به نام خداوند جان و خرد

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

سامانه‌های یادگیری‌ماشین توزیع شده

تمرین نوشتاری ۲

نام و نام خانوادگی: علی خرم فر

شماره دانشجویی: ۸۱۰۱۰۲۱۲۹

دی‌ماه ۱۴۰۳

فهرست مطالب

فهرست جداول

فهرست اشکال

۱_پاسخ سوال شماره ۱

در این روش‌ها، داده‌ها را به بخش‌های کوچک‌تر تقسیم می‌کنیم و آنها را بین چندین دیسک یا سرور توزیع می‌کنیم. با این کار امکان انجام عملیات خواندن و نوشتن به صورت همزمان روی چند دیسک فراهم می‌شود. به زبان ساده‌تر، اگر مثلا داده‌هایمان به چهار پارتیشن تقسیم شده باشند، می‌توانیم چهار برابر سریع‌تر از یک دیسک تکی، داده‌ها را پردازش کنیم، البته به شرطی که پهنای باند شبکه اجازه دهد.

علاوه بر این، ذخیره‌سازی توزیع‌شده معمولا از تکرار داده‌ها یا Replication استفاده می‌کند که مشابه این کار را در GFS در درس خواندیم. یعنی هر داده روی چند دیسک مختلف کپی می‌شود. این کار باعث می‌شود که اگر یکی از دیسک‌ها کند باشد یا خراب شود، به داده از روی دیسک‌های دیگر دسترسی داشته باشیم. همچنین، بعضی سیستم‌ها از تکنیک‌هایی مثل کدگذاری Erasure Coding استفاده می‌کنند که علاوه بر افزایش اطمینان، به کاهش حجم ذخیره‌سازی هم کمک می‌کند [1].

یکی دیگر از مزیت‌های دیگر این سیستم‌ها این است که می‌توانیم داده‌ها را بر اساس اهمیت یا الگوی استفاده، در مکان مناسب‌تری ذخیره کنیم. برای مثال، داده‌هایی که بیشتر استفاده می‌شوند روی دیسک‌های سریع‌تر قرار می‌گیرند و داده‌های کمتر استفاده‌شده به دیسک‌های کندتر منتقل می‌شوند [2].

در مجموع، با توزیع داده‌ها بین چند دیسک و استفاده از تکنیک‌هایی که گفتیم، سرعت خواندن و نوشتن در سیستم‌های ذخیره‌سازی توزیع‌شده به طور چشمگیری افزایش پیدا می‌کند و مشکل کندی دیسک‌ها برطرف می‌شود.

۲_ پاسخ سوال شماره ۲

۲-۱_ احتمال خرابی بدون ماشین پشتیبان

فرض کنیم Pi احتمال خرابی ماشین i در یک سیستم با n ماشین باشد. احتمال اینکه هیچ کدام از ماشین‌ها خراب نشوند برابر است با:

$$\prod_{i=1}^{n}(1-p_i)$$

از روش مکمل استفاده میکنیم پس احتمال خرابی حداقل یک ماشین برابر است با:

$$1-\prod_{i=1}^{n}(1-p_i)$$

۲-۲_ احتمال خرابی به همراه ماشین پشتیبان

در حالتی که یک ماشین پشتیبان داریم و احتمال خرابی آن صفر است، احتمال خرابی سیستم تنها زمانی رخ می‌دهد که مکمل حالتی باشد که هیچ کدام از ماشین‌ها خراب نشوند و اگر یکی خراب شد، بقیه سالم باشند(یک ماشین پشتیبان داریم)

این حالت با فرمول زیر توصیف می‌شود:

$$1-\left(\prod_{i=1}^{n}(1-p_i)+\sum_{i=1}^{n}p_i\prod_{j=1,j\neq i}^{n}(1-p_j)\right)$$

عبارت اول:

$$\prod_{i=1}^{n}(1-p_i)$$

احتمال اینکه هیچ یک از ماشین‌ها خراب نشوند است.

عبارت دوم:

$$\sum_{i=1}^{n}p_i\prod_{j=1,j\neq i}^{n}(1-p_j)$$

احتمال خرابی یک ماشین و سالم ماندن بقیه ماشین‌ها است، که در این حالت، ماشین پشتیبان وارد عمل می‌شود و سیستم دچار خرابی نمی‌شود.

۲-۳_ میزان کاهش احتمال خرابی

برای یافتن کاهش در احتمال خرابی، کافی است تفاوت احتمال خرابی بدون ماشین پشتیبان و با ماشین پشتیبان را پیدا کنیم.

$$\left[1-\prod_{i=1}^{n}(1-p_i)\right]$$

$$-\left[1-\left(\prod_{i=1}^{n}(1-p_i)+\sum_{i=1}^{n}p_i\prod_{j=1,j\neq i}^{n}(1-p_j)\right)\right]$$

$$= \prod_{i=1}^{n}(1-p_i) + \sum_{i=1}^{n} p_i \prod_{j=1,j\neq i}^{n}\left(1-p_j\right) - \prod_{i=1}^{n}(1-p_i)$$

$$= \sum_{i=1}^{n} p_i \prod_{j=1,j\neq i}^{n}\left(1-p_j\right)$$

برای ساده‌سازی این عبارت، می‌توان نوشت:

$$\sum_{i=1}^{n} p_i \prod_{j=1,j\neq i}^{n}(1-p_j) = \sum_{i=1}^{n} p_i \frac{\prod_{j=1}^{n}(1-p_j)}{1-p_i}$$

$$= \prod_{j=1}^{n}\left(1-p_j\right)\sum_{i=1}^{n}\frac{p_i}{1-p_i}$$

۳_ پاسخ سوال شماره ۳

پردازنده‌های مختلف از روش‌های متفاوتی برای ذخیره‌سازی داده‌ها استفاده می‌کنند. به طور خاص:

پردازنده‌های Intel x86 از روش little-endian استفاده می‌کنند که در آن بیت‌های کم‌ارزش‌تر زودتر ذخیره می‌شوند.

پردازنده‌های ARM از هر دو روش big-endian و little-endian پشتیبانی می‌کنند. البته نسخه‌های قدیمی‌تر پردازنده‌های ARM تنها از روش big-endian پشتیبانی می‌کردند.

پروتکل‌های شبکه مانند TCP/IP از روش big-endian استفاده می‌کند که در آن بیت‌های پرارزش‌تر زودتر ارسال می‌شوند.

۳-۱_ نمایش عدد به صورت Float32

عدد سوال به صورت Float32 است. باید آن را در در قالب IEEE نمایش دهیم:

بیت علامت (۱ بیت): چون عدد مثبت است، بیت علامت برابر با ۰ است.

بخش نمایی (۸ بیت)

مانتیس (۲۳ بیت): بخش کسری عدد

پس عدد (6.342×10^{-5}) راا به مبنای دودویی تبدیل می‌کنیم:

$$6.342 \times 10^{-5} = 1.00000111001001001010011_2 \times 2^{-14}$$

بخش نمایی را با استفاده از مقدار بایاس‌شده (۱۲۷) محاسبه می‌کنیم

$$127 - 14 = 113 \rightarrow \quad 1110001_2$$

مانتیس را با حذف بیت ۱ پرارزش در نظر می‌گیریم:

00001010000000001011010

فرمت نهایی : 0|01110001|00001010000000001011010

= 00111000100001010000000001011010

۳-۲_ ذخیره در قالب Little-Endian برای Intel x86

همانطور که گفتیم در Intel x86، عدد به صورت Little-Endian ذخیره می‌شود، یعنی بایت‌های کم‌ارزش‌تر ابتدا ذخیره می‌شوند. ابتدا قالب IEEEرا به بایت‌ها تقسیم می‌کنیم:

00111000 10000101 00000000 01011010

بایت ۱: 0x5A

بایت ۲: 0x00

بایت ۳: 0x85

بایت ۴: 0x38

[0x5A][0x00][0x85][0x38]

۳-۳_ تبدیل به Big-Endian برای TCP/IP

پروتکل TCP/IP از قالب Big-Endian استفاده می‌کند پس بایت‌های پرارزش‌تر ابتدا ارسال می‌کنیم.

[0x38][0x85][0x00][0x5A]

۳-۴_ بازسازی Deserialization

دریافت داده‌ها در مقصد – AMD x86

داده‌ها به فرم Big-Endian دریافت می‌شوند:

[0x38][0x85][0x00][0x5A]

در پردازنده AMD x86، باید داده‌ها به قالب Little-Endian تبدیل شوند تا قابل پردازش باشند:

[0x5A][0x00][0x85][0x38]

در نهایت بایت‌های دریافت‌شده را دوباره به قالب IEEE تبدیل می‌کنیم:

بیت علامت: ۰

بخش نمایی: ۰۱۱۱۰۰۰۱

مانتیس: ۰۰۰۰۱۰۱۰۰۰۰۰۰۰۰۱۰۱۱۰۱۰

$$6.342 \times 10^{-5}$$

۴_ پاسخ سوال شماره ۴

۴-۱_ توضیحات کد

ساخت دیتافریم

```
data = [("Alice", 25), ("Bob", 30), ("Cathy", 27)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)
```

ابتدا لیستی از داده‌ها تحت عنوان data تعریف کرده‌ایم که شامل نام و سن سه فرد است. ستون‌های دیتافریم نیز از طریق یک لیست دیگر به نام columns مشخص می‌شوند. با استفاده از متد createDataFrame از آبجکت spark، این داده‌ها به یک دیتافریم تبدیل می‌شوند. پس در این مرحله دیتافریمی با دو ستون Name و Age ساخته می‌شود.

فیلتر کردن داده‌ها

```
filtered_df = df.filter(df.Age > 26)
```

در اینجا ما از متد filter استفاده کرده‌ایم تا رکوردهایی که مقدار ستون Age آن‌ها کمتر یا مساوی ۲۶ است را حذف کنیم. بدیهی است پس از اعمال فیلتر، تنها رکوردهایی که سن بیشتر از ۲۶ دارند، در دیتافریم باقی می‌مانند.

اضافه کردن ستون جدید

```
transformed_df = filtered_df.withColumn("AgeIn5Years", filtered_df.Age + 5)
```

در این مرحله، با استفاده از متد withColumn، یک ستون جدید به نام AgeIn5Years به دیتافریم فیلترشده اضافه می‌کنیم. مقدار این ستون برابر است با مقدار ستون Age به‌علاوه ۵.

تغییر نام ستون

```
renamed_df = transformed_df.withColumnRenamed("AgeIn5Years", "AgeAfterFiveYears")
```

در نهایت، ستون AgeIn5Years را به AgeAfterFiveYears تغییر نام می‌دهیم تا نام آن بهتر باشد. از متد withColumnRenamed برای این کار استفاده شده.

ساخت دیتافریم دوم

```
data_2nd = [("James", 28), ("Josh", 32), ("Sarah", 21)]
columns_2nd = ["Name", "Age"]
df_2nd = spark.createDataFrame(data_2nd, columns_2nd)
```

در اینجا همان فرآیند ایجاد دیتافریم را برای مجموعه داده دیگری تکرار کرده‌ایم. این بار لیست داده‌ها شامل سه فرد با نام و سن متفاوت است. این داده‌ها به دیتافریمی با همان ساختار قبلی تبدیل می‌شوند.

فیلتر کردن داده‌ها

```
filtered_df_2nd = df_2nd.filter(df_2nd.Age > 26)
```

در این مرحله نیز مانند دیتافریم اول، رکوردهایی که مقدار ستون Age آن‌ها کمتر یا مساوی ۲۶ است، حذف می‌شوند.

اضافه کردن ستون جدید

```
transformed_df_2nd = filtered_df_2nd.withColumn("AgeIn15Years", filtered_df_2nd.Age + 15)
```

در اینجا نیز با استفاده از متد withColumn یک ستون جدید اضافه کرده‌ایم. مقدار این ستون برابر با سن فعلی افراد + ۱۵ است.

تغییر نام ستون

```
renamed_df_2nd = transformed_df_2nd.withColumnRenamed("AgeIn15Years", "AgeAfterFiftheenYears")
```

نام ستون جدید اضافه‌شده را به AgeAfterFiftheenYears تغییر می‌دهیم.

شمارش تعداد ردیف‌ها

```
row_count_2nd = renamed_df_2nd.count()
```

در این مرحله، تعداد رکوردهای دیتافریم نهایی محاسبه شد که این کار با استفاده از متد count انجام می‌شود.

جمع‌آوری داده‌ها

```
result_2nd = renamed_df_2nd.collect()
result = renamed_df.collect()
```

در نهایت، داده‌های موجود در دو دیتافریم نهایی (یکی مربوط به دیتافریم اول و دیگری مربوط به دیتافریم دوم) با استفاده از متد collect جمع‌آوری میک‌نیم. یک لیست از رکوردها در نهایت خواهیم داشت.

۲-۴_ ترتیب اجرا

باتوجه به مطالب درس Transformations مانند map، filter، flatMap و غیره که یک RDD جدید از یک RDD موجود می‌سازند این عملیات واقعی اجرا نمی‌شوند تا زمانی که یک Action درخواست شود یعنی Lazily هستند. چون Transformations تا زمان نیاز اجرا نمی‌شوند، ترتیب واقعی اجرای آن‌ها زمانی مشخص می‌شود که یک Action اجرا شود. Actionها مانند ()count، ()collect، reduce() عملیات‌هایی هستند که نتیجه‌ای نهایی را برای ما تولید می‌کنند.

```
data = [("Alice", 25), ("Bob", 30), ("Cathy", 27)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)
```

در کد بالا یک دیتافریم از داده‌های اولیه ساخته می‌شود. این مرحله عملیاتی مستقیم است و بلافاصله اجرا می‌شود نه Lazy. البته این مورد را مطمئن نیستم. اگر این هم Lazily باشد پس این هم اجرا نمی‌شود.

```
filtered_df = df.filter(df.Age > 26)
transformed_df = filtered_df.withColumn("AgeIn5Years", filtered_df.Age + 5)
renamed_df = transformed_df.withColumnRenamed("AgeIn5Years", "AgeAfterFiveYears")
```

سه Transformation تعریف شده هیچ‌کدام از این عملیات‌ها اجرا نمی‌شوند زیرا Actionای فراخوانی نشده است.

```
data_2nd = [("James", 28), ("Josh", 32), ("Sarah", 21)]
columns_2nd = ["Name", "Age"]
df_2nd = spark.createDataFrame(data_2nd, columns_2nd)
```

یک دیتافریم دیگر به نام df_2nd با داده‌های جدید ساخته می‌شود. این عملیات نیز بلافاصله اجرا می‌شود و Lazy نیست. البته من جستجو کردم نوشته شده بود که عملیات های مربوط به Load کردن lazily هستند. پس احتمال دارد این مورد هم lazily باشد. اگر باشد یعنی هیچی اجرا نشده است.

```
filtered_df_2nd = df_2nd.filter(df_2nd.Age > 26)
transformed_df_2nd = filtered_df_2nd.withColumn("AgeIn15Years", filtered_df_2nd.Age + 15)
renamed_df_2nd = transformed_df_2nd.withColumnRenamed("AgeIn15Years", "AgeAfterFiftheenYears")
```

این Transformation ها نیز اجرا نمی‌شوند زیرا هنوز Actionای فراخوانی نشده است.

ولی در خط ۱۴ داریم:

```
row_count_2nd = renamed_df_2nd.count()
```

فراخوانی متد count یک Action است. این Action باعث می‌شود تمام Transformations تعریف‌شده برای renamed_df_2nd به ترتیب وابستگی اجرا شوند و اصلا آن مواردی که مربوط به دیتافریم اول بودند اجرا نشوند. اینکه اصلا دیتافریم اول لود می شود یانه را مطمئن نیستم ولی ممکن است آن هم اجرا نشود. پس در اینجا:

ابتدا filter روی df_2nd اجرا می‌شود.سپس withColumn برای اضافه کردن ستون AgeIn15Years اجرا شده و در آخر withColumnRenamed برای تغییر نام ستون اجرا می‌شود. در نهایت تعداد ردیف‌های دیتافریم نهایی محاسبه شده و مقدار آن در متغیر row_count_2nd ذخیره می‌شود.

در خط ۱۶ هم کد زیر را داریم:

```
result_2nd = renamed_df_2nd.collect()
```

متد collect یک Action دیگر است. این Action نیز باعث اجرای تمام Transformations تعریف‌شده روی renamed_df_2nd می‌شود، اگرچه قبلا یک بار با count اجرا کردیم و خب RDD آن را در حافظه داریم.

خط ۱۷:

```
result = renamed_df.collect()
```

در نهایت در خط آخر فراخوانی متد collect برای دیتافریم renamed_df یک Action دیگر که باعث اجرای تمام Transformations تعریف‌شده روی renamed_df به ترتیب زیر می‌شود.

ابتدا filter روی df اجرا می‌شود. سپس withColumn برای اضافه کردن ستون AgeIn5Years اجرا می‌شود. در نهایت، withColumnRenamed برای تغییر نام ستون اجرا می‌شود. همانطور که مشاهده می‌کنیم اگرچه در کدمان این موارد را زودتر نوشته بودیم ولی در عمل دیرتر اجرا می‌شوند و در آخر است که داده‌های نهایی دیتافریم جمع‌آوری شده و در متغیر result ذخیره می‌شوند.

۵_ پاسخ سوال شماره ۵

برای پاسخ به این سوال از https://www.intechopen.com/chapters/68840 استفاده شد.

هدف در Serialization تبدیل یک Object به یک فرمت قابل انتقال یا ذخیره است. برای مثال، زمانی که می‌خواهیم اطلاعات یک شیء در Python را برای استفاده در یک سیستم دیگر مثلا Java ارسال کنیم، باید آن را به فرمت قابل فهمی برای هر دو سیستم تبدیل کنیم. در اسلاید درس برخی از چالش‌های آن را بررسی کردیم از جمله Object Representation و Object References که در ادامه نگاه دقیق‌تری به آن‌ها خواهیم داشت.

۵-۱_ Object Representation
چالش

نمایش یک شی در زبان‌های برنامه‌نویسی یا پلتفرم‌های مختلف می‌تواند متفاوت باشد. این تفاوت ممکن است شامل اندازه متغیرها، ترتیب بایت‌ها یا Endianness (که در سوال قبلی بررسی شد)، یا نحوه ذخیره داده‌ها باشد. به زبان ساده اگر یک داده را از یک سیستم serialize کنیم و بخواهیم آن را در سیستم

دیگری بازسازی کنیم، ممکن است سیستم دوم نتواند داده را به درستی تفسیر کند. برای مثال یک integer ممکن است در یک زبان برنامه‌نویسی به صورت ۴ بایت و در زبان دیگری به صورت ۸ بایت ذخیره شود.

مثال دیگر :

```
class Student:
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

اگر این شیء را با استفاده از pickle ذخیره کنیم و بخواهیم آن را در یک برنامه Java باز کنیم، Java قادر به تفسیر این داده‌ها نخواهد بود چرا که pickle مخصوص Python است.

راه‌حل

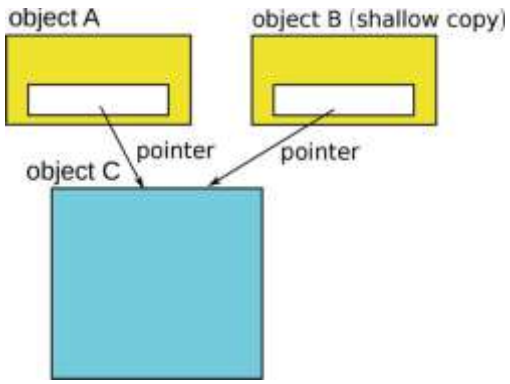
برای حل این مشکل از فرمت‌های مستقل از زبان استفاده می‌کنیم. یکی از بهترین گزینه‌ها Google Protocol Buffers (Protobuf) است. این فرمت با استفاده از یک Interface Description Language (IDL)، داده‌ها را به صورتی استاندارد ذخیره می‌کند.

۲-۵_ Object References

در برنامه‌های پیچیده اشیا به یکدیگر ارجاع می‌دهند. این ارجاعات هنگام serialization می‌تواند مشکلاتی ایجاد کند. اگر هنگام serialization، تمام ارجاعات موجود در یک شی را به درستی دنبال نکنیم، در زمان deserialization نسخه‌ای که ایجاد می‌شود دیگر نسخه‌ای دقیقی از منبع اصلی نخواهد بود.

چالش

هنگامی که یک شی شامل اشاره‌گرها یا ارجاعات به داده‌های دیگر باشد، لازم است که فرآیند serialization نه تنها آدرس این داده‌ها یا اشاره‌گرها را ذخیره کند، بلکه داده‌ای که به آن اشاره می‌شود نیز به طور کامل ذخیره شود. این کار برای اطمینان از بازسازی دقیق داده‌ها در زمان deserialization است.

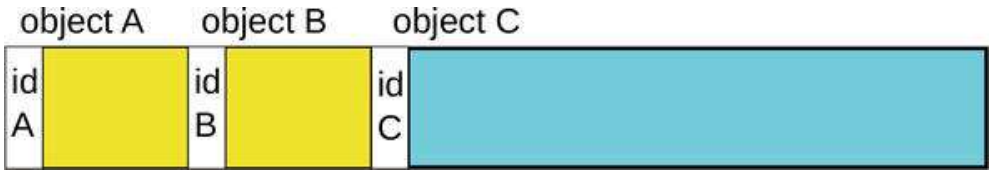


ساختارهای درختی

وقتی ارجاعات اشیا به صورت درختی باشند، مدیریت این ارجاعات ساده است. در این حالت می‌توانیم ساختار درختی را به صورتی در بیاوریم که داده‌های ارجاع‌شده را به‌عنوان بخشی از شی والد ذخیره کنیم.

چالش اصلی زمانی رخ می‌دهد که چندین شی به یک داده مشابه ارجاع دهند. به عنوان مثال اگر A و B هر دو به یک شی C ارجاع دهند، باید اطمینان حاصل کنیم که تنها یک نسخه از شی C ذخیره شود. در غیر این صورت، هنگام deserialization ممکن است شی C دوبار بازسازی شود که ساختار داده‌ها را تغییر می‌دهد.

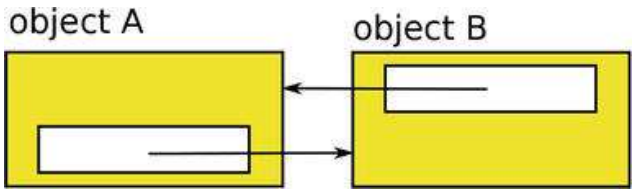
در شکل زیر، شی B یک shallow copy از شی A است و هر دو به C اشاره می‌کنند. برای مدیریت این وضعیت، ابزار serialization باید تنها یک نسخه از داده‌های C ذخیره کند و هر ارجاع را به همان نسخه ارجاع دهد.



ارتباطات بازگشتی

چالش دیگر زمانی پیش می‌آید که یک شیء به طور مستقیم یا غیرمستقیم به خودش ارجاع دهد و در چنین شرایطی، اگر ابزار serialization این ارجاعات را به درستی شناسایی نکند، ممکن است وارد یک حلقه بی نهایت شود.

در شکل زیر، یک شی دارای ارتباط بازگشتی است. برای جلوگیری از این مشکل، ابزار serialization باید ارجاعات را دنبال کرده و تنها یک نسخه از هر شی ذخیره کند.



راه‌حل

استفاده از Unique Identifiers: ابزار serialization می‌تواند به هر شی یک شناسه یکتا اختصاص دهد. در هنگام serialization، اگر شیئی قبلاً ذخیره شده به جای ذخیره مجدد آن، تنها شناسه یکتای آن در خروجی ذخیره می‌شود.

Reference Tracing: برای شناسایی ارجاعات تکراری یا بازگشتی، ابزار serialization باید ارجاعات را به‌صورت پویا ردیابی کند. این فرآیند بررسی می‌کند که آیا شی خاصی قبلاً ذخیره شده است یا خیر.

ابزارهایی مانند Protobuf و JSON-LD از قابلیت‌های مدیریت ارجاعات برخوردار هستند. این ابزارها به طور خودکار ارجاعات تکراری و بازگشتی را شناسایی و مدیریت می‌کنند.

برای پاسخ به این سوال از مقاله [3] استفاده شد.

۶_ پاسخ سوال شماره ۶

برای پاسخ به این سوال از منابع زیر استفاده شد:

https://www.geeksforgeeks.org/dynamic-vs-static-computational-graphs-pytorch-and-tensorflow/

https://medium.com/@abhishekjainindore24/static-vs-dynamic-computational-graphs-5a49d1e3030b

گراف محاسباتی یا Computational Graph در یادگیری عمیق یک ساختار داده‌ای است که عملیات‌ها و جریان داده‌ها را در یک مدل نشان می‌دهد که nodesها به عنوان عملیات مثل جمع، ضرب یا عملیات کانولوشن و یال ها (edges) به عنوان داده‌هایی که بین گره‌ها جریان پیدا می‌کنند تعریف می‌شود. این گراف به ما کمک می‌کند تا فرایند محاسبات و همچنین مراحل forwardو backpropagation برای به‌روزرسانی وزن‌ها در آموزش مدل را درک کنیم که در خود کتابخانه‌ها هم برای مدیریت مراحل استفاده می‌شود.

۶-۱_ تفاوت Computational Graph ایستا و پویا

در گراف محاسبات ایستا که نمونه‌ی آن در نسخه‌های اولیه‌ی TensorFlow قبل از نسخه ۲,۰ دیده می‌شود، ابتدا کل گراف به صورت کامل طراحی می‌شود و سپس اجرا می‌گردد. این گراف ثابت است و نمی‌توانیم در حین اجرا تغییری در ساختار آن بدهیم. مراحل کار به این صورت است که ابتدا گراف تعریف شده سپس وارد session می‌شویم و در آنجا داده‌ها را به گراف وارد کرده و نتایج را محاسبه می‌کنیم.

مزایا:

ساختار گراف از پیش مشخص است و سیستم می‌تواند آن را بهینه کند و منابع را بهتر مدیریت کند.

سرعت اجرا نسبت به گراف پویا کمی بیشتر است، زیرا گراف از قبل آماده شده است.

معایب:

انعطاف‌پذیری پایین: برای مثال، اگر داده‌هایی با ابعاد متفاوت داشته باشیم، باید کد اضافی بنویسیم یا گراف جدیدی تعریف کنیم.

مشکل در دیباگ کردن: اگر گراف اشتباه تعریف شود، ممکن است خطاها تنها در زمان اجرا مشخص شوند و این موضوع زمان و منابع زیادی را هدر می‌دهد.

در گراف محاسبات پویا که PyTorch از آن استفاده می‌کند، گراف همزمان با اجرای کد و انجام عملیات ساخته می‌شود. به عبارت دیگر، ساختار گراف به داده‌هایی که در لحظه وارد می‌کنیم وابسته است. این روش انعطاف‌پذیری بیشتری دارد، زیرا می‌توانیم ساختار گراف را در هر لحظه تغییر دهیم.

مزایا:

انعطاف‌پذیری بالا: به راحتی می‌توانیم مدل‌هایی با ساختار پیچیده یا داده‌هایی با ابعاد متغیر را پیاده‌سازی کنیم.

سهولت در دیباگ کردن: چون گراف به صورت لحظه‌ای ساخته می‌شود، می‌توانیم از ابزارهای معمول دیباگ در پایتون یا print استفاده کنیم تا مراحل اجرا را بررسی کنیم.

معایب

چون گراف در زمان اجرا ساخته می‌شود، بهینه‌سازی آن سخت‌تر است و ممکن است سرعت اجرای آن کمی پایین‌تر از گراف ایستا باشد و خب بر اساس هر batch داده آموزش یک گراف جدید ساخته می‌شود.

به طور کلی گراف محاسبات ایستا برای پروژه‌هایی مناسب است که از پیش مشخص هستند و تغییرات زیادی نیاز ندارند مثلا پروژه‌های بزرگ صنعتی. از سوی دیگر، گراف محاسبات پویا برای پژوهشگران و توسعه‌دهندگانی که نیاز به انعطاف‌پذیری بالا و دیباگ آسان دارند، گزینه‌ی بهتری است. شبکه‌های عصبی بازگشتی RNNs و عملیات شرطی نمونه‌هایی هستند که گراف‌های محاسباتی پویا در آن‌ها بسیار مفیدند،

زیرا این گراف‌ها امکان پردازش ورودی‌های با طول متغیر یا انجام عملیات متناسب با شرایط مختلف را به‌صورت بدون نیاز به تغییرات پیچیده در مدل فراهم می‌کنند.

منابع استفاده شده از مقالات وب‌سایت‌های [4] و [5] بودند.

۷_ پاسخ سوال شماره ۷

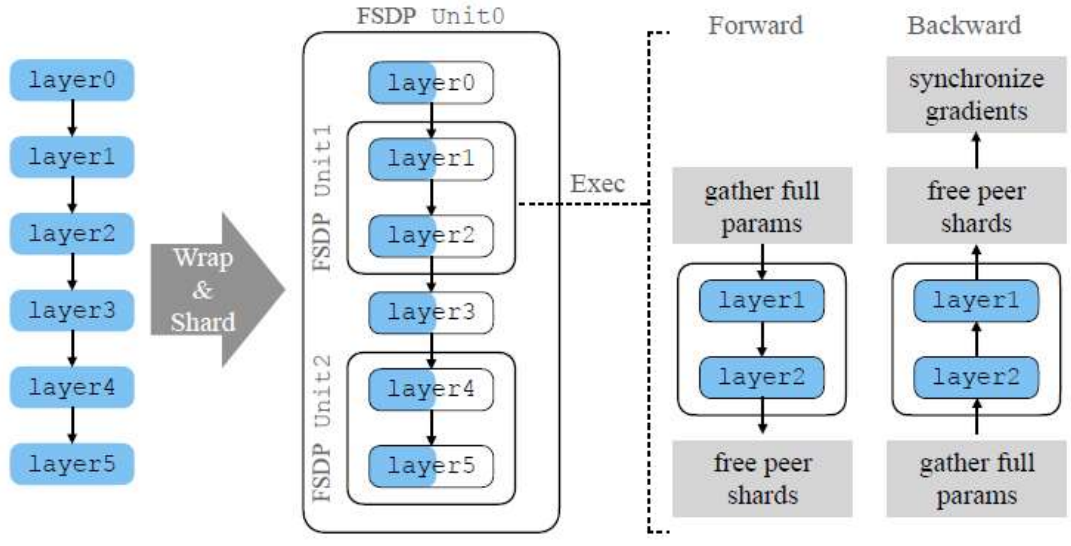
۷-۱_ مفهوم Model Sharding

با افزایش اندازه مدل‌های یادگیری عمیق و خصوصا مدل‌های زبانی بزرگ فعلی، نیاز به منابع محاسباتی بیشتری احساس می‌شود. در برخی موارد، مدل‌ها به قدری بزرگ هستند که نمی‌توانیم آن‌ها را به طور کامل در حافظه یک GPU قرار دهیم. در چنین شرایطی، Model Sharding به معنای تقسیم مدل به بخش‌های کوچکتر و توزیع آن‌ها بین چندین دستگاه محاسباتی مانند GPUها است تا بتوان از منابع بهینه‌تر استفاده کرد و مدل‌های بزرگ را آموزش داد.

۷-۲_ استراتژی‌های مختلف Model Sharding در PyTorch

:Fully Sharded Data Parallel (FSDP)

در این روش، پارامترهای مدل، گرادیان‌ها و وضعیت‌های بهینه‌ساز بین تمام دستگاه‌ها شارد می‌شوند. هر دستگاه تنها بخشی از پارامترها را نگه می‌دارد و در طول آموزش، پارامترهای کامل به صورت پویا جمع‌آوری و آزاد می‌شوند [6].



شکل ۷-۱ مروری بر الگوریتم FSDP

مزایا:

کاهش قابل توجه مصرف حافظه در هر دستگاه.

امکان آموزش مدل‌های بسیار بزرگ که در حافظه یک GPU جا نمی‌شوند.

معایب:

افزایش پیچیدگی ارتباطات بین دستگاه‌ها.

کاهش سرعت آموزش به دلیل نیاز به جمع‌آوری و توزیع مکرر پارامترها.

Tensor Parallelism

در این روش، تنسورهای وزن‌های مدل به بخش‌های کوچکتر تقسیم شده و بین دستگاه‌ها توزیع می‌شوند. این تکنیک به ویژه در مدل‌های ترنسفورمری بزرگ کاربرد دارد.

مزایا:

استفاده بهینه از منابع محاسباتی.

کاهش مصرف حافظه در هر دستگاه.

معایب:

پیچیدگی در پیاده‌سازی و نیاز به هماهنگی دقیق بین دستگاه‌ها.

کاربردها: مناسب برای مدل‌های با ماتریس‌های بزرگ، مانند مدل‌های زبانی بزرگ.

Sharding Embedding Tables

در سیستم‌های توصیه‌گر بزرگ، جداول Embedding داکیومنت‌ها می‌تواند بسیار بزرگ شوند. مثلا اگر از امبدینگ‌های Sparse مثل TFID استفاده شود. با استفاده از کتابخانه TorchRec در PyTorch، می‌توان این جداول را بین چندین GPU شارد کرد تا از حافظه بهینه‌تر استفاده شود.

مزایا:

امکان آموزش مدل‌های توصیه‌گر با جداول تعبیه‌سازی بزرگ.

کاهش مصرف حافظه در هر دستگاه.

معایب:

پیچیدگی در پیاده‌سازی و نیاز به هماهنگی بین دستگاه‌ها.

کاربردها: مناسب برای سیستم‌های توصیه‌گر با جداول امبدینگ بزرگ.

۷-۳_ مقایسه استراتژی‌ها

مصرف حافظه: FSDP بیشترین کاهش مصرف حافظه را فراهم می‌کند، در حالی که Tensor Parallelism و Sharding Embedding Tables نیز به کاهش مصرف حافظه کمک می‌کنند، اما به میزان کمتر.

پیچیدگی پیاده‌سازی: Tensor Parallelism و Sharding Embedding Tables نیاز به تنظیمات دقیق‌تری دارند، در حالی که FSDP با ارائه APIهای سطح بالا، پیاده‌سازی را ساده‌تر می‌کند.

سربار ارتباطات: FSDP نیاز به Communication بیشتری بین دستگاه‌ها دارد که می‌تواند به کاهش سرعت منجر شود، در حالی که Tensor Parallelism با کاهش Communication ، کارایی بهتری ارائه می‌دهد.

۷-۴_ موارد استفاده از Model Sharding

آموزش مدل‌های بزرگ: زمانی که مدل به قدری بزرگ است که در حافظه یک GPU جا نمی‌شود، Model Sharding امکان آموزش آن را فراهم می‌کند.

بهبود کارایی: با توزیع بار محاسباتی بین دستگاه‌ها، می‌توان زمان آموزش را کاهش داد و از منابع بهینه‌تر استفاده کرد.

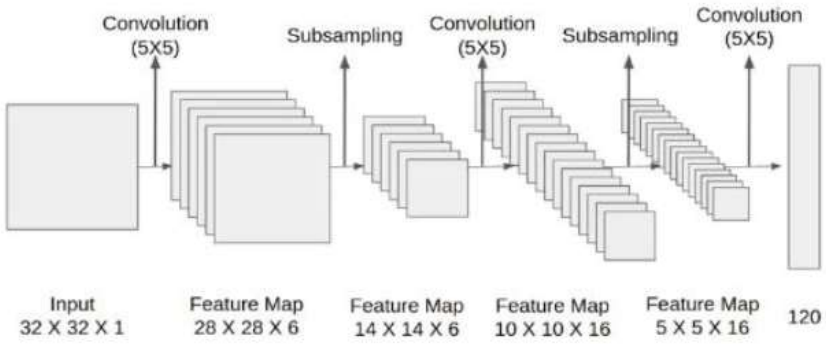
مقیاس‌پذیری: Model Sharding امکان مقیاس‌پذیری آموزش را با افزودن دستگاه‌های بیشتر فراهم می‌کند که مثلا میبینیم در آموزش مدل‌های زبانی بزرگ از چندین هزار GPU استفاده می‌شود.

۸_ پاسخ سوال شماره ۸

$$\text{Total Memory}_{\text{Training}} = \text{memory}_{\text{model}} + \text{memory}_{\text{optimizer}} + \text{memory}_{\text{gradients}} + \text{memory}_{\text{activations}}$$

۸-۱_ شمارش تعداد پارامترهای مدل LeNet

ابتدا باید تعداد پارامترهای مدل LeNet را محاسبه کنیم.



لایه Conv1

پارامترها = تعداد فیلتر × (کانال ورودی × اندازه کرنل + بایاس)

$$\text{Parameters} = (5 \times 5 \times 1 + 1) \times 6 = 156$$

لایه Subsampling

پارامتری ندارد.

لایه Conv2

$$\text{Parameters} = (5 \times 5 \times 6 + 1) \times 16 = 2416$$

لایه Subsampling2

پارامتری ندارد.

لایه Conv3

$$\text{Parameters} = (5 \times 5 \times 16 + 1) \times 120 = 48120$$

جمع کل پارامترها

$$\textbf{Parameters} = 156 + 2416 + 48120 = \textbf{50692}$$

۲-۸_ محاسبه حافظه مورد نیاز برای آموزش

محاسبه حافظه مدل

حافظه مدل = (تعداد پارامترها) × (اندازه هر پارامتر)

در دقت fp32 ۴ بایت به ازای هر پارامتر.

در دقت fp16 ۲ بایت به ازای هر پارامتر.

مثلاً اگر از fp32 استفاده کنیم:

$$\text{memory}_{\text{model}} = 4 \times 50692 = 202768 \quad \text{bytes} \approx 0.20 \quad \text{MB}$$

اگر از FP16 استفاده کنیم:

$$\text{memory}_{\text{model}} = 2 \times 50692 = 101384 \quad \text{bytes} \approx 0.10 \quad \text{MB}$$

اگر از mixed استفاده کنیم یک ۴ بایت به ازای هر پارامتر مدل هم باید ذخیره کنیم که این مورد را در آپتیمایزر در نظر میگیریم.

محاسبه حافظه اپتیمایزر

برای AdamW به‌ازای هر پارامتر، ۱۲ بایت نیاز است:

-fp32 copy of parameters: 4 bytes/param

-Momentum: 4 bytes/param

-Variance: 4 bytes/param

$$\text{memory}_{\text{optimizer}} = 12 \times 50692 = 608304 \quad \text{bytes} \approx 0.60 \quad \text{MB}$$

در SGD ۸ بایت به ازای هر پارامتر (وزن‌های مدل + ممنتوم):

-fp32 copy of parameters: 4 bytes/param

-Momentum: 4 bytes/param

$$\text{memory}_{\text{optimizer}} = 8 \times 50692 = 405536 \quad \text{bytes} \approx 0.40 \quad \text{MB}$$

آن ۴ بایتی که در قسمت قبل برای حالت mixed precision گفتیم اینجا پوشش داده شد.

محاسبه حافظه گرادیان‌ها

همانطور که در اسلایدها خواندیم معمولا مثل مدل ذخیره می‌شود.

اگر فرض کنیم گرادیان‌ها در فرمت fp32 ذخیره می‌شوند:

$$\text{memory}_{\text{model}} = 4 \times 50692 = 202768 \quad \text{bytes} \approx 0.20 \quad \text{MB}$$

اگر از FP16 استفاده کنیم:

$$\text{memory}_{\text{model}} = 2 \times 50692 = 101384 \quad \text{bytes} \approx 0.10 \quad \text{MB}$$

محاسبه حافظه Activations

حافظه اکتیویشن بستگی به batch size، تعداد لایه‌ها، ابعاد مخفی در معماری‌های ترنسفورمر دارد. در LeNet در اینجا از همان فرمول استفاده می‌کنیم طبق صحبت TA محترم در گروه . در صورت استفاده از فرمول کلی ارائه‌شده که بیشتر برای معماری‌های ترنسفورمر بیان شد سه سناریو داریم:

- No Recomputation (عدم محاسبه مجدد)
- Selective Recomputation (محاسبه انتخابی)
- Full Recomputation (محاسبه کامل)

s: تعداد توکن در ورودی در اینجا تصویر ۳۲ در ۳۲ است میتوانیم ۱۰۲۴ در نظر بگیریم.

b: اندازهBatch Size که در اینجا ۸ است.

h: Hidden Size . میتوانیم مثلا ۱۶ بگیریم که خروجی لایه کانولوشنی دوم ۱۶ کانال دارد.

L: تعداد لایه‌های مدل ۳ تا لایه اصلی کانولوشنی داریم. البته بستگی دارد میتوان Pool را جداگانه حساب کرد.

a: تعداد Attention Heads نداریم پس صفر است.

t: درجه Tensor Parallelism در اینجا ۱

حال ۳ حالت مختلف را محاسبه میکنیم:

حالت No Recomputation:

$$\text{memory}_{\text{No Recomputation activations}} = \text{sbhL}\left(10 + \frac{24}{t} + 5 \cdot \frac{a \cdot s}{h \cdot t}\right) \text{bytes}$$

$$\text{memory}_{\text{activations}} = s \cdot b \cdot h \cdot L = 32 \cdot 8 \cdot 16 \cdot 3 \cdot 34 = 417792 \quad \text{bytes} \approx 0.41 \quad \text{MB}$$

حالت Selective Recomputation:

$$\text{memory}_{\text{Selective Recomputation activations}} = \text{sbhL}\left(10 + \frac{24}{t}\right) \text{bytes}$$

$$\text{memory}_{\text{activations}} = s \cdot b \cdot h \cdot L = 32 \cdot 8 \cdot 16 \cdot 3 \cdot 34 = 417792 \quad \text{bytes} \approx 0.41 \quad \text{MB}$$

کاملا Recompute شوند:

$$\text{memory}_{\text{Full Recomputation activations}} = \text{sbhL bytes}$$

$$\text{memory}_{\text{activations}} = s \cdot b \cdot h \cdot L = 32 \cdot 8 \cdot 16 \cdot 3 = 12288 \quad \text{bytes} \approx 0.012 \quad \text{MB}$$

محاسبه حافظه کل برای آموزش

حالت های مختلفی داریم. ۱۶ بیتی یا ۳۲ بیتی. و اینکه activation ها ذخیره شوند یا نشوند.

$$\text{Total Memory}_{\text{Training}} = \text{memory}_{\text{model}} + \text{memory}_{\text{optimizer}} + \text{memory}_{\text{gradients}} + \text{memory}_{\text{activations}}$$

دقت	حالت اکتیویشن	حافظه کل	حافظه کل (مگابایت)
FP32	No Recomputation	202768 + 608304 + 202768 + 417792	1.43
FP32	Selective Recomputation	202768 + 608304 + 202768 + 417792	1.43
FP32	Full Recomputation	202768 + 608304 + 202768 + 12288	1.01
FP16	No Recomputation	101384 + 608304 + 101384 + 417792	1.22
FP16	Selective Recomputation	101384 + 608304 + 101384 + 417792	1.22
FP16	Full Recomputation	101384 + 608304 + 101384 + 12288	0.82

جدول ۱ حافظه مورد نیاز با استفاده از آپتیمایزر AdamW

دقت	حالت اکتیویشن	حافظه کل	حافظه کل (مگابایت)
FP32	No Recomputation	202768 + 405536 + 202768 + 417792	1.22
FP32	Selective Recomputation	202768 + 405536 + 202768 + 417792	1.22
FP32	Full Recomputation	202768 + 405536 + 202768 + 12288	0.82
FP16	No Recomputation	101384 + 405536 + 101384 + 417792	1.02
FP16	Selective Recomputation	101384 + 405536 + 101384 + 417792	1.02
FP16	Full Recomputation	101384 + 405536 + 101384 + 12288	0.62

جدول ۲ حافظه مورد نیاز با استفاده از آپتیمایزر SGD

محاسبه حافظه کل برای استنتاج:

در استنتاج با batch size = 1 تنها نیاز داریم وزن‌ها به همراه اکتیویشن‌های لایه‌ها که همان محاسبات میانی هستند را در GPU نگه داریم. طبق فرمول تخمینی در اسلاید:

$$\text{Total Memory}_{\text{Inference}} = 1.2 \times \text{memory}_{\text{model}}$$

$$\text{Total Memory}_{\text{Inference}} = 1.2 \times 50692 = 60830.4 \text{ bytes}$$

۳-۸_ روش دیگر محاسبه Activations

برای محاسبه حافظه مورد نیاز اکتیویشن‌ها در شبکه LeNet، می‌توان از فرمول زیر استفاده کرد که این فرمول از مدل زبانی دریافت شد و ممکن است معتبر نباشد:

$$\text{Activation Memory} = \text{Batch Size} \times \text{Output Height} \times \text{Output Width} \times \text{Number of Output Channels} \times \text{Bytes per Element}$$

Layer: Conv1

$$Memory: 8 \times 28 \times 28 \times 6 \times 4 = 150528 \ bytes$$

Layer: Pool1

$$Memory: 8 \times 14 \times 14 \times 6 \times 4 = 37632 \ bytes$$

Layer: Conv2

$$Memory: 8 \times 10 \times 10 \times 16 \times 4 = 51200 \ bytes$$

Layer: Pool2

$$Memory: 8 \times 5 \times 5 \times 16 \times 4 = 12800 \ bytes$$

Layer: Conv3

$$Memory: \ 8 \times 120 \times 1 \times 1 \times 4 = 3840 \ \ bytes$$

Total Activation Memory

$$Total\ Memory: 150528 + 37632 + 51200 + 12800 + 3840 = 256000 \ \ bytes$$

۹_ابزارهای استفاده شده

برای نگارش این تمرین از ابزارهای مبتنی بر مدل زبانی و همچنین ابزار پاک‌نویس برای رعایت نیم‌فاصله و علائم نگارشی استفاده شد.

پرامپت‌های استفاده شده :

Describe how different processors handle data endianness. Explain the difference between little-endian and big-endian formats with examples.

فرمت عدد 6.342×10^{-5}را به FP32 بر اساس استاندارد IEEE تبدیل کنید.

What are the challenges in data serialization in different programming languages? I need examples of object representation challenge.

Compare static and dynamic computational graphs in deep learning

Explain the different model sharding strategies used in PyTorch

تعداد کل پارامترهای LeNet چند تاست.

چگونه جداول امبدینگ در PyTorch شارد می‌شوند؟

آیا فرمولی برای تخمین حافظه مورد نیازactivation ها در یک مدل غیر از ترنسفورمر مثلا یک مدل کانولوشنی وجود دارد؟

۱۰_ منابع

[1] S. B. Balaji, M. N. Krishnan, M. Vajha, V. Ramkumar, B. Sasidharan, and P. V. Kumar, “Erasure coding for distributed storage: An overview,” Sci. China Inf. Sci., vol. 61, pp. 1–45, 2018.

[2] A. Nalajala, T. Ragunathan, R. Naha, and S. K. Battula, “Application and user-specific data prefetching and parallel read algorithms for distributed file systems,” Cluster Comput., vol. 27, no. 3, pp. 3593–3613, 2024, doi: 10.1007/s10586-023-04160-1.

[3] K. Grochowski, M. Breiter, and R. Nowak, “Serialization in Object-Oriented Programming Languages,” in Introduction to Data Science and Machine Learning, K. Sud, P. Erdogmus, and S. Kadry, Eds., Rijeka: IntechOpen, 2019, ch. 12. doi: 10.5772/intechopen.86917.

[4] GeeksforGeeks, “Dynamic vs Static Computational Graphs - PyTorch and TensorFlow,” 2022. [Online]. Available: https://www.geeksforgeeks.org/dynamic-vs-static-computational-graphs-pytorch-and-tensorflow/

[5] A. Jain, “Static vs Dynamic Computational Graphs,” 2019. [Online]. Available: https://medium.com/@abhishekjainindore24/static-vs-dynamic-computational-graphs-5a49d1e3030b

[6] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, and others, “Pytorch fsdp: experiences on scaling fully sharded data parallel,” arXiv Prepr. arXiv2304.11277, 2023.