تمرین دوم

نام درس: یادگیری عمیق

استاد درس: دکتر محمدرضا محمدی

نام: محمد حقیقت

شماره دانشجویی: 403722042

گرایش: هوش مصنوعی

دانشکده: مهندسی کامپیوتر

نیم سال دوم 1403-1404

سوال اول

الف)

1. حجم بالای KV Cache

حافظه مورد نیاز برای ذخیره‌سازی بردارهای Key-Value (KV Cache) به سرعت با افزایش درخواست‌ها رشد می‌کند.

- برای مثال، در مدل OPT-13B، ذخیره‌سازی KV Cache تنها برای یک توکن به ۸۰۰ کیلوبایت فضا نیاز دارد.
- اگر طول توالی به ۲۰۴۸ توکن برسد، هر درخواست ۱.۶ گیگابایت حافظه مصرف می‌کند.
- با محدودیت حافظه GPU ها (معمولا چند ده گیگابایت)، این موضوع تعداد درخواست‌های همزمان قابل پردازش را به شدت محدود می‌کند.

۲. تکه‌تکه شدن حافظه (Fragmentation)

الف) تکه‌تکه شدن داخلی(Internal Fragmentation)

سیستم‌های فعلی حافظه را بر اساس حداکثر طول ممکن توالی (مثلا ۲۰۴۸ توکن) تخصیص می‌دهند، در حالی که طول واقعی خروجی‌ها معمولا بسیار کوتاه‌تر است. این باعث هدررفت فضای حافظه می‌شود.

ب) تکه‌تکه شدن خارجی (External Fragmentation)

تخصیص حافظه به صورت بلوک‌های پیوسته با اندازه‌های متفاوت منجر به ایجاد فضاهای خالی غیرقابل استفاده بین درخواست‌ها می‌شود.

پروفایلینگ نشان می‌دهد که تنها ۲۰.۴% تا ۳۸.۲% از حافظه KV Cache واقعا مورد استفاده قرار می‌گیرد!

۳. عدم امکان اشتراک‌گذاری حافظه KV Cache

در الگوریتم‌های پیشرفته مانند نمونه‌برداری موازی (Parallel Sampling) یا جستجوی پرتو (Beam Search)، توالی‌های مختلف یک درخواست می‌توانند بخشی از KV Cache را به اشتراک بگذارند.

مثلاً در نمونه‌برداری موازی، تا ۱۲% صرفه‌جویی حافظه امکان‌پذیر است.

در جستجوی پرتو، این مقدار به ۵۵% می‌رسد.
اما سیستم‌های فعلی به دلیل ذخیره‌سازی پیوسته و غیرمنعطف، از این مزیت بی‌بهره می‌مانند.

۴. طول متغیر و غیرقابل پیش‌بینی ورودی/خروجی

طول پرامپت (ورودی) و تولید خروجی در LLM ها از قبل مشخص نیست و در حین پردازش تغییر می‌کند.

این موضوع مدیریت حافظه را پیچیده می‌کند، زیرا سیستم باید همزمان:

- از توالی‌های کوتاه پشتیبانی کند.
- با رشد خروجی، حافظه بیشتری اختصاص دهد.
- از اتمام حافظه (OOM) جلوگیری کند.

۵. الگوریتم‌های رمزگشایی پیچیده

روش‌هایی مثل Beam Search و Parallel Sampling نیازمند مدیریت پویای حافظه هستند، اما سیستم‌های فعلی به دلیل محدودیت‌های تخصیص پیوسته، قادر به پشتیبانی بهینه از آن‌ها نیستند.

۶. رزرو غیرضروری حافظه

سیستم‌های موجود برای هر درخواست، حافظه را برای حداکثر طول ممکن رزرو می‌کنند، حتی اگر نیاز واقعی بسیار کمتر باشد.

این کار ظرفیت دسته‌بندی (Batching) را کاهش می‌دهد و از پردازش همزمان درخواست‌های بیشتر جلوگیری می‌کند.

ب)

PagedAttention به چالش‌های کلیدی در سرویس‌دهی به مدل‌های زبانی بزرگ (LLMs) از جنبه‌ی مدیریت حافظه پاسخ می‌دهد. این چالش‌ها شامل اتلاف حافظه به دلیل تکه‌تکه شدن، ناتوانی در اشتراک حافظه بین درخواست‌ها، و دشواری در پیش‌بینی طول دنباله‌هاست.

رفع تکه‌تکه شدن حافظه
سیستم‌های پیشین معمولا حافظه‌ی KV Cache را به‌صورت پیوسته برای هر درخواست رزرو می‌کنند، که به دلیل عدم تطابق بین حافظه‌ی رزروشده و حافظه‌ی واقعا استفاده‌شده منجر به تکه‌تکه شدن داخلی و خارجی می‌شود.
PagedAttention با تقسیم KV Cache به بلوک‌هایی کوچک‌تر (مانند صفحه‌های حافظه مجازی در سیستم‌عامل‌ها) و تخصیص آن‌ها به‌صورت پویا، استفاده‌ی موثرتری از حافظه ممکن می‌سازد و این تکه‌تکه شدن را عملا از بین می‌برد.

امکان اشتراک حافظه بین درخواست‌ها
در سناریوهایی مانند نمونه‌گیری موازی یا beam search ، بخش‌هایی از KV Cache مانند اطلاعات مربوط به prompt مشترک بین چند دنباله قابل اشتراک است. با استفاده از مدل بلوکی، PagedAttention امکان به‌اشتراک‌گذاری این بلوک‌ها را بین درخواست‌ها فراهم می‌کند، که پیش‌تر به دلیل ساختار پیوسته حافظه ممکن نبود.

مدیریت انعطاف‌پذیر برای طول دنباله‌های نامشخص
با رشد تدریجی KV Cache در طول فرآیند تولید خودبازگشتی (autoregressive)، نیاز به حافظه در طول زمان افزایش می‌یابد و طول نهایی دنباله مشخص نیست.

PagedAttention با نگاشت بلوک‌های منطقی به فیزیکی به‌صورت پویا، به سیستم اجازه می‌دهد تا بدون رزرو حافظه برای حداکثر طول ممکن از ابتدا، به‌صورت تدریجی حافظه را تخصیص دهد.

پشتیبانی از الگوریتم‌های رمزگشایی پیچیده
طراحی بلوک‌محور این الگوریتم امکان پیاده‌سازی الگوریتم‌های متنوع مانند beam search و shared prefix را فراهم می‌کند، بدون نیاز به کپی‌های اضافی یا پیچیدگی‌های مدیریت حافظه.

در مجموع، PagedAttention با الهام از مفاهیم حافظه‌ی مجازی سیستم‌عامل، چارچوبی ارائه می‌دهد که حافظه را انعطاف‌پذیر، بهینه و قابل اشتراک‌گذاری می‌سازد و بدین ترتیب توان عملیاتی سرویس‌دهی مدل‌های زبانی بزرگ را به‌طور محسوسی افزایش می‌دهد.

ج)

اشتراک‌گذاری حافظه‌ی کش (Key-Value Cache) KV در سیستم‌های خدمات‌دهی به مدل‌های زبانی بزرگ (LLM) از اهمیت زیادی برخوردار است، زیرا این حافظه بخش قابل توجهی از منابع GPU را اشغال می‌کند و به‌طور مستقیم بر ظرفیت و توان عملیاتی سیستم اثر می‌گذارد.

اهمیت اشتراک‌گذاری حافظه‌ی کش KV :

کاهش مصرف حافظه‌ی تکراری:
در بسیاری از موارد، چند دنباله‌ی خروجی در یک درخواست مثلا در parallel sampling یا beam search یک بخش مشترک از ورودی (prompt) دارند. اگر برای هر دنباله نسخه‌ی جداگانه‌ای از حافظه‌ی KV ذخیره شود، حجم زیادی از حافظه بی‌جهت تکرار می‌شود.

افزایش تعداد درخواست‌های همزمان:
با اشتراک‌گذاری حافظه، استفاده‌ی موثرتر از فضای حافظه‌ی محدود GPU امکان‌پذیر شده و تعداد بیشتری از درخواست‌ها را می‌توان به صورت هم‌زمان پردازش کرد، که این امر مستقیما توان عملیاتی سیستم را افزایش می‌دهد.

افزایش انعطاف در پیاده‌سازی الگوریتم‌های پیشرفته:
الگوریتم‌هایی مثل beam search از دنباله‌های متعددی استفاده می‌کنند که ساختار درختی دارند و در بسیاری از مراحل دارای بخش‌های مشترک‌اند. اشتراک‌گذاری حافظه در این الگوریتم‌ها باعث کاهش نیاز به کپی‌کردن مداوم KV cache بین شاخه‌های مختلف می‌شود.

vLLM چگونه اشتراک‌گذاری حافظه‌ی KV را آسان می‌کند؟

vLLM با استفاده از الگوریتم PagedAttention و طراحی بلوک‌محور (block-based) حافظه، اشتراک‌گذاری حافظه‌ی KV را در چند سطح امکان‌پذیر می‌سازد:

نگاشت بلوک‌های منطقی به بلوک‌های فیزیکی:
هر دنباله به‌جای آن‌که حافظه‌ی KV خود را به‌طور پیوسته داشته باشد، آن را به‌صورت بلوک‌هایی از داده ذخیره می‌کند. این بلوک‌ها در حافظه‌ی فیزیکی GPU قابل اشتراک‌گذاری هستند و می‌توانند هم‌زمان به چند دنباله‌ی مختلف تخصیص یابند.

استفاده از شمارنده‌ی ارجاع (Reference Count):
برای هر بلوک فیزیکی، شمارنده‌ای نگه‌داری می‌شود تا تعداد دنباله‌هایی که از آن بلوک استفاده می‌کنند را مشخص کند. به‌محض کاهش این شمارنده به صفر، آن بلوک آزاد می‌شود.

مکانیزم Copy-on-Write :
اگر دو دنباله‌ی مختلف یک بلوک مشترک دارند و یکی از آن‌ها قصد تغییر آن را دارد، vLLM ابتدا آن بلوک را کپی می‌کند و نسخه‌ی جدید را تنها برای دنباله‌ی مورد نظر در نظر می‌گیرد. این سازوکار درست شبیه سیستم‌های حافظه‌ی مجازی در OS است.

پشتیبانی از سناریوهای مختلف اشتراک‌گذاری:

Parallel Sampling: دنباله‌های مختلفی از یک prompt مشترک ایجاد می‌شوند و بلوک‌های مربوط به prompt بین آن‌ها به اشتراک گذاشته می‌شود.

Beam Search: شاخه‌های مختلف یک دنباله می‌توانند بلوک‌هایی از مراحل اولیه را به اشتراک بگذارند، حتی با تغییر مسیرهای خروجی.

Shared Prefix: در برنامه‌های چندزبانه یا سیستم‌های توصیه‌گر، می‌توان بلوک‌های مربوط به پیش‌وندهای مشترک بین کاربران مختلف را از پیش محاسبه و در حافظه نگه داشت.

پیامدها برای توان عملیاتی سیستم:

افزایش محسوس در تعداد درخواست‌های قابل سرویس‌دهی هم‌زمان:
در آزمایش‌ها، vLLM توانسته است تا بیش از 2 تا 4 برابر تعداد بیشتری از درخواست‌ها را نسبت به سیستم‌های پیشین مانند Orca در یک batch جای دهد.

صرفه‌جویی در حافظه:
بر اساس داده‌های مقاله، میزان صرفه‌جویی حافظه از اشتراک‌گذاری بلوک‌ها در parallel sampling بین 6% تا 10% و در beam search تا 55% نیز گزارش شده است.

کاهش تاخیر (latency) در پردازش:
با استفاده‌ی موثرتر از حافظه و کاهش نیاز به کپی‌های اضافی، تاخیر ناشی از عملیات حافظه کاهش یافته و در نتیجه زمان پاسخ‌دهی به درخواست‌ها نیز بهتر شده است.

سوال دوم

الف)

چرا از توجه چندسری (Multi-Head Attention) استفاده می‌شود؟

چون اگر فقط یک سری توجه (attention head) داشته باشیم، مدل فقط می‌تونه یک نوع رابطه یا الگو بین کلمات رو یاد بگیره. اما زبان خیلی پیچیده‌تر از اینه؛ ممکنه لازم باشه مدل هم‌زمان چند جور ارتباط رو بررسی کنه.

مثلا: کلمه‌ها چطور از نظر معنایی به هم ربط دارن

ساختار دستوری جمله چطوره

کدوم کلمه به کدوم ارجاع داره (مثلا "او" به کی برمی‌گرده)

با چند head مختلف، مدل می‌تونه هر head رو روی یک نوع الگوی خاص تمرکز بده. بعد همه این اطلاعات ترکیب می‌شن و نمایی کامل‌تر از جمله به مدل داده می‌شه.

این سرهای توجه چه نوع اطلاعاتی را می‌توانند یاد بگیرند؟

هر head می‌تونه یک نوع خاص از رابطه بین کلمه‌ها رو یاد بگیره. برای مثال:

یکی ممکنه یاد بگیره که ضمیرها به چه اسمی برمی‌گردن

یکی ممکنه دنبال رابطه بین فعل و فاعل باشه

یکی دیگه ممکنه تمرکز کنه روی شباهت معنایی کلمات

در نتیجه، هر attention head به نوعی دیدگاه مخصوص به خودش رو از جمله می‌سازه و در نهایت، ترکیب این دیدگاه‌ها باعث می‌شه مدل درک بهتری از ساختار و معنای جمله داشته باشه.

فرض کنید یك مدل آموزش دیده داریم که بر پایه ي توجه چندسري multi-head attention ساخته شده است و مي خواهیم براي افزایش سرعت پیش بیني، سرهاي توجه کم اهمیت تر را حذف Prune کنیم. چگونه مي توانیم آزمایش هایي طراحي کنیم تا اهمیت هر سر توجه را اندازه گیري کنیم؟

ب)

1. خاموش کردن (mask کردن) تک‌تکhead ها و بررسی تأثیر آنها

در این روش:

یکی‌یکی سرهای توجه رو موقتا غیرفعال می‌کنیم مثل اینکه اون head اصلا وجود نداشته باشه.

مدل رو روی داده تست اجرا می‌کنیم.

اگر غیرفعال کردن یک head تاثیر زیادی روی عملکرد مدل (مثل کاهش دقت) نذاشت، یعنی اون head احتمالاً کم‌اهمیته.

1. استفاده از وزن‌های یادگرفته‌شده head ها

گاهی می‌تونیم ببینیم که خروجی بعضی head ها همیشه خیلی کوچک یا بی‌تاثیر هستن، یعنی مدل خودش یاد گرفته زیاد به اون‌ها توجه نکنه. اینو میشه از مقدار نرم وزن‌ها یا خروجی‌ها فهمید.

برای مثال:

محاسبه نرم (norm) یا واریانس خروجی هر head

اگر head ی خروجی‌اش تقریبا همیشه نزدیک صفر باشه، می‌تونه نشونه کم‌اهمیت بودنش باشه.

2. مقایسه تاثیر هر head در یادگیری ویژگی‌های خاص

مثلا می‌تونیم بررسی کنیم که:

آیا یک head در یادگیری وابستگی‌های نحوی (مثل فعل و فاعل) نقش داشته؟

آیا head ی به روابط معنایی خاصی توجه می‌کرده؟

3. استفاده از معیارهای یادگیری مبتنی بر گرادیان(Gradient-based)

در این روش بررسی می‌کنیم که اگر خروجی یک head تغییر کنه، گرادیان خطا چقدر نسبت به اون حساسه. اگر تغییر خروجی اون head تأثیر زیادی روی خطا نداشته باشه، پس می‌شه گفت اهمیت زیادی نداره.

4. تحلیل افزونگی بین سرها(Redundancy Analysis)

بررسی می کنیم که آیا برخی سرهای توجه اطلاعات مشابهی را یاد گرفته‌اند (افزونگی).

محاسبه شباهت مانند Cosine Similarity بین وزن‌های توجه یا خروجی‌های سرهای مختلف.

سرهایی که خروجی‌های مشابهی دارند، ممکن است افزونه باشند و کاندیدای حذف شوند.

برای این کار خروجی‌های هر سر توجه را برای مجموعه‌ای از ورودی‌ها استخراج می کنیم و ماتریس شباهت بین سرها را محاسبه می کنیم. سرهایی که شباهت بالایی با دیگران دارند را شناسایی می کنیم و تأثیر حذف آن‌ها را آزمایش می کنیم.

ج)

اثر حذف سرهای توجه روی وظایف پایین‌دستی

اگر سرهای کم‌اهمیت حذف بشن:

- سرعت پیش‌بینی مدل بهتر می‌شه.
- مصرف حافظه کمتر می‌شه.
- مدل ساده‌تر و سبک‌تر می‌شه.
- در بعضی موارد حتی عملکرد بهتر می‌شه چون حذف اطلاعات زائد ممکنه به مدل کمک کنه تمرکز بهتری داشته باشه

اما اگر اشتباهی سرهای مهم حذف بشن:

- دقت مدل ممکنه پایین بیاد.
- ممکنه مدل نتونه روابط معنایی یا دستوری مهمی رو تشخیص بده.
- کیفیت خروجی در کارهایی مثل ترجمه یا خلاصه‌سازی افت می‌کنه.

چه معیارهایی برای ارزیابی تأثیر حذف سرها داریم؟

برای اینکه بفهمیم حذف سرها چه تأثیری داشته، باید مدل رو قبل و بعد از حذف head ها ارزیابی کنیم. معیارها بسته به نوع وظیفه فرق دارن:

برای طبقه‌بندی (مثل تشخیص احساس یا دسته‌بندی متن):

- دقت(Accuracy)
- F1-score
- Precision / Recall

برای ترجمه یا خلاصه‌سازی:

1. BLEU score برای ترجمه ماشینی
2. ROUGE score برای خلاصه‌سازی
3. یا حتی ارزیابی انسانی اگه بخوایم دقیق‌تر بررسی کنیم

برای همه وظایف:

- سرعت inference مثلا چند میلی‌ثانیه طول می‌کشه
- استفاده از حافظه / تعداد پارامترها

د)

بله، می‌توان از RL استفاده کرد و این ایده‌ای کاربردی در شرایطی است که بخوایم مدل به شکل هوشمند تصمیم بگیره کدومhead ها رو در زمان اجرا فعال یا غیرفعال کنه، به‌جای اینکه از قبل تعدادی head رو به‌طور ثابت حذف کنیم.

چرا یادگیری تقویتی مناسب این کاره؟

در یادگیری تقویتی، یک عامل (agent) یاد می‌گیره که با انجام عمل (action) مناسب در یک وضعیت (state) خاص، پاداش (reward) بهتری دریافت کنه.

حالا در زمینه attention head ها:

وضعیت(state) : ویژگی‌های ورودی مدل یا وضعیت فعلی attention layer

عمل(action) : فعال یا غیرفعال کردن یک یا چند head

پاداش(reward) : ترکیبی از دقت مدل و صرفه‌جویی در منابع (مثل زمان یا حافظه)

مزیت انتخاب دینامیک head با یادگیری تقویتی

انعطاف‌پذیره: بسته به نوع جمله یا ورودی، head های مختلفی فعال می‌شن.

مصرف منابع بهینه‌تر می‌شه.

Head های بی‌اثر تو شرایط خاص به طور خودکار حذف می‌شن.

مدل می‌تونه بین دقت بالا و سرعت بیشتر تعادل برقرار کنه.

چالش‌ها و پیچیدگی‌ها

تعریف درست پاداش خیلی مهمه؛ باید دقت مدل، سرعت، و مصرف حافظه رو هم‌زمان در نظر گرفت.

آموزش یادگیری تقویتی معمولا زمان‌برتر و پیچیده‌تره.

پیاده‌سازی نیاز به طراحی دقیق سیاست تصمیم‌گیری (policy) داره.

سوال 3

الف)

جایگزین کردن توجه ضرب نقطه‌ای مقیاس‌شده (Scaled Dot-Product Attention) با توجه جمعی (Additive Attention) در مدل ترنسفورمر، از نظر تئوری امکان‌پذیر است، اما معمولا ایده‌ی مناسبی تلقی نمی‌شود.

1. کارایی محاسباتی

توجه جمعی نیاز به اعمال چند لایه‌ی خطی (مثل Wq، Wk) و یک تابع غیرخطی مانند tanh دارد که برای هر جفت کوئری و کی، به صورت جداگانه محاسبه می‌شود. این باعث افزایش هزینه‌ی محاسباتی و کاهش قابلیت موازی‌سازی (parallelization) در اجرا می‌شود. در مقابل، توجه ضرب نقطه‌ای بسیار ساده‌تر است و با ضرب ماتریسی قابل اجراست که روی سخت‌افزارهای مدرن (GPU/TPU) بسیار سریع‌تر و بهینه‌تر عمل می‌کند.

2. مقیاس‌پذیری

مدل ترنسفورمر برای پردازش حجم زیادی از داده‌ها (مانند توالی‌های بلند یا داده‌های زبانی عظیم) طراحی شده است. در چنین مقیاسی، توجه ضرب نقطه‌ای به دلیل ساختار ساده‌ترش، مقیاس‌پذیرتر از توجه جمعی است. استفاده از توجه جمعی می‌تواند باعث کاهش سرعت آموزش و افزایش مصرف حافظه شود.

3. عملکرد مدل

اگرچه توجه جمعی در مدل‌های کوچکتر مانند مدل‌های RNN قدیمی عملکرد خوبی داشته، در مدل‌های بزرگ مانند ترنسفورمرها، تجربه نشان داده که توجه ضرب نقطه‌ای به خوبی عمل می‌کند و در عمل نتایج بهتری نیز ارائه می‌دهد. بنابراین، جایگزینی آن با مکانیزمی که کارایی پایین‌تری دارد و به‌وضوح برتری تجربی‌ای ارائه نمی‌دهد، چندان منطقی نیست.

ب)

بله می توان ترکیب کرد و حتی در برخی تحقیقات و مدل‌های تجربی این ایده بررسی شده است. اما اینکه آیا این ترکیب مفید و کارآمد است، به هدف مدل و زمینه‌ی کاربرد آن بستگی دارد.

از نظر مفهومی چرا می‌توان ترکیب کرد؟

این دو نوع توجه، اساسا دو روش متفاوت برای محاسبه امتیاز شباهت بین Query و Key هستند:

توجه جمعی با استفاده از شبکه عصبی (غیرخطی) سعی می‌کند رابطه‌ی پیچیده‌تری بین Query و Key بسازد.

توجه ضرب نقطه‌ای با یک عملیات ساده و موثر، شباهت برداری را ارزیابی می‌کند.

ترکیب این دو می‌تواند کمک کند تا مدل هم ویژگی‌های خطی ساده را از طریق dot-product attention یاد بگیرد و هم روابط غیرخطی پیچیده‌تر را از طریق additive attention درک کند.

چطور می‌توان ترکیب کرد؟

روش‌هایی برای ترکیب وجود دارد:

میانگین‌گیری یا وزن‌دهی بین دو نوع امتیاز attention مثلا امتیاز dot-product و امتیازadditive را با وزنی ترکیب کنیم.

استفاده‌ی موازی از هر دو نوع attention و ادغام خروجی آن‌ها.

طراحی یک مکانیزم یادگیری برای انتخاب یا وزن‌دهی پویا بین این دو بر اساس ورودی.

مزایا و معایب احتمالی:

جنبه	مزایا	معایب
قدرت بیانی	ترکیب می‌تواند نمایشی غنی‌تر بسازد	پیچیدگی بیشتر و افزایش پارامتر
عملکرد	ممکن است در برخی وظایف خاص بهبود دهد	نیازمند ارزیابی دقیق و آزمایش تجربی
کارایی محاسباتی	می‌تواند کمک کند درک مدل عمیق‌تر شود	کندتر و سنگین‌تر نسبت به حالت ساده

ج)

برای محاسبه تعداد پارامترهای یک لایه‌ی Multi-Head Attention با مشخصات داده شده، مراحل زیر را دنبال می‌کنیم:

مشخصات مسئله:

تعداد سرها (heads): 3

ابعاد ورودی:

- query_dim = 10
- key_dim = 20
- value_dim = 30

ابعاد هر سر (head): 100 (head_dim)

بعد خروجی نهایی: 50 (output_dim)

طول دنباله‌ی ورودی: 64

بایاس (bias): فعال

محاسبه پارامترهای پروژکشن‌های Query، Key و Value :

برای هر یک از query، key و value یک لایه‌ی خطی (Dense) وجود دارد که آن‌ها را به فضای num_heads * head_dim نگاشت می‌دهد.

پروژکشن Query :

وزن:

$$query\_dim \times (num\_heads * head\_dim) = 10 \times (3 \times 100) = 10 \times 300$$

بایاس:

$$num\_heads * head\_dim = 300$$

تعداد پارامترها:

$$10 \times 300 + 300 = 3300$$

پروژکشن Key :

وزن:

$$key\_dim \times (num\_heads * head\_dim) = 20 \times 300$$

بایاس:

300

تعداد پارامترها:

$20 \times 300 + 300 = 6300$

پروژکشن Value :

وزن:

$value\_dim \times (num\_heads * head\_dim) = 30 \times 300$

بایاس

300

تعداد پارامترها

$30 \times 300 + 300 = 9300$

محاسبه پارامترهای پروژکشن خروجی:

پس از محاسبه‌ی توجه چندسر، یک لایه‌ی خطی دیگر برای ترکیب خروجی سرها و تبدیل آن به output_dim استفاده می‌شود.

پروژکشن خروجی:

وزن:

$(num\_heads * head\_dim) \times output\_dim = 300 \times 50$

بایاس:

50

تعداد پارامترها:

$300 \times 50 + 50 = 15050$

محاسبه کل پارامترها:

کل پارامترهای لایه‌ی توجه چندسر، مجموع پارامترهای تمام پروژکشن‌ها است:

کل پارامترها = پارامترهای Query + پارامترهای Key + پارامترهای Value + پارامترهای پروژکشن خروجی

3300 + 6300 + 9300 + 15050 = 33950

سوال 4

الف)

استفاده از مدل‌های ترنسفورمر در سری‌های زمانی وقتی مناسب‌تر از LSTM است که با شرایط زیر روبه‌رو باشیم:

وجود وابستگی‌های بلندمدت در داده‌ها:

ترنسفورمرها برخلاف LSTM محدود به وابستگی‌های زمانی نزدیک نیستند و می‌توانند روابط میان نقاط بسیار دور در سری زمانی را بهتر درک کنند. اگر الگوهای مهمی در فاصله‌های زمانی طولانی پنهان شده باشند، ترنسفورمر عملکرد بهتری خواهد داشت.

داده‌های حجیم و پیچیده:

زمانی که حجم داده زیاد باشد و سری زمانی شامل چندین ویژگی (multi-variate) یا ساختارهای پیچیده باشد، ترنسفورمرها با معماری موازی خود سریع‌تر و موثرتر آموزش می‌بینند.

نیاز به موازی‌سازی پردازش:

ترنسفورمرها برخلاف LSTM، وابسته به ترتیب زمانی در فرایند آموزش نیستند و می‌توانند داده‌ها را به صورت موازی پردازش کنند، که این ویژگی در پردازش‌های مقیاس‌پذیر یا بلادرنگ اهمیت دارد.

ب)

برای استفاده از مدل‌های ترنسفورمر در مسائل سری زمانی، داده‌ها باید با دقت و به شکلی خاص پیش‌پردازش شوند تا مدل بتواند ساختار زمانی را درک کند. مراحل اصلی پیش‌پردازش معمولا به صورت زیر است:

1. نرمال‌سازی یا استانداردسازی داده‌ها

ترنسفورمرها نسبت به مقیاس داده حساس هستند. بنابراین، باید داده‌ها را نرمال‌سازی (مانند Min-Max Scaling) یا استانداردسازی (مانند Z-score) کرد تا مقادیر ویژگی‌ها در یک بازه مشابه قرار بگیرند.

2. ساخت پنجره‌های زمانی (sliding windows)

از آنجا که ترنسفورمر ساختار توالی ندارد، باید سری زمانی را به قطعاتی با طول ثابت تقسیم کنیم. مثلاً اگر طول ورودی مدل 24 باشد، هر ورودی شامل 24 گام زمانی از گذشته است. این کار به مدل کمک می‌کند تا از داده‌های گذشته برای پیش‌بینی گام بعدی استفاده کند.

3. افزودن ویژگی موقعیت‌یابی(Positional Encoding)

ترنسفورمرها به ترتیب زمانی حساس نیستند؛ پس باید اطلاعات مربوط به ترتیب داده‌ها را به‌صورت مستقیم اضافه کرد. این کار معمولا با استفاده از کدگذاری موقعیتی (positional encoding) انجام می‌شود که به هر گام زمانی یک بردار ویژه مرتبط با موقعیت آن افزوده می‌شود.

4. در صورت وجود چند ویژگی(multi-variate) ترکیب ویژگی‌ها به صورت بردار

اگر سری زمانی چند بعدی باشد (مثلاً دما، فشار، رطوبت)، در هر گام زمانی باید این ویژگی‌ها را به‌صورت یک بردار به مدل داد.

5. تقسیم داده به بخش‌های آموزش، اعتبارسنجی و آزمون

داده‌ها باید به‌گونه‌ای تقسیم شوند که ترتیب زمانی حفظ شود؛ یعنی به‌صورت ترتیبی (نه تصادفی) به سه بخش آموزش، اعتبارسنجی و آزمون تقسیم شوند.

6. ساخت برچسب (label) برای پیش‌بینی

اگر هدف پیش‌بینی گام بعدی است، باید برای هر پنجره ورودی، مقدار واقعی گام بعدی را به‌عنوان برچسب (target) مشخص کرد.

در نهایت، داده‌ها باید به شکل مناسبی تبدیل شوند تا قابل استفاده در مدل ترنسفورمر باشند:
معمولاً شکل نهایی داده‌ها این‌گونه است:

(تعداد نمونه‌ها، طول توالی، تعداد ویژگی‌ها)

ج)

تفاوت بین ترنسفورمر استاندارد و ترنسفورمرهای مخصوص سری زمانی مثل Time Series Transformer، Informer، Autoformer و غیره در این است که نسخه‌های مخصوص سری زمانی برای غلبه بر چالش‌های خاص داده‌های زمانی طراحی شده‌اند. در ادامه، به‌صورت خلاصه و مفهومی به تفاوت‌های اصلی اشاره می‌کنم:

1. نوع Attention

ترنسفورمر استاندارد:
از full self-attention استفاده می‌کند؛ یعنی همه گام‌های زمانی به هم متصل‌اند. این روش وقتی طول توالی زیاد شود، بسیار پرمصرف از نظر حافظه و زمان پردازش است.

ترنسفورمرهای مخصوص سری زمانی مانند Informer :
از sparse attention یا probabilistic attention استفاده می‌کنند که فقط روی مهم‌ترین زمان‌ها تمرکز دارد، نه همه. این باعث افزایش سرعت و مقیاس‌پذیری می‌شود.

2. مدیریت وابستگی بلندمدت

ترنسفورمر استاندارد:
از لحاظ تئوری توانایی درک وابستگی‌های بلندمدت دارد، ولی در عمل در داده‌های زمان‌محور ممکن است دقت کافی نداشته باشد.

مدل‌هایی مثل Autoformer یا Informer :
ساختار attention را طوری تنظیم می‌کنند که روابط دوره‌ای و بلندمدت را بهتر یاد بگیرند مثلا با decomposition یا hierarchical learning

3. توجه به ساختار زمانی و فصلی

مدل‌های سری زمانی خاص مانند Autoformer :
اغلب داده را به بخش‌های trend (روند) و seasonality (فصل‌پذیری) تجزیه می‌کنند تا پیش‌بینی دقیق‌تری انجام دهند. این ویژگی در ترنسفورمر استاندارد وجود ندارد.

4. مدیریت خروجی‌ها

ترنسفورمر استاندارد:
معمولا برای پیش‌بینی یک‌به‌یک (step-by-step) طراحی شده است.

مدل‌هایی مثل Informer :
خروجی‌ها را به‌صورت چندگامی (multi-step forecasting) و در یک‌بار اجرا تولید می‌کنند که برای سری‌های زمانی بسیار مفید است.

5. طراحی سبک‌تر و سریع‌تر

ترنسفورمرهای ویژه سری زمانی:
اغلب تعداد پارامترها، زمان آموزش و مصرف حافظه را بهینه می‌کنند تا بتوانند روی داده‌های حجیم و طولانی اجرا شوند برای مثال، Informer تا توالی‌هایی با طول هزاران گام نیز به‌خوبی کار می‌کند.

د)

برای استفاده از ترنسفورمر در پیش‌بینی چند مرحله‌ای (multi-step forecasting) در سری‌های زمانی، باید هم داده‌ها را مناسب آماده کرد و هم ساختار مدل را طوری طراحی نمود که بتواند چند گام آینده را پیش‌بینی کند. در ادامه به‌صورت گام‌به‌گام توضیح می‌دهم که چگونه می‌توان این کار را انجام داد:

1. ساخت پنجره‌های زمانی مناسب

برای پیش‌بینی چند مرحله‌ای، باید هر ورودی شامل یک پنجره زمانی از گذشته و هر خروجی شامل چند مقدار آینده باشد. مثلا:

ورودی 24: گام زمانی گذشته

خروجی 12: گام زمانی آینده

این یعنی به‌جای داشتن یک خروجی (مثل پیش‌بینی فقط گام بعدی)، حالا خروجی مدل یک بردار از چند مقدار خواهد بود.

2. تنظیم ساختار مدل ترنسفورمر

دو روش رایج برای پیش‌بینی چندمرحله‌ای با ترنسفورمر:

روش اول Direct Multi-Step Output :

در این روش، مدل یک‌بار آموزش می‌بیند و مستقیما همه گام‌های آینده را در خروجی تولید می‌کند.

سریع و ساده برای inference

سخت‌تر برای یادگیری روابط بین گام‌های آینده

شکل ورودی و خروجی:

Input: (batch_size, input_length, num_features)

Output: (batch_size, forecast_length, num_features)

روش دوم Auto-Regressive Forecasting :

مدل فقط یک گام پیش‌بینی می‌کند و بعد از هر پیش‌بینی، خروجی را به ورودی اضافه می‌کند و مجددا پیش‌بینی می‌کند تا به تعداد مراحل دلخواه برسیم.

دقت بالا برای گام‌های اولیه

کند و مستعد انباشته شدن خطا در گام‌های بعدی

3. تنظیم Decoder در ترنسفورمر (در صورت استفاده)

در برخی معماری‌ها (مثل ترنسفورمر کامل)، از Encoder-Decoder استفاده می‌شود:

Encoder داده‌های گذشته را پردازش می‌کند.

Decoder دنباله‌ای از ورودی‌های قبلی (یا مکان‌های زمانی آینده) را گرفته و گام‌های آینده را پیش‌بینی می‌کند.

برای پیش‌بینی چندمرحله‌ای باید decoder ورودی‌هایی با طول forecast_length داشته باشد (مثلا فقط اطلاعات زمان یا مقادیر قبلی).

4. استفاده از Positional Encoding برای آینده

از آنجا که گام‌های آینده هنوز داده‌ای ندارند، در decoder می‌توان به‌جای مقدار، فقط اطلاعات موقعیت زمانی (positional encoding) آینده را استفاده کرد (یعنی مدل فقط بداند در حال پیش‌بینی چه زمانی است).

5. محاسبه خطا(Loss Function)

باید مدل را طوری آموزش داد که کل خروجی پیش‌بینی‌شده (مثلا 12 گام آینده) با خروجی واقعی مقایسه شود.

مثلا از MSE (mean squared error) یا MAE روی کل دنباله پیش‌بینی استفاده می‌شود.

۵)

مدل iTransformer مخفف (inverted Transformer) یک مدل جدید و مؤثر برای پیش‌بینی سری‌های زمانی (Time Series Forecasting) است که در مقاله‌ی «Inverted Transformers Are Effective for Time Series Forecasting» معرفی شده. این مدل با یک تغییر کلیدی در دیدگاه سنتی ترنسفورمر، توانسته هم عملکرد دقیق‌تری داشته باشد و هم سبک‌تر و سریع‌تر اجرا شود.

ایده‌ی اصلی iTransformer چیست؟

در ترنسفورمرهای معمول مثل Informer یا Autoformer داده به شکل زیر پردازش می‌شود:

هر توکن = اطلاعات تمام ویژگی‌ها در یک گام زمانی

مدل روی توالی زمانی تمرکز دارد

یعنی: توجه (attention) بین لحظات زمانی مختلف اعمال می‌شود

اما در iTransformer :

هر توکن = یک ویژگی خاص در طول زمان
یعنی: توجه (attention) بین ویژگی‌های مختلف اعمال می‌شود، نه بین زمان‌ها.

این تغییر زاویه دید، باعث بهینه‌سازی در یادگیری روابط پیچیده بین ویژگی‌ها می‌شود.

مراحل عملکرد iTransformer :

1. ورودی سری زمانی چندمتغیره(Multivariate Time Series)

فرض کن سری زمانی‌ای داریم با شکل:

Input: (batch_size, time_steps, num_features)

مثلا:

(8 ,96 ,32)

مدل داده را Transpose می‌کند:

(32, 8, 96)

یعنی: هر ویژگی به‌صورت یک دنباله زمانی مستقل دیده می‌شود.

2. اعمال Attention روی ویژگی‌ها

بر خلاف ترنسفورمرهای سنتی، در اینجا Self-Attention روی محور ویژگی‌ها (features) انجام می‌شود.

یعنی مدل یاد می‌گیرد که چه ویژگی‌هایی روی یکدیگر تاثیرگذارند، نه این‌که چه لحظاتی در زمان وابسته‌اند.

3. استفاده از Encoder-only Architecture

iTransformer از Encoder استفاده می‌کند و Decoder ندارد

ورودی شامل مقادیر گذشته است

خروجی، مقادیر آینده را مستقیم و در یک مرحله پیش‌بینی می‌کند (Multi-step forecasting)

4. استفاده از LayerNorm و FFN به تفکیک ویژگی‌ها

هر ویژگی به‌صورت جداگانه از شبکه عبور می‌کند.

یعنی مدل می‌تواند نمایش غنی و دقیق‌تری از هر متغیر زمانی ایجاد کند

مدل در نهایت یک توالی از مقادیر پیش‌بینی‌شده برای گام‌های آینده تولید می‌کند به شکل زیر:

Output: (batch_size, forecast_length, num_features)

مثلا :

(32, 24, 8)

برای رفع برخی ایرادات و ابهامات از AI استفاده شده است.