به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین پنجم

نام و نام خانوادگی	فربد سیاهکلی – سعید شکوفا
شماره دانشجویی	810198418 - 810198510
تاریخ ارسال گزارش	1401.10.13

فهرست

شکل‌ها

جدول‌ها

No table of figures entries found.

پاسخ 1. آشنایی با مفهوم توجه و پیاده سازی BERT

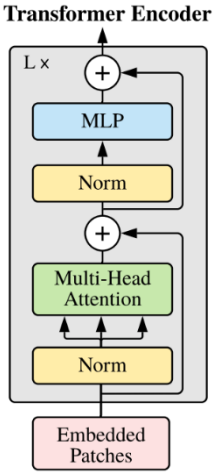
1- سوالات تشریحی:

مفهوم Attention: توجه تکنیکی است که هدف آن تقلید توجه شناختی است. این اثر بخش‌هایی از داده‌های ورودی را افزایش می‌دهد در حالی که بخش‌های دیگر را کاهش می‌دهد - انگیزه این است که شبکه باید تمرکز بیشتری را به بخش‌های کوچک، اما مهم داده‌ها اختصاص دهد مثلا در تصاویر اشیا پس زمینه تصویر برای ما اهمیت ندارد و نباید با آن توجه کرد بلکه باید به اشیا موجود توجه داشت که attention به همین موضوع اشاره دارد.

دلیل استفاده از Multi-head Attention به جای Single-head: مزیت اصلی Multi-head Attention، پایداری تمرین است، زیرا تعداد لایه های کمتری نسبت به Single-head Attention در هنگام حضور در همان تعداد position دارد. با توجه به اینکه سه مقدار(Q,K,V) را نیز باید ترین کند عملکرد بسیار بهتری نشان نسبت به single-head نشان میدهد.

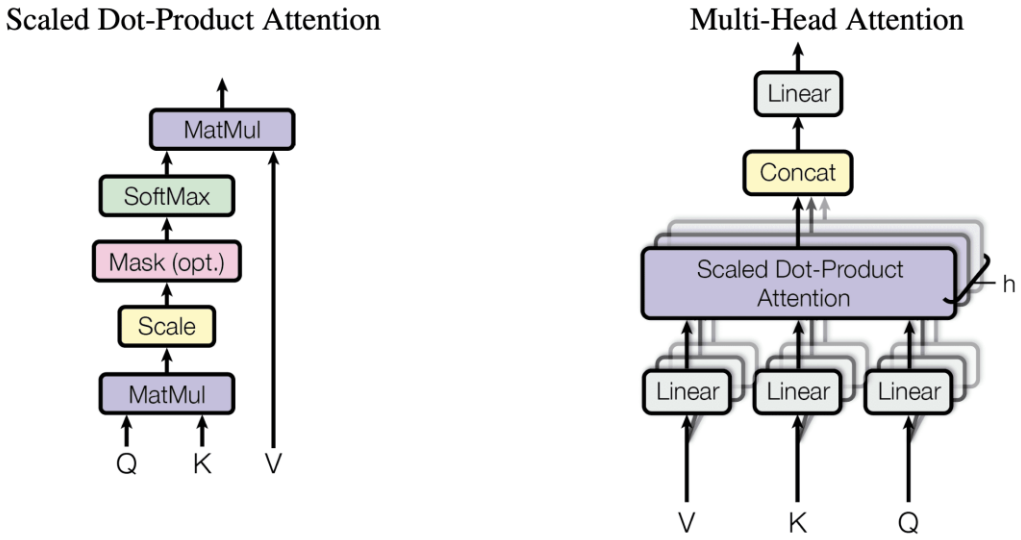
2- پیاده سازی:

در این بخش هدف پیاده سازی یک Encoder، BertEmbedding و یک Pooler است. بدین منظور ابتدا زیرلایه ها مانند AddNorm و MultiHeadAttention پیاده سازی می شوند.



شکل 1 شماتیک **Transformer Encoder**

ابتدا به ساخت یک MultiHeadAttention می پردازیم که به صورت زیر است:



شکل 2 شماتیک **Multi-Head Attention**

بدین منظور ابتدا لایه های Linear برای سه ورودی مستقل Key، Query و Value ساخته شده و سپس خروجی آنها به تابع Scaled Dot-Product Attention داده می شود که خروجی این تابع به صورت زیر محاسبه می گردد.

$$Attention(K,Q,V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

نهایتا نیز خروجی های تابع به یکدیگر Concatenate شده و از یک Linear به بعد خروجی Hidden Size عبور می‌کند.

حال تابع فعالسازی GELU پیاده سازی می شود که فرمول آن به شرح زیر است:

$$GELU(x) = xP(X \le x) = x\phi(x)$$
$$\rightarrow GELU(x) \approx 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

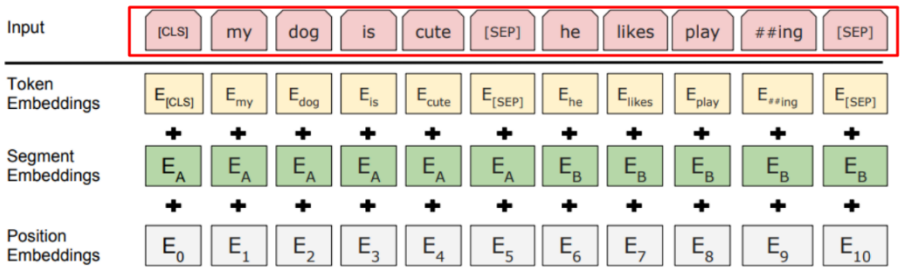
در ادامه لایه Feed Forward Network پیاده سازی می شود که شامل دو لایه Linear هستند که در میان آنها تابع فعالسازی GELU و یک Dropout انجام می شود.

در ادامه لایه Normalization پیاده سازی می شود که در آن ابتدا یک Dropout بر روی ورودی انجام می‌گیرد و سپس یک Layer Normalization به روی خروجی Dropout به علاوه‌ی اتصال Residual اعمال می‌شود.

حال می‌توان با استفاده از Sub Layer های تعریف شده اقدام به ساخت **Encoder** کرد.

در ادامه اقدام به ساخت BERT Embedding می‌کنیم. در این ماژول دو Embedding برای Token و Position تعریف می‌گردد. دیتای ورودی Tokenize شده از Token Embedding گذشته و یک رشته از اعداد طبیعی از صفر تا MaxLen از Positional Embedding می‌گذرند. نهایتا این دو خروجی با یکدیگر جمع شده تا ورودی به Encoder را تشکیل دهند.

مفهوم Segment Embedding: Segment Embedding اساساً شماره جمله‌ای هستند که در یک بردار کدگذاری می‌شوند. مدل باید بداند که آیا یک نشانه خاص متعلق به جمله A است یا جمله B در BERT. این امر با تولید نشانه ثابت دیگری به نام Segment Embedding - یک نشانه ثابت برای جمله A و یکی برای جمله B به دست می آید.



شکل 3 شماتیک **BERT Embeding**

در ادامه پس از تکرار کافی لایه‌ی Encoder، یک لایه Pooler در انتها قرار می‌گیرد تا خروجی را به بعدی برابر با Hidden Size تبدیل کند.

حال در تابع Create_BERT اقدام به قراردهی ماژول‌ها می‌کنیم.

```
def create_BERT(vocab_size, maxlen, hidden_size, num_layers, num_att_heads, intermediate_size, drop_rate=0.1):
  """   creates a BERT model based on the arguments provided
        Arguments:
        vocab_size: number of words in the vocabulary
        maxlen: maximum length of each sentence
        hidden_size: dimension of the hidden state of each encoder layer
        num_layers: number of encoder layers
        num_att_heads: number of attention heads in the multi-headed attention layer
        intermediate_size: dimension of the intermediate layer in the feed-forward sublayer of the encoders
        drop_rate: dropout rate of all the dropout layers used in the model
        returns: """
  inputs = layers.Input(shape=(maxlen,))
  embedding_layer = BertEmbedding(maxlen, vocab_size, hidden_size)
  x = embedding_layer(inputs)

  for i in range(num_layers):
    transformer_block = Encoder(hidden_size, num_heads, 4*num_heads)
    x = transformer_block(x)

  x = layers.GlobalAveragePooling1D()(x)
  x = layers.Dropout(0.1)(x)
  Pool_layer = Pooler(hidden_size)
  x = Pool_layer(x)
  outputs = layers.Dense(1, activation="sigmoid")(x)
  model = keras.Model(inputs=inputs, outputs=outputs)

  return model
```

شکل 4 پیاده سازی مدل **BERT** با استفاده از **Sub Layer** ها

برای ساخت مدل ورودی را به ترتیب از لایه های زیر عبور می‌دهیم:

- BERT Embedding Layer
- Encoder Layer (12 Times)
- Global Average Pooling
- Dropout
- Pooling Layer
- Linear Layer(out_features=1)

نهایتا مدل به شرح زیر می‌شود که حدود 45.5 میلیون پارامتر آموزشی دارد.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 32)]	0
bert_embedding (BertEmbeddi ng)	(None, 32, 768)	15355392
encoder (Encoder)	(None, 32, 768)	2439984
encoder_1 (Encoder)	(None, 32, 768)	2439984
encoder_2 (Encoder)	(None, 32, 768)	2439984
encoder_3 (Encoder)	(None, 32, 768)	2439984
encoder_4 (Encoder)	(None, 32, 768)	2439984
encoder_5 (Encoder)	(None, 32, 768)	2439984
encoder_6 (Encoder)	(None, 32, 768)	2439984
encoder_7 (Encoder)	(None, 32, 768)	2439984
encoder_8 (Encoder)	(None, 32, 768)	2439984
encoder_9 (Encoder)	(None, 32, 768)	2439984
encoder_10 (Encoder)	(None, 32, 768)	2439984
encoder_11 (Encoder)	(None, 32, 768)	2439984
global_average_pooling1d (G lobalAveragePooling1D)	(None, 768)	0
dropout_36 (Dropout)	(None, 768)	0
pooler (Pooler)	(None, 768)	590592
dense_73 (Dense)	(None, 1)	769

```
==========
Total params: 45,226,561
Trainable params: 45,226,561
Non-trainable params: 0
_________
```

شکل 5 مدل نهایی تشکل یافته به همراه تعداد پارامتر

حال اقدام به آموزش شبکه برای 2 ایپاک می‌کنیم. نتیجه به شرح زیر است:

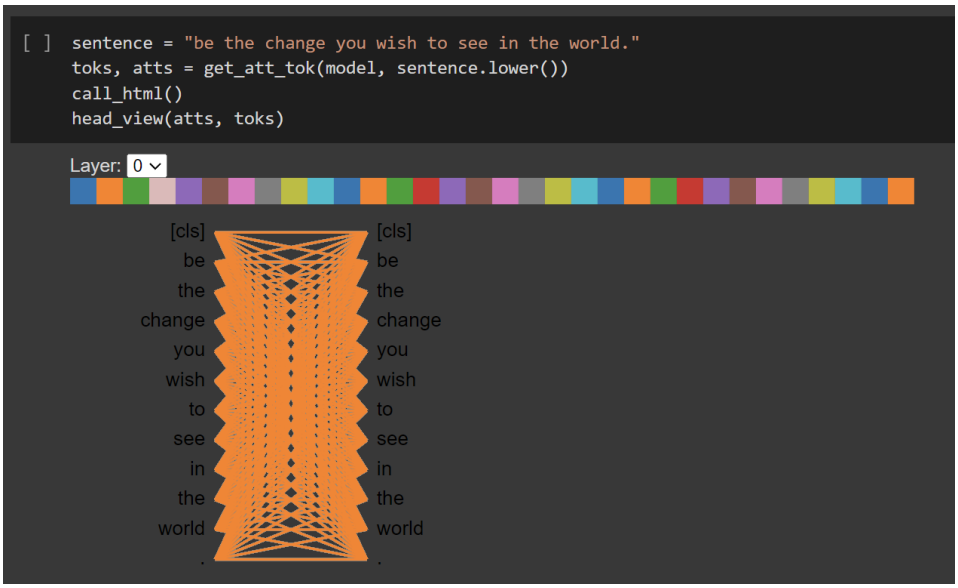
```
[ ] history = model.fit(
                    x_train,
                    y_train,
                    batch_size=128,
                    epochs=2,
                    validation_data=(x_test, y_test)
    )

    Epoch 1/2
    1970/1970 [==============================] - 823s 409ms/step - loss: 0.5225 - accuracy: 0.7279 - val_loss: 0.4645 - val_accuracy: 0.7772
    Epoch 2/2
    1970/1970 [==============================] - 802s 407ms/step - loss: 0.4250 - accuracy: 0.7981 - val_loss: 0.4594 - val_accuracy: 0.7841
```

شکل 6 آموزش مدل بر روی دیتاست داده شده

حال برای تابع Get Att Weights بدین صورت عمل کرده که ورودی را از شبکه گذرانده و به ترتیب attribute مدنظر یعنی att_weights را از تک تک Sub Layer های تعریف شده دریافت کرده و در یک لیست ذخیره می‌کنیم.

دریافت خروجی به ازای ورودی دلخواه:

نهایتا خروجی Visualization برای یک جمله دلخواه به صورت زیر است:

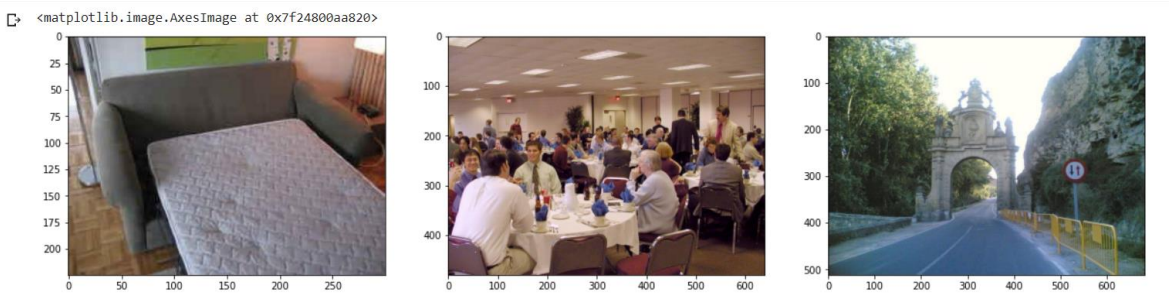


شکل 7 خروجی **Visualization** برای یک ورودی دلخواه

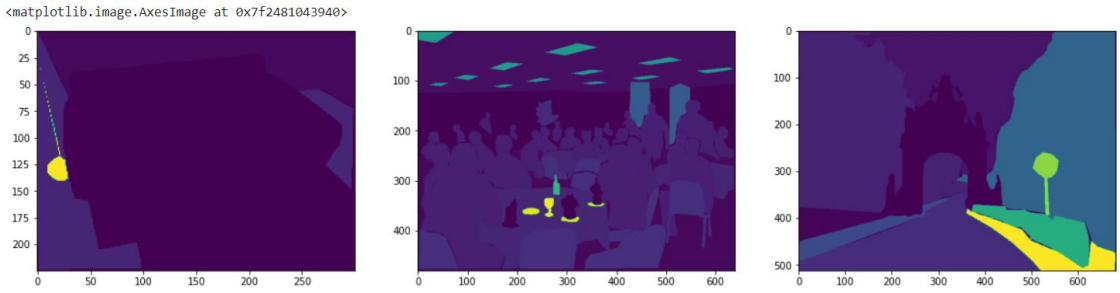
پاسخ ۲ – آشنایی با کاربرد تبدیل کننده‌ها در تصویر

الف) بازآموزش مدل BEiT:

اصل سه تصویر:



سه تصویر تقسیم بندی معنایی در مجموعه دادگان:



سه تصویر خروجی پس از بازآموزش:

در این بخش ابتدا پیش پردازش‌های لازمه بر روی دیتاست Scene_parse_150 انجام گرفت. سپس مدل BEiT برای تسک Segmentation به گونه‌ای طراحی گردید که خروجی آن فیچرهایی به تعداد کلاسهای موجود در شبکه باشد که هرکدام ابعادی برابر با تصویر ورودی دارند. در این بخش از مدل Pretrain در دسترس Microsoft در سایت Huggingface استفاده شده است.

در ادامه اقدام به آموزش شبکه کرده که متاسفانه پس از گذشت دو Batch از شبکه به مشکل CUDA Out of Memory برخورد کردیم که به دلیل تعداد پارامترهای بالای شبکه (حدود 163.4 میلیون پارامتر) و فضای بزرگی که آموزش آن دارد ایجاد شده است.

```
[15]  sum(p.numel() for p in model.parameters())

      163407980
```

شکل 8 تعداد پارامترهای شبکه **BEiT** برای **Segmentation**

ب) طبقه بندی تصاویر:

طبقه بند MLP تصاویر: مدل پیاده سازی شده برای این منظور:

```
Sequential(
  (0): Linear(in_features=3072, out_features=4096, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.2, inplace=False)
  (3): Linear(in_features=4096, out_features=8192, bias=True)
  (4): ReLU()
  (5): Dropout(p=0.2, inplace=False)
  (6): Linear(in_features=8192, out_features=4096, bias=True)
  (7): ReLU()
  (8): Dropout(p=0.2, inplace=False)
  (9): Linear(in_features=4096, out_features=1024, bias=True)
  (10): ReLU()
  (11): Linear(in_features=1024, out_features=256, bias=True)
  (12): ReLU()
  (13): Linear(in_features=256, out_features=10, bias=True)
)
```

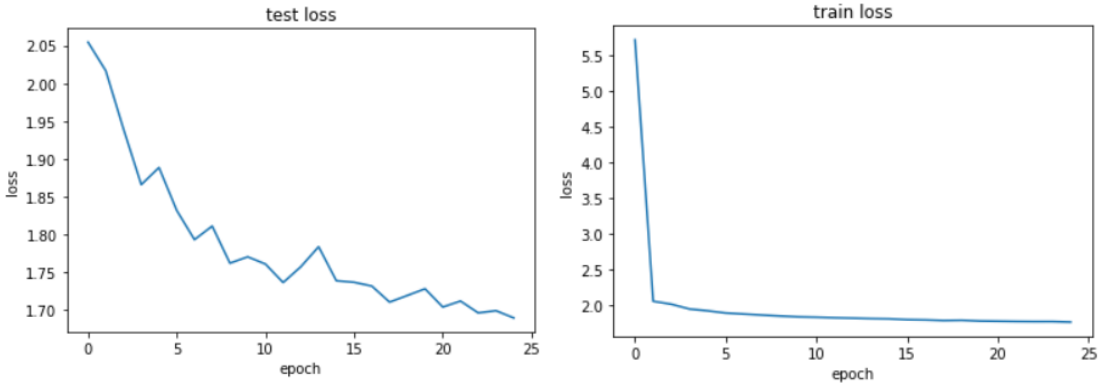
شکل 9 مدل **MLP** ساخته شده

لازم به ذکر است که شبکه MLP پیاده سازی شده به گونه ای تنظیم شده است که تعداد پارامتر های آن تقریبا با تعداد پارامتر های شبکه BEiT برابر باشد تا بتوان مقایسه خوبی میان این دو شبکه انجام داد. نتایج این شبکه پس از 25 ایپاک ترین:

```
epoch 23 train loss : 1.7702273738627532
epoch 23 test loss : 1.696425724029541
epoch 24 train loss : 1.7703936513589353
epoch 24 test loss : 1.6993773818016051
epoch 25 train loss : 1.7631912924805466
epoch 25 test loss : 1.6898074388504027
```

شکل 10 نتایج آموزش شبکه **MLP**

نمودار های خطا برای داده های ترین و تست:



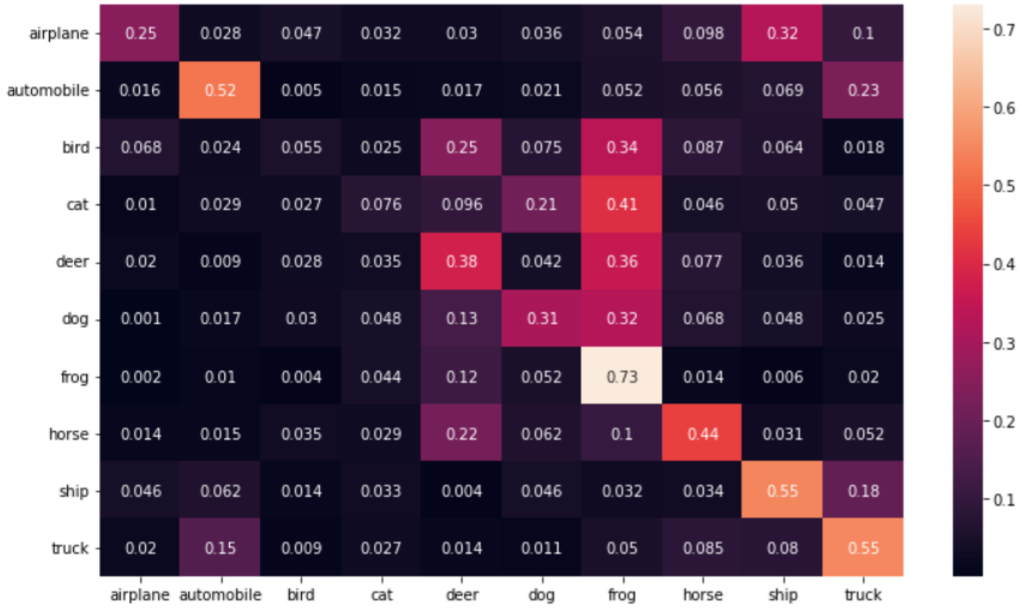
شکل 11 نمودار خطای **Train** و **Test** برای شبکه **MLP**

دقت و صحت در شبکه MLP :

```
test loss : 1.6898074388504027
Acc : 0.9044486284255981
Pre : 0.6841059923171997
```

شکل 12 مقادیر **Accuracy** و **Precision** برای شبکه **MLP**

ماتریس آشفتگی برای شبکه MLP:



شکل 13 ماتریس آشفتگی برای مدل **MLP**

طبقه بند BEiT تصاویر:

مدل BEiT ساخته شده و در آخرین لایه از یک Linear استفاده شده تا خروجی به بعد 10 تقلیل یابد. همچنین تعداد پارامترهای این مدل به حدود 85.7 میلیون پارامتر است.

```
        (layernorm): Identity()
        (pooler): BeitPooler(
          (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
      (classifier): Linear(in_features=768, out_features=10, bias=True)
    )

[16] sum(p.numel() for p in model.parameters())

    85769674
```

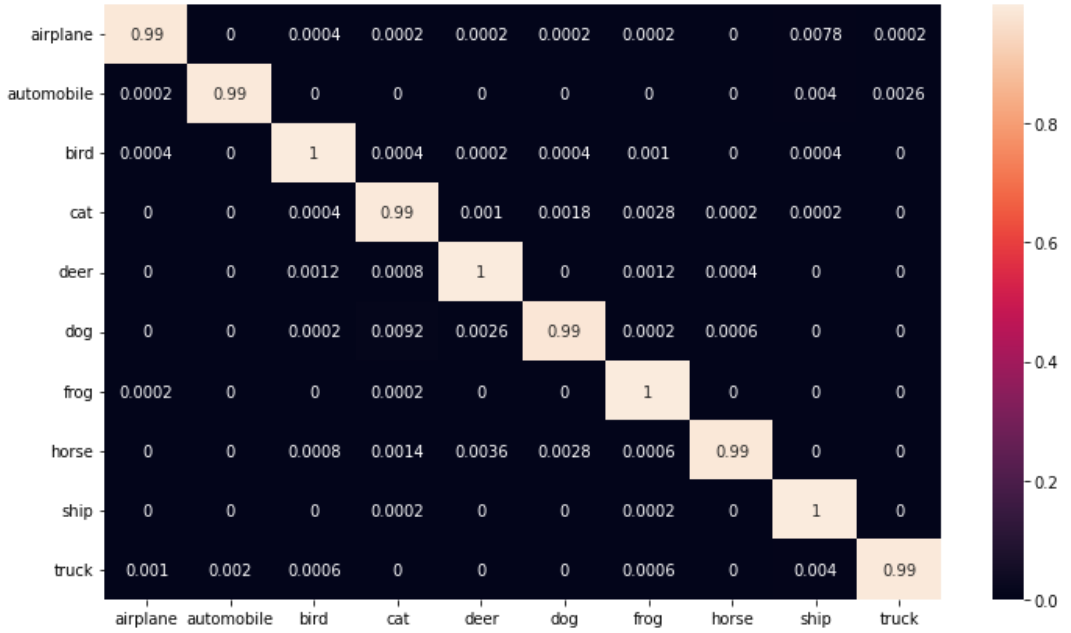
شکل 14 مدل **BEiT** مورد استفاده به همراه تعداد پارامتر قابل آموزش

دقت و صحت در شبکه BEiT :

```
test loss : 0.01721311990815642
Acc : 0.9988044500350952
Pre : 0.9942799806594849
```

شکل 15 مقادیر **Accuracy** و **Precision** برای شبکه **BEiT**

ماتریس آشفتگی برای شبکه BEiT:



شکل 16 ماتریس آشفتگی برای شبکه **BEiT**

مقایسه طبقه بندها: برای مقایسه عادلانه سعی شد که تعداد پارامتر های هر دو شبکه یکسان باشد. همانطور که انتظار میرفت شبکه MLP نتوانسته به خوبی classificationرا انجام دهد و خطای

بالایی را به همراه داشته است که این قضیه در ماتریس آشفتگی به خوبی مشخص است همچنین دقت و صحت آن نیز بالا نبود. اما در شبکه BEiT مدل پس از چند ایپاک به دقت بسیار خوبی رسیده و توانسته است که به خوبی عملیات classificationرا بر روی دیتای cifar10 انجام دهد که نشان از قدرت transformer ها در امر vision و به طور جزئی تر classification شود. همچنین این مدل دقت و صحت بسیار زیادی کسب کرد و ماتریس آشفتگی آن بیانگر آن است که با دقت بسیار خوبی تقریبا تمامی عکس ها را به درستی طبقه بندی کرده است به طوریکه دقت و صحت آن بالای 99 درصد بوده و نزدیک به صد است ولی دقت شبکه MLP 90 درصد و صحت آن 68.4 درصد بوده که در مقایسه با BEiT بسیار کمتر بوده و قدرت مدل BEiT از مدل NLP در این مسئله بسیار بیشتر است.

ج) پرسش ها:

در CNN در کدام بخش توجه رخ می‌دهد:

در یک CNN، مقادیر بسیاری از هسته های کانولوشن آموخته می شوند و زمانی که آموخته شوند، هسته ها ثابت هستند. در واقع در هر موقعیتی ورودی حاصل ضرب نقطه ای بین ورودی های داخل پنجره و همان هسته های CNN است و آنجایی که پنجره ضریب بیشتری دارد بیشتر مورد توجه قرار میگرد.

در یک شبکه ی عصبی، در ارتباط یک لایه با لایه ی بعد، چه تفاوتی میان یک شبکه ی convolution با شبکه ی توجه همگانی و شبکه ی توجه محلی وجود دارد؟

در کانولوشن در واقع دیتا های اطراف یک دیتا به همراه آن با اعمال یک تابع یکسان خروجی را تولید میکنند، در توجه محلی دیتاهای اطراف به همراه خود دیتا با اعمال تابع های متفاوتی خروجی را می دهند و در توجه همگانی تمامی دیتا های اطراف به همراه خود آن دیتا با اعمال توباع متفاوت خروجی می دهند.

د) درست و نادرست:

در بخشی از لایه های تبدیل کننده ی Vanilla از شبکه ی LSTM استفاده شده است.

نادرست: RNN های Vanilla حالت سلولی ندارند. آنها فقط حالت های پنهان دارند و آن حالت های پنهان به عنوان حافظه برای RNN ها عمل می کنند. در همین حال، LSTM دارای هر دو حالت سلولی

و حالت های پنهان است. حالت سلولی توانایی حذف یا اضافه کردن اطلاعات به سلول را دارد که توسط "Gate" تنظیم می شود.

یک تبدیل کننده از چند بلوک رمزگذار و چند بلوک رمزگشا تشکیل شده است. درست

ترانسفورماتور از معماری رمزگذار - رمزگشا استفاده می کند. رمزگذار ویژگی ها را از یک جمله ورودی استخراج می کند و رمزگشا از ویژگی ها برای تولید یک جمله خروجی (ترجمه) استفاده می کند. رمزگذار در ترانسفورماتور از چندین بلوک رمزگذار تشکیل شده است.

رمزگذار در ترانسفورمر از چندین بلوک رمزگذار تشکیل شده است. یک جمله ورودی از بلوک های رمزگذار عبور می کند و خروجی آخرین بلوک رمزگذار به ویژگی های ورودی رمزگشا تبدیل می شود. رمزگشا همچنین از چند بلوک رمزگشا تشکیل شده است. هر بلوک رمزگشا ویژگی ها را از رمزگذار دریافت می کند.(البته می تواند فقط یک رمزگذاز و یک رمزگشا داشته باشد)

Multi-Head-Attention از یک بخش توجه و چند لایه ی تمام متصل موازی تشکیل شده است. درست

Multi-Head-Attention یک ماژول برای مکانیسم های توجه است که چندین بار به طور موازی از یک مکانیسم Attention عبور می کند. سپس خروجی های توجه مستقل به هم پیوسته و به صورت خطی به بعد مورد انتظار تبدیل می شوند.

وجود Positional Encoding در ساختار یک تبدیل کننده حیاتی است و بدون آن شبکه از کار میافتد. درست

Positional Encoding یک سیگنال، زمان را به ورودی اضافه می کند. این امر ضروری است زیرا برخلاف RNN ها، هیچ بازگشتی در ترانسفورمرها وجود ندارد که اطلاعات موقعیتی را در خود معماری شبکه حمل می کند. جاسازی ها را می توان یاد گرفت، اما به خوبی به موقعیت هایی که در داده های آموزشی دیده نمی شود تعمیم نمی یابد.