به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین پنجم

نام و نام خانوادگی	آرمان فروزش – مهسا ندافی قهنوییه
شماره دانشجویی	810100490 - 111946
تاریخ ارسال گزارش	۱۴۰۱.10.20

فهرست

شکل‌ها

پاسخ 1. آشنایی با مفهوم توجه و پیاده سازی مدل BERT

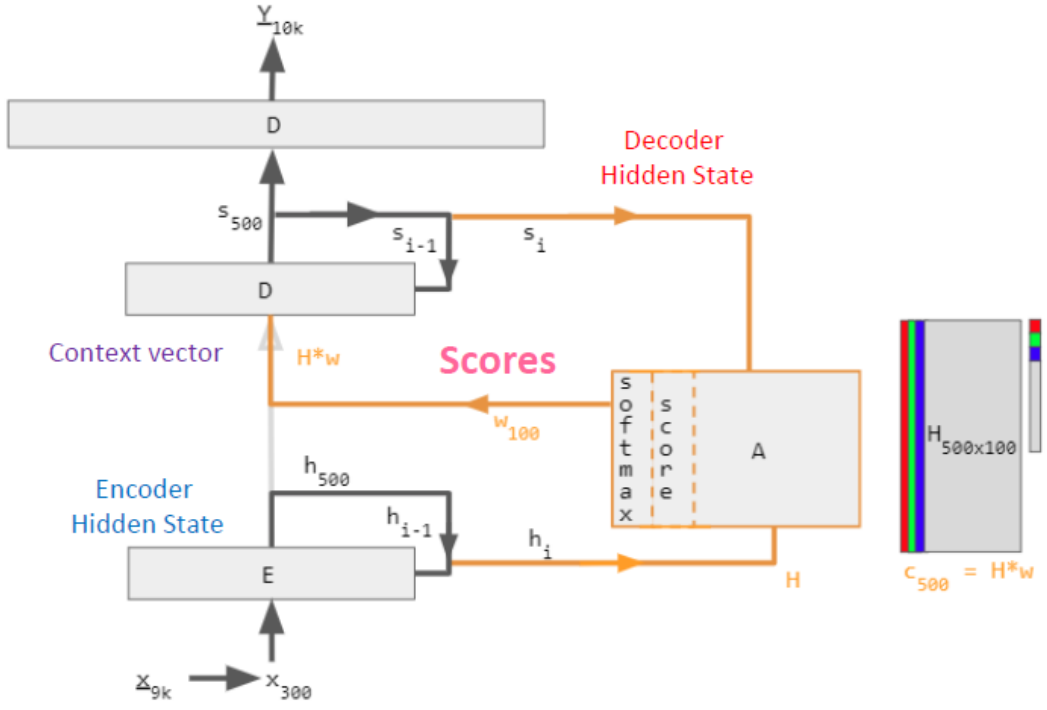
1-1 پیاده سازی کدگذار

پرسش ها:

1 . توضیح مختصری در مورد مفهوم توجه دهید.

با توجه به شکل زیر مکانیزم توجه در فضای hidden state ماتریسی مانند A ایجاد میکند که در هر ستون آن کلمات جمله قرار میگیرد در این فضا همبستگی بین کل کلمات و کلمه ی حاضر در decoder تخمین زده شده و خود را به شکل وزن برای بازتولید خروجی decoder نشان میدهد در واقع وزن تولید شده میزان اهمیت هر یک از کلمات جمله را برای ترجمه ی کلمه ی حاضر نشان میدهد و در این ترجمه بر اساس وزن خود تاثیر گذار خواهد بود.

بطور دقیق تر در این معماری علاوه بر state سلول کدگذار آخر، تمام خروجی‌های کدگذار را هم به کدگشا می‌فرستیم. تا در هر گام، کدگشا یک جمع وزن‌دار بین این خروجی‌ها حساب کند و همین وزن‌ها هستند که مفهوم توجه را پیاده می‌کنند؛ یعنی آنکه که وزن بیشتری داره، سهم بیشتری را در این بردار حاصل از جمع وزن‌دار دارد پس به آن توجه بیشتری می‌شود.

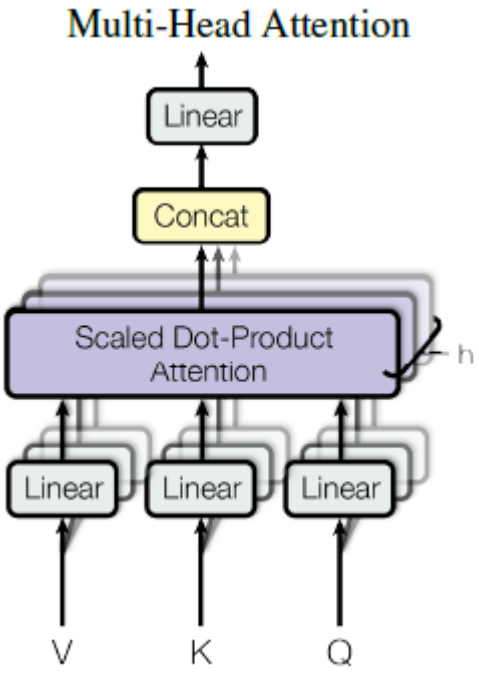


شکل 1-1 مکانیزم توجه

توسعه معماری ترانسفورمر نشان داد که مکانیسم‌های توجه به خودی خود قدرتمند هستند و پردازش متوالی مکرر داده‌ها برای دستیابی به دستاوردهای کیفیت شبکه عصبی بازگشتی با مکانیزم توجه ضروری نیست. ترانسفورمرها از مکانیزم توجه بدون شبکه عصبی بازگشتی استفاده می‌کنند، همه نشانه‌ها را همزمان پردازش می‌کنند و وزن توجه بین آنها را در لایه‌های متوالی محاسبه می‌کنند. از آنجایی که مکانیسم توجه فقط از اطلاعات مربوط به نشانه‌های دیگر از لایه‌های پایین‌تر استفاده می‌کند، می‌توان آن را برای همه نشانه‌ها به صورت موازی محاسبه کرد که منجر به بهبود سرعت تمرین می‌شود.

2 . چرا در تبدیل کننده از Multi-head attention به جای Single-head attention استفاده میشود ؟

Vanila Transformer در هر بلوک encoder , decoder 8 head می سازد. در واقع , self attention mask self attention , cross attention که به ترتیب در کد وجود دارد با initialization های متفاوت چیزی حدود 8 head مختلف را سعی میکنند ایجاد کنند. head های مختلف باعث افزونگی در مکانیزم توجه میشود و چون initialization تصادفی شکل میگیرد هرچند ورودی یکسان است گویی هر کدام از head ها در مکانیزم multi head attention function وجود دارد تلاش میکند یک توجه متفاوت از head دیگر بسازد و در انتها concat میشوند. این تعدد head ها افزونگی و پیچیدگی لازم برای مکانیزم توجه را بالا میبرد.



شکل 1-2 بلوک **multi-head Attention**

- **کامل سازی کد Attention**

```
class MultiHeadAttention(layers.Layer):
    def __init__(self, hidden_size, num_heads):

        super(MultiHeadAttention, self).__init__()
        self.hidden_size = hidden_size
        self.num_heads = num_heads
        self.projection_dim = hidden_size // num_heads
        self.Q = layers.Dense(hidden_size)
        self.K = layers.Dense(hidden_size)
        self.V = layers.Dense(hidden_size)
        self.out = layers.Dense(hidden_size)

    def attention(self, query, key, value, mask):
        s = tf.matmul(query, key, transpose_b=True) / tf.math.sqrt(tf
.cast(self.projection_dim, tf.float32))
        if mask is not None:# Apply mask to the attention scores
            s += -1e9 * mask
        weights = keras.backend.softmax(s)# Computing the weights by
a softmax operation
        output = tf.matmul(weights, value)
        return output, weights

    def separate_heads(self, x, batch_size):
        x = tf.reshape(x, (batch_size, -
1, self.num_heads, self.projection_dim))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, inputs, att_mask):
        batch_size = tf.shape(inputs)[0]
        query = self.separate_heads(self.Q(inputs)   , batch_size)
        key = self.separate_heads(self.K(inputs), batch_size)
        value = self.separate_heads(self.V(inputs) , batch_size)
        attention, self.att_weights = self.attention(query, key, valu
e, att_mask)
        attention = tf.transpose(attention, perm=[0, 2, 1, 3])
        concat_attention = tf.reshape(attention, (batch_size, -
1, self.hidden_size))
        output = self.out(concat_attention)
        return output
```

در این بخش با توجه به مقاله از فرمولهای زیر استفاده شده است:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- کد GELU

```
def GELU(x):
 output = 0.5 * x * (1 + T.tanh(T.sqrt(2 / np.pi) * (x + 0.044715 * T
.pow(x, 3))))
  return output
```

فرمول:

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

- **FNN**

```
  class FFN(layers.Layer):

    def __init__(self, intermediate_size, hidden_size, drop_rate):

      super(FFN, self).__init__()
      self.intermediate = layers.Dense(intermediate_size, activation=
GELU, kernel_initializer=TruncatedNormal(stddev=0.02))
      self.out = layers.Dense(hidden_size, kernel_initializer=Truncat
edNormal(stddev=0.02))
      self.drop = layers.Dropout(drop_rate)

    def call(self, inputs):
      fc1 = self.intermediate (inputs)
      fc2 = self.out (fc1)
      output = self.drop (fc2)
      return output
```

- **ADDNORM**

```
class AddNorm(layers.Layer):

  def __init__(self, LNepsilon, drop_rate):

    super(AddNorm, self).__init__()
    self.LN = layers.LayerNormalization(epsilon=LNepsilon)
    self.dropout = layers.Dropout(drop_rate)

  def call(self, sub_layer_in, sub_layer_out):

    sum = sub_layer_in + sub_layer_out
    LNoutput = self.LN(sum)
    output = self.dropout (LNoutput)

    return output
```

- **ENCODER**

```
class Encoder(layers.Layer):

    def __init__(self, hidden_size, num_heads, intermediate_size, dro
p_rate=0.1, LNepsilon=1e-12):
      super(Encoder, self).__init__()
      self.multihead_att = MultiHeadAttention(hidden_size, num_heads)
      self.addnormalization1 = AddNorm(LNepsilon, drop_rate)
      self.feedforward = FFN(intermediate_size, hidden_size, drop_rat
e)
      self.addnormalization2 = AddNorm(LNepsilon, drop_rate)

    def call(self, inputs, mask):
      multihead_output = self.multihead_att(inputs, mask)
      addnorm_output1 = self.addnormalization1(multihead_output)
      feedforward_output = self.feedforward(addnorm_output1)
      addnorm_output2 = self.addnormalization2(addnorm_output1, feedf
orward_output)

      return addnorm_output2

    def compute_mask(self, x, mask):

      return mask
```

2-1 پیاده سازی مدل Bert

مدل برت از لحاظ ساختاری یک ترنسفورمری است که تنها قسمت انکودر را دارد و فاقد قسمت دیکودر است. انکودر ترنسفورمر یک دنباله به طول N توکن را گرفته و در خروجی خود برای هر یک از این توکن‌ها یک بردار ویژگی تولید میکند، یعنی مثلا اگر سایز بردارهای ویژگی d باشد در خروجی دیکودر N تا بردار با سایز d خواهیم داشت. در برت بردار بازنمایی یک کلمه میتواند همزمان به کلمات سمت چپ و راست خود توجه کند.

در هنگام پیش آموزش برت با تسک مدل زبانی ماسک‌شده، به تصادف به جای پانزده درصد از توکن‌های دنباله، توکن MASK قرار می‌گیرد. برای مثال جمله "رنگ آسمان آبی است" به "رنگ آسمان MASK است" تغییر پیدا میکند. این جمله تغییریافته به برت به عنوان ورودی داده می‌شود و برت برای هر کدام از چهار کلمه یک بردار بازنمایی با سایز مثالی ۷۶۸ تولید میکند. حال بر روی بردار بازنمایی مربوط به

کلمه MASK یک لایه شبکه عصبی قرار می‌گیرد که تسک آن پیش‌بینی توزیع احتمال کلمات محتمل به جای MASK است.

هر ورودی کلمه با استفاده از تعبیه خام BERTدر یک توکن تعبیه می‌شود؛ این توکن بسته‌ای از کلمات، حدود ۳۰۰۰۰ کلمه، است. ورودی باید با یک توکن طبقه‌بندی خاص [CLS] آغاز شود و به دنبال آن یک یا چندین جمله که با یک توکن جداسازی [SEP] خاص جدا شده‌اند، قرار بگیرد. واحدساز به هر یک از توکن‌هایی که رمزگذاری شده‌اند یک مقدار اضافه می‌کند؛ این مقدار نشان‌دهنده شاخص جمله و شاخص ورودی مکانی است.

- **bertembedding**

```
class BertEmbedding(layers.Layer):

    def __init__(self, vocab_size, maxlen, hidden_size):

      super(BertEmbedding, self).__init__()
      self.TokEmb = layers.Embedding(input_dim=vocab_size, output_dim
=hidden_size, mask_zero=True)
      self.PosEmb = tf.Variable(tf.random.truncated_normal(shape=(max
len, hidden_size), stddev=0.02))
      self.LN = layers.LayerNormalization(epsilon=1e-12)
      self.dropout = layers.Dropout(0.1)

    def call(self, inputs):
      self.bert = hub.Module(
            name="{}_module".format(self.name)
        )
      inputs = [K.cast(x, dtype="int32") for x in inputs]
      input_ids, input_mask, segment_ids = inputs
      bert_inputs = dict(input_ids=input_ids, input_mask=input_mask,
segment_ids=segment_ids)
      result = self.bert(inputs=bert_inputs, signature="tokens", as_d
ict=True)
      return result
    def compute_mask(self, x, mask=None):
      m = 1-tf.cast(self.TokEmb.compute_mask(x), tf.float32)
      m = m[:, tf.newaxis, tf.newaxis, :]
      return m
```

- **pooler**

```
class Pooler(layers.Layer):

    def __init__(self, hidden_size):

      super(Pooler, self).__init__()
```

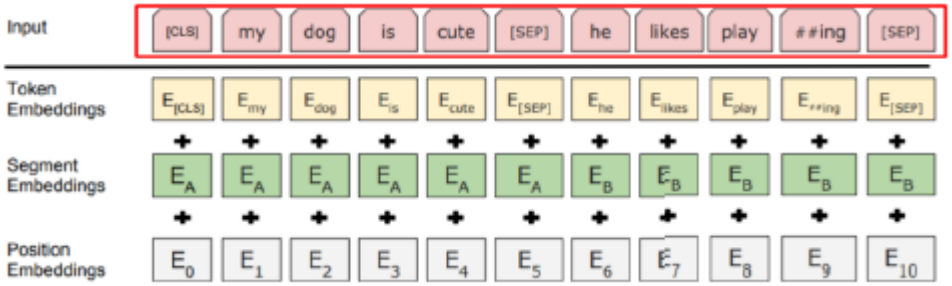
```
      self.dense = layers.Dense(hidden_size, activation='tanh')

    def call(self, encoder_out):

      out = tf.squeeze(encoder_out[:, 0:1, :], axis=1)
      return self.dense(out)
```

- **Create bert**

```
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 la
yer
        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:
  """
    bert_output = BertEmbedding(vocab_size, maxlen, hidden_size)
    pooler_output = Pooler(hidden_size)
    dense = keras.layers.Dense(256, activation='sigmoid')(pooler_outp
ut)
    drop = keras.layers.Dropout(drop_rate)(dense)
    dense = keras.layers.Dense(64, activation='sigmoid')(drop)

    model = keras.models.Model(inputs=vocab_size)


  return model
```

1. در مورد segment embeddings در BERT مختصر توضیح دهید.

Segment embedding شماره جمله ای است که در یک بردار کدگذاری میشود. مدل bert باید بداند که token خاص متعلق به جمله ی A است یا B این امر با تولید token ثابت برای هر جمله درsegment embedding صوت می پذیرد.

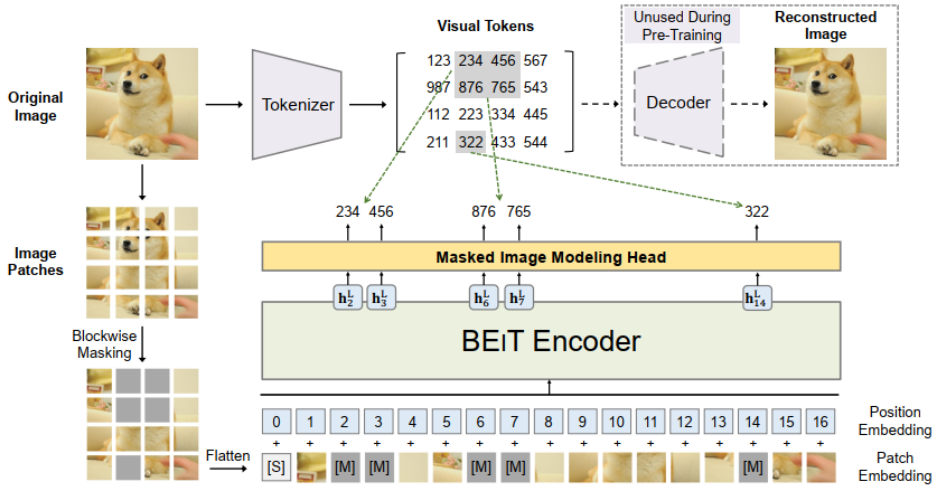


شکل 1-3 بخش ورودی مدل **Bert**

پاسخ 2. آشنایی با کاربرد تبدیل‌کننده‌ها در تصویر

2-۱. آشنایی با مدل BEiT

مدل BEIT یکی از مدل‌های تبدیل‌کننده برای تصاویر است که با بهرهگیری از مدل BERT ساخته شده است. مدل BEIT به شیوه‌ی خود نظارتی پیش آموزش یافته است؛ به این شکل که تصاویر به زیربخش های کوچکتری نسبت به تصویر اولیه تقسیم شده، سپس چندین بخش آن مخدوش شده، سپس به مدل داده می‌شوند. مدل باید سعی کند تا تصویر اصلی را بازسازی کند. پس از اتمام پیش آموزش، مدل به چند روش دیگر نیز آموزش میبیند و در نهایت برای استفاده در کارهای متفاوت مربوط به پردازش تصاویر آماده میشود. دو کار متداول در زمینهی پردازش تصویر، طبقه بندی ۴و تقسیم بندی معنایی5 تصاویر با تبدیل کننده‌ها است.



شکل 1.2 ساختار مدل **BEiT**

2-2. تقسیم‌بندی معنایی تصاویر

برای این بخش ابتدا دیتاست scene parse 150 را لود کرده و سپس بخشی از داده های مورد نظرمان را ذخیره میکنیم. این دیتاست شامل سه دسته image, annotation و scene category میباشد که ب اتوجه به اینکه هدف ما در این بخش تقسیم بندی معنایی است، تنها به داده های image و annotation نیاز داریم.

با توجه به اینکه مدل BEiT نیاز به ورودی هایی با سایز یکسان دارد، نیاز است که ایتدا دیتای مورد نظر برای ورود به مدل را resize و آماده کنیم. به منظور انجام این کار داده‌ها را ابتدا با استفاده از BeitFeatureExtractor پیش پردازش می‌کنیم. با این کار داده ها به سایز مورد نظر ما که در این مسئله با توجه به مدل انتخابی، سایز مورد نظر 640 میباشد resize می‌شوند. توجه شود که پس از استفاده از این feature extractor، داده های خروجی به شکل tensor 4 بعدی میشوند که بعد چهارم آن batchsize میباشد که با توجه به اینکه ما در ادامه خودمان میخواهیم batchsize را تعریف کنیم، با استفاده ا دستور squeeze() این بعد مربوط به batchsize را از بین می‌بریم. علاوه بر این feature extractor مربوطه خود دو مقدار pixel values و labels را که به ترتیب مقادیر پیکسل های موجود در عکس و کلاس‌های موجود در عکس می‌باشند را به عنوان خروجی می‌دهد که دقیقا داده های مورد نیاز مدل BEiT می‌باشند. مدل BEiT مورد استفاده در این سوال که با استفاده از دستور BeitForSemanticSegmentation فراخوانی میشود، beit-base-finetuned-ade-640-640 میباشد.

```
from transformers import BeitFeatureExtractor, BeitForSemanticSegmentation
import torch

model_name = "microsoft/beit-base-finetuned-ade-640-640"
feature_extractor = BeitFeatureExtractor(do_resize=True, size=640, do_center_crop=False)
model = BeitForSemanticSegmentation.from_pretrained(model_name)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

شکل 2.2 کد مربوط به مدل **BEiT**

در ادامه برای ارزیابی مدل بعد از train از شاخص mean_iou استفاده میکنیم. بهینه ساز مورد استفاده برای آموزش مدل نیز با توجه به مقاله AdamW میباشد. برای اینکه با محدودیت حافظه روبرو هستیم

batchsize=2 قرار داده شده است و سعی میکنیم که مدل را برای 20 ایپاک آموزش دهیم که در نهایت باز هم با ارور کمبود حافظه مواجه میشویم.

```
# define optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=0.00006)
# move model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
Epoch: 0
0%                                                  0/25 [00:00<?, ?it/s]
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
<ipython-input-29-6b6829b45f0a> in <module>
     11
     12         # forward + backward + optimize
---> 13         outputs = model(pixel_values=pixel_values, labels=labels)
     14         loss, logits = outputs.loss, outputs.logits
     15

                                3 frames
/usr/local/lib/python3.8/dist-packages/torch/nn/functional.py in interpolate(input, size, scale_factor, mode,
align_corners, recompute_scale_factor, antialias)
   3948         if antialias:
   3949             return torch._C._nn._upsample_bilinear2d_aa(input, output_size, align_corners, scale_factors)
-> 3950         return torch._C._nn.upsample_bilinear2d(input, output_size, align_corners, scale_factors)
   3951     if input.dim() == 5 and mode == "trilinear":
   3952         assert align_corners is not None

OutOfMemoryError: CUDA out of memory. Tried to allocate 8.17 GiB (GPU 0; 14.76 GiB total capacity; 10.16 GiB already
allocated; 3.20 GiB free; 10.28 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting
max_split_size_mb to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

شکل 3.2 کد مربوط به آموزش مدل **BEiT**

در نهایت با استفاده از مدل های pre-trained موجود برای این مدل، تقسیم بتئ=دی معنایی را برای 3 نمونه از داده های scene parse 150 انجام میدهیم. به منظور انجام این کار، مقدار pixel_value محاسبه شده از feature extractor را به عنوان ورودی به مدل داده و خروجی را محاسبه . مقادیر logits مربوط به خروجی را با استفاده از ade_palette که به هر کلاس یک رنگ مجزا اختصاص میدهد.

```
pixel_values = feature_extractor(image, return_tensors="pt").pixel_values.to(device)
```

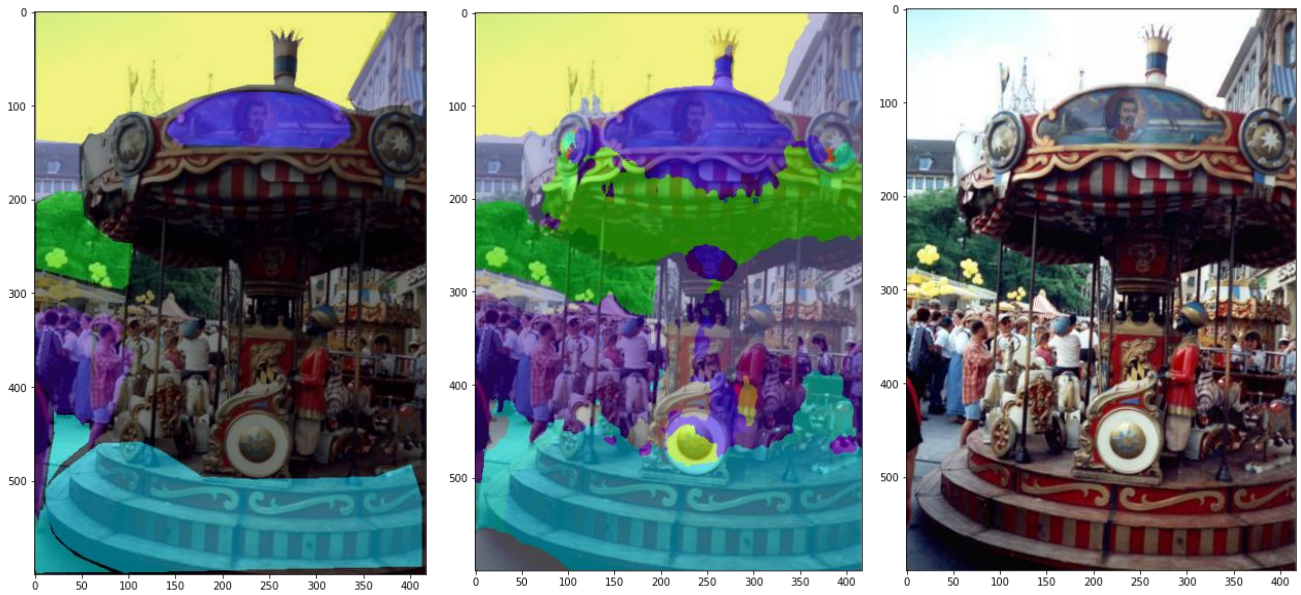
```
outputs = model(pixel_values)
logits = outputs.logits
```

```
logits[0, :3, :3]
```

```
tensor([[[-1.4569,  1.4434,  1.2371],
         [ 1.0061,  2.6839,  2.6044],
         [ 1.0215,  2.4831,  2.3756]],

        [[-7.4924, -5.8586, -6.4493],
         [-6.4254, -5.0267, -5.9024],
         [-6.4296, -5.2805, -6.0122]],

        [[-6.2638, -4.3394, -5.1230],
         [-5.4162, -4.3672, -5.1931],
         [-5.4288, -4.5351, -5.5584]]], grad_fn=<SliceBackward0>)
```

شکل 4.2 محاسبه خروجی مدل برای تقسیم‌بندی معنایی

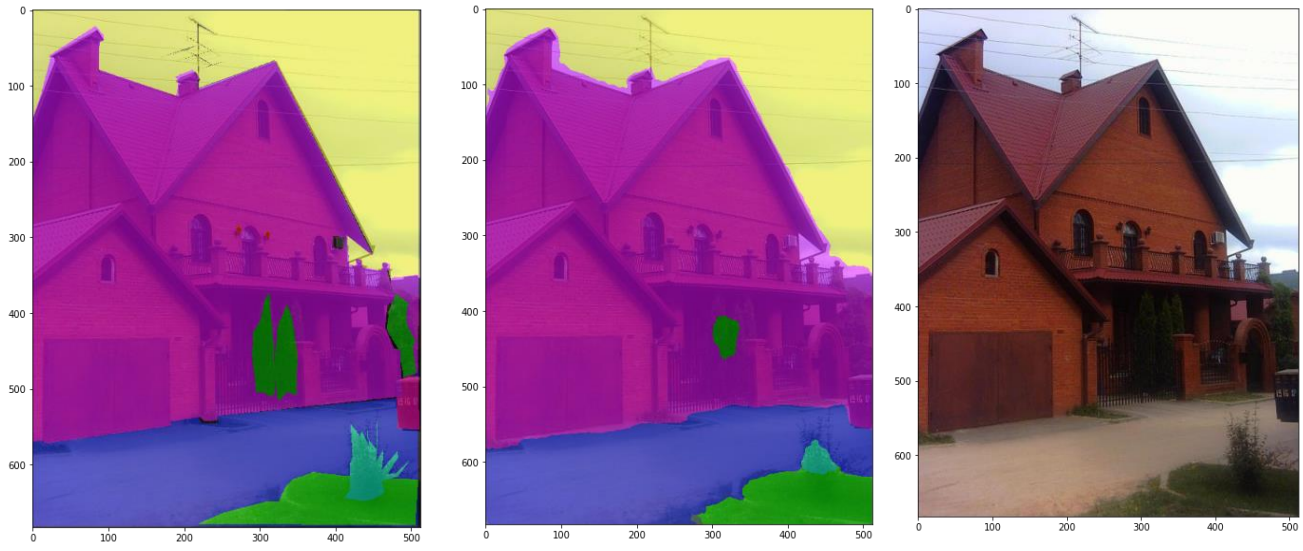
خروجی شبکه برای تصویر اول:



(ج) (ب) (الف)

شکل 5.2 الف)عکس ورودی ب)عکس خروجی شبکه ج)مقدار **annotation** مربوط به عکس

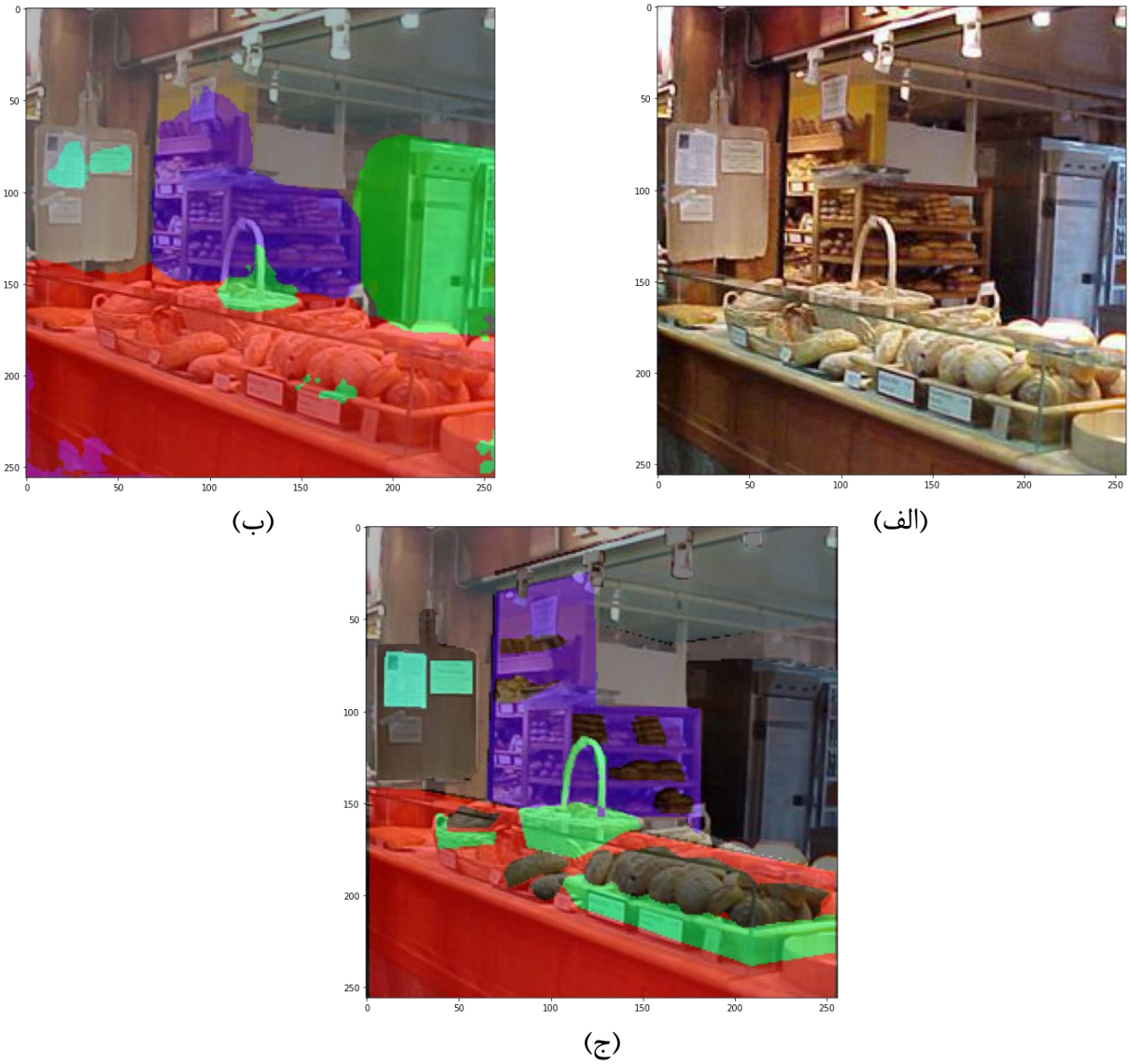
خروجی شبکه برای تصویر دوم:



(ج) (ب) (الف)

شکل 6.2 الف)عکس ورودی ب)عکس خروجی شبکه ج)مقدار **annotation** مربوط به عکس

خروجی شبکه برای تصویر سوم:



(ب) (الف)

(ج)

شکل 7.2 الف)عکس ورودی ب)عکس خروجی شبکه ج)مقدار **annotation** مربوط به عکس

همانگونه که دیده میشود این مدل باتوجه به اینکه روی دیتای ورودی fine-tune نشده است عملکرد نسبتا مناسبی داشته و عملکرد آن در تصاویری که کلاس های کمتری در آنها وجود دارد همچون شکل 6.2 بهتر است.

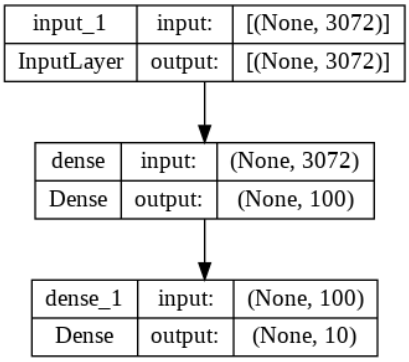
****فایل Segmentation_Training_BEiT.ipynb کد مربوط به آموزش BEiT است که با ارور کمبود حافظه مواجه شد و فایل Segmentation_BEiT_SceneParse150.ipynb کد مربوط به تقسیم‌بندی معنایی تصاویر با استفاده از مدل از پیش آموزش دیده BEiT است و تصاویر خروجی نیز در فولدر Images هستند****

3-2. طبقه‌بندی تصاویر

شبکه mlp :

در این بخش ابتدا داده های دیتاست cifar10 را که شامل 50000 داده آموزش و 10000 داده تست می‌شود را با استفاده از یک شبکه mlp ساده که در شکل 8.2 مشاهده میکنید طبقه‌بندی میکنیم.

برای اینکه مقایسه مناسبی بین دو مدل BEiT و mlp صورت بگیرد هردو مدل را 10 ایپاک و با batch_size=4 آموزش میدهیم.



شکل 8.2 ساختار شبکه **mlp** مورد استفاده برای طبقه‌بندی

```
Total data for traing: 50000, Train X shape: (50000, 32, 32, 3), Train y shape: (50000, 1)
Total data for test: 10000, Test X shape: (10000, 32, 32, 3), Test y shape: (10000, 1)
```

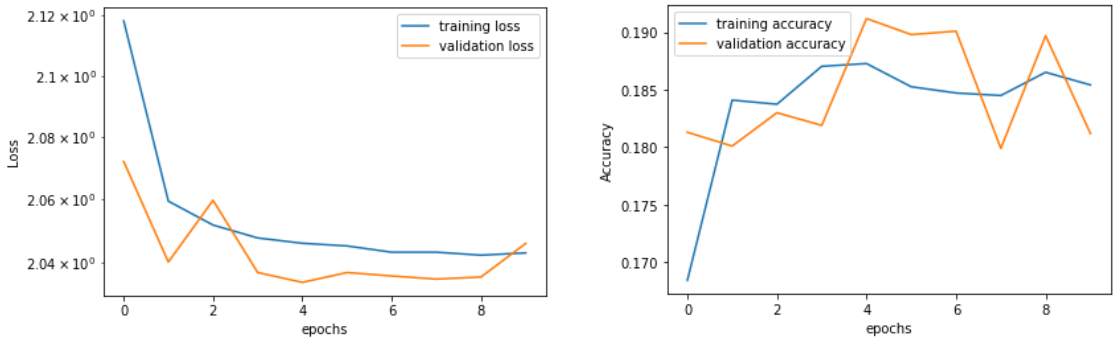
شکل 9.2 داده های دیتاست **cifar10**

همانگونه که در شکل 9.2 مشاهده میشود داده ها به دلیل RGB بودن، سه بعدی هستند و برای اینکه این داده ها را به عنوان ورودی به شبکه mlp بدهیم باید ابعاد آن را به یک بعد تغیر دهیم که در نهایت به صورت (n, 3*32*32) آنها را reshape میکنیم.

در مدل mlp از تابع هزینه cross entropy و بهینه ساز adam و تابع فعالسازی RelU در همه لایه ها و در لایه آخر از softmax استفاده میکنیم و در نهایت شبکه را آموزش میدهیم.

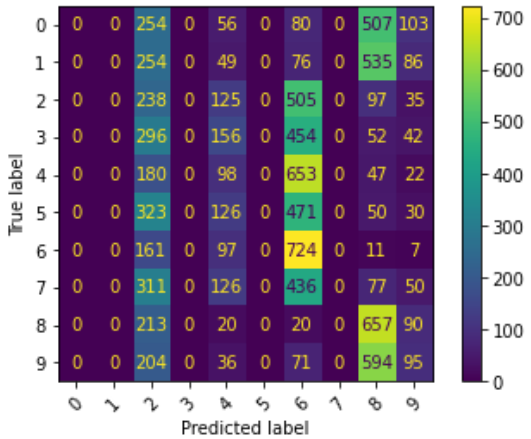
```
Epoch 1/10
12500/12500 [==============================] - 45s 3ms/step - loss: 2.1181 - accuracy: 0.1684 - val_loss: 2.0722 - val_accuracy: 0.1813
Epoch 2/10
12500/12500 [==============================] - 42s 3ms/step - loss: 2.0594 - accuracy: 0.1841 - val_loss: 2.0400 - val_accuracy: 0.1801
Epoch 3/10
12500/12500 [==============================] - 43s 3ms/step - loss: 2.0518 - accuracy: 0.1837 - val_loss: 2.0597 - val_accuracy: 0.1830
Epoch 4/10
12500/12500 [==============================] - 42s 3ms/step - loss: 2.0477 - accuracy: 0.1870 - val_loss: 2.0367 - val_accuracy: 0.1819
Epoch 5/10
12500/12500 [==============================] - 46s 4ms/step - loss: 2.0460 - accuracy: 0.1873 - val_loss: 2.0336 - val_accuracy: 0.1912
Epoch 6/10
12500/12500 [==============================] - 48s 4ms/step - loss: 2.0451 - accuracy: 0.1853 - val_loss: 2.0367 - val_accuracy: 0.1898
Epoch 7/10
12500/12500 [==============================] - 55s 4ms/step - loss: 2.0431 - accuracy: 0.1847 - val_loss: 2.0356 - val_accuracy: 0.1901
Epoch 8/10
12500/12500 [==============================] - 54s 4ms/step - loss: 2.0431 - accuracy: 0.1845 - val_loss: 2.0346 - val_accuracy: 0.1799
Epoch 9/10
12500/12500 [==============================] - 47s 4ms/step - loss: 2.0422 - accuracy: 0.1865 - val_loss: 2.0352 - val_accuracy: 0.1897
Epoch 10/10
12500/12500 [==============================] - 49s 4ms/step - loss: 2.0429 - accuracy: 0.1854 - val_loss: 2.0459 - val_accuracy: 0.1812
```

شکل 10.2 آموزش شبکه **mlp**



شکل 11.2 نتایج آموزش شبکه **mlp**

با توجه به نتایج به دست آمده در شکل 11.2 میبینیم که شبکه به خوبی آموزش دیده و training loss روند نزولی مناسبی را داشته، هرچند که مقدار loss در نهایت به مقدار زیاد 2.04 و دقت حدودا 18.5 درصد رسیده است که نشان دهنده این است که شبکه های mlp توانایی طبقه بندی داده های تصویری را ندارند. درنهایت نتایج ارزیابی این شبکه برروی داده های تست را در شکل 12.2 مشاهده میکنید.



	precision	recall	f1-score	support
0	0.00	0.00	0.00	1000
1	0.00	0.00	0.00	1000
2	0.10	0.24	0.14	1000
3	0.00	0.00	0.00	1000
4	0.11	0.10	0.10	1000
5	0.00	0.00	0.00	1000
6	0.21	0.72	0.32	1000
7	0.00	0.00	0.00	1000
8	0.25	0.66	0.36	1000
9	0.17	0.10	0.12	1000
accuracy			0.18	10000
macro avg	0.08	0.18	0.10	10000
weighted avg	0.08	0.18	0.10	10000

شکل 12.2 نتایج ارزیابی شبکه **mlp** بر روی داده های تست

در نهایت این شبکه به دقت 18 روی داده های تست رسده و با توجه به ماتریس آشفتگی آن، عملکرد ضعیفی در طبقه بندی داده داشته است.

مدل BEiT :

برای استفاده از مدل BEiT برای طبقه بندی تصاویر از دستور BeitForImageClassification استفاده کرده و از مدل از پیش آموزش دیده beit-base-patch16-224 استفاده میکنیم. در این مدل لایه FC با 10 نورون که تعداد کلاس های موجود در دیتاست میباشد به شبکه BEiT اضافه میشود و ما در این مسئله تنها همین لایه 10 نورونی را برروی داده های cifar10 آموزش میدهیم (برای کاهش حجم محاسبات و محدودیت های colab). در این مدل داده ها در ابعاد (3,224,224) در می‌آیند و در بخش tokenize کردن نیز تعداد patch ها 16 می‌باشد. قبل از آماده سازی مدل، داده های را با استفاده از BeitImageProcessor.from_pretrained("microsoft/beit-base-patch16-224") پیش پردازش میکنیم و سایز آنها را از 32*32 به 224*224 تغییر میدهیم. در نهایت برای آموزش این شبکه از Trainer که مربوط به گروه HuggingFace میباشد استفاده میکنیم.

```python
from transformers import TrainingArguments, Trainer

metric_name = "accuracy"

args = TrainingArguments(
    f"test-cifar-10",
    save_strategy="epoch",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=10,
    per_device_eval_batch_size=4,
    num_train_epochs=10,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model=metric_name,
    logging_dir='logs',
    report_to="wandb",
    remove_unused_columns=False,
)
```

شکل 13.2 پارامترهای **trainer** مدل **BEiT**

همانگونه که در شکل 13.2 مشاهده میشود برای این مدل نیز از batch_size = 4 و epochs = 10 استفاده شده است. Loss function استفاده شده در BEiT نیز cross entropy میباشد.

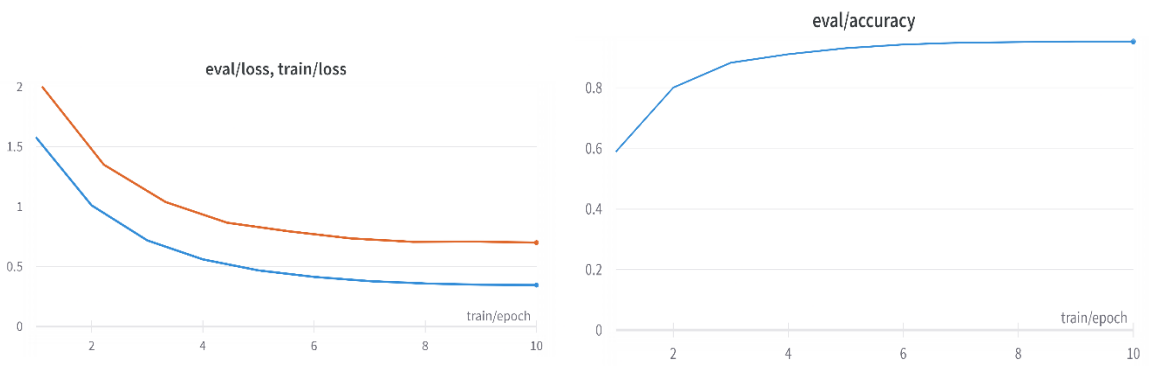
همانگونه که در شکل 14.2 مشاهده میشود در مجموع 7690 پارامتر برای آموزش شبکه وجود دارد که مربوط به وزن‌های لایه آخر که یک لایه FC با 10 نورون است، میباشد.

```
***** Running training *****
  Num examples = 4500
  Num Epochs = 10
  Instantaneous batch size per device = 10
  Total train batch size (w. parallel, distributed & accumulation) = 10
  Gradient Accumulation steps = 1
  Total optimization steps = 4500
  Number of trainable parameters = 7690
```

[4500/4500 11:34, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy
1	No log	1.578304	0.588000
2	2.001000	1.010285	0.800000
3	1.349800	0.717848	0.882000
4	1.038100	0.559435	0.910000
5	0.865100	0.467424	0.930000
6	0.792800	0.413084	0.942000
7	0.733900	0.377704	0.948000
8	0.705400	0.358354	0.950000
9	0.708100	0.347907	0.952000
10	0.699200	0.345098	0.952000

شکل 14.2 نتایج آموزش شبکه **BEiT**

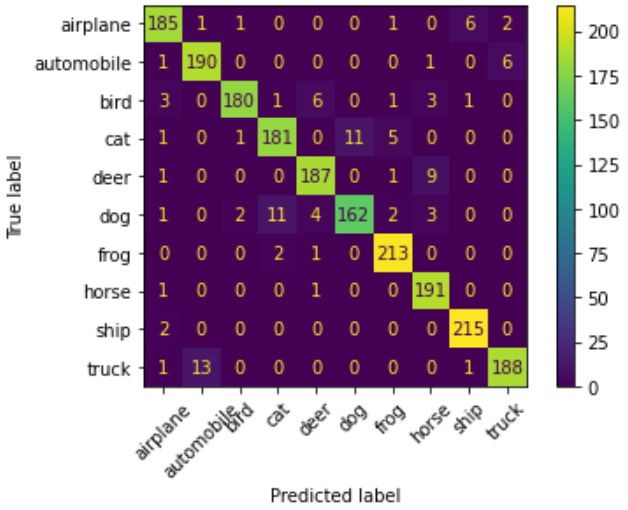


شکل 15.2 نمودار های دقت و خطا در فرایند آموزش **BEiT**

با توجه به نتایج به دست آمده که در شکل های 14.2 و 15.2 مشاهده میشوند، شبکه به خوبی آموزش دیده و نمودار خطا روند نزولی مناسبی داشته و در نهایت به مقدار مناسب 0.35 رسیده و به دقت بیش از 95درصد بررئی داده های آموزش دست پیدا کرده است.

در نهایت این مدل را برروی دادها های تست ارزیابی میکنیم که نتایج آن در شکل 16.2 قابل مشاهده است.

```
'test_loss': 0.32708892226219l8, 'test_accuracy': 0.946
```



شکل 16.2 ارزیابی مدل **BEiT** برروی داده های تست

در نهایت میبینیم که این شبکه به دقت 94.4 و خطای 0.3 رسیده است که نتایج بسیار مطلوبیست و با توجه به ماتریس آشفتگه بدست آمده نیز میبینیم که این مدل به خوبی میتواند داده های دیتاست cifar10 را طبقه بندی کند.

2-4. پرسش‌ها

در شبکه‌های CNN در کدام بخش مفهومی مانند مفهوم توجه اتفاق می‌افتد؟

در شبکه های CNN با global average گرفتن از feature map ها و سپس عبور آنها از softmax عملکردی مشابه مفهوم توجه (attention mechanism) صورت میگیرد.

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

در شبکه کانولوشن ارتباط بین لایه ها از نوع Feed Forward است اما در شبکه های Global attention و Local attention ارتباط بین لایه ها از نوع Recurrent است و بسته به نوع آن که cross attention یا self attention باشد، دیتای recurrentی که وارد لایه ها میشود میتواند از به ترتیب از لایه های بالاتر و یا همان لایه باشد.

درست یا نادرستی جملات زیر را تعیین کنید:

1. **در بخشی از لایه‌های تبدیل کننده‌ی Vanilla از شبکه‌ی LSTM استفاده شده است.**
   نادرست ، در این شبکه از بلوک attention استفاده می‌شود.

2. **یک تبدیل کننده از چند بلوک رمزگذار و چند بلوک رمزگشا تشکیل شده است.**
   نادرست ، در این شبکه تنها یک بلوک Encoder و یک بلوک Decoder وجود دارد و در آنها از چندین لایه attention بصورت موازی(multi-head attention) استفاده می‌شود.

3. **multi head attention از یک بخش توجه و چند لایه‌ی تمام متصل موازی تشکیل شده است.**

نادرست ، از چندین بخش Attention (بصورت موازی) تشکیل شده که در نهایت خروجی آنها وارد یک لایه Fully Connected می‌شود.

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

نادرست ، وجود این بلوک تنها باعث تسریع فرایند یادگیری می‌شود. این کار از طریق اختصاص دادن ضریب هایی متفاوت به ورودی‌ها با جایگاه های زوج و فرد صورت می‌گیرد.