پیاده سازی یک مدل زبانی مبتنی بر GPT

Notebook (bigram(an intro to GPT))

Introduction

میخواهیم یک مدل زبانی مبتنی بر ترنسفورمر ترین کنیم.

این مدلی که میخواهیم پیاده سازی کنیم در سطح کاراکتر (Character level or Word chunk level) هست.

اما chatGPT در سطح توکن هست (Token level).

روی دیتاست محدودی ترین میخواهیم بکنیم نه chunk of the internet به اسم tiny Shakespeare که تمام کارهای شکسپیر را شامل میشود.

میخواهیم این را مدل کنیم که کاراکترها چگونه همدیگر را دنبال میکنند.

یک چانک از کاراکترها را به مدل بدهیم بعد شبکه ترنسفورمر کاراکتر بعدی را پیش بینی کند.

و میخواهیم برای ما دنباله ای از کاراکترها مشابه با شکسپیر تولید کند در حقیقت میخواهیم پترن موجود در دیتا را مدل کنیم.

برای اینکه کاراکترهای unique را ببینیم از set استفاده می کنیم که فقط unique ها را نگه میدارد.

در واقع vocab_size تعداد المان های ممکن از دنباله ما را نشان میدهد.

در واقع chars همان vocabulary ما است و کاراکترهای ممکنی است که مدل میتواند ببیند یا منتشر کند.

Tokenization

میخواهیم متن ورودی (input text) را توکنایز کنیم.

منظور از توکنایز تبدیل تکست خام ورودی (به عنوان یک رشته) به دنباله ای از اعداد صحیح مطابق با بعضی لغات از المان های ممکن است.

ما میخواهیم یک character language model بسازیم پس به دنبال ترنسلیت (ترجمه) هر کاراکتر به اعداد صحیح هستیم.

برای این منظور هم انکودر هم دیکودر را میسازیم.

این یکی از راه های Encoding یا Tokenizer هست.

میتوان از sentencepiece (پیاده شده توسط گوگل) نیز به عنوان Tokenizer استفاده کرد که در سطح subword (بین سطوح subwords و character) این کار را انجام میدهد.

یا حتی از tiktoken (پیاده شده توسط openai) به عنوان Tokenizer استفاده کرد. به جای آنکه ۶۵ کاراکتر یا توکن داشته باشد حدودا ۵۰۰۰۰ توکن دارد.

یک تریدآفی بین code book size یا همان vocabulary size (در کد ما ۶۵ در tiktoken حدودا ۵۰۲۵۷) و طول دنباله اعداد صحیح (خروجی انکدینگ یا توکنایزیشن) (مثلا برای "hii there" در کد ما طول دنباله ۹ بود اما در tiktoken برابر ۳ است.) وجود دارد.

اما ما در این مدل همین Character lavel tokenizer ساده را نگه میداریم.

حال میخواهیم کل دیتاست (شکسپیر) را توکنایز کنیم.

دیتا تنسور را که یک دنباله خیلی بزرگ از اعداد صحیح است، تشکیل میدهیم.

Split Data to Train and Validation sets

۹۰ درصد اول دیتاست را به ترین و بقیه را به تست اختصاص میدهیم.

این به ما کمک میکنه تا بفهمیم تا چه حد مدل ما دارد overfit میشود. (نمیخواهیم مدل ما دیتاست را حفظ کند فقط!)

میخواهیم مدل شبکه عصبی ما متنی مشابه متن شکسپیر بسازد.

حال میخواهیم این دنباله اعداد را به ترنسفورمر وارد کنیم تا روی آن ترین شده و الگوی آن را یاد بگیرد.

Chunking

ما تمام دیتاست را به یکباره به مدلمون نمیدهیم چراکه از لحاظ محاسباتی بسیار سنگین خواهد بود.

در عوض ما دیتاست را به صورت chunk (چانکهای کوچک رندوم) شده به مدل میدهیم.

این چانک ها یک حداکثر طول دارند در کد block_size نام دارد شاید در جاهای دیگر اسم آن را context length نیز بگذارند.

در این کد سایز بلاک ۸ در نظر گرفته شده است.

میخواهیم یک چانک از دیتا را سمپل بکنیم:

ابتدا ۸+۱ المان اول دیتای ترین را جدا میکنیم، این یک چانک از دیتاست ترین ما است.

تمام این کاراکترهای موجود در این چانک یکدیگر را دنبال میکنند.

وقتی این چانک را به ترنسفورمر میدهیم مدل ما به صورت همزمان روی آن ترین میشود تا هر یک از پوزیشن های موجود در چانک را پیش بینی کند.

در یک چانک با ۹ کاراکتر (اعداد صحیح) در حقیقت ۸ مثل (example) جداگانه وجود دارد که در آنجا بسته بندی شده است.

مثلا exampleها به این صورت هستند (برای اولین چانک دیتاست ترین): در چارچوب کاراکتر (کاراکتر تبدیل شده به عدد صحیح) ۱۸، احتمالاً ۴۷ کاراکتر بعدی است. در چارچوب ۱۸ و ۴۷، احتمالا ۵۶ کاراکتر بعدی است. در مجموع برای یک چانک ۸ تا از این پیشبینی ها (مثل ها) میتونیم داشته باشیم.

در بلاک ۱۱ام کولب X در حقیقت ورودی ترنسفورمر میباشد y در واقع بلاک کاراکتری بعدی میباشد (آفست برابر ۱). y در حقیقت همان هدف ما برای هر پوزیشن میباشد.

پس ما ۸ مثل داریم که در یک چانک ۹ کاراکتره پنهان شده است.

مدل روی تمام این ۸ مثل ترین میشود نه فقط به دلیل بهینگی محاسباتی بلکه با این رویکرد شبکه ترنسفورمر میتواند محتوا را از کوچک ترین حالت تا بزرگترین حالت (block size) و بین این دو ببیند.

اینجوری ترنسفورمر میدونه چجوری کاراکتر بعدی رو پیش بینی کنه با تمام رویکرد های ۱محتوایی تا بلاک سایز محتوایی!

اما بعد از بلاک سایز آن را قطع میکنیم چون ترنسفورمر بیشتر از بلاک سایز را هرگز دریافت نمیکند.

تا الآن به بعد (dimension) زمان تنسورها پرداختیم.

حال به بعد batch باید بپردازیم (Multiple chunks).

چون میتوانیم از GPU استفاده کنیم مدل میتواند چند چانک را به صورت همزمان و موازی پردازش کند. اما هر چانک از دیگری مستقل است.

Batching

در مرحله بعد یک تابع نوشته شده که یا فراخوانی آن یک batch از دیتاست (شامل ۴ چانک) تولید میشود.

برای آنکه موقعیت های تصادفی از دیتاست را نمونه برداری میکنیم (seed تعریف میکنیم تا نتایج مشابه با مرجع بگیریم).

متغیر batch_size مشخص میکند چه تعداد دنباله ی مستقل (چانک ها) در هر forward-backward ترنسفورمر پردازش میشود.

در خروجی این تابع ما x: ورودی ترنسفورمر که یک تنسور با سایز ۸*۴ خواهد بود (۴ تعداد چانک ها و ۸ تعداد کاراکترهای درون هر چانک یا سایز هر چانک) و y: تارگتها که یک تنسور با سایز ۸*۴ خواهد بود (۴ تعداد چانک ها و ۸ تعداد تارگتهای هر چانک) را داریم.

با اعمال بعد batch ما در واقع ۳۲ example مستقل داریم که درون یک بچ بسته بندی (pack) شده است (هر چانک ۸ example یا به عبارتی پیش بینی! و در کل برای ۴ چانک میشود ۳۲ examples).

ترنسفورمر این ۳۲ example را به صورت همزمان پردازش میکند و به تارگت ها نگاه میکند تا پیشبینی کند هر یک از موقعیت ها را.

Bigram Language Model: The Simplest Neural Network Language Model

ابتدا ورودی ساخته شده را به یک مدل زبانی ساده به اسم bigram میدهیم.

درون کلاس BigramLanguageModel در قسمت Constructor یک token_embedding_table با سایز vocab_size*vocab_size میسازیم.

هر یک از اعداد صحیح موجود در ورودی ما به این embedding table ارجاع داده میشوند. متناظر با هر index ما یک ردیف از embedding table را جدا میکنیم.

مثلا برای عدد صحیح ۲۴ ما به ردیف ۲۴ام از embedding table مراجعه میکنیم.

پایتورچ تمام این ها را در یک تنسور با سایز batch, time, channel (بچ برابر ۴، زمان برابر ۸ و چنل برابر vocab size یا همان ۶۵ میباشد) مرتب میکند.

قرار است این تنسور را به عنوان logits (که اساسا همان امتیاز یا پیشبینی یا احتمال برای کاراکتر بعدی در دنباله میباشد) تفسیر کنیم.

پایتورچ برای محاسبه ی CrossEntropyLoss انتظار دارد که channel در بعد دوم باشد در حالی که برای ما در بعد سوم است به همین علت نمیتواند loss را بین targets و logits حساب کند.

برای راحتی کار محاسبه ی Loss در پایتورچ ابعاد تنسور logits را از B, T, C به B*T, C تغییر میدهیم (از ۳ بعد به ۲ بعد) و سایز targets را از B, T به B*T (از ۲ بعد به ۱ بعد).

قرارهست که روی بعد زمان کاراکتر تولید کنیم تا max_new_tokens برای هر batch.

برای generate از یک تنسور ۱*۱ صفر شروع میکنیم چون به نظر میرسد ۰ که معادل new line در code book ما است برای شروع یک متن شکسپیر مناسب باشد.

چون مدل ما تا به این جا یک مدل تصادفی هست چیزی که تولید میشود هم رندوم و بی محتواست.

مدلی که تا به اینجا ساختیم برای ساخت کاراکتر بعدی تنها به کاراکتر قبل نگاه میکند در حالیکه باید علاوه بر آن به تمام کاراکترهای قبل نیز نگاه کند.

در مرحله ی بعد میخواهیم مدل نوشته شده را ترین کنیم.

برای دیتاست های کوچک نرخ یادگیری ۱e-4 خیلی کوچک است و باید بزرگتر (۱e-3) انتخاب شود.

صفر کردن گرادیان های مرحله ی قبل: optimizer.zero_grad

گرفتن گرادیان برای تمام پارامترها: loss.backward

استفاده از این گرادیان ها برای بروزرسانی پارامترهامون: optimizer.step

اما این مدلی که پیاده شده خیلی ساده است چون توکن ها با یکدیگر صحبت نمیکنند!

و همچنین به این خاطر که متن های تولید شده فقط به کاراکتر انتهایی نگاه میکنند و کاراکتر بعدی را پیشبینی میکنند عملکرد این مدل خوب نیست.

توکنها باید با یکدیگر حرف بزنند تا بفهمند که محتوا (context) چیست، برای این منظور به سراغ ترنسفورمرها میرویم.

Script (bigram.py)

تمام کدهای موجود در نوت بوک bigram برای راحتی و سادگی به یک اسکریپت منتقل شد به همراه تغییرات زیر.
برای استفاده از GPU لازم است دیتاست و پارامترهای مدل را با دستور زیر به gpu منتقل کنیم تا محاسبات آنجا انجام شود که بسیار سریع تر خواهد بود.
.to(device)

در training loop که نوشتیم ما یک معیار خیلی نویزی از loss فعلی داشتیم.
برای رفع این مشکل یک estimate loss function مینویسیم که از lossها روی چند batch میانگین میگیرد.
torch.no_grad یک context manager است و میگوید که تمام چیزهایی که درون تابع زیر اتفاق می افتد ما نمیخواهیم که backward روی آن انجام شود (گرادیان محاسبه نشود no_grad). اینجوری از لحاظ استفاده رم خیلی بهینه تر خواهد بود چراکه مجبور نیست تمام متغیرهای میانی را ذخیره کند.
(مناسب برای زمانی است که نمیخواهیم back propagation انجام شود.)

Script (bigram_v2.py)

در BigramLanguageModel دیگر نیازی به vocab size نداریم چرا به عنوان متغیر global در شروع اسکریپت تعریف شده است.

متغیر n_embd را تعریف میکنیم که در حقیقت ابعاد embedding را تعیین میکند.
self.token_embedding_table دیگر نباید به ما مستقیما logits رو برگردونه در حقیقت باید به ما token embedding رو بده.
برای اینکه از روی token embedding به logits برسیم به یک لایه خطی (self.lm_head) نیاز داریم.
C موجود در tok_emb با C موجود در logits برابر نیستند اولی c embedding هست دومی vocab size.
حال میخواهیم علاوه بر انکود کردن هویت توکن ها موقعیت توکن ها را نیز انکود کنیم پس یک position embedding table میسازیم.
تمام اعداد صحیح از ۰ تا T-1 از طریق position embedding table عملیات embedding روی آن ها اجرا میشود.
حال token embedding + position embedding را محاسبه میکنیم و بعد به lm_head میدهیم تا logits محاسبه شود.
اما مدل bigram خیلی مدل ساده ای است، در این مدل توکن ها نمیتوانند با هم ارتباط برقرار کنند(برای این کار نیاز داریم بلاک self-attention را پیاده سازی کنیم).
position embedding نقش مهمی در self attention block ایفا میکند.

Notebook (Self-Attention)

برای پردازش توکن ها ما نیاز داریم این بلاک را پیاده سازی کنیم. در واقع این بلاک قلب تپنده ی ترنسفورمر هست.
به دنبال این هستیم که توکن های ما بتونن با هم حرف بزنن.

C: در هر نقطه از دنباله ما یکسری اطلاعات در C وجود دارد.

۴ بچ (B)، حداکثر ۸ توکن (T) و ۲ کانال (C) داریم.
برای پیاده سازی self attention رویکرد زیر را پیش میگیریم:

توکن موجود در بعد tام فقط باید اطلاعات توکن های قبل از خود را دریافت کند (با استفاده از میانگین گیری) چون توکن های بعدی رو میخواد پیش بینی کنه.
روش ۱:
میانگین گیری با استفاده از حلقه ی for روی توکن ها در هر کانال (بهینه نیست!)
در هر ردیف (که مرتبط با هر توکن است) اطلاعات توکن های قبل تا همان ردیف میانگین گیری میشود.

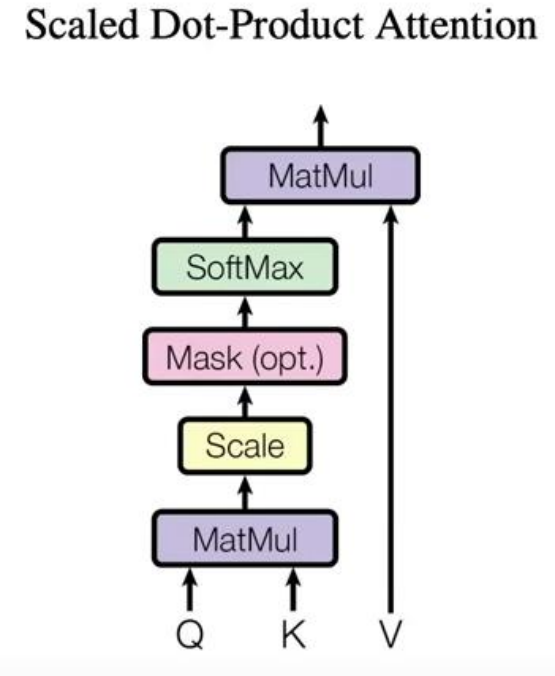
روش ۲:
جمع وزن دار: روش بهینه تر با استفاده از ضرب ماتریسی، هر batch به صورت موازی ضرب ماتریسی روش انجام میشه.

روش ۳:
استفاده از softmax مزیت این روش این است که
wei که درون آن صفر ریخته شده است (torch.zeros) در هر ردیف به ما میگوید چه مقدار از توکن های گذشته را میخواهیم میانگین بگیریم.
در masked_fill کاری میکنیم که توکن مورد بحث نتونه با توکن های آینده ارتباط برقرار کنه.
بعد با استفاده از softmax نرمالیزه می کنیم و بعد با استفاده از ضرب ماتریسی میانگین میگیریم.
از این روش ۳ برای پیاده سازی Self attention اصلی استفاده میکنیم.

Self Attention Block (main)

روش ۴:
میخواهیم یک self attention کوچک برای یک head پیاده سازی کنیم.
سایز بچ برابر ۴، سایز زمان برابر ۸ و سایز کانال برابر ۳۲.
اطلاعات موجود در هر توکن در ابعاد ۳۲ تایی قرار دارد (همان C).
اطلاعات توکن های قبلی و توکن فعلی با هم ترکیب می شود.
نمیخواهیم که wei توزیع یکنواخت داشته باشد چون برای توکن های مختلف سایر توکن ها به یک میزان جالب نیستند و ما میخواهیم این وابسته به دیتا باشد.
این مسئله ای است که self attention برای ما حل می کند.
برای هر تک توکن در هر پوزیشن دو بردار تعریف میشود، یک query (میگه من به دنبال چی هستم؟) و یک key (من چه چیزی را شامل میشوم).
روشی که ما با آن وابستگی بین توکن ها را در یک دنباله به دست می آوریم، ضرب داخلی بین query و key می باشد.
query ضرب داخلی می شود با تمام keyهای بقیه ی توکن ها و نتیجه می شود همان wei.
اگر key, query با هم همسو باشند ضرب داخلی آن ها مقدار بالایی خواهد داشت.

Head of Self Attention (Scaled Dot-Product Attention)



یکسری هایپرپارامترها نیاز داریم که برای این head تعریف کنیم. مثلا head size.
این پارامتر head size در حقیقت ابعاد بردارهای query, key, value را مشخص می کند. در ادامه با این بردارها بیشتر آشنا خواهیم شد.
ماژول linear صرفا ضرب ماتریسی با ضرایب ثابت انجام میدهد بایاس هم نداریم اینجا.
تمام توکن ها در تمام پوزیشن ها به صورت موازی و کاملا مستقل یک key, query تولید میکنند (هنوز هیچ ارتباطی بین آنها شکل نگرفته است).
ارتباط وقتی شکل میگیره که تمام queryها با تمام keyها ضرب داخلی شوند.
حال که query, key را با هم ضرب کردیم نتیجه میشود wei با سایز B, T, T. (دقت شود که query باید transpose شود تا ضرب ماتریسی انجام شود.)
(سایز key: B, T, 16 و سایز query: B, T, 16).
با ضرب داخلی key, value یک روش وابسته به داده پیاده سازی کردیم به جای اینکه wei را از صفر شروع کنیم.
قبلا wei برای هر batch یکسان بود اما الان هر batch وزن های متفاوتی دارد چرا که توکن های متفاوتی را شامل میشود.
آخرین ردیف از بچ اول wei توکن ۸ام میباشد و توکن ۸ام میداند چه محتوایی و چه پوزیشنی دارد.
دوباره با استفاده از masking نباید اجازه بدهیم هر توکن با توکن ها ی بعدی ارتباط بگیرد. بعد از آن برای نرمالیزه کردن از softmax استفاده میکنیم.
با این کار توانستیم این مفهوم را پیاده سازی کنیم که چه مقدار اطلاعات از توکن های قبلی تجمیع میشود.

در self attentaion ما دقیقا توکن ها (x) را تجمیع نمی کنیم بلکه value آن ها را تجمیع میکنیم، برای این کار متغیر value را تعریف میکنیم.
یک مثال ساده: توکن nام یکسری اطلاعات دارد (identity) که اطلاعات آن در وکتور x ذخیره شده است، چیزی که توجه توکن را جلب میکند query است، چیزی که دارد key هست و چیزی که در v ذخیره می شود آن چیزی است که توکن می گوید اگر من توکن مورد نظر شما باشم با آن با شما ارتباط می گیرم.

attention یک مکانیزم ارتباطی است که در آن هر نود (توکن) یک وکتور از اطلاعات دارد و آن نود از طریق جمع وزن دار تمام نودهایی که به او اشاره میکنند، تمام اطلاعات را تجمیع میکند.
نود اول فقط به خودش اشاره میکند. نود دوم به نود اول و خودش اشاره میکند. تا نود هشتم که به تمام نودهای قبل از خود و خودش اشاره میکند.

همچنین به هر نود اطلاعات موقعیت آن را هم میدهیم. (positional embedding) این برخلاف عملیات convolution هست چرا که اطلاعات مکانی در عملیات convolution به طور ذاتی وجود دارد اما در attention باید جداگانه اضافه شود.

وقتی ۴ بچ داریم و ۸ توکن انگار ۴ فضای مجزا داریم که در هر یک ۸ توکن دارن با هم ارتباط میگیرند و این ۴ فضای مجزا هیچگونه ارتباطی با هم نمیگیرند و تماما پردازش به صورت موازی انجام میشود.

نکته: در اینجا ما اجازه نمیدهیم توکن های آینده با گذشته ارتباط برقرار کنند اما در حالت کلی میتونه این طوری نباشه مثلا در آنالیز احساس جمله میتونیم اجازه بدیم تمام توکن ها با همدیگه ارتباط برقرار کنند.

در ساختار ترنسفورمر قسمت Encoder ما نیازی نداریم که اجازه ندهیم توکن های آینده با گذشته ارتباط برقرار کنند (از طریق masked_fill) این محدودیت را فقط در بلاک Decoder پیاده سازی میکنیم.

چرا اسمش self attention هست چون key, query, value همگی از یک منبع (x) می آیند. اما مفهوم attention گسترده تر است مثلا این سناریو میتواند اتفاق بیفتد که queries از x تولید بشود اما keys, values از یک منبع خارجی (مثلا بلاک encoder) تولید بشوند به این حالت میگویند cross attention. (یک منبع خارجی از نودها وجود دارد که میخواهیم از آنها اطلاعات را منتقل کنیم به درون نودهای خودمان.)

فرمول attention:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

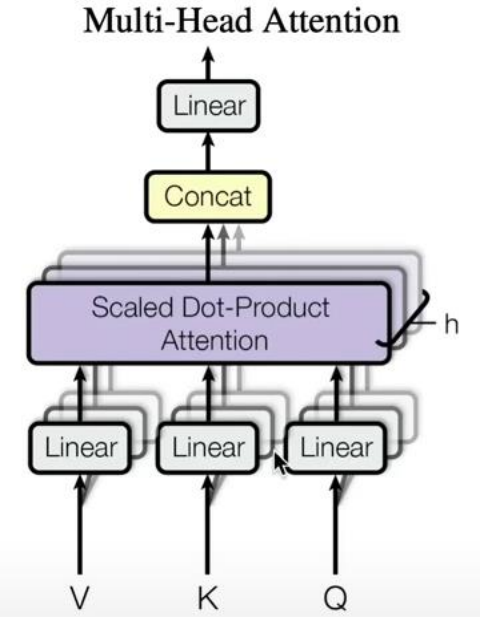
dk همان head size است. تقسیم بر رادیکال dk رو بهش میگن scaled attention.
اگر این scale را انجام ندهیم واریانس wei در مرتبه ی head size (حول و حوش ۱۶) خواهد بود، اما اگر انجام بدهیم اطراف ۱ خواهد بود.
در مراحل مقدار دهی اولیه خیلی مهم هست که wei پراکنده (diffuse) باشد اگر واریانس کنترل نشود در مقداردهی اولیه، softmax مقادیر ورودی خود را خیلی شارپ می کند.

Script (bigram_attention.py)

در کد کلاس Head نوشته می شود که در واقع self attention را پیاده و به مدل زبانی bigram اعمال می کند.
می بینیم که loss با اعمال self-attention از ۲.۵ به ۲.۴ کاهش پیدا می کند که نشانه ی پیشرفت مدل می باشد.

Script (bigram_MHA.py)

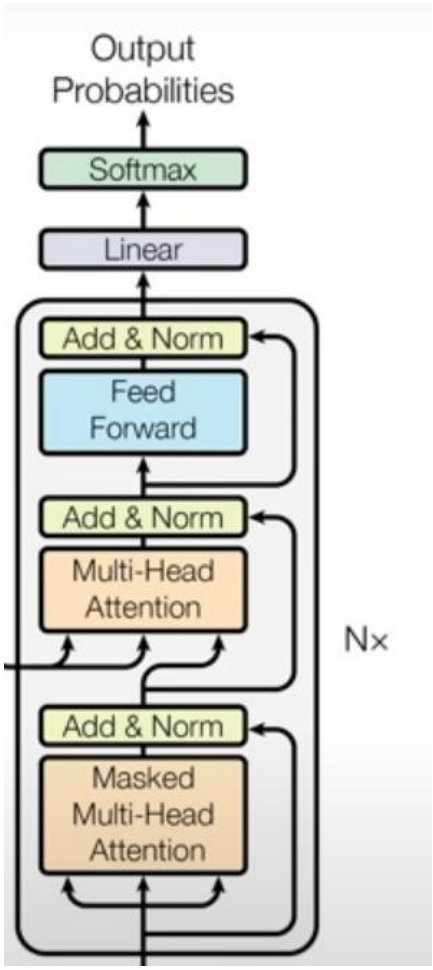
Multi-Head Attention



چندین attention را به صورت موازی انجام می دهد و نتایج را با هم روی بعد کانال، concatenate می کند.
تعداد headها ۴ تنظیم شده است به نحوی که هر head هشت بعد دارد و روی این ۴ head بصورت موازی پردازش انجام میشود و در نهایت همه ی ۸ بعد ۴ head با هم concatenate شده و ۳۲ یعنی n_embd را میسازند.
به کد مرحله ی قبل کلاس MultiHeadAttention اضافه می شود که در واقع MultiHeadAttention را پیاده و به مدل زبانی bigram اعمال می کند.
همچنان متن تولید شده خوب نیست با این که ساختار شکسپیر را حفظ کرده است، اما می بینیم که loss با اعمال MultiHeadAttention از ۲.۴ به ۲.۲ کاهش پیدا می کند که نشانه ی پیشرفت مدل می باشد.

Script (bigram_FF.py)

Position-wise Feed Forward Network



بخش Feed Forward در قسمت Decoder شبکه ی transformer در واقع یک Multi Layer Perceptron است.

با یک MLP کوچک ساده محاسبات را به شبکه وارد میکنیم. محاسبات در سطح هر توکن اعمال می شود.

توکن ها با یکدیگر ارتباط برقرار میکنند اما زمان کافی برای فهمیدن اینکه چه چیزی از بقیه توکن ها فهمیده اند را ندارند.

با استفاده از مکانیزم attention دیتا جمع می شود (communication) و بعد با Feed Forward روی دیتا فکر می شود (computation).

Script (bigram_RC.py)

Residual Connection

Class Block در واقع همون بلاک دیکودر در ساختار ترنسفورمر هست به جز بخش Cross Attention.

بلاک از بخش های

communication (Multi-Head Self Attention), computation (Feed Forward Network)

تشکیل شده است.

Number of embedding (Embedding dimension): 32
Number of head: 4
Head size: 8

این مقادیر این امکان را برای ترنسفورمر فراهم می کنند که همه چیز به صورت channel wise کار کند.

self_lm کار decode را انجام میدهد.

اما این شبکه (Bigram+MultiHead+FFN) به خوبی کار نمیکند چرا که با یک شبکه عصبی عمیق کار می کند که مشکلات بهینه سازی دارد.

برای رفع این مشکل ۲ روش بهینه سازی به کار میبرن:

مورد یک از Skip connection or Residual connection استفاده کرده اند. x = x + self.sa x

عملگر جمع، گرادیان را به صورت مساوی توزیع میکند. به همین علت از عملگر جمع در Residual connection استفاده کرده اند.

با اعمال این قسمت به کد، loss تا ۲.۰۹ کاهش پیدا میکند و متن تولید شده نه تنها ساختار شکسپیر را ارائه میکند بلکه بسیار شبیه به زبان انگلیسی متن ها را تولید میکند.

```
step 0: train loss 4.6328, val loss 4.6313
step 500: train loss 2.3721, val loss 2.3678
step 1000: train loss 2.2581, val loss 2.2624
step 1500: train loss 2.1730, val loss 2.1980
step 2000: train loss 2.1311, val loss 2.1731
step 2500: train loss 2.0985, val loss 2.1464
step 3000: train loss 2.0638, val loss 2.1337
step 3500: train loss 2.0500, val loss 2.1063
step 4000: train loss 2.0207, val loss 2.1038
step 4500: train loss 1.9972, val loss 2.0936

Whim?
Whirkince.

SOROLOUS:
Which the buby to tarther'ds me?
Bearse:
Warther usquet to bardethat a endway, my facks a wizr our
Yourselvef is heart mile dill, begrisee:
You, milal Het drove the deave ther on you muselflind me uptence, on him sobught is all, yet lord.
In am patelives
Mome:
Whod moth keed Winso what evings the modourive ches, me sto-deal the the deard nunrupter son; if himm:
'T thy fled
at that Prike my of.

HENRY:
Hartioblisan adape the age, thisin cour aard no I whist,
And for hi
```

Script (bigram_LN.py)

Layer Normalization

مورد دو LayerNorm هست.

layer norm بسیار به batch norm شبیه است با این تفاوت که Class BatchNorm1D روی batch ها (روی ستون ورودی) نرمالیزیشن را اعمال میکند (میانگین ۰ و انحراف معیار ۱ می شود(توزیع گاوسی)) اما Layer Norm روی ردیف نرمالیزیشن را انجام می دهد.

در ۵ سال گذشته تغییرات کمی در transformer ها ایجاد شده است اما جدیدا خیلی رایج است که Layer Norm را قبل از Transformation همان MLP یا Feed Forwrd اعمال می کنند (به آن می گویند pre-norm formulation).

Layer Norm ویژگی های ما را نرمال میکند.

میبینیم که با اضافه کردنLayer Norm میزان loss تا ۲.۰۶ کاهش پیدا کرده است و متن تولید شده به زبان انگلیسی و ساختار شکسپیر شبیه تر شده است.

```
step 0: train loss 4.3103, val loss 4.3100
step 500: train loss 2.3805, val loss 2.3801
step 1000: train loss 2.2506, val loss 2.2551
step 1500: train loss 2.1568, val loss 2.1838
step 2000: train loss 2.1184, val loss 2.1584
step 2500: train loss 2.0698, val loss 2.1253
step 3000: train loss 2.0395, val loss 2.1215
step 3500: train loss 2.0329, val loss 2.1008
step 4000: train loss 2.0049, val loss 2.0903
step 4500: train loss 1.9901, val loss 2.0929
step 5000: train loss 1.9752, val loss 2.0614

Whent I not
wcowfarchis and the a seep obe die.
Save-daved agatands:
Warthienus him vetbard that ane away, my feanstarry of my
Yourselfof is heart my would buges is ensent, night Hervirevett, and Wind.

DUKE Wanses liking tear.
-huch one my speap; and plitty. Haventure afdpets aves
And this demeth?

Korling; what evicks with toour will now; now poor of his but thand thrust for are
grean whith is ale oftates dare?

KISTINGSHKING ELLIO:
Prisar adave and Edwast hiin coup aare no vry to chan you!
```

Script (bigram_SU_Final.py)

Scaling up

برای scale کردن مدلی که داشتیم تعداد لایه های Block را تعیین میکنیم با متغیر n_layer همان N.

همچنین تعداد headها را نیز درون یک متغیر میریزیم به نام n_head.

یک لایه Layer Norm بعد از آخرین Bloack قرار میدهیم.

همچنین dropout را به مدلمان وارد میکنیم. این لایه از overfitting جلوگیری میکند.

عملکرد dropout این است که به برخی از nodeها به صورت تصادفی اجازه نمی دهد با یکدیگر ارتباط بگیرند.

همچنین مقادیر برخی هایپرپارامترها را به منظور تقویت و بزرگ کردن مدل نیز تغییر داده ایم.

batch_size = 32 → 64
block_size = 8 → 256
learning_rate = 1e-3 → 3e-4
n_embd = 32 → 384
n_head = 4 → 6
n_layer = 4 → 6
dropout = 0 → 0.2

در ویدیو گفته شده است کد scale شده روی A100 GPU اجرا شده و ۱۵ دقیقه طول کشیده است. من روی گوگل کولب با GPU کد مربوط به مدل scale شده را اجرا کردم و برای ۵۰۰۰ iteration ترین شدن مدل ۵۰ دقیقه طول کشید.

```
step 500: train loss 1.8866, val loss 2.0025
step 1000: train loss 1.5356, val loss 1.7221
step 1500: train loss 1.3950, val loss 1.6046
step 2000: train loss 1.3079, val loss 1.5495
step 2500: train loss 1.2523, val loss 1.5167
step 3000: train loss 1.2004, val loss 1.4906
step 3500: train loss 1.1592, val loss 1.4795
step 4000: train loss 1.1220, val loss 1.4814
step 4500: train loss 1.0846, val loss 1.4726

RICHARD:
Where come buy steeds un thy fame?
For my Capider, the Ludlanc of Warwick's groan;
Where stood is to Harry, no more hour.
The in the Fred by my sall-body, and toys
What made mercy sleep the recomples, make him right;
Meant some changed your own land hence on the gard,
He could knock me to fled thy tred thy lodg-head!
As tall your again, am i' famour tomb!
You call me a glad nose, ay's leave, you may.
Do that leave it with this conscience, I do six you
You his rephising report. But he's
```

با اجرای مدل scale شده loss از ۲.۰۶ به ۱.۴۸ کاهش پیدا میکند که کاهش بسیار خوبی را نشان میدهد و متن تولید شده بسیار به ساختار شکسپیر و زبان انگلیسی نزدیک است و اکثر کلمات آن نیز به تنهایی معنا دارند اما جملات تولید شده خیلی مفهومی ندارند.

پس ما توانستیم یک مدل ترنسفورمر که در سطح کاراکتر روی دیتاست متنی tiny shakespeare با ۱ میلیون کاراکتر آموزش داده شد متن مشابه با شکسپیر را تولید کنیم.

اجرای کد روی CPU بسیار زمانبر خواهد بود و توصیه نمیشود. شاید بتوان با تغییر تعداد لایه ها به کمتر از ۶ کمتر و کاهش ابعاد embedding، به زمان معقول تری دست پیدا کرد.
در ویدیو فقط قسمت decoder موجود در مقاله transformer را پیاده کرده است و قسمت encoder را پیاده نکرده است چرا که هدف تنها تولید text مشابه متن شکسپیر است.

در مقاله اصلی چون میخواهد یک machine translation بسازد به این دو بلاک احتیاج دارد.
در بخش دیکودر ۲ توکن ویژه به اسامی Start, End داریم. در قسمت انکودر هیچ maskای وجود ندارد و تمام توکن ها اجازه دارند با هم صحبت کنند تا در نهایت تمام محتوای موجود را انکود کنند. در خروجی انکودر ما یک تقاطع با دیکودر داریم که cross attention را میسازد.
queris همچنان از x تولید میشود اما keys, values از طرف encoder می آیند.
در این ویدیو مدلی که ساختیم فقط از روی متن شکسپیر تقلید میکند.

Chat GPT

chat gpt از دو بخش اصلی تشکیل شده است:

1. Pre-Training

مدل Transformer را روی یک چانک بزرگ از اینترنت ترین می کنیم و سعی می کنیم فقط یک Transformer Decoder داشته باشیم. تا این مرحله هنوز مدلی نداریم که به سوالات جواب بدهد و کمک کننده باشد صرفا نوعی document completer هست.

2. Fine-Tuning

در این بخش ۳ مرحله وجود دارد:

۱- جمع آوری داده ی آموزشی به طور ویژه مشابه با یک assistant. یک حالت مشابه Q/A. یک تعداد زیادی از این documentها جمع آوری میشود. وقتی سوال پرسیده میشود با این documentها مقایسه شده و آن که بیشترین همسویی را دارد انتخاب میشود.

۲- به مدل اجازه میدهیم پاسخ تولید کند و چند Rater آن پاسخ ها را بررسی و بر اساس ترجیحات بین جواب ها rank می کنند. در این مرحله یک reward model آموزش داده میشود که چه مقدار از پاسخ ها مطلوب است.

۳- با استفاده از Reinforcement learning یک ارزیابی از عملکرد reward model انجام میدهد.

بسیاری از اطلاعات و داده های قسمت Fine-Tuning در دسترس نیست و این بخش قابلیت تکرار پذیری ندارد.

در این مرحله هست که مدل ما مانند یک دستیار عمل خواهد کرد. در واقع روی داکیومنت هایی مشابه با پرسش و پاسخ fine tune می شود.

NanoGPT که یکی دیگر از پروژه های در حال اجرای سازنده ی ویدئو هست که مدل پیچیده تری را ارائه میکند نیز صرفا روی بخش Pre-Training کار می کند.