



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مدیریت، علم و فناوری

گزارش سوال سوم از تمرین سری سوم درس پردازش داده‌های حجیم

چت بات مبتنی بر معماری ترانسفورمر برای سوال و جواب

نگارش

علیرضا کیوانی مهر

فاطمه حسینی سعدی

مهدی فلسفی

استاد راهنما

دکتر سعید شریفیان

استاد مشاور

مهندس مهدی امینی

بهمن ۱۴۰۲

چکیده

چت بات پرسش و پاسخ نوعی ربات نرم افزاری است که برای ارائه پاسخ های فوری به پرسش های کاربران طراحی شده است. این چت بات ها اغلب از فناوری های پردازش زبان طبیعی (NLP) و یادگیری ماشین (ML) پشتیبانی می کنند و به آن ها امکان می دهند تا سؤالات کاربر را به صورت مکالمه درک کنند و به آنها پاسخ دهند. ترانسفورماتور یک ساختار شبکه عصبی رمزنگار-رمزگذار است که به دلیل توانایی که در مدیریت ورودی با اندازه متغیر با استفاده از پشته های لایه های توجه به خود به جای RNN یا CNN دارد، به ربات های گفتگوی پرسش و پاسخ بسیار مرتبط هست. این معماری به مدل اجازه می دهد تا هیچ فرضی در مورد روابط زمانی/مکانی بین داده ها نداشته باشد، خروجی های لایه را به صورت موازی محاسبه کند و وابستگی های دوربرد را یاد بگیرد. این ویژگی ها Transformers را به ویژه برای کارهایی مانند تولید متن و پاسخگویی به سؤالات مؤثر می کند.

واژه های کلیدی: چت بات، مکانیزم خود توجهی، ترانسفورماتور

ب.....	چکیده.....
۱.....	فصل اول مقدمه (پیش نیازها).....
۸.....	فصل دوم آماده سازی دیتاست.....
۹.....	آماده سازی دیتاست.....
۱۳.....	فصل سوم ساخت مدل رمزگذار-رمزگشا.....
۲۵.....	فصل چهارم آموزش مدل و خروجی.....
۲۶.....	۴-۱- عملکرد مدل.....
۲۸.....	۴-۲- علت ضعف مدل.....
۲۸.....	۴-۳- راه حل برای بهبود.....
۳۰.....	پیوست‌ها.....
۳۱.....	Abstract.....

شکل ۱	Anaconda installer	دانلود	۳
شکل ۲	انتخاب محل نصب		۴
شکل ۳	انتخاب نوع نصب		۵
شکل ۴	تایید نسخه نصبی و ادامه		۵
شکل ۵	اتمام نصب		۶
شکل ۶	Anaconda Navigator	شمای کلی از	۶
شکل ۷	فعال سازی محیط مجازی ecg		۷
شکل ۸	نحوه جمع آوری داده		۱۰
شکل ۹	فایل prepare.py	بخش اول	۱۱
شکل ۱۰	فایل prepare.py	بخش دوم	۱۲
شکل ۱۱	ساختار رمزنگار رمزگشا ترانسفورماتور		۱۵
شکل ۱۲	ماژول مکانیزم خود توجهی و بلوک توجه		۱۶
شکل ۱۳	LayerNorm	بلوک	۱۶
شکل ۱۴	CausalSelfAttention	بخش اول	۱۷
شکل ۱۵	CausalSelfAttention	بخش دوم	۱۷
شکل ۱۶	GPTConfig	بلوک	۱۹
شکل ۱۷	GPT	بخش اول	۱۹
شکل ۱۸	GPT	بخش دوم	۲۰
شکل ۱۹	GPT	بخش سوم	۲۰
شکل ۲۰	GPT	بخش چهارم	۲۱
شکل ۲۱	GPT	بخش پنجم	۲۲
شکل ۲۲	GPT	بخش ششم	۲۳
شکل ۲۳	GPT	بخش هفتم	۲۳
شکل ۲۴	GPT	بخش هشتم	۲۴
شکل ۲۵	پارامترهای آموزش		۲۷

صفحه

فهرست جداول

جدول ۱ مقایسه تاثیر پارامترها بر روی عملکرد مدل ۲۷

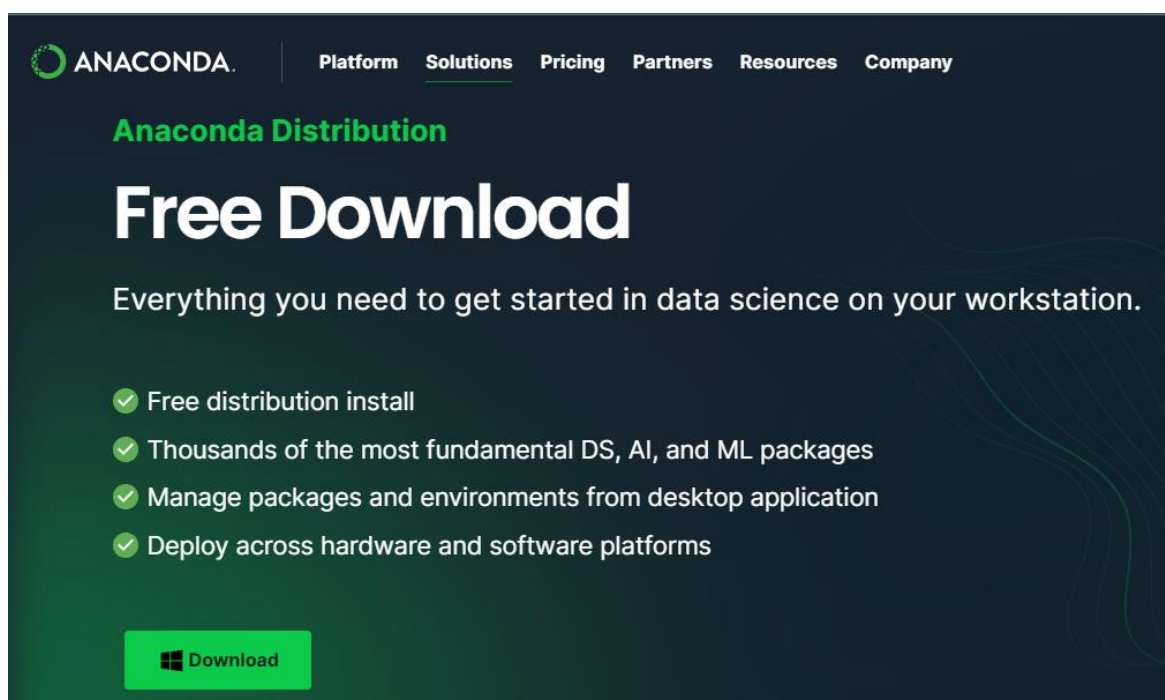
فصل اول

مقدمه (پیش نیازها)

مقدمه

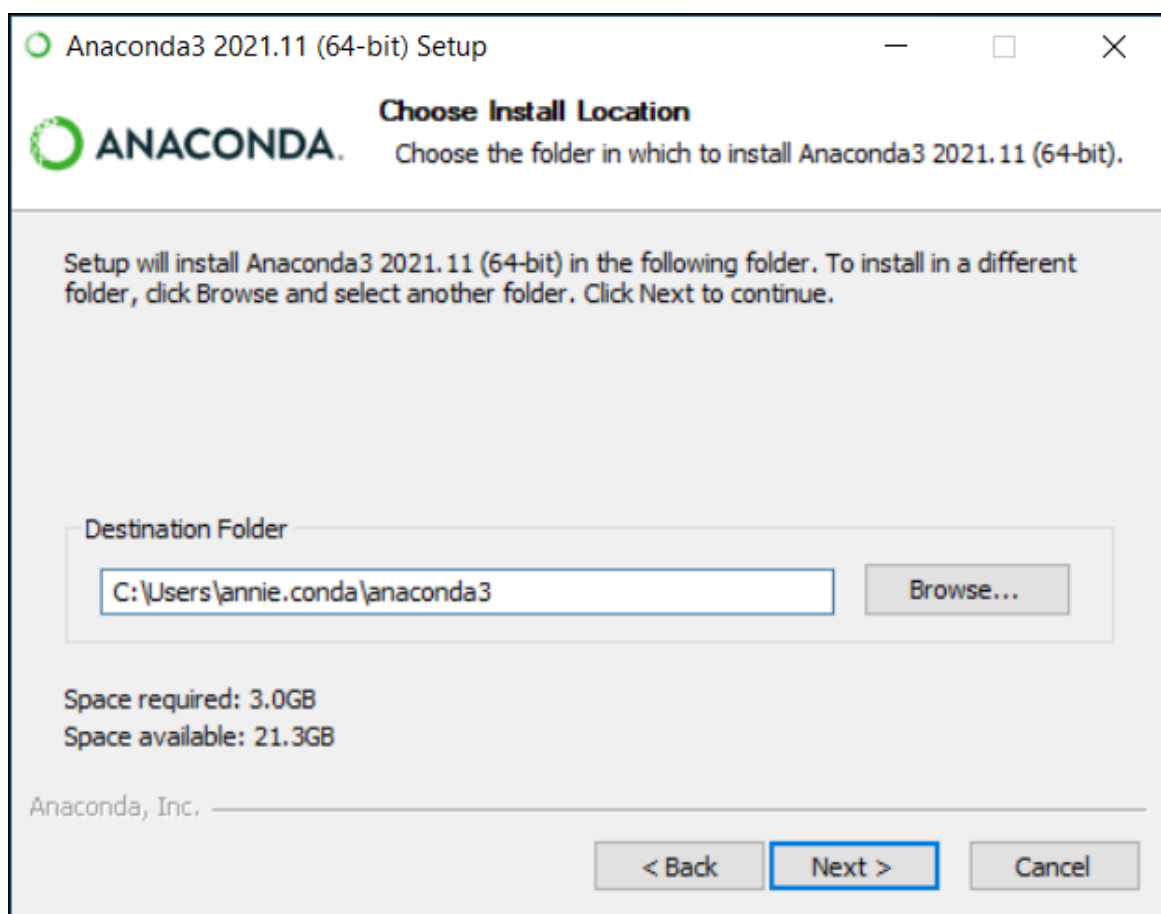
این بخش قبل از شروع بحث اصلی نیازمندی‌های ضروری برای پیاده سازی ساختار ترانسفورماتور را شرح می‌دهیم، ابتدا باید Anaconda یا نسخه سبک‌تر آن miniconda را نصب کنیم. Anaconda یک توزیع رایگان و منبع باز از زبان برنامه نویسی پایتون است که برای محاسبات علمی و علوم داده طراحی شده است. هدف آن راحت‌تر کردن مدیریت پکیج‌های مختلف است. برخی از ویژگی‌های کلیدی آن‌اکنوندا به شرح زیر است:

- **کتابخانه‌های از پیش بسته بندی شده پایتون:** Anaconda دارای بیش از ۱۵۰۰ کتابخانه از پیش آماده Python و R برای کاربردهای علم داده و یادگیری ماشین است. این شامل کتابخانه‌های محبوبی مانند NumPy، SciPy، Pandas و TensorFlow¹ می‌شود.
 - **محیط‌های توسعه یکپارچه:** چندین IDE از جمله Jupyter Notebook و Spyder در anaconda موجود است. Jupyter Notebook به دلیل ماهیت ماژولار و توانایی ذخیره نمودارها در خود نوت بوک مورد علاقه دانشمندان داده است. Spyder یک محیط برنامه نویسی دیگری است که با Anaconda ارائه می‌شود و برای توسعه Python استفاده می‌شود.
 - مدیریت پکیج‌ها:** Anaconda از یک مدیر بسته به نام Conda استفاده می‌کند که برای مدیریت کتابخانه‌ها و وابستگی‌ها را در پروژه‌های مختلف به کار می‌آید.
 - نسخه به روز پایتون:** وقتی Anaconda را نصب می‌کنید، به روزترین نسخه پایتون را نیز نصب به همراه آن نصب می‌شود و نیاز به نصب مجدد پایتون نیست.
 - Miniconda یک ورژن سبک‌تر Anaconda است که خیلی از ویژگی‌های نسخه اصلی را ندارد و فقط از مدیر بسته conda استفاده می‌کند. در نسخه miniconda همه پکیج‌ها باید به صورت مجزا و از طریق conda نصب شوند. برای نصب آن‌اکنوندا مراحل زیر را باید طی کنیم:
- الف)** با مراجعه به این [لینک](#) ابتدا installer آن‌اکنوندا را نصب می‌کنیم.



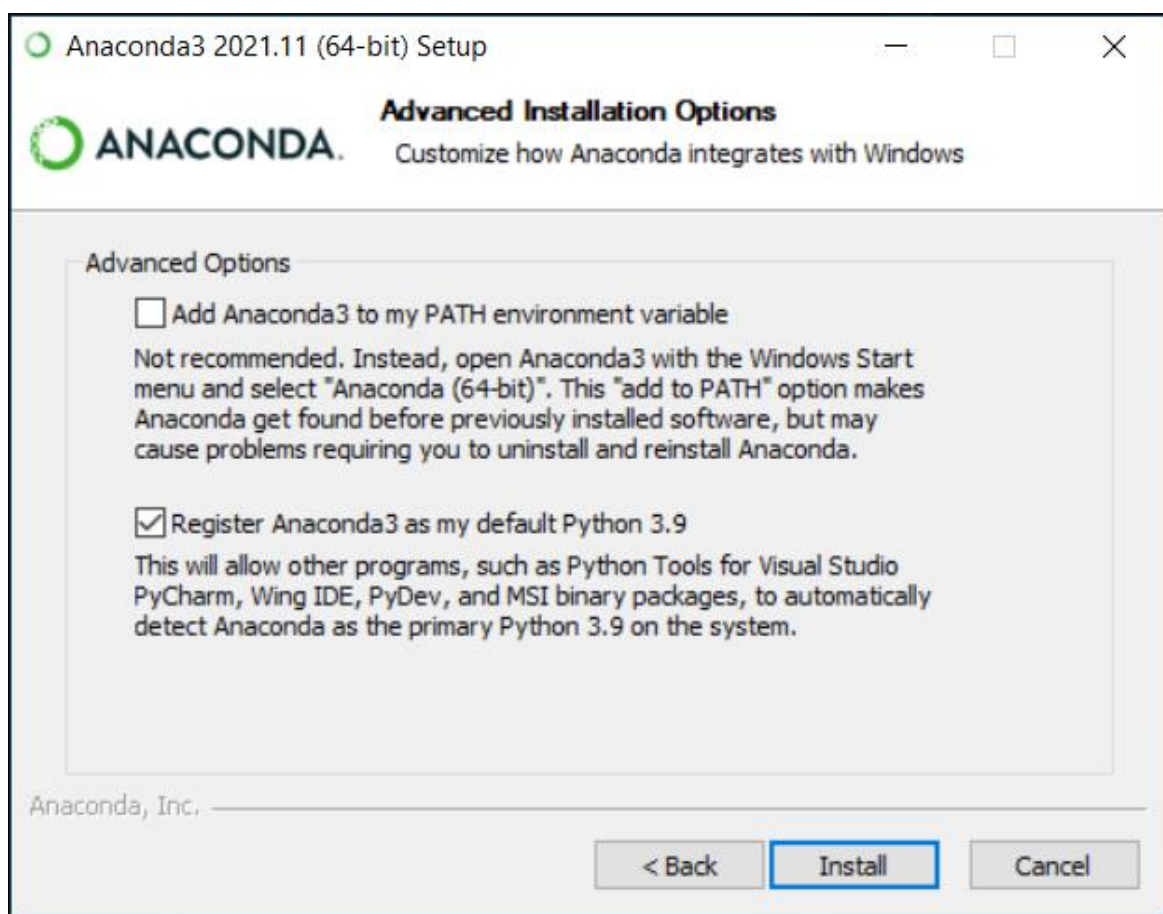
شکل ۱: دانلود Anaconda installer

ب) بعد از اتمام دانلود به فایل دانلود شده در فولدر downloads ویندوز رجوع کرده و فرایند نصب را طی می‌کنیم. یک پوشه مقصد را برای نصب Anaconda انتخاب کنید و روی Next کلیک کنید. Anaconda را در مسیر دایرکتوری که حاوی فاصله یا کاراکترهای یونیکد نیست نصب کنید. همچنین ایت امکان وجود دارد که آناکوندا را به متغیر محیطی PATH خود اضافه کنید یا آناکوندا را به عنوان پایتون پیش فرض خود ثبت کنید. معمولاً توصیه نمی‌شود که آناکوندا را به متغیر محیطی PATH اضافه شود، زیرا ممکن است با نرم افزارهای دیگر تداخل ایجاد کند.



شکل ۲ انتخاب محل نصب

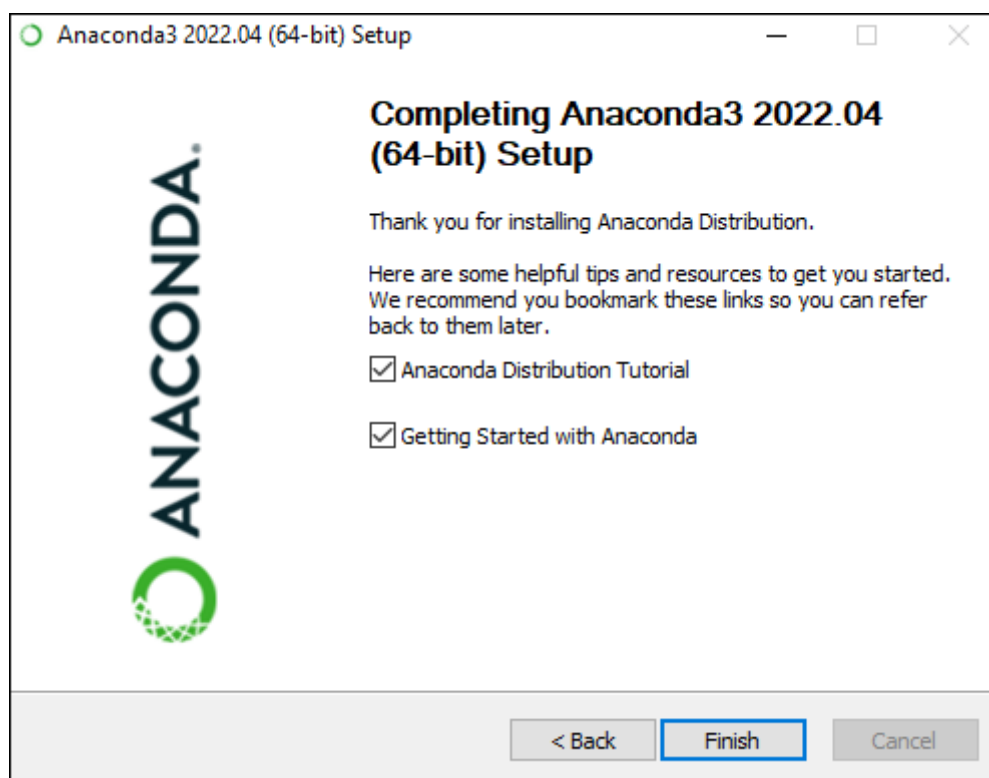
مگر اینکه قصد نصب و اجرای چندین نسخه آناکوندا یا چندین نسخه پایتون را دارید، پیش فرض را بپذیرید و این کادر را علامت بزنید. در عوض، با باز کردن Anaconda Navigator یا Anaconda Prompt از منوی Start، از نرم افزار Anaconda استفاده کنید.



شکل ۳ انتخاب نوع نصب

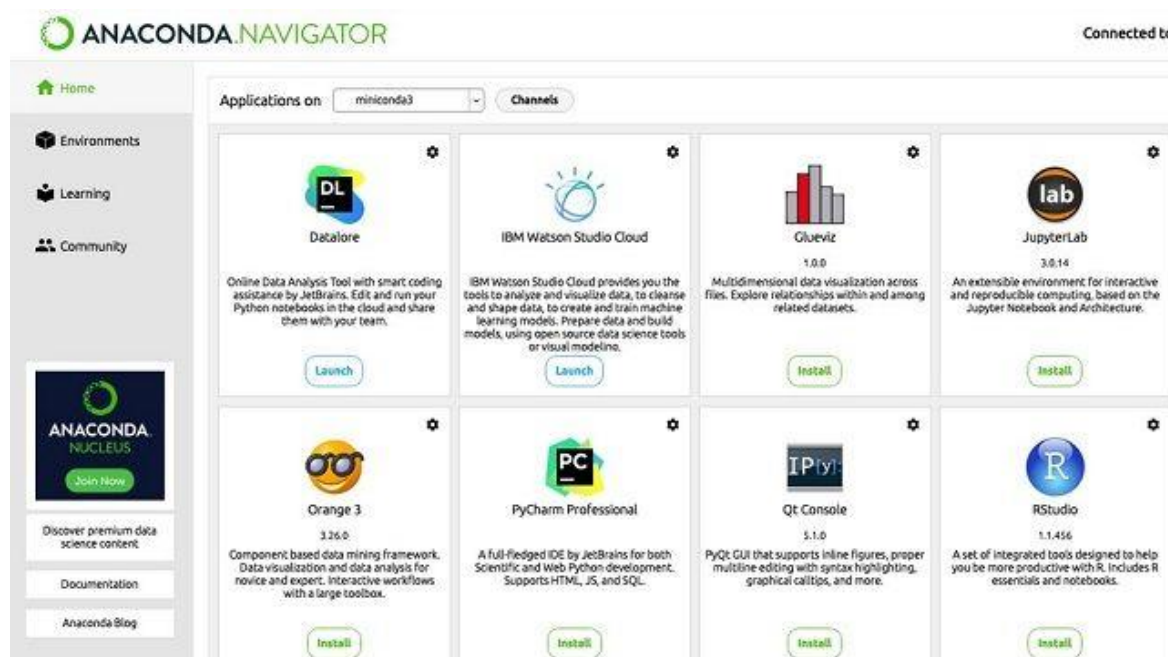


شکل ۴ تایید نسخه نصبی و ادامه



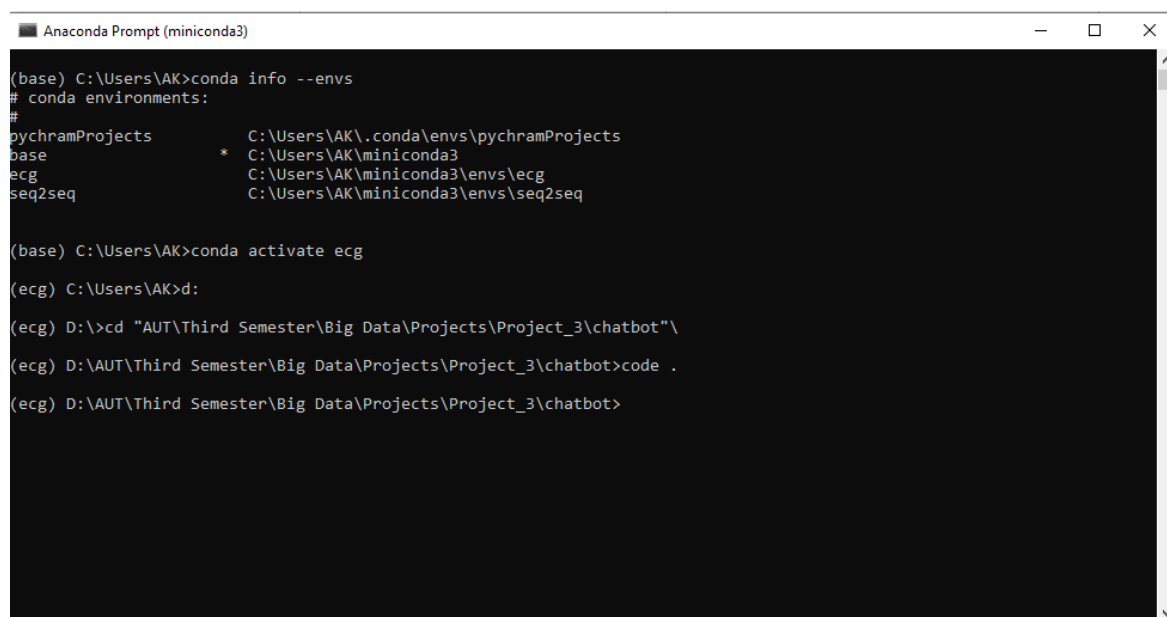
شکل ۵ اتمام نصب

بعد از نصب آناکوندا اکنون می‌توانیم راجع به محیط کاربری آن صحبت کنیم، مطابق شکل ۶ داشبورد اصلی شامل تمامی IDE‌های پیش فرض را نشان می‌دهد که می‌توانیم استفاده کنیم.



شکل ۶ شمای کلی از Anaconda Navigator

با آماده شدن محیط اناکوندا اکنون می توانیم برای پروژه خودمان یک محیط مجازی اختصاصی آماده کنیم تا تمامی dependency ها را درون آن نصب کنیم. چون ما از قبل یک محیط مجازی تحت عنوان ecg با اکثر پکیج های مورد نیاز را ساخته بودیم برای صرفه جویی در وقت پروژه را داخل همان محیط تعریف می کنیم. طی مراحل زیر وارد command prompt اختصاصی اناکوندا می شویم و visual studio code را در محل ذخیره سازی فایل های پروژه بالا می آوریم.



```

Anaconda Prompt (miniconda3)

(base) C:\Users\AK>conda info --envs
# conda environments:
#
pychramProjects      C:\Users\AK\.conda\envs\pychramProjects
base                  * C:\Users\AK\miniconda3
ecg                   C:\Users\AK\miniconda3\envs\ecg
seq2seq               C:\Users\AK\miniconda3\envs\seq2seq

(base) C:\Users\AK>conda activate ecg
(ecg) C:\Users\AK>d:
(ecg) D:\>cd "AUT\Third Semester\Big Data\Projects\Project_3\chatbot\"
(ecg) D:\AUT\Third Semester\Big Data\Projects\Project_3\chatbot>code .
(ecg) D:\AUT\Third Semester\Big Data\Projects\Project_3\chatbot>

```

شکل ۷ فعال سازی محیط مجازی ecg

پروژه دو فایل اصلی با نام های model.py و train.py دارد که به ترتیب برای ساخت مدل مبتنی بر encoder-decoder و آموزش مدل ساخته شده بر روی دیتاست WikiQA نوشته شده اند. در فصل های بعدی به هر کدام از این موارد می پردازیم.

فصل دوم

آماده سازی دیتاست

آماده سازی دیتاست

این تمرین بر اساس مجموعه داده WIKIQA انجام شده است، مجموعه از جفت‌های سؤال و جواب در دسترس عموم که برای تحقیق در مورد پاسخ‌گویی به سؤالات دامنه باز جمع‌آوری و حاشیه‌نویسی شده است. WIKIQA با استفاده از یک فرآیند طبیعی ساخته شده است و سائز آن چندین مرتبه بزرگتر از مجموعه داده قبلی است. علاوه بر این، مجموعه داده WIKIQA شامل سؤالاتی است که جملات صحیحی برای آنها وجود ندارد، و محققان را قادر می‌سازد تا روی راه‌اندازی پاسخ که یک جزء حیاتی در هر سیستم QA است، کار کنند. در این تمرین ما چندین سیستم را در مورد وظیفه انتخاب جمله پاسخ در هر دو مجموعه داده مقایسه می‌کنیم و همچنین عملکرد یک سیستم را در مورد مشکل راه‌اندازی پاسخ با استفاده از مجموعه داده WIKIQA توصیف می‌کنیم.

به منظور انعکاس نیازهای اطلاعاتی واقعی کاربران از گزارش‌های جستجوی Bing به عنوان منبع سؤال استفاده شده است. با در نظر گرفتن گزارش‌ها از ۱ می ۲۰۱۰ تا ۳۱ ژوئیه ۲۰۱۱، ابتدا پرس‌وجوهای سؤال‌مانند با استفاده از روش‌های اکتشافی ساده، مانند جستارهایی که با یک کلمه WH شروع می‌شوند (به عنوان مثال، «چه» یا «چگونه») و با علامت سؤال تمام می‌شود به عنوان پرس‌وجو^۱ انتخاب شده‌اند. علاوه بر این، ما برخی از پرسش‌های موجود را که قوانین را رعایت می‌کنند، فیلتر کردیم، مانند برنامه تلویزیونی «چگونه با مادرت آشنا شدم». در نهایت تقریباً ۲ درصد از پرسش‌ها انتخاب شدند. برای تمرکز روی سؤالات واقعی و بهبود کیفیت سؤال، فقط سؤالاتی را انتخاب کردیم که توسط حداقل ۵ کاربر منحصر به فرد پرسیده شده بودند و روی ویکی پدیا کلیک شده بودند. در میان آن‌ها، ۳۰۵۰ سؤال بر اساس فرکانس پرس و جو نمونه برداری شده‌اند.

¹ Query

The screenshot shows the 'WikiSentQA Pilot' interface. At the top, it displays 'Time Left: 01:06' and 'User: Scott Yih'. There is a link to 'Report a technical issue'. The main content area contains a question: 'Question: Who was the first black heavyweight champion?'. Below the question, there is a section titled 'Please comment on this question:'. This section includes two questions: 'Is this a legitimate question (you know what it's asking)?' with radio buttons for 'Yes' and 'No' (selected), and 'Do you think Wikipedia would be a good reference or source to find answers to the question?' with radio buttons for 'Yes' and 'No' (selected). A yellow bracket on the right side of the form groups these two questions under the heading 'Part 1: Comments on question'.

شکل ۸ نحوه جمع آوری داده

همانطور که در بالا ذکر شد برای جمع آوری داده از ویکی پدیا کمک می گیریم، این کار مستلزم آن است که جملاتی را از ویکی پدیا شناسایی کنید که می توانند به یک سوال پاسخ دهند. کل کار از سه قسمت تشکیل شده است. بخش اول می پرسد که آیا سؤال داده شده یک سؤال مشروع است یا خیر و آیا انتظار دارید که پاسخ آن را در ویکی پدیا پیدا کنید. بخش دوم پاراگراف کوتاهی از ویکی پدیا ارائه می کند و می پرسد که آیا پاسخ سؤال را می توان در این پاراگراف یافت؟ اگر پاسخ مثبت است، به شما این امکان را می دهد که انتخاب کنید کدام جملات در این پاراگراف می توانند به صورت مجزا به سؤال پاسخ دهند. جزئیات این سه قسمت به همراه مثال در زیر آورده شده است.

فایل `prepare.py` برای آماده سازی داده ها جهت استفاده برای آموزش مدل است. برای این تمرین مجموعه داده را برای مدل سازی زبان در سطح کاراکتر^۲ آماده کرده ایم. بنابراین به جای رمزگذاری با توکن های BPE GPT-2، فقط کاراکترها را به `ints` نگاشت می کنیم. `val.bin`، `train.bin` حاوی شناسه ها و متا را ذخیره می کند. `pkl` حاوی رمزگذار و رمزگشا و برخی اطلاعات مرتبط دیگر.

در بخش اول فایل `prepare` که در شکل ۹ نشان داده شده است اول فایل `WikiQA.txt` که قبلاً از روی گوگل درایو دانلود کرده بودیم می خوانیم. تعداد کل کاراکترهای دیتاست برابر 1115394 تا است. تابع `encode` یک استرینگ دریافت می کند و با توجه به نگاشت هر کاراکتر یک لیست از اعداد خروجی می دهد. تابع `decode` دقیقاً برعکس این کار را انجام می دهد. بعد از اینکود و دیکود کردن داده را به دو مجموعه آموزش و تست تقسیم می کنیم.

^۲ Character-level modeling

```

data > prepare.py
1  import os
2  import pickle
3  import requests
4  import numpy as np
5
6  input_file_path = os.path.join(os.path.dirname(__file__), 'WikiQA-train.txt')
7
8  with open(input_file_path, 'r') as f:
9      data = f.read()
10 print(f"length of dataset in characters: {len(data):,}")
11
12 # get all the unique characters that occur in this text
13 chars = sorted(list(set(data)))
14 vocab_size = len(chars)
15 print("all the unique characters:", ''.join(chars))
16 print(f"vocab size: {vocab_size:,}")
17
18 # create a mapping from characters to integers
19 stoi = { ch:i for i,ch in enumerate(chars) }
20 itos = { i:ch for i,ch in enumerate(chars) }
21 def encode(s):
22     return [stoi[c] for c in s] # encoder: take a string, output a list of integers
23 def decode(l):
24     return ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string
25
26 # create the train and test splits
27 n = len(data)
28 train_data = data[:int(n*0.9)]
29 val_data = data[int(n*0.9):]
30

```

شکل ۹ فایل prepare.py بخش اول

در ادامه فایل prepare.py که در شکل ۱۰ نشان داده شده است داده‌های آموزش و ارزیابی را به صورت مجموعه توکن در train_ids و val_ids ذخیره می‌کنیم. در حال حاضر ما 3301686 تا توکن در مجموعه آموزش و 366855 تا توکن در مجموعه validation داریم. خروجی نهایی این فایل و چیزی که به عنوان ورودی به مدل داده خواهد شد دو فایل train.bin و val.bin است که در همان پوشه data ذخیره می‌شوند. یک فایل مهم دیگر meta.pkl است که حاوی متاداده دیتاست است که شامل اندازه vocabulary، نگاشت‌های اینکودر و دیکودر است.


```
# encode both to integers
train_ids = encode(train_data)
val_ids = encode(val_data)
print(f"train has {len(train_ids):,} tokens")
print(f"val has {len(val_ids):,} tokens")

# export to bin files
train_ids = np.array(train_ids, dtype=np.uint16)
val_ids = np.array(val_ids, dtype=np.uint16)
train_ids.tofile(os.path.join(os.path.dirname(__file__), 'train.bin'))
val_ids.tofile(os.path.join(os.path.dirname(__file__), 'val.bin'))

# save the meta information as well, to help us encode/decode later
meta = {
    'vocab_size': vocab_size,
    'itos': itos,
    'stoi': stoi,
}
with open(os.path.join(os.path.dirname(__file__), 'meta.pkl'), 'wb') as f:
    pickle.dump(meta, f)
```

شکل ۱۰ فایل prepare.py بخش دوم

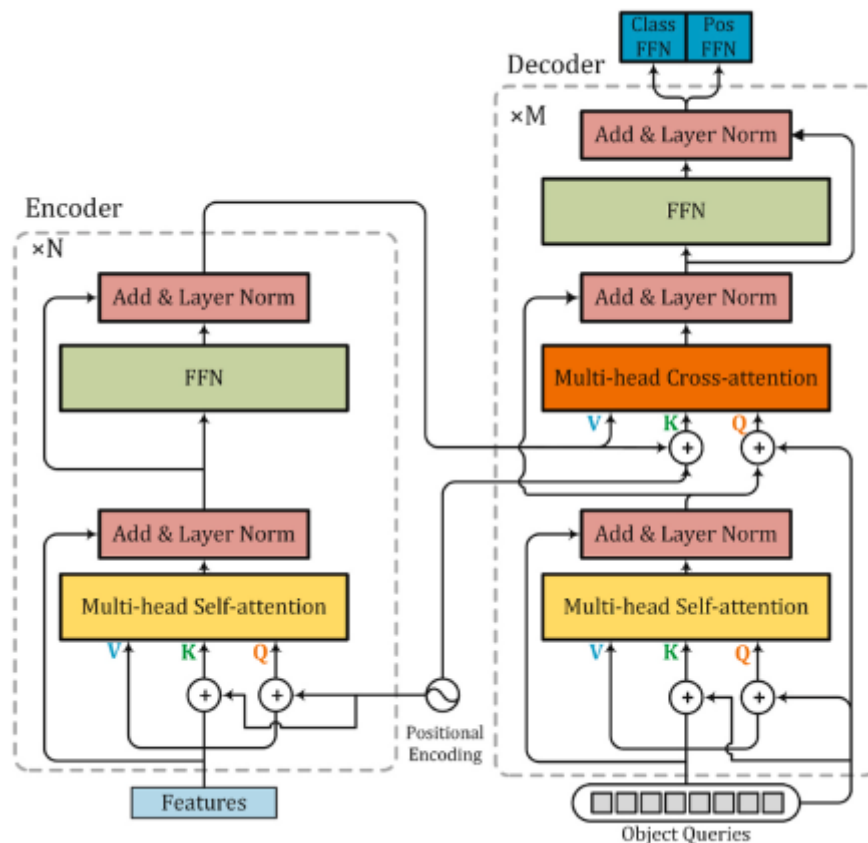
فصل سوم

ساخت مدل رمزگذار-رمزگشا

ساخت مدل رمزگذار – رمزگشا

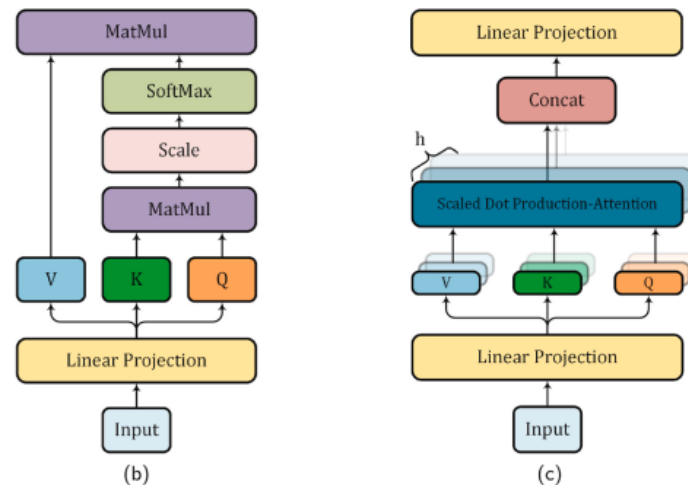
مدل ترانسفورماتور، همانطور که در مقاله "توجه تنها چیزی است که شما نیاز دارید" معرفی شده است، یک معماری منحصر به فرد است که تنها به مکانیسم های توجه بستگی دارد و نیاز به تکرار و پیچیدگی را از بین می برد. این ساختار از یک رمزگذار-رمزگشا پیروی می کند، با هر لایه در رمزگذار و رمزگشا دارای یک مکانیسم خودتوجهی چند سر و یک شبکه پیشخور کاملاً متصل از نظر موقعیت است. رمزگشا دارای یک لایه فرعی اضافی برای توجه چند سر بر روی خروجی رمزگذار است. مکانیسم توجه، به ویژه «توجه محصول مقیاس شده» و «توجه چند سر» به مدل اجازه می دهد تا اهمیت مقادیر ورودی مختلف را بر اساس زمینه آن ها بسنجید و روی موقعیت های مختلف به طور همزمان تمرکز کند.

برای حفظ نظم در توالی ورودی، ترانسفورماتور کدهای موقعیتی را در خود جای داده است که به تعبیه های ورودی اضافه می شوند و در طول آموزش یاد می گیرند. هر لایه در انکودر و رمزگشا شامل یک شبکه فید فوروارد کاملاً متصل است که به طور یکسان در هر موقعیت اعمال می شود. این مدل از اتصالات باقیمانده در اطراف هر زیر لایه، به دنبال نرمال سازی لایه، برای تسهیل جریان گرادیان در طول انتشار پس انداز استفاده می کند. خروجی رمزگشا برای پیش بینی نشانه بعدی در دنباله با استفاده از تبدیل خطی و تابع softmax پیش بینی می شود. اتکای ترانسفورماتور به مکانیسم های توجه، آن را قادر می سازد تا وابستگی های دوربرد در داده ها را به طور مؤثرتری مدیریت کند و مدل را بسیار موازی پذیر می کند و در نتیجه زمان آموزش را کاهش می دهد. شکل ۱۱ ساختار کلی ترانسفورماتور با ماژول های خود توجهی و شبکه های کاملاً در هم تنیده را نشان می دهد.



شکل ۱۱ ساختار رمزنگار رمزگشا ترانسفورماتور

مهم ترین عنصر معماری ترانسفورمر مکانیزم خود توجهی است که هم در encoder و هم در decoder استفاده می شود. شکل ۱۲ اجزای سازنده بلوک خود توجهی را در نشان می دهد، شکل 12-b یک لایه attention را نشان می دهد که سه مقدار key، query و value را از embedding های ورودی دریافت کرده خروجی را که ضرب ماتریسی این مقادیر است تولید می کند. شکل 12-c مجموعه لایه های خود توجهی را نشان می دهد که Multi-head self-attention را می سازند که در نهایت خروجی نهایی تلفیق همه سرهای توجه است.



شکل ۱۲ ماژول مکانیزم خود توجهی و بلوک توجه

کلاس LayerNorm برای نرمال سازی لایه ورودی استفاده می‌شود. در این جا میانگین و انحراف معیار بر روی آخرین بعد محاسبه می‌شود (بر خلاف نرمال سازی دسته ای که در آن بر روی بعد اول محاسبه می‌شود). این باعث می‌شود نرمال سازی لایه برای شبکه های عصبی مکرر و مدل های ترانسفورماتور موثر باشد، جایی که اندازه batch می‌تواند به صورت پویا تغییر کند. این به بهبود پایداری فرآیند یادگیری شبکه عصبی کمک می‌کند.

```

model.py
1  import math
2  import inspect
3  from dataclasses import dataclass
4
5  import torch
6  import torch.nn as nn
7  from torch.nn import functional as F
8
9  class LayerNorm(nn.Module):
10
11      def __init__(self, ndim, bias):
12          super().__init__()
13          self.weight = nn.Parameter(torch.ones(ndim))
14          self.bias = nn.Parameter(torch.zeros(ndim)) if bias else None
15
16      def forward(self, input):
17          return F.layer_norm(input, self.weight.shape, self.weight, self.bias, 1e-5)
18

```

شکل ۱۳ بلوک LayerNorm

ماژول CausalSelfAttention که در شکل ۱۴ آمده است، کارش محاسبه یک سر خود توجهی است، به این صورت که سه متغیر k ، q و v بر اساس c_attn می‌سازد و همه را ضرب ماتریسی می‌کند.

```

class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        # flash attention make GPU go brrrrr but support is only in PyTorch >= 2.0
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
            # causal mask to ensure that attention is only applied to the left in the input sequence
            self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head forward to be the batch dim
        q, k, v = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

```

شکل ۱۴ بلوک CausalSelfAttention بخش اول

```

# causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
if self.flash:
    # efficient attention using Flash Attention CUDA kernels
    y = torch.nn.functional.scaled_dot_product_attention(q, k, v, attn_mask=None, dropout_p=self.dropout if self.training else 0, is_causal=True)
else:
    # manual implementation of attention
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:, :, T, T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)
    y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side

# output projection
y = self.resid_dropout(self.c_proj(y))
return y

```

شکل ۱۵ بلوک CausalSelfAttention بخش دوم

شکل ۱۶ بلوک MLP که خروجی تجمیع شده ماژول Self-attention به آن وارد می‌شود را نشان می‌دهد که از دو لایه fully-connected و یک Gelu در وسط تشکیل شده است. ساختار Block برای پیکربندی تمام ماژول‌هایی است که تا الان تعریف شده اند. بعد از آن ماژول اصلی یعنی GPT را داریم که برای ساخت مدل اصلی استفاده می‌شود. ابتدا باید پارامترهای پیش فرض را تعیین کنیم: سائز بلوک‌ها، سائز دیکشنری، تعداد لایه‌ها، تعداد سرهای خود توجهی، اندازه embedding و مقدار Dropout از این موارد هستند.

```
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)
        self.gelu = nn.GELU()
        self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x
```

هر کلاس دارای یک متد «forward» برای محاسبه خروجی داده شده با یک ورودی و یک متد «__init__» برای مقداردهی اولیه لایه، از جمله هر زیر لایه و پارامتر است. کلاس «nn.Module» پایه از PyTorch با استفاده از «super().__init__» مقداردهی اولیه می شود و روش های ضروری برای کار با لایه های PyTorch را ارائه می دهد.

کلاس «nn.Linear» برای لایه های خطی کاملاً متصل و «nn.Dropout» برای منظم سازی استفاده می شود. تابع فعال سازی واحد خطی خطای گاوسی (GELU) با «nn.GELU» پیاده سازی می شود. تابع "hasattr" بررسی می کند که آیا ماژول "torch.nn.functional" تابع "product_attention" را برای محاسبه توجه دارد یا خیر. اگر نه، پیاده سازی دستی کندتر استفاده می شود. روش «register_buffer» تنسوری را ثبت می کند که پارامتر مدل نیست، اما بخشی از حالت مدل است و باید با بقیه مدل به دستگاه (CPU یا GPU) منتقل شود. عملیات تانسور استاندارد در PyTorch، مانند تغییر شکل، جابجایی، اطمینان از مجاورت حافظه، تقسیم یک تانسور به چند تانسور، و ضرب ماتریس، با استفاده از "view"، "transpose"، "contiguous"، "split" و "@" انجام می شود.

```

class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x

@dataclass
class GPTConfig:
    block_size: int = 1024
    vocab_size: int = 50304 # GPT-2 vocab_size of 50257, padded up to nearest multiple of 64 for efficiency
    n_layer: int = 12
    n_head: int = 12
    n_embd: int = 768
    dropout: float = 0.0
    bias: bool = True # True: bias in Linears and LayerNorms, like GPT-2. False: a bit better and faster

```

شکل ۱۶ بلوک GPTConfig

```

class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = LayerNorm(config.n_embd, bias=config.bias),
        ))

        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        self.transformer.wte.weight = self.lm_head.weight

        # init all weights
        self.apply(self._init_weights)
        # apply special scaled init to the residual projections, per GPT-2 paper
        for pn, p in self.named_parameters():
            if pn.endswith('c_proj.weight'):
                torch.nn.init.normal_(p, mean=0.0, std=0.02/math.sqrt(2 * config.n_layer))

        # report number of parameters
        print("number of parameters: %.2fM" % (self.get_num_params()/1e6,))

```

شکل ۱۷ بلوک GPT بخش اول


```

def get_num_params(self, non_embedding=True):
    n_params = sum(p.numel() for p in self.parameters())
    if non_embedding:
        n_params -= self.transformer.wpe.weight.numel()
    return n_params

def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        if module.bias is not None:
            torch.nn.init.zeros_(module.bias)
    elif isinstance(module, nn.Embedding):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

def forward(self, idx, targets=None):
    device = idx.device
    b, t = idx.size()
    assert t <= self.config.block_size, f"Cannot forward sequence of length {t}, block size is only {self.config.block_size}"
    pos = torch.arange(0, t, dtype=torch.long, device=device) # shape (t)

    # forward the GPT model itself
    tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
    pos_emb = self.transformer.wpe(pos) # position embeddings of shape (t, n_embd)
    x = self.transformer.drop(tok_emb + pos_emb)
    for block in self.transformer.h:
        x = block(x)
    x = self.transformer.ln_f(x)

```

شکل ۱۸ بلوک GPT بخش دوم

```

    if targets is not None:
        # if we are given some desired targets also calculate the loss
        logits = self.lm_head(x)
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1)
    else:
        # inference-time mini-optimization: only forward the lm_head on the very last position
        logits = self.lm_head(x[:, [-1], :]) # note: using list [-1] to preserve the time dim
        loss = None

    return logits, loss

def crop_block_size(self, block_size):
    assert block_size <= self.config.block_size
    self.config.block_size = block_size
    self.transformer.wpe.weight = nn.Parameter(self.transformer.wpe.weight[:block_size])
    for block in self.transformer.h:
        if hasattr(block.attn, 'bias'):
            block.attn.bias = block.attn.bias[:, :, :block_size, :block_size]

@classmethod
def from_pretrained(cls, model_type, override_args=None):
    assert model_type in {'gpt2', 'gpt2-medium', 'gpt2-large', 'gpt2-xl'}
    override_args = override_args or {} # default to empty dict
    # only dropout can be overridden see more notes below
    assert all(k == 'dropout' for k in override_args)
    from transformers import GPT2LMHeadModel
    print("loading weights from pretrained gpt: %s" % model_type)

```

شکل ۱۹ بلوک GPT بخش سوم

```

# n_layer, n_head and n_embd are determined from model_type
config_args = {
    'gpt2': dict(n_layer=12, n_head=12, n_embd=768), # 124M params
    'gpt2-medium': dict(n_layer=24, n_head=16, n_embd=1024), # 350M params
    'gpt2-large': dict(n_layer=36, n_head=20, n_embd=1280), # 774M params
    'gpt2-xl': dict(n_layer=48, n_head=25, n_embd=1600), # 1558M params
}[model_type]
print("forcing vocab_size=50257, block_size=1024, bias=True")
config_args['vocab_size'] = 50257 # always 50257 for GPT model checkpoints
config_args['block_size'] = 1024 # always 1024 for GPT model checkpoints
config_args['bias'] = True # always True for GPT model checkpoints
# we can override the dropout rate, if desired
if 'dropout' in override_args:
    print(f"overriding dropout rate to {override_args['dropout']}")
    config_args['dropout'] = override_args['dropout']
# create a from-scratch initialized minGPT model
config = GPTConfig(**config_args)
model = GPT(config)
sd = model.state_dict()
sd_keys = sd.keys()
sd_keys = [k for k in sd_keys if not k.endswith('.attn.bias')] # discard this mask / buffer, not a param

# init a huggingface/transformers model
model_hf = GPT2LMHeadModel.from_pretrained(model_type)
sd_hf = model_hf.state_dict()

```

شکل ۲۰ بلوک GPT بخش چهارم

بعد از GPTConfig پارامترهای GPT را که شامل wte و wpe می‌شود را مقدار دهی اولیه می‌کنیم، این ها embedding های ورودی ما متناسب با سایز دیکشنری و بلوک هستند. در مرحله بعد وزن‌ها را مقداردهی اولیه می‌کنیم و با توجه به قدرت سیستم از بین مدل‌های gpt2، gpt2-medium، gpt2-large و gpt2-xl مطابق شکل ۲۱ یکی را به عنوان مدل پایه انتخاب می‌کنیم. هر کدام از این مدل‌ها پیکربندی منحصر به فرد خودشان را دارند و تعداد پارامترهای هر کدام به صورت افزایشی تغییر می‌کند.

```
# copy while ensuring all of the parameters are aligned and match in names and shapes
sd_keys_hf = sd_hf.keys()
sd_keys_hf = [k for k in sd_keys_hf if not k.endswith('.attn.masked_bias')] # ignore these, just a buffer
sd_keys_hf = [k for k in sd_keys_hf if not k.endswith('.attn.bias')] # same, just the mask (buffer)
transposed = ['attn.c_attn.weight', 'attn.c_proj.weight', 'mlp.c_fc.weight', 'mlp.c_proj.weight']
# basically the openai checkpoints use a "Conv1D" module, but we only want to use a vanilla Linear
# this means that we have to transpose these weights when we import them
assert len(sd_keys_hf) == len(sd_keys), f"mismatched keys: {len(sd_keys_hf)} != {len(sd_keys)}"
for k in sd_keys_hf:
    if any(k.endswith(w) for w in transposed):
        # special treatment for the Conv1D weights we need to transpose
        assert sd_hf[k].shape[:-1] == sd[k].shape
        with torch.no_grad():
            sd[k].copy_(sd_hf[k].t())
    else:
        # vanilla copy over the other parameters
        assert sd_hf[k].shape == sd[k].shape
        with torch.no_grad():
            sd[k].copy_(sd_hf[k])

return model
```

شکل ۲۱ بلوک GPT بخش پنجم

تابع "configure_optimizers" مطابق شکل ۲۳ برای تعریف یک بهینه ساز برای مدل PyTorch طراحی شده است. با جمع آوری تمام پارامترهای مدل که برای یادگیری نیاز به گرادیان دارند، شروع می شود. سپس این پارامترها به دو گروه تقسیم می شوند: "decay_params" (معمولاً وزن ها) و "nodecay_params" (معمولاً بایاس). این تقسیم بر اساس تمرین یادگیری عمیق رایج است که در آن کاهش وزن فقط برای وزن های مدل اعمال می شود، نه سوگیری ها. سپس این تابع گروه های بهینه ساز را با این پارامترها و کاهش وزن مربوطه تنظیم می کند.

این تابع همچنین تعداد تانسورهای پارامتر پوسیده و غیر پوسیده و کل پارامترهای آنها را شمارش و چاپ می کند. در نهایت، یک بهینه ساز AdamW با نرخ یادگیری مشخص شده و نسخه بتا ایجاد می کند. اگر نسخه «AdamW fused» موجود باشد و نوع دستگاه «cuda» باشد، از این نسخه برای مزایای بالقوه عملکرد استفاده می کند. تابع بهینه ساز ایجاد شده را برمی گرداند، که می تواند برای به روز رسانی پارامترهای مدل در طول آموزش استفاده شود.

```
def configure_optimizers(self, weight_decay, learning_rate, betas, device_type):
    # start with all of the candidate parameters
    param_dict = {pn: p for pn, p in self.named_parameters()}
    # filter out those that do not require grad
    param_dict = {pn: p for pn, p in param_dict.items() if p.requires_grad}
    # create optim groups. Any parameters that is 2D will be weight decayed, otherwise no.
    # i.e. all weight tensors in matmuls + embeddings decay, all biases and layernorms don't.
    decay_params = [p for n, p in param_dict.items() if p.dim() >= 2]
    nodecay_params = [p for n, p in param_dict.items() if p.dim() < 2]
    optim_groups = [
        {'params': decay_params, 'weight_decay': weight_decay},
        {'params': nodecay_params, 'weight_decay': 0.0}
    ]
    num_decay_params = sum(p.numel() for p in decay_params)
    num_nodecay_params = sum(p.numel() for p in nodecay_params)
    print(f"num decayed parameter tensors: {len(decay_params)}, with {num_decay_params:,} parameters")
    print(f"num non-decayed parameter tensors: {len(nodecay_params)}, with {num_nodecay_params:,} parameters")
    # Create AdamW optimizer and use the fused version if it is available
    fused_available = 'fused' in inspect.signature(torch.optim.AdamW).parameters
    use_fused = fused_available and device_type == 'cuda'
    extra_args = dict(fused=True) if use_fused else dict()
    optimizer = torch.optim.AdamW(optim_groups, lr=learning_rate, betas=betas, **extra_args)
    print(f"using fused AdamW: {use_fused}")

    return optimizer
```

شکل ۲۲ بلوک GPT بخش ششم

```
def estimate_mfu(self, fwdbwd_per_iter, dt):
    """ estimate model flops utilization (MFU) in units of A100 bfloat16 peak FLOPS """
    # first estimate the number of flops we do per iteration.
    # see PaLM paper Appendix B as ref: https://arxiv.org/abs/2204.02311
    N = self.get_num_params()
    cfg = self.config
    L, H, Q, T = cfg.n_layer, cfg.n_head, cfg.n_embd//cfg.n_head, cfg.block_size
    flops_per_token = 6*N + 12*L*H*Q*T
    flops_per_fwdbwd = flops_per_token * T
    flops_per_iter = flops_per_fwdbwd * fwdbwd_per_iter
    # express our flops throughput as ratio of A100 bfloat16 peak flops
    flops_achieved = flops_per_iter * (1.0/dt) # per second
    flops_promised = 312e12 # A100 GPU bfloat16 peak flops is 312 TFLOPS
    mfu = flops_achieved / flops_promised
    return mfu
```

شکل ۲۳ بلوک GPT بخش هفتم

```
@torch.no_grad()
def generate(self, idx, max_new_tokens, temperature=1.0, top_k=None):
    for _ in range(max_new_tokens):
        # if the sequence context is growing too long we must crop it at block_size
        idx_cond = idx if idx.size(1) <= self.config.block_size else idx[:, -self.config.block_size:]
        # forward the model to get the logits for the index in the sequence
        logits, _ = self(idx_cond)
        # pluck the logits at the final step and scale by desired temperature
        logits = logits[:, -1, :] / temperature
        # optionally crop the logits to only the top k options
        if top_k is not None:
            v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
            logits[logits < v[:, [-1]]] = -float('Inf')
        # apply softmax to convert logits to (normalized) probabilities
        probs = F.softmax(logits, dim=-1)
        # sample from the distribution
        idx_next = torch.multinomial(probs, num_samples=1)
        # append sampled index to the running sequence and continue
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

شکل ۲۴ بلوک GPT بخش هشتم

فصل چهارم آموزش مدل و خروجی

آموزش مدل و خروجی

پروژه شامل فایل‌های `train.py`، `model.py`، `configurator.py`، `sample.py`، پوشه `data` که حاوی فایل `prepare.py` و دیتاست `WIKIQA.txt` است و یک پوشه `config` که حاوی یک فایل `train_wikiqa.py` و همین‌طور فایل `BigData_Project3_3_(Chatbot).ipynb` می‌شود. برای مشاهده عملکرد مدل تنها کافیست فایل `BigData_Project3_(Chatbot)` را در محیط کولب و روی GPU T4 اجرا کنید. با اجرای این فایل فرایند آموزش مدل با سایز BATCH معادل ۳۲ شروع شده و یک پوشه جدید با نام `out` در دایرکتوری ساخته می‌شود که آخرین ورژن مدل را بعد از ۱۰۰ checkpoint ذخیره می‌کند. اگر نمی‌خواهید مدل را آموزش دهید و تنها می‌خواهید خروجی‌های مدل `benchmark` را مشاهده کنید سلول `Benchmark` را از حالت کامنت خارج کنید و به جای اجرای سلول `Train model` این سلول را اجرا کنید. نمونه خروجی‌های مدل نیز در فایل `sample output` آورده شده است.

۱-۴--عملکرد مدل-

فایل `train.py` تمامی فرایند آموزش مدل را انجام می‌دهد، پارامترهایی که برای آموزش مدل در نظر گرفته شده اند و مقادیرشان در شکل زیر آورده شده است.

```

51 gradient_accumulation_steps = 5 * 8 # used to simulate larger batch sizes
52 batch_size = 12 # if gradient_accumulation_steps > 1, this is the micro-batch size
53 block_size = 256
54 # model
55 n_layer = 6
56 n_head = 6
57 n_embd = 384
58 dropout = 0.0 # for pretraining 0 is good, for finetuning try 0.1+
59 bias = False # do we use bias inside LayerNorm and Linear layers?
60 # adamw optimizer
61 learning_rate = 6e-4 # max learning rate
62
63 max_iters = 5000 # total number of training iterations
64
65 weight_decay = 1e-1
66 beta1 = 0.9
67 beta2 = 0.95
68 grad_clip = 1.0 # clip gradients at this value, or disable if == 0.0
69 # learning rate decay settings
70 decay_lr = True # whether to decay the learning rate
71
72 # warmup_iters = 2000 # how many steps to warm up for
73 warmup_iters = 200 # how many steps to warm up for
74
75 lr_decay_iters = 6000 # should be ~= max_iters per Chinchilla
76 min_lr = 6e-5 # minimum learning rate, should be ~= learning_rate/10 per Chinchilla
77 # DDP settings
78 backend = 'nccl' # 'nccl', 'gloo', etc.
79 # system
80 device = 'cuda' # examples: 'cpu', 'cuda', 'cuda:0', 'cuda:1' etc., or try 'mps' on macbooks
81 dtype = 'bfloat16' if torch.cuda.is_available() and torch.cuda.is_bf16_supported() else 'float16'
82 compile = True # use PyTorch 2.0 to compile the model to be faster

```

شکل ۲۵ پارامترهای آموزش

لازم به ذکر است که در ابتدا مقادیر اکثر پارامترها متفاوت بود که در جدول زیر مقادیر قبل و بعد و عملکرد مدل با هر دو مجموعه پارامتر آورده شده است.

جدول ۱ مقایسه تاثیر پارامترها بر روی عملکرد مدل

پارامتر	مقادیر اولیه	Loss	MFU	مقادیر ثانویه	Loss	MFU
N_layers	12	5.4	4.5%	6	1.2	3.24%
N_heads	12	5.4	4.5%	6	1.2	3.24%
N_embd	768	5.4	4.5%	384	1.2	3.24%
Max_iters	6000000	5.4	4.5%	5000	1.2	3.24%
Warmup_iters	2000	5.4	4.5%	200	1.2	3.24%
Number of parameters	85M	5.4	4.5%	10.4M	1.2	3.24%

۲-۴- علت ضعف مدل

مدل اولیه که بیشتر شبیه GPT-2 بود دارای ۸۵ میلیون پارامتر بود مشکل اول که با این مدل وجود داشت نیاز به واحد پردازش گرافیکی زیاد و طولانی بودن زمان آموزش (حدود ۲۴ ساعت) بود. متأسفانه نتایج به دست آمده از این مدل فقط از Checkpoint های ۳۰۰ iter بود که خروجی به شدت درهم و بدون کوچک ترین شباهتی به زبان انگلیسی تولید می کرد. مشکل دوم سنگین بودن فایل ckpt.pt خروجی بود که حدود ۵۰۰ مگابایت حجم داشت و استفاده مجدد از آن سخت بود.

۳-۴- راه حل برای بهبود

مشکل اساسی که با مدل اولیه داشتیم این بود که Gpu گوگل کولب برای iteration های بالا جوابگو نبود و آموزش بعد از طی کردن نهایت ۸۰۰ دور با ارور Runtime مواجه می شد. برای بهبود فرایند آموزش آسون ترین راه یعنی کاهش تعداد کل پارامترهای قابل آموزش را در نظر گرفتیم و تعداد لایه ها، سرهای خود توجهی، تعداد دور آموزش و نرخ یادگیری را به طور قابل ملاحظه ای کم کردیم. نتیجه با اینکه هم چنان دارای متن های غیر قابل فهم است ولی به شدت نسبت به حالت قبل به محتوای دیتاست نزدیک تر است. در شکل زیر یک نمونه از خروجی های مدل را ملاحظه می کنید.

what freezes faster ? rategorizations (see below) . 0

what freezes faster ? hot or cold water ? There have been damaged in turn down , gametates is the celebration of gameth holds and the measurement of proposalhers of the book in work Witchiustream Party . 0

when did the vietnam war end The campaign interest and most powerful weeks . 0

when did the vietnam war end The Vietnam War was gulfed by the XPtom(Cup) is a warren and sealed , while enter six beetween 1978 and 1994 (1985) , under t

how many people are in the world According to the Town of Erie and is sometimes known as the November 1865 for the 'ning figure , four number inerways , by which included sides other assesses , such as muscle , too , aeither , and other as rules , to seed up only for the tooth . 0

Who hold make the role of the first successful amendments to win commitmes of the British around the Moon are initially called Islands . 0

who has nellie furtado collaborated with Nelly Lenn `` Prince 's Moon 's sell-f

پیوست ها

فایل گوگل کولب پروژه

https://colab.research.google.com/drive/1hK0Lb-I_tt4oJZ5hInSuTu-BLXmDbb4c?usp=sharing

Abstract

A question-and-answer chatbot is a software bot designed to provide immediate answers to users' questions. These chatbots often support natural language processing (NLP) and machine learning (ML) technologies, allowing them to understand and respond to user questions conversationally. Transformer is an encoder-decoder neural network structure that is very relevant to question-and-answer chatbots due to its ability to handle variable-sized input using stacks of attentional layers instead of RNNs or CNNs. This architecture allows the model to make no assumptions about the temporal/spatial relationships between data, compute layer outputs in parallel, and learn long-range dependencies. These features make Transformers particularly effective for tasks such as text generation and question answering.

Key Words: chatbot, Self-attention mechanism, Transformer



**Amirkabir University of Technology
(Tehran Polytechnic)**

... Department of Management, Science, and Technology ...

Project 3, Section 3

Question answering chatbot

**By
Alireza Keivanimehr
Fatemeh Hosseini Sadi
Mahdi Falsafi**

**Supervisor
Dr. Saeed Sharifian**

**Advisor
Mahdi Amini**

February 2024