



به نام خدا

دانشگاه تهران



پرهام بیچرانلویی - آناهیتا هاشم زاده

نام و نام خانوادگی

۸۱۰۱۰۰۵۰۲ - ۸۱۰۱۰۰۳۰۳

شماره دانشجویی

۱۴۰۱، ۱۰، ۲۰

تاریخ ارسال گزارش

دانشکده هندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین پنجم

فهرست

۱	پاسخ ۱. آشنایی با مفهوم توجه و پیاده سازی مدل BERT
۱	۱-۱. پیاده سازی کدگذار
۳	
۴	
۶	۱-۲. پیاده سازی مدل BERT
۶	
۷	
۹	پاسخ ۲- آشنایی با کاربرد تبدیل کننده‌ها در تصویر
۹	۲-۱. آشنایی با مدل BEIT
۹	۲-۲. تقسیم بندی معنایی تصاویر
۱۴	۳-۲. طبقه بندی تصاویر
۲۲	۴-۲. پرسش‌ها

شکل‌ها

- ۱ شکل ۱. لایه feed forward + normalization + لایه multi head attention
شکل ۲ . لایه encoder
شکل ۳. جزئیات مربوط به لایه encoder
شکل ۴. جزئیات لایه multi-head attention
شکل ۵ . ورودی و embeddings مدل BERT
شکل ۶ . اصل تصویر ۱۱۵
شکل ۷ . سگمنت پیشفرض تصویر ۱۱۵
شکل ۸ .. اصل تصویر ۳۰
شکل ۹ . سگمنت پیشفرض تصویر ۳۰
شکل ۱۰ . اصل تصویر ۴۰
شکل ۱۱ . سگمنت پیشفرض تصویر ۴۰
شکل ۱۲ . سگمنت تصویر ۱۱۵ با مدل BeIT
شکل ۱۳ . سگمنت تصویر ۳۰ با مدل BeIT
شکل ۱۴ . سگمنت تصویر ۴۰ با مدل BeIT
شکل ۱۵ . متریک IoU
شکل ۱۶ . نمودار training loss برای مدل MLP
شکل ۱۷ . نمودار training accuracy برای مدل MLP
شکل ۱۸ . نمودار validation loss برای مدل MLP
شکل ۱۹ . نمودار validation accuracy برای مدل MLP
شکل ۲۰ . نمودار ماتریس آشفتگی برای مدل MLP
شکل ۲۱ . نمودار training loss برای مدل BeIT
شکل ۲۲ . نمودار validation loss برای مدل BeIT
شکل ۲۳ . ماتریس آشفتگی مدل BeIT
شکل ۲۴ . تفاوت توجه محلی و سراسری با کانولوشن

جدول‌ها

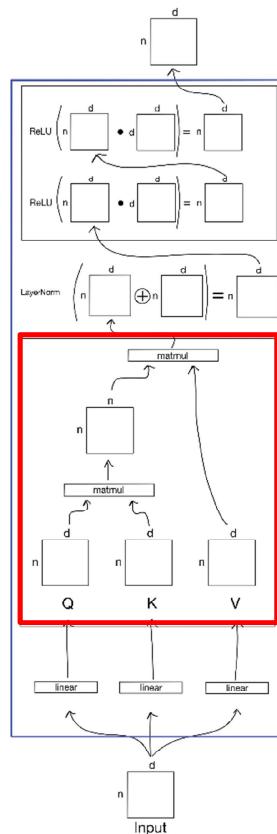
۶hyper parameters . جدول ۱.

پاسخ ۱. آشنایی با مفهوم توجه و پیاده سازی مدل BERT

۱-۱. پیاده سازی کدگذار

در این قسمت بخش مربوط به encoder مدل را کامل می کنیم، به این صورت که ابتدا تابع MultiHeadAttention را به صورت زیر تعریف میکنیم:

```
1. def attention(self, query, key, value, mask):
2.     ##### complete this part #####
3.     score = tf.matmul(query, key, transpose_b=True)
4.     dim_key = tf.cast(tf.shape(key)[-1], tf.float32)
5.     scaled_score = score / tf.math.sqrt(dim_key)
6.     maxlen = tf.cast(tf.shape(scaled_score)[-1], tf.int64)
7.     m = tf.repeat(mask, maxlen, axis=2) * (-1e9)
8.     scaled_score += m
9.     weights = tf.nn.softmax(scaled_score, axis=-1)
```



شکل ۱. لایه $+ normalization$ + $multi head attention$ + $feed forward$ لایه + لایه

در مدل BERT از تابع فعال ساز intermediate dense GELU در لایه استفاده می شود، به این

تابع به صورت زیر پیاده سازی می شود:

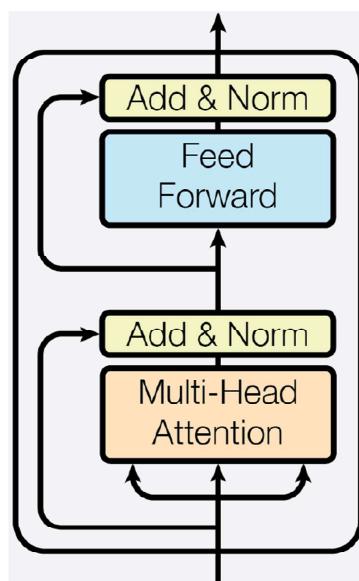
$$GELU(x) = xp(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \right]$$

$$\text{approximate GELU with: } 0.5x(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right])$$

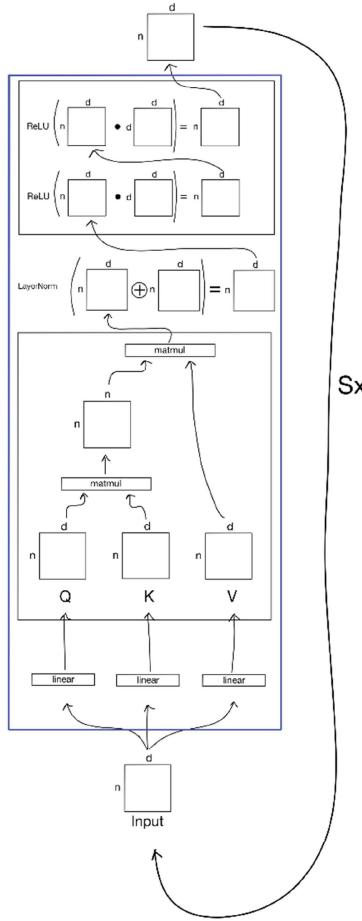
```
1. def GELU(x, approximate=False):
2.     ##### complete this part #####
3.     if approximate==False:
4.         cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
5.     else:
6.         cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3))))) )
7.     return x*cdf
```

زیر لایه feed forward مربوط به encoder دارای دو لایه dense می باشد، یکی لایه dropout و دیگری لایه intermediate است و سپس یک output بر روی خروجی مربوط به لایه intermediate اجرا می شود.

در تابع AddNorm مربوط به encoder بر روی خروجی هر زیرلایه dropout اجرا شده سپس با ورودی زیرلایه جمع می شود و درنهایت از یک لایه normalization عبور می کند. سپس در کلاس Encoder توابع مربوط به MultiHeadAttention، FFN و AddNorm را که پیشتر تعریف کردیم فراخوانی میکنیم.



شکل ۱ . لایه encoder



شکل ۲. جزییات مربوط به لایه **encoder**

.۱

در مغز مکانیزم توجه باعث تمرکز بیشتر بر روی یک بخش از ورودی یا حافظه، (در عین حال توجه کمتری به دیگران) می شود پس از این طریق فرایند استدلال را هدایت می کند. مکانیزم توجه برای ارتقا کارایی مدل encoder-decoder معرفی شد که ایده پشت آن برای دستیابی decoder به مرتبط ترین بخش های دنباله ورودی از طریق ترکیب وزن دار همه بردارهای ورودی encoded می باشد به گونه ای که به بردارهای مرتبط تر وزن های بالاتری نسبت می دهد. مکانیزم توجه اولین بار در سال ۲۰۱۵ در مقاله ای که توسط Bahdanau نوشته شده بود، برای حل مسئله ترجمه ماشینی معرفی شد. می توان گفت استفاده از مکانیزم توجه کمک می کند علاوه بر مسئله اصلی، یک مسئله از نوع alignment را هم حل کنیم.

به بیان دیگر مکانیسم‌های توجه، از تکنیک‌های پردازش ورودی برای شبکه‌های عصبی هستند که به شبکه اجازه می‌دهند تا بر جنبه‌های خاص یک ورودی پیچیده تمرکز کند، تا زمانی که کل مجموعه داده طبقه‌بندی شود. هدف این است که وظایف پیچیده را به مناطق کوچکتر توجهی که به صورت متوالی پردازش می‌شوند، تقسیم کنیم.

یک انکودر یک لیست از بردارها را به عنوان ورودی می‌گیرد. انکودر این بردارها را با ارسال به لایه‌ی Attention و سپس لایه‌ی شبکه عصبی Feed-Forward پردازش کرده و سپس خروجی را به سمت انکودر بعدی ارسال می‌کند. در واقعتابع توجه را می‌توان یک map برای query و یک مجموعه key بر خروجی دانست که خروجی به عنوان مجموع وزن دار مقادیر بدست می‌آید بطوری که وزن اختصاص داده شده به هر مقدار توسطتابع سازگاری query با key مربوطه محاسبه می‌شود.

توجه به خود Self-Attention نوع خاصی از دسته‌بندی‌های مکانیزم توجه است که برای درک بهتر مدل‌های تبدیل‌کننده یا transformer به آن هست. هدف Self-Attention این است که ارتباط اجزای موجود در یک دنباله را با یکدیگر بسنجد تا بتواند برداشت درست‌تری از کل دنباله داشته باشد. تنها تفاوت این مکانیزم با توجه عادی این است که در توجه به خود، دنباله‌ی ورودی و خروجی یکسانند.

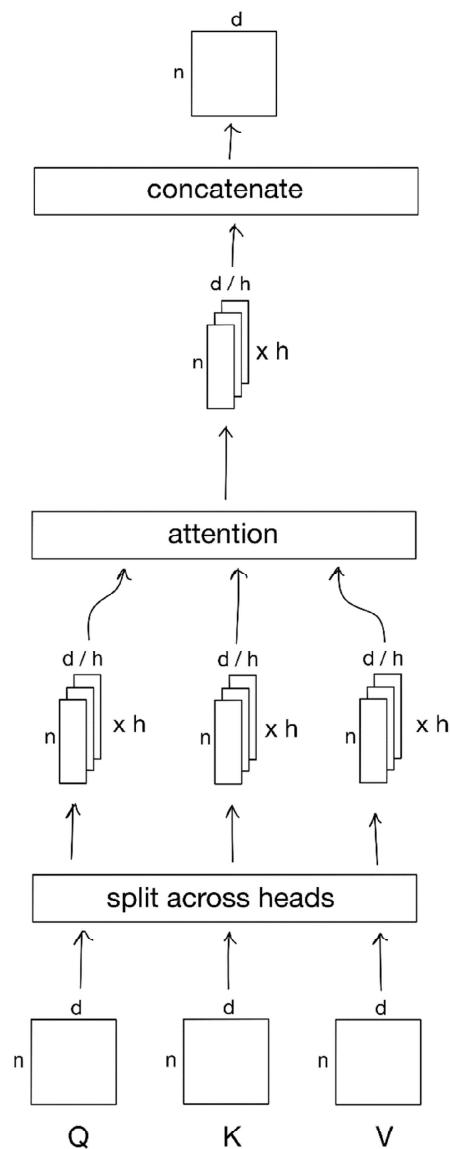
مکانیزم توجه چندسر (Multi-Head Attention) را می‌توان تعمیم‌یافته‌ی نسخه قبلی آن دانست. در توجه چندسر ما با سه موجودیت مختلف به نام کلید (Key)، مقدار (Value) و پرسش (Query) سروکار داریم. هدف این کار، تعیین میزان ارتباط بین هر جزء جمله خروجی با تمامی اعضای جمله ورودی است. در نوع تعمیم‌یافته‌ی توجه، کلید و مقدار به ازای هر جزء رشته ورودی و پرسش به ازای اجزای جمله خروجی مشخص می‌شوند. هر سه ورودی این بلوک توجه از یک منبع که همان جمله‌ی ورودی است می‌آیند، پس از نوع توجه به خود یا Self Attention است و هدف آن تعیین ارتباط و درک بهتر اجزای جمله‌ی زبان مبدأ به یکدیگر است.

۲.

در multi-head attention اجزاهای transformer در ترکیب اطلاعات بین قطعات یک دنباله ورودی کنترل شود، که باعث ایجاد نمایش غنی تر در نتیجه باعث افزایش عملکرد در وظایف یادگیری ماشین می‌شود. در واقع به جای اجرای یکتابع single attention با کلید و مقادیر و query بهتر است به صورت خطی کلیدها، queries و مقادیر را h مرتبه با پیش‌بینی d_{model} – dimentional های خطی متفاوت و آموزش دیده با ابعاد d_k , d_v و d_k نمایش داد. چون در هر یک از این نسخه‌های پیش‌بینی شده از keys, values و queries تابع توجه به صورت موازی اجرا می‌شود و مقادیر خروجی

projected می آید و در نهایت با هم concatenate می شوند و دوباره $d_v - dimensional$ میشوند.

اجازه می دهد که مدل به طور مشترک اطلاعات از زیرفضاهای مختلف در موقعیت های مختلف را بررسی کند. در حالی که در single attention head میانگین گیری از این موضوع جلوگیری میکند. با توجه به کاهش ابعاد هر head کل هزینه محاسباتی در این روش مشابع هزینه single head attention با ابعاد کامل است.



شكل ۳. جزئیات لایه multi-head attention

۱-۲. پیاده سازی مدل BERT

در قسمت قبل encoder را پیاده سازی کردیم و در این قسمت قصد داریم BERT را پیاده سازی کنیم. همانطور که خواسته شده ابتدا دو لایه token embedding و position embedding را در کلاس BertEmbedding تعریف میکنیم، به این صورت که همه embedding ها را با هم جمع کرده از dropout normalization عبور می دهیم سپس یک لایه pooler را بر روی آن اجرا می کنیم. آخرین لایه ای است که به آن نیاز داریم، در واقع pooler از طریق یک لایه dense برای هر جمله ورودی آخر انکوادر را به hidden state تغییر می دهد.

جدول ۱. hyper parameters

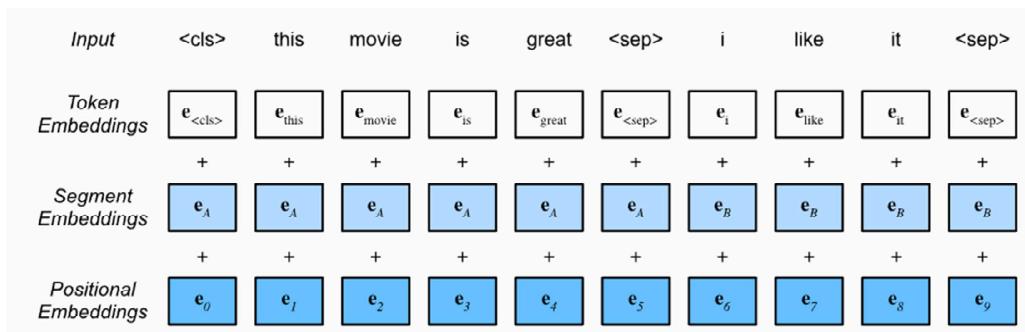
در انتهای نیز تابع create_BERT را کامل می کنیم. در این تابع مدل BERT را با استفاده از ورودی های داده شده (hyper-parameter) می سازیم. در این مدل همانطور که گفته شده تعداد لایه های انکوادر برابر ۱۲ می باشد.

Network Used Hyper Parameters	
Batch size	128
epoch	2
MAXLEN	32
hidden Size	768
Num attention heads	12
Num layers (encoders)	12
Vocab size	tokenizer.vocab_size
Intermediate size	12
Drop rate	0.1

.۱

در مدل bert segment embedding زمانی که ورودی مدل دارای جفت جمله (sentence pairs) باشد کاربرد دارد. در واقع segment embedding شماره جمله هایی که در وکتور encode شده اند می باشد. مدل باید بداند که یک token خاص در مدل به جمله A یا به جمله B تعلق دارد، و این امر

با تولید یک token ثبت به نام bert segment embedding بدست می آید. مدل bert توانایی حل مسائل nlp که شامل طبقه بندی متن با یک جفت متن ورودی می باشد. یک مثال برای این موضوع طبقه بندی اینکه آیا دو قطعه متن از نظر معنایی مشابه هستند می باشد. جفت متن ها concatenate شده سپس به مدل وارد می شوند و bert از طریق segment embedding ورودی جفت داده شده را تشخیص می دهد.



شکل ۴ . ورودی و BERT embeddings مدل

.۲

در بخش های قبل کدهای مربوط به encoder و مدل کامل شده، سپس با پارامترهای گفته شده مدل تعریف و آموزش داده شد. در تابع get_att_weights ابتدا ورودی token list را به مدل می دهیم و وزن های attention آنها را بدست می آوریم سپس یک لیست از وزن های مربوط به هر انکودر مربوط به زیرلایه multi head attention می سازیم. بعد از بدست آوردن token ها و لایه های توجه یه نمونه را به مدل داده و attention را نمایش دادیم.

```

1. sentence = "I enjoyed watching this movie"
2. toks, atts = get_att_tok(model, sentence.lower())
3. call_html()
4. head_view(atts, toks)

```

i
enjoyed
watching
this
movie

i
enjoyed
watching
this
movie

پاسخ ۲ - عنوان پرسش دوم به فارسی

۱-۲. آشنایی با مدل BEiT

همانطور که در سوال هم گفته شده است اساس کار ترانسفورها شامل سه مرحله اساسی است.

اول اینکه چون مدل پارامترهای زیادی دارد برای اینکه مدل overfit نکند و دقت بالایی بگیریم نیاز است که آن را با دیتاستهای بسیار بزرگ آموزش دهند و به خروجی این مدل‌ها اصطلاحاً مدل-pre train شده می‌گویند. حالا ما می‌توانیم مدل را با دیتاست کوچک خودمون برای کاربرد خاص‌مون دوباره آموزش بدھیم تا مقادیر پارامترهای مدل مناسب دیتای ما تغییر کنند. به این کار fine tune کردن مدل می‌گویند. در آخر هم مدل رو برای دیتاست تست‌مون آزمایش می‌کنیم.

چند نکته:

- محیط اجرا: Google Colab
- پردازنده: GPU-T4
- کتابخونه‌های اصلی: torch – transformers – numpy – datasets – evaluate
- نام کد قسمت ۲-۲: image_segmentation_using_beit_v2.ipynb
- نام کد قسمت ۳-۲: Q2_MLP_vs_BeIT(Image classification).ipynb

۲-۲. تقسیم بندی معنایی تصاویر

برای بارگذاری دیتای 'load_dataset' از کتابخونه datasets و ماژول scene_parse_150 استفاده کردیم(قبلش آن را نصب می‌کنیم). که دیتاست را در قالب یک دیکشنری با فیچرهای image, annotation, scene_category می‌دهد.

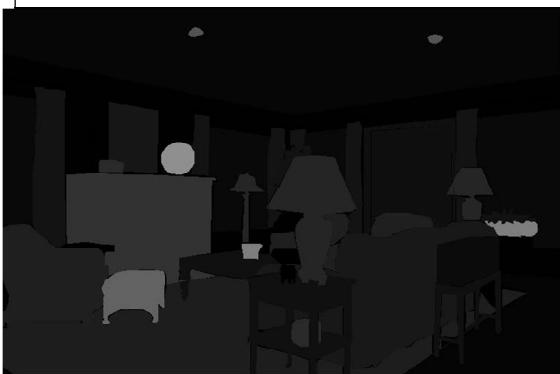
حالا سه تا عکس با ایندکس‌های ۱۱۵ و ۳۰ و ۴۰ را از این دیتاست خواندیم و تصویر اصلی و تصویر با تقسیم بندی معنایی آن‌ها به ترتیب در شکل زیر آمده است. که همانطور که در تصاویر سمت چپ مشخص است سگمنت پیشفرض عکس‌ها طیف رنگ خاکستری داره.



شکل ۷. سگمنت پیشفرض تصویر ۱۱۵



شکل ۶. اصل تصویر ۱۱۵



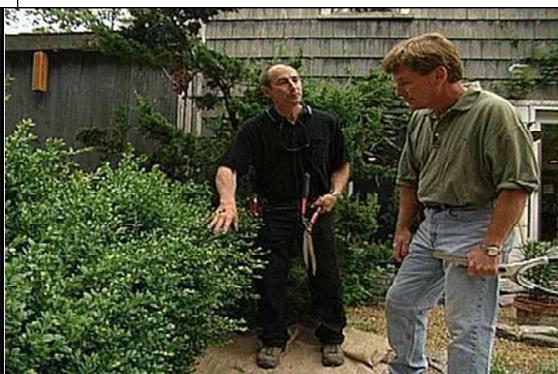
شکل ۹. سگمنت پیشفرض تصویر ۳۰



شکل ۸. اصل تصویر ۳۰



شکل ۱۱. سگمنت پیشفرض تصویر ۴۰



شکل ۱۰. اصل تصویر ۴۰

برای تقسیم بندی معنایی با استفاده از BEiT از کتابخونه transformers استفاده کردیم.

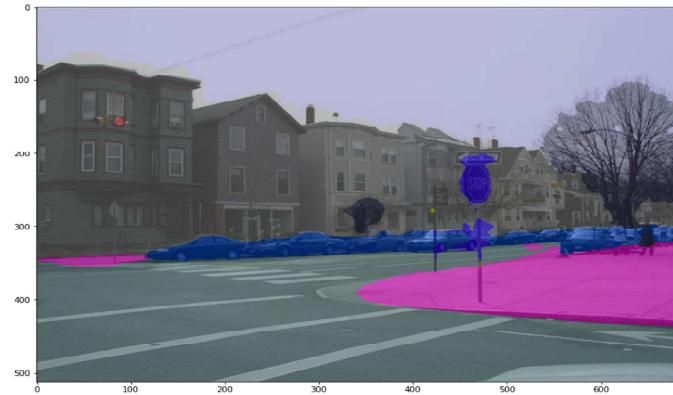
چون در اصلاحیه سوال آمده است که میتوانید از خود مدل pre_train برای سگمنت کردن استفاده کنید و ما محدودیت حافظه برای اجرای fine-tune کردن داشتیم دیگر از همان مدل pre_train مستقیم استفاده کردیم. کل این کار را در تابع segmentation_beit انجام دادیم.

این تابع ابتدا برای آمده کردن تصاویر برای دادن به مدل از مژول AutoFeatureExtractor در کتابخونه transformers استفاده میکنیم. و مژول from_pretrained آن را با پارامتر "microsoft/beit-base-finetuned-ade-640-640" فراخوانی میکنیم. که این پارامتر اسم یک مدل pre_train شده روی BeIT است. کلا کار feature_extractor از تغییر اندازه و نرمالیز کردن تصاویر برای مدل خلاصه میشے. حالا مدل را تعریف میکنیم. این کار با استفاده از مژول BeitForSemanticSegmentation از کتابخونه transformers به سادگی و در یک خط قابل انجام است. و برای انتخاب مدل pre_train شده همانطور که گفتیم از "microsoft/beit-base-finetuned-ade-640-640" استفاده میکنیم.

پس از دادن تصاویر به feature_extractor آنها را به مدل تعریف شده میدهیم. و از خروجی آن logits را فراخوانی میکنیم که نمره پیش بینی برای هر توکن قبل از softmax را برمیگرداند. حالا logits را به سایز اصلی تصویر تغییر سایز میدهیم. برای این کار از روش interpolate استفاده میکنیم. در آخر برای بصری کردن سگمنتیشن از colormap استفاده میکنیم. که کارش این است که به هر سگمنتی یک رنگ مشخص و استاندار اختصاص دهد. برای دیدن این مقادیر به تابع create_ade20k_label_colormap در کد مراجعه شود.

سپس خروجی را با استفاده از کتابخانه matplotlib نمایش میدهیم.

برای سه شکل ۱۱۵ و ۳۰ و ۴۰ این تابع گفته شده را فراخوانی میکنیم که نتایج آن در صفحه بعد آمده است. اگر به خروجی‌ها دقیق مدل تا حد خوبی توانسته از پس سگمنت کردن تصاویر برباید. البته برای بعضی شکل‌ها مثل تصویر ۳۰ یکم نادرست است. که علت آن اینکه مدل را ما-fine-tune با داده‌های خود نکردیم. برای همین چون توزیع داده‌های ما رو ندیده است دقیق ندارد اما باز هم توانسته که تا حد قابل قبولی اشیا را سگمنت کند. که این نشان از قدرت ترنسفورمر BeIT دارد.



شکل ۱۲. سگمنت تصویر ۱۱۵ با مدل BeIT



شکل ۱۳. سگمنت تصویر ۳۰ با مدل BeIT



تصویر ۱۴. سگمنت تصویر ۴۰ با مدل BeIT

در اصلاحیه تمرین گفته شده نیاز نیست fine-tune کنید اما کد آن زده شود. برای همین آن را هم در ادامه توضیح خواهم داد.(در انتهای کد هم زده شده است).

ابتدا داده آموزشی را به train و test می کنیم. خود کتابخونه datasets قبل آنها را جدا کرده و ما فقط آنها را می خوانیم.

حالا دیتا را پیش پردازش می کنیم. کارهایی مثل تبدیل لیبل به آیدی و برعکس که این بعضی جاهای کارمون رو آسان تر می کند. و برای آگمنت کردن داده از کتابخونه ColorJitter استفاده می کنیم که به صورت تصادفی روشنایی، کنتراست و اشباع شدگی تصویر را تغییر می دهد.(این مرحله اختیاری است). کلا آگمنتیشن کمک می کنه مدل مقاوم تر شود.

مرحله بعد آماده کردن توابع برای evaluate کردن مدل است. کتابخانه evaluate را نصب و import کرده. تا بتوانیم از روش IoU (Intersection over Union) استفاده کنیم. در این روش ناحیه ای که بین سگمنتیشن پیش بینی شده و واقعیت مشترک است تقسیم بر اجتماع آنها می شود.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} = \frac{\text{Intersection}}{\text{Union}}$$

شکل ۱۵. متریک IoU

در تابع compute_metric از این متریک برای اندازه گیری دقیق سگمنتیشن روی تصاویر استفاده می شود.

بعد از آن نوبت آموزش شبکه می رسد. که برای آن از کتابخونه transformer ماذول های Trainer و TreainingArguments وارد می کنیم. این ماذول های آماده آموزش شبکه را بسیار راحت می کنند. در واقع یک api فراهم کرده اند که ما بتولیم به سرعت عملیات هایی مثل fine_tune کردن، تعریف مدل های ترنسفورمر و ... را انجام دهیم. ماذول TrainingArguments پارامترهایی مثل نرخ یادگیری، تعداد ایپاک ها، سایز بچ، نحوه ارزیابی(هر ایپاک یا هر مرحله) و ... را برای نحوه آموزش شبکه مشخص می کند.

و مازول Trainer مدلی که می‌خواهیم آموزش دهیم (در اینجا BeIT)، پارامترهای مدل، دیتای آموزش و ارزیابی و نحوه ارزیابی مدل را مشخص می‌کند.

حالا با فراخوانی `trainer.train()` تابع را آموزش می‌دهیم. که به خاطر پر شدن حافظه این مرحله را نتوانستیم اجرا کنیم. دلیلش این اینکه این مدل ۱۶۳۴۰۷۹۸۰ پارامتر دارد! و محیط اجرای کولب فقط ۸ گیگ رم می‌دهد که اصلاً کافی نیست. حتی با بخش کوچکی از داده هم نتوانستم این مدل را بازآموزش دهم.

بعد از آن هم مثل چیزی که قبلاً گفته‌یم می‌توانیم خروجی را با انجام چند عملیات ساده نشان دهیم.

کد این تمرین در فایل `image_segmentation_using_beit_v2.ipynb` پیاده سازی شده است که همراه گزارش در فایل ارسالی ضمیمه شده است.

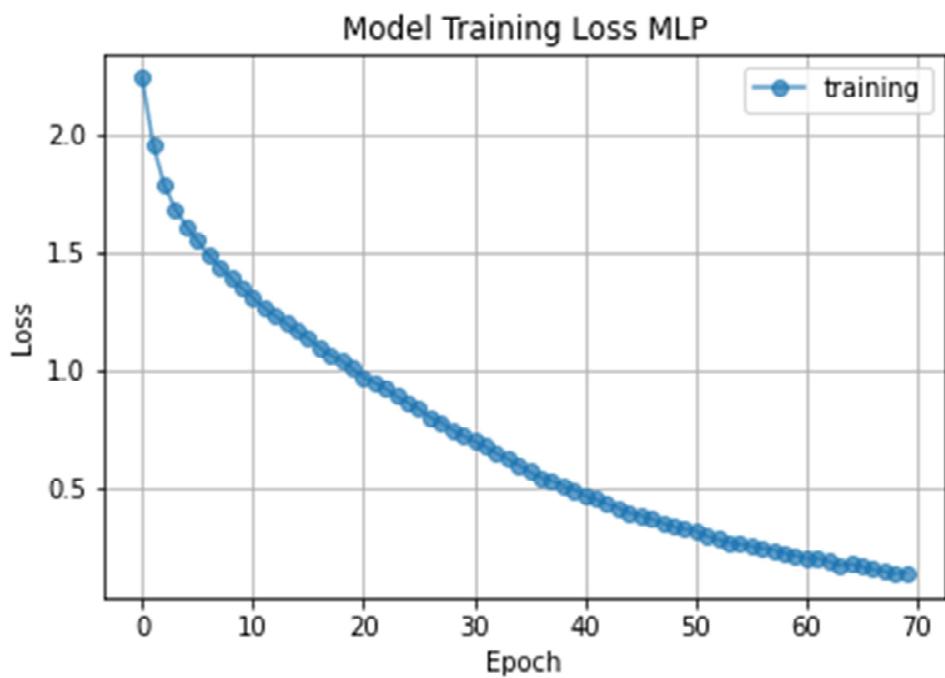
۳-۲. طبقه بندی تصاویر

دیتاست CIFAR-10 را با دو روش بارگذاری می‌کنیم یکبار با `torchVision` برای استفاده در مدل MLP. که بعد از آن با `dataLoader` آن را آماده آموزش و ارزیابی می‌کنیم. یکبار هم با کتابخانه `datasets` برای استفاده در مدل BeIT. چون از برخی مازولهای آماده برای بازآموزش BeIT استفاده می‌کنیم خواندن با این روش کار ما را ساده‌تر می‌کند.

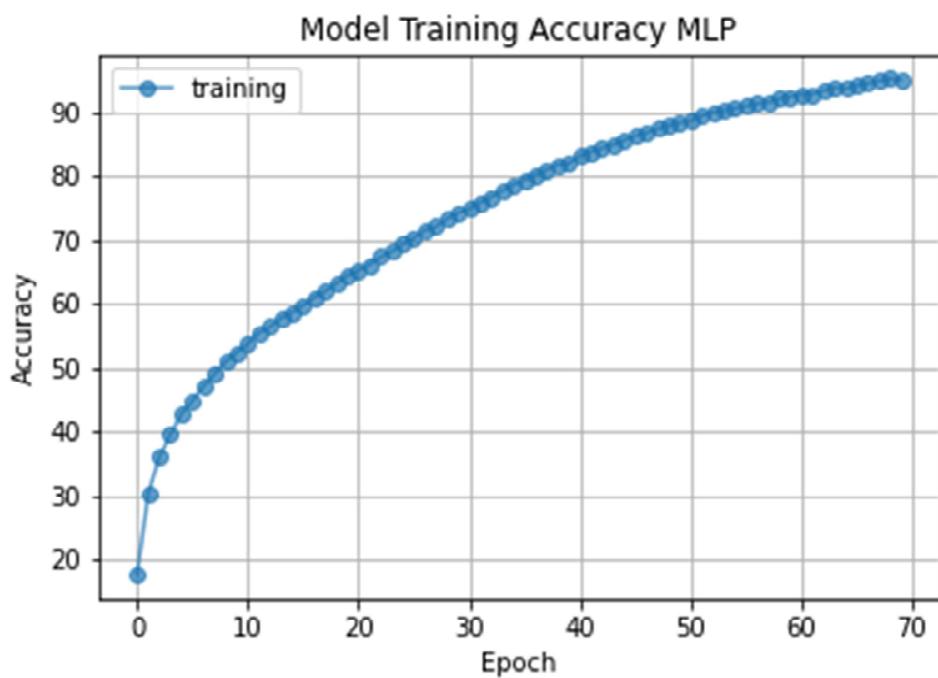
برای ساخت طبقه MLP از کتابخانه پایتورچ استفاده می‌کنیم. از ۴ لایه فولی کانکت استفاده می‌کنیم که هر لایه به ترتیب ۵۱۲، ۲۵۶، ۱۲۸، ۱۰ نورون دارند. تابع فعال ساز هم `relu` است البته به جز لایه آخر که از `log_softmax` استفاده می‌کند. قبل از لایه آخر هم از `drop out` برای جلوگیری از overfitting استفاده می‌کنیم. نکته دیگه اینکه ورودی چون تصویر است قبل از دادن به اولین لایه باید فلت بشه که با تابع `view` این کار رو می‌کنیم.

حالا باید یک بهینه ساز و هایپرپارامترهای مدل را مشخص کنیم. که از SGD همراه مومنتوم استفاده کردیم. نرخ یادگیری هم ۰.۰۰۱ قرار دادیم. برای loss هم معیار CrossEntropy را استفاده می‌کنیم.

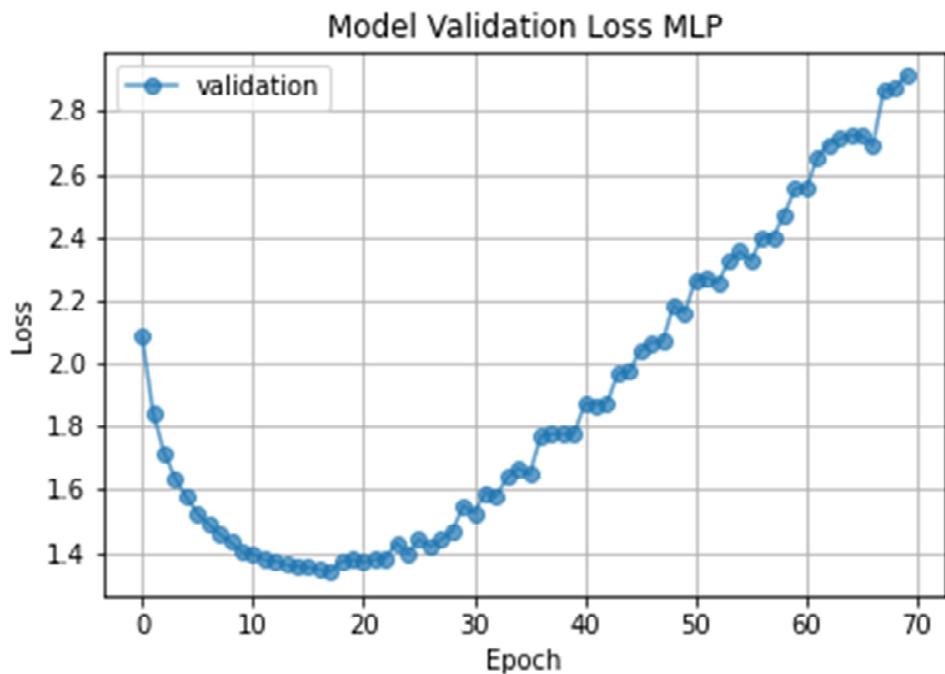
پس از اون مدل MLP نسبتاً ساده خودمون رو که خیلی هم برای یافتن هایپرپارامترهای خوب برای آن تلاش نشده است را در ۴۰ ایپاک آموزش می‌دهیم. و در هر ایپاک مقدار loss و دقت را برای داده train و validation بدست می‌اریم. و بعد آن‌ها را با کتابخانه `matplotlib` رسم می‌کنیم:



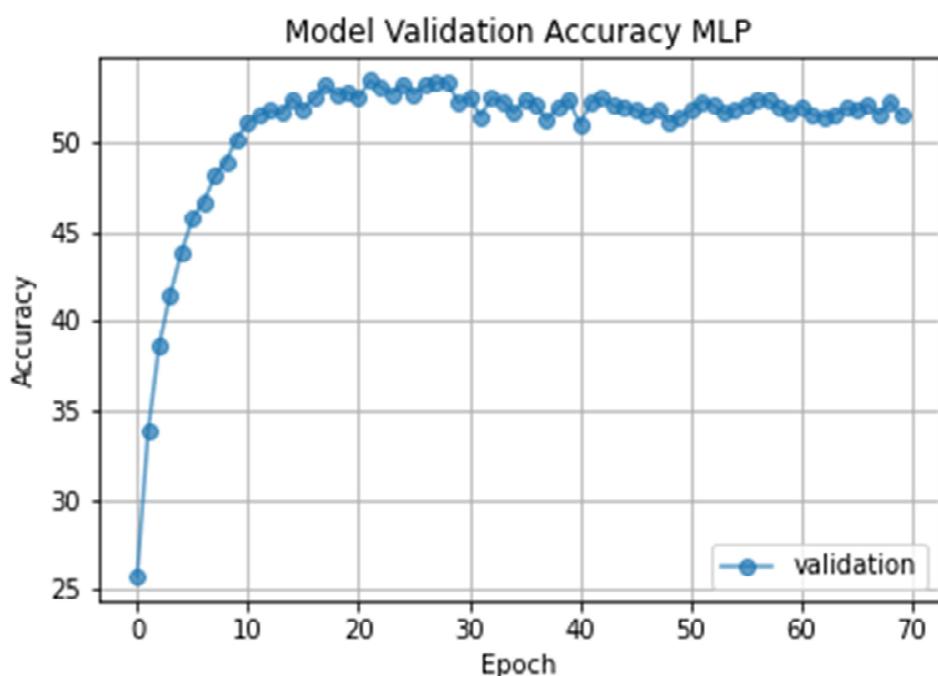
شکل ۱۶. نمودار training loss برای مدل MLP



شکل ۱۷. نمودار training accuracy برای مدل MLP



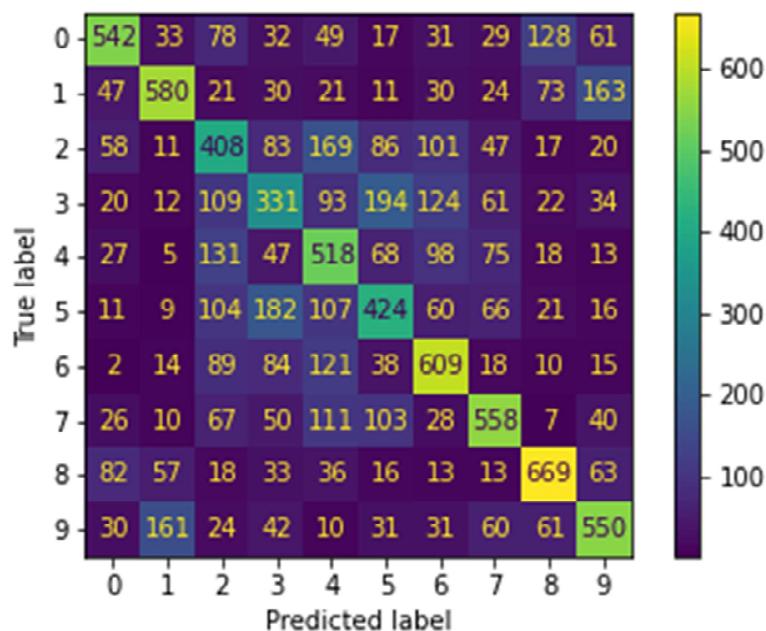
شکل ۱۸. نمودار validation loss برای مدل MLP



شکل ۱۹. نمودار validation accuracy برای مدل MLP

همانطور که در شکل واضح است می‌توان استنباط کرد قبل از اینکه ۲۰ مدل شروع به overfit شدن می‌کند چون loss برای دیتای validation آن بعد از این نقطه شروع به بیشتر شدن می‌کند. در حالیکه loss برای دیتای آموزشی همچنان در حال کاهش است. برای حل این مشکل کارهای مختلفی می‌شود انجام داد که یکیش گذاشتن dropout بیشتر در مدل است. و کار دیگر early stopping است. البته این روش فقط قبل از اینکه overfit شدن رخ بدهد مدل را متوقف می‌کند. اما چون این‌ها خیلی مورد هدف سوال نیست از اینکارها صرف نظر شده است. وقت شود که دقت validation تقریباً ثابت مونده پس early stopping هم نداشته باشیم که نداریم باعث بدتر شدن مدل نمی‌شود. فقط کمی زمان اجرا هدر دادیم ولی به جاش مدل ساده‌تر و خواناتر است!

حالا می‌خواهیم ماتریس آشتفتگی را رسم کنیم. برای اینکار اول باید کلاس داده‌های validation را با مدل پیش‌بینی کنیم (از تابع predict_mlp استفاده می‌کنیم که تعریف کردیم) و با لیبل‌های واقعی آن‌ها مقایسه کنیم. برای مقایسه و بدست آوردن این ماتریس مقادیر پیش‌بینی شده و لیبل‌های واقعی را به تابع confusion_matrix می‌دهیم. و برای نمایش هیئت مپ آن ماتریس خروجی را به مازول ConfusionMatrixDisplay می‌دهیم. در شکل زیر آن را نمایش دادیم:



شکل ۲۰. نمودار ماتریس آشتفتگی برای مدل MLP

در آخر هم گزارش طبقه بندی‌ها را برای داده‌های ارزیابی روی مدل MLP را استفاده از ماژول

ارائه می‌دهیم که پایین‌تر گزارش شده است:

precision recall f1-score support

class 0	0.64	0.54	0.59	1000
class 1	0.65	0.58	0.61	1000
class 2	0.39	0.41	0.40	1000
class 3	0.36	0.33	0.35	1000
class 4	0.42	0.52	0.46	1000
class 5	0.43	0.42	0.43	1000
class 6	0.54	0.61	0.57	1000
class 7	0.59	0.56	0.57	1000
class 8	0.65	0.67	0.66	1000
class 9	0.56	0.55	0.56	1000

accuracy 0.52 10000

macro avg 0.52 0.52 0.52 10000

weighted avg 0.52 0.52 0.52 10000

دقت نهایی ۵۲ درصد است اگرچه برای کلاس‌های مختلف این دقต کمی بالا و پایین‌تر است.

که این نتیجه خیلی بد هم نیست. از مدل Rndom که دقتش ۱۰ درصد است به طور قابل توجهی بیشتر است. گرچه استفاده از مدل‌هایی مثل CNN خیلی بهتر بود چون از استخراج ویژگی‌های محلی که در تصاویر مهم است بهره می‌برد. در صورتی که در MLP اصلا جایگاه پیکسل‌ها مهم نیست و همان اول فلت می‌کنیم. برای همین هرچه قدرم با پارامترها بازی کنیم مدل MLP ذاتا نمی‌توانه دقت خیلی بالایی بدهد.

حال نوبت استفاده از مدل BeIT است. به طور خلاصه مدل Pre_train شده "microsoft/beit-" fine-tune "base-patch16-224" را برای پایه مدل استفاده می‌کنیم. سپس روی داده CIFAR-10 آن را می‌کنیم بعد هم نمودارهای خواسته شده سوال را گزارش می‌دهیم.

خب قدم اول نصب کتابخانه transormers[torch] است. که همانطور که گفتیم این فرآیندهای

گفته شده را با api هایی که در اختیارمان می‌zarه آسون‌تر می‌کنند. بعد از این کتابخونه ماژول‌های

BeitImageProcessor, BeitForImageClassification را وارد می‌کنیم.

برای آماده کردن تصویر برای دادن به مدل از ماژول `beitImageProcessor` استفاده می‌شود. که با تابع `map` آن را روی دیتاست اعمال می‌کنیم. این مرحله برای پیش پردازش تصاویر کافی است. توجه شود اگر بخوايد این مرحله را اجرا کنید کمی طولانیست پس صبور باشید!

حالا یک بخشی از این داده در حد 5000 داده از `training` و 2000 داده از `test` را به صورت تصادفی انتخاب می‌کنیم. علت این کار جلوگیری از سرریز رم و سرعت بخشیدن به آموزش است.(چاره دیگری نبود!)

حالا نوبت اینکه آرگیومنت‌های آموزش دادن را تعیین کنیم. سایز بج را ۸ در نظر گرفتیم. نرخ یادگیری را $6e-5$ رد نظر گرفتیم. لاگ‌ها رو هم برای هر ایپاک ذخیره می‌کنیم تا بتونیم به اینها در هر ایپاک دسترسی داشته باشیم. استراتژی ارزیابی رو هم برابر ایپاک قرار میدیم یعنی بعد هر ایپاک دقت و `loss` مدل نمایش داده شود. نکته مهم دیگر قرار دادن `True` برای `load_best_model_at_end` است که از اسمش مشخص است که پارامترهای مدل در بهترین دقت را ذخیره می‌کند.

در گام پایانی با ماژول `Trainer` مدل را آماده `train` می‌کنیم. به آن مدل `bert` که شده است را به عنوان مدل پایه میدیم. آرگیومنت‌های تعریف شده در بالا را هم به عنوان ورودی میدیم تا در آموزش لحاظ کند. دیتاست گه گفتم شامل 5000 داده `training` و 2000 داده `test` است(یعنی تقریباً ۱۰ درصد دیتاست) را هم می‌دهیم. برای متريک هم از خروجی مدل `argmax` می‌گيریم و با لیبل واقعی مقایسه می‌کنیم.

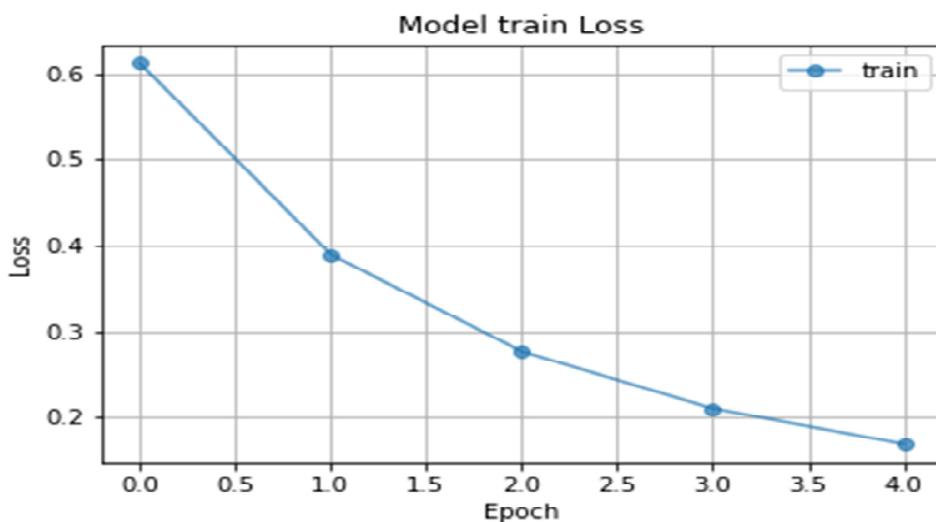
حالا نوبت آموزش مدل با دستور ساده `trainer.train()` است. که در حین اجرا مقادیر زیر بعد از هر ایپاک نمایش داده می‌شود. که ما خروجی نهایی را اينجا نمایش دادیم.(اجرا با GPU دو ساعت طول کشید).

Epoch	Training Loss	Validation Loss	Accuracy
1	0.502600	0.294116	0.924500
2	1.277000	0.281722	0.938500
3	0.002100	0.231993	0.958000
4	0.000100	0.163215	0.971000
5	0.000300	0.174979	0.963000

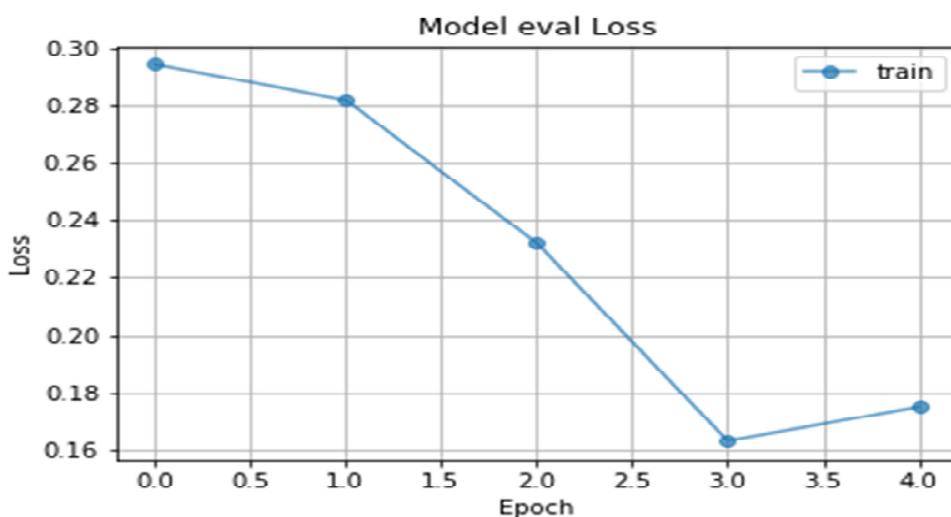
همانطور که مشاهده می‌کنید مدل در همان ایپاک اول به دقت بالایی رسیده است. و تنها بعد چند ایپاک به نظر `overfit` کرده است.(البته باید چند ایپاک دیگر هم اجرا می‌شد تا با قاطعیت نظر داد اما چون پارامترها نزدیک نود میلیون است و فقط 5000 داده دادیم قابل حدس است که بعد ایپاک

چهارم overfit رخ داده است. در کل این نتیجه به دلیل اینکه فقط نزدیک ۱۰ درصد داده را استفاده کردیم خیلی قابل اطمینان نیست. و چون هدف سوال یادگیری بود و در سخت افزار محدودیت داشتیم به همین نتیجه اکتفا می‌کنیم.

برای دسترسی به این داده‌ها تابع `loss` را از `state.log_history` فراخوانی می‌کنیم. و بعد با چند خط کد مقدار Loss در هر اپاک برای training و validation را استخراج می‌کنیم. و با کتابخونه `matplotlib` آن‌ها را نمایش می‌دهیم:



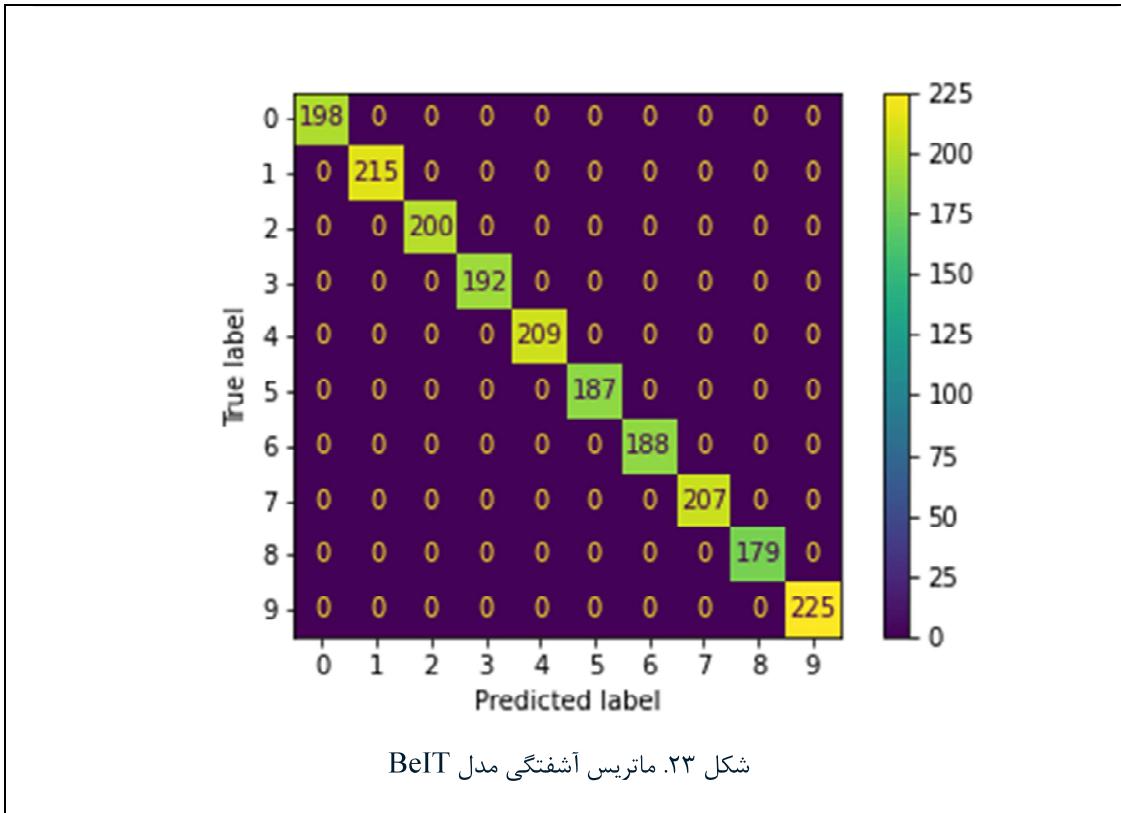
شکل ۲۱. نمودار training loss برای مدل BeIT



شکل ۲۲. نمودار validation loss برای مدل BeIT

خب اینجا هم overfit کردن را می‌توان راحت‌تر تشخیص داد.(البته احتمالا)

سپس با تابع predict از trainer برای داده validation پیش بینی می کنیم. که کلاس پیش بینی شده در label_ids قرار دارد. این مقادیر را همراه لیبل واقعی به تابع confusion_matrix می دهیم تا ماتریس آشفتگی بدست بیاید و با تابع confusionMatrixDisplay آن را به صورت نمودار نشان می دهیم:



در نهایت گزارش طبقه بندی را با کتابخونه classification_report و دادن خروجی ها نمایش می دهیم.

precision recall f1-score support

	precision	recall	f1-score	support
class 0	0.95	0.94	0.97	198
class 1	0.96	0.95	0.98	215
class 2	0.95	0.96	0.94	200
class 3	0.93	0.97	0.96	192
class 4	0.98	0.98	0.97	209
class 5	0.97	0.97	0.98	187
class 6	0.97	0.98	0.96	188

class 7	0.95	0.97	0.97	207
class 8	0.98	0.99	0.97	179
class 9	0.97	0.96	0.98	225
accuracy		0.97	2000	
macro avg	0.97	0.98	0.97	2000
weighted avg	0.97	0.97	0.97	2000

که دقت مدل ۹۷ درصد است. با اینکه برای هر کلاس این دقت فرق می‌کند اما همه تقریباً در همین حدود دقت می‌دهند. که این دقت بسیار بالا است که قدرت مدل BeIT در Image Classification را نشان می‌دهد.

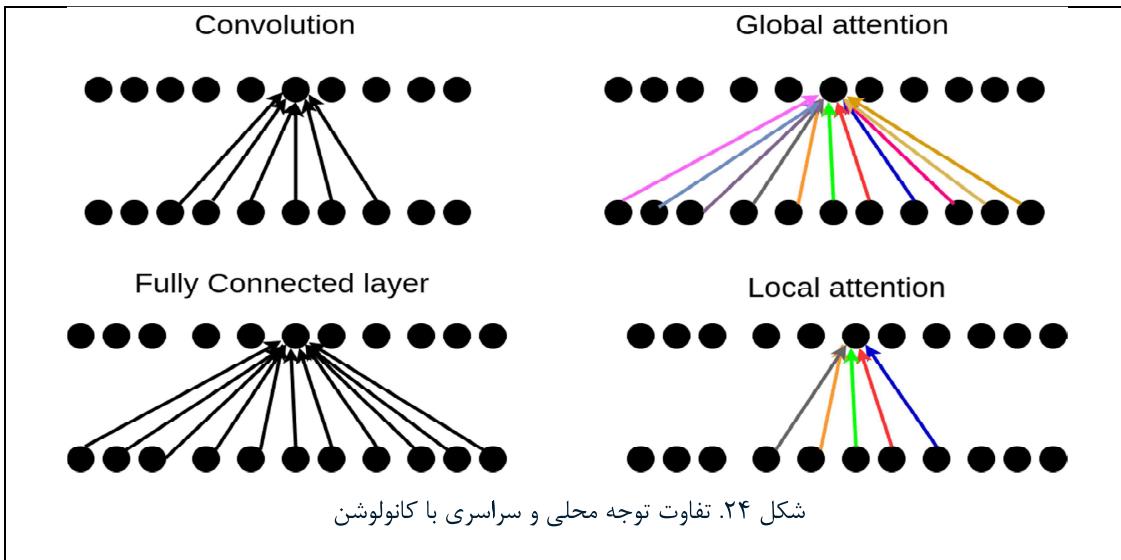
مقایسه دو مدل: همانطور که از نتایج مشخص است مدل BeIT ۴۵ درصد از مدل MLP بهتر است. این اختلاف به طور معناداری زیاد است. اولین دلیل که برای هر کسی قابل درک است تعداد پارامترها است. تعداد پارامترهای مدل MLP در حد چند هزار تاست در حالی که پارامترهای BeIT میلیونی است. دومین دلیل این است که MLP از موقعیت پیکسل‌ها کنار هم هیچ استفاده‌ای نمی‌برد. در حالی که می‌دانیم پیکسل‌ها کنار هم می‌توانند فیچرهای قدرتمندی را ایجاد کنند. در مدل BeIT درست است که تصویر فلت می‌شود اما با پوزیشن کانکت می‌شود و بعد به دیکودر داده می‌شود. و علاوه بر اون با توجه به اینکه از برت استفاده می‌کند. اهمیت هر پیکسل برای پیکسل‌های دیگر سنجیده می‌شود. که می‌توانه فیچرهای مهمی ایجاد کنه. البته چون این تعداد از مرتبه n به توان ۲ است اول تصاویر را به تکه‌های کوچکتر تقسیم می‌کنند بعد به مدل میدهند. تا تعداد پارامترها کاهش یابد.

۴-۲. پرسش‌ها

تشریحی:

۱. لایه کانولوشن شبیه مفهوم توجه عمل می‌کند. که اهمیت و رابطه هر پیکسل را نسبت به همسایگانش می‌سنجد. و مقادیر فیلتر همان ضرایب توجه هستند. حتی مثلاً میشه گفت مدل‌های -fully-attentional تعمیم یافته‌ی لایه کانولوشن هستند. که علاوه بر اینکه مثل لایه کانولوشن مقادیر فیلتر را یاد می‌گیرند، پترن کرنل‌ها رو هم باید یاد بگیرند که دیگه این در لایه کانولوشن مطرح نیست و در آن پترن کرنل فقط محلی است.

۲. اولاً تفاوت شبکه کانولوشنال با توجه سراسری این است که در کانولوشنال تعدادی از نورون‌های همسایه لایه قبل با نورون‌های لایه‌های بعد در ارتباط هستند. در باره تفاوت‌ش با توجه محلی هم این است که یادگیری در کانولوشنال کند است اما در توجه محلی وزن‌ها دائم در حال تغییر هستند.



صحیح و غلط:

۱. غلط. فقط از feedforwaeaf و multi-head attention و ... تشکیل شده است.
۲. درست است. در بخش انکودر چند انکور استک شدند که هر کدام کلمه مشخصی را اینکود کردند. در بخش دیکودر هم چند دیکودر استک شدند که وظیفه دیکود کردن کلمات را دارند.
۳. غلط. البته یکم سوال ابهام دارد. Multi head attention از چندتا head تشکیل شده است که هر کدام از آن‌ها یک بخش توجه و چند لایه تمام متصل تشکیل شده‌اند. (ابهام اینجاست که موازی به این لایه‌های تمام متصل اشاره داره یا به یک بخش توجه و چند لایه تمام متصل که همون head میشه؟ اگه اولی منظور باشه جواب غلط است ولی اگه منظور دومی باشد جواب درست است.)
۴. غلط. شبکه از کار نمی‌افتد بلکه کارایی آن در اکثر موارد کاهش می‌یابد. چون تعییه سازی پوزیشن به طور معناداری در اکثر موارد به یادگیری بازنمایی مفهومی از کلمه در جاهای مختلف کمک می‌کنه. البته طبق [مقاله‌ای](#) برای برخی مدل‌ها مثل multi-layer autoregressive self-attention حتی حذف پوزیشن باعث بهبود جزئی مدل هم شده است. چراکه اگر عمیق باشند به صورت خودکار اطلاعات مکانی را یاد می‌گیرند.