Multi-Purpose NLP Models:

Multi-purpose models are the talk of the NLP world. These models power the NLP applications we are excited about –

- machine translation,question answering systems, chatbots, sentiment analysis, etc.
- A core component of these multi-purpose NLP models is the **concept of language modelling.**

In simple terms, the aim of a language model is to predict the next word or character in a sequence.

We' ll understand this as we look at each model here.

If you' re a NLP enthusiast, you' re going to love this section. Now, let' s dive into 5 state-of-the-art multi-purpose NLP model frameworks.

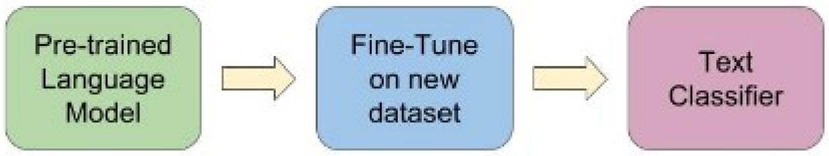
ULMFiT:

ULMFiT was proposed and designed by fast.ai' s Jeremy Howard and DeepMind' s Sebastian Ruder.

You could say that ULMFiT was the release that got the transfer learning party started last year.

ULMFiT achieves state-of-the-art results using novel NLP techniques. This method involves fine-tuning a pretrained language model, trained on the Wikitext 103 dataset,

to a new dataset in such a manner that it does not forget what it previously learned.



ULMFiT outperforms numerous state-of-the-art on text classification tasks.

The Advantage of Transfer Learning:

Import Required Libraries

Most of the popular libraries like pandas, numpy, matplotlib, nltk, and keras, come preinstalled with Colab.

However, 2 libraries, PyTorch and fastai v1 (which we need in this exercise), will need to be installed manually. So, let's load them into our Colab environment:

```
!pip install torch_nightly –f https://download.pytorch.org/whl/nightly/cu92/torch_nightly.html

!pip install fastai
```

# import libraries

```
import fastai
from fastai import *
from fastai.text import *
import pandas as pd
import numpy as np
from functools import partial
import io
import os
```

Import the dataset which we downloaded earlier.

```
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers',
'quotes'))
documents = dataset.data
```

Let's create a dataframe consisting of the text documents and their corresponding labels (newsgroup names).

```
df = pd.DataFrame({'label':dataset.target, 'text':dataset.data})

df.shape
```

(11314, 2)

We' ll convert this into a binary classification problem by selecting only 2 out of the 20 labels present in the dataset. We will select labels 1 and 10 which correspond to 'comp.graphics' and 'rec.sport.hockey', respectively.

```
df = df[df['label'].isin([1,10])]
df = df.reset_index(drop = True)
```

Let' s have a quick look at the target distribution.

```
df['label'].value_counts()

10    600
1     584
Name: label, dtype: int64
```

The distribution looks pretty even. Accuracy would be a good evaluation metric to use in this case

Data Preprocessing

It' s always a good practice to feed clean data to your models, especially when the data comes in the form of unstructured text. Let' s clean our text by retaining only alphabets and removing everything else.

```
df['text'] = df['text'].str.replace("[^a-zA-Z]", " ")
```

Now, we will get rid of the stopwords from our text data. If you have never used stopwords before, then you will have to download them from the nltk package as I' ve shown below:

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

from nltk.corpus import stopwords
stop_words = stopwords.words('english')

# tokenization
tokenized_doc = df['text'].apply(lambda x: x.split())

# remove stop-words
tokenized_doc = tokenized_doc.apply(lambda x: [item for item in x if item not in
stop_words])

# de-tokenization
detokenized_doc = []
for i in range(len(df)):
     t = ' '.join(tokenized_doc[i])
     detokenized_doc.append(t)
```

```
df['text'] = detokenized_doc
```

Now let’s split our cleaned dataset into training and validation sets in a 60:40 ratio.

```
from sklearn.model_selection import train_test_split
```

```
# split data into training and validation set
df_trn, df_val = train_test_split(df, stratify = df['label'], test_size = 0.4,
random_state = 12)

df_trn.shape, df_val.shape

((710, 2), (474, 2))
```

Before proceeding further, we’ll need to prepare our data for the language model and for the classification model separately. The good news? This can be done quite easily using the fastai library:

```
# Language model data
```

```
data_lm = TextLMDataBunch.from_df(train_df = df_trn, valid_df = df_val, path = "")

# Classifier model data
data_clas = TextClasDataBunch.from_df(path = "", train_df = df_trn, valid_df = df_val,
vocab=data_lm.train_ds.vocab, bs=32)
```

Fine-Tuning the Pre-Trained Model and Making Predictions:

Fine-Tuning the Pre-Trained Model and Making Predictions

We can use the data_lm object we created earlier to fine-tune a pre-trained language model. We can create a learner object, ‘learn’, that will directly create a model, download the pre-trained weights, and be ready for fine-tuning:

```
learn = language_model_learner(data_lm, pretrained_model=URLs.WT103, drop_mult=0.7)
```

```
# train the learner object with learning rate = 1e-2
```

```
learn.fit_one_cycle(1, 1e-2)
```

Total time: 00:09

epoch	train_loss	valid_loss	accuracy
1	7.803613	6.306118	0.139369

We will save this encoder to use it for classification later.

```
learn.save_encoder('ft_enc')
```

Let's now use the data_clas object we created earlier to build a classifier with our fine-tuned encoder.

```
learn = text_classifier_learner(data_clas, drop_mult=0.7)
learn.load_encoder('ft_enc')
```

We will again try to fit our model.

```
learn.fit_one_cycle(1, 1e-2)
```

Total time: 00:32

epoch	train_loss	valid_loss	accuracy
1	0.534962	0.377784	0.907173

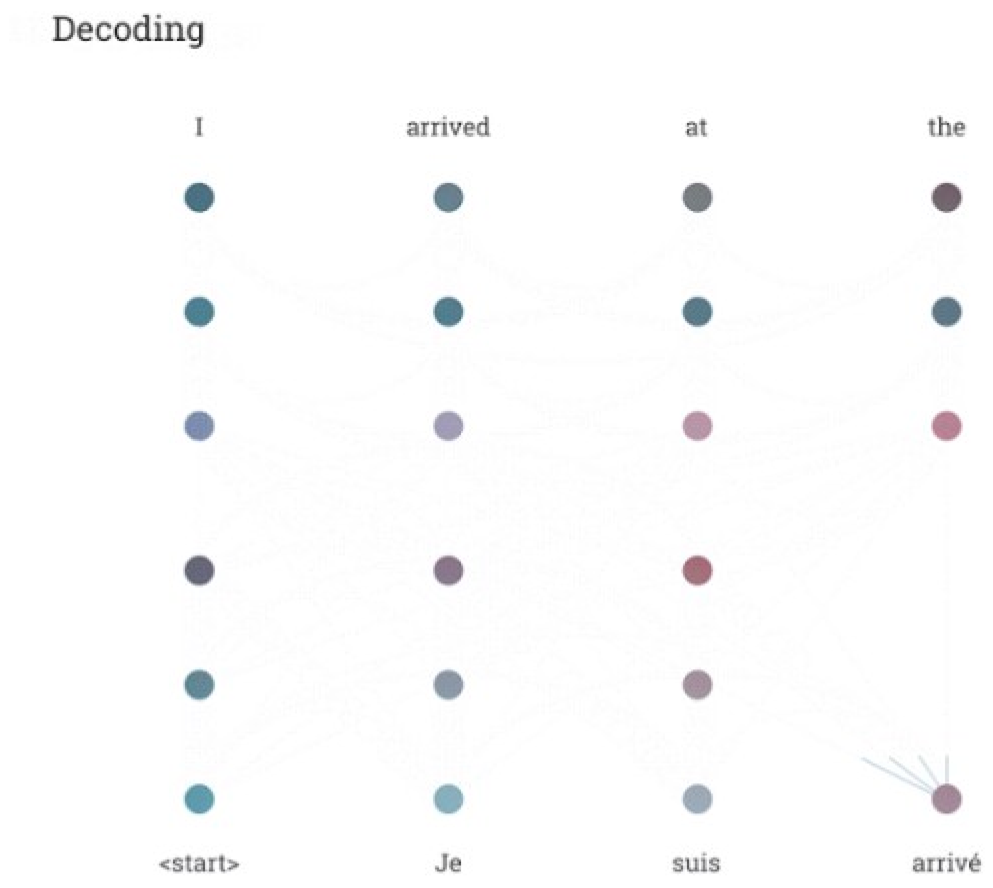
Transformer:

The Transformer architecture is at the core of almost all the recent major developments in NLP.

It was introduced in 2017 by Google. Back then, recurrent neural networks (RNN) were being used for language tasks, like machine translation and question answering systems.

As per Google, Transformer "applies a self-attention mechanism which directly models relationships between all words in a sentence, regardless of their respective position". It does so using a fixed-sized context (aka the previous words).

"She found the shells on the bank of the river." The model needs to understand that "bank" here refers to the shore and not a financial institution. Transformer understands this in a single step. I encourage you to read the full paper I have linked below to gain an understanding of how this works. It will blow your mind.



Google's BERT

The BERT framework has been making waves ever since Google published their results, and then open sourced the code behind it.

We can debate whether this marks "a new era in NLP", but there's not a shred of doubt that BERT is a very useful framework that generalizes well to a variety of NLP tasks.

BERT, short for Bidirectional Encoder Representations, considers the context from both sides (left and right) of a word. All previous efforts considered one side of a word at a time – either the left or the right. This bidirectionality helps the model gain a much better understanding of the context in which the word(s) was used. Additionally, BERT is designed to do multi-task learning, that is, it can perform different NLP tasks simultaneously.

BERT is the first unsupervised, deeply bidirectional system for pretraining NLP models. It was trained using only a plain text corpus.

At the time of its release, BERT was producing state-of-the-art results on 11 Natural Language Processing (NLP) tasks. Quite a monumental feat! You can train your own NLP model (such as a question-answering system) using BERT in just a few hours (on a single GPU).

Getting BERT downloaded and set up. We will be using the PyTorch version provided by the amazing folks at Hugging Face.

•Converting a dataset in the .csv format to the .tsv format that BERT knows and loves.

•Loading the .tsv files into a notebook and converting the text representations to a feature representation (think numerical) that the BERT model can work with.

•Setting up a pretrained BERT model for fine-tuning.

•Fine-tuning a BERT model.

•Evaluating the performance of the BERT model

•

Install the PyTorch version of BERT from Hugging Face.

```
pip install pytorch-pretrained-bert
```

•Preparing data

•Before we can cook the meal, we need to prepare the ingredients!

•Most datasets you find will typically come in the csv format and the Yelp Reviews dataset is no exception.

•Let's load it in with pandas and take a look.

Refer the Github: https://github.com/ThilinaRajapakse/BERT_binary_text_classification

•