تمرین سوم شناسایی الگو

بخش A

پس از دریافت فایل دیتاست، باید تمامی کاراکتر های یکتای درون متن را بیابیم:

```
letters = sorted(set(char for char in text))
letters_size = len(letters)
print(''.join(letters))
print(letters_size)
```

```
 !"'()*,-./0123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
79
```

در این بخش تمامی کاراکتر های درون متن را پیمایش می کنیم و درون مجموعه میریزیم تا یکتا باشند و سپس آن را سورت می کنیم.

سپس کاراکتر های یکتا و تعداد آن را چاپ میکنیم.

```
text = text.replace('-', ' ')
pattern = r"[^a-zA-Z0-9.\n ]"
new_text = re.sub(pattern, '', text)
lines = new_text.split('.')
joined_lines = " ".join(lines)
total_words = joined_lines.lower().split()
words = ['.']
for word in total_words:
    if word not in words:
        words.append(word)
```

در ادامه هر چی '–' در متن وجود دارد را حذف میکنیم و کاراکتر های اضافی موجود در متن به جز حروف الفبا و اعداد و ... را با استفاده از ریجکس حذف می کنیم.

سپس متن را با نقطه split می کنیم تا جمله ها بدست بیاید. در ادامه کلمات یکتای موجود در متن را با یک حلقه فور پیدا می کنیم که تعداد آن به شرح زیر است:

```
Total no. of lines:  30588
Total unique words:  17877
```

سپس باید دو دیکشنری برای تبدیل کلمه به عدد و برعکس ایجاد کنیم که برای این کار از یک حلقه فور برای پیمایش اعداد یکتا استفاده می کنیم که به ترتیب به هر کدام یک عدد نسبت میدهد.

```
stoi = {word: index for index, word in enumerate(words)}
itos = {index: word for word, index in stoi.items()}
print(len(itos))
```

```
17877
```

```
def encode(text):
    return [stoi[word] for word in text.lower().split()]

def decode(ints):
    return ' '.join([itos[index] for index in ints])

print(encode("you are a teacher"))
print(decode(encode("you are a teacher")))
```

حال دو تابع برای تبدیل متن به بردار عددی مناسب با کلمات آن و بر عکس نیاز داریم که از 2 دیکشنری مربوطه استفاده می کند.

```
[18, 7, 84, 3909]
you are a teacher
```

حال باید داده هایمان را به یک تنسور تبدیل کنیم که به سایز زیر است:

```
encoded_data = encode(joined_lines)
data = torch.tensor(encoded_data)
print(data.shape, data.dtype)
```

```
torch.Size([565963]) torch.int64
```

در ادامه یک تابع ایجاد می کنیم اگر ورودی آن train بود با داده های train کار میکند و در غیر این صورت با داده های validation.

```
def get_batch(split):
    if split == 'train':
        data = train_data
    else:
        data = val_data
    start_idx = torch.randint(0, len(data) - n_context, (batch_size,))

    x = torch.stack([data[idx:idx+n_context] for idx in start_idx]).to(device)
    y = torch.stack([data[idx+1:idx+n_context+1] for idx in start_idx]).to(device)
    return x, y
```

این تابع برای ایجاد هر batch یک عدد تصادفی از بین ایندکس های داده ها به تعداد batch_size تا انتخاب میکند و در آرایه مربوطه می ریزد.

در ادامه برای هر ایندکس که بالا به صورت رندوم انتخاب شد داده های ورودی از idx تا n_context تا جلوتر انتخاب می شود و برای خروجی نیز از یکی جلوتر تر idx شروع میشود که به تعداد batch_size است که خروجی آن به شکل زیر است:

```
inputs:
torch.Size([4, 8])
tensor([[   34,    84,  6227,  1878,    23,    95,   558,   524],
        [  294,  2773,   855,    13, 11823,  7908,    64,    13],
        [   13,   127,    48,  3207,    15,    18,    48,   196],
        [  264,    13,  3437,    12,   198,  6989,    52,  4249]],
       device='cuda:0')
targets:
torch.Size([4, 8])
tensor([[   84,  6227,  1878,    23,    95,   558,   524,    93],
        [ 2773,   855,    13, 11823,  7908,    64,    13, 11952],
        [  127,    48,  3207,    15,    18,    48,   196,    95],
        [   13,  3437,    12,   198,  6989,    52,  4249,    52]],
       device='cuda:0')
```

در ادامه برای هر کدام یک لایه fully connected از نوع Linear را تعریف می کنیم که ورودی(ویژگی) آن d_model و خروجی آن head_size است.

```
key = nn.Linear(d_model, head_size, bias=False)
query = nn.Linear(d_model, head_size, bias=False)
value = nn.Linear(d_model, head_size, bias=False)
```

(Key): بردار هایی که نشان دهنده اطلاعات موجود در هر توکن هستند.

(Query): بردارهایی که برای توجه به دیگر توکن ها استفاده می شوند.

(Value): اطلاعات واقعی که به اشتراک گذاشته می شود.

bias=False تنظیم می کند که هیچ بایاسی به این لایه ها اضافه نشود.

```
number_of_heads = d_model // head_size

key = nn.Linear(d_model, head_size, bias=False)
query = nn.Linear(d_model, head_size, bias=False)
value = nn.Linear(d_model, head_size, bias=False)
k = key(x)
q = query(x)
wei = torch.bmm(q, k.transpose(-2, -1))
wei = wei / (head_size ** 0.5)
```

در ادامه تعداد head را از تقسیم d_model بر اندازه هر head بدست می آوریم. این 3 لایه داده های ورودی را از فضای 768 بعدی به فضای 64 بعدی تبدیل می کنند.

مقدار wei که نشان دهنده توجه اولیه بین هر جفت توکن در توالی است را از ضرب داخلی k و q بدست میاوریم و در ادامه برای جلوگیری از رشد بیش ازحد مقادیر ان را بر 8 که مجذور 64 است تقسیم می کنیم که خروجی آن به شکل زیر است:

```
x dimention is:  torch.Size([1, 6, 768])
Dot product:
 tensor([[[ 2.6206, -1.4998,  1.5052,  3.6420,  4.3157,  1.1602],
         [-0.1661, -0.3164, -0.5046,  2.4511, -1.2876, -0.3483],
         [-2.0078,  3.3452,  2.6498,  1.1954, -1.5011,  1.4158],
         [-2.0249, -0.9772, -2.2888, -0.7966, -2.2507, -2.1566],
         [-1.6540, -0.7019, -0.8243,  2.3766, -4.2695, -5.9061],
         [-3.2899,  1.3121,  1.9521,  0.2806,  1.2954, -2.9294]]],
       grad_fn=<UnsafeViewBackward0>)
```

ایجاد mask برای موقعیت هایی که یک توکن نتواند به توکن های آینده در ادامه نگاه کند.

```
mask = torch.triu(torch.ones(n_context, n_context), diagonal=1).to(x.device)
mask = mask.masked_fill(mask == 1, float('-inf'))

# now fill mask with our attention scores
wei = wei + mask
print("(Scaling + Mask): \n", wei)
print('-'*80)
```

یک ماتریس بالا مثلثی ایجاد میکنیم که در بالای قطر آن 1 است که آن را با منفی بی نهایت جایگزین می کنیم و در ادامه با wei جمع میزنیم که به شکل زیر است:

```
(Scaling + Mask):
 tensor([[[ 0.3276,    -inf,    -inf,    -inf,    -inf,    -inf],
         [-0.0208, -0.0395,    -inf,    -inf,    -inf,    -inf],
         [-0.2510,  0.4182,  0.3312,    -inf,    -inf,    -inf],
         [-0.2531, -0.1221, -0.2861, -0.0996,    -inf,    -inf],
         [-0.2068, -0.0877, -0.1030,  0.2971, -0.5337,    -inf],
         [-0.4112,  0.1640,  0.2440,  0.0351,  0.1619, -0.3662]]],
       grad_fn=<AddBackward0>)
```

در لایه Normalization مقادیر را برای هر ویژگی در یک نمونه (در آخرین بعد) نرمال می کنیم که تابع __call__ آن به صورت زیر است:

```
def __call__(self, x):
  mean = x.mean(dim=-1, keepdim=True)
  variance = x.var(dim=-1, keepdim=True, unbiased=False)
  x_normalized = (x - mean) / torch.sqrt(variance + self.eps)
  self.out = self.gamma * x_normalized + self.beta
  return self.out
```

ابتدا میانگین هر نمونه در آخرین بعد محاسبه می شود سپس واریانس آن محاسبه می شود و در ادامه طبق فرمول نرمال سازی مقایر x نرمال سازی میشوند که مقادیر بتا و گاما در تابع __init__ کلاس مقداردهی شده اند.

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(d_model, head_size, bias=False)
        self.query = nn.Linear(d_model, head_size, bias=False)
        self.value = nn.Linear(d_model, head_size, bias=False)
        self.register_buffer("mask", torch.triu(torch.ones(n_context, n_context, dtype=torch.bool), diagonal=1))
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(head_size)

    def forward(self, x):
        batch_size, n_context, d_model = x.shape
        k = self.key(x)
        q = self.query(x)
        v = self.value(x)
        wei = torch.matmul(q, k.transpose(-2, -1)) / (k.shape[-1] ** 0.5)
        wei = wei.masked_fill(self.mask[:n_context, :n_context], float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        out = torch.matmul(wei, v)
        out = F.gelu(out)
        out = self.ln(out)

        return out
```

تعریف کلاس head که یک زیرمجموعه از مکانیزم Self-Attention است.در این کلاس از query key و value سه لایه خطی که برای تبدیل ورودی به بردارهای از query key و value استفاده می شوند.سپس از ماسک برای جلوگیری از دسترسی به اطلاعات آینده در پیش بینی استفاده می شود. در ادامه نرمال سازی با LayerNorm و از تابع فعال سازی GELU استفاده می کنیم.

```
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(num_heads * head_size, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([head(x) for head in self.heads], dim=-1)
        out = self.proj(out)
        out = self.dropout(out)
        return out
```

این کلاس چندین Head را با هم ترکیب می کند و به مدل اجازه می دهد که همزمان از جنبه های مختلف به کلمات نگاه کند. مثلاً یک head ممکن است به ترتیب زمانی توجه کند، درحالی که head دیگر به نقش گرامری کلمات توجه می کند.

در این کلاس خروجی تمامی head ها به هم متصل (torch.cat) شده و وارد یک لایه خطی (proj) می شود تا اطلاعات ترکیب شوند. و در ادامه برای جلوگیری از اورفیتینگ پس از ترکیب head ها dropout اعمال می شود.

```python
class FeedForward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, d_model):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, 4 * d_model),
            nn.GELU(),
            nn.Linear(4 * d_model, d_model),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)
```

این بخش روی هر کلمه (توکن) به صورت مستقل کار می کند و ویژگی های پیچیده تری را از اطلاعاتی که از مرحله attention گرفته را استخراج می کند.

ورودی به یک فضای ویژگی بزرگ تر (4 * d_model) برده می شود. این بزرگ تر شدن باعث می شود مدل بتواند روابط پیچیده تری را یاد بگیرد.

از تابع GELU و Dropout نیز استفاده کردیم.

```python
class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, d_model, n_head):
        super().__init__()
        head_size = d_model // n_head

        self.attn = MultiHeadAttention(n_head, head_size)
        self.ff = FeedForward(d_model)
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.ff(self.ln2(x))
        return x
```

این کلاس یک واحد کامل از مدل ترانسفورمر است که شامل دو مرحله اصلی است-Self Attention که به بررسی روابط بین کلمات می پردازد و FeedForward که درک عمیق تر از هر کلمه که هر بلاک اطلاعاتی که از مراحل قبلی به دست آورده را تقویت می کند.

اجزای آن:

Self-Attention: ابتدا داده ها وارد لایه MultiHeadAttention می شوند سپس خروجی نرمال سازی می شود (ln1) و با ورودی اولیه جمع می شود.(Residual Connection)

FeedForward: خروجی مرحله قبل وارد لایه feedforward می شود و خروجی نرمال سازی شده (ln2) و دوباره با ورودی جمع می شود.

Residual Connection: اضافه کردن ورودی اولیه به خروجی هر بخش برای حفظ اطلاعات اولیه و پایداری آموزش.

بخشی از خروجی با temp = 0.4

```
112.720341 M parameters
step 0: train loss 10.7403, val loss 10.7435
step 100: train loss 6.7458, val loss 6.9332
step 200: train loss 6.3471, val loss 6.6088
step 300: train loss 5.9497, val loss 6.3292
step 400: train loss 5.6397, val loss 6.1192
step 500: train loss 5.3833, val loss 6.0120
step 600: train loss 5.1959, val loss 5.9733
step 700: train loss 5.0439, val loss 5.9705
step 800: train loss 4.8763, val loss 6.0359
step 900: train loss 4.6733, val loss 6.0748
step 1000: train loss 4.5809, val loss 6.1573
step 1100: train loss 4.4150, val loss 6.1903
step 1200: train loss 4.2868, val loss 6.2442
step 1300: train loss 4.1410, val loss 6.2919
step 1400: train loss 4.0431, val loss 6.4175
step 1499: train loss 3.9592, val loss 6.4065
. of the game in a aches voice was living he left arm which was kutuzovs king triumph over he began speaking to duck in white borzoi dashed downwards
```

بخشی از خروجی با temp = 1

```
112.720341 M parameters
step 0: train loss 9.9209, val loss 9.9221
step 100: train loss 6.1235, val loss 6.3681
step 200: train loss 5.6394, val loss 6.0577
step 300: train loss 5.3535, val loss 5.9374
step 400: train loss 5.1503, val loss 5.8712
step 500: train loss 4.9660, val loss 5.8736
step 600: train loss 4.8203, val loss 5.9168
step 700: train loss 4.7259, val loss 5.9314
step 800: train loss 4.6057, val loss 5.9935
step 900: train loss 4.4456, val loss 6.0408
step 1000: train loss 4.4201, val loss 6.0863
step 1100: train loss 4.2546, val loss 6.1272
step 1200: train loss 4.1311, val loss 6.2096
step 1300: train loss 4.1080, val loss 6.2587
step 1400: train loss 3.9654, val loss 6.3275
step 1499: train loss 3.9596, val loss 6.4124
. of which seemed as it has hardly have perished he left moscow or have kutuzovs report the club or tomorrow they say against any violence would be de
```

بخشی از خروجی با temp = 2

```
112.720341 M parameters
step 0: train loss 9.8144, val loss 9.8150
step 100: train loss 6.2789, val loss 6.4600
step 200: train loss 5.8267, val loss 6.1424
step 300: train loss 5.5421, val loss 5.9797
step 400: train loss 5.3676, val loss 5.9023
step 500: train loss 5.2613, val loss 5.8707
step 600: train loss 5.1989, val loss 5.8895
step 700: train loss 5.0946, val loss 5.8763
step 800: train loss 5.0348, val loss 5.9252
step 900: train loss 4.9558, val loss 5.9100
step 1000: train loss 4.8543, val loss 5.9081
step 1100: train loss 4.7541, val loss 5.9219
step 1200: train loss 4.6687, val loss 5.9519
step 1300: train loss 4.6236, val loss 5.9972
step 1400: train loss 4.5125, val loss 6.0405
step 1499: train loss 4.6655, val loss 6.0710
. of which seemed as it has had never left them and bowing him and kutuzovs company which he had begun and even against any violence which was lit him
```

بخش B

```
filename = "/content/all-data.csv"
df = pd.read_csv(filename, encoding='latin-1', names=["sentiment", "text"])

positive_samples = df[df['sentiment'] == 'positive'].sample(n=600, random_state=10)
negative_samples = df[df['sentiment'] == 'negative'].sample(n=600, random_state=10)
neutral_samples = df[df['sentiment'] == 'neutral'].sample(n=600, random_state=10)

X_train = [positive_samples.iloc[:300], negative_samples.iloc[:300], neutral_samples.iloc[:300]]
X_test = [positive_samples.iloc[300:], negative_samples.iloc[300:], neutral_samples.iloc[300:]]
```

برای این سوال ابتدا فایل دیتاست را دریافت کرده و درون متغیر می ریزیم. سپس 600 داده از هر کلاس را جدا می کنیم.

در ادامه برای داده های Train از هر کلاس 300 تای اول را جدا میکنیم و برای داده های test 300 تا دوم انتخاب می شوند.

```
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",
    quantization_config=bnb_config,
)
```

در ادامه یک مدل از کلاس AutoModelForCausalLM ایجاد می کنیم که مدل از مسیری که در متغیر model_name ذخیره شده بارگذاری میشود. مدل device که قرار است باهاش کار کند را به صورت اتوماتیک انتخاب میکند و در ادامه پارامتر کوانتیزاسیون رو که بالاتر ست شده به مدل می دهیم.

```
tokenizer = AutoTokenizer.from_pretrained(model_name,trust_remote_code=True)
```

سپس Tokenizer که مسئول تبدیل متن به توکن ها است را از کلاس AutoTokenizer که به صورت خودکار توکنایزر مرتبط با مدل را بارگذاری می کند ایجاد میکنیم. همانند مدل توکنایزر نیز از مسیری که در model_name مشخص شده بارگذاری می شود.

دقت نهایی مدل به شرح زیر است:

```
Accuracy: 0.708
Accuracy for label 0: 0.910
Accuracy for label 1: 0.677
Accuracy for label 2: 0.537

Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.91      0.79       300
           1       0.64      0.68      0.66       300
           2       0.82      0.54      0.65       300

    accuracy                           0.71       900
   macro avg       0.72      0.71      0.70       900
weighted avg       0.72      0.71      0.70       900


Confusion Matrix:
[[273  27   0]
 [ 62 203  35]
 [ 52  87 161]]
```

بخش C

```
remaining_data = df[~df.index.isin(pd.concat([X_train , X_test] ,axis=0).index)]

positive_eval = remaining_data[remaining_data['sentiment'] == 'positive'].sample(n=50, random_state=10)
negative_eval = remaining_data[remaining_data['sentiment'] == 'negative'].sample(n=50, random_state=10)
neutral_eval = remaining_data[remaining_data['sentiment'] == 'neutral'].sample(n=50, random_state=10)
eval_data = pd.concat([positive_eval, negative_eval, neutral_eval])
X_eval = eval_data.reset_index(drop=True)
```

برای این بخش داده هایی که در مرحله قبل وجود نداشتند را انتخاب میکنیم و در ادامه برای هر کلاس 50 داده انتخاب میکنیم. سپس آنها را ترکیب می کنیم و ایندکس های جدیدی برای داده ها ست می کنیم.

در ادامه باید کانفیگ لورا را ست کنیم که rank آن را 64 میزاریم و lora_alpha فاکتور مقیاس بندی و میزان dropout را ست می کنیم.

```
peft_config = LoraConfig(
    r=64,
    lora_alpha=16,
    lora_dropout=0,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
                    "gate_proj", "up_proj", "down_proj",],
)
```

در این بخش یک trainer برای آموزش مدل استفاده میکنیم که از کلاس SFTTrainer است و مدلی که قبلا از از طریق LoRAپیکربندی شده است را به عنوان ورودی می دهیم. سپس کانفیگ peft و tokenizer را که در بخش قبل مشخص کردیم به مدل می دهیم.

```
trainer = SFTTrainer(
    model=model,
    args=training_arguments,
    train_dataset=train_data,
    eval_dataset=eval_data,
    peft_config=peft_config,
    dataset_text_field="text",
    tokenizer=tokenizer,
    packing=False,
    dataset_kwargs={
        "add_special_tokens": False,
        "append_concat_token": False,
    },
)
```

در ادامه پس از اجرای این کد به این نتیجه میرسیم:

```
trainer.train()
```

[336/336 43:52, Epoch 2/3]

Epoch	Training Loss	Validation Loss
0	0.928200	0.873886
2	0.512900	0.756812

```
TrainOutput(global_step=336, training_loss=0.7805818205788022, metrics={'train_runtime': 2643.5043, 'train_samples_per_second': 1.021, 'train_steps_per_second': 0.127, 'total_flos': 1.0159126312771584e+16, 'train_loss': 0.7805818205788022, 'epoch': 2.986666666666667})
```

و دقت نهایی برابر است با :

```
Accuracy: 0.720
Accuracy for label 0: 0.907
Accuracy for label 1: 0.710
Accuracy for label 2: 0.543

Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.91      0.80       300
           1       0.65      0.71      0.68       300
           2       0.84      0.54      0.66       300

    accuracy                           0.72       900
   macro avg       0.74      0.72      0.71       900
weighted avg       0.74      0.72      0.71       900


Confusion Matrix:
[[272  28   0]
 [ 57 213  30]
 [ 49  88 163]]
```

خلاصه مقاله LoRA: Low-Rank Adaptation of Large Language Models

این مقاله یک روش جدیدی برای بهبود کارایی LLM ها ابداع میکند به صورتی که به جای تنظیم کامل پارامترها از ماتریس های مرتبه پایین برای بهینه سازی تغییرات پارامترهای لایه های Dense در مدل های Transformer استفاده می کند.

رویکرد کلی مدل:

- تعداد پارامترهای قابل تنظیم را هزاران برابر کاهش می دهد.
- حافظه GPU موردنیاز را تا سه برابر کاهش می دهد.
- بدون کاهش کیفیت مدل، سرعت آموزش را افزایش می دهد.
- برای کاربردهای متنوعی مانند GPT-3 و RoBERTa اثبات شده است.

چالش اصلی LLM ها این است که آموزش دادن دوباره آن ها یا همان فاین تیونینگ به منابع زیادی مثل کارت های گرافیک قوی و حافظه بالا نیاز دارد و تغییر همه وزن های مدل هم زمان زیادی میگیرد و هم هزینه زیادی دارد. حال به تعریف کلی لورا می پردازیم:

این یک روش هوشمندانه است که به ما اجازه می دهد مدل های بزرگ را خیلی سریع تر و با هزینه کمتر آموزش دهیم. به جای تغییر کل مدل باید لایه هایی سبک و افزایشی به مدل اضافه کنیم که تنها این لایه ها در فرآیند آموزش تغییر کنند. به این ترتیب وزن های اصلی مدل دست نخورده باقی می مانند.

مزای این روش:

- صرفه جویی در منابع: دیگر نیازی نیست که برای هر کاری مدل را کامل آموزش دهیم.
- انعطاف پذیری: می توان از این روش برای کاربردهای مختلف استفاده کرد.
- حفظ دانش قبلی: مدل پایه مثل LLaMA هر چیزی که از قبل یاد گرفته را فراموش نمی کند.

استفاده LoRA روی LLaMA چطور انجام می شود؟

در این روش مدل LLaMA به عنوان یک مدل پایه استفاده می شود. سپس پارامترهای LoRA به آن اضافه می شوند و فقط همین پارامترها آموزش داده می شوند. این باعث می شود زمان و منابع مورد نیاز کاهش پیدا کند.

محمد حقیقت – 403722042

برای رفع برخی ایرادات و ابهامات از Chatgpt استفاده شده است.