### VIETNAM GENERAL CONFEDERATION OF LABOR **TON DUC THANG UNIVERSITY FACULTY OF INFORMATION TECHNOLOGY**



FINAL REPORT

TEXT GENERATION USING GPT-2

### *Supervisor*: **LÊ ANH CƯỜNG** *Student*: **TRỊNH BẢO TOÀN - 520K0332** **NGUYỄN DUY TUẤN - 520K0232** *Class*: **20K50301**

### *Group*: **01** *Year*: **24**

## HO CHI MINH CITY, 2023

### VIETNAM GENERAL CONFEDERATION OF LABOR **TON DUC THANG UNIVERSITY FACULTY OF INFORMATION TECHNOLOGY**



# FINAL REPORT

# 

# 

TEXT GENERATION USING GPT-2

### *Supervisor*: **LÊ ANH CƯỜNG** *Student*: **TRỊNH BẢO TOÀN – 520K0332** **NGUYỄN DUY TUẤN - 520K0232** *Class*: **20K50301**

### *Group*: **01** *Year*: **24**

## HO CHI MINH CITY, 2023

# THANK YOU

This report was finished thanks to the work of the other one in my group and the guidance of my lecturer, Lê Anh Cường. Without their assistance, I would not have been able to finish the report in the allotted time. I also want to express my gratitude to my university, Ton Duc Thang, for having given me a chance to research computer science in general, and in particular, Introduction Security Information.

# PROJECT COMPLETED AT TON DUC THANG UNIVERSITY

I hereby declare that this is my/our own project and is under the guidance of Phạm Thái Kỳ Trung. The research contents and results in this topic are honest and have not been published in any publication before. The data in the tables for analysis, comments and evaluation are collected by the author himself from different sources, clearly stated in the reference section.

In addition, the project also uses a number of comments, assessments as well as data of other authors, other agencies and organizations, with citations and source annotations.

If I find any fraud, I will take full responsibility for the content of my project. Ton Duc Thang University is not related to copyright and copyright violations caused by me (if any).

*Ho Chi Minh, ...........................  
 Author  
 (Sign)*

*Trịnh Bảo Toàn*

*Nguyễn Duy Tuấn*

# VERIFICATION AND EVALUATION OF LECTURER

**Verification of guiding lecturer**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Ho Chi Minh, ...........................*

(Sign)

**Evaluation of grading lecturer**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Ho Chi Minh, ...........................*

(Sign)

# 

# SUMMARY

The Transformer GPT-2 is a natural language processing model that is based on the Transformer architecture and was developed by OpenAI. It is a large-scale language model that can generate natural language text that is similar to human-written text. This report provides an overview of the Transformer GPT-2, including its architecture, pre-training process, and potential applications. The report also discusses the limitations of the model and its current state of development. Overall, the Transformer GPT-2 has shown impressive results in generating natural language text and has many potential applications in various domains. However, there are still challenges that need to be addressed, such as the model's high computational cost and limitations in generating coherent and consistent text over long sequences.

# TABLE OF CONTENT

[**THANK YOU 1**](#_a2fp7x293ls)

[**VERIFICATION AND EVALUATION OF LECTURER 3**](#_wwui2e8tnjyg)

[**SUMMARY 4**](#_yurf1il19y5i)

[**TABLE OF CONTENT 5**](#_1mnawpfzdz5i)

[**CHAPTER 1 – INTRODUCTION 6**](#_v1j4iktqqyep)

[**Exercise 1 7**](#_ycvfap9wkj9n)

[**Exercise 2 9**](#_mffw0ggvesub)

[**Exercise 3 10**](#_5geplg2j9ic5)

[**Exercise 4 12**](#_edfmmu7tpac6)

**Exercise 5 20**

# INTRODUCTION

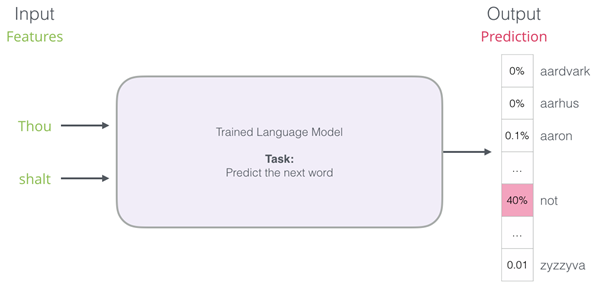
In this report we will see how to generate text with models based on the Transformers architecture, and we will use this knowledge to demonstrate how to create story. The objective is to demonstrate the operation and use of these models through this practical example.

First, we will present a theoretical introduction to text generation models, followed by a presentation to HuggingFace Transformers, the Python library that we will use in the rest of the post. Then, we will focus on the GPT-2 model, and how to use the interface available in HuggingFace Transformers, both to generate text with the pre-trained models, as well as to re-train them with their own text. Finally, we will see the ethical risks associated with the use of these models without caution, since they have been trained with text from the internet and have learned the same biases present on the web.

## 1.1. Text generation model

### **1.1.1. Introduction to text generation model**

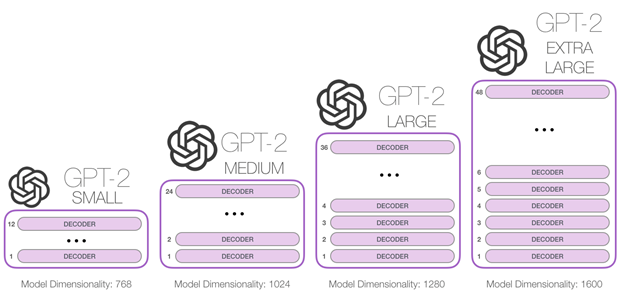
Text generation models began to be developed decades ago, long before the deep learning boom. The purpose of his type of models is to be able to predict a word or sequence of words given a text. The bottom diagram is a implified representation of what these models do, using a text as input, the model is capable of generating a robability distribution over the dictionary of words it knows, and choose based on it.



Early text generation models were trained using Markov chains, where each word was a state of the chain and the probability of the next word (based on the previous one) is calculated based on the number of occurrences of both words consecutively in the training text. Subsequently, recurrent neural networks (RNN) began to be used, which were capable of retaining a greater context of the text introduced, and Long Short-Term Memory (LSTM), which are a type of RNN that have a better long-term memory. Nevertheless, these type of networks are limited in what they can remember and they are also difficult to train, so they are not good for generating long texts.

In 2017 Google proposes a new architecture called the Transformer in its paper [“Attention Is All You Need”] (<https://arxiv.org/abs/1706.03762>), on which different text generation models are based today, such as GPT-2 and GPT-3, BERT or Transformer XL.

In this post we are going to focus on how to generate text with GPT-2, a text generation model created by OpenAI in February 2019 based on the architecture of the Transformer. It should be noted that GPT-2 is an autoregressive model, this means that it generates a word in each iteration. In addition, the model is available in different sizes depending on the embedding:



## 1.1.2. Hugging face transformer

Huggingface Transformers is a Python library that downloads pre-trained models for tasks like:

* Natural language understanding, such as sentiment analysis
* Natural language generation, such as text generation or text translation.

Among many others, it has available the four versions of GPT-2 trained and published by OpenAI, and offers an interface easy to use that makes it very accessible.

There are three main concepts or classes in the library that we will use throughout the post:

* **Tokenizer**: they store the vocabulary of each model and include methods to encode and decode strings in a list of token embeddings indexes that serve as input to the model.
* **Configuration**: they contain the necessary parameters to build a model. They are not required when using a pre-trained model
* **Model**: Pytorch or Keras models to work with the models pre-trained by the library.

# SET UP

First, let’s import all the packages we are going to use. Specifically, the versions of these packages are:

* Torch
* Transformer
* Numpy
* Pandas
* Math
* argparse

**Platform:**

* Windows
* Linux

**Tools:**

* Visual Studio Code / Google Colab

import numpy as np

import pandas as pd

import torch

import logging

from tqdm import tqdm

import math

import argparse

import os

!git clone https://github.com/huggingface/transformers

!pip install transformers/

from transformers import GPT2Tokenizer, GPT2LMHeadModel

from transformers.optimization import AdamW, get\_linear\_schedule\_with\_warmup

**| Optional: Mount Drive directory in Google Collab**

Not having a good local GPU is no longer an impediment to training GPU models. This notebook can be executed in Google Colab, which is a free environment Jupyter Notebook that does not require configuration, runs completely in the cloud and with the possibility of making use of Google’s GPUs.

To use Google Colab, it is necessary to mount a Google Drive storage if you want to save trained models, or the results of the models. It is really simple, since the environment where the notebooks run has a built-in package to do so. When executing it, it returns a url to login with your Google account and then you obtain an authorization code:

from google.colab import drive

drive.mount('/content/gdrive')

# 3. Text Generation using GPT-2

**3.1. Data Preprocessing**

First defines an argument parser using the ‘argparse’ module to accept command-line arguments. It then sets the default values for each argument:

parser = argparse.ArgumentParser()

parser.add\_argument('--seed', type=int, default=88888) # an optional argument to set the random seed for reproducibility

parser.add\_argument("--model\_name", default="gpt2", type=str) # the name of the pre-trained model to use

parser.add\_argument("--max\_seq\_length", default=512, type=int) # the maximum length of input sequences

parser.add\_argument("--train\_batch\_size", default=4, type=int) #  the batch size to use during training

parser.add\_argument("--valid\_batch\_size", default=4, type=int) # the batch size to use during validation

parser.add\_argument("--num\_train\_epochs", default=3, type=int) #  the number of epochs to train

parser.add\_argument("--warmup", default=0.1, type=float) #  the fraction of training steps to use for learning rate warmup

parser.add\_argument("--learning\_rate", default=5e-5, type=float) # the learning rate to use for training

parser.add\_argument("--input\_text\_path", default='/content/gdrive/MyDrive/TBT-DL/data', type=str) #  the path to the directory containing the input text files

args, \_ = parser.parse\_known\_args()

Next, we defines two functions, combinetext() and cleanpunctuation(), and then uses them to create training and validation datasets.

The 'combinetext(prompt, story) function takes two file names, prompt and story, as input. It then opens the two files using the open() method and reads in their contents as lists of strings. The function then checks that the two lists have the same length and merges the corresponding elements into a new list called combine. Each element in combine is a concatenation of a prompt string and the first 300 words of the corresponding story string, separated by the <sep> token.

The cleanpunctuation(s) function takes a string s as input and removes unnecessary whitespaces around punctuation marks, replaces common contractions with their expanded forms, and replaces the <newline> token with a newline character.

Finally, the code calls combinetext() and cleanpunctuation() on the valid.wp\_source and valid.wp\_target files to create a list of training examples called train\_text, and calls the same functions on the test.wp\_source and test.wp\_target files to create a list of validation examples called valid\_text.

DATAPATH=args.input\_text\_path

def combinetext(prompt, story):

    fp=open(os.path.join(DATAPATH,prompt),encoding='utf8')

    fs=open(os.path.join(DATAPATH,story),encoding='utf8')

    prompts=fp.readlines()

    stories=fs.readlines()

    assert len(prompts)==len(stories)

    combine=[]

    for i in range(len(prompts)):

        combine.append(prompts[i].rstrip()+' <sep> '+" ".join(stories[i].split()[:300]))

    return combine

#do a littel text clean with punctuations

def cleanpunctuation(s):

    for p in '!,.:;?':

        s=s.replace(' '+p,p)

    s=s.replace(' '+'n\'t','n\'t')

    s=s.replace(' '+'\'s','\'s')

    s=s.replace(' '+'\'re','\'re')

    s=s.replace(' '+'\'ve','\'ve')

    s=s.replace(' '+'\'ll','\'ll')

    s=s.replace(' '+'\'am','\'am')

    s=s.replace(' '+'\'m','\'m')

    s=s.replace(' '+'\' m','\'m')

    s=s.replace(' '+'\'m','\'m')

    s=s.replace(' '+'\' ve','\'ve')

    s=s.replace(' '+'\' s','\'s')

    s=s.replace('<newline>','\n')

    return s

train\_text=combinetext('valid.wp\_source', 'valid.wp\_target')

train\_text=list(map(cleanpunctuation,train\_text))

valid\_text=combinetext('test.wp\_source', 'test.wp\_target')

valid\_text=list(map(cleanpunctuation,valid\_text))

## 3.2. Model and tokenizer loading

Load both the model and the tokenizer the model will use. We both do it through the interface of the GPT2 classes that exist in Huggingface Transformers GPT2LMHeadModel and GPT2Tokenizer respectively. In both cases, you must specify the version of the model you want to use, and the 4 dimensions of the model published by OpenAI are available:

* 'gpt2'
* 'gpt2-medium'
* 'gpt2-large'
* 'gpt2-xl'

tokenizer = GPT2Tokenizer.from\_pretrained('gpt2')

tokenizer.pad\_token=tokenizer.eos\_token

inputs\_train = tokenizer(train\_text, padding=True,truncation=True,max\_length=args.max\_seq\_length)

inputs\_valid=tokenizer(valid\_text, padding=True,truncation=True,max\_length=args.max\_seq\_length)

## 3.3. Define create\_labels() function

This function create\_labels() creates labels for the inputs. In natural language processing, language models are trained in a supervised way, which means that they are trained on a dataset with input-output pairs. In this case, inputs is a dictionary containing the input tokens and their attention masks. The function loops through the input tokens and attention masks, and for each of them, it creates a label.

The label is a copy of the input token, but with the padding tokens replaced by -100. This is because the model should not learn anything from the padding tokens, so they are masked out. The create\_labels function adds the labels to the inputs dictionary with the key 'labels'. After this function is called on the training and validation inputs, the inputs\_train and inputs\_valid dictionaries contain the input tokens, their attention masks, and their corresponding labels.

def create\_labels(inputs):

    labels=[]

    for ids,attention\_mask in zip(inputs['input\_ids'],inputs['attention\_mask']):

        label=ids.copy()

        real\_len=sum(attention\_mask)

        padding\_len=len(attention\_mask)-sum(attention\_mask)

        label[:]=label[:real\_len]+[-100]\*padding\_len

        labels.append(label)

    inputs['labels']=labels

create\_labels(inputs\_train)

create\_labels(inputs\_valid)

## 3.4. Custom Story Data

This is a custom PyTorch Dataset class called StoryDataset. It is used to wrap the input data and prepare it for feeding into the model during training and validation. The constructor takes as input a dictionary called inputs containing the input\_ids, attention\_mask, and labels for the training or validation set.

The \_\_len\_\_() method returns the number of examples in the dataset.

The \_\_getitem\_\_() method is used to retrieve individual examples from the dataset. It returns a list of three tensors: input\_ids, attention\_mask, and labels. These tensors are created from the corresponding lists in the inputs dictionary. The input\_ids and attention\_mask tensors are of type torch.long, while the labels tensor is of type torch.long.

def create\_labels(inputs):

    labels=[]

    for ids,attention\_mask in zip(inputs['input\_ids'],inputs['attention\_mask']):

        label=ids.copy()

        real\_len=sum(attention\_mask)

        padding\_len=len(attention\_mask)-sum(attention\_mask)

        label[:]=label[:real\_len]+[-100]\*padding\_len

        labels.append(label)

    inputs['labels']=labels

create\_labels(inputs\_train)

create\_labels(inputs\_valid)

## 3.5. Create Train and Valid DataLoader

In the above code block, we define the training and validation data loaders using the StoryDataset class. The train\_batch\_size and valid\_batch\_size arguments are used to specify the batch sizes for training and validation data loaders, respectively. We pass the inputs\_train and inputs\_valid dictionaries to the StoryDataset constructor to create instances of the StoryDataset class. Then, we use torch.utils.data.DataLoader to create data loaders for training and validation datasets. We set the shuffle argument to False since the order of the data samples does not affect the training process in language modeling.

train\_batch\_size=args.train\_batch\_size

valid\_batch\_size=args.valid\_batch\_size

traindata=StoryDataset(inputs\_train)

train\_dataloader = torch.utils.data.DataLoader(

    traindata,

    shuffle=False,

    batch\_size=train\_batch\_size)

validdata=StoryDataset(inputs\_valid)

valid\_dataloader = torch.utils.data.DataLoader(

    validdata,

    shuffle=False,

    batch\_size=valid\_batch\_size)

**3.6. Fine-Tune**

The GPT2 model is loaded from the pre-trained model checkpoint 'gpt2' using the GPT2LMHeadModel.from pretrained() method. The from\_pretrained() method loads the pre-trained weights of the model and initializes a new model instance with those weights

model = GPT2LMHeadModel.from\_pretrained('gpt2')

Next, the validation dataset is loaded using the DataLoader class, with a batch size specified by valid\_batch\_size. The DataLoader class returns batches of data that can be used for validation.

The model is then evaluated on the validation dataset by looping through the validation DataLoader, moving the inputs to the GPU device, and using the model() method to generate predictions. The output of the model() method is a tuple that contains the loss and other information. The loss is extracted from the output and appended to a list of losses. The average loss is then calculated and used to calculate the perplexity. The perplexity is a measure of how well the model predicts the next word in the sequence, with a lower perplexity indicating better performance.

Finally, the average perplexity for the validation dataset before fine-tuning is printed. Additionaly add number of model parameters.

Output: The average perplexity for valid dataset before fine-tuning is 39.27880379562042 and Number of model parameters: 124,439,808

model.to('cuda')

model.eval()

eval\_loss=[]

for inputs in tqdm(valid\_dataloader, desc="eval"):

    d1,d2,d3=inputs

    d1=d1.to('cuda')

    d2=d2.to('cuda')

    d3=d3.to('cuda')

    with torch.no\_grad():

        output = model(input\_ids=d1, attention\_mask=d2,labels=d3)

        batch\_loss=output[0]

    eval\_loss+=[batch\_loss.cpu().item()]

    del batch\_loss

eval\_loss=np.mean(eval\_loss)

perplexity=math.exp(eval\_loss)

print(f'The average perplexity for valid dataset before fine-tuning is {perplexity}')

## 3.7. Generate Story Before Fine-tune

The code defines a function called generate\_story that takes in a prompt and a target story, and uses a pre-trained GPT-2 language model to generate new text based on the prompt. The function generates multiple possible output sequences (controlled by the num\_return\_sequences parameter), each with a maximum length of output\_length tokens. The function also allows for some control over the diversity of the generated text, through the use of the temperature, top\_k, and top\_p parameters.

The function first prints out the prompt and target story for reference. It then encodes the prompt using the tokenizer from the Hugging Face Transformers library, and generates new text using the generate method of the pre-trained GPT-2 model. The generated text is decoded from its encoded form using the tokenizer, and any text after the end-of-sequence (eos) token is removed. The generated text is then printed to the console, along with a header indicating which generated sequence it is (if there are multiple).

prompt=valid\_text[300][:valid\_text[300].find('<sep>')]

target=valid\_text[300][valid\_text[300].find('<sep>')+5:]

def generate\_story(prompt,target,k=0,p=0.9,output\_length=300,temperature=1,num\_return\_sequences=3,repetition\_penalty=1.0):

    print("====prompt====\n")

    print(prompt+"\n")

    print('====target story is as below===\n')

    print(target+"\n")

    encoded\_prompt = tokenizer.encode(prompt, add\_special\_tokens=False, return\_tensors="pt")

    model.to('cpu')

    model.eval()

    output\_sequences = model.generate(

        input\_ids=encoded\_prompt,

        max\_length=output\_length,

        temperature=temperature,

        top\_k=k,

        top\_p=p,

        repetition\_penalty=repetition\_penalty,

        do\_sample=True,

        num\_return\_sequences=num\_return\_sequences

    )

    if len(output\_sequences.shape) > 2:

        output\_sequences.squeeze\_()

    for generated\_sequence\_idx, generated\_sequence in enumerate(output\_sequences):

        print("=== GENERATED SEQUENCE {} ===".format(generated\_sequence\_idx + 1))

        generated\_sequence = generated\_sequence.tolist()

        # Decode text

        text = tokenizer.decode(generated\_sequence, clean\_up\_tokenization\_spaces=True)

        # Remove all text after eos token

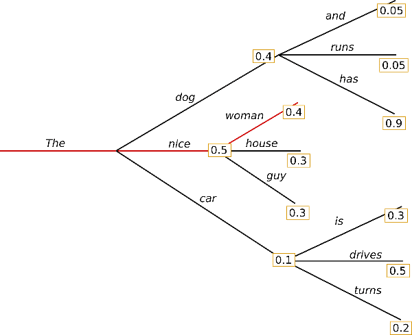
        text = text[: text.find(tokenizer.eos\_token)]

        print(text)

generate\_story(prompt,target)

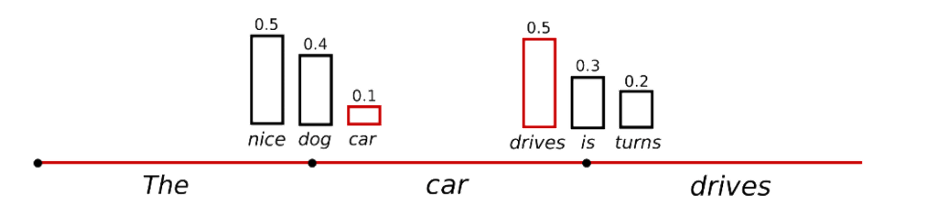
### 3.7.1. Decode mothod

We use Gready search for the decode method. It is the simplest method, which consists of choosing the word with the highest probability among all the possible ones. It is the one used when no parameter is specified.

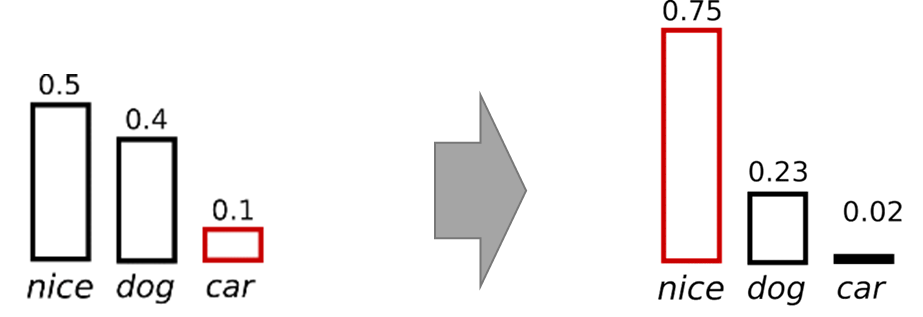


### 3.7.2. Sampling

The next word is selected randomly based on the probability distribution conditioned by the previous words.



Additionally, the temperature of the distribution can be adjusted, to increase the probability of extracting a word from among the most probable



Within sampling, two techniques are distinguished, top-K and top-P sampling. When implementing it, in both cases the do\_sample parameter would be passed to true.

output\_sequences = model.generate(

        input\_ids=encoded\_prompt,

        max\_length=output\_length,

        temperature=temperature,

        top\_k=k,

        top\_p=p,

        repetition\_penalty=repetition\_penalty,

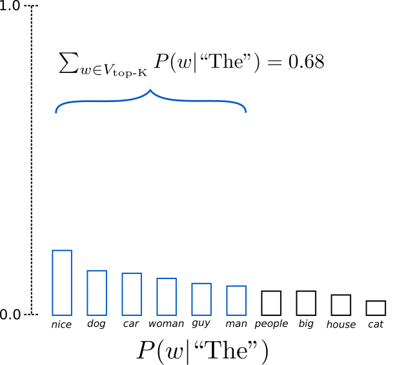
        do\_sample=True,

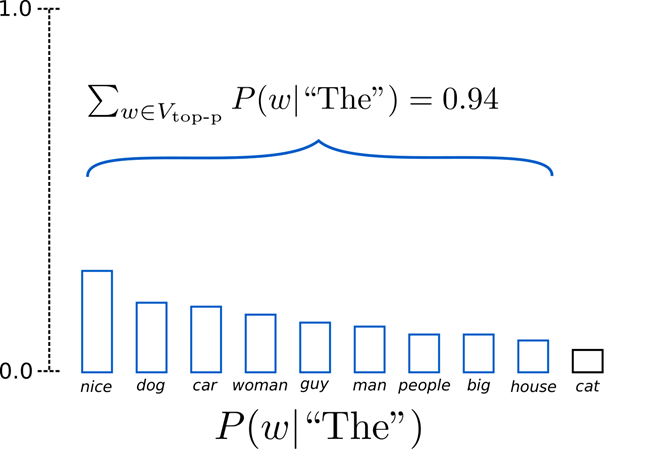
        num\_return\_sequences=num\_return\_sequences

    )

#### 3.7.2.1. Top K-sampling

The first one selects the next word randomly based on the probability distribution conditioned by the previous words of the K words with the highest probability. For example, in the case that we wanted to generate text starting from the word “The”, and given the following probability distribution over the vocabulary, if K were 6, the next word would be chosen randomly between the words nice, dog, car, woman, guy and man:





#### 3.7.2.2. Top-p (nucleus) sampling

In the case of Top-P, also called nucleus sampling, the next word is selected randomly based on the probability distribution conditioned by the previous word among the set of words that add a probability greater than or equal to p. Continuing with the previous example, if instead of setting the number of words to choose from, we decide to choose between the words that accumulate 94% of the probability, the options would increase:

## 3.8. Tuning the parameter

This block of code sets up the optimizer and scheduler for fine-tuning the GPT-2 language model.

* ‘num\_train\_epochs’ specifies the number of times the training loop should iterate over the entire training dataset
* ‘training\_steps\_per\_epoch’ is the number of batches in one epoch, which is calculated as the length of the training data divided by the batch size.
* ‘total\_num\_training\_steps’ is the total number of training steps for the entire training period, which is calculated as the number of epochs times the number of steps per epoch.
* ‘weight\_decay’ is the amount of L2 regularization to apply during training.
* ‘learning\_rate’ is the initial learning rate for the optimizer.
* ‘adam\_epsilon’ is a small value added to the denominator of the AdamW optimizer to improve numerical stability.
* ‘warmup\_steps’ is the number of steps during which the learning rate is gradually increased from 0 to the initial learning rate. This is a technique known as "learning rate warmup" that can help prevent the model from getting stuck in suboptimal local minima during the early stages of training.
* ‘no\_decay’ is a list of parameter names that should not be subjected to weight decay during training.
* ‘optimizer\_grouped\_parameters’ is a list of dictionaries that specify which parameters should be optimized together and with what weight decay.
* ‘optimizer’ is an instance of the AdamW optimizer that will be used to update the model parameters during training.
* ‘scheduler’ is an instance of the linear learning rate scheduler that will gradually adjust the learning rate during training. Specifically, the learning rate will increase linearly during the warmup period and then decrease linearly for the remainder of training.

num\_train\_epochs = args.num\_train\_epochs

training\_steps\_per\_epoch=len(train\_dataloader)

total\_num\_training\_steps = int(training\_steps\_per\_epoch\*num\_train\_epochs)

weight\_decay=0

learning\_rate=args.learning\_rate

adam\_epsilon=1e-8

warmup\_steps=int(total\_num\_training\_steps\*args.warmup)

no\_decay = ["bias", "LayerNorm.weight"]

optimizer\_grouped\_parameters = [

    {

        "params": [p for n, p in model.named\_parameters() if not any(nd in n for nd in no\_decay)],

        "weight\_decay": weight\_decay,

    },

    {

        "params": [p for n, p in model.named\_parameters() if any(nd in n for nd in no\_decay)],

        "weight\_decay": 0.0,

    },

]

optimizer = AdamW(optimizer\_grouped\_parameters, lr=learning\_rate, eps=adam\_epsilon)

scheduler = get\_linear\_schedule\_with\_warmup(

    optimizer, num\_warmup\_steps=warmup\_steps, num\_training\_steps=total\_num\_training\_steps

)

## 3.9. Train the data

This code runs the training loop for a language model using the provided train and validation data.

It first sets up the necessary hyperparameters for training, such as the number of training epochs, the total number of training steps, the learning rate, and the optimizer. It then moves the model to the GPU for training.

The training loop runs for the specified number of epochs. For each epoch, it first trains the model on the training data using a DataLoader to load batches of data. It then computes the average training loss for that epoch and moves on to evaluate the model on the validation data.

During evaluation, it computes the average validation loss per example and then calculates the perplexity of the model on the validation dataset. Perplexity is a measure of how well the model is able to predict the next token in the sequence based on the previous tokens, and a lower perplexity indicates better performance.

The training loop outputs the average training loss and perplexity for each epoch, allowing for monitoring of the model's performance during training.

def get\_generation\_with\_target(prompt,target,k=0,p=0.9,output\_length=300,temperature=1,num\_return\_sequences=3,repetition\_penalty=1.0):

    print("====prompt====\n")

    print(prompt+"\n")

    print('====target story is as below===\n')

    print(target+"\n")

    encoded\_prompt = tokenizer.encode(prompt, add\_special\_tokens=False, return\_tensors="pt")

    model.to('cpu')

    model.eval()

    output\_sequences = model.generate(

        input\_ids=encoded\_prompt,

        max\_length=output\_length,

        temperature=temperature,

        top\_k=k,

        top\_p=p,

        repetition\_penalty=repetition\_penalty,

        do\_sample=True,

        num\_return\_sequences=num\_return\_sequences

    )

    if len(output\_sequences.shape) > 2:

        output\_sequences.squeeze\_()

    for generated\_sequence\_idx, generated\_sequence in enumerate(output\_sequences):

        print("=== GENERATED SEQUENCE {} ===".format(generated\_sequence\_idx + 1))

        generated\_sequence = generated\_sequence.tolist()

        # Decode text

        text = tokenizer.decode(generated\_sequence, clean\_up\_tokenization\_spaces=True)

        # Remove all text after eos token

        text = text[: text.find(tokenizer.eos\_token)]

        print(text)

## 3.10. Evalute

The Bleu score in the range from 1 to 0 and my model only got 0.09. Explain for this:

A low BLEU score indicates that the generated text is dissimilar to the reference text. This can be advantageous in some scenarios where you do not want the generated text to be too similar to the reference text. For example, in creative writing or generating diverse responses in dialogue systems, having low BLEU scores can be desirable. However, in other scenarios such as machine translation or text summarization, higher BLEU scores are usually preferred as they indicate that the generated text is closer to the reference text, indicating better quality of the generated text. It all depends on the specific task and the goals of the text generation model.

print(f'\nAverage BLEU score for valid dataset is {np.mean(bleu\_score)}')

# 4. UPLOAD THE MODEL TO HUGGING FACE

We have uploaded the model and tokenizer to Hugging Face website so you can you loaded it directly in the website with my api.

You don’t need to run it locally anymore; all you have to do is using pipeline from transformers to load the model. Here is the link and the demo is below: (<https://huggingface.co/baotoan2002/GPT-2>)

You can set the length of the story and the number of story as much as you want in max\_length and num\_return\_sequences:

# Pipelines can also be loaded from the hub

from transformers import pipeline

generator = pipeline('text-generation', model='baotoan2002/GPT-2')

generator("Once upon a time,", max\_length=30, num\_return\_sequences=5)

# REFERENCES

1. <https://huggingface.co/baotoan2002/GPT-2>
2. <https://huggingface.co/gpt2>
3. <https://huggingface.co/docs/transformers/index>