به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین امتیازی

نام و نام خانوادگی	مهسا همت‌پناه – محمد هادی بابالو
شماره دانشجویی	۸۱۰۱۹۹۵۸۴ - ۸۱۰۱۹۹۳۸۰
تاریخ ارسال گزارش	۱۴۰۲,۱۰.۲۷

فهرست

شکل‌ها

جدول ها

پاسخ ۲. LoRA

در این بخش می‌خواهیم با استفاده از مدل RoBERTa که مدلی برگرفته شده از معماری BERT است، یک تسک پردازش زبان طبیعی sequence classification بر روی مجموعه داده Quora Question Pairs را با استفاده از PyTorch و ابزارهای موجود انجام دهیم.

```
RobertaForSequenceClassification(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 1024, padding_idx=1)
      (position_embeddings): Embedding(514, 1024, padding_idx=1)
      (token_type_embeddings): Embedding(1, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-23): 24 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024, bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=1024, out_features=1024, bias=True)
              (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=1024, out_features=4096, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=4096, out_features=1024, bias=True)
            (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): RobertaClassificationHead(
    (dense): Linear(in_features=1024, out_features=1024, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=1024, out_features=2, bias=True)
  )
)
```

شکل ۱ - معماری کلی مدل **RoBERTa** استفاده شده

۲-۱. LoRA چگونه عمل می‌کند؟

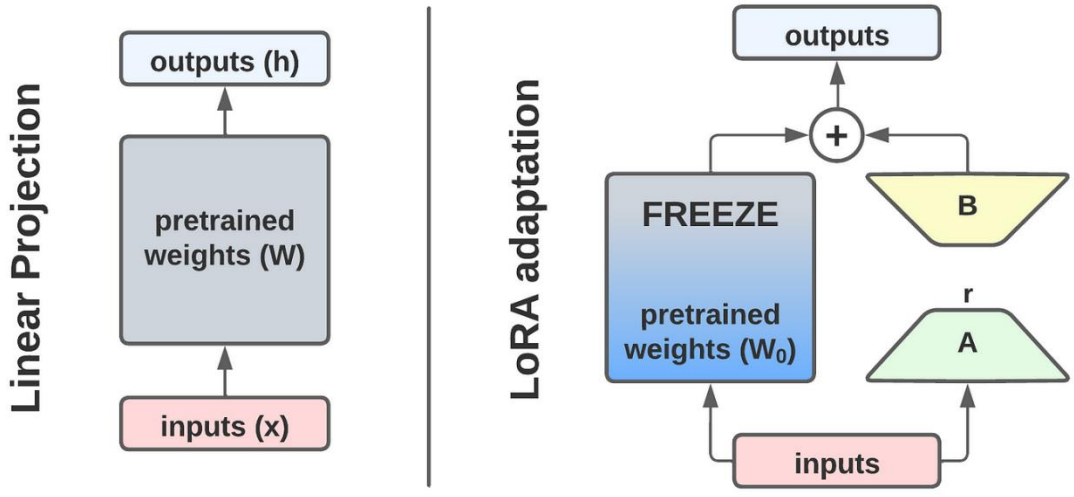
برای fine-tune کردن یک مدلی که از پیش آموزش دیده است، چندین کار می‌توانیم انجام دهیم. ساده‌ترین راه fine-tune کردن این است که تمام پارامترهای مدل با داده‌های جدید و برای task جدید آموزش ببینند. در این روش هم feature های سطح بالا و هم feature های سطح پایین ما درگیر تغییر

می‌شوند. این روش معمولا انعطاف بیشتری دارد و زمانی استفاده می‌شود که task ما با task مدل pre-trained مشابه هم باشند یا داده‌های ما توزیع مشابهی با داده‌های آموزش اصلی مدل داشته باشند.

روش دیگر آموزش فقط برخی از لایه‌ها و در واقع freeze کردن بقیه لایه‌های مدل است. در این روش معمولا لایه‌های اولیه مدل که نزدیک به ورودی هستند و feature های سطح بالاتر را تشخیص می‌دهند freeze می‌شوند و عملا کار feature extraction را برای ما انجام می‌دهند.

از لحاظ محاسباتی طبعا آموزش تمام پارامترهای مدل نیاز به هزینه و زمان بیشتری خواهد داشت. از طرفی اگر داده‌های ما کم باشد، خطر overfit شدن مدل در صورت آموزش تمام پارامترهای آن بیشتر است. اما در صورت مشابهت task ها و در صورت fine-tune کردن تمام پارامترهای مدل، احتمال اینکه نتیجه نهایی بهتری از مدل fine-tune شده بگیریم، بیشتر است.

LoRA یا همان Low-Rank Adaption روشی برای fine-tune کردن مدل‌های pre-trained با پارامترهای بسیار زیاد به صورت بهینه است. مبنای کار LoRA بر این اساس است که تنها بخش کوچکی از پارامترها نیاز به آپدیت شدن در هنگام fine-tune شدن مدل دارند و در نتیجه ماتریس آپدیت وزن‌ها در این فرایند باید low-rank باشد. در LoRA به جای آپدیت مستقیم پارامترهای مدل، تعداد کمی adaptive layer در میان مدل افزوده می‌شوند که تعداد پارامترهای بسیار بسیار کمتری نسبت به مدل اصلی دارندو در هنگام fine-tune کردن مدل، تنها این لایه‌های adaptive متعلق به LoRA درگیر آپدیت می‌شوند و پارامترهای اصلی مدل همه freeze شده باقی می‌مانند. تعداد کم پارامترهای این لایه‌های adaptive منجر به ماتریس آپدیت وزن low-rank و در نتیجه فرایند fine-tune بسیار بهینه‌تر و سریع‌تر می‌شود. همچنین از آنجا که اضافه شدن این لایه‌ها در جای جای مدل انجام می‌شود، این تغییرات محلی کوچک در مجموع می‌توانند عملکرد کلی مد نظر ای مدل را برای ما به ارمغان بیاورد.



شکل ۲ - عملکرد کلی روش **LoRA**

۲-۲. قرار است چه کاری را روی چه داده‌هایی انجام دهیم؟

در این پروژه ما از دیتاست Quora Question Pairs یا همان QQP استفاده می‌کنیم. این دیتاست شامل جفت سوالات و یک لیبل is_duplicate است که نشان می‌دهد آیا این دو سوال یک مفهوم را می‌رسانند یا خیر. همچنین به ازای سوالات استفاده شده در هر رکورد id آن‌ها هم وجود دارد که البته در این تسک نیازی به آن نداریم.

```
length of raw dataset: 404290
sample of raw dataset: {
                        'questions': {
                            'id': [1, 2],
                            'text': ['What is the step by step guide to invest in share market in india?',
                                    'What is the step by step guide to invest in share market?']},
                        'is_duplicate': False
                        }
```

شکل ۳ - نمونه داده موجود در دیتاست **QQP**

از آنجا که تسک ما sequence classification است، ابتدا باید با استفاده از tokenizer خود مدل RoBERTa جفت سوالات را tokenize کنیم و input_ids، attention_mask و token_type_ids متناظر با آن‌ها را به دست بیاوریم. سپس می‌توانیم این اطلاعات را به عنوان ورودی به RoBERTa بدهیم.

```
train dataset: Dataset({
    features: ['labels', 'input_ids', 'attention_mask'],
    num_rows: 1600
})
length of train dataset: 1600
sample of train dataset: {
        'labels': tensor(0),
        'input_ids': tensor([    0,  2264,    16,    24,     7,    28,    10, 21427,   116,     2,
                             2,  6179,   473,    24,   619,     7,    28,    10, 21427,   116,
                             2,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
                             1,     1,     1,     1,     1,     1,     1,    1]),
        'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
                             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                             0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                             0, 0, 0, 0, 0, 0, 0, 0])
    }
```

شکل ۴ - دیتاست پس از **tokenization**

در ابتدا این ورودی‌های tokenize شده در ReBERTa Embedding پردازش شده و سپس به بخش Encoder داده می‌شوند. نتایج این بخش هم به بخش نهایی یا همان output layers مدل می‌رود تا خروجی

را تولید کند. همچنین از آنجا که ما با تسک classification طرف هستیم، نیاز داریم که یک بخش classifier هم به انتهای مدل اضافه کنیم که شامل دو لایه linear و یک لایه dropout در میان آن‌ها است تا در نهایت خروجی کل مدل ما که همان label مد نظر ما است را تولید کند.

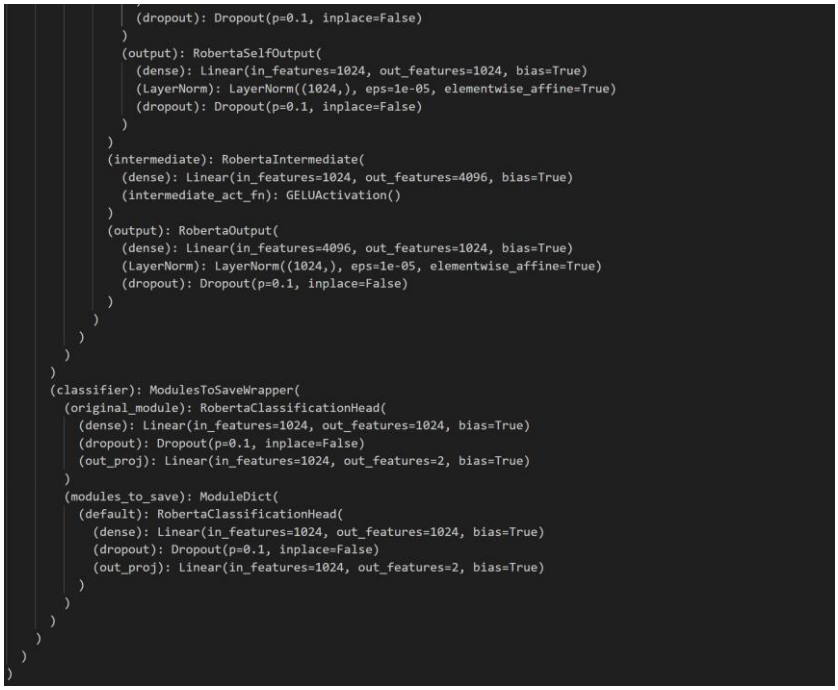
به طور خاص در هنگام اعمال LoRA روی RoBERTa، لایه‌های adaptive بعد از هر لایه multi-head attention و همچنین هر لایه feed-forward قرار می‌گیرند. هر adaptive module شامل یک لایه برای کاهش dimension، یک لایه افزودن غیرخطی بودن به مدل و سپس یک لایه بازیابی dimension است. پارامترهای استفاده شده در این adaptive layer ها تنها پارامترهایی هستند که در حین fine-tune کردن مدل آپدیت می‌شوند که این در حدود یک درصد کل پارامترهای مدل است. این کار باعث می‌شود که با آپدیت‌کردن وزن‌های بسیار کمتری ولی در جاهای مختلف مدل، بتوانیم به طور بسیار بهینه‌تری و بدون تغییر مستقیم وزن‌های مدل اصلی، کل مدل را برای تسک خودمان fine-tune کنیم.

۲-۳. آموزش مدل

همانطور که می‌بینید، پس از اعمال LoRA، معماری مدل ما تغییر کرده است و لایه‌های adaptive مربوط به LoRA به مدل افزوده شده‌اند.

```
PeftModelForSequenceClassification(
  (base_model): LoraModel(
    (model): RobertaForSequenceClassification(
      (roberta): RobertaModel(
        (embeddings): RobertaEmbeddings(
          (word_embeddings): Embedding(50265, 1024, padding_idx=1)
          (position_embeddings): Embedding(514, 1024, padding_idx=1)
          (token_type_embeddings): Embedding(1, 1024)
          (LayerNorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (encoder): RobertaEncoder(
          (layer): ModuleList(
            (0-23): 24 x RobertaLayer(
              (attention): RobertaAttention(
                (self): RobertaSelfAttention(
                  (query): lora.Linear(
                    (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.05, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=1024, out_features=8, bias=False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=8, out_features=1024, bias=False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                  )
                  (key): Linear(in_features=1024, out_features=1024, bias=True)
                  (value): lora.Linear(
                    (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.05, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=1024, out_features=8, bias=False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=8, out_features=1024, bias=False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                  )
                  (dropout): Dropout(p=0.1, inplace=False)
```

شکل ۵ - بخش اول معماری **RoBERTa** با اعمال **LoRA**



شکل ۶ - بخش دوم معماری **RoBERTa** با اعمال **LoRA**

در این بخش به دلیل محدودیت‌های پردازشی موجود، آموزش دو مدل تنها روی بخش کوچکی از دیتاست اصلی که شامل بیش از ۴۰۰۰۰۰ رکورد است انجام شد. فرایند fine-tune مدل به صورت عادی و با استفاده از LoRA در شکل‌های زیر قابل مشاهده است.

[750/750 20:43, Epoch 15/15]

Step	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
33	No log	0.677457	0.580000	0.000000	0.000000	0.000000
66	No log	0.594012	0.695000	0.676923	0.523810	0.590604
99	No log	0.509240	0.750000	0.663462	0.821429	0.734043
132	No log	0.454956	0.780000	0.696078	0.845238	0.763441
165	No log	0.431628	0.792500	0.709360	0.857143	0.776280
198	No log	0.412000	0.820000	0.807692	0.750000	0.777778
231	No log	0.500872	0.820000	0.730769	0.904762	0.808511
264	No log	0.582330	0.825000	0.760638	0.851190	0.803371
297	No log	0.556004	0.845000	0.804598	0.833333	0.818713
330	No log	0.634422	0.820000	0.800000	0.761905	0.780488
363	No log	0.762472	0.842500	0.807018	0.821429	0.814159
396	No log	1.006132	0.817500	0.774566	0.797619	0.785924
429	No log	1.139539	0.820000	0.775862	0.803571	0.789474
462	No log	1.173148	0.820000	0.763736	0.827381	0.794286
495	No log	1.189623	0.820000	0.789157	0.779762	0.784431
528	0.246200	1.250234	0.815000	0.752688	0.833333	0.790960
561	0.246200	1.237660	0.832500	0.785311	0.827381	0.805797
594	0.246200	1.261419	0.822500	0.786982	0.791667	0.789318
627	0.246200	1.275833	0.830000	0.768817	0.851190	0.807910
660	0.246200	1.279803	0.830000	0.784091	0.821429	0.802326
693	0.246200	1.313250	0.822500	0.770950	0.821429	0.795389
726	0.246200	1.316469	0.827500	0.779661	0.821429	0.800000

شکل ۷ - فرایند **fine-tune** کردن عادی مدل

[750/750 15:54, Epoch 15/15]

Step	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1
33	No log	0.658162	0.620000	0.000000	0.000000	0.000000
66	No log	0.619732	0.620000	0.000000	0.000000	0.000000
99	No log	0.498071	0.752500	0.646409	0.769737	0.702703
132	No log	0.462018	0.790000	0.761538	0.651316	0.702128
165	No log	0.505541	0.777500	0.641256	0.940789	0.762667
198	No log	0.474529	0.787500	0.663415	0.894737	0.761905
231	No log	0.474137	0.795000	0.743056	0.703947	0.722973
264	No log	0.536153	0.812500	0.834783	0.631579	0.719101
297	No log	0.459422	0.782500	0.675676	0.822368	0.741840
330	No log	0.582590	0.805000	0.824561	0.618421	0.706767
363	No log	0.484421	0.825000	0.744048	0.822368	0.781250
396	No log	0.647801	0.827500	0.821705	0.697368	0.754448
429	No log	0.628175	0.817500	0.713514	0.868421	0.783383
462	No log	0.499374	0.822500	0.751553	0.796053	0.773163
495	No log	0.833487	0.830000	0.813433	0.717105	0.762238
528	0.374100	0.695551	0.815000	0.746835	0.776316	0.761290
561	0.374100	0.691672	0.825000	0.773333	0.763158	0.768212
594	0.374100	0.882975	0.830000	0.783784	0.763158	0.773333
627	0.374100	0.907536	0.827500	0.782313	0.756579	0.769231
660	0.374100	1.015270	0.825000	0.766234	0.776316	0.771242
693	0.374100	1.144618	0.825000	0.797101	0.723684	0.758621
726	0.374100	1.164353	0.820000	0.770270	0.750000	0.760000

شکل ۸ - فرایند **fine-tune** کردن مدل با استفاده از **LoRA**

همانطور که دیده می‌شود، در هر دو حالت در epoch های ابتدایی مدل به یک local optimum می‌رسد و مقداری را در همانجا باقی می‌ماند. این موضوع به دلیل unbalance بودن توزیع داده‌های ما و بیشتر بودن تعداد داده‌های با لیبل is_duplicated=False (تقریبا به نسبت ۶۰ به ۴۰) در دیتاست است که البته با تغییر learning rate و استفاده از optimizer مناسب، وزن‌دار کردن loss استفاده شده و یا متوازن کردن داده‌های آموزش، می‌توان این مشکل را حل کرد.

۲-۴. چرا LoRA؟

همانطور که در شکل‌های مربوط به معماری مدل‌ها می‌بینیم، تفاوت مدل RoBERTa قبل و بعد از اعمال LoRA صرفا در لایه‌های adaptive اضافه شده توسط LoRA در میان لایه‌های خود مدل RoBERTa است. تعداد پارامترهای مدل‌ها در شکل‌های زیر قابل مشاهده هستند.

```
Number of parameters: 355361794
Number of learnable parameters: 355361794
```

شکل ۹ - تعداد پارامترهای مدل **RoBERTa-Large**

```
Number of parameters: 357199876
Number of learnable parameters: 1838082
Ratio of learnable parameters: 0.51%
```

شکل ۱۰ - تعداد پارامترهای مدل با استفاده از **LoRA**

همانطور که دیده می‌شود، با استفاده از LoRA تعداد کمی پارامتر به کل مدل افزوده می‌شود ولی صرفا همان پارامترها در فرایند fine-tuning شرکت می‌کنند و پارامترهای اصلی مدل که تعداد بسیار بسیار بیشتری دارند freeze شده و آپدیت نمی‌شوند. همین اتفاق باعث می‌شود که مدت زمان صرف شده جهت fine-tune کردن مدل مخصوصا در صورت بزرگ بودن مدل استفاده شده و زیاد بودن داده‌های مورد استفاده در فرایند fine-tuning به طرز قابل توجهی کاهش پیدا کند.

نتایج و metric های مختلف classification در حین فرایند fine-tuning در شکل‌های ۷ و ۸ برای fine-tuning تمام پارامترها و همچنین fine-tuning با استفاده از LoRA نمایش داده شده‌اند. نتایج نهایی مدل‌ها هم در شکل‌های زیر قابل مشاهده هستند.

```
Loss:          1.3162
Accuracy:      83.00
Precision:     77.78
Recall:        83.33
F1:            80.46
```

شکل ۱۱ - نتایج نهایی مدل **fine-tune** شده

```
Loss:          1.1396
Accuracy:      82.50
Precision:     76.97
Recall:        76.97
F1:            76.97
```

شکل ۱۲ - نتایج نهایی مدل **fine-tune** شده با **LoRA**

همانطور که می‌بینیم با اینکه با استفاده از LoRA پارامترهای بسیار کمتری در آموزش مشارکت دارند، نتایج مدل با استفاده از LoRA بسیار نزدیک به عملکرد مدل اصلی است و ما شاهد افت عملکرد قابل توجهی در نتایج مدل نیستیم و ما هر دو روش ما به دقت خوب و قابل قبولی روی این دیتاست می‌رسیم.

همانطور که توضیح داده شد، LoRA با صرفا freeze کردن بعضی لایه‌های مدل اصلی تفاوت دارد و روشی متفاوت است.

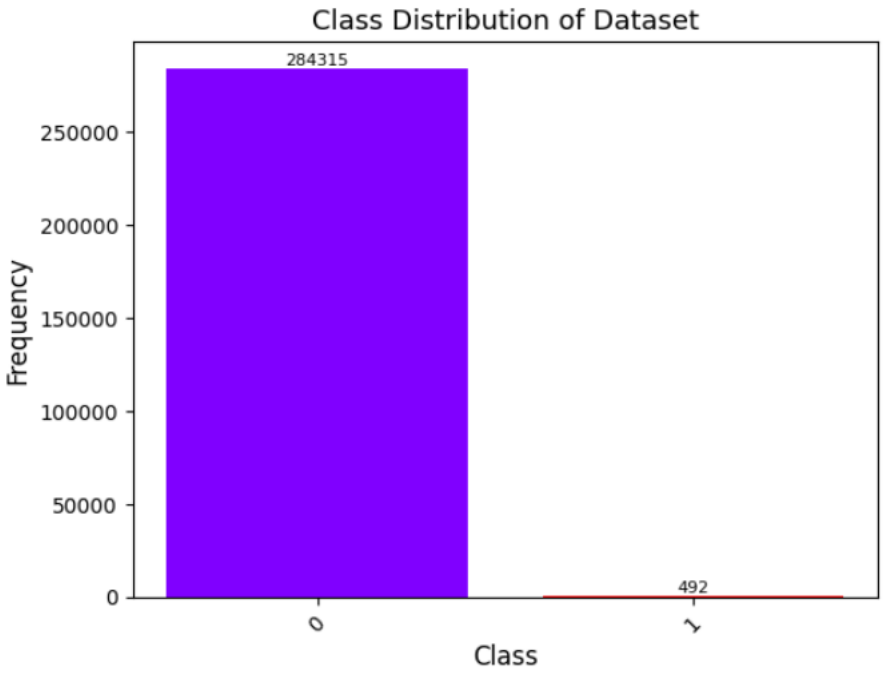
برای استفاده از مدل روی تسک‌های متفاوت، در صورت fine-tune کردن کل مدل نیاز است که برای هر تسک به صورت جداگانه مدل‌ها fine-tune شوند و تمام مدل‌های با حجم زیاد را مخصوص هر تسک ذخیره و استفاده کنیم و هیچ پارامتری را بین این مدل‌های مختلف نمی‌توانیم به اشتراک بگذاریم. در صورت freeze کردن تعدادی از لایه‌ها هم باز نیاز است که یک مدل برای هر تسک به صورت جدا استفاده کنیم ولی در صورت مشابهت لایه‌هایی که در حین fine-tuning توسط ما freeze شده‌اند، پارامترهای این لایه‌ها می‌توانند بین مدل‌های مختلف اشتراک گذاشته شوند. در صورت استفاده از LoRA اما، تنها adaptive layer های افزوده شده توسط LoRA برای هر تسک آموزش داده می‌شوند و خود مدل کاملا دست‌نخورده باقی می‌ماند. همچنین از آنجا که این پارامترها به نسبت پارامترهای مدل بسیار بسیار کمتر هستند، ذخیره‌سازی آن‌ها بسیار ساده‌تر است. در نهایت هم ما می‌توانیم یک مدل RoBERTa داشته باشیم که تعدادی task-specific adapter دارد و برای همه تسک‌های fine-tune شده می‌تواند استفاده شود. این کار بهینه‌تر شدن مدل و حداکثرسازی اشتراک گذاری پارامترها بین مدل‌های مربوط به تسک‌های مختلف می‌شود و duplication ای نیز نداریم. در نتیجه LoRA به ما قابلیت رسیدگی به چندین downstream task متفاوت به صورت همزمان با استفاده از LoRA adapters را می‌دهد.

پاسخ ۳ . تشخیص تقلب

۳-۱. آشنایی با دیتاست

با توجه به اینکه کد این سوال داخل kaggle زده شده است، داده را بطور مستقیم از همین سایت داخل نوت بوک import می کنیم. ابتدا با دستوراتی مانند ()info. و ()describe. اطلاعات کلی دیتاست را مشاهده می کنیم. در این دیتاست تراکنش ها با ۲۹ ویژگی توصیف می شوند (V1، V2، ...، V28، و Amount) و برچسب کلاس ها برای تراکنش های متقلبانه برابر با "۱" و برای تراکنش عادی برابر با "۰" هستند.

۱. نمودار هیستوگرام کلاسهای داده را رسم کرده و آن را تحلیل کنید.



شکل ۱۳ - نمودار توزیع داده در کلاس ها

شکل ۱ نشان دهنده هیستوگرام داده ها در کلاس ها است. همانطور که قابل مشاهده است داده ها در کلاس هاس ۰ و ۱ به شدت نامتوازن هستند. تنها ۴۹۲ تراکنش از مجموع ۲۸۴۸۰۷ تراکنش تقلبی هستند که به معنی ۰,۱۷۲ درصد از کل معاملات است و این نشان دهنده عدم توازن بالای مجموعه داده است.

۲. اگر مدل را با این داده ها آموزش دهیم چه مشکلی به وجود می آید؟ توضیح دهید.

آموزش مدل با داده های نامتوازن، مانند مجموعه داده های تشخیص تقلب در کارت اعتباری که در این سوال از آن استفاده شده است، مشکلات زیر را خواهد داشت:

- Biased model: مدل نسبت به کلاس اکثریت (معاملات عادی) سوگیری دارد زیرا نمونه های بیشتری از آن کلاس را در طول آموزش مشاهده می کند.
- تعمیم ضعیف: مدل ممکن است تعمیم پذیری ضعیفی به کلاس اقلیت داشته باشد (معاملات متقلبانه) زیرا نمونه های متنوعی از کلاس تقلب وجود ندارد.
- False Negatives بالا: مدل ممکن است تمایل بیشتری به پیش بینی کلاس اکثریت داشته باشد که منجر به تعداد بالای منفی کاذب (FN) برای کلاس اقلیت می شود. در تشخیص تقلب، این بدین معناست که مدل ممکن است بسیاری از تراکنش‌های جعلی را از دست بدهد و بیشتر داده ها را در کلاس اکثریت و تراکنش های عادی تشخیص دهد.
- معیارهای ارزیابی گمراه کننده: دقت یا accuracy به تنهایی می تواند یک معیار گمراه کننده در داده های نامتوازن باشد زیرا مدلی که همه نمونه ها را در کلاس اکثریت پیش بینی کند همچنان می تواند به دقت بالایی دست یابد.

۳-۲. پیاده سازی معماری مقاله

برای پیش پردازش داده ها ابتدا داده های تکراری یا duplicate رو پیدا و حذف کردیم. همچنین در این دیتاست داده null نداریم پس نیازی به رسیدگی به آن ها نداریم. در ادامه طبق درصدهای آورده شده در مقاله مربوطه، داده ها را به سه بخش تقسیم کردیم. ۶۰ درصد داده های آموزشی، ۲۰ درصد داده‌های ارزیابی و ۲۰ درصد داده‌های آزمایشی. با استفاده از متد train_test_split در کتابخانه sklearn، مجموعه داده ابتدا به مجموعه داده های آموزشی تصادفی (۸۰٪) و آزمایشی (۲۰٪) تقسیم شد و در ادامه ۲۵٪ از مجموعه داده آموزشی به عنوان مجموعه داده ارزیابی در نظر گرفته شد که معادل ۲۰٪ از مجموعه داده ها است.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
self.model.fit(X, y, validation_split=0.25, epochs=epoch)
```

شکل ۱۴ – کد مربوط به تقسیم بندی داده ها

در ادامه نیز از متد StandardScaler که در مقاله مربوطه نیز از آن استفاده شده است برای نرمالایز کردن داده ها استفاده کرده ایم.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

شکل ۱۵ – کد مربوط به نرمالایز کردن داده ها

معماری پیشنهاد شده دارای سه لایه اصلی است که عبارتند از:

لایه convolution ، لایه pooling و لایه fully connected.

لایه convolution : بلوک اصلی مدل و جایی است که اکثر محاسبات و استخراج ویژگی انجام می شود که شامل فرآیندهای خطی و غیرخطی است، یعنی عملیات کانولوشن و تابع فعال سازی relu.

لایه pooling : این یک عملیات down sampling را انجام می دهد و اندازه پارامترهای قابل یادگیری متوالی را کاهش می دهد.

لایه fully connected : خروجی لایه کانولوشن نهایی یا لایه pooling به یک آرایه یک بعدی تبدیل می‌شود و به یک یا چند لایه fully connected که dense نیز نامیده می‌شوند با تابع فعال سازی relu پیوند داده می‌شود. در این لایه، برای دسته بندی بعد از دومین لایه dense آورده شده در معماری یک لایه جدید با ۲ خروجی و تابع هزینه softmax گذاشته شده است تا احتمال هر کلاس محاسبه شود.

جدول ۱ – معماری مدل پیشنهاد شده

Layer	Description
1	Conv1D Layer (filters = 32, kernel_size = 2, activation = relu)
2	BatchNormalization
3	MaxPool1D (pool_size = 2)
4	Dropout (rate = 0.2)
5	Conv1D Layer (filters = 64, kernel_size = 2, activation = relu)
6	BatchNormalization
7	MaxPool1D (pool_size = 2)
8	Dropout (rate = 0.5)
9	Flatten
10	Dense (units = 64, activation = relu)
11	Dropout (rate = 0.5)
12	Dense (units = 64, activation = relu)

پارامتر های یادگیری در جدول زیر ذکر شده است. این پارامتر ها همانند مقادیر داخل مقاله مقدار داده شده اند.

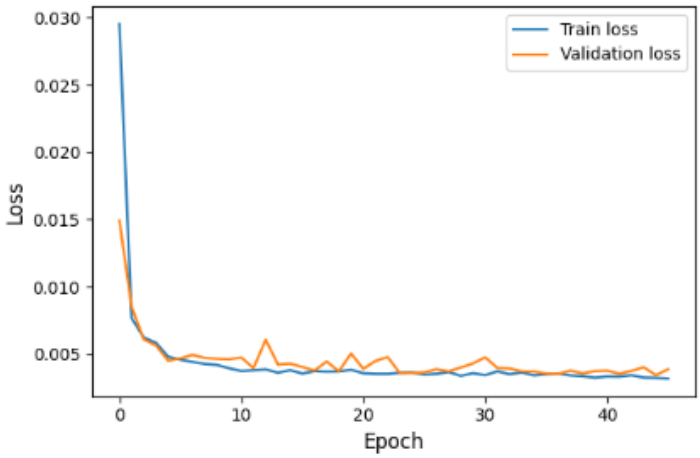
جدول ۲ – پارامتر های مدل

پارامتر	مقدار
Learning Rate	**0.0001**
# Epochs	**46**
Optimizer	**Adam**
Loss Function	**binary_crossentropy**
Metrics	**Accuracy**

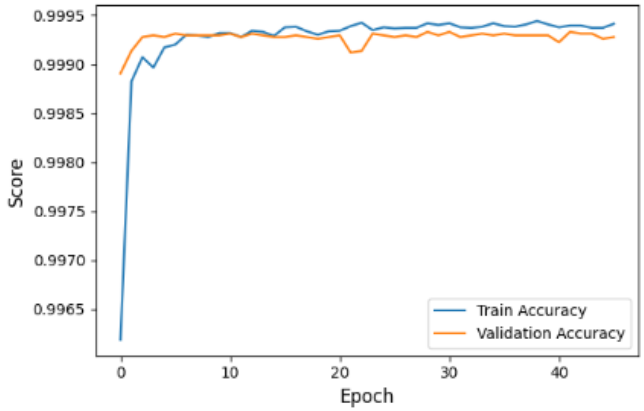
۱. نمودار loss و accuracy را برای داده های آموزش و ارزیابی رسم کنید و آن را تحلیل کنید. توضیح دهید که آیا مدل overfit شده است یا نه؟

مدل آموزش داده شده روی داده های نامتوازن را مدل ۱ در نظر گرفته ایم.

نمودار های دقت و loss داده های آموزش و ارزیابی در طول ایپاک ها مقادیر نزدیک به همی دارند. هم در داده های آموزش و هم ارزیابی دقت با شیب تندی افزایش می یابد و در همان ایپاک های اول به بیشترین میزان خود میرسد. مقدار loss نیز هم در داده های آموزش و هم ارزیابی با شیب تندی کاهش می یابد و در همان ایپاک های اول به کمترین میزان خود میرسد و مدل رسما آموزش چندانی در ایپاک های بعدی نمی بیند. آموزش مدل بر روی داده های نامتوازن باعث biased prediction می شود و مدل تمایل دارد تا تمام داده ها را در کلاس اکثریت دسته بندی کند. با توجه به این موضوع دقت برای داده های ارزیابی نیز زیاد می شود زیرا در این مجموعه نیز تعداد داده های کلاس اقلیت در مقابل تعداد داده های کلاس اکثریت، ناچیز است. پس مدل برای متریک دقت و loss برای هر دو مجموعه داده آموزش و ارزیابی خوب عمل می کند و overfit نمی شود.



شکل ۱۶ – نمودار **loss** در هر ایپاک مدل ۱



شکل ۱۷ – نمودار دقت در هر ایپاک مدل ۱

۲. پیشبینی مدل برای داده های تست را بدست آورید و نمودار heatmap را برای confusion matrix رسم کنید. همچنین معیارهای precision،recall و fl-score را گزارش کنید.

در قسمت قبل در مشکلات آموزش مدل بر روی داده های نامتوازن به این مورد اشاره کردیم که دقت در این داده ها معیار خوبی نیست چون اگر تمام داده ها در کلاس اکثریت پیش بینی شوند به دقت بالا می رسیم. در نتیجه معیار های دیگری مانند precision،recall و fl-score برای این کار بهتر هستند زیرا این معیار ها به تعداد پیش های انجام شده در کلاس اقلیت نیز حساس هستند. بدین دلیل با وجود دقت بالا نزدیک به 0.999 معیار های دیگر پایین هستند. مقدار برای f1-score , precision و recall در کلاس اقلیت به ترتیب 0.88 ، 0.75 و 0.66 است.

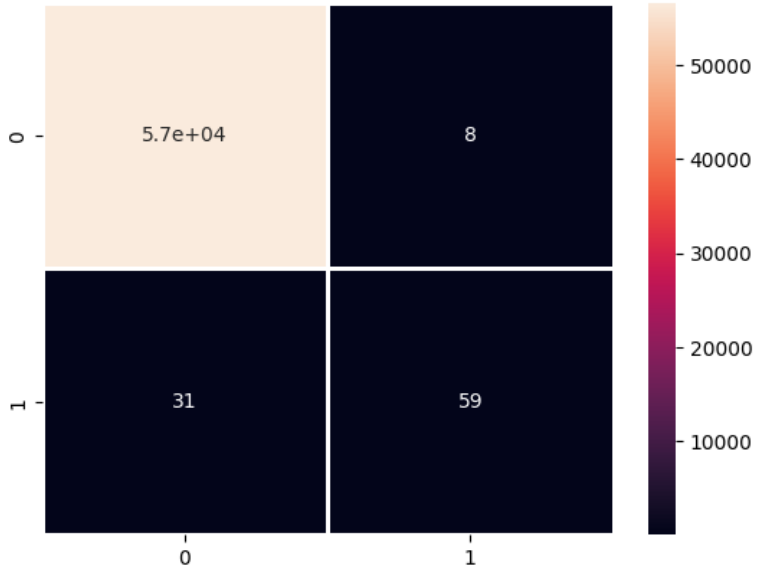
جدول ۳ – جدول **classification report** داده تست مدل ۱

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56656
1	0.88	0.66	0.75	90
accuracy			1.00	56746
macro avg	0.94	0.83	0.88	56746
weighted avg	1.00	1.00	1.00	56746

مقدار متریک ها برای مدل ۱ در زیر گزارش شده است.

Accuracy: 0.9993
Overall Precision: 0.8806
Recall: 0.6556
F1-score: 0.7516

در نمودار heatmap را برای confusion matrix نیز می توان مشاهده کرد که از بین 90 داده تست در کلاس ۱ تنها 59 نمونه درست پیش بینی شده است 31 مورد اشتباه در کلاس 0 پیش بینی شده است. بیشتر داده ها یعنی 56679 مورد که 99.88 درصد کل داده های تست می شود در کلاس تراکنش های عادی یا 0 دسته بندی شده اند. بطور کلی مدل تمایل دارد همه داده ها را در کلاس 0 پیش بینی کند.



شکل ۱۸ – نمودار heatmap را برای confusion matrix مدل ۱

۳. توضیح دهید که آیا accuracy میتواند به تنهایی عملکرد مدل را به خوبی نمایش دهد؟

مدلی که بر روی داده های نامتوازن آموزش داده شود در عمل در اکثر مواقع نمونه‌ها را به کلاس اکثریت دسته‌بندی خواهد کرد. همانطور که در قسمت های قبل توضیح داده شد، در دیتاست این سوال، تنها ۴۹۲ تراکنش از مجموع ۲۸۴۸۰۷ تراکنش تقلبی هستند که به معنی 0.172 درصد از کل معاملات است و این نشان دهنده عدم تعادل بالای مجموعه داده است. در این صورت اگر مدل همه داده را در کلاس معالات عادی که کلاس اکثریت است، پیش بینی کند با توجه به تعداد ناچیز معاملات تقلبی، به دقت بسیار بالایی میرسد. اما در این حالت دقت دارد توزیع داده‌ها در کلاس ها را نشان میدهد تا اینکه مدل را ارزیابی کند.

معادله دقت به شکل زیر است و زمانی که تمام داده ها در کلاس اکثریت پیش بینی شوند مقدار TN و FP صفر می شود. در نتیجه در این حالت نسبت داده های کلاس اکثریت را محاسبه می کنیم که 0.99 درصد است.

$$Accuracy = \frac{TP + FN}{TP + TN + FN + FP} \xrightarrow{TN\ .FP=0} \frac{TP}{TP + FN}$$

۳-۳. نمونه برداری

۱. این روش را توضیح دهید و مزایا و معایب آن را نسبت به روشهای دیگر نمونه برداری بیان کنید.

نمونه گیریADASYN یک روش برای مدیریت داده های نامتوازن در مجموعه داده های یادگیری ماشین است. تمرکز آن بر تولید نمونه های مصنوعی برای کلاس اقلیت است تا توزیع کلاس ها را متوازن کند. این روش برای تولید داده صرفا داده های کلاس اقلیت رو کپی نمی کند و به تولید داده هایی که یادگیری آن ها دشوارتر است، توجه می کند. در حقیقت داده های دشوار برای یادگیری را با استفاده از محاسبه k-nearest neighbours پیدا می کند. داده هایی در کلاس اقلیت برای یادگیری دشوارتر هستند و نمونه هایی بیشتری از آن ها باید تولید شود که در همسایگی خود دارای نمونه‌های کلاس اکثریت بیشتری هستند. این کار به الگوریتم ADASYN ماهیت adaptive بودن می دهد.

- مزایا

- نمونه برداری adaptive: ADASYN با local density نمونه های کلاس اقلیت سازگار می شود و روی تولید نمونه های مصنوعی برای موارد چالش برانگیزتر تمرکز می کند.

- جلوگیری از Overfit: با تولید نمونه های مصنوعی بر اساس امتیازهای اهمیت، ADASYN قصد دارد از Overfit به نمونه های noisy در کلاس اقلیت جلوگیری کند.

- معایب

پیچیدگی محاسباتی: با توجه به اینکه در این روش k-nearest neighbours و امتیازهای اهمیت را برای نمونه ها محاسبه می کنیم، این روش می تواند از نظر محاسباتی برای مجموعه داده های بزرگ بار زیادی را تحمیل کند.

حساسیت به نویز: این روش ممکن است به نویز در مجموعه داده حساس باشد، زیرا بر density distribution نمونه ها متکی است.

- مقایسه با سایر روش های نمونه گیری:

Oversampling/Undersampling تصادفی: این روش ها ممکن است دشواری یادگیری نمونه های کلاس اقلیت را در نظر نگیرند و overfit کنند. آنها در مقایسه با ADASYN سازگاری کمتری دارند.

SMOTE: این روش نمونه های مصنوعی را به طور یکنواخت و uniform در کلاس اقلیت تولید می کند و دشواری نمونه های یادگیری را در نظر گرفتن نمی گیرد. ADASYN این محدودیت را با تطبیق با local density برطرف می کند.

به طور خلاصه، ADASYN یک رویکرد adaptive برای تولید نمونه مصنوعی ارائه می‌کند و تلاش می‌کند بر روی نمونه‌های چالش برانگیز تمرکز کند. با این حال، پیچیدگی محاسباتی و حساسیت آن به نویز باید در هنگام انتخاب روش نمونه گیری بر اساس ویژگی های مجموعه داده در نظر گرفته شود.

۲. این روش را پیاده سازی کنید و به طور مختصر روش خود را توضیح دهید. توجه کنید که در این قسمت نمیتوانید از کتابخانه های آماده استفاده کنید.

- محاسبه نسبت عدم توازن و تعداد نمونه مصنوعی: نسبت عدم توازن به عنوان نسبت تعداد نمونه های کلاس اکثریت به تعداد نمونه های کلاس اقلیت تعریف می شود.

```
# Separate minority and majority class samples
minority_samples = X[y == 1]
majority_samples = X[y == 0]
len_minority_samples = len(minority_samples)
len_majority_samples  = len(majority_samples)

# Calculate imbalance ratio
imbalance_ratio = len_majority_samples / len_minority_samples

# Calculate number of synthetic samples to generate
num_synthetic_samples = (len_majority_samples - len_minority_samples) * self.ratio
```

شکل ۱۹ – کد محاسبه نسبت عدم توازن

- محاسبه نمرات اهمیت یا نسبت توزیع: برای هر نمونه کلاس اقلیت، یک امتیاز اهمیت یا نسبت توزیع محاسبه می شود. این امتیاز نشان دهنده دشواری یادگیری مثال بر اساس توزیع چگالی k-nearest neighbours آن در فضای ویژگی است که در این پیاده سازی k برابر با ۵ در نظر گرفته شده است. نمونه هایی با تراکم کمتر امتیازهای اهمیت بالاتری دریافت می کنند. توزیع چگالی هر نمونه اقلیت از تقسیم تعداد نمونه های کلاس اکثریت در همسایه های k-nearest neighbours آن نمونه بر تعداد تمام همسایه ها آن نمونه یعنی k محاسبه می شود.

```
for sample in minority_samples:
    xi = sample.reshape(1, -1)
    indices = neighbors.kneighbors(xi, n_neighbors=self.n_neighbors, return_distance=False)[0]
    ratio = len([i for i in indices if y[i] != 1]) / self.n_neighbors
    ratios.append(ratio)
```

شکل ۲۰ – کد محاسبه نمرات اهمیت یا نسبت توزیع

- نرمالایز کردن نمرات اهمیت یا نسبت توزیع

```
ratios = np.array(ratios)
ratios = ratios / np.sum(ratios)
```

شکل ۲۱ - نرمالایز کردن نمرات اهمیت یا نسبت توزیع

- تولید نمونه های مصنوعی: نمونه های مصنوعی برای هر نمونه کلاس اقلیت تولید می شود. تعداد نمونه های مصنوعی برای تولید متناسب با امتیاز اهمیت است. نمونه های مصنوعی بیشتری برای نمونه هایی ایجاد می شود که یادگیری آنها سخت تر است. داده جدید از ترکیب

یک مقدار رندوم و اختلاف فاصله نمونه مورد نظر و یک نمونه از کلاس اقلیت در همسایگی همان نمونه مورد نظر بدست می آید. در این پیاده سازی نمونه های مصنوعی بر اساس ترکیب خطی دو نمونه از کلاس اقلیت در همسایگی هم ساخته می شوند.

```
for i, sample in enumerate(minority_samples):
    # Determine the number of synthetic samples to generate for this sample
    num = int(ratios[i] * num_synthetic_samples)

    for _ in range(num):
        neighbor = X[np.random.choice(Minority_per_samples[i])]
        diff = neighbor - sample
        synthetic_sample = sample + np.random.random() * diff
        self.synthetic_samples.append(synthetic_sample)
```

شکل ۲۲ - تولید نمونه های مصنوعی

- ترکیب نمونه های اصلی و مصنوعی: نمونه‌های کلاس اقلیت اصلی با نمونه‌های مصنوعی جدید تولید شده ترکیب و shuffle می‌شوند، که منجر به یک مجموعه داده با توزیع کلاس متوازن تر می‌شود.

```
resampled_X = np.vstack((X, np.array(self.synthetic_samples)))
resampled_y = np.hstack((y, np.ones(len(self.synthetic_samples))))
shuffle_indices = np.random.permutation(len(resampled_X))
resampled_X = resampled_X[shuffle_indices]
resampled_y = resampled_y[shuffle_indices]
return resampled_X, resampled_y
```

شکل ۲۳ - ترکیب نمونه های اصلی و مصنوعی

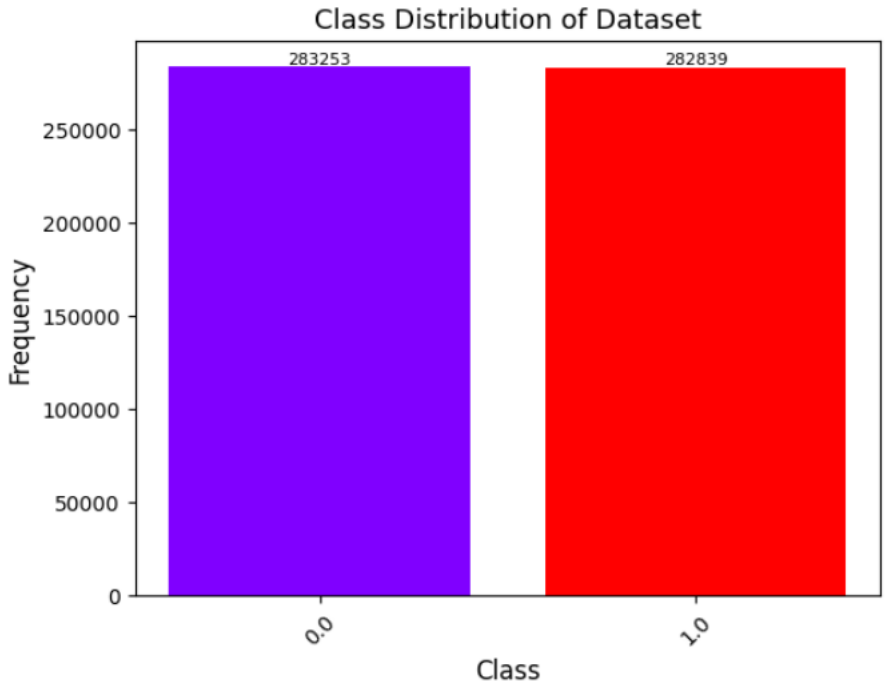
۳. توضیح دهید که آیا نمونه برداری باید قبل از تقسیم کردن داده ها به داده های آموزش و تست انجام شود یا بعد از آن؟

هنگام نمونه برداری ، فقط مجموعه آموزشی باید نمونه برداری شود و نه مجموعه تست. در نتیجه ابتدا باید داده ها را به مجموعه های آموزشی و تست تقسیم کرد و سپس مجموعه آموزشی را نمونه برداری کرد. مجموعه تست به منظور نشان دادن داده‌هایی هست که مدل هنگام پیش‌بینی در دنیای واقعی با آن مواجه می‌شود یعنی توزیع واقعی و دست نخورده داده ها. این مجموعه تست باید با آنچه که در تولید و دنیای واقعی با آن مواجه می شوند تا حد امکان مطابقت داشته باشند. نمونه‌برداری از تست به طور کلی

باعث می‌شود که مشکل طبقه‌بندی آسان‌تر از آنچه هست به نظر برسد و منجر به ارزیابی‌های خوش بینانه و گمراه‌کننده عملکرد مدل می‌شود.

علاوه بر این، همیشه باید مراقب بود تا از leakage در ارزیابی مدل جلوگیری شود. در نتیجه اگر نمونه‌برداری قبل از تقسیم انجام شود، ممکن است اطلاعات از داده‌های تست به داده‌های آموزش منتقل شده و مدل از اطلاعات آینده استفاده کند که ممکن است در ارزیابی به نتایج نادرستی منجر شود.

۴. با استفاده از روش پیاده سازی شده، نمونه برداری را برای داده های دیتاست انجام دهید و هیستوگرام کلاسها را برای داده های جدید رسم کنید.

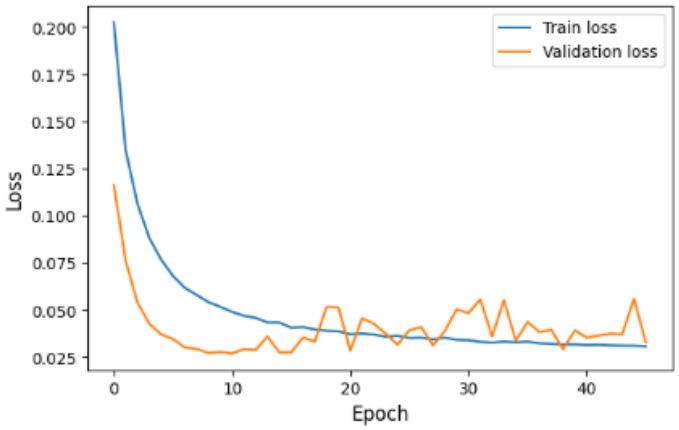


شکل ۲۴ - نمودار توزیع داده در کلاس ها با نمونه برداری

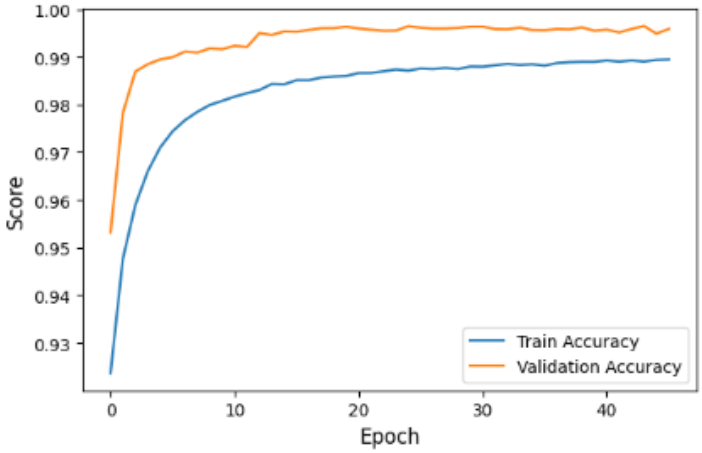
۳- ۴. آموزش مدل با داده های جدید

در زیر نتایج مدل ها برای مقایسه در شکل و جدول هایی آورده شده است. مدل شماره ۱ بر روی داده های ناتوازن آموزش داده شده است و مدل شماره ۲ روی داده های نمونه برداری شده آموزش داده شده است.

با توجه به نمودار دقت و loss مدل ۲ می توان دریافت که برخلاف زمانی که داده های نامتوازن داشتیم، در این حالت مدل با سرعت خوب و معقولی شروع به یادگیری می کند. دقت در مدل ۲ برای داده های آموزش و ارزیابی با شیب ملایمی افزایش می یابد. مقدار loss نیز در مدل ۲ با شیب خوبی کاهش می یابد و شاهد نوساناتی نیز برای داده ارزیابی در نمودار هستیم.



شکل ۲۵ - نمودار **loss** در هر ایپاک مدل ۲



شکل ۲۶ - نمودار دقت در هر ایپاک مدل ۲

همچنین شاهد پیشرفت معیار های precision،recall و fl-score برای مدل ۲ نسبت به مدل ۱ هستیم. با توجه به اینکه مشکل نامتوازن بودن داده ها حل شده است دیگر مدل biased prediction ندارد و می تواند داده های کلاس تقلب را به درستی پیش بینی کند.

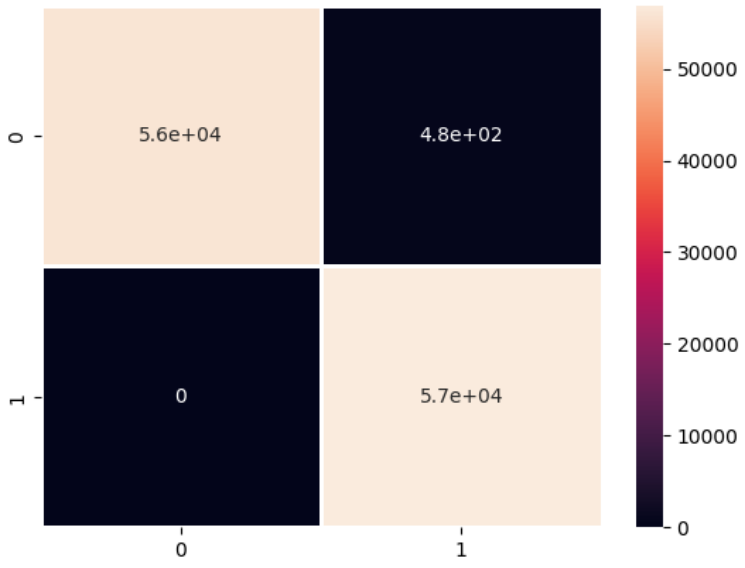
جدول ۴ - جدول classification report داده تست مدل ۲

	precision	recall	f1-score	support
0.0	1.00	0.99	1.00	56303
1.0	0.99	1.00	1.00	56916
accuracy			1.00	113219
macro avg	1.00	1.00	1.00	113219
weighted avg	1.00	1.00	1.00	113219

مقادیر متریک ها برای مدل ۲ به شکل زیر است.

Accuracy: 0.9958
Overall Precision: 0.9916
Recall: 1.0000
F1-score: 0.9958

نتایج confusion matrix نیز نشان می دهد که اکثر داده های کلاس تقلب به درستی پیش بینی شده اند و مدل دیگر تمایل ندارد 99 درصد داده های تست را در کلاس تراکنش های عادی دسته بندی کند. همانطور که قابل مشاهده است مدل ۲ تمام داده های تست کلاس 1 را درست پیش بینی کرده ولی ۴۸۱ داده از کلاس 0 را نیز به اشتباه در کلاس 1 پیش بینی کرده است.



شکل ۲۷ - نمودار **heatmap** را برای **confusion matrix** مدل ۲

جدول ۵ – مقادیر **Confusion Matrix** برای داده تست

Model	Confusion Matrix			
	TN	FP	FN	TP
Model_1	56648	8	31	59
Model_2	55822	481	0	56916

جدول ۶ – مقادیر متریک های ارزیابی برای داده تست

Model	Evaluation Results			
	Accuracy	Precision	Recall	F1 Score
Model_1	0.9993	0.8806	0.6556	0.7516
Model_2	0.9958	0.9916	1.0000	0.9958

اگرچه مقدار دقت مدل ۱ برای داده تست کمی بالاتر از مدل ۲ است، امتیازات برای سه معیار دیگر به میزان قابل توجهی افزایش یافته است. مقدار precision در مرحله تست از 0.8806 به 0.9916 و recall از 0.6556 به 1.0000 افزایش یافته است و F1 Score از 0.7516 به 0.9958 افزایش یافته است. به عبارت دیگر، نتایج نشان می‌دهد که درصد تراکنش‌های کارت اعتباری که به‌عنوان تراکنش تقلبی دسته بندی شده‌اند و در واقع تقلبی بوده اند نسبت به کل تراکنش‌های تقلبی پیش‌بینی‌شده 99.16 درصد است. به همین ترتیب، توانایی پیش‌بینی مدل برای دسته‌بندی صحیح تراکنش‌های جعلی 100 است.

نتایج به مقدار بسیار کمی با مقادیر گزارش شده در مقاله تفاوت دارند. برای مثال مقدار FN و FP برای مدل ۲ در پیاده سازی ما 0 و ۳۹۲ در مقاله ۴ و ۲۰۲ است. درصد های بدست آمده برای داده تست نیز همگی اختلاف کمی با مقادیر گزارش شده در مقاله دارند.