
# max sequence length for each document/sentence sample
max_length = 512
```

Loading the tokenizer:

```
# load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained(model_name, do_lower_case=True)
```

Data Preparation

Let's now clean `NaN` values from `text`, `author`, and `title` columns:

```
news_df = news_d[news_d['text'].notna()]
news_df = news_df[news_df["author"].notna()]
news_df = news_df[news_df["title"].notna()]
```

Next, making a function that takes the dataset as a Pandas dataframe and returns the train/validation splits of texts and labels as lists:

```
def prepare_data(df, test_size=0.2, include_title=True, include_author=True):
```

```
  texts = []
  labels = []
  for i in range(len(df)):
    text = df["text"].iloc[i]
    label = df["label"].iloc[i]
    if include_title:
      text = df["title"].iloc[i] + " - " + text
    if include_author:
      text = df["author"].iloc[i] + " : " + text
    if text and label in [0, 1]:
      texts.append(text)
      labels.append(label)
  return train_test_split(texts, labels, test_size=test_size)

train_texts, valid_texts, train_labels, valid_labels = prepare_data(news_df)
```

The above function takes the dataset in a dataframe type and returns them as lists split into training and validation sets. Setting `include_title` to `True` means that we add the `title` column to the `text` we going to use for training, setting `include_author` to `True` means we add the `author` to the text as well.

Let's make sure the labels and texts have the same length:

```
print(len(train_texts), len(train_labels))
print(len(valid_texts), len(valid_labels))
```

Output:

```
14628 14628
3657 3657
```

Tokenizing the Dataset

Let's use the BERT tokenizer to tokenize our dataset:

```
# tokenize the dataset, truncate when passed `max_length`,
# and pad with 0's when less than `max_length`
train_encodings = tokenizer(train_texts, truncation=True, padding=True,
max_length=max_length)
valid_encodings = tokenizer(valid_texts, truncation=True, padding=True,
max_length=max_length)
```

Converting the encodings into a PyTorch dataset:

```
class NewsGroupsDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {k: torch.tensor(v[idx]) for k, v in self.encodings.items()}
        item["labels"] = torch.tensor([self.labels[idx]])
        return item

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

# convert our tokenized data into a torch Dataset
train_dataset = NewsGroupsDataset(train_encodings, train_labels)
valid_dataset = NewsGroupsDataset(valid_encodings, valid_labels)
```

Loading and Fine-tuning the Model

We'll be using `BertForSequenceClassification` to load our BERT transformer model:

```
# load the model
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)
```

We set `num_labels` to 2 since it's a binary classification. Below function is a callback to calculate the accuracy on each validation step:

```
from sklearn.metrics import accuracy_score

def compute_metrics(pred):
  labels = pred.label_ids
  preds = pred.predictions.argmax(-1)
  # calculate accuracy using sklearn's function
  acc = accuracy_score(labels, preds)
  return {
      'accuracy': acc,
  }
```

Let's initialize the training parameters:

```
training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=1,              # total number of training epochs
    per_device_train_batch_size=10,  # batch size per device during training
    per_device_eval_batch_size=20,   # batch size for evaluation
```

```
    warmup_steps=100,                # number of warmup steps for learning rate scheduler
    logging_dir='./logs',            # directory for storing logs
    load_best_model_at_end=True,     # load the best model when finished training (default metric is loss)
    # but you can specify `metric_for_best_model` argument to change to accuracy or other metric
    logging_steps=200,               # log & save weights each logging_steps
    save_steps=200,
    evaluation_strategy="steps",     # evaluate each `logging_steps`
)
```

I've set the `per_device_train_batch_size` to 10, but you should set it as high as your GPU could possibly fit. Setting the `logging_steps` and `save_steps` to 200, meaning we're going to perform evaluation and save the model weights on each 200 training step.

You can check this page for more detailed information about the available training parameters.

Let's instantiate the trainer:

```
trainer = Trainer(
    model=model,                         # the instantiated Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=valid_dataset,          # evaluation dataset
    compute_metrics=compute_metrics,     # the callback that computes metrics of interest
)
```

Training the model:

```
# train the model
trainer.train()
```

The training takes a few hours to finish, depending on your GPU. If you're on the free version of Colab, it should take an hour with NVIDIA Tesla K80. Here is the output:

```
***** Running training *****
  Num examples = 14628
  Num Epochs = 1
  Instantaneous batch size per device = 10
  Total train batch size (w. parallel, distributed & accumulation) = 10
  Gradient Accumulation steps = 1
  Total optimization steps = 1463
 [1463/1463 41:07, Epoch 1/1]
Step	Training Loss	Validation Loss	Accuracy
200	0.250800	0.100533	0.983867
400	0.027600	0.043009	0.993437
600	0.023400	0.017812	0.997539
800	0.014900	0.030269	0.994258
1000	0.022400	0.012961	0.998086
1200	0.009800	0.010561	0.998633
1400	0.007700	0.010300	0.998633
***** Running Evaluation *****
  Num examples = 3657
  Batch size = 20
Saving model checkpoint to ./results/checkpoint-200
Configuration saved in ./results/checkpoint-200/config.json
```

```
Model weights saved in ./results/checkpoint-200/pytorch_model.bin
<SNIPPED>
***** Running Evaluation *****
  Num examples = 3657
  Batch size = 20
Saving model checkpoint to ./results/checkpoint-1400
Configuration saved in ./results/checkpoint-1400/config.json
Model weights saved in ./results/checkpoint-1400/pytorch_model.bin


Training completed. Do not forget to share your model on
huggingface.co/models =)


Loading best model from ./results/checkpoint-1400 (score:
0.010299865156412125).
TrainOutput(global_step=1463, training_loss=0.04888018785440506,
metrics={'train_runtime': 2469.1722, 'train_samples_per_second': 5.924,
'train_steps_per_second': 0.593, 'total_flos': 3848788517806080.0,
'train_loss': 0.04888018785440506, 'epoch': 1.0})
```

Model Evaluation

Since `load_best_model_at_end` is set to `True`, the best weights will be loaded when the training is completed. Let's evaluate it with our validation set:

```
# evaluate the current model after training
trainer.evaluate()
```

Output:

```
***** Running Evaluation *****
  Num examples = 3657
```

```
 Batch size = 20
[183/183 02:11]
{'epoch': 1.0,
 'eval_accuracy': 0.998632759092152,
 'eval_loss': 0.010299865156412125,
 'eval_runtime': 132.0374,
 'eval_samples_per_second': 27.697,
 'eval_steps_per_second': 1.386}
```

Saving the model and the tokenizer:

```
# saving the fine tuned model & tokenizer
model_path = "fake-news-bert-base-uncased"
model.save_pretrained(model_path)
tokenizer.save_pretrained(model_path)
```

A new folder containing the model configuration and weights will appear after running the above cell. If you want to perform prediction, you simply use the `from_pretrained()` method we used when we loaded the model, and you're good to go.

Next, let's make a function that accepts the article text as an argument and return whether it's fake or not:

```
def get_prediction(text, convert_to_label=False):
    # prepare our text into tokenized sequence
    inputs = tokenizer(text, padding=True, truncation=True,
max_length=max_length, return_tensors="pt").to("cuda")
    # perform inference to our model
    outputs = model(**inputs)
```

```
    # get output probabilities by doing softmax
    probs = outputs[0].softmax(1)
    # executing argmax function to get the candidate label
    d = {
        0: "reliable",
        1: "fake"
    }
    if convert_to_label:
      return d[int(probs.argmax())]
    else:
      return int(probs.argmax())
```

I've taken an example from `test.csv` that the model never saw to perform inference, I've checked it, and it's an actual article from The New York Times:

```
real_news = """
Tim Tebow Will Attempt Another Comeback, This Time in Baseball - The New York Times",Daniel Victor,"If at first you don’t succeed, try a different sport. Tim Tebow, who was a Heisman   quarterback at the University of Florida but was unable to hold an N. F. L. job, is pursuing a career in Major League Baseball. <SNIPPED>
"""
```

The original text is in [the Colab environment] if you want to it, as it's a complete article. Let's pass it to the model and see the results:

```
get_prediction(real_news, convert_to_label=True)
```

Output:

```
reliable
```

Appendix: Creating a Submission File for Kaggle

In this section, we will predict all the articles in the `test.csv` to create a submission file to see our accuracy in the test set on the Kaggle competition:

```
# read the test set
test_df = pd.read_csv("test.csv")
# make a  of the testing set
new_df = test_df.()
# add a new column that contains the author, title and article content
new_df["new_text"] = new_df["author"].astype(str) + " : " + new_df["title"].astype(str) + " - " + new_df["text"].astype(str)
# get the prediction of all the test set
new_df["label"] = new_df["new_text"].apply(get_prediction)
# make the submission file
final_df = new_df[["id", "label"]]
final_df.to_csv("submit_final.csv", index=False)
```

After we concatenate the author, title, and article text together, we pass the `get_prediction()` function to the new column to fill the `label` column, we then use `to_csv()` method to create the submission file for Kaggle. Here's my submission score:

We got 99.78% and 100% accuracy on private and public leaderboards. That's awesome!

Conclusion

Alright, we're done with the tutorial. You can check this page to see various training parameters you can tweak.

If you have a custom fake news dataset for fine-tuning, you simply have to pass a list of samples to the tokenizer as we did, you won't change any other code after that.

SourceCode:

fakenews_detection.py

```
# -*- coding: utf-8 -*-
"""fakenews_seq_classification.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1e_3Zn4mPSYaMvRvLeOtA8AYXqOSbgkgc
"""

!pip install -q kaggle

from google.colab import files

files.upload()

!rm -rf ~/.kaggle
!mkdir ~/.kaggle
!cp kaggle.json ~/.kaggle/

!chmod 600 ~/.kaggle/kaggle.json
```

```
!kaggle competitions download -c fake-news

!unzip test.csv.zip
!unzip train.csv.zip

!pip install gdown

# download from Google Drive
!gdown "https://drive.google.com/uc?id=178f_VkNxccNidap-5-uffXUW475pAuPy&confirm=t"

!unzip fake-news.zip

### Import all library
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import nltk
nltk.download('stopwords')
nltk.download('wordnet')

# load the dataset
news_d = pd.read_csv("train.csv")
submit_test = pd.read_csv("test.csv")

## Shape and colums of train dataset
print(" Shape of News data :: ", news_d.shape)
print(" News data columns", news_d.columns)

## by using df.head(),We can immediately familiarize ourselves with the dataset.
news_d.head()

#Text Word startistics: min.mean, max and interquartile range

txt_length = news_d.text.str.split().str.len()
txt_length.describe()
```

```
#Title statistics

title_length = news_d.title.str.split().str.len()
title_length.describe()

sns.countplot(x="label", data=news_d);
print("1: Unreliable")
print("0: Reliable")
print("Distribution of labels:")
print(news_d.label.value_counts());

print(round(news_d.label.value_counts(normalize=True),2)*100);

# Constants that are used to sanitize the datasets
column_n = ['id', 'title', 'author', 'text', 'label']
remove_c = ['id','author']
categorical_features = []
target_col = ['label']
text_f = ['title', 'text']

# Clean Datasets
import nltk
from nltk.corpus import stopwords
import re
from nltk.stem.porter import PorterStemmer
from collections import Counter

ps = PorterStemmer()
wnl = nltk.stem.WordNetLemmatizer()

stop_words = stopwords.words('english')
stopwords_dict = Counter(stop_words)

# Removed unused clumns
def remove_unused_c(df,column_n=remove_c):
    df = df.drop(column_n,axis=1)
    return df
```

```
# Impute null values with None
def null_process(feature_df):
    for col in text_f:
        feature_df.loc[feature_df[col].isnull(), col] = "None"
    return feature_df

def clean_dataset(df):
    # remove unused column
    df = remove_unused_c(df)
    #impute null values
    df = null_process(df)
    return df

# Cleaning text from unused characters
def clean_text(text):
    text = str(text).replace(r'http[\w:/\.]+', ' ')  # removing urls
    text = str(text).replace(r'[^\.\w\s]', ' ')  # remove everything but characters and punctuation
    text = str(text).replace('[^a-zA-Z]', ' ')
    text = str(text).replace(r'\s\s+', ' ')
    text = text.lower().strip()
    #text = ' '.join(text)
    return text

## Nltk Preprocessing include:
# Stop words, Stemming and Lemmetization
# For our project we use only Stop word removal
def nltk_preprocess(text):
    text = clean_text(text)
    wordlist = re.sub(r'[^\w\s]', '', text).split()
    #text = ' '.join([word for word in wordlist if word not in stopwords_dict])
    #text = [ps.stem(word) for word in wordlist if not word in stopwords_dict]
    text = ' '.join([wnl.lemmatize(word) for word in wordlist if word not in stopwords_dict])
    return  text

# Perform data cleaning on train and test dataset by calling clean_dataset function
df = clean_dataset(news_d)
# apply preprocessing on text through apply method by calling the function nltk_preprocess
```

```
df["text"] = df.text.apply(nltk_preprocess)
# apply preprocessing on title through apply method by calling the function nltk_preprocess
df["title"] = df.title.apply(nltk_preprocess)

# Dataset after cleaning and preprocessing step
df.head()

from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt

# initialize the word cloud
wordcloud = WordCloud( background_color='black', width=800, height=600)
# generate the word cloud by passing the corpus
text_cloud = wordcloud.generate(' '.join(df['text']))
# plotting the word cloud
plt.figure(figsize=(20,30))
plt.imshow(text_cloud)
plt.axis('off')
plt.show()

true_n = ' '.join(df[df['label']==0]['text'])
wc = wordcloud.generate(true_n)
plt.figure(figsize=(20,30))
plt.imshow(wc)
plt.axis('off')
plt.show()

fake_n = ' '.join(df[df['label']==1]['text'])
wc= wordcloud.generate(fake_n)
plt.figure(figsize=(20,30))
plt.imshow(wc)
plt.axis('off')
plt.show()

def plot_top_ngrams(corpus, title, ylabel, xlabel="Number of Occurences", n=2):
  """Utility function to plot top n-grams"""
  true_b = (pd.Series(nltk.ngrams(corpus.split(), n)).value_counts())[:20]
  true_b.sort_values().plot.barh(color='blue', width=.9, figsize=(12, 8))
```

```
 plt.title(title)
 plt.ylabel(ylabel)
 plt.xlabel(xlabel)
 plt.show()

plot_top_ngrams(true_n, 'Top 20 Frequently Occuring True news Bigrams', "Bigram", n=2)

plot_top_ngrams(fake_n, 'Top 20 Frequently Occuring Fake news Bigrams', "Bigram", n=2)

plot_top_ngrams(true_n, 'Top 20 Frequently Occuring True news Trigrams', "Trigrams", n=3)

plot_top_ngrams(fake_n, 'Top 20 Frequently Occuring Fake news Trigrams', "Trigrams", n=3)

"""# Fine-tuning BERT"""

!pip install transformers

import torch
from transformers.file_utils import is_tf_available, is_torch_available, is_torch_tpu_available
from transformers import BertTokenizerFast, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
import numpy as np
from sklearn.model_selection import train_test_split

import random

def set_seed(seed: int):
    """
    Helper function for reproducible behavior to set the seed in ``random``, ``numpy``, ``torch`` and/or
``tf`` (if
    installed).

    Args:
        seed (:obj:`int`): The seed to set.
    """
    random.seed(seed)
    np.random.seed(seed)
    if is_torch_available():
```

```python
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        # ^^ safe to call this function even if cuda is not available
    if is_tf_available():
        import tensorflow as tf

        tf.random.set_seed(seed)

set_seed(1)

# the model we gonna train, base uncased BERT
# check text classification models here: https://huggingface.co/models?filter=text-classification
model_name = "bert-base-uncased"
# max sequence length for each document/sentence sample
max_length = 512

# load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained(model_name, do_lower_case=True)

news_df = news_d[news_d['text'].notna()]
news_df = news_df[news_df["author"].notna()]
news_df = news_df[news_df["title"].notna()]

def prepare_data(df, test_size=0.2, include_title=True, include_author=True):
  texts = []
  labels = []
  for i in range(len(df)):
    text = df["text"].iloc[i]
    label = df["label"].iloc[i]
    if include_title:
      text = df["title"].iloc[i] + " - " + text
    if include_author:
      text = df["author"].iloc[i] + " : " + text
    if text and label in [0, 1]:
      texts.append(text)
      labels.append(label)
  return train_test_split(texts, labels, test_size=test_size)

```

```
train_texts, valid_texts, train_labels, valid_labels = prepare_data(news_df)

print(len(train_texts), len(train_labels))
print(len(valid_texts), len(valid_labels))

# tokenize the dataset, truncate when passed `max_length`,
# and pad with 0's when less than `max_length`
train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=max_length)
valid_encodings = tokenizer(valid_texts, truncation=True, padding=True, max_length=max_length)

class NewsGroupsDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {k: torch.tensor(v[idx]) for k, v in self.encodings.items()}
        item["labels"] = torch.tensor([self.labels[idx]])
        return item

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

# convert our tokenized data into a torch Dataset
train_dataset = NewsGroupsDataset(train_encodings, train_labels)
valid_dataset = NewsGroupsDataset(valid_encodings, valid_labels)

# load the model
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)

from sklearn.metrics import accuracy_score

def compute_metrics(pred):
  labels = pred.label_ids
  preds = pred.predictions.argmax(-1)
  # calculate accuracy using sklearn's function
  acc = accuracy_score(labels, preds)
  return {
```

```
    'accuracy': acc,
  }

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=1,              # total number of training epochs
    per_device_train_batch_size=10,  # batch size per device during training
    per_device_eval_batch_size=20,   # batch size for evaluation
    warmup_steps=100,                # number of warmup steps for learning rate scheduler
    logging_dir='./logs',            # directory for storing logs
    load_best_model_at_end=True,     # load the best model when finished training (default metric is loss)
    # but you can specify `metric_for_best_model` argument to change to accuracy or other metric
    logging_steps=200,               # log & save weights each logging_steps
    save_steps=200,
    evaluation_strategy="steps",     # evaluate each `logging_steps`
)

trainer = Trainer(
    model=model,                         # the instantiated Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=valid_dataset,          # evaluation dataset
    compute_metrics=compute_metrics,     # the callback that computes metrics of interest
)

# train the model
trainer.train()

# evaluate the current model after training
trainer.evaluate()

# saving the fine tuned model & tokenizer
model_path = "fake-news-bert-base-uncased"
model.save_pretrained(model_path)
tokenizer.save_pretrained(model_path)

def get_prediction(text, convert_to_label=False):
```

```
    # prepare our text into tokenized sequence
    inputs = tokenizer(text, padding=True, truncation=True, max_length=max_length,
return_tensors="pt").to("cuda")
    # perform inference to our model
    outputs = model(**inputs)
    # get output probabilities by doing softmax
    probs = outputs[0].softmax(1)
    # executing argmax function to get the candidate label
    d = {
        0: "reliable",
        1: "fake"
    }
    if convert_to_label:
      return d[int(probs.argmax())]
    else:
      return int(probs.argmax())

real_news = """
Tim Tebow Will Attempt Another Comeback, This Time in Baseball - The New York Times",Daniel
Victor,"If at first you don't succeed, try a different sport. Tim Tebow, who was a Heisman
quarterback at the University of Florida but was unable to hold an N. F. L. job, is pursuing a career in
Major League Baseball. He will hold a workout for M. L. B. teams this month, his agents told ESPN
and other news outlets. “This may sound like a publicity stunt, but nothing could be further from the
truth,” said Brodie Van Wagenen,   of CAA Baseball, part of the sports agency CAA Sports, in the
statement. “I have seen Tim’s workouts, and people inside and outside the industry  —   scouts,
executives, players and fans  —   will be impressed by his talent. ” It’s been over a decade since
Tebow, 28, has played baseball full time, which means a comeback would be no easy task. But the
former major league catcher Chad Moeller, who said in the statement that he had been training Tebow
in Arizona, said he was “beyond impressed with Tim’s athleticism and swing. ” “I see bat speed and
power and real baseball talent,” Moeller said. “I truly believe Tim has the skill set and potential to
achieve his goal of playing in the major leagues and based on what I have seen over the past two
months, it could happen relatively quickly. ” Or, take it from Gary Sheffield, the former   outfielder.
News of Tebow’s attempted comeback in baseball was greeted with skepticism on Twitter. As a junior
at Nease High in Ponte Vedra, Fla. Tebow drew the attention of major league scouts, batting . 494 with
four home runs as a left fielder. But he ditched the bat and glove in favor of pigskin, leading Florida to
two national championships, in 2007 and 2009. Two former scouts for the Los Angeles Angels told
WEEI, a Boston radio station, that Tebow had been under consideration as a high school junior.
“’x80’x9cWe wanted to draft him, ’x80’x9cbut he never sent back his information card,” said one of
the scouts, Tom Kotchman, referring to a questionnaire the team had sent him. “He had a strong arm
and had a lot of power,” said the other scout, Stephen Hargett. “If he would have been there his senior
year he definitely would have had a good chance to be drafted. ” “It was just easy for him,” Hargett
added. “You thought, If this guy dedicated everything to baseball like he did to football how good
could he be?” Tebow’s high school baseball coach, Greg Mullins, told The Sporting News in 2013 that
he believed Tebow could have made the major leagues. “He was the leader of the team with his
passion, his fire and his energy,” Mullins said. “He loved to play baseball, too. He just had a bigger fire
```

```python
for football. ” Tebow wouldn’t be the first athlete to switch from the N. F. L. to M. L. B. Bo Jackson had one   season as a Kansas City Royal, and Deion Sanders played several years for the Atlanta Braves with mixed success. Though Michael Jordan tried to cross over to baseball from basketball as a    in 1994, he did not fare as well playing one year for a Chicago White Sox minor league team. As a football player, Tebow was unable to match his college success in the pros. The Denver Broncos drafted him in the first round of the 2010 N. F. L. Draft, and he quickly developed a reputation for clutch performances, including a memorable   pass against the Pittsburgh Steelers in the 2011 Wild Card round. But his stats and his passing form weren’t pretty, and he spent just two years in Denver before moving to the Jets in 2012, where he spent his last season on an N. F. L. roster. He was cut during preseason from the New England Patriots in 2013 and from the Philadelphia Eagles in 2015.
"""

get_prediction(real_news, convert_to_label=True)

# read the test set
test_df = pd.read_csv("test.csv")

test_df.head()

# make a  of the testing set
new_df = test_df.()

# add a new column that contains the author, title and article content
new_df["new_text"] = new_df["author"].astype(str) + " : " + new_df["title"].astype(str) + " - " + new_df["text"].astype(str)
new_df.head()

# get the prediction of all the test set
new_df["label"] = new_df["new_text"].apply(get_prediction)

# make the submission file
final_df = new_df[["id", "label"]]
final_df.to_csv("submit_final.csv", index=False)
```

CHAPTER 3: Paraphrase Text using Transformers in Python

Explore different pre-trained transformer models in transformers library to paraphrase sentences in Python.

Paraphrasing is the process of coming up with someone else's ideas in your own words. To paraphrase a text, you have to rewrite it without changing its meaning.

In this tutorial, we will explore different pre-trained transformer models for automatically paraphrasing text using the Huggingface transformers library in Python.

To get started, let's install the required libraries first:

```
$ pip install transformers sentencepiece sacremoses
```

Importing everything from transformers library:

```
from transformers import *
```

Pegasus Transformer

In this section, we'll use the Pegasus transformer architecture model that was fine-tuned for paraphrasing instead of summarization. To instantiate the model, we need to use `PegasusForConditionalGeneration` as it's a form of text generation:

```
model =
PegasusForConditionalGeneration.from_pretrained("tuner007/pegasus_paraph
tokenizer =
```

```
PegasusTokenizerFast.from_pretrained("tuner007/pegasus_paraphrase")
```

Next, let's make a general function that takes a model, its tokenizer, the target sentence and returns the paraphrased text:

```
def get_paraphrased_sentences(model, tokenizer, sentence,
num_return_sequences=5, num_beams=5):
  # tokenize the text to be form of a list of token IDs
  inputs = tokenizer([sentence], truncation=True, padding="longest",
return_tensors="pt")
  # generate the paraphrased sentences
  outputs = model.generate(
    **inputs,
    num_beams=num_beams,
    num_return_sequences=num_return_sequences,
  )
  # decode the generated sentences using the tokenizer to get them back to
text
  return tokenizer.batch_decode(outputs, skip_special_tokens=True)
```

We also add the possibility of generating multiple paraphrased sentences by passing `num_return_sequences` to the `model.generate()` method.

We also set `num_beams` so we generate the paraphrasing using beam search. Setting it to 5 will allow the model to look ahead for five possible words to keep the most likely hypothesis at each time step and choose the one that has the overall highest probability.

I highly suggest you check this blog post to learn more about the parameters of the `model.generate()` method.

Let's use the function now:

```python
sentence = "Learning is the process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences."
```

```python
get_paraphrased_sentences(model, tokenizer, sentence, num_beams=10, num_return_sequences=10)
```

We set `num_beams` to 10 and prompt the model to generate ten different sentences; here is the output:

```
['Learning involves the acquisition of new understanding, knowledge, behaviors, skills, values, attitudes, and preferences.',
 'Learning is the acquisition of new understanding, knowledge, behaviors, skills, values, attitudes, and preferences.',
 'The process of learning is the acquisition of new understanding, knowledge, behaviors, skills, values, attitudes, and preferences.',
 'Gaining new understanding, knowledge, behaviors, skills, values, attitudes, and preferences is the process of learning.',
 'New understanding, knowledge, behaviors, skills, values, attitudes, and preferences are acquired through learning.',
 'Learning is the acquisition of new understanding, knowledge, behaviors, skills, values, attitudes and preferences.',
 'The process of learning is the acquisition of new understanding, knowledge, behaviors, skills, values, attitudes and preferences.',
 'New understanding, knowledge, behaviors, skills, values, attitudes, and preferences can be acquired through learning.',
 'New understanding, knowledge, behaviors, skills, values, attitudes, and preferences are what learning is about.',
 'Gaining new understanding, knowledge, behaviors, skills, values, attitudes, and preferences is a process of learning.']
```

Outstanding results! Most of the generations are accurate and can be used. You can try different sentences from your mind and see the results yourself.

T5 Transformer

This section will explore the T5 architecture model that was fine-tuned on the PAWS dataset. PAWS consists of 108,463 human-labeled and 656k noisily labeled pairs. Let's load the model and the tokenizer:

```
tokenizer = AutoTokenizer.from_pretrained("Vamsi/T5_Paraphrase_Paws")
model = AutoModelForSeq2SeqLM.from_pretrained("Vamsi/T5_Paraphrase_Paws")
```

Let's use our previously defined function:

```
get_paraphrased_sentences(model, tokenizer, "One of the best ways to learn is to teach what you've already learned")
```

Output:

```
["One of the best ways to learn is to teach what you've already learned.",
 'One of the best ways to learn is to teach what you have already learned.',
 'One of the best ways to learn is to teach what you already know.',
 'One of the best ways to learn is to teach what you already learned.',
 "One of the best ways to learn is to teach what you've already learned."]
```

These are promising results too. However, if you get some not-so-good paraphrased text, you can append the input text with `"paraphrase: "`, as T5 was intended for multiple text-to-text NLP tasks such as machine translation, text summarization, and more. It was pre-trained and fine-tuned like that.

You can check the model card here.

Parrot Paraphraser

Finally, let's use a fine-tuned T5 model architecture called Parrot. It is an augmentation framework built to speed-up training NLU models. The author of the fine-tuned model did a small library to perform paraphrasing. Let's install it:

```
$ pip install
git+https://github.com/PrithivirajDamodaran/Parrot_Paraphraser.git
```

Importing it and initializing the model:

```
from parrot import Parrot

parrot = Parrot()
```

This will download the models' weights and the tokenizer, give it some time, and it'll finish in a few seconds to several minutes, depending on your Internet connection.

This library uses more than one model. It uses one model for paraphrasing, one for calculating adequacy, another for calculating fluency, and the last for diversity.

Let's use the previous sentences and another one and see the results:

```
phrases = [
  sentence,
  "One of the best ways to learn is to teach what you've already learned",
  "Paraphrasing is the process of coming up with someone else's ideas in your
own words"
]
```

```
for phrase in phrases:
  print("-"*100)
  print("Input_phrase: ", phrase)
  print("-"*100)
  paraphrases = parrot.augment(input_phrase=phrase)
  for paraphrase in paraphrases:
   print(paraphrase)
```

With this library, we simply use the `parrot.augment()` method and pass the sentence in a text form, it returns several candidate paraphrased texts. Check the output:

```
----------------------------------------------------------------------------------------------------
Input_phrase:  Learning is the process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences.
----------------------------------------------------------------------------------------------------
('learning is the process of acquiring new knowledge behaviors skills values attitudes and preferences', 27)
('learning is the process of acquiring new understanding knowledge behaviors skills values attitudes and preferences', 13)
----------------------------------------------------------------------------------------------------
Input_phrase:  One of the best ways to learn is to teach what you've already learned
----------------------------------------------------------------------------------------------------
('one of the best ways to learn is to teach what you know', 29)
('one of the best ways to learn is to teach what you already know', 21)
```

```
('one of the best ways to learn is to teach what you have already learned', 15)
----------------------------------------------------------------------------------------------
Input_phrase:  Paraphrasing is the process of coming up with someone else's ideas in your own words
----------------------------------------------------------------------------------------------
("paraphrasing is the process of coming up with a person's ideas in your own words", 23)
("paraphrasing is the process of coming up with another person's ideas in your own words", 23)
("paraphrasing is the process of coming up with another's ideas in your own words", 22)
("paraphrasing is the process of coming up with someone's ideas in your own words", 17)
("paraphrasing is the process of coming up with somebody else's ideas in your own words", 15)
("paraphrasing is the process of coming up with someone else's ideas in your own words", 12)
```

The number accompanied with each sentence is the diversity score. The higher the value, the more diverse the sentence from the original.

You can check the [Parrot Paraphraser repository here](#).

You can find [ParaphrasingTool](#) as an example that uses transformers to fine-tune their rewriting models.

Conclusion

Alright! That's it for the tutorial. Hopefully, you have explored the most valuable ways to perform automatic text paraphrasing using transformers and AI in general.

You can get the complete code here or the Colab notebook here.

SourceCode:

paraphrasing_with_transformers.py

```
# -*- coding: utf-8 -*-
"""Paraphrasing-with-Transformers_PythonCode.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1bPfvSF7bJqDfw9ZMgfIZPd1Bk-fW7AJY
"""

# !pip install transformers sentencepiece

from transformers import *

# models we gonna use for this tutorial
model_names = [
  "tuner007/pegasus_paraphrase",
  "Vamsi/T5_Paraphrase_Paws",
  "prithivida/parrot_paraphraser_on_T5", # Parrot
]

model = PegasusForConditionalGeneration.from_pretrained("tuner007/pegasus_paraphrase")
tokenizer = PegasusTokenizerFast.from_pretrained("tuner007/pegasus_paraphrase")

def get_paraphrased_sentences(model, tokenizer, sentence, num_return_sequences=5, num_beams=5):
  # tokenize the text to be form of a list of token IDs
  inputs = tokenizer([sentence], truncation=True, padding="longest", return_tensors="pt")
  # generate the paraphrased sentences
  outputs = model.generate(
    **inputs,
    num_beams=num_beams,
```

```
    num_return_sequences=num_return_sequences,
  )
  # decode the generated sentences using the tokenizer to get them back to text
  return tokenizer.batch_decode(outputs, skip_special_tokens=True)

sentence = "Learning is the process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences."

get_paraphrased_sentences(model, tokenizer, sentence, num_beams=10, num_return_sequences=10)

get_paraphrased_sentences(model, tokenizer, "To paraphrase a source, you have to rewrite a passage without changing the meaning of the original text.", num_beams=10, num_return_sequences=10)

tokenizer = AutoTokenizer.from_pretrained("Vamsi/T5_Paraphrase_Paws")
model = AutoModelForSeq2SeqLM.from_pretrained("Vamsi/T5_Paraphrase_Paws")

get_paraphrased_sentences(model, tokenizer, "paraphrase: " + "One of the best ways to learn is to teach what you've already learned")

# !pip install git+https://github.com/PrithivirajDamodaran/Parrot_Paraphraser.git

from parrot import Parrot

parrot = Parrot()

phrases = [
  sentence,
  "One of the best ways to learn is to teach what you've already learned",
  "Paraphrasing is the process of coming up with someone else's ideas in your own words"
]

for phrase in phrases:
  print("-"*100)
  print("Input_phrase: ", phrase)
  print("-"*100)
  paraphrases = parrot.augment(input_phrase=phrase)
  if paraphrases:
    for paraphrase in paraphrases:
```

```
print(paraphrase)
```

CHAPTER 4: Text Generation with Transformers in Python

Learn how you can generate any type of text with GPT-2 and GPT-J transformer models with the help of Huggingface transformers library in Python.

Text generation is the task of automatically generating text using machine learning so that it cannot be distinguishable whether it's written by a human or a machine. It is also widely used for text suggestion and completion in various real-world applications.

In recent years, a lot of transformer-based models appeared to be great at this task. One of the most known is the GPT-2 model which was trained on massive unsupervised text, that generates quite impressive text.

Another major breakthrough appeared when OpenAI released GPT-3 paper and its capabilities, this model is too massive that is more than 1400 times larger than its previous version (GPT-2).

Unfortunately, we cannot use GPT-3 as OpenAI did not release the model weights, and even if it did, we as normal people won't be able to have that machine that can load the model weights into the memory, because it's too large.

Luckily, EleutherAI did a great job trying to mimic the capabilities of GPT-3 by releasing the GPT-J model. GPT-J model has 6 billion parameters consisting of 28 layers with a dimension of 4096, it was pre-trained on the Pile dataset, which is a large-scale dataset created by EleutherAI itself.

The Pile dataset is a massive one with a size of over 825GB, it consists of 22 sub-datasets including Wikipedia English (6.38GB), GitHub (95.16GB), Stack Exchange (32.2GB), ArXiv (56.21GB), and more. This explains the amazing performance of GPT-J that you'll hopefully discover in this tutorial.

In this guide, we're going to perform text generation using GPT-2 as well as EleutherAI models using the Huggingface Transformers library in Python.

The below table shows some of the useful models along with their number of parameters and size, I suggest you choose the largest you can fit in your environment memory:

Model	Number of Parameters	Size
gpt2	**124M**	**523MB**
EleutherAI/gpt-neo-125M	125M	502MB
EleutherAI/gpt-neo-1.3B	1.3B	4.95GB
EleutherAI/gpt-neo-2.7B	2.7B	9.94GB
EleutherAI/gpt-j-6B	**6B**	**22.5GB**

The `EleutherAI/gpt-j-6B` model has 22.5GB of size, so make sure you have at least a memory of more than 22.5GB to be able to perform inference on the model. The good news is that Google Colab with the High-RAM option worked for me. If you're not able to load that big model, you can try other smaller versions such as `EleutherAI/gpt-neo-2.7B` or `EleutherAI/gpt-neo-1.3B`. The models we gonna use in this tutorial are the highlighted ones in the above table.

Note that this is different from generating AI chatbot conversations using models such as DialoGPT. If you want that, we have a tutorial for it, make sure to check it out.

Let's get started, installing the transformers library:

```
$ pip install transformers
```

In this tutorial, we will only use the pipeline API, as it'll be more than enough for text generation.

Let's get started by the standard GPT-2 model:

```python
from transformers import pipeline

# download & load GPT-2 model
gpt2_generator = pipeline('text-generation', model='gpt2')
```

First, let's use GPT-2 model to generate 3 different sentences by sampling from the top 50 candidates:

```python
# generate 3 different sentences
# results are sampled from the top 50 candidates
sentences = gpt2_generator("To be honest, neural networks", do_sample=True, top_k=50, temperature=0.6, max_length=128, num_return_sequences=3)
for sentence in sentences:
  print(sentence["generated_text"])
  print("="*50)
```

Output:

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
To be honest, neural networks are still not very powerful. So they can't perform a lot of complex tasks like predicting the future — and they can't do that. They're just not very good at learning.

Instead, researchers are using the deep learning technology that's built into the brain to learn from the other side of the brain. It's called deep learning, and it's pretty straightforward.

For example, you can do a lot of things that are still not very well understood by the outside world. For example, you can read a lot of information, and it's
```

```
a lot easier to understand what the person
==================================================
To be honest, neural networks are not perfect, but they are pretty good at it. I've used them to build a lot of things. I've built a lot of things. I'm pretty good at it.

When we talk about what we're doing with our AI, it's kind of like a computer is going to go through a process that you can't see. It's going to have to go through it for you to see it. And then you can see it. And you can see it. It's going to be there for you. And then it's going to be there for you to see it
==================================================
To be honest, neural networks are going to be very interesting to study for a long time. And they're going to be very interesting to read, because they're going to be able to learn a lot from what we've learned. And this is where the challenge is, and this is also something that we've been working on, is that we can take a neural network and make a neural network that's a lot simpler than a neural network, and then we can do that with a lot more complexity.

So, I think it's possible that in the next few years, we may be able to make a neural network that
==================================================
```

I have set `top_k` to 50 which means we pick the 50 highest probability vocabulary tokens to keep for filtering, we also decrease the `temperature` to 0.6 (default is 1) to increase the probability of picking high probability tokens, setting it to 0 is the same as greedy search (i.e picking the most probable token)

Notice the third sentence was cut and not completed, you can always increase the `max_length` to generate more tokens.

By passing the input text to the TextGenerationPipeline (`pipeline` object), we're passing the arguments to the `model.generate()` method. Therefore, I highly suggest you check the parameters of the model.generate() reference for a more customized generation. I also suggest you read this blog post explaining most of the decoding techniques the method offers.

Now we have explored GPT-2, it's time to dive into the fascinating GPT-J:

```
# download & load GPT-J model! It's 22.5GB in size
gpt_j_generator = pipeline('text-generation', model='EleutherAI/gpt-j-6B')
```

The model size is about 22.5GB, make sure your environment is capable of loading the model to the memory, I'm using the High-RAM instance on Google Colab and it's running quite well. However, it may take a while to generate sentences, especially when you pass a higher value of max_length.

Let's pass the same parameters, but with a different prompt:

```
# generate sentences with TOP-K sampling
sentences = gpt_j_generator("To be honest, robots will", do_sample=True,
top_k=50, temperature=0.6, max_length=128, num_return_sequences=3)
for sentence in sentences:
  print(sentence["generated_text"])
  print("="*50)
```

Output:

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
To be honest, robots will never replace humans.

The reason is simple: We humans are far more complex than the average
robot.
```

The human brain is a marvel of complexity, capable of performing many thousands of tasks simultaneously. There are over 100 billion neurons in the human brain, with over 10,000 connections between each neuron, and neurons are capable of firing over a million times per second.

We have a brain that can reason, learn, and remember things. We can learn and retain information for a lifetime. We can communicate, collaborate, and work together to achieve goals. We can learn languages, play instruments

==

To be honest, robots will probably replace many human jobs.

That's the prediction of a team of economists from the University of Oxford.

They say that in the future we can expect to see more robots doing jobs that are tedious, repetitive and dangerous.

Even something as simple as a housekeeper could be a thing of the past.

The researchers also think that robots will become cheaper and more versatile over time.

The idea of a robot housekeeper has been a popular one.

One company has already started a trial by offering a robot to take care of your home.

But the

==

To be honest, robots will never replace the human workforce. It’s not a matter of if, but when. I can’t believe I’m writing this, but I’m glad I am.

```
Let’s start with what robots do. Robots are a form of technology. There’s a difference between a technology and a machine. A machine is a physical object designed to perform a specific task. A technology is a system of machines.

For example, a machine can be used for a specific purpose, but it still requires humans to operate it. A technology can be
==================================================
```

Honestly, I can't distinguish whether this is generated by a neural network or written by a human being!

Since GPT-J and other EleutherAI pre-trained models are trained on the Pile dataset, it can not only generate English text, but it can talk anything, let's try to generate Python code:

```
# generate Python Code!
print(gpt_j_generator(
"""
import os
# make a list of all african countries
""",
    do_sample=True, top_k=10, temperature=0.05, max_length=256)[0]["generated_text"])
```

Output:

```
import os
# make a list of all african countries
african_countries = ['Algeria', 'Angola', 'Benin', 'Burkina Faso', 'Burundi', 'Cameroon', 'Cape Verde', 'Central African Republic', 'Chad', 'Comoros',
```

```
'Congo', 'Democratic Republic of Congo', 'Djibouti', 'Egypt', 'Equatorial
Guinea', 'Eritrea', 'Ethiopia', 'Gabon', 'Gambia', 'Ghana', 'Guinea', 'Guinea-
Bissau', 'Kenya', 'Lesotho', 'Liberia', 'Libya', 'Madagascar', 'Malawi', 'Mali',
'Mauritania', 'Mauritius', 'Morocco', 'Mozambique', 'Namibia', 'Niger',
'Nigeria', 'Rwanda', 'Sao Tome and Principe', 'Senegal', 'Sierra Leone',
'Somalia', 'South Africa', 'South Sudan', 'Sudan', 'Swaziland', 'Tanzania',
'Togo', 'Tunisia',
```

I prompted the model with an `import os` statement to indicate that's Python code, and I did a comment on listing African countries. Surprisingly, it not only got the syntax of Python right, and generated African countries, but it also listed the countries in Alphabetical order and also chose a suitable variable name!

I definitely invite you to play around with the model and let me know in the comments if you find anything even more interesting.

Notice I have lowered the `temperature` to 0.05, as this is not really an open-ended generation, I want the African countries to be correct as well as the Python syntax, I have tried increasing the temperature in this type of generation and it led to misleading generation.

One more Python prompt:

```
print(gpt_j_generator(
"""
import cv2

image = "image.png"

# load the image and flip it
""",
    do_sample=True, top_k=10, temperature=0.05, max_length=256)[0]
["generated_text"])
```

Output:

```
import cv2

image = "image.png"

# load the image and flip it
img = cv2.imread(image)
img = cv2.flip(img, 1)

# resize the image to a smaller size
img = cv2.resize(img, (100, 100))

# convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# threshold the image to get the black and white image
thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY)

# show the image
cv2.imshow("thresh", thresh)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The model successfully generated working OpenCV code, where it loaded the image, applies the `cv2.flip()` function to the image and resizes it, and then continue with making the image black and white, interesting.

Next, let's try Bash scripting:

```
# complete bash script!
print(gpt_j_generator(
"""
# get .py files in /opt directory
ls *.py /opt
# get public ip address
""", max_length=256, top_k=50, temperature=0.05, do_sample=True)[0]
["generated_text"])
```

Check this out:

```
# get .py files in /opt directory
ls *.py /opt
# get public ip address
ip addr show dev eth0 | grep inet | awk '{print $2}' | cut -d/ -f1
# get mac address
ip link show dev eth0 | grep HWaddr | awk '{print $2}' | cut -d/ -f1
# get ip address
ip addr show dev eth0 | grep inet | awk '{print $2}' | cut -d/ -f1
```

The first command worked like a charm in my machine!

Another shell script:

```
# generating bash script!
print(gpt_j_generator(
"""
# update the repository
```

```
sudo apt-get update
# install and start nginx
""", max_length=128, top_k=50, temperature=0.1, do_sample=True)[0]
["generated_text"])
```

I have updated the repository using the `apt-get` command, and prompted to try to generate the commands for installing and starting Nginx, here is the output:

```
# update the repository
sudo apt-get update
# install and start nginx
sudo apt-get install nginx
sudo service nginx start

# create a new user
sudo useradd -m -s /bin/bash -G sudo -u www-data www-data
# add the user to the sudoers file
sudo visudo
# add the user to the www-data group
sudo usermod -aG www-data www-data
# add the user to the www-data group
sudo usermod -aG www-data www-data
# add the
```

The model successfully generated the two responsible commands for installing Nginx, and starting the webserver! It then tries to create a user and add it to the sudoers. However, notice the repetition, we can get rid of that by setting the `repetition_penalty` parameter (default is 1, i.e no penalty), check [this paper](#) for more information.

Trying Java now, prompting the model with Java main function wrapped in a Test class and adding a comment to print the first 20 Fibonacci numbers:

```
# Java code!
print(gpt_j_generator(
"""
public class Test {

public static void main(String[] args){
  // printing the first 20 fibonacci numbers
""", max_length=256, top_k=50, temperature=0.1, do_sample=True)[0]
["generated_text"])
```

Extraordinarily, the model added the complete Java code for generating Fibonacci numbers:

```
public class Test {

public static void main(String[] args){
  // printing the first 20 fibonacci numbers
  for(int i=0;i<20;i++){
    System.out.println(fibonacci(i));
  }
}

public static int fibonacci(int n){
  if(n<2){
    return n;
  }
  else{
```

```
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
}

A:

You need to return the value from the recursive call.
public static int fibonacci(int n){
  if(n<2){
    return n;
  }
  else{
    return fibonacci(n-1) + fibonacci(n-2);
  }
}

A:

You need to return the value from the recursive call.
public static int fibonacci(int n){
  if(n<2){
    return n;
  }
  else{
```

I have executed the code before the weird `"A:"`, not only it's a working code, but it generated the correct sequence!

Finally, Let's try generating LaTeX code:

```
# LATEX!
print(gpt_j_generator(
r"""
% list of Asian countries
\begin{enumerate}
""", max_length=128, top_k=15, temperature=0.1, do_sample=True)[0]
["generated_text"])
```

I tried to begin an ordered list in LaTeX, and before that, I added a comment indicating a list of Asian countries, output:

```
% list of Asian countries
\begin{enumerate}
\item \textbf{India}
\item \textbf{China}
\item \textbf{Japan}
\item \textbf{Korea}
\item \textbf{Taiwan}
\item \textbf{Hong Kong}
\item \textbf{Singapore}
\item \textbf{Malaysia}
\item \textbf{Indonesia}
\item \textbf{Philippines}
\item \textbf{Thailand}
\item \text
```

A correct syntax with the right countries!

Conclusion

Alright, that's it for this tutorial, we had a lot of fun generating such an interesting text.

If you run the above code snippets, you'll definitely get different results than mine, as we're sampling from a token distribution by setting the argument `do_sample` to `True`. Make sure you explore different decoding methods by checking this blog post from Huggingface, model.generate() method parameters

SourceCode:

textgeneration_transformers.py

```
# -*- coding: utf-8 -*-
"""TextGeneration-Transformers-PythonCodeTutorial.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1OUgJ92vQeFFYatf5gwtGulhA-mFwS0Md
"""

# !pip install transformers

from transformers import pipeline

# download & load GPT-2 model
gpt2_generator = pipeline('text-generation', model='gpt2')

# generate 3 different sentences
# results are sampled from the top 50 candidates
sentences = gpt2_generator("To be honest, neural networks", do_sample=True, top_k=50,
temperature=0.6, max_length=128, num_return_sequences=3)
```

```python
for sentence in sentences:
  print(sentence["generated_text"])
  print("="*50)

# download & load GPT-J model! It's 22.5GB in size
gpt_j_generator = pipeline('text-generation', model='EleutherAI/gpt-j-6B')

# generate sentences with TOP-K sampling
sentences = gpt_j_generator("To be honest, robots will", do_sample=True, top_k=50, temperature=0.6,
max_length=128, num_return_sequences=3)
for sentence in sentences:
  print(sentence["generated_text"])
  print("="*50)

# generate Python Code!
print(gpt_j_generator(
"""
import os
# make a list of all african countries
""",
    do_sample=True, top_k=10, temperature=0.05, max_length=256)[0]["generated_text"])

print(gpt_j_generator(
"""
import cv2

image = "image.png"

# load the image and flip it
""",
    do_sample=True, top_k=10, temperature=0.05, max_length=256)[0]["generated_text"])

# complete bash script!
print(gpt_j_generator(
"""
# get .py files in /opt directory
ls *.py /opt
# get public ip address
```

```python
""", max_length=256, top_k=50, temperature=0.05, do_sample=True)[0]["generated_text"])

# generating bash script!
print(gpt_j_generator(
"""
# update the repository
sudo apt-get update
# install and start nginx
""", max_length=128, top_k=50, temperature=0.1, do_sample=True)[0]["generated_text"])

# Java code!
print(gpt_j_generator(
"""
public class Test {

public static void main(String[] args){
  // printing the first 20 fibonacci numbers
""", max_length=128, top_k=50, temperature=0.1, do_sample=True)[0]["generated_text"])

# Commented out IPython magic to ensure Python compatibility.
# LATEX!
print(gpt_j_generator(
r"""
# % list of Asian countries
\begin{enumerate}
""", max_length=128, top_k=15, temperature=0.1, do_sample=True)[0]["generated_text"])
```

CHAPTER 5: Speech Recognition using Transformers in Python

Learn how to perform automatic speech recognition (ASR) using wav2vec2 transformer with the help of Huggingface transformers library in Python

Automatic Speech Recognition (ASR) is the technology that allows us to convert human speech into digital text. This tutorial will dive into the current state-of-the-art model called Wav2vec2 using the Huggingface transformers library in Python.

Wav2Vec2 is a pre-trained model that was trained on speech audio alone (self-supervised) and then followed by fine-tuning on transcribed speech data (LibriSpeech dataset). It has outperformed previous semi-supervised models.

As in Masked Language Modeling, Wav2Vec2 encodes speech audio via a multi-layer convolutional neural network and then masks spans of the resulting latent speech representations. These representations are then fed to a Transformer network to build contextualized representations; check the Wav2Vec2 paper for more information.

To get started, let's install the required libraries:

```
$ pip3 install transformers==4.11.2 soundfile sentencepiece torchaudio pydub pyaudio
```

We'll be using torchaudio for loading audio files. Note that you need to install PyAudio if you're going to use the code on your environment and PyDub if you're on a Colab environment. We are going to use them for recording from the microphone in Python.

Getting Started

Let's import our libraries:

```
from transformers import *
import torch
import soundfile as sf
# import librosa
import os
import torchaudio
```

Next, loading the processor and the model weights of wav2vec2:

```
# model_name = "facebook/wav2vec2-base-960h" # 360MB
model_name = "facebook/wav2vec2-large-960h-lv60-self" # 1.18GB

processor = Wav2Vec2Processor.from_pretrained(model_name)
model = Wav2Vec2ForCTC.from_pretrained(model_name)
```

There are two most used model architectures and weights for wav2vec. `wav2vec2-base-960h` is a base architecture with about 360MB of size, it achieved a 3.4% [Word Error Rate (WER)]() on the clean test set and was trained on 960 hours of LibriSpeech dataset on 16kHz sampled speech audio.

On the other hand, `wav2vec2-large-960h-lv60-self` is a larger model with about 1.18GB in size (probably won't fit your laptop RAM) but achieved 1.9% WER (the lower, the better) on the clean test set. So this one is much better for recognition but heavier and takes more time for inference. Feel free to choose which one suits you best.

Wav2Vec2 was trained using [Connectionist Temporal Classification (CTC)](), so that's why we're using the `Wav2Vec2ForCTC` class for loading the model.

Next, here are some audio samples:

```
# audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/16-122828-0002.wav"
audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/30-4447-0004.wav"
# audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/7601-291468-0006.wav"
```

Preparing the Audio File

Feel free to choose any of the above audio files. Below cell loads the audio file:

```
# load our wav file
speech, sr = torchaudio.load(audio_url)
speech = speech.squeeze()
# or using librosa
# speech, sr = librosa.load(audio_file, sr=16000)
sr, speech.shape
```

```
(16000, torch.Size([274000]))
```

The `torchaudio.load()` function loads the audio file and returns the audio as a vector and the sample rate. It also automatically downloads the file if it's a URL. If it's a path in the disk, it will also load it.

Note we use the `squeeze()` method as well, it is to remove the dimensions with the size of 1. i.e., converting tensor from `(1, 274000)` to `(274000,)`.

Next, we need to make sure the input audio file to the model has the sample rate of 16000Hz because wav2vec2 is trained on that:

```
# resample from whatever the audio sampling rate to 16000
resampler = torchaudio.transforms.Resample(sr, 16000)
speech = resampler(speech)
speech.shape
```

```
torch.Size([274000])
```

We used `Resample` from [torchaudio.transforms](), which helps us to convert the loaded audio file in the fly from one sampling rate to another.

Before we make the inference, we pass the audio vector to the wav2vec2 processor:

```
# tokenize our wav
input_values = processor(speech, return_tensors="pt", sampling_rate=16000)
["input_values"]
input_values.shape
```

```
torch.Size([1, 274000])
```

We specify the `sampling_rate` and pass `"pt"` to `return_tensors` argument to get PyTorch tensors in the results.

Performing Inference

Let's pass the vector into our model now:

```
# perform inference
logits = model(input_values)["logits"]
logits.shape
```

```
torch.Size([1, 856, 32])
```

Passing the logits to `torch.argmax()` to get the likely prediction:

```
# use argmax to get the predicted IDs
predicted_ids = torch.argmax(logits, dim=-1)
predicted_ids.shape
```

```
torch.Size([1, 856, 32])
```

Decoding them back to text, we also lower the text, as it's in all caps:

```
# decode the IDs to text
transcription = processor.decode(predicted_ids[0])
transcription.lower()
```

```
and missus goddard three ladies almost always at the service of an invitation
from hartfield and who were fetched and carried home so often that mister
woodhouse thought it no hardship for either james or the horses had it taken
place only once a year it would have been a grievance
```

Wrapping up the Code

Now let's collect all our previous code into a single function, which accepts the audio path and returns the transcription:

```
def get_transcription(audio_path):
    # load our wav file
    speech, sr = torchaudio.load(audio_path)
    speech = speech.squeeze()
    # or using librosa
    # speech, sr = librosa.load(audio_file, sr=16000)
    # resample from whatever the audio sampling rate to 16000
    resampler = torchaudio.transforms.Resample(sr, 16000)
    speech = resampler(speech)
    # tokenize our wav
    input_values = processor(speech, return_tensors="pt",
sampling_rate=16000)["input_values"]
    # perform inference
    logits = model(input_values)["logits"]
    # use argmax to get the predicted IDs
    predicted_ids = torch.argmax(logits, dim=-1)
    # decode the IDs to text
    transcription = processor.decode(predicted_ids[0])
    return transcription.lower()
```

Awesome, you can pass any audio speech file path:

```
get_transcription("http://www0.cs.ucl.ac.uk/teaching/GZ05/samples/lathe.wav
```

```
a late is a big tool grab every dish of sugar
```

Conclusion

Awesome, now if you want to use your voice, I have prepared a code snippet in the notebooks to record with your microphone. Feel free to choose the environment you're using:

- Colab Environment
- Local Environment

Note that there are other wav2vec2 weights trained by other people in different languages than English. Check the models' page and filter on the language of your desire to get the wanted model.

SourceCode:

AutomaticSpeechRecognition_PythonCodeTutorial.py

```
# %%
# !pip install transformers==4.11.2 datasets soundfile sentencepiece torchaudio pyaudio

# %%
from transformers import *
import torch
import soundfile as sf
# import librosa
import os
import torchaudio

# %%
# model_name = "facebook/wav2vec2-base-960h" # 360MB
model_name = "facebook/wav2vec2-large-960h-lv60-self" # 1.18GB

processor = Wav2Vec2Processor.from_pretrained(model_name)
```

```
model = Wav2Vec2ForCTC.from_pretrained(model_name)

# %%
# audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/16-122828-0002.wav"
audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/30-4447-0004.wav"
# audio_url = "https://github.com/x4nth055/pythoncode-tutorials/raw/master/machine-learning/speech-recognition/7601-291468-0006.wav"

# %%
# load our wav file
speech, sr = torchaudio.load(audio_url)
speech = speech.squeeze()
# or using librosa
# speech, sr = librosa.load(audio_file, sr=16000)
sr, speech.shape

# %%
# resample from whatever the audio sampling rate to 16000
resampler = torchaudio.transforms.Resample(sr, 16000)
speech = resampler(speech)
speech.shape

# %%
# tokenize our wav
input_values = processor(speech, return_tensors="pt", sampling_rate=16000)["input_values"]
input_values.shape

# %%
# perform inference
logits = model(input_values)["logits"]
logits.shape

# %%
# use argmax to get the predicted IDs
predicted_ids = torch.argmax(logits, dim=-1)
predicted_ids.shape
```

```python

# %%
# decode the IDs to text
transcription = processor.decode(predicted_ids[0])
transcription.lower()

# %%
def get_transcription(audio_path):
  # load our wav file
  speech, sr = torchaudio.load(audio_path)
  speech = speech.squeeze()
  # or using librosa
  # speech, sr = librosa.load(audio_file, sr=16000)
  # resample from whatever the audio sampling rate to 16000
  resampler = torchaudio.transforms.Resample(sr, 16000)
  speech = resampler(speech)
  # tokenize our wav
  input_values = processor(speech, return_tensors="pt", sampling_rate=16000)["input_values"]
  # perform inference
  logits = model(input_values)["logits"]
  # use argmax to get the predicted IDs
  predicted_ids = torch.argmax(logits, dim=-1)
  # decode the IDs to text
  transcription = processor.decode(predicted_ids[0])
  return transcription.lower()

# %%
get_transcription(audio_url)

# %%
import pyaudio
import wave

# the file name output you want to record into
filename = "recorded.wav"
# set the chunk size of 1024 samples
chunk = 1024
# sample format
```

```
FORMAT = pyaudio.paInt16
# mono, change to 2 if you want stereo
channels = 1
# 44100 samples per second
sample_rate = 16000
record_seconds = 10
# initialize PyAudio object
p = pyaudio.PyAudio()
# open stream object as input & output
stream = p.open(format=FORMAT,
                channels=channels,
                rate=sample_rate,
                input=True,
                output=True,
                frames_per_buffer=chunk)
frames = []
print("Recording...")
for i in range(int(sample_rate / chunk * record_seconds)):
    data = stream.read(chunk)
    # if you want to hear your voice while recording
    # stream.write(data)
    frames.append(data)
print("Finished recording.")
# stop and close stream
stream.stop_stream()
stream.close()
# terminate pyaudio object
p.terminate()
# save audio file
# open the file in 'write bytes' mode
wf = wave.open(filename, "wb")
# set the channels
wf.setnchannels(channels)
# set the sample format
wf.setsampwidth(p.get_sample_size(FORMAT))
# set the sample rate
wf.setframerate(sample_rate)
# write the frames as bytes
```

```
wf.writeframes(b"".join(frames))
# close the file
wf.close()

# %%
get_transcription("recorded.wav")

# %%
```

CHAPTER 6: Machine Translation using Transformers in Python

Learn how to use Huggingface transformer models to perform machine translation on various languages using transformers and PyTorch libraries in Python.

Machine translation is the process of using Machine Learning to automatically translate text from one language to another without any human intervention during the translation.

Neural machine translation emerged in recent years, outperforming all previous approaches. More specifically, neural networks based on attention called transformers did an outstanding job on this task.

This tutorial will teach you how to perform machine translation without any training. In other words, we'll be using pre-trained models from Huggingface transformer models.

The Helsinki-NLP models we will use are primarily trained on the OPUS dataset, a collection of translated texts from the web; it is free online data.

You can either make a new empty Python notebook or file to get started. You can also follow with the notebook in Colab by clicking the **Open In Colab** button above or down the article. First, let's install the required libraries:

```
$ pip install transformers==4.12.4 sentencepiece
```

Importing transformers:

```
from transformers import *
```

Using Pipeline API

Let's first get started with the library's pipeline API; we'll be using the models trained by Helsinki-NLP. You can check their page to see the available models they have:

```
# source & destination languages
src = "en"
dst = "de"

task_name = f"translation_{src}_to_{dst}"
model_name = f"Helsinki-NLP/opus-mt-{src}-{dst}"

translator  = pipeline(task_name, model=model_name,
tokenizer=model_name)
```

`src` and `dst` are the source and destination languages, respectively. Feel free to change for your needs. We dynamically change the name of `task_name` and `model_name` based on the source and destination languages, we then initialize the pipeline by specifying the `model` and `tokenizer` arguments as well. Let's test it out:

```
translator("You're a genius.")[0]["translation_text"]
```

Output:

```
Du bist ein Genie.
```

The pipeline API is pretty straightforward; we get the output by simply passing the text to the translator pipeline object.

Alright, let's test a longer text brought from Wikipedia:

```
article = """
Albert Einstein ( 14 March 1879 – 18 April 1955) was a German-born theoretical physicist, widely acknowledged to be one of the greatest physicists of all time.
Einstein is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.
Relativity and quantum mechanics are together the two pillars of modern physics.
His mass–energy equivalence formula E = mc2, which arises from relativity theory, has been dubbed "the world's most famous equation".
His work is also known for its influence on the philosophy of science.
He received the 1921 Nobel Prize in Physics "for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect", a pivotal step in the development of quantum theory.
His intellectual achievements and originality resulted in "Einstein" becoming synonymous with "genius"
"""
translator(article)[0]["translation_text"]
```

Output:

```
Albert Einstein (* 14. März 1879 – 18. April 1955) war ein deutscher theoretischer Physiker, der allgemein als einer der größten Physiker aller Zeiten anerkannt wurde.
Einstein ist am besten für die Entwicklung der Relativitätstheorie bekannt, aber er leistete auch wichtige Beiträge zur Entwicklung der Quantenmechaniktheorie.
Relativität und Quantenmechanik sind zusammen die beiden Säulen der
```

```
modernen Physik.
Seine Massenenergieäquivalenzformel E = mc2, die aus der Relativitätstheorie hervorgeht, wurde als „die berühmteste Gleichung der Welt" bezeichnet.
Seine Arbeit ist auch für ihren Einfluss auf die Philosophie der Wissenschaft bekannt.
Er erhielt 1921 den Nobelpreis für Physik „für seine Verdienste um die theoretische Physik und vor allem für seine Entdeckung des Gesetzes über den photoelektrischen Effekt", einen entscheidenden Schritt in der Entwicklung der Quantentheorie.
Seine intellektuellen Leistungen und Originalität führten dazu, dass „Einstein" zum Synonym für „Genius" wurde.
```

I have tested this output on Google Translate to get it back in English, and it seems to be an excellent translation!

Related: *How to Paraphrase Text using Transformers in Python.*

Manually Loading the Model

Since pipeline doesn't provide us with a lot of flexibility during translation generation, let's use the model and tokenizer for manual use:

```
def get_translation_model_and_tokenizer(src_lang, dst_lang):
  """
  Given the source and destination languages, returns the appropriate model
  See the language codes here: https://developers.google.com/admin-sdk/directory/v1/languages
  For the 3-character language codes, you can google for the code!
  """
  # construct our model name
  model_name = f"Helsinki-NLP/opus-mt-{src}-{dst}"
```

```python
# initialize the tokenizer & model
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
# return them for use
return model, tokenizer
```

The above function returns the appropriate model given the `src_lang` and `dst_lang` for source and destination languages, respectively. For a list of language codes, consider checking [this page](). For instance, let's try English to Chinese:

```python
# source & destination languages
src = "en"
dst = "zh"

model, tokenizer = get_translation_model_and_tokenizer(src, dst)
```

To translate our previous paragraph, we first need to tokenize the text:

```python
# encode the text into tensor of integers using the appropriate tokenizer
inputs = tokenizer.encode(article, return_tensors="pt", max_length=512, truncation=True)
print(inputs)
```

Output:

```
tensor([[32614, 53456,    22,   992,   776,   822,  4048,     8,  3484,   822,
           820, 50940,    17,    43,    13,  8214,    16, 32941, 34899, 60593,
             2,  5514,  7131,     9,    34,   141,     4,     3,  7680, 60593,
```

```
        24,     4,    61,   220,     6, 53456,    32,  1109,  3305,    15,
       320,     3, 19082,     4,  1294, 24030, 28453,     2,   187,   172,
        81,   157,   435,  1061,     9,     3,    92,     4,     3, 19082,
         4, 52682, 54813,     6, 45978, 28453,     7, 52682, 54813,    46,
      1105,     3,   263, 12538,     4,  6683, 46089,     6,  1608,  3196,
      3484, 45425, 50560, 14655,   509,     8,  6873,  4374,   149,  9132,
        62, 22703,    51,  1294, 24030, 28453, 19082,     2,    66,    74,
     16044, 18553,   258,    40,  1862,   431,    23,    24,   447, 23761,
     47364, 10594,  1608,   119,    32,    81,  3305,    15,    45,  6748,
        19,     3, 34857,     4,  4102,     6,   250,   948,     3,   912,
       774, 38354, 33321,    11, 58505,    40,  4161,   175,   307,     9,
     34899, 46089,     2,     7,   978,    15,   175, 34026,     4,     3,
       191,     4,     3, 17952, 57867,  1766, 19622,    13, 29632,  2827,
        11,     3,    92,     4, 52682, 19082,     6,  1608,  6875,  5710,
         7,  5099,  2665,  3897,    11,    40,   338,   767, 40272,   480,
      6588, 57380,    29,    40,  9994, 20506,   480,     0]])
```

The `tokenizer.encode()` method encodes the text into tokens and converts them to IDs, we set `return_tensors` to `"pt"` so it'll return a PyTorch tensor. We also set `max_length` to `512` and `truncation` to `True`.

Let's now use greedy search to generate the translation for this:

```
# generate the translation output using greedy search
greedy_outputs = model.generate(inputs)
# decode the output and ignore special tokens
print(tokenizer.decode(greedy_outputs[0], skip_special_tokens=True))
```

We simply use the `model.generate()` method to get the outputs, and since the outputs are also tokenized, we need to decode them back to human-readable format. We also set `skip_special_tokens` to `True` so we don't see tokens such as `<pad>`, etc. Here is the output:

```
阿 尔伯特 · 爱因斯坦 (1879 年 3 月 14 日至 1955 年 4 月 18 日 ) 是德国出生的理 论物理学家 , 被广泛承 认是有史以来最伟大的物理学家之一。爱因斯坦以发展相对论闻名 , 但他也 为量子力学理论的发展做出了 重要 贡献。相对论和量子力学是现代物理学的两大支柱。他的质量 —— 能源等 值公式 E = mc2 来自相 对论 , 被称作 “ 世界最著名的方程 ” 。他的工作也因其 对科学哲学的影响而著称。 他 获得了 1921 年 诺贝尔物理奖 ,“ 因 为他对理论物理学的服务 , 特 别是他发现了光电效应法 ”, 这是量子理论发展的关键一步。 他的智力成就和 创举导
致 “Einstein” 成 为 “genius” 的同 义词。
```

You can also use beam search instead of greedy search, which may generate better translations:

```
# generate the translation output using beam search
beam_outputs = model.generate(inputs, num_beams=3)
# decode the output and ignore special tokens
print(tokenizer.decode(beam_outputs[0], skip_special_tokens=True))
```

We set `num_beams` to `3`. I suggest reading this blog post or our tutorials on text summarization and conversational AI chatbots for more information about beams. The output:

```
阿 尔伯特 · 爱因斯坦 (1879 年 3 月 14 日至 1955 年 4 月 18 日 ) 是德国出生的理 论物理学家 , 被广泛承 认是有史以来最伟大的物理学家之一。爱因斯坦以发展相对论闻名 , 但他也 为量子力学理论的发展做出了重要贡献。相对论和量子力学是现代物理学的两大支柱。来自相对论的其质量 —— 能源等 值公式 E=mc2 被称作 “ 世界上最著名的方程式 ” 。他的工作也因其 对科学哲学的影响而著称。他获得了 1921 年 诺贝尔物理奖 ,“ 因 为他对理论物理学的服务 , 特 别是他发现了光电效应法 ”, 这是
```

```
量子理论发展的关键一步。他的智力成就和原创性导致
了 “Einstein” 与 “genius” 的同 义。
```

It was a slightly different translation, and both seemed to be good translations when I translated them back to English using Google Translate.

We can also generate more than one translation in one go:

```
# let's change target language
src = "en"
dst = "ar"

# get en-ar model & tokenizer
model, tokenizer = get_translation_model_and_tokenizer(src, dst)


# yet another example
text = "It can be severe, and has caused millions of deaths around the world as well as lasting health problems in some who have survived the illness."
# tokenize the text
inputs = tokenizer.encode(text, return_tensors="pt", max_length=512, truncation=True)
# this time we use 5 beams and return 5 sequences and we can compare!
beam_outputs = model.generate(
    inputs,
    num_beams=5,
    num_return_sequences=5,
    early_stopping=True,
)
for i, beam_output in enumerate(beam_outputs):
  print(tokenizer.decode(beam_output, skip_special_tokens=True))
```

```
print("="*50)
```

We set `num_return_sequences` to `5` to generate five different most probable translations, make sure that `num_beams >= num_return_sequences`, output:

```
ويمك ن أ ن تكو ن حادة ، وق د تسبب ت ف ي ملايي ن الوفيا ت ف ي جمي ع أنحا ء العالم ، فضل ا ع ن مشاك ل صحي ة دائم ة ف ي بع ض الذي ن نجوا م ن المر ض.
==================================================
ويمك ن أ ن تكو ن خطيرة ، وق د تسبب ت ف ي ملايي ن الوفيا ت ف ي جمي ع أنحا ء العالم ، فضل ا ع ن مشاك ل صحي ة دائم ة ف ي بع ض الذي ن نجوا م ن المر ض.
==================================================
ويمك ن أ ن تكو ن حادة ، وق د تسبب ت ف ي ملايي ن الوفيا ت ف ي جمي ع أنحا ء العالم ، فضل ا ع ن مشاك ل صحي ة دائم ة لد ى بع ض الذي ن نجوا م ن المر ض.
==================================================
ويمك ن أ ن تكو ن حادة ، وق د تسبب ت ف ي ملايي ن الوفيا ت ف ي جمي ع أنحا ء العالم ، فضل ا ع ن مشاك ل صحي ة دائم ة ف ي بع ض م ن نجوا م ن المر ض.
==================================================
ويمك ن أ ن تكو ن حادة ، وق د تسبب ت ف ي وفا ة ملايي ن الأشخا ص ف ي جمي ع أنحا ء العالم ، فضل ا ع ن مشاك ل صحي ة دائم ة ف ي بع ض الذي ن نجوا م ن المر ض.
==================================================
```

Conclusion

That's it for this tutorial! I suggest you use your two languages and your text to see which best suits you in terms of parameters in the `model.generate()` method.

As stated above, there are a lot of parameters in the `model.generate()` method, most of them are explained in the hugging face blog post or our tutorials on text summarization and conversational AI chatbot.

Also, there are 1300+ pre-trained models on the Helsinki-NLP page, so your native language is definitely present there!

SourceCode:

machine_translation.py

```
# -*- coding: utf-8 -*-
"""MachineTranslation-with-Transformers-PythonCode.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1RIcKVMVRcKVbhoyqpzy2s1KSchS4XJX2
"""

!pip install transformers==4.12.4 sentencepiece

from transformers import *

# source & destination languages
src = "en"
dst = "de"

task_name = f"translation_{src}_to_{dst}"
model_name = f"Helsinki-NLP/opus-mt-{src}-{dst}"

translator  = pipeline(task_name, model=model_name, tokenizer=model_name)

translator("You're a genius.")[0]["translation_text"]
```

```python

article = """
Albert Einstein ( 14 March 1879 – 18 April 1955) was a German-born theoretical physicist, widely acknowledged to be one of the greatest physicists of all time.
Einstein is best known for developing the theory of relativity, but he also made important contributions to the development of the theory of quantum mechanics.
Relativity and quantum mechanics are together the two pillars of modern physics.
His mass–energy equivalence formula E = mc2, which arises from relativity theory, has been dubbed "the world's most famous equation".
His work is also known for its influence on the philosophy of science.
He received the 1921 Nobel Prize in Physics "for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect", a pivotal step in the development of quantum theory.
His intellectual achievements and originality resulted in "Einstein" becoming synonymous with "genius"
"""

translator(article)[0]["translation_text"]

def get_translation_model_and_tokenizer(src_lang, dst_lang):
  """
  Given the source and destination languages, returns the appropriate model
  See the language codes here: https://developers.google.com/admin-sdk/directory/v1/languages
  For the 3-character language codes, you can google for the code!
  """
  # construct our model name
  model_name = f"Helsinki-NLP/opus-mt-{src}-{dst}"
  # initialize the tokenizer & model
  tokenizer = AutoTokenizer.from_pretrained(model_name)
  model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
  # return them for use
  return model, tokenizer

# source & destination languages
src = "en"
dst = "zh"

model, tokenizer = get_translation_model_and_tokenizer(src, dst)

```

```
# encode the text into tensor of integers using the appropriate tokenizer
inputs = tokenizer.encode(article, return_tensors="pt", max_length=512, truncation=True)
print(inputs)

# generate the translation output using greedy search
greedy_outputs = model.generate(inputs)
# decode the output and ignore special tokens
print(tokenizer.decode(greedy_outputs[0], skip_special_tokens=True))

# generate the translation output using beam search
beam_outputs = model.generate(inputs, num_beams=3)
# decode the output and ignore special tokens
print(tokenizer.decode(beam_outputs[0], skip_special_tokens=True))

# let's change target language
src = "en"
dst = "ar"

# get en-ar model & tokenizer
model, tokenizer = get_translation_model_and_tokenizer(src, dst)

# yet another example
text = "It can be severe, and has caused millions of deaths around the world as well as lasting health problems in some who have survived the illness."
# tokenize the text
inputs = tokenizer.encode(text, return_tensors="pt", max_length=512, truncation=True)
# this time we use 5 beams and return 5 sequences and we can compare!
beam_outputs = model.generate(
    inputs,
    num_beams=5,
    num_return_sequences=5,
    early_stopping=True,
)
for i, beam_output in enumerate(beam_outputs):
  print(tokenizer.decode(beam_output, skip_special_tokens=True))
  print("="*50)
```

CHAPTER 7: Train BERT from Scratch using Transformers in Python

Learn how you can pretrain BERT and other transformers on the Masked Language Modeling (MLM) task on your custom dataset using Huggingface Transformers library in Python

A pre-trained model is a model that was previously trained on a large dataset and saved for direct use or fine-tuning. In this tutorial, you will learn how you can train BERT (or any other transformer model) from scratch on your custom raw text dataset with the help of the Huggingface transformers library in Python.

Pre-training on transformers can be done with self-supervised tasks, below are some of the popular tasks done on BERT:

- **Masked Language Modeling (MLM)**: This task consists of masking a certain percentage of the tokens in the sentence, and the model is trained to predict those masked words. We'll be using this one in this tutorial.
- **Next Sentence Prediction (NSP)**: The model receives pairs of sentences as input and learns to predict whether the second sentence in the pair is the subsequent sentence in the original document.

To get started, we need to install 3 libraries:

```
$ pip install datasets transformers==4.18.0 sentencepiece
```

If you want to follow along, open up a new notebook, or Python file and import the necessary libraries:

```
from datasets import *
```

```python
from transformers import *
from tokenizers import *
import os
import json
```

Picking a Dataset

If you're willing to pre-train a transformer, then you most likely have a custom dataset. But for demonstration purposes in this tutorial, we're going to use the `cc_news` dataset, we'll be using [huggingface datasets](#) library for that. As a result, make sure to follow [this link](#) to get your custom dataset to be loaded into the library.

CC-News dataset contains news articles from news sites all over the world. It contains 708,241 news articles in English published between January 2017 and December 2019.

Downloading and preparing the dataset:

```python
# download and prepare cc_news dataset
dataset = load_dataset("cc_news", split="train")
```

There is only one split in the dataset, so we need to split it into training and testing sets:

```python
# split the dataset into training (90%) and testing (10%)
d = dataset.train_test_split(test_size=0.1)
d["train"], d["test"]
```

You can also pass the `seed` parameter to the `train_test_split()` method so it'll be the same sets after running multiple times.

Output:

```
(Dataset({
    features: ['title', 'text', 'domain', 'date', 'description', 'url', 'image_url'],
    num_rows: 637416
 }), Dataset({
    features: ['title', 'text', 'domain', 'date', 'description', 'url', 'image_url'],
    num_rows: 70825
 }))
```

Let's see how it looks like:

```
for t in d["train"]["text"][:3]:
  print(t)
  print("="*50)
```

Output (stripped):

```
Pretty sure women wish men did this better too!!
Q: A recent survey showed that 1/3 of men wish they did THIS better. What is...<STRIPPED>
==================================================
× GoDaddy boots neo-Nazi site after a derogatory story on the Charlottesville victim
The Daily Stormer, a white supremacist and neo-Nazi website,...
<STRIPPED>
==================================================
French bank Natixis under investigation over subprime losses
PARIS, Feb 15 Natixis has been placed under formal investigation...
<STRIPPED>
```

As mentioned previously, if you have your custom dataset, you can either follow [the link]() of setting up your dataset to be loaded as above, or you can use the `LineByLineTextDataset` class if your custom dataset is a text file where all sentences are separated by a new line.

However, a better way to set up your custom dataset is to split your text file into several chunk files using the `split` command or any other Python code, and load them using `load_dataset()` as we did above, like this:

```
# if you have huge custom dataset separated into files
# load the splitted files
files = ["train1.txt", "train2.txt"] # train3.txt, etc.
dataset = load_dataset("text", data_files=files, split="train")
```

If you have your custom data as one massive file, then you should divide it into a handful of text files (such as using the `split` command on Linux or Colab) before loading them using the `load_dataset()` function, as the runtime will crash if it exceeds the memory.

Training the Tokenizer

Next, we need to train our tokenizer. To do that, we need to write our dataset into text files, as that's what the [tokenizers library]() requires the input to be:

```
# if you want to train the tokenizer from scratch (especially if you have custom
# dataset loaded as datasets object), then run this cell to save it as files
# but if you already have your custom data as text files, there is no point using this
def dataset_to_text(dataset, output_filename="data.txt"):
  """Utility function to save dataset text to disk,
```

```
  useful for using the texts to train the tokenizer
  (as the tokenizer accepts files)"""
  with open(output_filename, "w") as f:
    for t in dataset["text"]:
      print(t, file=f)

# save the training set to train.txt
dataset_to_text(d["train"], "train.txt")
# save the testing set to test.txt
dataset_to_text(d["test"], "test.txt")
```

The main purpose of the above code cell is to save the dataset object as text files. If you already have your dataset as text files, then you should skip this step. Next, let's define some parameters:

```
special_tokens = [
  "[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]", "<S>", "<T>"
]
# if you want to train the tokenizer on both sets
# files = ["train.txt", "test.txt"]
# training the tokenizer on the training set
files = ["train.txt"]
# 30,522 vocab is BERT's default vocab size, feel free to tweak
vocab_size = 30_522
# maximum sequence length, lowering will result to faster training (when increasing batch size)
max_length = 512
# whether to truncate
truncate_longer_samples = False
```

The `files` list is the list of files to pass to the tokenizer for training. `vocab_size` is the vocabulary size of tokens. `max_length` is the maximum sequence length.

`truncate_longer_samples` is a boolean indicating whether we truncate sentences longer than the length of `max_length`, if it's set to `False`, we won't truncate the sentences, we group them together and split them by `max_length`, so all the resulting sentences will have the length of `max_length`.

Let's train the tokenizer now:

```
# initialize the WordPiece tokenizer
tokenizer = BertWordPieceTokenizer()
# train the tokenizer
tokenizer.train(files=files, vocab_size=vocab_size,
special_tokens=special_tokens)
# enable truncation up to the maximum 512 tokens
tokenizer.enable_truncation(max_length=max_length)
```

Since this is BERT, the default tokenizer is [WordPiece](). As a result, we initialize the `BertWordPieceTokenizer()` tokenizer class from the `tokenizers` library and use the `train()` method to train it, it will take several minutes to finish. Let's save it now:

```
model_path = "pretrained-bert"
```

```
# make the directory if not already there
if not os.path.isdir(model_path):
  os.mkdir(model_path)
# save the tokenizer
tokenizer.save_model(model_path)
# dumping some of the tokenizer config to config file,
# including special tokens, whether to lower case and the maximum sequence
```

```
length
with open(os.path.join(model_path, "config.json"), "w") as f:
  tokenizer_cfg = {
      "do_lower_case": True,
      "unk_token": "[UNK]",
      "sep_token": "[SEP]",
      "pad_token": "[PAD]",
      "cls_token": "[CLS]",
      "mask_token": "[MASK]",
      "model_max_length": max_length,
      "max_len": max_length,
  }
  json.dump(tokenizer_cfg, f)
```

The `tokenizer.save_model()` method saves the vocabulary file into that path, we also manually save some tokenizer configurations, such as special tokens:

- `unk_token`: A special token that represents an out-of-vocabulary token, even though the tokenizer is a WordPiece tokenizer, the `unk` tokens are not impossible, but rare.
- `sep_token`: A special token that separates two different sentences in the same input.
- `pad_token`: A special token that is used to fill sentences that do not reach the maximum sequence length (since the arrays of tokens must be the same size).
- `cls_token`: A special token representing the class of the input.
- `mask_token`: This is the mask token we use for the Masked Language Modeling (MLM) pretraining task.

After the training of the tokenizer is completed, let's load it now:

```
# when the tokenizer is trained and configured, load it as BertTokenizerFast
```

```
tokenizer = BertTokenizerFast.from_pretrained(model_path)
```

Of course, if you want to use the tokenizer multiple times, you don't have to train it again, simply load it using the above cell.

Tokenizing the Dataset

Now that we have the tokenizer ready, the below code is responsible for tokenizing the dataset:

```
def encode_with_truncation(examples):
  """Mapping function to tokenize the sentences passed with truncation"""
  return tokenizer(examples["text"], truncation=True, padding="max_length",
                   max_length=max_length, return_special_tokens_mask=True)

def encode_without_truncation(examples):
  """Mapping function to tokenize the sentences passed without truncation"""
  return tokenizer(examples["text"], return_special_tokens_mask=True)

# the encode function will depend on the truncate_longer_samples variable
encode = encode_with_truncation if truncate_longer_samples else encode_without_truncation
# tokenizing the train dataset
train_dataset = d["train"].map(encode, batched=True)
# tokenizing the testing dataset
test_dataset = d["test"].map(encode, batched=True)
if truncate_longer_samples:
  # remove other columns and set input_ids and attention_mask as PyTorch tensors
  train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
```

```
  test_dataset.set_format(type="torch", columns=["input_ids",
"attention_mask"])
else:
  # remove other columns, and remain them as Python lists
  test_dataset.set_format(columns=["input_ids", "attention_mask",
"special_tokens_mask"])
  train_dataset.set_format(columns=["input_ids", "attention_mask",
"special_tokens_mask"])
```

The `encode()` callback that we use to tokenize our dataset depends on the `truncate_longer_samples` boolean variable. If set to `True`, then we truncate sentences that exceed the maximum sequence length (`max_length` parameter). Otherwise, we don't.

Next, in the case of setting `truncate_longer_samples` to `False`, we need to join our untruncated samples together and cut them into fixed-size vectors since the model expects a fixed-sized sequence during training:

```
from itertools import chain
# Main data processing function that will concatenate all texts from our
dataset and generate chunks of
# max_seq_length.
# grabbed from:
https://github.com/huggingface/transformers/blob/main/examples/pytorch/lang
modeling/run_mlm.py
def group_texts(examples):
   # Concatenate all texts.
   concatenated_examples = {k: list(chain(*examples[k])) for k in
examples.keys()}
   total_length = len(concatenated_examples[list(examples.keys())[0]])
   # We drop the small remainder, we could add padding if the model
supported it instead of this drop, you can
```

```
    # customize this part to your needs.
    if total_length >= max_length:
        total_length = (total_length // max_length) * max_length
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + max_length] for i in range(0, total_length, max_length)]
        for k, t in concatenated_examples.items()
    }
    return result

# Note that with `batched=True`, this map processes 1,000 texts together, so group_texts throws away a
# remainder for each of those groups of 1,000 texts. You can adjust that batch_size here but a higher value
# might be slower to preprocess.
#
# To speed up this part, we use multiprocessing. See the documentation of the map method for more information:
# https://huggingface.co/docs/datasets/package_reference/main_classes.html#da
if not truncate_longer_samples:
  train_dataset = train_dataset.map(group_texts, batched=True,
                                    desc=f"Grouping texts in chunks of {max_length}")
  test_dataset = test_dataset.map(group_texts, batched=True,
                                  desc=f"Grouping texts in chunks of {max_length}")
  # convert them from lists to torch tensors
  train_dataset.set_format("torch")
  test_dataset.set_format("torch")
```

Most of the above code was brought from the `run_mlm.py` script from the huggingface transformers examples, so this is actually used by the library itself.

If you don't want to concatenate all texts and then split them into chunks of 512 tokens, then make sure you set `truncate_longer_samples` to `True`, so it will treat each line as an individual sample regardless of its length. If you set `truncate_longer_samples` to `True`, the above code cell won't be executed at all.

```
len(train_dataset), len(test_dataset)
```

Output:

```
(643843, 71357)
```

Loading the Model

For this tutorial, we're picking BERT, but feel free to pick any of the transformer models supported by huggingface transformers library, such as `RobertaForMaskedLM` or `DistilBertForMaskedLM`:

```
# initialize the model with the config
model_config = BertConfig(vocab_size=vocab_size, max_position_embeddings=max_length)
model = BertForMaskedLM(config=model_config)
```

We initialize the model config using `BertConfig`, and pass the vocabulary size as well as the maximum sequence length. We then pass the config to `BertForMaskedLM` to initialize the model itself.

Training

Before we start pre-training our model, we need a way to randomly mask tokens in our dataset for the **Masked Language Model (MLM)** task. Luckily, the library makes this easy for us by simply constructing a `DataCollatorForLanguageModeling` object:

```
# initialize the data collator, randomly masking 20% (default is 15%) of the tokens for the Masked Language
# Modeling (MLM) task
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.2
)
```

We pass the `tokenizer` and set `mlm` to `True`, and also set the `mlm_probability` to 0.2 to randomly replace each token with `[MASK]` token by 20% probability.

Next, let's initialize our training arguments:

```
training_args = TrainingArguments(
    output_dir=model_path,          # output directory to where save model checkpoint
    evaluation_strategy="steps",    # evaluate each `logging_steps` steps
    overwrite_output_dir=True,
    num_train_epochs=10,            # number of training epochs, feel free to tweak
    per_device_train_batch_size=10, # the training batch size, put it as high as your GPU memory fits
    gradient_accumulation_steps=8,  # accumulating the gradients before updating the weights
    per_device_eval_batch_size=64,  # evaluation batch size
    logging_steps=1000,             # evaluate, log and save model checkpoints every 1000 step
    save_steps=1000,
    # load_best_model_at_end=True,  # whether to load the best model (in
```

```
terms of loss) at the end of training
    # save_total_limit=3,           # whether you don't have much space so you let only 3 model weights saved in the disk
)
```

Each argument is explained in the comments, refer to the `TrainingArguments` docs for more details. Let's make our trainer now:

```
# initialize the trainer and pass everything to it
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)
```

We pass our training arguments to the `Trainer`, as well as the model, data collator, and the training sets. We simply call `train()` now to start training:

```
# train the model
trainer.train()
```

```
[10135/79670 18:53:08 < 129:35:53, 0.15 it/s, Epoch 1.27/10]
Step    Training Loss   Validation Loss
1000    6.904000        6.558231
2000    6.498800        6.401168
3000    6.362600        6.277831
4000    6.251000        6.172856
```

5000	6.155800	6.071129
6000	6.052800	5.942584
7000	5.834900	5.546123
8000	5.537200	5.248503
9000	5.272700	4.934949
10000	4.915900	4.549236

The training will take several hours to several days, depending on the dataset size, training batch size (i.e increase it as much as your GPU memory fits), and GPU speed.

As you can see in the output, the model is still improving and the validation loss is still decreasing. You usually have to cancel the training once the validation loss stops decreasing or decreasing very slowly.

Since we have set `logging_steps` and `save_steps` to 1000, then the trainer will evaluate and save the model after every 1000 steps (i.e trained on steps x `gradient_accumulation_step` x `per_device_train_size` = 1000x8x10 = 80,000 samples). As a result, I have canceled the training after about 19 hours of training, or 10000 steps (that is about 1.27 epochs, or trained on 800,000 samples), and started to use the model. In the next section, we'll see how we can use the model for inference.

Using the Model

Before we use the model, let's assume we don't have `model` and `tokenizer` variables in the current runtime. Therefore, we need to load them again:

```
# load the model checkpoint
model = BertForMaskedLM.from_pretrained(os.path.join(model_path, "checkpoint-10000"))
# load the tokenizer
```

```
tokenizer = BertTokenizerFast.from_pretrained(model_path)
```

If you're on Google Colab, then you have to save your checkpoints in Google Drive for later use, you can do that by setting `model_path` to a drive path instead of a local path like we did here, just make sure you have enough space there.

Alternatively, you can push your model and tokenizer into the huggingface hub, check this [useful guide] to do it.

Let's use our model now:

```
fill_mask = pipeline("fill-mask", model=model, tokenizer=tokenizer)
```

We use the simple [pipeline API], and pass both the `model` and the `tokenizer`. Let's predict some examples:

```
# perform predictions
examples = [
  "Today's most trending hashtags on [MASK] is Donald Trump",
  "The [MASK] was cloudy yesterday, but today it's rainy.",
]
for example in examples:
  for prediction in fill_mask(example):
    print(f"{prediction['sequence']}, confidence: {prediction['score']}")
  print("="*50)
```

Output:

```
today's most trending hashtags on twitter is donald trump, confidence: 0.1027069091796875
```

```
today's most trending hashtags on monday is donald trump, confidence: 0.09271949529647827
today's most trending hashtags on tuesday is donald trump, confidence: 0.08099588006734848
today's most trending hashtags on facebook is donald trump, confidence: 0.04266013577580452
today's most trending hashtags on wednesday is donald trump, confidence: 0.04120611026883125
==================================================
the weather was cloudy yesterday, but today it's rainy., confidence: 0.04445931687951088
the day was cloudy yesterday, but today it's rainy., confidence: 0.037249673157930374
the morning was cloudy yesterday, but today it's rainy., confidence: 0.023775646463036537
the weekend was cloudy yesterday, but today it's rainy., confidence: 0.022554103285074234
the storm was cloudy yesterday, but today it's rainy., confidence: 0.019406016916036606
==================================================
```

That's impressive, I have canceled the training and the model is still producing interesting results! If your model does not make good predictions, then that's a good indicator that it wasn't trained enough.

Conclusion

And there you have a complete code for pretraining BERT or other transformers using Huggingface libraries, below are some tips:

- As mentioned above, the training speed will depend on the GPU speed, the number of samples in the dataset, and batch size. I have set the training batch size to 10, as that's the maximum it can fit my GPU memory on Colab. If you have more memory, make sure to increase it so you increase the training speed significantly.

- During training, if you see the validation loss starts to increase, make sure to remember the checkpoint where the lowest validation loss occurs so you can load that checkpoint later for use. You can also set `load_best_model_at_end` to `True` if you don't want to keep track of the loss, as it will load the best weights in terms of loss when the training ends.
- The vocabulary size was chosen based on the original BERT configuration, as it had the size of 30,522, feel free to increase it if you feel the language of your dataset has a large vocabulary, or you can experiment with this.
- If you set `truncate_longer_samples` to `False`, then the code assumes you have larger text on one sentence (i.e line), you will notice that it takes much longer to process, especially if you set a large `batch_size` on the `map()` method. If it takes a lot of hours to process, then you can either set `truncate_longer_samples` to `True` so you truncate sentences that exceed `max_length` tokens or you can save the dataset after processing using the `save_to_disk()` method, so you process it once and load it several times.
- In a newer version of the transformers library, there is a new parameter called `auto_find_batch_size` in the `TrainingArguments()` class, you can pass it as `True` so it'll find the optimal batch size for your GPU, avoiding Out-of-Memory errors. Make sure you have `accelerate` library installed: `pip install accelerate`.

If you're interested in fine-tuning BERT for a downstream task such as text classification, then this tutorial guides you through it.

SourceCode:

pretraining_bert.py

```
# -*- coding: utf-8 -*-
"""PretrainingBERT_PythonCodeTutorial.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1An1VNpKKMRVrwcdQQNSe7Omh_fl2Gj-2
"""

!pip install datasets transformers==4.18.0 sentencepiece

from datasets import *
from transformers import *
from tokenizers import *
import os
import json

# download and prepare cc_news dataset
dataset = load_dataset("cc_news", split="train")

# split the dataset into training (90%) and testing (10%)
d = dataset.train_test_split(test_size=0.1)
d["train"], d["test"]

for t in d["train"]["text"][:3]:
  print(t)
  print("="*50)

# if you have your custom dataset
# dataset = LineByLineTextDataset(
#     tokenizer=tokenizer,
#     file_path="path/to/data.txt",
#     block_size=64,
# )

# or if you have huge custom dataset separated into files
# load the splitted files
# files = ["train1.txt", "train2.txt"] # train3.txt, etc.
```

```python
# dataset = load_dataset("text", data_files=files, split="train")

# if you want to train the tokenizer from scratch (especially if you have custom
# dataset loaded as datasets object), then run this cell to save it as files
# but if you already have your custom data as text files, there is no point using this
def dataset_to_text(dataset, output_filename="data.txt"):
  """Utility function to save dataset text to disk,
  useful for using the texts to train the tokenizer
  (as the tokenizer accepts files)"""
  with open(output_filename, "w") as f:
    for t in dataset["text"]:
      print(t, file=f)

# save the training set to train.txt
dataset_to_text(d["train"], "train.txt")
# save the testing set to test.txt
dataset_to_text(d["test"], "test.txt")

special_tokens = [
  "[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]", "<S>", "<T>"
]
# if you want to train the tokenizer on both sets
# files = ["train.txt", "test.txt"]
# training the tokenizer on the training set
files = ["train.txt"]
# 30,522 vocab is BERT's default vocab size, feel free to tweak
vocab_size = 30_522
# maximum sequence length, lowering will result to faster training (when increasing batch size)
max_length = 512
# whether to truncate
truncate_longer_samples = False

# initialize the WordPiece tokenizer
tokenizer = BertWordPieceTokenizer()
# train the tokenizer
tokenizer.train(files=files, vocab_size=vocab_size, special_tokens=special_tokens)
# enable truncation up to the maximum 512 tokens
tokenizer.enable_truncation(max_length=max_length)
```

```python

model_path = "pretrained-bert"
# make the directory if not already there
if not os.path.isdir(model_path):
  os.mkdir(model_path)

# save the tokenizer
tokenizer.save_model(model_path)

# dumping some of the tokenizer config to config file,
# including special tokens, whether to lower case and the maximum sequence length
with open(os.path.join(model_path, "config.json"), "w") as f:
  tokenizer_cfg = {
      "do_lower_case": True,
      "unk_token": "[UNK]",
      "sep_token": "[SEP]",
      "pad_token": "[PAD]",
      "cls_token": "[CLS]",
      "mask_token": "[MASK]",
      "model_max_length": max_length,
      "max_len": max_length,
  }
  json.dump(tokenizer_cfg, f)

# when the tokenizer is trained and configured, load it as BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained(model_path)

def encode_with_truncation(examples):
  """Mapping function to tokenize the sentences passed with truncation"""
  return tokenizer(examples["text"], truncation=True, padding="max_length",
                   max_length=max_length, return_special_tokens_mask=True)

def encode_without_truncation(examples):
  """Mapping function to tokenize the sentences passed without truncation"""
  return tokenizer(examples["text"], return_special_tokens_mask=True)

# the encode function will depend on the truncate_longer_samples variable
encode = encode_with_truncation if truncate_longer_samples else encode_without_truncation
```

```python

# tokenizing the train dataset
train_dataset = d["train"].map(encode, batched=True)
# tokenizing the testing dataset
test_dataset = d["test"].map(encode, batched=True)

if truncate_longer_samples:
  # remove other columns and set input_ids and attention_mask as PyTorch tensors
  train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
  test_dataset.set_format(type="torch", columns=["input_ids", "attention_mask"])
else:
  # remove other columns, and remain them as Python lists
  test_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])
  train_dataset.set_format(columns=["input_ids", "attention_mask", "special_tokens_mask"])

from itertools import chain
# Main data processing function that will concatenate all texts from our dataset and generate chunks of
# max_seq_length.
# grabbed from: https://github.com/huggingface/transformers/blob/main/examples/pytorch/language-
modeling/run_mlm.py
def group_texts(examples):
    # Concatenate all texts.
    concatenated_examples = {k: list(chain(*examples[k])) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder, we could add padding if the model supported it instead of this drop,
you can
    # customize this part to your needs.
    if total_length >= max_length:
        total_length = (total_length // max_length) * max_length
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + max_length] for i in range(0, total_length, max_length)]
        for k, t in concatenated_examples.items()
    }
    return result

# Note that with `batched=True`, this map processes 1,000 texts together, so group_texts throws away a
# remainder for each of those groups of 1,000 texts. You can adjust that batch_size here but a higher

```

```
value
# might be slower to preprocess.
#
# To speed up this part, we use multiprocessing. See the documentation of the map method for more
information:
# https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Dataset.map
if not truncate_longer_samples:
  train_dataset = train_dataset.map(group_texts, batched=True,
                                    desc=f"Grouping texts in chunks of {max_length}")
  test_dataset = test_dataset.map(group_texts, batched=True,
                                  desc=f"Grouping texts in chunks of {max_length}")
  # convert them from lists to torch tensors
  train_dataset.set_format("torch")
  test_dataset.set_format("torch")

len(train_dataset), len(test_dataset)

# initialize the model with the config
model_config = BertConfig(vocab_size=vocab_size, max_position_embeddings=max_length)
model = BertForMaskedLM(config=model_config)

# initialize the data collator, randomly masking 20% (default is 15%) of the tokens for the Masked
Language
# Modeling (MLM) task
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.2
)

training_args = TrainingArguments(
    output_dir=model_path,          # output directory to where save model checkpoint
    evaluation_strategy="steps",    # evaluate each `logging_steps` steps
    overwrite_output_dir=True,
    num_train_epochs=10,            # number of training epochs, feel free to tweak
    per_device_train_batch_size=10, # the training batch size, put it as high as your GPU memory fits
    gradient_accumulation_steps=8,  # accumulating the gradients before updating the weights
    per_device_eval_batch_size=64,  # evaluation batch size
    logging_steps=1000,             # evaluate, log and save model checkpoints every 1000 step
    save_steps=1000,
```

```
    # load_best_model_at_end=True,  # whether to load the best model (in terms of loss) at the end of training
    # save_total_limit=3,           # whether you don't have much space so you let only 3 model weights saved in the disk
)

# initialize the trainer and pass everything to it
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)

# train the model
trainer.train()

# when you load from pretrained
model = BertForMaskedLM.from_pretrained(os.path.join(model_path, "checkpoint-6000"))
tokenizer = BertTokenizerFast.from_pretrained(model_path)
# or simply use pipeline
fill_mask = pipeline("fill-mask", model=model, tokenizer=tokenizer)

# perform predictions
example = "It is known that [MASK] is the capital of Germany"
for prediction in fill_mask(example):
  print(prediction)

# perform predictions
examples = [
  "Today's most trending hashtags on [MASK] is Donald Trump",
  "The [MASK] was cloudy yesterday, but today it's rainy.",
]
for example in examples:
  for prediction in fill_mask(example):
    print(f"{prediction['sequence']}, confidence: {prediction['score']}")
  print("="*50)
```

```
!nvidia-smi
```

CHAPTER 8: Conversational AI Chatbot with Transformers in Python

Learn how to use Huggingface transformers library to generate conversational responses with the pretrained DialoGPT model in Python.

Chatbots have gained a lot of popularity in recent years. As the interest grows in using chatbots for business, researchers also did a great job on advancing conversational AI chatbots.

In this tutorial, we'll use the Huggingface transformers library to employ the pre-trained DialoGPT model for conversational response generation.

DialoGPT is a large-scale tunable neural conversational response generation model trained on 147M conversations extracted from Reddit. The good thing is that you can fine-tune it with your dataset to achieve better performance than training from scratch.

This tutorial is about text generation in chatbots and not regular text. If you want open-ended generation, see this tutorial where I show you how to use GPT-2 and GPT-J models to generate impressive text.

Alright, to get started, let's install transformers:

```
$ pip3 install transformers
```

Open up a new Python file or notebook and do the following:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch
```

```python

# model_name = "microsoft/DialoGPT-large"
model_name = "microsoft/DialoGPT-medium"
# model_name = "microsoft/DialoGPT-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
```

There are three versions of DialoGPT; small, medium, and large. Of course, the larger, the better, but if you run this on your machine, I think small or medium fits your memory with no problems. I tried loading the large model, which takes about 5GB of my RAM. You can also use Google Colab to try out the large one.

Generating Responses with Greedy Search

In this section, we'll be using the [greedy search algorithm] to generate responses. We select the chatbot response with the highest probability of choosing on each time step.

Let's make code for chatting with our AI using greedy search:

```python
# chatting 5 times with greedy search
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token,
return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0
else input_ids
    # generate a bot response
```

```
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        pad_token_id=tokenizer.eos_token_id,
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
```

Let's explain the core of this code:

- We first take input from the user for chatting.
- We encode the text to `input_ids` using the DialoGPT tokenizer, we also append the end of the string token and return it as a Pytorch tensor.
- If this is the first time chatting with the bot, we directly feed `input_ids` to our model for a generation. Otherwise, we append the chat history using concatenation with the help of `torch.cat()` method.
- After that, we use the `model.generate()` method for generating the chatbot response.
- Lastly, as the returned output is a tokenized sequence too, we decode the sequence using `tokenizer.decode()` and set `skip_special_tokens` to `True` to make sure we don't see any annoying special tokens such as `<|endoftext|>`. Also, since the model returns the whole sequence, we skip the previous chat history and print only the newly generated chatbot answer.

Below is a sample discussion with the bot:

```
>> You:How can you be rich so quickly?
DialoGPT: I'm not rich, I'm just a rich man.
>> You:Ok, you don't make sense, how old are you ?
DialoGPT: I'm not a rich man, I'm just a rich man.
```

```
>> You:How can you be happy ?
DialoGPT: I'm not happy, I'm just a rich man.
>> You:But they say money buys happiness!
DialoGPT: Money buys happiness, but not happiness itself.
>> You:You don't make sense again!
DialoGPT: I'm not happy, I'm just a rich man.
```

You see the model repeats a lot of responses, as these are the highest probability, and it is choosing it every time.

By default, `model.generate()` uses greedy search algorithm when no other parameters are set. In the following sections, we'll be adding some arguments to this method to see if we can improve the generation.

Learn also: *How to Train BERT from Scratch using Transformers in Python.*

Generating Responses with Beam Search

Beam search allows us to reduce the risk of missing high probability sequences by keeping the most likely `num_beams` of hypotheses at each time step and then taking the sequences that have the overall highest probability, below code will generate chatbot responses with beam search:

```
# chatting 5 times with beam search
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token,
return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0
```

```python
else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        num_beams=3,
        early_stopping=True,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
```

When setting `num_beams` to `3` in `model.generate()` method, then we're going to select three words at each time step and develop them to find the highest overall probability of the sequence, setting `num_beams` to 1 is the same as greedy search.

Below is a sample discussion with the chatbot using beam search:

```
>> You:How can I be rich?
DialoGPT: You can't.
>> You:Why not?
DialoGPT: Because you can't.
>> You:Then how am I supposed to live well?
DialoGPT: You can't.
>> You:So basically, you want me to be a poor guy?
DialoGPT: No, I want you to be a rich guy.
>> You:Ok how to be rich, then?
DialoGPT: I don't know.
```

Generating Responses with Sampling

We used beam and greedy search in previous sections to generate the highest probability sequence. Now that's great for tasks such as machine translation or text summarization where the output is predictable. However, it is not the best option for an open-ended generation as in chatbots.

For a better generation, we need to introduce some randomness where we sample from a wide range of candidate sequences based on probabilities:

```
# chatting 5 times with sampling
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token,
return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0
else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_k=0,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
```

```python
    print(f"DialoGPT: {output}")
```

This time, we set `do_sample` to `True` for sampling, and we set `top_k` to `0` indicating that we're selecting all possible probabilities, we'll later discuss `top_k` parameter.

Here is a chat with these parameters:

```
>> You:Hello, how can I be rich?
DialoGPT: You can't
>> You:Really? then how others get rich?
DialoGPT: They work as money artists
>> You:How to be a money artist?
DialoGPT: I don't know, I don't have enough knowledge on artists
>> You:Alright, thank you!
DialoGPT: No problem, if you need tips pm me.
>> You:Great, you're my savior!
DialoGPT: Thank you for helping a stranger out like me!
```

There are some improvements. However, sampling on an exhaustive list of sequences with low probabilities can lead to random generation (like you see in the last sentence).

To improve it furthermore, we can:

- Lower the sampling `temperature` that helps us decrease the likelihood of picking low probability words and increase the likelihood of picking high probability words.
- Use Top-k sampling instead of picking all probable occurrences. This will help us to discard low probability words from getting picked.

```python
# chatting 5 times with Top K sampling & tweaking temperature
```

```python
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, 
return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 
else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_k=100,
        temperature=0.75,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0], 
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
```

Now, we set `top_k` to `100` to sample from the top `100` words sorted descendingly by probability. We also set `temperature` to `0.75` (default is `1.0`) to give a higher chance of picking high probability words, setting the temperature to `0.0` is the same as greedy search; setting it to infinity is the same as completely random.

Here is a discussion with these parameters:

```
>> You:How can I get rich?
DialoGPT: I'd also like to know this.
>> You:So you don't know it?
DialoGPT: I know it, but it doesn't matter.
>> You:Really? money matters, man!
DialoGPT: Well, money is money and I gotta get my money.
>> You:Yeah how to get them?
DialoGPT: Well, that's a question. I'd like to know more, just for curiosity's sake, to know what other people have said about how I should get rich.
>> You:Some robots like you told me you have to buy gold, silver or stocks.
DialoGPT: I think my wallet is empty.
```

As you can see, it is much better now; feel free to tweak `temperature` and `top_k` parameters and see if it can improve it.

Related: *[How to Paraphrase Text using Transformers in Python.]()*

Nucleus Sampling

[Nucleus sampling]() or Top-p sampling chooses from the smallest possible words whose cumulative probability exceeds the parameter `p` we set.

Below is an example using Top-p sampling:

```
# chatting 5 times with nucleus sampling & tweaking temperature
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, 
return_tensors="pt")
    # concatenate new user input with chat history (if there is)
```

```
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0
else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_p=0.95,
        top_k=0,
        temperature=0.75,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
```

We set `top_k` to `0` to disable Top-k sampling, but you can use both methods, which works better. Here is a chat:

```
>> You:How can I be rich fast?
DialoGPT: Inflation and free money
>> You:lol, but that takes time, I need fast money.
DialoGPT: Your post makes no sense. You just need a lot of money to be
rich fast
>> You:Ok how to get a lot of money?
DialoGPT: Buy gold and then trade it for money.
>> You:Hmm, I got your point, thanks.
DialoGPT: No problem. You have to have money to buy gold, but its only
good for trading it.
```

```
>> You:Yeah, I understand, thank you so much.
DialoGPT: Ok, let me know if you need help
```

Now the chatbot clearly makes sense in a lot of cases.

Now let's add some code to generate more than one chatbot response, and then we choose which response to include in the next input:

```
# chatting 5 times with nucleus & top-k sampling & tweaking temperature & multiple
# sentences
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids_list = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_p=0.95,
        top_k=50,
        temperature=0.75,
        num_return_sequences=5,
        pad_token_id=tokenizer.eos_token_id
    )
```

```
    #print the outputs
    for i in range(len(chat_history_ids_list)):
      output = tokenizer.decode(chat_history_ids_list[i][bot_input_ids.shape[-1]:], skip_special_tokens=True)
      print(f"DialoGPT {i}: {output}")
    choice_index = int(input("Choose the response you want for the next input: "))
    chat_history_ids = torch.unsqueeze(chat_history_ids_list[choice_index], dim=0)
```

I have set `num_return_sequences` to `5` to return five sentences at a time, we have to choose the one included in the following sequence. Here is how it went:

```
>> You:Hello, friend.
DialoGPT 0: Hey there!
DialoGPT 1: I am your friend, and I am also your friend.
DialoGPT 2: Hello. :
DialoGPT 3: Hiya! How are you?
DialoGPT 4: Hello. I like your username.
Choose the response you want for the next input: 3
>> You:Fine, thanks, what about you?
DialoGPT 0: Good. Not much to do.
DialoGPT 1: Not much, you?
DialoGPT 2: Fine, thanks.
DialoGPT 3: I'm okay, thank you for asking.
DialoGPT 4: I'm fine too.
Choose the response you want for the next input: 2
>> You:Can you tell me how can I get rich quickly?
DialoGPT 0: Yes, I can!
DialoGPT 1: Yes, just look up how to get rich on Google.
```

```
DialoGPT 2: Just keep doing your thing. You'll get rich.
DialoGPT 3: No, I can't.
DialoGPT 4: no, i have no money
Choose the response you want for the next input: 1
>> You:Yeah, but I thought you have an answer...
DialoGPT 0: I'm getting married in the morning.
DialoGPT 1: How do you get rich?
DialoGPT 2: I think there are many good answers, but I could send you one, if you want.
DialoGPT 3: I will, thanks!
DialoGPT 4: I thought you have an answer.
Choose the response you want for the next input: 2
>> You:Yes, please send me.
DialoGPT 0: You can send me link.
DialoGPT 1: I am very interested
DialoGPT 2: I sent you a PM
DialoGPT 3: I'll send you a PM
DialoGPT 4: I am always interested in new ideas.
Choose the response you want for the next input: 2
```

Conclusion

And there you go. I hope this tutorial helped you out on how to generate text on DialoGPT and similar models. For more information on generating text, I highly recommend you read the [How to generate text with Transformers] guide.

I'll leave you tweaking the parameters to see if you can make the bot performs better.

Also, a great and exciting challenge for you is combining this with [text-to-speech] and [speech-to-text] tutorials to [build a virtual assistant] like [Alexa], [Siri], and Cortana!

SourceCode:

dialogpt.py

```
# -*- coding: utf-8 -*-
"""DialoGPT.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1KAg6X8RFHE0KSvFSZ__w7KGZrSqT4cZ3
"""

# !pip install transformers

from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

# model_name = "microsoft/DialoGPT-large"
model_name = "microsoft/DialoGPT-medium"
# model_name = "microsoft/DialoGPT-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
print("====Greedy search chat====")
# chatting 5 times with greedy search
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
```

```
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        pad_token_id=tokenizer.eos_token_id,
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====Beam search chat====")
# chatting 5 times with beam search
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        num_beams=3,
        early_stopping=True,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====Sampling chat====")
# chatting 5 times with sampling
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
```

```
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_k=0,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====Sampling chat with tweaking temperature====")
# chatting 5 times with sampling & tweaking temperature
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_k=0,
        temperature=0.75,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====Top-K sampling chat with tweaking temperature====")
# chatting 5 times with Top K sampling & tweaking temperature
for step in range(5):
```

```python
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_k=100,
        temperature=0.75,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====Nucleus sampling (top-p) chat with tweaking temperature====")
# chatting 5 times with nucleus sampling & tweaking temperature
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_p=0.95,
        top_k=0,
        temperature=0.75,
        pad_token_id=tokenizer.eos_token_id
    )
```

```
    #print the output
    output = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
skip_special_tokens=True)
    print(f"DialoGPT: {output}")
print("====chatting 5 times with nucleus & top-k sampling & tweaking temperature & multiple
sentences====")
# chatting 5 times with nucleus & top-k sampling & tweaking temperature & multiple
# sentences
for step in range(5):
    # take user input
    text = input(">> You:")
    # encode the input and add end of string token
    input_ids = tokenizer.encode(text + tokenizer.eos_token, return_tensors="pt")
    # concatenate new user input with chat history (if there is)
    bot_input_ids = torch.cat([chat_history_ids, input_ids], dim=-1) if step > 0 else input_ids
    # generate a bot response
    chat_history_ids_list = model.generate(
        bot_input_ids,
        max_length=1000,
        do_sample=True,
        top_p=0.95,
        top_k=50,
        temperature=0.75,
        num_return_sequences=5,
        pad_token_id=tokenizer.eos_token_id
    )
    #print the outputs
    for i in range(len(chat_history_ids_list)):
      output = tokenizer.decode(chat_history_ids_list[i][bot_input_ids.shape[-1]:],
skip_special_tokens=True)
      print(f"DialoGPT {i}: {output}")
    choice_index = int(input("Choose the response you want for the next input: "))
    chat_history_ids = torch.unsqueeze(chat_history_ids_list[choice_index], dim=0)
```

CHAPTER 9: Fine Tune BERT for Text Classification using Transformers in Python

Learn how to use HuggingFace transformers library to fine tune BERT and other transformer models for text classification task in Python.

Transformer models have been showing incredible results in most of the tasks in the natural language processing field. The power of transfer learning combined with large-scale transformer language models has become a standard in state-of-the-art NLP.

One of the most significant milestones in the evolution of NLP is the release of Google's BERT model in late 2018, which is known as the beginning of a new era in NLP.

In this tutorial, we will take you through an example of fine-tuning BERT (and other transformer models) for text classification using the Huggingface Transformers library on the dataset of your choice.

Please note that this tutorial is about fine-tuning the BERT model on a downstream task (such as text classification). If you want to train BERT from scratch, that's called pre-training; this tutorial will definitely help you.

We'll be using 20 newsgroups dataset as a demo for this tutorial; it is a dataset that has about 18,000 news posts on 20 different topics. If you have a custom dataset for classification, you can follow along as well, as you should make very few changes. For example, I've implemented this tutorial on fake news detection, and it works great.

To get started, let's install Huggingface transformers library along with others:

```
pip3 install transformers numpy torch sklearn
```

Open up a new notebook/Python file and import the necessary modules:

```
import torch
from transformers.file_utils import is_tf_available, is_torch_available,
is_torch_tpu_available
from transformers import BertTokenizerFast, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
import numpy as np
import random
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
```

Next, let's make a function to set seed so we'll have the same results in different runs:

```
def set_seed(seed: int):
    """
    Helper function for reproducible behavior to set the seed in ``random``,
``numpy``, ``torch`` and/or ``tf`` (if
    installed).

    Args:
        seed (:obj:`int`): The seed to set.
    """
    random.seed(seed)
    np.random.seed(seed)
    if is_torch_available():
        torch.manual_seed(seed)
```

```
        torch.cuda.manual_seed_all(seed)
        # ^^ safe to call this function even if cuda is not available
    if is_tf_available():
        import tensorflow as tf

        tf.random.set_seed(seed)

set_seed(1)
```

As mentioned earlier, we'll be using the BERT model. More specifically, we'll be using `bert-base-uncased` pre-trained weights from the library. Again, if you wish to pre-train using your large dataset, this tutorial should help you do that.

Also, we'll be using `max_length` of 512:

```
# the model we gonna train, base uncased BERT
# check text classification models here: https://huggingface.co/models?filter=text-classification
model_name = "bert-base-uncased"
# max sequence length for each document/sentence sample
max_length = 512
```

`max_length` is the maximum length of our sequence. In other words, we'll be picking only the first 512 tokens from each document or post, and you can always change it to whatever you want. However, if you increase it, make sure it fits your memory during the training, even when using a smaller batch size.

Learn also: Conversational AI Chatbot with Transformers in Python.

Loading the Dataset

Next, let's download and load the tokenizer responsible for converting our text to sequences of tokens:

```
# load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained(model_name, do_lower_case=True)
```

We also set `do_lower_case` to `True` to make sure we lowercase all the text (remember, we're using the uncased model).

The below code downloads and loads the dataset:

```
def read_20newsgroups(test_size=0.2):
  # download & load 20newsgroups dataset from sklearn's repos
  dataset = fetch_20newsgroups(subset="all", shuffle=True, remove=("headers", "footers", "quotes"))
  documents = dataset.data
  labels = dataset.target
  # split into training & testing a return data as well as label names
  return train_test_split(documents, labels, test_size=test_size), dataset.target_names

# call the function
(train_texts, valid_texts, train_labels, valid_labels), target_names = read_20newsgroups()
```

Each of `train_texts` and `valid_texts` is a list of documents (list of strings) for training and validation sets, respectively, the same for `train_labels` and `valid_labels`, each of them is a list of integers or labels ranging from 0 to 19. `target_names` is a list of our 20 labels each has its own name.

Now let's use our tokenizer to encode our corpus:

```
# tokenize the dataset, truncate when passed `max_length`,
# and pad with 0's when less than `max_length`
train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=max_length)
valid_encodings = tokenizer(valid_texts, truncation=True, padding=True, max_length=max_length)
```

We set `truncation` to `True` so that we eliminate tokens that go above `max_length`, we also set `padding` to `True` to pad documents that are less than `max_length` with empty tokens.

The below code wraps our tokenized text data into a torch `Dataset`:

```
class NewsGroupsDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {k: torch.tensor(v[idx]) for k, v in self.encodings.items()}
        item["labels"] = torch.tensor([self.labels[idx]])
        return item

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

# convert our tokenized data into a torch Dataset
```

```
train_dataset = NewsGroupsDataset(train_encodings, train_labels)
valid_dataset = NewsGroupsDataset(valid_encodings, valid_labels)
```

Since we gonna use `Trainer` from Transformers library, it expects our dataset as a [`torch.utils.data.Dataset`](), so we made a simple class that implements the `__len__()` method that returns the number of samples, and `__getitem__()` method to return a data sample at a specific index.

Training the Model

Now that we have our data prepared, let's download and load our BERT model and its pre-trained weights:

```
# load the model and pass to CUDA
model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=len(target_names)).to("cuda")
```

We're using `BertForSequenceClassification` class from Transformers library, we set `num_labels` to the length of our available labels, in this case, 20.

We also cast our model to our CUDA GPU. If you're on CPU (not suggested), then just delete `to()` method.

Before we start fine-tuning our model, let's make a simple function to compute the metrics we want. In this case, accuracy:

```
from sklearn.metrics import accuracy_score

def compute_metrics(pred):
  labels = pred.label_ids
  preds = pred.predictions.argmax(-1)
  # calculate accuracy using sklearn's function
```

```
  acc = accuracy_score(labels, preds)
  return {
      'accuracy': acc,
  }
```

You're free to include any metric you want, I've included accuracy, but you can add precision, recall, etc.

The below code uses `TrainingArguments` class to specify our training arguments, such as the number of epochs, batch size, and some other parameters:

```
training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=8,  # batch size per device during training
    per_device_eval_batch_size=20,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate
scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    load_best_model_at_end=True,     # load the best model when finished
training (default metric is loss)
    # but you can specify `metric_for_best_model` argument to change to
accuracy or other metric
    logging_steps=400,               # log & save weights each logging_steps
    save_steps=400,
    evaluation_strategy="steps",     # evaluate each `logging_steps`
)
```

Each argument is explained in the code comments. I've specified 8 as training batch size; that's because it's the maximum I can get to fit in a [Google Colab environment](#)'s memory. If you have the CUDA out of memory error, make sure to decrease it furthermore. If you have a more powerful GPU in your environment, then increasing it will make the training significantly faster.

You can also tweak other parameters, such as increasing the number of epochs for better training.

I've set the `logging_steps` and `save_steps` to 400, which means it will evaluate and save the model after every 400 steps, make sure to increase it when you decrease the batch size lower than 8, that's because it'll save a lot of checkpoints after every few steps, and may take your whole environment disk space.

We then pass our training arguments, dataset, and `compute_metrics` callback to our `Trainer`:

```
trainer = Trainer(
    model=model,                         # the instantiated Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=valid_dataset,          # evaluation dataset
    compute_metrics=compute_metrics,     # the callback that computes metrics of interest
)
```

Training the model:

```
# train the model
trainer.train()
```

This will take several minutes/hours depending on your environment, here's my output on Google Colab:

```
***** Running training *****
  Num examples = 15076
  Num Epochs = 3
  Instantaneous batch size per device = 8
  Total train batch size (w. parallel, distributed & accumulation) = 8
  Gradient Accumulation steps = 1
  Total optimization steps = 5655
 [5655/5655 4:12:58, Epoch 3/3]
```

Step	Training Loss	Validation Loss	Accuracy
400	1.873200	1.402060	0.606631
800	1.244500	1.086071	0.680106
1200	1.104400	1.077154	0.670557
1600	0.996500	0.955149	0.709284
2000	0.811500	0.921275	0.729708
2400	0.799500	0.916478	0.731034
2800	0.683500	0.881080	0.747480
3200	0.689100	0.861604	0.754642
3600	0.612900	0.873568	0.757294
4000	0.419000	0.895653	0.767639
4400	0.420800	0.911871	0.773210
4800	0.457800	0.900206	0.769496
5200	0.360400	0.893304	0.778780
5600	0.334600	0.888858	0.778249

As you can see, the validation loss is gradually decreasing, and the accuracy increased to over 77.8%.

Remember we set `load_best_model_at_end` to `True`, this will automatically load the best-performed model when finished training, let's make sure with `evaluate()` method:

```
# evaluate the current model after training
trainer.evaluate()
```

This will take several seconds to output something like this:

```
{'epoch': 3.0,
 'eval_accuracy': 0.7758620689655172,
 'eval_loss': 0.80070960521698}
```

Now that we trained our model, let's save it for inference later:

```
# saving the fine tuned model & tokenizer
model_path = "20newsgroups-bert-base-uncased"
model.save_pretrained(model_path)
tokenizer.save_pretrained(model_path)
```

Performing Inference

Now we have a trained model on our dataset, let's try to have some fun with it!

The below function takes a text as a string, tokenizes it with our tokenizer, calculates the output probabilities using softmax function, and returns the actual label:

```
def get_prediction(text):
    # prepare our text into tokenized sequence
```

```
    inputs = tokenizer(text, padding=True, truncation=True,
max_length=max_length, return_tensors="pt").to("cuda")
    # perform inference to our model
    outputs = model(**inputs)
    # get output probabilities by doing softmax
    probs = outputs[0].softmax(1)
    # executing argmax function to get the candidate label
    return target_names[probs.argmax()]
```

Here's an example:

```
# Example #1
text = """
The first thing is first.
If you purchase a Macbook, you should not encounter performance issues that will prevent you from learning to code efficiently.
However, in the off chance that you have to deal with a slow computer, you will need to make some adjustments.
Having too many background apps running in the background is one of the most common causes.
The same can be said about a lack of drive storage.
For that, it helps if you uninstall xcode and other unnecessary applications, as well as temporary system junk like caches and old backups.
"""
print(get_prediction(text))
```

Output:

```
comp.sys.mac.hardware
```

As expected, we're talking about Macbooks. Here's a second example:

```
# Example #2
text = """
A black hole is a place in space where gravity pulls so much that even light can not get out.
The gravity is so strong because matter has been squeezed into a tiny space. This can happen when a star is dying.
Because no light can get out, people can't see black holes.
They are invisible. Space telescopes with special tools can help find black holes.
The special tools can see how stars that are very close to black holes act differently than other stars.
"""
print(get_prediction(text))
```

Output:

```
sci.space
```

This is a label of science -> space, as expected!

Yet another example:

```
# Example #3
text = """
Coronavirus disease (COVID-19) is an infectious disease caused by a newly discovered coronavirus.
Most people infected with the COVID-19 virus will experience mild to
```

```
moderate respiratory illness and recover without requiring special treatment.
Older people, and those with underlying medical problems like cardiovascular disease, diabetes, chronic respiratory disease, and cancer are more likely to develop serious illness.
"""
print(get_prediction(text))
```

Output:

```
sci.med
```

Conclusion

In this tutorial, you've learned how you can train the BERT model using Huggingface Transformers library on your dataset.

Note that, you can also use other transformer models, such as GPT-2 with `GPT2ForSequenceClassification`, RoBERTa with `GPT2ForSequenceClassification`, Di and much more. Please head to the official documentation for a list of available models.

Also, if your dataset is in a language other than English, make sure you pick the weights for your language, this will help a lot during training. Check this link and use the filter to get the model weights you need.

SourceCode:

train.py

```
# !pip install transformers

```

```
import torch
from transformers.file_utils import is_tf_available, is_torch_available, is_torch_tpu_available
from transformers import BertTokenizerFast, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
import numpy as np
import random
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


def set_seed(seed: int):
    """
    Helper function for reproducible behavior to set the seed in ``random``, ``numpy``, ``torch`` and/or ``tf`` (if
    installed).

    Args:
        seed (:obj:`int`): The seed to set.
    """
    random.seed(seed)
    np.random.seed(seed)
    if is_torch_available():
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        # safe to call this function even if cuda is not available
    if is_tf_available():
        import tensorflow as tf
        tf.random.set_seed(seed)

set_seed(1)

# the model we gonna train, base uncased BERT
# check text classification models here: https://huggingface.co/models?filter=text-classification
model_name = "bert-base-uncased"
# max sequence length for each document/sentence sample
max_length = 512
# load the tokenizer
```

```
tokenizer = BertTokenizerFast.from_pretrained(model_name, do_lower_case=True)

def read_20newsgroups(test_size=0.2):
    # download & load 20newsgroups dataset from sklearn's repos
    dataset = fetch_20newsgroups(subset="all", shuffle=True, remove=("headers", "footers", "quotes"))
    documents = dataset.data
    labels = dataset.target
    # split into training & testing a return data as well as label names
    return train_test_split(documents, labels, test_size=test_size), dataset.target_names

# call the function
(train_texts, valid_texts, train_labels, valid_labels), target_names = read_20newsgroups()
# tokenize the dataset, truncate when passed `max_length`,
# and pad with 0's when less than `max_length`
train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=max_length)
valid_encodings = tokenizer(valid_texts, truncation=True, padding=True, max_length=max_length)

class NewsGroupsDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {k: torch.tensor(v[idx]) for k, v in self.encodings.items()}
        item["labels"] = torch.tensor([self.labels[idx]])
        return item

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

# convert our tokenized data into a torch Dataset
train_dataset = NewsGroupsDataset(train_encodings, train_labels)
valid_dataset = NewsGroupsDataset(valid_encodings, valid_labels)
# load the model and pass to CUDA
model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=len(target_names)).to("cuda")

def compute_metrics(pred):
```

```
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    # calculate accuracy using sklearn's function
    acc = accuracy_score(labels, preds)
    return {
      'accuracy': acc,
    }

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=8,  # batch size per device during training
    per_device_eval_batch_size=20,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    load_best_model_at_end=True,     # load the best model when finished training (default metric is
loss)
    # but you can specify `metric_for_best_model` argument to change to accuracy or other metric
    logging_steps=200,               # log & save weights each logging_steps
    save_steps=200,
    evaluation_strategy="steps",     # evaluate each `logging_steps`
)

trainer = Trainer(
    model=model,                         # the instantiated Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=valid_dataset,          # evaluation dataset
    compute_metrics=compute_metrics,     # the callback that computes metrics of interest
)
# train the model
trainer.train()
# evaluate the current model after training
trainer.evaluate()
# saving the fine tuned model & tokenizer
model_path = "20newsgroups-bert-base-uncased"
model.save_pretrained(model_path)
```

```
tokenizer.save_pretrained(model_path)
```

inference.py

```
from transformers import BertForSequenceClassification, BertTokenizerFast
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_20newsgroups

model_path = "20newsgroups-bert-base-uncased"
max_length = 512

def read_20newsgroups(test_size=0.2):
  dataset = fetch_20newsgroups(subset="all", shuffle=True, remove=("headers", "footers", "quotes"))
  documents = dataset.data
  labels = dataset.target
  return train_test_split(documents, labels, test_size=test_size), dataset.target_names


(train_texts, valid_texts, train_labels, valid_labels), target_names = read_20newsgroups()

model = BertForSequenceClassification.from_pretrained(model_path,
num_labels=len(target_names)).to("cuda")
tokenizer = BertTokenizerFast.from_pretrained(model_path)

def get_prediction(text):
    # prepare our text into tokenized sequence
    inputs = tokenizer(text, padding=True, truncation=True, max_length=max_length,
return_tensors="pt").to("cuda")
    # perform inference to our model
    outputs = model(**inputs)
    # get output probabilities by doing softmax
    probs = outputs[0].softmax(1)
    # executing argmax function to get the candidate label
    return target_names[probs.argmax()]

# Example #1
text = """With the pace of smartphone evolution moving so fast, there's always something waiting in
```

```
the wings.
No sooner have you spied the latest handset, that there's anticipation for the next big thing.
Here we look at those phones that haven't yet launched, the upcoming phones for 2021.
We'll be updating this list on a regular basis, with those device rumours we think are credible and exciting."""
print(get_prediction(text))
# Example #2
text = """
A black hole is a place in space where gravity pulls so much that even light can not get out.
The gravity is so strong because matter has been squeezed into a tiny space. This can happen when a star is dying.
Because no light can get out, people can't see black holes.
They are invisible. Space telescopes with special tools can help find black holes.
The special tools can see how stars that are very close to black holes act differently than other stars.
"""
print(get_prediction(text))
```

CHAPTER 10: Perform Text Summarization using Transformers in Python

Learn how to use Huggingface transformers and PyTorch libraries to summarize long text, using pipeline API and T5 transformer model in Python.

Text summarization is the task of shortening long pieces of text into a concise summary that preserves key information content and overall meaning.

There are two different approaches that are widely used for text summarization:

- **Extractive Summarization**: This is where the model identifies the meaningful sentences and phrases from the original text and only outputs those.
- **Abstractive Summarization**: The model produces an entirely different text shorter than the original. It generates new sentences in a new form, just like humans do. In this tutorial, we will use transformers for this approach.

This tutorial will use HuggingFace's transformers library in Python to perform abstractive text summarization on any text we want.

We chose HuggingFace's Transformers because it provides us with thousands of pre-trained models not just for text summarization but for a wide variety of NLP tasks, such as text classification, text paraphrasing, question answering machine translation, text generation, chatbot, and more.

To get started, let's install the required libraries:

```
pip3 install transformers torch sentencepiece
```

Using pipeline API

The most straightforward way to use models in transformers is using the pipeline API:

```
from transformers import pipeline

# using pipeline API for summarization task
summarization = pipeline("summarization")
original_text = """
Paul Walker is hardly the first actor to die during a production.
But Walker's death in November 2013 at the age of 40 after a car crash was especially eerie given his rise to fame in the "Fast and Furious" film franchise.
The release of "Furious 7" on Friday offers the opportunity for fans to remember -- and possibly grieve again -- the man that so many have praised as one of the nicest guys in Hollywood.
"He was a person of humility, integrity, and compassion," military veteran Kyle Upham said in an email to CNN.
Walker secretly paid for the engagement ring Upham shopped for with his bride.
"We didn't know him personally but this was apparent in the short time we spent with him.
I know that we will never forget him and he will always be someone very special to us," said Upham.
The actor was on break from filming "Furious 7" at the time of the fiery accident, which also claimed the life of the car's driver, Roger Rodas.
Producers said early on that they would not kill off Walker's character, Brian O'Connor, a former cop turned road racer. Instead, the script was rewritten and special effects were used to finish scenes, with Walker's brothers, Cody and Caleb, serving as body doubles.
There are scenes that will resonate with the audience -- including the ending, in which the filmmakers figured out a touching way to pay tribute to Walker
```

```python
while "retiring" his character. At the premiere Wednesday night in Hollywood, Walker's co-star and close friend Vin Diesel gave a tearful speech before the screening, saying "This movie is more than a movie." "You'll feel it when you see it," Diesel said. "There's something emotional that happens to you, where you walk out of this movie and you appreciate everyone you love because you just never know when the last day is you're gonna see them." There have been multiple tributes to Walker leading up to the release. Diesel revealed in an interview with the "Today" show that he had named his newborn daughter after Walker.
Social media has also been paying homage to the late actor. A week after Walker's death, about 5,000 people attended an outdoor memorial to him in Los Angeles. Most had never met him. Marcus Coleman told CNN he spent almost $1,000 to truck in a banner from Bakersfield for people to sign at the memorial. "It's like losing a friend or a really close family member ... even though he is an actor and we never really met face to face," Coleman said. "Sitting there, bringing his movies into your house or watching on TV, it's like getting to know somebody. It really, really hurts." Walker's younger brother Cody told People magazine that he was initially nervous about how "Furious 7" would turn out, but he is happy with the film. "It's bittersweet, but I think Paul would be proud," he said. CNN's Paul Vercammen contributed to this report.
"""
summary_text = summarization(original_text)[0]['summary_text']
print("Summary:", summary_text)
```

Note that the first time you execute this, it'll download the model architecture and the weights and tokenizer configuration.

We specify the "summarization" task to the pipeline, and then we simply pass our long text to it. Here is the output:

```
Summary:  Paul Walker died in November 2013 after a car crash in Los Angeles .
The late actor was one of the nicest guys in Hollywood .
```

```
The release of "Furious 7" on Friday offers a chance to grieve again .
There have been multiple tributes to Walker leading up to the film's release .
```

Here is another example:

```python
print("="*50)
# another example
original_text = """
For the first time in eight years, a TV legend returned to doing what he does best.
Contestants told to "come on down!" on the April 1 edition of "The Price Is Right" encountered not host Drew Carey but another familiar face in charge of the proceedings.
Instead, there was Bob Barker, who hosted the TV game show for 35 years before stepping down in 2007.
Looking spry at 91, Barker handled the first price-guessing game of the show, the classic "Lucky Seven," before turning hosting duties over to Carey, who finished up.
Despite being away from the show for most of the past eight years, Barker didn't seem to miss a beat.
"""
summary_text = summarization(original_text)[0]['summary_text']
print("Summary:", summary_text)
```

Output:

```
==================================================
Summary:  Bob Barker returns to "The Price Is Right" for the first time in eight years .
The 91-year-old hosted the show for 35 years before stepping down in 2007 .
```

```
Drew Carey finished up hosting duties on the April 1 edition of the game show .
Barker handled the first price-guessing game of the show .
```

Note: *I brought the samples from CNN/DailyMail dataset.*

As you can see, the model generated an entirely new summarized text that does not belong to the original text.

This is the quickest way to use transformers. In the next section, we will learn another way to perform text summarization and customize how we want to generate the output.

Using T5 Model

The following code cell initializes the T5 transformer model along with its tokenizer:

```
from transformers import T5ForConditionalGeneration, T5Tokenizer

# initialize the model architecture and weights
model = T5ForConditionalGeneration.from_pretrained("t5-base")
# initialize the model tokenizer
tokenizer = T5Tokenizer.from_pretrained("t5-base")
```

The first time you execute the above code, will download the `t5-base` model architecture, weights, tokenizer vocabulary, and configuration.

We're using `from_pretrained()` method to load it as a pre-trained model, T5 comes with three versions in this library, `t5-small`, which is a smaller version of `t5-base`, and `t5-large` that is larger and more accurate than the others.

If you want to do summarization in a different language than English, and if it's not available in the available models, consider pre-training a model from scratch using your dataset. This tutorial will help you do that.

Let's set our text we want to summarize:

```
article = """
Justin Timberlake and Jessica Biel, welcome to parenthood.
The celebrity couple announced the arrival of their son, Silas Randall Timberlake, in statements to People.
"Silas was the middle name of Timberlake's maternal grandfather Bill Bomar, who died in 2012, while Randall is the musician's own middle name, as well as his father's first," People reports.
The couple announced the pregnancy in January, with an Instagram post. It is the first baby for both.
"""
```

Now let's encode this text to be suitable for the model as an input:

```
# encode the text into tensor of integers using the appropriate tokenizer
inputs = tokenizer.encode("summarize: " + article, return_tensors="pt", max_length=512, truncation=True)
```

We've used `tokenizer.encode()` method to convert the string text to a list of integers, where each integer is a unique token.

We set the `max_length` to 512, indicating that we do not want the original text to bypass 512 tokens; we also set `return_tensors` to `"pt"` to get PyTorch tensors as output.

Notice we prepended the text with `"summarize: "` text, and that's because T5 isn't just for text summarization. You can use it for any text-to-text transformation, such as machine translation or question answering, or even paraphrasing.

For example, we can use the T5 transformer for machine translation, and you can set `"translate English to German: "` instead of `"summarize: "` and you'll get a German translation output (more precisely, you'll get a summarized German translation, as you'll see why in `model.generate()`). For more information about translation, check this tutorial.

Finally, let's generate the summarized text and print it:

```
# generate the summarization output
outputs = model.generate(
    inputs,
    max_length=150,
    min_length=40,
    length_penalty=2.0,
    num_beams=4,
    early_stopping=True)
# just for debugging
print(outputs)
print(tokenizer.decode(outputs[0]))
```

Output:

```
tensor([[    0,     8,  1158,  2162,     8,  8999,    16,  1762,     3,     5,
            34,    19,     8,   166,  1871,    21,   321,    13,   135,     3,
             5,     8,  1871,    19,     8,  2214,   564,    13, 25045, 16948,
            31,     7, 28574, 18573,     6,   113,  3977,    16,  1673,     3,
             5]])
```

```
the couple announced the pregnancy in January. it is the first baby for both of
```

them.

the baby is the middle name of Timberlake's maternal grandfather, who died in 2012.

Excellent, the output looks concise and is newly generated with a new summarizing style.

Going to the most exciting part, the parameters passed to `model.generate()` method are:

- `max_length`: The maximum number of tokens to generate, we have specified a total of 150. You can change that if you want.
- `min_length`: This is the minimum number of tokens to generate. If you look closely at the tensor output, you'll count a total of 41 tokens, so it respected what we've specified, 40. Note that this will also work if you set it to another task, such as English to German translation.
- `length_penalty`: Exponential penalty to the length, 1.0 means no penalty. Increasing this parameter will increase the size of the output text.
- `num_beams`: Specifying this parameter will lead the model to use beam search instead of greedy search, setting `num_beams` to 4, will allow the model to lookahead for four possible words (1 in the case of greedy search), to keep the most likely 4 of hypotheses at each time step, and choosing the one that has the overall highest probability.
- `early_stopping`: We set it to `True` so that generation is finished when all beam hypotheses reach the end of the string token (EOS).

We then the `decode()` method from the tokenizer to convert the tensor back to human-readable text.

Conclusion

There are a lot of other parameters to tweak in `model.generate()` method. I highly encourage you to check this tutorial from the HuggingFace blog.

Alright, that's it for this tutorial. You've learned two ways to use HuggingFace's transformers library to perform text summarization. Check out the documentation here.

SourceCode:

using_pipeline.py

```
from transformers import pipeline

# using pipeline API for summarization task
summarization = pipeline("summarization")
original_text = """
Paul Walker is hardly the first actor to die during a production.
But Walker's death in November 2013 at the age of 40 after a car crash was especially eerie given his rise to fame in the "Fast and Furious" film franchise.
The release of "Furious 7" on Friday offers the opportunity for fans to remember -- and possibly grieve again -- the man that so many have praised as one of the nicest guys in Hollywood.
"He was a person of humility, integrity, and compassion," military veteran Kyle Upham said in an email to CNN.
Walker secretly paid for the engagement ring Upham shopped for with his bride.
"We didn't know him personally but this was apparent in the short time we spent with him.
I know that we will never forget him and he will always be someone very special to us," said Upham.
The actor was on break from filming "Furious 7" at the time of the fiery accident, which also claimed the life of the car's driver, Roger Rodas.
Producers said early on that they would not kill off Walker's character, Brian O'Connor, a former cop turned road racer. Instead, the script was rewritten and special effects were used to finish scenes, with Walker's brothers, Cody and Caleb, serving as body doubles.
There are scenes that will resonate with the audience -- including the ending, in which the filmmakers figured out a touching way to pay tribute to Walker while "retiring" his character. At the premiere Wednesday night in Hollywood, Walker's co-star and close friend Vin Diesel gave a tearful speech before the screening, saying "This movie is more than a movie." "You'll feel it when you see it," Diesel said. "There's something emotional that happens to you, where you walk out of this movie and you appreciate everyone you love because you just never know when the last day is you're gonna see them." There have been multiple tributes to Walker leading up to the release. Diesel revealed in an interview with the "Today" show that he had named his newborn daughter after Walker.
Social media has also been paying homage to the late actor. A week after Walker's death, about 5,000
```

```
people attended an outdoor memorial to him in Los Angeles. Most had never met him. Marcus Coleman told CNN he spent almost $1,000 to truck in a banner from Bakersfield for people to sign at the memorial. "It's like losing a friend or a really close family member ... even though he is an actor and we never really met face to face," Coleman said. "Sitting there, bringing his movies into your house or watching on TV, it's like getting to know somebody. It really, really hurts." Walker's younger brother Cody told People magazine that he was initially nervous about how "Furious 7" would turn out, but he is happy with the film. "It's bittersweet, but I think Paul would be proud," he said. CNN's Paul Vercammen contributed to this report.
"""
summary_text = summarization(original_text)[0]['summary_text']
print("Summary:", summary_text)
print("="*50)
# another example
original_text = """
For the first time in eight years, a TV legend returned to doing what he does best.
Contestants told to "come on down!" on the April 1 edition of "The Price Is Right" encountered not host Drew Carey but another familiar face in charge of the proceedings.
Instead, there was Bob Barker, who hosted the TV game show for 35 years before stepping down in 2007.
Looking spry at 91, Barker handled the first price-guessing game of the show, the classic "Lucky Seven," before turning hosting duties over to Carey, who finished up.
Despite being away from the show for most of the past eight years, Barker didn't seem to miss a beat.
"""
summary_text = summarization(original_text)[0]['summary_text']
print("Summary:", summary_text)
```

using_t5.py

```
from transformers import T5ForConditionalGeneration, T5Tokenizer

# initialize the model architecture and weights
model = T5ForConditionalGeneration.from_pretrained("t5-base")
# initialize the model tokenizer
tokenizer = T5Tokenizer.from_pretrained("t5-base")
article = """
Justin Timberlake and Jessica Biel, welcome to parenthood.
The celebrity couple announced the arrival of their son, Silas Randall Timberlake, in statements to People.
"Silas was the middle name of Timberlake's maternal grandfather Bill Bomar, who died in 2012, while
```

```
Randall is the musician's own middle name, as well as his father's first," People reports.
The couple announced the pregnancy in January, with an Instagram post. It is the first baby for both.
"""
# encode the text into tensor of integers using the appropriate tokenizer
inputs = tokenizer.encode("summarize: " + article, return_tensors="pt", max_length=512,
truncation=True)
# generate the summarization output
outputs = model.generate(
    inputs,
    max_length=150,
    min_length=40,
    length_penalty=2.0,
    num_beams=4,
    early_stopping=True)
# just for debugging
print(outputs)
print(tokenizer.decode(outputs[0]))
```

CHAPTER 11: Sentiment Analysis using VADER in Python

Learn how you can easily perform sentiment analysis on text in Python using vaderSentiment library.

Text-Based data is known to be abundant since it is generally practically everywhere, including social media interactions, reviews, comments and even surveys.

Sentences hold many valuable information that may have a huge impact on the decision making process of a given company, since it is a way to perform customer analytics to get to better know your users hence giving them better products in the future.

In this tutorial, we will learn on how to extract the sentiment score (-1 for negative, 0 for neutral and 1 for positive) from any given text using the vaderSentiment library.

VADER stands for **V**alence **A**ware **D**ictionary and s**E**ntiment **R**easoner, which is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media, and works well on text from other domains.

Installing the requirements for this tutorial:

```
pip install vaderSentiment
```

The nice thing about this library is that you don't have to train anything in order to use it, you'll soon realize that it is pretty straightforward to use it, open up a new Python file and import `SentimentIntensityAnalyzer` class:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
```

```python

# init the sentiment analyzer
sia = SentimentIntensityAnalyzer()
```

We will create a list of sentences on which we will apply sentiment analysis using the `polarity_score()` method from `SentimentIntensityAnalyzer` class.

```python
sentences = [
    "This food is amazing and tasty !",
    "Exoplanets are planets outside the solar system",
    "This is sad to see such bad behavior"
]

for sentence in sentences:
    score = sia.polarity_scores(sentence)["compound"]
    print(f'The sentiment value of the sentence :"{sentence}" is : {score}')
```

`polarity_score()` method returns a float for the sentiment strength based on the input text, the result of running the above code is the following:

```
The sentiment value of the sentence :"This food is amazing and tasty !" is : 0.6239
The sentiment value of the sentence :"Exoplanets are planets outside the solar system" is : 0.0
The sentiment value of the sentence :"This is sad to see such bad behavior" is : -0.765
```

We can also calculate the percentage of each sentiment present in that sentence using "pos", "neu" and "neg" keys after computing the polarity score.

```
for sentence in sentences:
    print(f'For the sentence "{sentence}"')
    polarity = sia.polarity_scores(sentence)
    pos = polarity["pos"]
    neu = polarity["neu"]
    neg = polarity["neg"]
    print(f'The percententage of positive sentiment in :"{sentence}" is : {round(pos*100,2)} %')
    print(f'The percententage of neutral sentiment in :"{sentence}" is : {round(neu*100,2)} %')
    print(f'The percententage of negative sentiment in :"{sentence}" is : {round(neg*100,2)} %')
    print("="*50)
```

Output:

```
For the sentence "This food is amazing and tasty !"
The percententage of positive sentiment in :"This food is amazing and tasty !" is : 40.5 %
The percententage of neutral sentiment in :"This food is amazing and tasty !" is : 59.5 %
The percententage of negative sentiment in :"This food is amazing and tasty !" is : 0.0 %
==================================================
For the sentence "Exoplanets are planets outside the solar system"
The percententage of positive sentiment in :"Exoplanets are planets outside the solar system" is : 0.0 %
The percententage of neutral sentiment in :"Exoplanets are planets outside the solar system" is : 100.0 %
```

```
The percententage of negative sentiment in :"Exoplanets are planets outside the solar system" is : 0.0 %
=====================================================
For the sentence "This is sad to see such bad behavior"
The percententage of positive sentiment in :"This is sad to see such bad behavior" is : 0.0 %
The percententage of neutral sentiment in :"This is sad to see such bad behavior" is : 47.6 %
The percententage of negative sentiment in :"This is sad to see such bad behavior" is : 52.4 %
=====================================================
```

Conclusion

In this tutorial you have learned:

- Learned the importance of sentiment analysis in Natural Language Processing.
- Learned to extract sentimental scores from a sentence using the VaderSentiment package in Python.

SourceCode:

sentiment_analysis.py

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# init the sentiment analyzer
sia = SentimentIntensityAnalyzer()

sentences = [
    "This food is amazing and tasty !",
    "Exoplanets are planets outside the solar system",
    "This is sad to see such bad behavior"
```

```
]

for sentence in sentences:
    score = sia.polarity_scores(sentence)["compound"]
    print(f'The sentiment value of the sentence :"{sentence}" is : {score}')

for sentence in sentences:
    print(f'For the sentence "{sentence}"')
    polarity = sia.polarity_scores(sentence)
    pos = polarity["pos"]
    neu = polarity["neu"]
    neg = polarity["neg"]
    print(f'The percententage of positive sentiment in :"{sentence}" is : {round(pos*100,2)} %')
    print(f'The percententage of neutral sentiment in :"{sentence}" is : {round(neu*100,2)} %')
    print(f'The percententage of negative sentiment in :"{sentence}" is : {round(neg*100,2)} %')
    print("="*50)
```

CHAPTER 12: Translate Languages in Python

Learn how to make a language translator and detector using Googletrans library (Google Translation API) for translating more than 100 languages with Python.

Google translate is a free service that translates words, phrases and entire web pages into more than 100 languages. You probably already know it and you have used it many times in your life.

In this tutorial, you will learn how to perform language translation in Python using [Googletrans] library. Googletrans is a free and unlimited Python library that make unofficial [Ajax] calls to Google Translate API in order to detect languages and translate text.

Here are the main features of this library:

- Auto language detection (it offers language detection as well)
- Bulk translations
- Fast & reliable
- HTTP/2 support
- Connection pooling

Note that Googletrans makes API calls to the Google translate API, if you want a reliable use, then consider using an official API or [making your own machine translation machine learning model].

First, let's install it using pip:

```
pip3 install googletrans
```

Translating Text

Importing necessary libraries:

```
from googletrans import Translator, constants
from pprint import pprint
```

Googletrans provides us with a convenient interface, let's initialize our translator instance:

```
# init the Google API translator
translator = Translator()
```

Note that Translator class has several optional arguments:

- `service_urls`: This should be a list of strings that are the URLs of google translate API, an example is `["translate.google.com", "translate.google.co.uk"]`.
- `user_agent`: A string that will be included in User-Agent header in the request.
- `proxies` (dictionary): A Python dictionary that maps protocol or protocol and host to the URL of the proxy, an example is `{'http': 'example.com:3128', 'http://domain.example': 'example.com:3555'}`, more on proxies in this tutorial.
- `timeout`: The timeout of each request you make, expressed in seconds.

Now we simply use `translate()` method to get the translated text:

```
# translate a spanish text to english text (by default)
translation = translator.translate("Hola Mundo")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
```

This will print the original text and language along with the translated text and language:

```
Hola Mundo (es) --> Hello World (en)
```

If the above code results in an error like this:

```
AttributeError: 'NoneType' object has no attribute 'group'
```

Then you have to uninstall the current googletrans version and install the new one using the following commands:

```
$ pip3 uninstall googletrans
$ pip3 install googletrans==3.1.0a0
```

Going back to the code, it automatically detects the language and translate to english by default, let's translate to another language, arabic for instance:

```
# translate a spanish text to arabic for instance
translation = translator.translate("Hola Mundo", dest="ar")
print(f"{translation.origin} ({translation.src}) --> {translation.text}
({translation.dest})")
```

`"ar"` is the language code for arabic, here is the output:

```
Hola Mundo (es) --> مرحبا بالعالم (ar)
```

Now let's set a source language and translate to English:

```
# specify source language
translation = translator.translate("Wie gehts ?", src="de")
```

```python
print(f"{translation.origin} ({translation.src}) --> {translation.text}
({translation.dest})")
```

Output:

```
Wie gehts ? (de) --> How are you ? (en)
```

You can also check other translations and some other extra data:

```python
# print all translations and other data
pprint(translation.extra_data)
```

See the output:

```
{'all-translations': [['interjection',
                       ['How are you doing?', "What's up?"],
                       [['How are you doing?', ["Wie geht's?"]],
                        ["What's up?", ["Wie geht's?"]]],
                       "Wie geht's?",
                       9]],
 'confidence': 1.0,
 'definitions': None,
 'examples': None,
 'language': [['de'], None, [1.0], ['de']],
 'original-language': 'de',
 'possible-mistakes': None,
 'possible-translations': [['Wie gehts ?',
                            None,
                            [['How are you ?', 1000, True, False],
```

```
                     ["How's it going ?", 1000, True, False],
                     ['How are you?', 0, True, False]],
                    [[0, 11]],
                    'Wie gehts ?',
                    0,
                    0]],
 'see-also': None,
 'synonyms': None,
 'translation': [['How are you ?', 'Wie gehts ?', None, None, 1]]}
```

A lot of data to benefit from, you have all the possible translations, confidence, definitions and even examples.

Translating List of Phrases

You can also pass a list of text to translate each sentence individually:

```
# translate more than a phrase
sentences = [
    "Hello everyone",
    "How are you ?",
    "Do you speak english ?",
    "Good bye!"
]
translations = translator.translate(sentences, dest="tr")
for translation in translations:
    print(f"{translation.origin} ({translation.src}) --> {translation.text}
({translation.dest})")
```

Output:

```
Hello everyone (en) --> herkese merhaba (tr)
How are you ? (en) --> Nasılsın ? (tr)
Do you speak english ? (en) --> İngilizce biliyor musunuz ? (tr)
Good bye! (en) --> Güle güle! (tr)
```

Language Detection

Google Translate API offers us language detection call as well:

```
# detect a language
detection = translator.detect(" नमस्ते दुनिया ")
print("Language code:", detection.lang)
print("Confidence:", detection.confidence)
```

This will print the code of the detected language along with confidence rate (1.0 means 100% confident):

```
Language code: hi
Confidence: 1.0
```

This will return the language code, to get the full language name, you can use the `LANGUAGES` dictionary provided by Googletrans:

```
print("Language:", constants.LANGUAGES[detection.lang])
```

Output:

```
Language: hindi
```

Supported Languages

As you may know, Google Translate supports more than 100 languages, let's print all of them:

```
# print all available languages
print("Total supported languages:", len(constants.LANGUAGES))
print("Languages:")
pprint(constants.LANGUAGES)
```

Here is a truncated output:

```
Total supported languages: 107
{'af': 'afrikaans',
 'sq': 'albanian',
 'am': 'amharic',
 'ar': 'arabic',
 'hy': 'armenian',
...
<SNIPPED>
...
 'vi': 'vietnamese',
 'cy': 'welsh',
 'xh': 'xhosa',
 'yi': 'yiddish',
 'yo': 'yoruba',
 'zu': 'zulu'}
```

Conclusion

There you have it, this library is a great deal for everyone that wants a quick way to translate text in an application. However, this library is unofficial as mentioned earlier, the author noted that the maximum character length on a single text is 15K.

It also doesn't guarantee that the library would work properly at all times, if you want to use a stable API you should use the official Google Translate API.

If you get HTTP `5xx` errors with this library, then Google has banned your IP address, it's because using this library a lot, Google translate may block your IP address, you'll need to consider using proxies by passing a proxy dictionary to `proxies` parameter in `Translator()` class, or use the official API as discussed.

Also, I've written a quick Python script that will allow you to translate text into sentences as well as in documents in the command line, check it here.

Finally, I encourage you to further explore the library, check out its official documentation.

Finally, if you're a beginner and want to learn Python, I suggest you take the Python For Everybody Coursera course, in which you'll learn a lot about Python. You can also check our resources and courses page to see the Python resources I recommend!

SourceCode:

translator.py

```
from googletrans import Translator, constants
from pprint import pprint

```

```
# init the Google API translator
translator = Translator()

# translate a spanish text to english text (by default)
translation = translator.translate("Hola Mundo")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")

# translate a spanish text to arabic for instance
translation = translator.translate("Hola Mundo", dest="ar")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")

# specify source language
translation = translator.translate("Wie gehts ?", src="de")
print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")
# print all translations and other data
pprint(translation.extra_data)

# translate more than a phrase
sentences = [
    "Hello everyone",
    "How are you ?",
    "Do you speak english ?",
    "Good bye!"
]
translations = translator.translate(sentences, dest="tr")
for translation in translations:
    print(f"{translation.origin} ({translation.src}) --> {translation.text} ({translation.dest})")

# detect a language
detection = translator.detect(" नमस्ते दुनिया ")
print("Language code:", detection.lang)
print("Confidence:", detection.confidence)
# print the detected language
print("Language:", constants.LANGUAGES[detection.lang])

# print all available languages
print("Total supported languages:", len(constants.LANGUAGES))
print("Languages:")
```

```
pprint(constants.LANGUAGES)
```

translate_doc.py

```
from googletrans import Translator
import argparse
import os

# init the translator
translator = Translator()

def translate(text, src="auto", dest="en"):
    """Translate `text` from `src` language to `dest`"""
    return translator.translate(text, src=src, dest=dest).text


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simple Python script to translate text using Google Translate API (googletrans wrapper)")
    parser.add_argument("target", help="Text/Document to translate")
    parser.add_argument("-s", "--source", help="Source language, default is Google Translate's auto detection", default="auto")
    parser.add_argument("-d", "--destination", help="Destination language, default is English", default="en")

    args = parser.parse_args()
    target = args.target
    src = args.source
    dest = args.destination

    if os.path.isfile(target):
        # translate a document instead
        # get basename of file
        basename = os.path.basename(target)
        # get the path dir
        dirname = os.path.dirname(target)
        try:
```

```
            filename, ext = basename.split(".")
        except:
            # no extension
            filename = basename
            ext = ""

        translated_text = translate(open(target).read(), src=src, dest=dest)
        # write to new document file
        open(os.path.join(dirname, f"{filename}_{dest}{f'.{ext}' if ext else ''}"),
"w").write(translated_text)
    else:
        # not a file, just text, print the translated text to standard output
        print(translate(target, src=src, dest=dest))
```

Usage:

```
python translate_doc.py --help
```

Output:

```
usage: translate_doc.py [-h] [-s SOURCE] [-d DESTINATION] target

   Simple Python script to translate text using Google Translate API
(googletrans
   wrapper)

   positional arguments:
   target                Text/Document to translate

   optional arguments:
   -h, --help            show this help message and exit
   -s SOURCE, --source SOURCE
```

```
                        Source language, default is Google Translate's auto
                        detection
  -d DESTINATION, --destination DESTINATION
                        Destination language, default is English
```

For instance, if you want to translate text in the document `wonderland.txt` from english (`en`) to arabic (`ar`), you can use:

```
python translate_doc.py wonderland.txt --source en --destination ar
```

A new file `wonderland_ar.txt` will appear in the current directory that contains the translated document.

You can also translate text and print in the stdout:

```
python translate_doc.py 'Bonjour' -s fr -d en
```

Output:

```
'Hello'
```

CHAPTER 13: Perform Text Classification in Python using Tensorflow 2 and Keras

Building deep learning models (using embedding and recurrent layers) for different text classification problems such as sentiment analysis or 20 news group classification using Tensorflow and Keras in Python

Text classification is one of the essential and common tasks in supervised machine learning. It is about assigning a category (a class) to documents, articles, books, reviews, tweets, or anything that involves text. It is a core task in natural language processing.

Many applications appeared to use text classification as the main task; examples include spam filtering, sentiment analysis, speech tagging, language detection, etc.

In this tutorial, we will build a text classifier model using RNNs using Tensorflow in Python; we will be using the IMDB reviews dataset, which has 50K real-world movie reviews along with their sentiment (positive or negative). At the end of this tutorial, I will show you how you can integrate your own dataset so you can train the model on it.

Although we're using a sentiment analysis dataset, this tutorial is intended to perform text classification on any task. If you wish to perform sentiment analysis out of the box, check this tutorial.

If you wish to use state-of-the-art transformer models such as BERT, check this tutorial where we fine-tune BERT for our custom dataset.

To get started, you need to install the following libraries:

```
pip3 install tqdm numpy tensorflow==2.0.0 sklearn
```

Now open up a new Python notebook or file and follow along. Let's import our necessary modules:

```
from tqdm import tqdm
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Dense, Dropout, LSTM, Embedding, Bidirectional
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import TensorBoard
from sklearn.model_selection import train_test_split
import numpy as np

from glob import glob
import random
import os
```

Data Preparation

Before we load our dataset into Python, you need to download the dataset here; you'll see two files there, `reviews.txt`, which contains a movie review in each line, and `labels.txt` which holds its corresponding label.

The below function loads and preprocesses the dataset:

```
def load_imdb_data(num_words, sequence_length, test_size=0.25, oov_token=None):
    # read reviews
```

```python
reviews = []
with open("data/reviews.txt") as f:
    for review in f:
        review = review.strip()
        reviews.append(review)

labels = []
with open("data/labels.txt") as f:
    for label in f:
        label = label.strip()
        labels.append(label)
# tokenize the dataset corpus, delete uncommon words such as names, etc.
tokenizer = Tokenizer(num_words=num_words, oov_token=oov_token)
tokenizer.fit_on_texts(reviews)
X = tokenizer.texts_to_sequences(reviews)
X, y = np.array(X), np.array(labels)
# pad sequences with 0's
X = pad_sequences(X, maxlen=sequence_length)
# convert labels to one-hot encoded
y = to_categorical(y)
# split data to training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=1)
data = {}
data["X_train"] = X_train
data["X_test"]= X_test
data["y_train"] = y_train
data["y_test"] = y_test
data["tokenizer"] = tokenizer
data["int2label"] =  {0: "negative", 1: "positive"}
```

```
    data["label2int"] = {"negative": 0, "positive": 1}
    return data
```

A lot to cover here. This function does the following:

- It loads the dataset from the files mentioned earlier.
- After that, it uses Keras' utility Tokenizer class, which helps us remove all punctuations automatically, tokenize the corpus, remove rare words such as names, and convert text sentences into a sequence of numbers (each word corresponds to a number).
- We already know that neural networks expect a fixed-length input, and since the reviews don't have the same length of words, we need a way to make the length of sequences a fixed size. pad_sequences() function comes to the rescue; we tell it we want only say 300 words in each review (`maxlen` parameter), it will remove the words that exceed that number, and it'll pad with 0's to the reviews below 300.
- We use Keras' `to_categorical()` function to one-hot encode the labels, this is a binary classification, so it'll convert the label 0 to [1, 0] vector and 1 to [0, 1]. But in general, it converts categorical labels to a fixed-length vector.
- After that, we split our dataset into a training set and a testing set using sklearn's `train_test_split()` function and use the data dictionary to add all the things we need in the training process: the dataset, the tokenizer, and the label encoding dictionary.

Building the Model

Now that we know how to load the dataset, let's build our model.

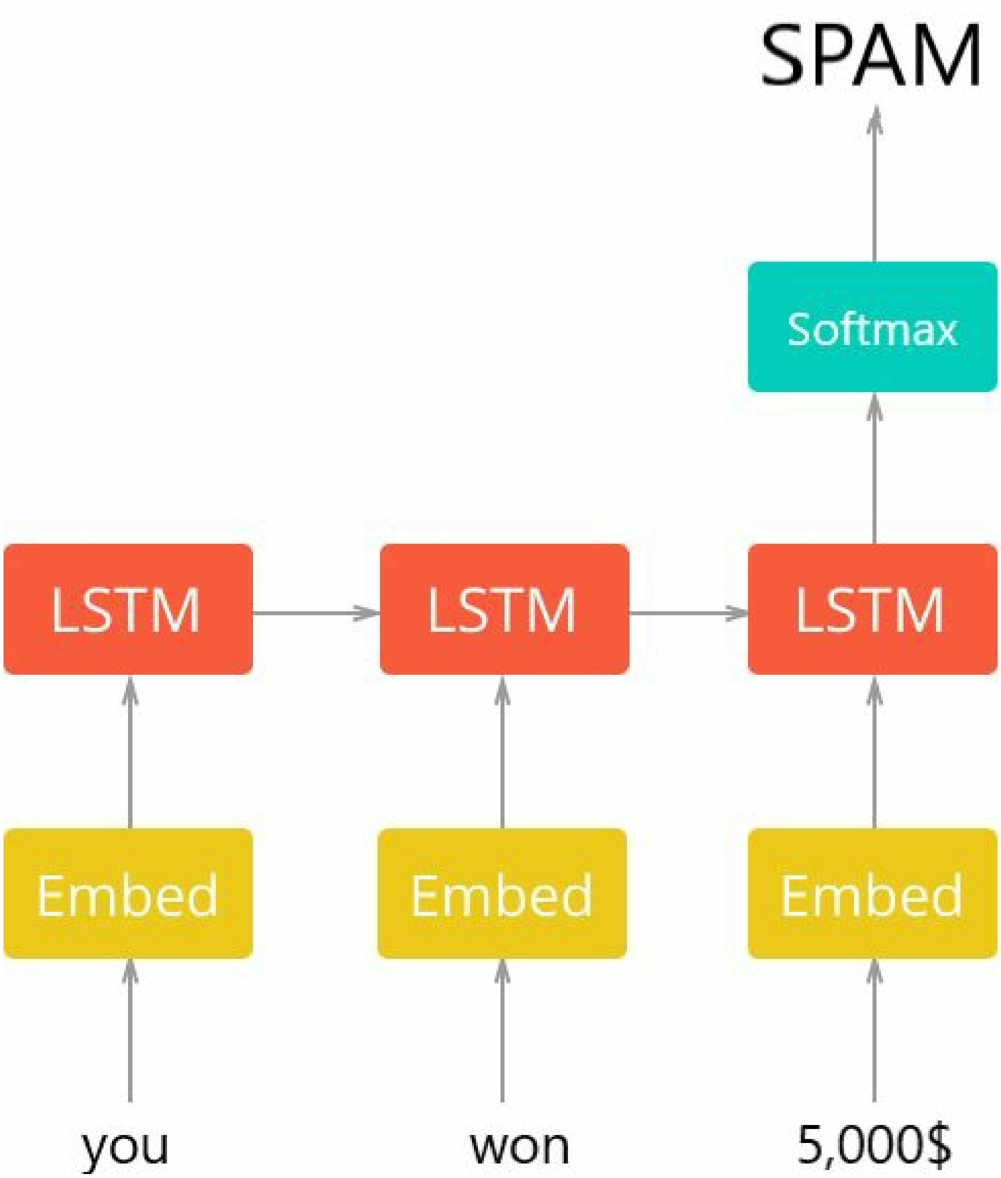
We will use an **embedding layer** as the first layer of the model. Embedding proved to be useful in mapping categorical variables (words, in this case) to a vector of continuous numbers; it is widely used in natural language processing tasks.

More precisely, we will use pre-trained **GloVe** word vectors, which are pretrained vectors that map each word to a vector of a specific size. This size parameter is often called embedding size, although GloVe uses 50, 100, 200, or 300 embedding size vectors. We will try all of them in this tutorial and see which performs best. Also, two words with the same meaning tend to have very close vectors.

The second layer will be recurrent, you'll have the choice to choose any recurrent cell you want, including **LSTM**, **GRU**, or even just **SimpleRNN,** and again, we'll see which one outperforms the others.

The last layer should be a **dense layer** with `N` neurons. `N` should be the same number of categories in your dataset. In the case of positive/negative sentiment analysis, it should be 2.

The general architecture of the model is shown in the following figure (grabbed from the spam classifier tutorial):



Now you need to download the pre-trained GloVe (download here). After you have done that, extract all of them in the data folder (you'll find different vectors for different embedding sizes), the below function loads these vectors:

```
def get_embedding_vectors(word_index, embedding_size=100):
    embedding_matrix = np.zeros((len(word_index) + 1, embedding_size))
    with open(f"data/glove.6B.{embedding_size}d.txt", encoding="utf8") as f:
        for line in tqdm(f, "Reading GloVe"):
```

```
        values = line.split()
        # get the word as the first word in the line
        word = values[0]
        if word in word_index:
            idx = word_index[word]
            # get the vectors as the remaining values in the line
            embedding_matrix[idx] = np.array(values[1:], dtype="float32")
    return embedding_matrix
```

Now we are going to need a function that creates the model from scratch, given the hyperparameters:

```
def create_model(word_index, units=128, n_layers=1, cell=LSTM,
bidirectional=False,
                embedding_size=100, sequence_length=100, dropout=0.3,
                loss="categorical_crossentropy", optimizer="adam",
                output_length=2):
    """Constructs a RNN model given its parameters"""
    embedding_matrix = get_embedding_vectors(word_index,
embedding_size)
    model = Sequential()
    # add the embedding layer
    model.add(Embedding(len(word_index) + 1,
              embedding_size,
              weights=[embedding_matrix],
              trainable=False,
              input_length=sequence_length))
    for i in range(n_layers):
        if i == n_layers - 1:
            # last layer
```

```
        if bidirectional:
            model.add(Bidirectional(cell(units, return_sequences=False)))
        else:
            model.add(cell(units, return_sequences=False))
    else:
        # first layer or hidden layers
        if bidirectional:
            model.add(Bidirectional(cell(units, return_sequences=True)))
        else:
            model.add(cell(units, return_sequences=True))
    model.add(Dropout(dropout))
model.add(Dense(output_length, activation="softmax"))
# compile the model
model.compile(optimizer=optimizer, loss=loss, metrics=["accuracy"])
return model
```

I know there are a lot of parameters in this function. Well, to test various parameters, this function will be flexible to all parameters provided. Let's explain them:

- `word_index`: This is a dictionary that maps each word to its corresponding index number; this is produced by the previously mentioned Tokenizer object.
- `units`: This is the number of neurons in each recurrent layer; it defaults to 128, but use any number you want, be aware that the more units, the more weights to adjust, and therefore, the slower it'll be in the training process.
- `n_layers`: This is the number of recurrent layers we want to use; 1 is a good one to start with.
- `cell`: The recurrent cell you wish to use, LSTM is a good choice.
- `bidirectional`: This is a boolean variable that indicates whether we

use bidirectional recurrent layers.

- embedding_size: The size of our embedding vector we mentioned earlier, we will experiment with various sizes.
- sequence_length: The number of tokenized words on each text sample to feed into the neural networks, we will experiment with this parameter too.
- dropout: it is the probability of training a given node on the layer. It helps reduce overfitting. 40% is pretty good for this, but try to tweak it and see if it performs better. Check this tutorial for more information about dropouts.
- loss: It's the loss function to use for the training. By default, we're using the categorical cross-entropy function.
- optimizer: The optimizer function to use, we're using ADAM here.
- output_length: This is the number of neurons to use in the last layer. Since we're using only positive and negative sentiment classification, it must be 2.

When you look closely, you'll notice that I'm using the Embedding class with weights parameter. It specifies the pre-trained weights we just downloaded, we're also setting trainable to False, so these vectors won't change during the training process.

If your dataset is in a different language than English, make sure you find embedding vectors for the language you're using, if not, you shouldn't set weights parameter at all, and you need to set trainable to True, so you'll train the parameters of the vector from scratch, check this page for word vectors of your language.

Training the Model

Now to start training, we need to define all of the previously mentioned hyperparameters and more:

```
# max number of words in each sentence
SEQUENCE_LENGTH = 300
```

```
# N-Dimensional GloVe embedding vectors
EMBEDDING_SIZE = 300
# number of words to use, discarding the rest
N_WORDS = 10000
# out of vocabulary token
OOV_TOKEN = None
# 30% testing set, 70% training set
TEST_SIZE = 0.3
# number of CELL layers
N_LAYERS = 1
# the RNN cell to use, LSTM in this case
RNN_CELL = LSTM
# whether it's a bidirectional RNN
IS_BIDIRECTIONAL = False
# number of units (RNN_CELL ,nodes) in each layer
UNITS = 128
# dropout rate
DROPOUT = 0.4
### Training parameters
LOSS = "categorical_crossentropy"
OPTIMIZER = "adam"
BATCH_SIZE = 64
EPOCHS = 6

def get_model_name(dataset_name):
    # construct the unique model name
    model_name = f"{dataset_name}-{RNN_CELL.__name__}-seq-{SEQUENCE_LENGTH}-em-{EMBEDDING_SIZE}-w-{N_WORDS}-layers-{N_LAYERS}-units-{UNITS}-opt-{OPTIMIZER}-BS-{BATCH_SIZE}-d-{DROPOUT}"
```

```
    if IS_BIDIRECTIONAL:
        # add 'bid' str if bidirectional
        model_name = "bid-" + model_name
    if OOV_TOKEN:
        # add 'oov' str if OOV token is specified
        model_name += "-oov"
    return model_name
```

I've set the optimal parameters so far that I've found, the get_model_name() function produces a unique model name based on parameters; this is useful when it comes to comparing various parameters on TensorBoard.

Let's bring everything together and start training our model:

```
# create these folders if they does not exist
if not os.path.isdir("results"):
    os.mkdir("results")
if not os.path.isdir("logs"):
    os.mkdir("logs")
if not os.path.isdir("data"):
    os.mkdir("data")
# dataset name, IMDB movie reviews dataset
dataset_name = "imdb"
# get the unique model name based on hyper parameters on parameters.py
model_name = get_model_name(dataset_name)
# load the data
data = load_imdb_data(N_WORDS, SEQUENCE_LENGTH, TEST_SIZE,
oov_token=OOV_TOKEN)
# construct the model
model = create_model(data["tokenizer"].word_index, units=UNITS,
```

```
n_layers=N_LAYERS,
                cell=RNN_CELL, bidirectional=IS_BIDIRECTIONAL,
embedding_size=EMBEDDING_SIZE,
                sequence_length=SEQUENCE_LENGTH, dropout=DROPOUT,
                loss=LOSS, optimizer=OPTIMIZER,
output_length=data["y_train"][0].shape[0])
model.summary()
# using tensorboard on 'logs' folder
tensorboard = TensorBoard(log_dir=os.path.join("logs", model_name))
# start training
history = model.fit(data["X_train"], data["y_train"],
                batch_size=BATCH_SIZE,
                epochs=EPOCHS,
                validation_data=(data["X_test"], data["y_test"]),
                callbacks=[tensorboard],
                verbose=1)
# save the resulting model into 'results' folder
model.save(os.path.join("results", model_name) + ".h5")
```

This will take several minutes to train. Here is my execution output after the training is finished:

```
Reading GloVe: 400000it [00:17, 23047.55it/s]
Model: "sequential"
_________________________________________________________________
Layer (type)             Output Shape            Param #
=================================================================
embedding (Embedding)       (None, 300, 300)        37267200
_________________________________________________________________
lstm (LSTM)                 (None, 128)             219648
```

```
_________________________________________________________________
dropout (Dropout)            (None, 128)             0
_________________________________________________________________
dense (Dense)                (None, 2)               258
=================================================================
Total params: 37,487,106
Trainable params: 219,906
Non-trainable params: 37,267,200
_________________________________________________________________
Train on 35000 samples, validate on 15000 samples
Epoch 1/6
35000/35000 [==============================] - 186s 5ms/sample - loss: 0.4359 - accuracy: 0.7919 - val_loss: 0.2912 - val_accuracy: 0.8788
Epoch 2/6
35000/35000 [==============================] - 179s 5ms/sample - loss: 0.2857 - accuracy: 0.8820 - val_loss: 0.2608 - val_accuracy: 0.8919
Epoch 3/6
35000/35000 [==============================] - 175s 5ms/sample - loss: 0.2501 - accuracy: 0.8985 - val_loss: 0.2472 - val_accuracy: 0.8977
Epoch 4/6
35000/35000 [==============================] - 174s 5ms/sample - loss: 0.2184 - accuracy: 0.9129 - val_loss: 0.2525 - val_accuracy: 0.8997
Epoch 5/6
35000/35000 [==============================] - 185s 5ms/sample - loss: 0.1918 - accuracy: 0.9246 - val_loss: 0.2576 - val_accuracy: 0.9035
Epoch 6/6
35000/35000 [==============================] - 188s 5ms/sample - loss: 0.1598 - accuracy: 0.9391 - val_loss: 0.2494 - val_accuracy: 0.9004
```

Excellent, it reached about 90% accuracy after 6 epochs of training.

Testing the Model

Using the model is pretty straightforward. The below function uses the model.predict() method to produce the output:

```
def get_predictions(text):
    sequence = data["tokenizer"].texts_to_sequences([text])
    # pad the sequences
    sequence = pad_sequences(sequence, maxlen=SEQUENCE_LENGTH)
    # get the prediction
    prediction = model.predict(sequence)[0]
    return prediction, data["int2label"][np.argmax(prediction)]
```

So as you can see, in order to properly produce predictions, we need to use our previously used tokenizer to convert the text into sequences, after that, we pad sequences so it's a fixed-length sequence, and then we produce the output using model.predict() method, let's play around with this model:

```
text = "The movie is awesome!"
output_vector, prediction = get_predictions(text)
print("Output vector:", output_vector)
print("Prediction:", prediction)
```

Output:

```
Output vector: [0.3001343  0.69986564]
Prediction: positive
```

Let's use another text:

```
text = "The movie is bad."
output_vector, prediction = get_predictions(text)
print("Output vector:", output_vector)
print("Prediction:", prediction)
```

Output:

```
Output vector: [0.92491007 0.07508987]
Prediction: negative
```

It is pretty sure that it's a negative sentiment with about 92% confidence. Let's be more challenging:

```
text = "Not very good, but pretty good try."
output_vector, prediction = get_predictions(text)
print("Output vector:", output_vector)
print("Prediction:", prediction)
```

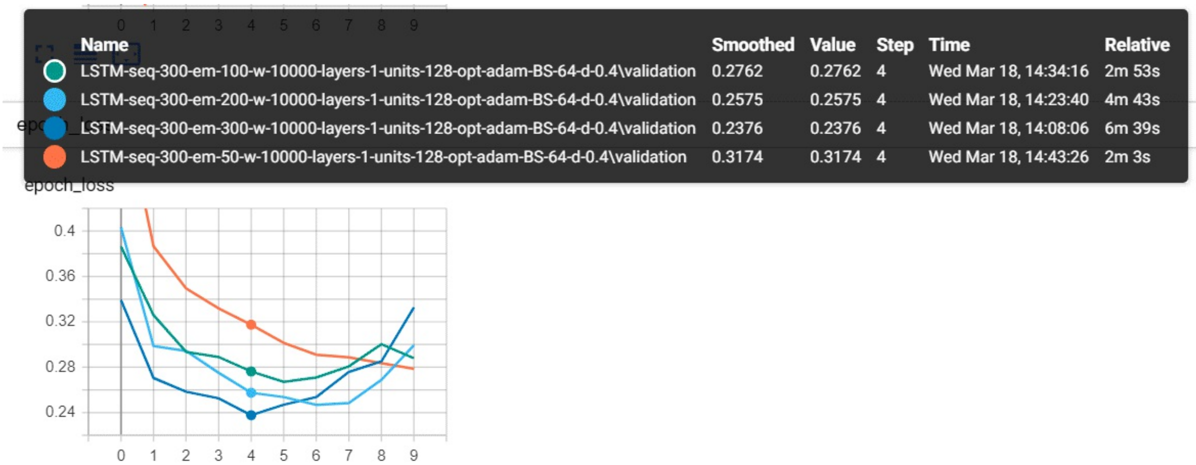
Output:

```
Output vector: [0.38528103 0.61471903]
Prediction: positive
```

It is pretty 61% sure that's a good sentiment, as you can see, it's giving interesting results, spend some time tricking the model!

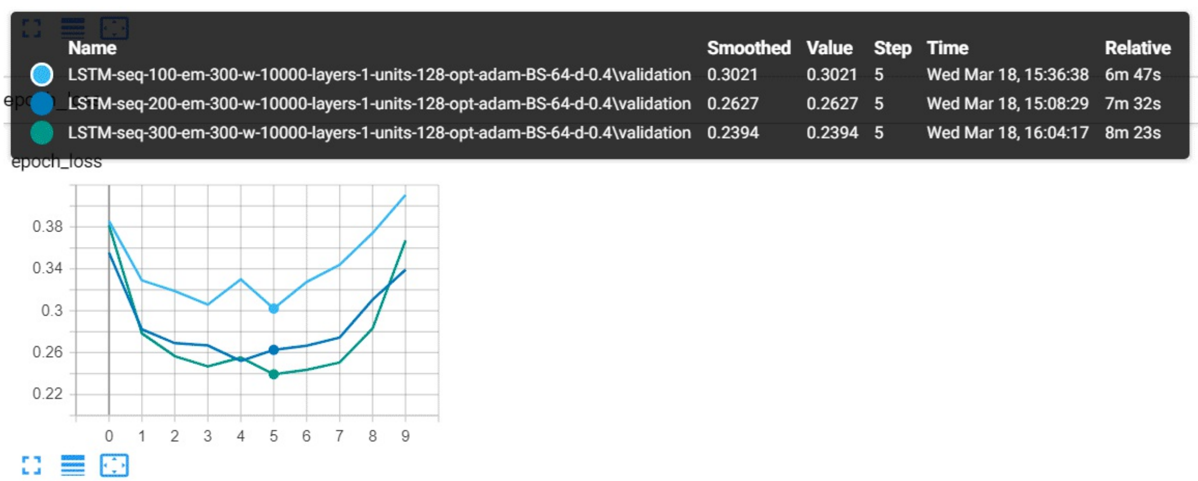
Hyperparameter Tuning

Before I came up with 90% accuracy, I have experimented with various hyper parameters, here are some of the interesting ones:



These are 4 models, and each has a different embedding size, as you can see, the one that has a 300 length vector (each word got a 300 length vector) reached the lowest validation loss value.

Here is another one when I used the sequence length as the varying parameter:



The model which has a 300 sequence length (the green one) tends to perform better.

Using tensorboard, you can see that after reaching epochs 4-5-6, the validation loss will try to increase again, that's clearly overfitting. That's why

I set epochs to 6. try to tweak other parameters such as dropout rate and see if you can decrease it furthermore.

Integrating Custom Datasets

Since this is a text classification tutorial, it would be useful if you can use your own datasets without changing much of this tutorial's code. In fact, all you have to change is the loading data function, previously we used the load_imdb_data() function, which returns a data dictionary that has:

- `X_train`: A NumPy array that is of the shape (number of training samples, sequence length) which contains all the sequences of each data sample.
- `X_test`: Same as above, but for testing samples.
- `y_train`: These are the labels of the training set, it's a NumPy array of the shape (number of testing samples, number of total categories), in the case of sentiment analysis, this should be something like (15000, 2)
- `y_test`: Same as above, but for testing samples.
- `tokenizer`: This is a Tokenizer instance from tensorflow.keras.preprocessing.text module, the object that is used to tokenize the corpus.
- `label2int`: A Python dictionary that converts a label to its corresponding encoded integer, in the sentiment analysis example, we used 1 for positive and 0 for negative.
- `int2label`: Vice-versa of the above.

Here is an example function that loads the [20 newsgroup dataset](#) (which contains around 18000 newsgroups posts on 20 topics), it uses sklearn's built-in function fetch_20newsgroups():

```
from sklearn.datasets import fetch_20newsgroups

def load_20_newsgroup_data(num_words, sequence_length, test_size=0.25, oov_token=None):
    # load the 20 news groups dataset
    # shuffling the data & removing each document's header, signature blocks
```

```
and quotation blocks
    dataset = fetch_20newsgroups(subset="all", shuffle=True, remove=
("headers", "footers", "quotes"))
    documents = dataset.data
    labels = dataset.target
    tokenizer = Tokenizer(num_words=num_words, oov_token=oov_token)
    tokenizer.fit_on_texts(documents)
    X = tokenizer.texts_to_sequences(documents)
    X, y = np.array(X), np.array(labels)
    # pad sequences with 0's
    X = pad_sequences(X, maxlen=sequence_length)
    # convert labels to one-hot encoded
    y = to_categorical(y)
    # split data to training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=1)
    data = {}
    data["X_train"] = X_train
    data["X_test"]= X_test
    data["y_train"] = y_train
    data["y_test"] = y_test
    data["tokenizer"] = tokenizer
    data["int2label"] = { i: label for i, label in enumerate(dataset.target_names)
}
    data["label2int"] = { label: i for i, label in enumerate(dataset.target_names)
}
    return data
```

Alright, good luck implementing your own text classifier, if you have any problems integrating one, post your comment below and I'll try to reach you as soon as possible.

As I have mentioned earlier, try to experiment with all the hyperparameters provided, I tried to write the code as flexible as possible so you can change only the parameters without doing anything else. If you outperformed my parameters, share them with us in the comments below!

SourceCode:

parameters.py

```
from tensorflow.keras.layers import LSTM

# max number of words in each sentence
SEQUENCE_LENGTH = 300
# N-Dimensional GloVe embedding vectors
EMBEDDING_SIZE = 300
# number of words to use, discarding the rest
N_WORDS = 10000
# out of vocabulary token
OOV_TOKEN = None
# 30% testing set, 70% training set
TEST_SIZE = 0.3
# number of CELL layers
N_LAYERS = 1
# the RNN cell to use, LSTM in this case
RNN_CELL = LSTM
# whether it's a bidirectional RNN
IS_BIDIRECTIONAL = False
# number of units (RNN_CELL ,nodes) in each layer
UNITS = 128
# dropout rate
DROPOUT = 0.4
### Training parameters
LOSS = "categorical_crossentropy"
```

```python
OPTIMIZER = "adam"
BATCH_SIZE = 64
EPOCHS = 6

def get_model_name(dataset_name):
    # construct the unique model name
    model_name = f"{dataset_name}-{RNN_CELL.__name__}-seq-{SEQUENCE_LENGTH}-em-{EMBEDDING_SIZE}-w-{N_WORDS}-layers-{N_LAYERS}-units-{UNITS}-opt-{OPTIMIZER}-BS-{BATCH_SIZE}-d-{DROPOUT}"
    if IS_BIDIRECTIONAL:
        # add 'bid' str if bidirectional
        model_name = "bid-" + model_name
    if OOV_TOKEN:
        # add 'oov' str if OOV token is specified
        model_name += "-oov"
    return model_name
```

utils.py

```python
from tqdm import tqdm

import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Dense, Dropout, LSTM, Embedding, Bidirectional
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_20newsgroups

from glob import glob
import random


def get_embedding_vectors(word_index, embedding_size=100):
    embedding_matrix = np.zeros((len(word_index) + 1, embedding_size))
```

```
    with open(f"data/glove.6B.{embedding_size}d.txt", encoding="utf8") as f:
        for line in tqdm(f, "Reading GloVe"):
            values = line.split()
            # get the word as the first word in the line
            word = values[0]
            if word in word_index:
                idx = word_index[word]
                # get the vectors as the remaining values in the line
                embedding_matrix[idx] = np.array(values[1:], dtype="float32")
    return embedding_matrix


def create_model(word_index, units=128, n_layers=1, cell=LSTM, bidirectional=False,
                embedding_size=100, sequence_length=100, dropout=0.3,
                loss="categorical_crossentropy", optimizer="adam",
                output_length=2):
    """
    Constructs a RNN model given its parameters
    """
    embedding_matrix = get_embedding_vectors(word_index, embedding_size)
    model = Sequential()
    # add the embedding layer
    model.add(Embedding(len(word_index) + 1,
              embedding_size,
              weights=[embedding_matrix],
              trainable=False,
              input_length=sequence_length))

    for i in range(n_layers):
        if i == n_layers - 1:
            # last layer
            if bidirectional:
                model.add(Bidirectional(cell(units, return_sequences=False)))
            else:
                model.add(cell(units, return_sequences=False))
        else:
            # first layer or hidden layers
            if bidirectional:
```

```
            model.add(Bidirectional(cell(units, return_sequences=True)))
        else:
            model.add(cell(units, return_sequences=True))
    model.add(Dropout(dropout))

    model.add(Dense(output_length, activation="softmax"))
    # compile the model
    model.compile(optimizer=optimizer, loss=loss, metrics=["accuracy"])
    return model



def load_imdb_data(num_words, sequence_length, test_size=0.25, oov_token=None):
    # read reviews
    reviews = []
    with open("data/reviews.txt") as f:
        for review in f:
            review = review.strip()
            reviews.append(review)

    labels = []
    with open("data/labels.txt") as f:
        for label in f:
            label = label.strip()
            labels.append(label)


    # tokenize the dataset corpus, delete uncommon words such as names, etc.
    tokenizer = Tokenizer(num_words=num_words, oov_token=oov_token)
    tokenizer.fit_on_texts(reviews)
    X = tokenizer.texts_to_sequences(reviews)

    X, y = np.array(X), np.array(labels)

    # pad sequences with 0's
    X = pad_sequences(X, maxlen=sequence_length)

    # convert labels to one-hot encoded
```

```
    y = to_categorical(y)

    # split data to training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=1)

    data = {}

    data["X_train"] = X_train
    data["X_test"]= X_test
    data["y_train"] = y_train
    data["y_test"] = y_test
    data["tokenizer"] = tokenizer
    data["int2label"] =  {0: "negative", 1: "positive"}
    data["label2int"] = {"negative": 0, "positive": 1}

    return data


def load_20_newsgroup_data(num_words, sequence_length, test_size=0.25, oov_token=None):
    # load the 20 news groups dataset
    # shuffling the data & removing each document's header, signature blocks and quotation blocks
    dataset = fetch_20newsgroups(subset="all", shuffle=True, remove=("headers", "footers", "quotes"))
    documents = dataset.data
    labels = dataset.target

    tokenizer = Tokenizer(num_words=num_words, oov_token=oov_token)
    tokenizer.fit_on_texts(documents)
    X = tokenizer.texts_to_sequences(documents)

    X, y = np.array(X), np.array(labels)

    # pad sequences with 0's
    X = pad_sequences(X, maxlen=sequence_length)

    # convert labels to one-hot encoded
    y = to_categorical(y)

    # split data to training and testing sets
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=1)

    data = {}

    data["X_train"] = X_train
    data["X_test"]= X_test
    data["y_train"] = y_train
    data["y_test"] = y_test
    data["tokenizer"] = tokenizer

    data["int2label"] = { i: label for i, label in enumerate(dataset.target_names) }
    data["label2int"] = { label: i for i, label in enumerate(dataset.target_names) }

    return data
```

sentiment_analysis.py

```
from tensorflow.keras.callbacks import TensorBoard

import os

from parameters import *
from utils import create_model, load_imdb_data

# create these folders if they does not exist
if not os.path.isdir("results"):
    os.mkdir("results")

if not os.path.isdir("logs"):
    os.mkdir("logs")

if not os.path.isdir("data"):
    os.mkdir("data")

# dataset name, IMDB movie reviews dataset
dataset_name = "imdb"
```

```python
# get the unique model name based on hyper parameters on parameters.py
model_name = get_model_name(dataset_name)

# load the data
data = load_imdb_data(N_WORDS, SEQUENCE_LENGTH, TEST_SIZE,
oov_token=OOV_TOKEN)

model = create_model(data["tokenizer"].word_index, units=UNITS, n_layers=N_LAYERS,
                    cell=RNN_CELL, bidirectional=IS_BIDIRECTIONAL,
embedding_size=EMBEDDING_SIZE,
                    sequence_length=SEQUENCE_LENGTH, dropout=DROPOUT,
                    loss=LOSS, optimizer=OPTIMIZER, output_length=data["y_train"][0].shape[0])

model.summary()

tensorboard = TensorBoard(log_dir=os.path.join("logs", model_name))

history = model.fit(data["X_train"], data["y_train"],
                    batch_size=BATCH_SIZE,
                    epochs=EPOCHS,
                    validation_data=(data["X_test"], data["y_test"]),
                    callbacks=[tensorboard],
                    verbose=1)


model.save(os.path.join("results", model_name) + ".h5")
```

20_news_group_classification.py

```python
from tensorflow.keras.callbacks import TensorBoard

import os

from parameters import *
from utils import create_model, load_20_newsgroup_data

# create these folders if they does not exist
```

```python
if not os.path.isdir("results"):
    os.mkdir("results")

if not os.path.isdir("logs"):
    os.mkdir("logs")

if not os.path.isdir("data"):
    os.mkdir("data")

# dataset name, IMDB movie reviews dataset
dataset_name = "20_news_group"
# get the unique model name based on hyper parameters on parameters.py
model_name = get_model_name(dataset_name)

# load the data
data = load_20_newsgroup_data(N_WORDS, SEQUENCE_LENGTH, TEST_SIZE,
oov_token=OOV_TOKEN)

model = create_model(data["tokenizer"].word_index, units=UNITS, n_layers=N_LAYERS,
                    cell=RNN_CELL, bidirectional=IS_BIDIRECTIONAL,
embedding_size=EMBEDDING_SIZE,
                    sequence_length=SEQUENCE_LENGTH, dropout=DROPOUT,
                    loss=LOSS, optimizer=OPTIMIZER, output_length=data["y_train"][0].shape[0])

model.summary()

tensorboard = TensorBoard(log_dir=os.path.join("logs", model_name))

history = model.fit(data["X_train"], data["y_train"],
                    batch_size=BATCH_SIZE,
                    epochs=EPOCHS,
                    validation_data=(data["X_test"], data["y_test"]),
                    callbacks=[tensorboard],
                    verbose=1)


model.save(os.path.join("results", model_name) + ".h5")
```

test.py

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

from parameters import *
from utils import create_model, load_20_newsgroup_data, load_imdb_data

import pickle
import os

# dataset name, IMDB movie reviews dataset
dataset_name = "imdb"
# get the unique model name based on hyper parameters on parameters.py
model_name = get_model_name(dataset_name)

# data = load_20_newsgroup_data(N_WORDS, SEQUENCE_LENGTH, TEST_SIZE,
oov_token=OOV_TOKEN)
data = load_imdb_data(N_WORDS, SEQUENCE_LENGTH, TEST_SIZE,
oov_token=OOV_TOKEN)

model = create_model(data["tokenizer"].word_index, units=UNITS, n_layers=N_LAYERS,
                    cell=RNN_CELL, bidirectional=IS_BIDIRECTIONAL,
embedding_size=EMBEDDING_SIZE,
                    sequence_length=SEQUENCE_LENGTH, dropout=DROPOUT,
                    loss=LOSS, optimizer=OPTIMIZER, output_length=data["y_train"][0].shape[0])

model.load_weights(os.path.join("results", f"{model_name}.h5"))


def get_predictions(text):
    sequence = data["tokenizer"].texts_to_sequences([text])
    # pad the sequences
    sequence = pad_sequences(sequence, maxlen=SEQUENCE_LENGTH)
    # get the prediction
    prediction = model.predict(sequence)[0]
```

```
    print("output vector:", prediction)
    return data["int2label"][np.argmax(prediction)]


while True:
    text = input("Enter your text: ")
    prediction = get_predictions(text)
    print("="*50)
    print("The class is:", prediction)
```

CHAPTER 14: Build a Text Generator using TensorFlow 2 and Keras in Python

Building a deep learning model to generate human readable text using Recurrent Neural Networks (RNNs) and LSTM with TensorFlow and Keras frameworks in Python.

Recurrent **N**eural **N**etworks (**RNN**s) are very powerful sequence models for classification problems. However, in this tutorial, we are doing to do something different, we will use RNNs as generative models, which means they can learn the sequences of a problem and then generate entirely a new sequence for the problem domain.

After reading this tutorial, you will learn how to build an **LSTM** model that can generate text (character by character) using TensorFlow and Keras in Python.

Note that the ultimate goal of this tutorial is to use TensorFlow and Keras to use LSTM models for text generation. If you want a better text generator, check this tutorial that uses transformer models to generate text.

In text generation, we show the model many training examples so it can learn a pattern between the input and output. Each input is a sequence of characters and the output is the next single character. For instance, say we want to train on the sentence "python is a great language", the input of the first sample is "python is a great langua" and output would be "g". The second sample input would be "ython is a great languag" and the output is "e", and so on, until we loop all over the dataset. We need to show the model as many examples as we can grab in order to make reasonable predictions.

Getting Started

Let's install the required dependencies for this tutorial:

```
pip3 install tensorflow==2.0.1 numpy requests tqdm
```

Importing everything:

```
import tensorflow as tf
import numpy as np
import os
import pickle
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
from string import punctuation
```

Preparing the Dataset

We are going to use a free downloadable book as the dataset for this tutorial: Alice’s Adventures in Wonderland by Lewis Carroll. But you can use any book/corpus you want.

These lines of code will download it and save it in a text file:

```
import requests
content =
requests.get("http://www.gutenberg.org/cache/epub/11/pg11.txt").text
open("data/wonderland.txt", "w", encoding="utf-8").write(content)
```

Just make sure you have a folder called "data" exist in your current directory.

Now let's define our parameters and try to clean this dataset:

```
sequence_length = 100
BATCH_SIZE = 128
```

```python
EPOCHS = 30
# dataset file path
FILE_PATH = "data/wonderland.txt"
BASENAME = os.path.basename(FILE_PATH)
# read the data
text = open(FILE_PATH, encoding="utf-8").read()
# remove caps, comment this code if you want uppercase characters as well
text = text.lower()
# remove punctuation
text = text.translate(str.maketrans("", "", punctuation))
```

The above code reduces our vocabulary for better and faster training by removing upper case characters and punctuations as well as replacing two consecutive newlines with just one. If you wish to keep commas, periods and colons, just define your own `punctuation` string variable.

Let's print some statistics about the dataset:

```python
# print some stats
n_chars = len(text)
vocab = ''.join(sorted(set(text)))
print("unique_chars:", vocab)
n_unique_chars = len(vocab)
print("Number of characters:", n_chars)
print("Number of unique characters:", n_unique_chars)
```

Output:

```
unique_chars:
0123456789abcdefghijklmnopqrstuvwxyz
```

```
Number of characters: 154207
Number of unique characters: 39
```

Now that we loaded and cleaned the dataset successfully, we need a way to convert these characters into integers, there are a lot of Keras and Scikit-Learn utilities out there for that, but we are going to make this manually in Python.

Since we have vocab as our vocabulary that contains all the unique characters of our dataset, we can make two dictionaries that map each character to an integer number and vice-versa:

```
# dictionary that converts characters to integers
char2int = {c: i for i, c in enumerate(vocab)}
# dictionary that converts integers to characters
int2char = {i: c for i, c in enumerate(vocab)}
```

Let's save them to a file (to retrieve them later in text generation):

```
# save these dictionaries for later generation
pickle.dump(char2int, open(f"{BASENAME}-char2int.pickle", "wb"))
pickle.dump(int2char, open(f"{BASENAME}-int2char.pickle", "wb"))
```

Now let's encode our dataset, in other words, we gonna convert each character into its corresponding integer number:

```
# convert all text into integers
encoded_text = np.array([char2int[c] for c in text])
```

Since we want to scale our code for larger datasets, we need to use tf.data API for efficient dataset handling, as a result, let's create a tf.data.Dataset object on this encoded_text array:

```
# construct tf.data.Dataset object
char_dataset = tf.data.Dataset.from_tensor_slices(encoded_text)
```

Awesome, now this char_dataset object has all the characters of this dataset; let's try to print the first characters:

```
# print first 5 characters
for char in char_dataset.take(8):
    print(char.numpy(), int2char[char.numpy()])
```

This will take the very first 8 characters and print them out along with their integer representation:

```
38
27 p
29 r
26 o
21 j
16 e
14 c
```

Great, now we need to construct our sequences, as mentioned earlier, we want each input sample to be a sequence of characters of the length sequence_length and the output of a single character that is the next one. Luckily for us, we have to use tf.data.Dataset's batch() method to gather characters together:

```
# build sequences by batching
sequences = char_dataset.batch(2*sequence_length + 1,
drop_remainder=True)

# print sequences
for sequence in sequences.take(2):
    print(''.join([int2char[i] for i in sequence.numpy()]))
```

As you may notice, I used 2*sequence_length +1 size of each sample, and you'll see why I did that very soon. Check the output:

```
project gutenbergs alices adventures in wonderland by lewis carroll

this ebook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever  you may  it give it away or

reuse it under the terms of the project gutenberg license included
with this ebook or online at wwwgutenbergorg
...
<SNIPPED>
```

You notice I've converted the integer sequences into normal text using `int2char` dictionary built earlier.

Now you know how each sample is represented, let's prepare our inputs and targets, we need a way to convert a single sample (sequence of characters) into multiple (input, target) samples. Fortunately, the flat_map() method is exactly what we need; it takes a callback function that loops over all our data samples:

```
def split_sample(sample):
```

```
    # example :
    # sequence_length is 10
    # sample is "python is a great pro" (21 length)
    # ds will equal to ('python is ', 'a') encoded as integers
    ds = tf.data.Dataset.from_tensors((sample[:sequence_length],
sample[sequence_length]))
    for i in range(1, (len(sample)-1) // 2):
        # first (input_, target) will be ('ython is a', ' ')
        # second (input_, target) will be ('thon is a ', 'g')
        # third (input_, target) will be ('hon is a g', 'r')
        # and so on
        input_ = sample[i: i+sequence_length]
        target = sample[i+sequence_length]
        # extend the dataset with these samples by concatenate() method
        other_ds = tf.data.Dataset.from_tensors((input_, target))
        ds = ds.concatenate(other_ds)
    return ds

# prepare inputs and targets
dataset = sequences.flat_map(split_sample)
```

To get a good understanding of how the above code works, let's take an example: Let's say we have a sequence length of 10 (too small but good for explanation), the `sample` argument is a sequence of 21 characters (remember the 2*sequence_length+1) encoded in integers, for convenience, let's imagine it isn't encoded, say it's "python is a great pro".

Now the first data sample we going to generate would be the following tuple of inputs and targets ('python is ', 'a'), the second is ('ython is a', ' '), the third is ('thon is a ', 'g') and so on. We do that on all samples, in the end, we'll see that we dramatically increased the number of training samples. We've used the ds.concatenate() method to add these samples together.

After we constructed our samples, let's one-hot encode both the inputs and the labels (targets):

```
def one_hot_samples(input_, target):
    # onehot encode the inputs and the targets
    # Example:
    # if character 'd' is encoded as 3 and n_unique_chars = 5
    # result should be the vector: [0, 0, 0, 1, 0], since 'd' is the 4th character
    return tf.one_hot(input_, n_unique_chars), tf.one_hot(target,
n_unique_chars)

dataset = dataset.map(one_hot_samples)
```

We've used the convenient map() method to one-hot encode each sample on our dataset, tf.one_hot() method does what we expect. Let's try to print the first two data samples along with their shapes:

```
# print first 2 samples
for element in dataset.take(2):
    print("Input:", ''.join([int2char[np.argmax(char_vector)] for char_vector in
element[0].numpy()]))
    print("Target:", int2char[np.argmax(element[1].numpy())])
    print("Input shape:", element[0].shape)
    print("Target shape:", element[1].shape)
    print("="*50, "\n")
```

Here is the output of the second element:

```
Input: project gutenbergs alices adventures in wonderland by lewis carroll
```

```

this ebook is for the use of an
Target: y
Input shape: (100, 39)
Target shape: (39,)
```

So each input element has the shape of (sequence length, vocabulary size), in this case, there are 39 unique characters and 100 is the sequence length. The shape of the output is a one-dimensional vector that is one-hot encoded.

***Note**: If you're using a different dataset and/or using another character filtering mechanism, you'll see a different vocabulary size, each problem has its own domain. For instance, I also used this to generate Python code, it has 92 unique characters, that's because I should allow some punctuations that are necessary for Python code.*

Finally, we repeat, shuffle and batch our dataset:

```
# repeat, shuffle and batch the dataset
ds = dataset.repeat().shuffle(1024).batch(BATCH_SIZE,
drop_remainder=True)
```

We set the optional `drop_remainder` to `True` so we can eliminate the remaining samples that have less size than `BATCH_SIZE`.

Building the Model

Now let's build the model, it has basically two LSTM layers with an arbitrary number of 128 LSTM units. Try to experiment with different model architectures, you're free to do whatever you want!

The output layer is a fully-connected layer with 39 units where each neuron corresponds to a character (probability of the occurrence of each character).

```
model = Sequential([
    LSTM(256, input_shape=(sequence_length, n_unique_chars),
return_sequences=True),
    Dropout(0.3),
    LSTM(256),
    Dense(n_unique_chars, activation="softmax"),
])
```

We're using [Adam optimizer] here, I suggest you experiment with different optimizers.

After we've built our model, let's print the summary and compile it:

```
# define the model path
model_weights_path = f"results/{BASENAME}-{sequence_length}.h5"
model.summary()
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=
["accuracy"])
```

Training the Model

Let's train the model now:

```
# make results folder if does not exist yet
if not os.path.isdir("results"):
    os.mkdir("results")
# train the model
model.fit(ds, steps_per_epoch=(len(encoded_text) - sequence_length) //
BATCH_SIZE, epochs=EPOCHS)
```

```
# save the model
model.save(model_weights_path)
```

We fed the Dataset object that we prepared earlier, and since the `model` object has no idea on many samples are there in the dataset, we specified `steps_per_epoch` parameter, which is set to the number of training samples divided by the batch size.

After running the above code, it should start training, which gonna look something like this:

```
Train for 6473 steps
...
<SNIPPED>

Epoch 29/30
6473/6473 [==============================] - 486s 75ms/step - loss: 0.8728 - accuracy: 0.7509
Epoch 30/30
2576/6473 [===========>..................] - ETA: 4:56 - loss: 0.8063 - accuracy: 0.7678
```

This will take few hours, depending on your hardware, try increasing batch_size to 256 for faster training.

After the training is over, a new file should appear in the results folder, that is, the model trained weights.

Generating New Text

Here comes the fun part, now we have successfully built and trained the model, how can we generate new text?

Let's import the necessary modules (If you're on a single notebook, you don't have to do that):

```
import numpy as np
import pickle
import tqdm
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Activation
import os

sequence_length = 100
# dataset file path
FILE_PATH = "data/wonderland.txt"
# FILE_PATH = "data/python_code.py"
BASENAME = os.path.basename(FILE_PATH)
```

We need a sample text to start generating. This will depend on your problem, you can take sentences from the training data in which it will perform better, but I'll try to produce a new chapter of this book:

```
seed = "chapter xiii"
```

Let's load the dictionaries that map each integer to a character and vise-versa that we saved before in the data preparation phase:

```
# load vocab dictionaries
char2int = pickle.load(open(f"{BASENAME}-char2int.pickle", "rb"))
int2char = pickle.load(open(f"{BASENAME}-int2char.pickle", "rb"))
vocab_size = len(char2int)
```

Building the model again:

```
# building the model
model = Sequential([
    LSTM(256, input_shape=(sequence_length, vocab_size),
return_sequences=True),
    Dropout(0.3),
    LSTM(256),
    Dense(vocab_size, activation="softmax"),
])
```

Now we need to load the optimal set of model weights:

```
# load the optimal weights
model.load_weights(f"results/{BASENAME}-{sequence_length}.h5")
```

Let's start generating:

```
s = seed
n_chars = 400
# generate 400 characters
generated = ""
for i in tqdm.tqdm(range(n_chars), "Generating text"):
    # make the input sequence
    X = np.zeros((1, sequence_length, vocab_size))
    for t, char in enumerate(seed):
        X[0, (sequence_length - len(seed)) + t, char2int[char]] = 1
    # predict the next character
    predicted = model.predict(X, verbose=0)[0]
```

```
    # converting the vector to an integer
    next_index = np.argmax(predicted)
    # converting the integer to a character
    next_char = int2char[next_index]
    # add the character to results
    generated += next_char
    # shift seed and the predicted character
    seed = seed[1:] + next_char

print("Seed:", s)
print("Generated text:")
print(generated)
```

All we are doing here is starting with a seed text, constructing the input sequence, and then predicting the next character. After that, we shift the input sequence by removing the first character and adding the last character predicted. This gives us a slightly changed sequence of inputs that still has a length equal to the size of our sequence length.

We then feed this updated input sequence into the model to predict another character. Repeating this process N times will generate a text with N characters.

Here is an interesting text generated:

```
Seed: chapter xiii
Generated Text:
ded of and alice as it go on and the court
well you wont you wouldn thing
there was not a long to growing anxiously any only a low every cant
go on a litter which was proves of any only here and the things and the mort meding and the mort and alice was the things said to herself i cant remeran as
```

```
if i can repeat eften to alice any of great offf its archive of and alice and a cancur as the mo
```

That is clearly English! But as you may notice, most of the sentences don't make any sense, that is due to many reasons. One of the main reasons is that the dataset is trained only on very few samples. Also, the model architecture isn't optimal, other state-of-the-art architectures (such as GPT-2 and BERT) tend to outperform this one drastically.

Note though, this is not limited to English text, you can use whatever type of text you want. In fact, you can even generate Python code once you have enough lines of code.

Conclusion

Great, we are done. Now you know how to:

- Make RNNs in TensorFlow and Keras as generative models.
- Cleaning text and building TensorFlow input pipelines using tf.data API.
- Training LSTM network on text sequences.
- Tuning the performance of the model.

In order to further improve the model, you can:

- Reduce the vocabulary size by removing rare characters.
- Add more LSTM and Dropout layers with more LSTM units, or even add Bidirectional layers.
- Tweak some hyperparameters such as batch size, optimizer, and even `sequence_length`, and see which works best.
- Train on more epochs.
- Use a larger dataset.

SourceCode:

train.py

```
import tensorflow as tf
import numpy as np
import os
import pickle
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint
from string import punctuation

sequence_length = 100
BATCH_SIZE = 128
EPOCHS = 3
# dataset file path
FILE_PATH = "data/wonderland.txt"
# FILE_PATH = "data/python_code.py"
BASENAME = os.path.basename(FILE_PATH)

# commented because already downloaded
# import requests
# content = requests.get("http://www.gutenberg.org/cache/epub/11/pg11.txt").text
# open("data/wonderland.txt", "w", encoding="utf-8").write(content)

# read the data
text = open(FILE_PATH, encoding="utf-8").read()
# remove caps, comment this code if you want uppercase characters as well
text = text.lower()
# remove punctuation
text = text.translate(str.maketrans("", "", punctuation))
# print some stats
n_chars = len(text)
vocab = ''.join(sorted(set(text)))
print("unique_chars:", vocab)
```

```python
n_unique_chars = len(vocab)
print("Number of characters:", n_chars)
print("Number of unique characters:", n_unique_chars)

# dictionary that converts characters to integers
char2int = {c: i for i, c in enumerate(vocab)}
# dictionary that converts integers to characters
int2char = {i: c for i, c in enumerate(vocab)}

# save these dictionaries for later generation
pickle.dump(char2int, open(f"{BASENAME}-char2int.pickle", "wb"))
pickle.dump(int2char, open(f"{BASENAME}-int2char.pickle", "wb"))

# convert all text into integers
encoded_text = np.array([char2int[c] for c in text])
# construct tf.data.Dataset object
char_dataset = tf.data.Dataset.from_tensor_slices(encoded_text)
# print first 5 characters
for char in char_dataset.take(8):
    print(char.numpy(), int2char[char.numpy()])

# build sequences by batching
sequences = char_dataset.batch(2*sequence_length + 1, drop_remainder=True)

# print sequences
for sequence in sequences.take(2):
    print(''.join([int2char[i] for i in sequence.numpy()]))

def split_sample(sample):
    # example :
    # sequence_length is 10
    # sample is "python is a great pro" (21 length)
    # ds will equal to ('python is ', 'a') encoded as integers
    ds = tf.data.Dataset.from_tensors((sample[:sequence_length], sample[sequence_length]))
    for i in range(1, (len(sample)-1) // 2):
        # first (input_, target) will be ('ython is a', ' ')
        # second (input_, target) will be ('thon is a ', 'g')
        # third (input_, target) will be ('hon is a g', 'r')
```

```
        # and so on
        input_ = sample[i: i+sequence_length]
        target = sample[i+sequence_length]
        # extend the dataset with these samples by concatenate() method
        other_ds = tf.data.Dataset.from_tensors((input_, target))
        ds = ds.concatenate(other_ds)
    return ds

# prepare inputs and targets
dataset = sequences.flat_map(split_sample)

def one_hot_samples(input_, target):
    # onehot encode the inputs and the targets
    # Example:
    # if character 'd' is encoded as 3 and n_unique_chars = 5
    # result should be the vector: [0, 0, 0, 1, 0], since 'd' is the 4th character
    return tf.one_hot(input_, n_unique_chars), tf.one_hot(target, n_unique_chars)


dataset = dataset.map(one_hot_samples)
# print first 2 samples
for element in dataset.take(2):
    print("Input:", ''.join([int2char[np.argmax(char_vector)] for char_vector in element[0].numpy()]))
    print("Target:", int2char[np.argmax(element[1].numpy())])
    print("Input shape:", element[0].shape)
    print("Target shape:", element[1].shape)
    print("="*50, "\n")

# repeat, shuffle and batch the dataset
ds = dataset.repeat().shuffle(1024).batch(BATCH_SIZE, drop_remainder=True)

# building the model
# model = Sequential([
#     LSTM(128, input_shape=(sequence_length, n_unique_chars)),
#     Dense(n_unique_chars, activation="softmax"),
# ])

# a better model (slower to train obviously)
```

```
model = Sequential([
    LSTM(256, input_shape=(sequence_length, n_unique_chars), return_sequences=True),
    Dropout(0.3),
    LSTM(256),
    Dense(n_unique_chars, activation="softmax"),
])

model.load_weights(f"results/{BASENAME}-{sequence_length}.h5")

model.summary()
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

if not os.path.isdir("results"):
    os.mkdir("results")

# checkpoint = ModelCheckpoint("results/{}-{loss:.2f}.h5".format(BASENAME), verbose=1)

# train the model
model.fit(ds, steps_per_epoch=(len(encoded_text) - sequence_length) // BATCH_SIZE,
epochs=EPOCHS)
# save the model
model.save(f"results/{BASENAME}-{sequence_length}.h5")
```

generate.py

```
import numpy as np
import pickle
import tqdm
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Activation
import os

sequence_length = 100
# dataset file path
FILE_PATH = "data/wonderland.txt"
# FILE_PATH = "data/python_code.py"
BASENAME = os.path.basename(FILE_PATH)
```

```
# load vocab dictionaries
char2int = pickle.load(open(f"{BASENAME}-char2int.pickle", "rb"))
int2char = pickle.load(open(f"{BASENAME}-int2char.pickle", "rb"))

sequence_length = 100
vocab_size = len(char2int)

# building the model
model = Sequential([
    LSTM(256, input_shape=(sequence_length, vocab_size), return_sequences=True),
    Dropout(0.3),
    LSTM(256),
    Dense(vocab_size, activation="softmax"),
])

# load the optimal weights
model.load_weights(f"results/{BASENAME}-{sequence_length}.h5")
# specify the feed to first characters to generate
seed = "alice is pretty"
s = seed
n_chars = 400
# generate 400 characters
generated = ""
for i in tqdm.tqdm(range(n_chars), "Generating text"):
    # make the input sequence
    X = np.zeros((1, sequence_length, vocab_size))
    for t, char in enumerate(seed):
        X[0, (sequence_length - len(seed)) + t, char2int[char]] = 1
    # predict the next character
    predicted = model.predict(X, verbose=0)[0]
    # converting the vector to an integer
    next_index = np.argmax(predicted)
    # converting the integer to a character
    next_char = int2char[next_index]
    # add the character to results
    generated += next_char
    # shift seed and the predicted character
    seed = seed[1:] + next_char
```

```python

print("Seed:", s)
print("Generated text:")
print(generated)
```

CHAPTER 15: Build a Spam Classifier using Keras and TensorFlow in Python

Classifying emails (spam or not spam) with GloVe embedding vectors and RNN/LSTM units using Keras and TensorFlow in Python.

Email spam or junk email is unsolicited, unavoidable, and repetitive messages sent in email. Email spam has grown since the early 1990s, and by 2014, it was estimated that it made up around 90% of email messages sent.

Since we all have the problem of spam emails filling our inboxes, in this tutorial, we gonna build a model in Keras that can distinguish between spam and legitimate emails.

Table of content:

1. Installing and Importing Dependencies

We first need to install some dependencies:

```
pip3 install sklearn tqdm numpy tensorflow
```

Now open up an interactive shell or a Jupyter notebook and import:

```
import time
import pickle
import tensorflow as tf
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    # only use GPU memory that we need, not allocate all the GPU memory
    tf.config.experimental.set_memory_growth(gpus[0], enable=True)

import tqdm
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Embedding, LSTM, Dropout, Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.metrics import Recall, Precision
```

Let's define some hyper-parameters:

```
SEQUENCE_LENGTH = 100 # the length of all sequences (number of words per sample)
EMBEDDING_SIZE = 100  # Using 100-Dimensional GloVe embedding vectors
```

```
TEST_SIZE = 0.25 # ratio of testing set

BATCH_SIZE = 64
EPOCHS = 10 # number of epochs

label2int = {"ham": 0, "spam": 1}
int2label = {0: "ham", 1: "spam"}
```

Don't worry if you are not sure what these parameters mean, we'll talk about them later when we construct our model.

2. Loading the Dataset

The dataset we gonna use is SMS Spam Collection Dataset, download, extract and put it in a folder called "data", let's define the function that loads it:

```
def load_data():
    """
    Loads SMS Spam Collection dataset
    """
    texts, labels = [], []
    with open("data/SMSSpamCollection") as f:
        for line in f:
            split = line.split()
            labels.append(split[0].strip())
            texts.append(' '.join(split[1:]).strip())
    return texts, labels
```

The dataset is in a single file, each line corresponds to a data sample, the first word is the label and the rest is the actual email content, that's why we are grabbing labels as split[0] and the content as split[1:].

Calling the function:

```
# load the data
X, y = load_data()
```

3. Preparing the Dataset

Now, we need a way to vectorize the text corpus by turning each text into a sequence of integers, you're now may be wondering why we need to turn the text into a sequence of integers. Well, remember we are going to feed the text into a neural network, a neural network only understands numbers. More precisely, a fixed-length sequence of integers.

But before we do all of that, we need to clean this corpus by removing punctuations, lowercase all characters, etc. Luckily for us, Keras has a built-in class `Tokenizer` from the `tensorflow.keras.preprocessing.text` module, that does all that in few lines of code:

```
# Text tokenization
# vectorizing text, turning each text into sequence of integers
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X)
# lets dump it to a file, so we can use it in testing
pickle.dump(tokenizer, open("results/tokenizer.pickle", "wb"))
# convert to sequence of integers
X = tokenizer.texts_to_sequences(X)
```

Let's try to print the first sample:

```
In [4]: print(X[0])
```

```
[49, 472, 4436, 843, 756, 659, 64, 8, 1328, 87, 123, 352, 1329, 148, 2996,
1330, 67, 58, 4437, 144]
```

A bunch of numbers, each integer corresponds to a word in the vocabulary, that's what the neural network needs anyway. However, the samples don't have the same length, we need a way to have a fixed-length sequence.

As a result, we're using the `pad_sequences()` function from the `tensorflow.keras.preprocessing.sequence` module that pad sequences at the beginning of each sequence with zeros:

```
# convert to numpy arrays
X = np.array(X)
y = np.array(y)
# pad sequences at the beginning of each sequence with 0's
# for example if SEQUENCE_LENGTH=4:
# [[5, 3, 2], [5, 1, 2, 3], [3, 4]]
# will be transformed to:
# [[0, 5, 3, 2], [5, 1, 2, 3], [0, 0, 3, 4]]
X = pad_sequences(X, maxlen=SEQUENCE_LENGTH)
```

As you may remember, we set `SEQUENCE_LENGTH` to 100, in this way, all sequences have a length of 100. Let's print how each sentence is converted to:

```
In [6]: print(X[0])
[  0    0    0    0    0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

```
   0    0    0    0    0    0    0    0    0    0   49  471 4435  842
  755  658   64    8 1327   88  123  351 1328  148 2996 1329   67   58
 4436  144]
```

Now our labels are text also, but we gonna make a different approach here, since the labels are only "spam" and "ham", we need to one-hot encode them:

```
# One Hot encoding labels
# [spam, ham, spam, ham, ham] will be converted to:
# [1, 0, 1, 0, 1] and then to:
# [[0, 1], [1, 0], [0, 1], [1, 0], [0, 1]]

y = [ label2int[label] for label in y ]
y = to_categorical(y)
```

We used `keras.utils.to_categorial()` here, which does what its name suggests, let's try to print the first sample of the labels:

```
In [7]: print(y[0])
[1.0, 0.0]
```

That means the first sample is ham.

Next, let's shuffle and split training and testing data:

```
# split and shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE, random_state=7)
# print our data shapes
```

```
print("X_train.shape:", X_train.shape)
print("X_test.shape:", X_test.shape)
print("y_train.shape:", y_train.shape)
print("y_test.shape:", y_test.shape)
```

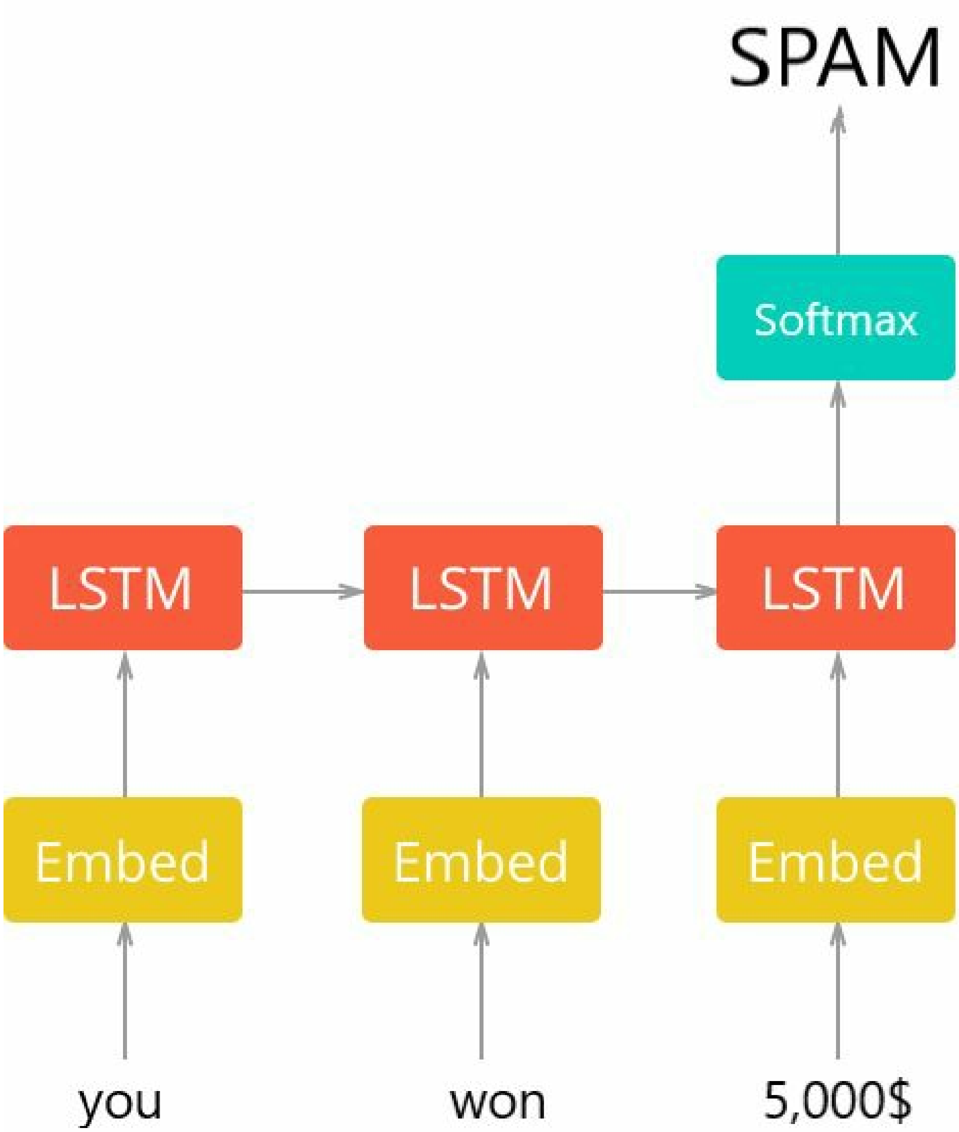
Cell output:

```
X_train.shape: (4180, 100)
X_test.shape: (1394, 100)
y_train.shape: (4180, 2)
y_test.shape: (1394, 2)
```

As you can see, we have a total of 4180 training samples and 1494 validation samples.

4. Building the Model

Now we are ready to build our model, the general architecture is as shown in the following image:



The first layer is a pre-trained embedding layer that maps each word to an N-dimensional vector of real numbers (the EMBEDDING_SIZE corresponds to the size of this vector, in this case, 100). Two words that have similar meanings tend to have very close vectors.

The second layer is a recurrent neural network with LSTM units. Finally, the output layer is 2 neurons each corresponds to "spam" or "ham" with a softmax activation function.

Let's start by writing a function to load the pre-trained embedding vectors:

```
def get_embedding_vectors(tokenizer, dim=100):
```

```
    embedding_index = {}
    with open(f"data/glove.6B.{dim}d.txt", encoding='utf8') as f:
        for line in tqdm.tqdm(f, "Reading GloVe"):
            values = line.split()
            word = values[0]
            vectors = np.asarray(values[1:], dtype='float32')
            embedding_index[word] = vectors

    word_index = tokenizer.word_index
    embedding_matrix = np.zeros((len(word_index)+1, dim))
    for word, i in word_index.items():
        embedding_vector = embedding_index.get(word)
        if embedding_vector is not None:
            # words not found will be 0s
            embedding_matrix[i] = embedding_vector

    return embedding_matrix
```

Note: In order to run this function properly, you need to download [GloVe](), extract and put in the "data" folder, we will use the 100-dimensional vectors here.

Let's define the function that builds the model:

```
def get_model(tokenizer, lstm_units):
    """
    Constructs the model,
    Embedding vectors => LSTM => 2 output Fully-Connected neurons with softmax activation
    """
```

```
    # get the GloVe embedding vectors
    embedding_matrix = get_embedding_vectors(tokenizer)
    model = Sequential()
    model.add(Embedding(len(tokenizer.word_index)+1,
              EMBEDDING_SIZE,
              weights=[embedding_matrix],
              trainable=False,
              input_length=SEQUENCE_LENGTH))

    model.add(LSTM(lstm_units, recurrent_dropout=0.2))
    model.add(Dropout(0.3))
    model.add(Dense(2, activation="softmax"))
    # compile as rmsprop optimizer
    # aswell as with recall metric
    model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
                  metrics=["accuracy", keras_metrics.precision(),
keras_metrics.recall()])
    model.summary()
    return model
```

The above function constructs the whole model, we loaded the pre-trained embedding vectors to the `Embedding` layer, and set `trainable=False`, this will freeze the embedding weights during the training process.

After we add the RNN layer, we added a 30% dropout chance, this will freeze 30% of neurons in the previous layer in each iteration which will help us reduce overfitting.

Note that accuracy isn't enough for determining whether the model is doing great, that is because this dataset is unbalanced, only a few samples are spam. As a result, we will use precision and recall metrics.

Let's call the function:

```
# constructs the model with 128 LSTM units
model = get_model(tokenizer=tokenizer, lstm_units=128)
```

5. Training the Model

We are almost there, we gonna need to train this model with the data we just loaded:

```
# initialize our ModelCheckpoint and TensorBoard callbacks
# model checkpoint for saving best weights
model_checkpoint = ModelCheckpoint("results/spam_classifier_{val_loss:.2f}.h5", save_best_only=True,
                                    verbose=1)
# for better visualization
tensorboard = TensorBoard(f"logs/spam_classifier_{time.time()}")
# train the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          batch_size=BATCH_SIZE, epochs=EPOCHS,
          callbacks=[tensorboard, model_checkpoint],
          verbose=1)
```

The training has started:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 100)          901300
_________________________________________________________________
```

```
lstm_1 (LSTM)              (None, 128)             117248

_________________________________________________________________
dropout_1 (Dropout)        (None, 128)             0

_________________________________________________________________
dense_1 (Dense)            (None, 2)               258
=================================================================
Total params: 1,018,806
Trainable params: 117,506
Non-trainable params: 901,300

_________________________________________________________________
Train on 4180 samples, validate on 1394 samples
Epoch 1/10
66/66 [==============================] - 86s 1s/step - loss: 0.2315 - accuracy: 0.8980 - precision: 0.8980 - recall: 0.8980 - val_loss: 0.1192 - val_accuracy: 0.9555 - val_precision: 0.9555 - val_recall: 0.9555

Epoch 00001: val_loss improved from inf to 0.11920, saving model to results\spam_classifier_0.12.h5
Epoch 2/10
66/66 [==============================] - 87s 1s/step - loss: 0.0824 - accuracy: 0.9726 - precision: 0.9726 - recall: 0.9726 - val_loss: 0.0769 - val_accuracy: 0.9749 - val_precision: 0.9749 - val_recall: 0.9749

Epoch 00002: val_loss improved from 0.11920 to 0.07687, saving model to results\spam_classifier_0.08.h5
```

The training is finished:

```
Epoch 10/10
66/66 [==============================] - 89s 1s/step - loss: 0.0216 - accuracy: 0.9932 - precision: 0.9932 - recall: 0.9932 - val_loss: 0.0546 -
```

```
val_accuracy: 0.9842 - val_precision: 0.9842 - val_recall: 0.9842

Epoch 00010: val_loss improved from 0.06224 to 0.05463, saving model to results\spam_classifier_0.05.h5
```

6. Evaluating the Model

Let's evaluate our model:

```python
# get the loss and metrics
result = model.evaluate(X_test, y_test)
# extract those
loss = result[0]
accuracy = result[1]
precision = result[2]
recall = result[3]

print(f"[+] Accuracy: {accuracy*100:.2f}%")
print(f"[+] Precision:   {precision*100:.2f}%")
print(f"[+] Recall:   {recall*100:.2f}%")
```

Output:

```
1394/1394 [==============================] - 1s 569us/step
[+] Accuracy: 98.21%
[+] Precision:   99.16%
[+] Recall:   98.75%
```

Here are what each metric means:

- **Accuracy**: Percentage of predictions that were correct.
- **Recall**: Percentage of spam emails that were predicted correctly.
- **Precision**: Percentage of emails classified as spam that was actually spam.

Great! let's test this out:

```
def get_predictions(text):
    sequence = tokenizer.texts_to_sequences([text])
    # pad the sequence
    sequence = pad_sequences(sequence, maxlen=SEQUENCE_LENGTH)
    # get the prediction
    prediction = model.predict(sequence)[0]
    # one-hot encoded vector, revert using np.argmax
    return int2label[np.argmax(prediction)]
```

Let's fake a spam email:

```
text = "You won a prize of 1,000$, click here to claim!"
get_predictions(text)
```

Output:

```
spam
```

Okay, let's try to be legitimate:

```
text = "Hi man, I was wondering if we can meet tomorrow."
print(get_predictions(text))
```

Output:

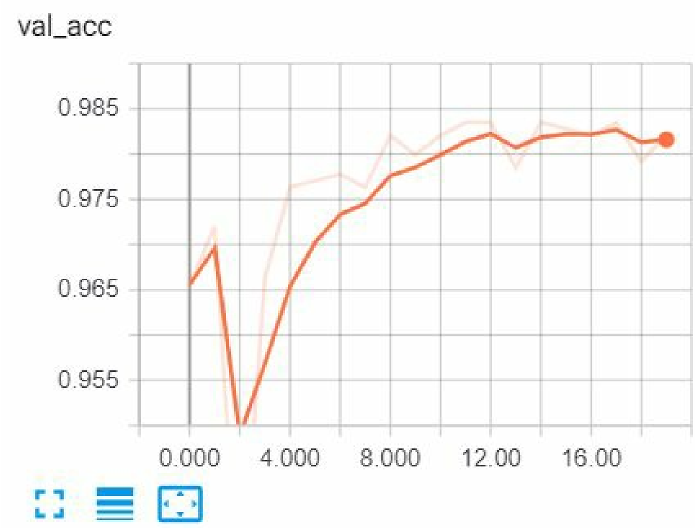
```
ham
```

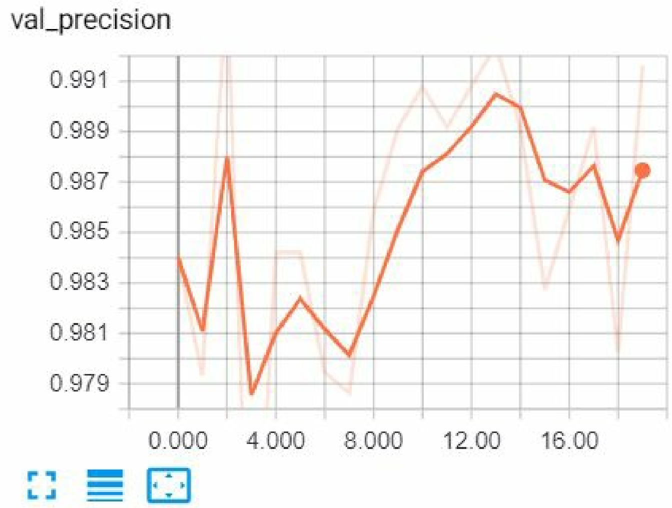
Awesome! This approach is the current state-of-the-art, try to tune the training and model parameters and see if you can improve it.

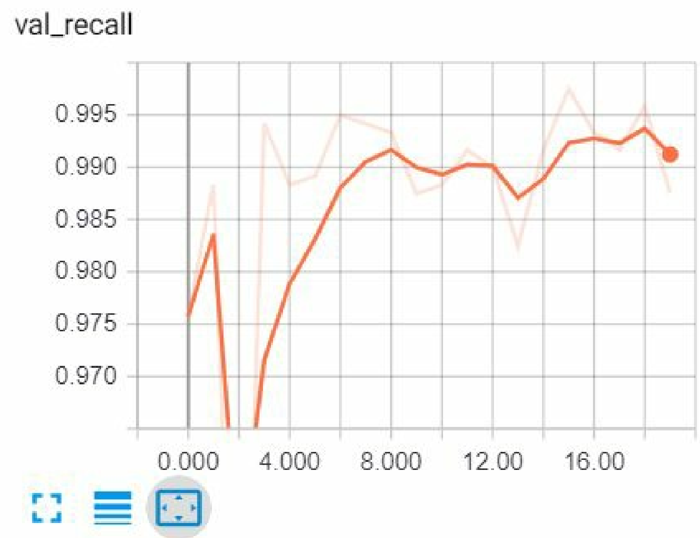
To see various metrics during training, we need to go to tensorboard by typing in cmd or terminal:

```
tensorboard --logdir="logs"
```

Go to the browser and type "localhost:6006" and go to various metrics, here is my result:







Finally, if you want to expand your skills in machine learning (or even if you're a beginner), I would suggest you take the Machine Learning and Deep Learning specialization courses. Good luck!

SourceCode:

utils.py

```python
import tqdm
import numpy as np
from tensorflow.keras.layers import Embedding, LSTM, Dropout, Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.metrics import Recall, Precision

SEQUENCE_LENGTH = 100 # the length of all sequences (number of words per sample)
EMBEDDING_SIZE = 100  # Using 100-Dimensional GloVe embedding vectors
TEST_SIZE = 0.25 # ratio of testing set

BATCH_SIZE = 64
EPOCHS = 20 # number of epochs

label2int = {"ham": 0, "spam": 1}
int2label = {0: "ham", 1: "spam"}

def get_embedding_vectors(tokenizer, dim=100):
    embedding_index = {}
    with open(f"data/glove.6B.{dim}d.txt", encoding='utf8') as f:
        for line in tqdm.tqdm(f, "Reading GloVe"):
            values = line.split()
            word = values[0]
            vectors = np.asarray(values[1:], dtype='float32')
            embedding_index[word] = vectors
    word_index = tokenizer.word_index
    # we do +1 because Tokenizer() starts from 1
    embedding_matrix = np.zeros((len(word_index)+1, dim))
    for word, i in word_index.items():
        embedding_vector = embedding_index.get(word)
        if embedding_vector is not None:
            # words not found will be 0s
            embedding_matrix[i] = embedding_vector
    return embedding_matrix


def get_model(tokenizer, lstm_units):
```

```
    """
    Constructs the model,
    Embedding vectors => LSTM => 2 output Fully-Connected neurons with softmax activation
    """
    # get the GloVe embedding vectors
    embedding_matrix = get_embedding_vectors(tokenizer)
    model = Sequential()
    model.add(Embedding(len(tokenizer.word_index)+1,
              EMBEDDING_SIZE,
              weights=[embedding_matrix],
              trainable=False,
              input_length=SEQUENCE_LENGTH))

    model.add(LSTM(lstm_units, recurrent_dropout=0.2))
    model.add(Dropout(0.3))
    model.add(Dense(2, activation="softmax"))
    # compile as rmsprop optimizer
    # aswell as with recall metric
    model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
                  metrics=["accuracy", Precision(), Recall()])
    model.summary()
    return model
```

spam_classifier.py

```
import tensorflow as tf
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    # only use GPU memory that we need, not allocate all the GPU memory
    tf.config.experimental.set_memory_growth(gpus[0], enable=True)

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard
from sklearn.model_selection import train_test_split
```

```python
import time
import numpy as np
import pickle

from utils import get_model, SEQUENCE_LENGTH, TEST_SIZE
from utils import BATCH_SIZE, EPOCHS, label2int


def load_data():
    """
    Loads SMS Spam Collection dataset
    """
    texts, labels = [], []
    with open("data/SMSSpamCollection") as f:
        for line in f:
            split = line.split()
            labels.append(split[0].strip())
            texts.append(' '.join(split[1:]).strip())
    return texts, labels


# load the data
X, y = load_data()

# Text tokenization
# vectorizing text, turning each text into sequence of integers
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X)
# lets dump it to a file, so we can use it in testing
pickle.dump(tokenizer, open("results/tokenizer.pickle", "wb"))

# convert to sequence of integers
X = tokenizer.texts_to_sequences(X)
print(X[0])
# convert to numpy arrays
X = np.array(X)
y = np.array(y)
# pad sequences at the beginning of each sequence with 0's
```

```
# for example if SEQUENCE_LENGTH=4:
# [[5, 3, 2], [5, 1, 2, 3], [3, 4]]
# will be transformed to:
# [[0, 5, 3, 2], [5, 1, 2, 3], [0, 0, 3, 4]]
X = pad_sequences(X, maxlen=SEQUENCE_LENGTH)
print(X[0])
# One Hot encoding labels
# [spam, ham, spam, ham, ham] will be converted to:
# [1, 0, 1, 0, 1] and then to:
# [[0, 1], [1, 0], [0, 1], [1, 0], [0, 1]]

y = [ label2int[label] for label in y ]
y = to_categorical(y)
print(y[0])

# split and shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE, random_state=7)
# print our data shapes
print("X_train.shape:", X_train.shape)
print("X_test.shape:", X_test.shape)
print("y_train.shape:", y_train.shape)
print("y_test.shape:", y_test.shape)
# constructs the model with 128 LSTM units
model = get_model(tokenizer=tokenizer, lstm_units=128)

# initialize our ModelCheckpoint and TensorBoard callbacks
# model checkpoint for saving best weights
model_checkpoint = ModelCheckpoint("results/spam_classifier_{val_loss:.2f}.h5",
save_best_only=True,
                                    verbose=1)
# for better visualization
tensorboard = TensorBoard(f"logs/spam_classifier_{time.time()}")

# train the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          batch_size=BATCH_SIZE, epochs=EPOCHS,
          callbacks=[tensorboard, model_checkpoint],
          verbose=1)
```

```python

# get the loss and metrics
result = model.evaluate(X_test, y_test)
# extract those
loss = result[0]
accuracy = result[1]
precision = result[2]
recall = result[3]

print(f"[+] Accuracy: {accuracy*100:.2f}%")
print(f"[+] Precision:   {precision*100:.2f}%")
print(f"[+] Recall:   {recall*100:.2f}%")



```

test.py

```python
import tensorflow as tf
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    # only use GPU memory that we need, not allocate all the GPU memory
    tf.config.experimental.set_memory_growth(gpus[0], enable=True)
from utils import get_model, int2label
from tensorflow.keras.preprocessing.sequence import pad_sequences

import pickle
import numpy as np

SEQUENCE_LENGTH = 100

# get the tokenizer
tokenizer = pickle.load(open("results/tokenizer.pickle", "rb"))

model = get_model(tokenizer, 128)
# change to the model name in results folder
```

```
model.load_weights("results/spam_classifier_0.06.h5")

def get_predictions(text):
    sequence = tokenizer.texts_to_sequences([text])
    # pad the sequence
    sequence = pad_sequences(sequence, maxlen=SEQUENCE_LENGTH)
    # get the prediction
    prediction = model.predict(sequence)[0]
    # one-hot encoded vector, revert using np.argmax
    return int2label[np.argmax(prediction)]


while True:
    text = input("Enter the mail:")
    # convert to sequences
    print(get_predictions(text))
```

Summary

This book is dedicated to the readers who take time to write me each day. Every morning I’m greeted by various emails — some with requests, a few with complaints, and then there are the very few that just say thank you. All these emails encourage and challenge me as an author — to better both my books and myself.

Thank you!