به نام خدا

تمرین سری اول فهم زبان

سامان محمدی رئوف – ۴۰۲۱۳۱۰۶۳

فهرست

مقدمه

در این تمرین هدف پیاده‌سازی رویکردی برای تعیین وزن عروضی اشعار با استفاده از دریافت یک مصرع از آن به عنوان ورودی است. به عنوان برچسب برای یادگیری مدل، از دنباله‌های وزنی آن مصرع مانند "مفاعیلن مفاعیلن مفاعیلن فعولن" استفاده شده است.

در بخش الف، با در نظر گرفتن این مسئله به عنوان یک مسئله دسته‌بندی بدان پرداخته شده است. در بخش ب، سعی شده است با در نظر گرفتن مسئله به عنوان یک مسئله دنباله به دنباله مسئله حل شود. در این راستا از ساختار انکدر، دیکدر با معماری RNN استفاد شده. در بخش ج، همین رویکرد دنباله به دنباله با استفاده از ساختار Transformer پیاده‌سازی شده است.

در نهایت در بخش د، برای بررسی اینکه نتیجه بهتری خواهیم گرفت یا خیر، از یک مدل از پیش آموزش دیده mt5 استفاده شده و روی این تسک Fine-tune شده است. نتایج بخش د، نشان می‌دهند که این رویکرد از رویکرد سایر بخش‌ها بنظر بهتر است.

ساختار فایل‌های ارسالی، شامل ۴ فولدر می‌باشد که در هر کدام، کد مربوطه و فایل csv حاوی پیش‌بینی‌ها برای مجموعه داده‌های تست آورده شده است. همان‌طور که اشاره شد، بهترین نتایج برای حالت fine-tune کردن mt5 می‌باشد.

در ادامه به هر یک از این بخش‌ها به صورت کامل پرداخته شده است.

الف) رویکرد دسته‌بندی

پیش‌پردازش و آماده‌سازی

برای اینکه این مسئله را بتوان به شکل یک مسئله دسته‌بندی حل نمود، باید ابتدا تعداد کلاس‌ها و تعداد نمونه‌های متعلق به هر یک از کلاس‌ها را بدست آورد.

با لود کردن دیتای train, val, test ابتدا تعداد کلاس‌های موجود را بررسی می‌کنیم.

```
unique_classes = train_df["metre"].nunique()
print(f"Number of unique classes: {unique_classes}")
```

```
Number of unique classes: 48
```

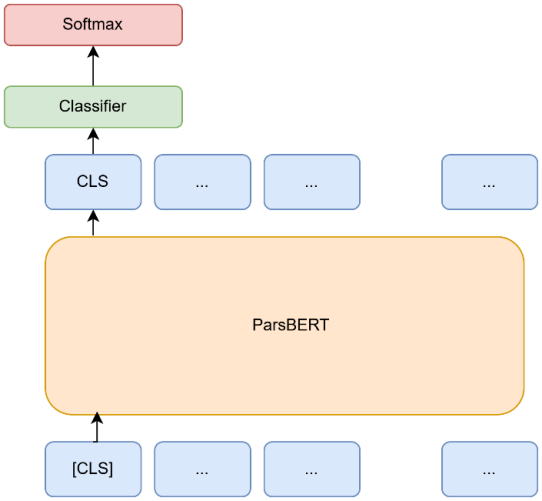
در کل ۴۸ حالت خروجی داریم. سپس تعداد نمونه‌های موجود در هر یک از کلاس‌ها را در می‌آوریم.

metre	
149803	مفاعیلن مفاعیلن فعولن
135237	فعولن فعولن فعولن فعل
116341	فاعلاتن فاعلاتن فاعلن
73436	فعلاتن مفاعلن فعلن
35149	فاعلاتن فاعلاتن فاعلاتن فاعلن
33924	مفاعلن فعلاتن مفاعلن فعلن
27574	مفعول فاعلات مفاعیل فاعلن
25714	مفعول مفاعیل مفاعیل فعل
25262	مفعول مفاعلن فعولن
22069	فعلاتن فعلاتن فعلاتن فعلن
15813	مفتعلن مفتعلن فاعلن
15170	مفاعیلن مفاعیلن مفاعیلن مفاعیلن
14151	مفعول مفاعیل مفاعیل فعولن
7579	فعلاتن فعلاتن فعلن
7330	مستفعلن مستفعلن مستفعلن مستفعلن
6953	مفعول فاعلاتن مفعول فاعلاتن
6650	مفعول مفاعیلن مفعول مفاعیلن
4871	مفتعلن فاعلن مفتعلن فاعلن
4235	مفتعلن مفاعلن مفتعلن مفاعلن
4091	فعولن فعولن فعولن فعولن
3065	مفتعلن فاعلات مفتعلن فع
2722	مفعول مفاعلن مفاعیلن
2689	فعلات فاعلاتن فعلات فاعلاتن
1576	فعلاتن فعلاتن فعلاتن فع
...	
18	فعلاتن مفاعلن فعلاتن
12	فاعلات فع فاعلات فع
10	مفعول مفاعیل فاعلن
4	مفاعلن فع مفاعلن فع

در شکل صفحه پیشین قابل مشاهده است که تعداد نمونه‌های برخی کلاس‌ها بسیار زیاد و برخی دیگر بسیار کم است. این مشکل Imbalanced بودن مجموعه داده‌ها، روی نتایج تاثیر خواهد گذاشت؛ و باعث می‌شود که مدل به سمت پیش‌بینی کلاس با تعداد نمونه‌های بیشتر سوق داده شود و کلاس‌های دارای نمونه کم را پیش‌بینی نکند. یکی از راهکارهای مقابله با این مشکل استفاده از وزن‌دهی به تاثیر هر کلاس روی گرادیان برای بروزرسانی وزن‌های مدل است که در ادامه از آن استفاده شده است.

همچنین برای بدست آوردن embedding مناسب جهت ورودی دادن به classifier از یکی از اعضای خانواده مدل BERT به نام ParsBERT استفاده شده است. این مدل به شکل خاص برای زبان فارسی آموزش داده شده است. از این جهت استفاده از pre-trained این شبکه می‌تواند موثر باشد.

در کل می‌توان معماری و ایده مدنظر برای حل این سوال را به شکل زیر مشاهده نمود.



با این توضیحات، از کد زیر برای لود مدل parsBERT استفاده کرده‌ام.

```
config = AutoConfig.from_pretrained("HooshvareLab/bert-base-parsbert-uncased")
tokenizer = AutoTokenizer.from_pretrained("HooshvareLab/bert-base-parsbert-uncased")
model = AutoModel.from_pretrained("HooshvareLab/bert-base-parsbert-uncased")
```

و همچنین مقادیر poem_text و metre را به شکل زیر در train_texts و train_labels ذخیره‌سازی کردم.

```
train_texts = train_df["poem_text"].tolist()
train_labels = train_df["metre"].astype('category').cat.codes.values
```

با توجه به رویکردی که پیش‌تر توضیح داده شد، در ابتدای هر یک از جملات ورودی یک توکن [CLS] اضافه شده است. استفاده از این توکن یک روش رایج برای استفاده از مدل‌های BERT جهت دسته‌بندی است. این توکن‌ها می‌توانند نماینده‌ای کلی از متن ورودی باشند.

```
train_texts = ["[CLS] " + text for text in train_texts]
```

پس متون (مصرع‌های) ورودی به شکل زیر شدند.

```
['[CLS] شرح بسیاری بگفت از کائنات',
 '[CLS] چه جای رفتن باغ است و گشتن بستان',
 '[CLS] پای خود آرم برون وبر پرم',
 '[CLS] چو گوهر برآمود زنگی به تاج',
```

در گام بعدی، نیاز است تا ورودی را توکنایز کنیم. این کار هر متن را به توکن‌هایی تبدیل کرده و آنان را به فرمت عددی در می‌آورد. زیرا شبکه‌های عصبی با این اعداد کار می‌کنند.

چون مدل parsBERT روی متون فارسی آموزش دیده است، بازنمایی عددی معنادارتری از توکن‌های مصرع‌های فارسی می‌دهد. در هنگام توکنایز کردن، امکان truncation , padding را True قرار دادم. زیرا برای اینکه تمامی ورودی‌ها طول یکسانی داشته باشند این کار نیاز است. توکن‌هایی که به جهت یکسان سازی طول ورودی‌ها اضافه می‌شوند، در ادامه قابل تشخیص و حذف هستند. بدین ترتیب با استفاده از کد زیر دنباله ورودی به شکل دنباله ای ۱۲ تایی از توکن‌ها می‌شود.

```
inputs = tokenizer(train_texts, return_tensors="pt", padding=True,
truncation=True, max_length=12)
labels = torch.tensor(train_labels, dtype=torch.long)
```

چون برای اجرای این کد از محیط google colab استفاده شده، تمامی مقادیر را به روی device می‌برم که در اینجا پردازنده گرافیکی دارای cuda می‌باشد.

```
inputs = {k: v.to(device) for k, v in inputs.items()}
labels = labels.to(device)
```

خروجی توکنایزر، که ورودی مدل هستند و اینجا با inputs نام گذاری کردیم، دارای دو مقدار input_ids و attention_mask می‌باشد. مقادیر input_ids همان اعداد متناظر با توکن‌های داخل هر دنباله را نشان می‌دهند. مقادیر attention_mask نشان می‌دهند که یک توکن، توکن واقعی و مربوط به متن است و یا برای یکسان سازی طول دنباله اضافه شده است. بدین شکل این دو دسته توکن از هم متمایز می‌گردند. برای نهایی سازی داده‌های آموزشی، آنان را به شکل زیر در batchهای ۵۱۲ تایی قرار می‌دهیم. (با اینکه بزرگ بودن مقدار batch_size باعث کمتر شدن تعداد بروزرسانی‌ها و بزرگتر شدن گرادیان می‌شود، اما به دلیل حجم داده‌ها و مدت زمان آموزش، آزمایش batch_sizeهای کوچک‌تر امکان‌پذیر نبود).

```
train_dataset = TensorDataset(inputs['input_ids'], inputs['attention_mask'],
labels)
train_dataloader = DataLoader(train_dataset, batch_size=512, shuffle=True)
```

به دلیل اینکه تعداد نمونه کلاس‌های مختلف برابر نیست و برای برخی کلاس‌ها تعداد نمونه‌های آموزشی بسیار کم است، وزن‌هایی برای تاثیر خطای آنان در بروزرسانی تعریف می‌کنیم. هر چه تعداد نمونه‌های یک کلاس کمتر باشد، نیاز است تا این وزن بیشتر باشد. از این روی این وزن‌ها نسبت عکس با تعداد نمونه‌های کلاس دارند. بنابراین وزن را برابر با مقدار 1/class_samples_count قرار می‌دهیم.

```
class_counts = train_df['metre'].value_counts().sort_index()
class_weights = 1.0 / torch.tensor(class_counts.values, dtype=torch.float)
class_weights = class_weights.to(device)
```

آماده‌سازی داده‌های validation و test نیز به شکل مشابه انجام می‌شود.

حال به اضافه کردن لایه classifier به مدل می‌پردازیم. این لایه، یک fully connected است که سایز خروجی را از بازنمایی خروجی مدل parsBERT به تعداد کلاس‌ها تصویر می‌کند.

```
hidden_size = model.config.hidden_size
classifier = nn.Linear(hidden_size, unique_classes).to(device)
```

فرآیند آموزش

برای آموزش مدل، دو رویکرد مختلف را مورد ارزیابی قرار دادم. در حالت اول، فقط classifier را آموزش دادم که منجر به نتایج چندان مطلوبی نشد (حدود ۴۵ درصد f1). بنظر freeze کردن لایه‌های مدل parsBERT به صورت کامل باعث می‌شد که بازنمایی‌های مناسبی برای پیش‌بینی این وظیفه پیدا نشوند.

پس در رویکرد دوم از دو learning rate متفاوت برای بروزرسانی وزن‌های شبکه استفاده کردم. لایه classifier آموزش ندیده است پس باید نرخ بروزرسانی و یادگیری بالایی داشته باشد. اما در طرف مقابل مدل parsBERT قبلا روی داده‌های زیادی آموزش دیده شده و نباید نرخ یادگیری آن را بالا در نظر گرفت چرا که ممکن است باعث از دست دادن مزیت از پیش آموزش دیده بودن آن گردد.

```
classifier_optimizer = optim.Adam(list(classifier.parameters()), lr=1e-4,
weight_decay=0.01)
bert_model_optimizer = optim.Adam(list(model.parameters()), lr=3e-5,
weight_decay=0.01)
```

در هنگام آموزش از CrossEntropyLoss به عنوان تابع زیان استفاده شده که وزن خطا برای هر کلاس را نیز به طور جداگانه‌ای در نظر می‌گیرد.

```
criterion = nn.CrossEntropyLoss(weight=class_weights)
```

حال متغیرهای زیر را تعریف کرده و در نظر می‌گیریم. Num_epoch تعداد اپاک‌های آموزش را بیان می‌کند. مقدار validation_interval بیان می‌کند که بعد از هر چند اپاک، یکبار داده‌های validation مورد ارزیابی قرار گیرند. این کار برای تعریف مکانیزیم early_stopping مورد نیاز است.

```
num_epochs = 5
validation_interval = 700
```

در زیر تعداد دفعاتی که val_loss حساب می‌شود و اگر بهبود نیافته بود، آموزش متوقف می‌گردد ذکر شده است. در اینجا این مقدار ۵ در نظر گرفته شده. با توجه به تعداد batch های هر اپاک، در هر اپاک دوبار val_loss حساب می‌شود و بنابراین این عدد ۵ معادل حدود ۲.۵ اپاک است. با اینکه این امکان وجود داشت که بعد از هر اپاک یکبار val_loss را حساب کرد، اما از این جهت که زمان بسیار زیادی برای آموزش نیاز است تصمیم گرفتم که در هر اپاک دوبار این مقدار محاسبه گردد تا در صورت بهبود نیافتن، زودتر آموزش متوقف شود.

```
best_val_loss = float('inf')
patience = 5
epochs_without_improvement = 0
```

در حلقه زیر برای آموزش، در هر اپاک ابتدا model , classifier در حالت training گذاشته می‌شوند. علت اینکار این است که جلوتر در انتهای هر حلقه و برای بررسی عملکرد آنان روی داده‌های validation داریم شبکه را روی حالت eval قرار می‌دهیم؛ پس برای ادامه آموزش نیاز است تا دوباره آنان را به فرمت train در آوریم.

در ادامه گرادیان هر دو بخش model , classifier را برابر صفر قرار می‌دهیم. با ورودی دادن input_ids یعنی همان توکن‌ها و attention mask مربوطه‌شان، مدل بازنمایی‌ها را می‌دهد. اما ما بازنمایی cls token را می‌خواهیم که توکن اول می‌باشد. با گرفتن این بازنمایی، آن را به classifier می‌دهیم. این لایه بازنمایی‌ها را به ابعاد تعداد کلاس‌ها تصویر می‌کند. سپس loss را با استفاده از مقایسه کلاس خروجی مدل (کلاس با بیشترین مقدار احتمال) با کلاس درست بدست می‌آوریم و این loss را باز انتشار می‌دهیم.

```
for epoch in range(num_epochs):
    model.train()
    classifier.train()
    total_loss = 0
    for i, batch in enumerate(train_dataloader):
        input_ids, attention_mask, batch_labels = batch

        classifier_optimizer.zero_grad()
        bert_model_optimizer.zero_grad()

        try:
            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            cls_token_output = outputs.last_hidden_state[:, 0, :]
            logits = classifier(cls_token_output)

            loss = criterion(logits, batch_labels)
            total_loss += loss.item()
            print(f"Epoch {epoch+1}/{num_epochs}, Batch
{i+1}/{len(train_dataloader)}, Loss: {loss.item()}")

            loss.backward()

            classifier_optimizer.step()
            bert_model_optimizer.step()
```

بعد از این فرایند و هر زمان که به تعداد کافی (در اینجا ۷۰۰ تا batch) مدل آموزش دید و بروزرسانی شد، یکبار loss را روی مجموعه داده‌های validation حساب می‌کنیم. برای اینکار، classifierو model را روی حالت eval قرار می‌دهیم و به شکل مشابه با حلقه قبلی، خروجی مدل را حساب کرده و loss را حساب می‌کنیم.

برای مکانیزیم early stopping بهترین مقدار بدست آمده برای val_loss را ذخیره می‌کنیم. مقدار val_loss را هر بار با آن مقایسه می‌کنیم، اگر بهتر شده بود آن را جایگزین می‌کنیم و اگر بهتر نشده بود در نظر میگیریم که برای یک بار بهبودی در نتایج دیده نشده. اگر این عدم مشاهده بهبود به ۵ بار برسد، آموزش متوقف می‌گردد.

```
        if i % validation_interval == 0:
            model.eval()
            classifier.eval()
            with torch.no_grad():
                total_val_loss = 0
                correct = 0
                for i, batch in enumerate(validation_dataloader):
                    input_ids, attention_mask, batch_labels = batch

                    outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
                    cls_token_output = outputs.last_hidden_state[:, 0, :]
                    logits = classifier(cls_token_output)

                    loss = criterion(logits, batch_labels)
                    total_val_loss += loss.item()

                    _, predicted = torch.max(logits, dim=1)
                    correct += (predicted == batch_labels).sum().item()

                accuracy = correct / len(validation_dataloader.dataset)
                val_loss = total_val_loss / len(validation_dataloader)
                print(f"Epoch {epoch+1}/{num_epochs}, Validation Loss:
{val_loss:.4f}, Accuracy: {accuracy:.4f}")

                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    epochs_without_improvement = 0
                    torch.save(classifier.state_dict(), 'best_model.pth')
                else:
                    epochs_without_improvement += 1
                    if epochs_without_improvement >= patience:
                        print(f"Early stopping at epoch {epoch+1} due to no
improvement on validation set")
                        break
```

ارزیابی

حال و بعد از آموزش، می‌خواهیم تعدادی از خروجی‌ها را روی validation set بررسی کنیم. برای اینکار، دقیقا مشابه به قسمت‌های پیشین، در یک حلقه هر batch از داده‌های validation را به مدل ورودی می‌دهیم و خروجی cls token را می‌گیریم و کلاس با بالاترین مقدار را باز می‌گردانیم. برای 20 sample این کار را انجام داده و با استفاده از tokenizer.decode مقادیر ورودی را به متن تبدیل کرده و در کنار کلاس پیش‌بینی شده و کلاس درست چاپ می‌کنیم.

```
model.eval()
classifier.eval()
```

```
metre_categories = validation_df["metre"].astype('category').cat.categories

all_predictions = []
all_labels = []
samples = []
with torch.no_grad():
    for batch in validation_dataloader:
        input_ids, attention_mask, batch_labels = [x.to(device) for x in batch]

        validation_outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
        validation_cls_token_output = validation_outputs.last_hidden_state[:, 0,
:]
        validation_logits = classifier(validation_cls_token_output)

        predictions = torch.argmax(validation_logits, dim=1)  # like softmax
        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(batch_labels.cpu().numpy())

        # Get 20 samples
        if len(samples) < 20:
            for i in range(len(input_ids)):
                if len(samples) < 20:
                    sample = {
                        'input': input_ids[i].cpu().numpy(),
                        'attention_mask': attention_mask[i].cpu().numpy(),
                        'prediction': predictions[i].cpu().numpy(),
                        'label': batch_labels[i].cpu().numpy()
                    }
                    samples.append(sample)

        if len(samples) == 20:
            break

for i, sample in enumerate(samples):
    print(f"Sample {i+1}:")
    print(f"Input: {tokenizer.decode(sample['input'],
skip_special_tokens=True)}")
    print(f"Attention Mask: {sample['attention_mask']}")
    print(f"Prediction: {metre_categories[int(sample['prediction'])]}")
    print(f"Label: {metre_categories[int(sample['label'])]}")
    print("------------------------")
```

به عنوان مثال داریم:

```
Sample 1:
Input: که من با چو و با تو را نمیدانم
Attention Mask: [1 1 1 1 1 1 1 1 1 1 1]
Prediction: مفاعیلن مفاعیلن مفاعیلن مفاعیلن
Label: مفاعیلن مفاعیلن مفاعیلن مفاعیلن
------------------------
Sample 2:
Input: نیست ان جز حیله نفس لیم
Attention Mask: [1 1 1 1 1 1 1 1 0 0 0]
Prediction: فعلاتن مفاعلن فعلن
Label: فاعلاتن فاعلاتن فاعلن
------------------------
Sample 3:
Input: پراپرند زطمع بازو ، جغدکان
Attention Mask: [1 1 1 1 1 1 1 1 1 1 1]
Prediction: مفعول فاعلات مفاعیل فاعلن
Label: مفاعلن فعلاتن مفاعلن فعلن
------------------------
Sample 4:
Input: ولی شد چار دای از چار یارش
Attention Mask: [1 1 1 1 1 1 1 1 1 1 0 0]
Prediction: مفاعیلن مفاعیلن فعولن
Label: مفاعیلن مفاعیلن فعولن
```

حال به بخش ارزیابی و محاسبه معیارها می‌رسیم. برای این بخش، ابتدا به شکل مشابه ولی برای تمام نمونه‌های validation مقادیر پیش‌بینی را بدست می‌آوریم.

```
model.eval()
classifier.eval()

all_predictions = []
all_labels = []
with torch.no_grad():
    for batch in validation_dataloader:
        input_ids, attention_mask, batch_labels = [x.to(device) for x in batch]

        validation_outputs = model(input_ids=input_ids,
attention_mask=attention_mask)
        validation_cls_token_output = validation_outputs.last_hidden_state[:, 0,
:]
        validation_logits = classifier(validation_cls_token_output)

        predictions = torch.argmax(validation_logits, dim=1)
        all_predictions.extend(predictions.cpu().numpy())
```

```
        all_labels.extend(batch_labels.cpu().numpy())
```

حال با استفاده از توابع recall_score, accuracy_score, precision_score, f1_score از کتابخانه sklearn.metrics مقادیر معیارها را محاسبه می‌کنیم.

```
from sklearn.metrics import recall_score, f1_score, accuracy_score,
precision_score

accuracy = accuracy_score(all_labels, all_predictions)
f1 = f1_score(all_labels, all_predictions, average='macro')
recall = recall_score(all_labels, all_predictions, average='macro')
precision = precision_score(all_labels, all_predictions, average='macro')

print('---------- Report on validation set ----------')
print(f'accurcy : {accuracy}')
print(f'f1 macro score: {f1}')
print(f'recall: {recall}')
print(f'precision: {precision}')
```

مقادیر بدست آمده به شکل زیر می‌باشند.

```
---------- Report on validation set ----------
accurcy : 0.8171112939240804
f1 macro score: 0.7310837816017143
recall: 0.7781261284731812
precision: 0.7050044231722697
```

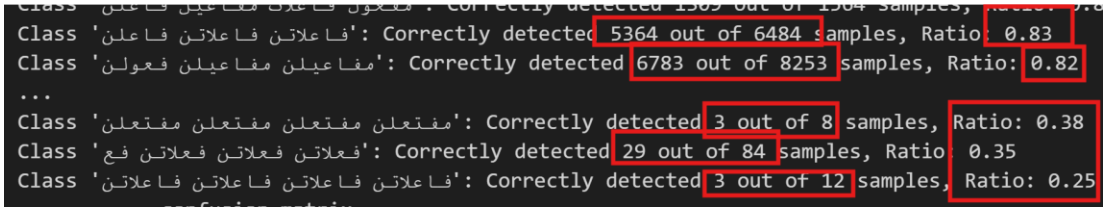
همان‌طور که قابل مشاهده است، مقدار f1score در حدود ۷۳ می‌باشد. این مقادیر به صورت macro حساب شده‌اند. حالت macro شیوه مناسب‌تری برای مجموعه داده‌های imbalanced می‌باشد چرا که معیارها را در سطح هر کلاس (شامل کلاس با تعداد نمونه‌های کم یا زیاد) به طور جداگانه محاسبه می‌کند و سپس میانگین می‌گیرد.

برای بررسی دقیق‌تر و آنالیز نتایج کلاس‌ها، سعی شد که confusion matrix برای این کلاس‌ها کشیده شود. همچنین تلاش شد تا با حلقه روی کلاس‌ها، تعداد نمونه‌هایی که به درستی از آن کلاس پیش‌بینی شده‌اند به کل نمونه‌های آن کلاس تقسیم گردد. در نهایت این لیست بدست آمده از نتایج هر کلاس مرتب‌سازی شده است و confusion matrix به صورت تصویری رسم شده است.

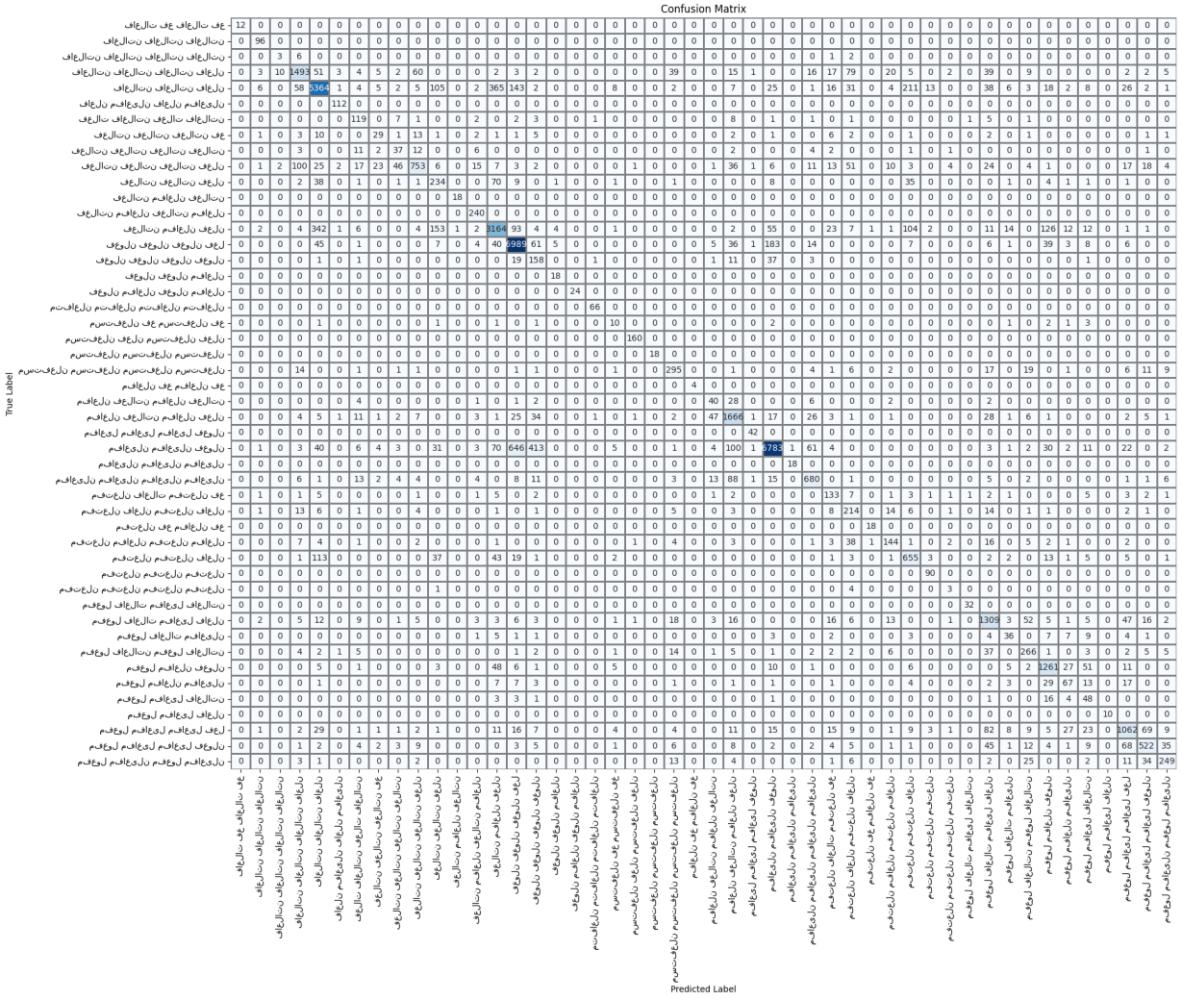
```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

print(f'----------sorted classes (correctly classified / total number of each class) -------')
conf_matrix = confusion_matrix(all_labels, all_predictions)
correct_per_class = conf_matrix.diagonal()
```

```
class_ratios = []
for idx, count in enumerate(correct_per_class):
    class_label = metre_categories[idx]
    total_samples = conf_matrix[idx].sum()
    ratio = count / total_samples if total_samples > 0 else 0
    class_ratios.append((class_label, count, total_samples, ratio))

class_ratios = sorted(class_ratios, key=lambda x: x[3], reverse=True)

print("Class-wise Correct Classification Ratios:")
for class_label, count, total_samples, ratio in class_ratios:
    print(f"Class '{class_label}': Correctly detected {count} out of {total_samples} samples,
Ratio: {ratio:.2f}")

print('------------confusion matrix--------------')
plt.figure(figsize=(20, 16))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=metre_categories,
yticklabels=metre_categories, linewidths=1, linecolor='gray', cbar=False, annot_kws={'size':
10})
plt.xticks(rotation=90, fontsize=10)
plt.yticks(fontsize=10)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

در ابتدای لیست مرتب‌سازی شده، کلاس‌هایی که پیش‌بینی آنان بهتر صورت گرفته آورده شده است و هر چه به انتهای لیست نزدیک‌تر می‌شویم، نتایج بدتر می‌شوند.



همان‌طور که در شکل دیده می‌شود، کلاس‌های انتهای لیست تعداد نمونه‌های کمتری نیز دارند. در ادامه می‌توان شکل confusion matrix بدست آمده را مشاهده نمود.



همان‌طور که از ماتریس درهمریختگی مشخص است، قطر اصلی این ماتریس پررنگ‌تر بوده و نمونه‌های بیشتری دارد که این نشانه خوبی است. یعنی مدل ما تطابق خوبی با برچسب‌های درست داشته.

پیش‌بینی داده‌های آزمون

در نهایت برای حالت دسته‌بندی، مجموعه داده‌های ارزیابی لود شده، به شکل مشابه پردازش شده و آماده ورودی داده شدن به مدل گشته است.

```
test_df = pd.read_csv("Poem Meter Dataset/test_samples.csv")
test_texts = test_df["poem_text"].tolist()

# Adding [CLS] token at the beginning of the inputs
test_texts = ["[CLS] " + text for text in test_texts]

test_inputs = tokenizer(test_texts, return_tensors="pt", padding=True,
truncation=True, max_length=32)

test_dataset = TensorDataset(test_inputs['input_ids'], test_inputs['attention_mask'])
test_dataloader = DataLoader(test_dataset, batch_size=512)
```

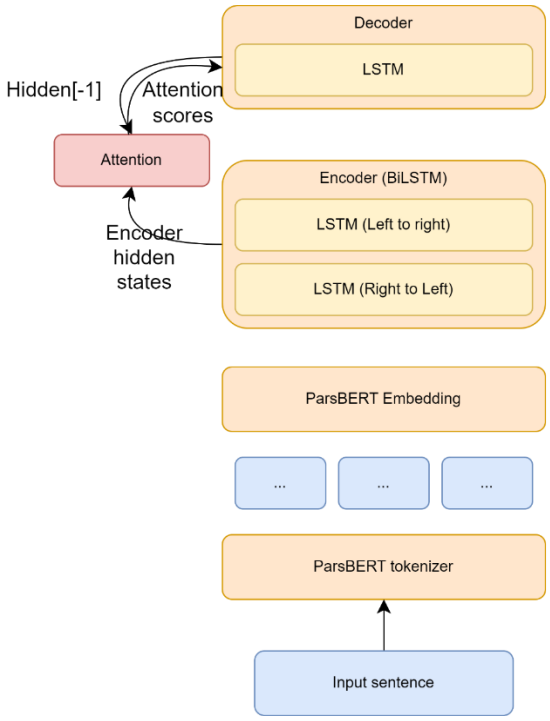
در انتها برای مجموعه داده آزمون، نتایج مدل را با قرار دادن model , classifier در حالت eval بدست می‌آوریم. برای این کار باز با گرفتن بازنمایی cls token روی آن classifier را اعمال می‌کنیم و کلاس با بیشترین احتمال را خروجی می‌گیریم. در نهایت این نتایج در فایل test_samples_with_predictions.csv و در ستون predicted_metre ذخیره‌سازی می‌شوند.

```
model.eval()
classifier.eval()

all_test_predictions = []
with torch.no_grad():
    for batch in test_dataloader:
        input_ids, attention_mask = [x.to(device) for x in batch]

        test_outputs = model(input_ids=input_ids, attention_mask=attention_mask)
        test_cls_token_output = test_outputs.last_hidden_state[:, 0, :]
        test_logits = classifier(test_cls_token_output)

        test_predictions = torch.argmax(test_logits, dim=1)
        all_test_predictions.extend(test_predictions.cpu().numpy())

test_text_labels = [metre_categories[code] for code in all_test_predictions]
test_df["predicted_metre"] = test_text_labels
test_df.to_csv("Poem Meter Dataset/test_samples_with_predictions.csv", index=False)
```

نتایج حاصل از حالت classification برای این مسئله، در فایل test_samples_with_predictions.csv موجود است.

بخش ب) رویکرد دنباله به دنباله RNN

در این بخش قصد داریم تسک تشخیص وزن مصرع را با دید یک مسئله دنباله به دنباله حل کنیم. معماری مدنظر برای حل این سوال به شکل زیر است.



پیش‌پردازش و آماده‌سازی

ابتدا با استفاده از کد زیر مدل ParsBERT را لود می‌کنیم. از این مدل در بخش الف نیز استفاده شد. این مدل به دلیل اینکه روی منابع فارسی آموزش دیده است، برای تسک ما که مختص زبان فارسی است مناسب است و بازنمایی‌های بهتری را در اختیارمان قرار می‌دهد.

```
tokenizer = AutoTokenizer.from_pretrained("HooshvareLab/bert-base-parsbert-uncased")
bert_model = AutoModel.from_pretrained("HooshvareLab/bert-base-parsbert-uncased").to(device)
```

از مدل ParsBERT برای توکنایز کردن ورودی استفاده می‌کنیم. اما برای خروجی، که تعداد توکن‌های آن بسیار محدودتر می‌باشد استفاده از این Embedding مناسب نیست. چرا که بسیار وسیع‌تر از تسک مدنظر ماست و تعداد توکن‌های خروجی ما بسیار محدودتر از آن‌چه که این مدل روی آن آموزش دیده شده است می‌باشد.

از این روی برای این تسک یک Simple Tokenizer طراحی کرده‌ام که مختص خروجی است.

برای اینکه در نهایت بتوان خروجی عددی تولید شده توسط مدل را به حروف تبدیل کرد، در کلاس این توکنایزر سه موجودیت تعریف شده است.

```
class SimpleSpaceTokenizer:
    def __init__(self):
        self.token2id = {}
        self.id2token = {}
        self.vocab_size = 0
```

در متغیر اول، token2id، دیکشنری از token های textual به id های هر یک ذخیره می‌کنیم. در id2token به صورت برعکس یعنی از id به token را داریم. در vocab_size نیز سایز vocabulary را نگه می‌داریم.

حال تابعی را می‌نویسیم که به کمک آن قرار است این متغیرها مقدار دهی شوند.

```
    def fit_on_texts(self, texts):
        unique_tokens = set()
        for text in texts:
            tokens = text.split(" ")
            unique_tokens.update(tokens)

        self.token2id = {token: idx for idx, token in
enumerate(unique_tokens, start=1)}
        self.id2token = {idx: token for token, idx in self.token2id.items()}
        self.vocab_size = len(self.token2id) + 1  # Adding 1 for padding
token
```

برای اینکار ابتدا با استفاده از Space کلمات متن خروجی را از هم جدا می‌کنیم، سپس Uniqueشان را نگه می‌داریم. در نهایت دو دیکشنری id2token و token2id را مقدار دهی می‌کنیم.

در تابع بعدی عمل tokenize انجام می‌شود. برای این کار متن ورودی و max_length داده می‌شوند. در متون خروجی label های ما، space جدا کننده خوبی است. پس با همین جدا کننده توکن ها را در آورده و سپس با دیکشنری‌هایی که در تابع قبلی مقداردهی کردیم توکن ها را به یک عدد id تصویر می‌کنیم.

حال هر چه قدر که خروجی تولید شده تا max_length فاصله داشت، ۰ اضافه می‌کنیم.

```
    def tokenize(self, texts, max_length=48):
        tokenized_texts = []
```

```
        for text in texts:
            tokens = text.split(" ")
            token_ids = [self.token2id.get(token, 0) for token in
tokens][:max_length]
            padding_length = max_length - len(token_ids)
            token_ids += [0] * padding_length
            tokenized_texts.append(token_ids)
        return torch.tensor(tokenized_texts)
```

در نهایت تابع Decode نیز برای تبدیل اعداد به متن نوشته شده است.

```
    def decode(self, token_ids):
        return " ".join([self.id2token.get(token_id, "") for token_id in
token_ids if token_id != 0])
```

حال با استفاده از مواردی که پیش‌تر برای tokenize بیان شد، دیتاست training را لود کرده و پردازش لازم را روی آن انجام می‌دهیم.

```
train_data = pd.read_csv(f'Poem Meter Dataset/train_samples.csv')
```

ستون ورودی و خروجی را جدا می‌کنیم.

```
poem_text = train_data['poem_text']
metre = train_data['metre'].astype(str)
```

ورودی (مصرع شعر) را به pars bert tokenizer می‌دهیم. این امکان را به توکنایزر می‌دهیم تا همه را به طول مساوی ۱۴ تبدیل کند. در این راستا می‌تواند truncate یا padding انجام دهد. خروجی id های معادل هستند. از طرفی Attention_mask نشان می‌دهد که کدام خروجی‌ها مرتبط با متن و کدام برای padding و .. اضافه شده‌اند و معنای خاصی ندارند.

```
inputs = tokenizer(poem_text.tolist(), padding=True, truncation=True,
return_tensors="pt", max_length=14)
input_ids = inputs['input_ids'].squeeze().to(device)
attention_mask = inputs['attention_mask'].squeeze().to(device)
```

برای دنباله خروجی از توکنایزری که پیش‌تر کد آن را نوشتیم استفاده می‌کنیم.

```
label_tokenizer = SimpleSpaceTokenizer()
label_tokenizer.fit_on_texts(metre.tolist())
labels = label_tokenizer.tokenize(metre.tolist(), max_length=6).to(device)
```

برای آماده‌سازی مجموعه داده برای ورودی داده شدن به مدل، آن را با استفاده از DataLoader لود می‌کنیم.

```
train_loader = DataLoader(torch.utils.data.TensorDataset(input_ids, attention_mask,
labels), batch_size=512, shuffle=True)
```

همین کارها را به شکل مشابه برای مجموعه داده‌های validation , test انجام می‌دهیم.

```
val_data = pd.read_csv(f'Poem Meter Dataset/validation_samples.csv')

val_poem_text = val_data['poem_text']
val_metre = val_data['metre'].astype(str)

val_inputs = tokenizer(val_poem_text.tolist(), padding=True, truncation=True,
return_tensors="pt", max_length=14)
val_input_ids = val_inputs['input_ids'].squeeze().to(device)
val_attention_mask = val_inputs['attention_mask'].squeeze().to(device)
val_labels = label_tokenizer.tokenize(val_metre.tolist(), max_length=6).to(device)

val_loader = DataLoader(torch.utils.data.TensorDataset(val_input_ids,
val_attention_mask, val_labels), batch_size=512, shuffle=True)
```

```
test_data = pd.read_csv(f'Poem Meter Dataset/test_samples.csv')

test_poem_text = test_data['poem_text']

test_inputs = tokenizer(test_poem_text.tolist(), padding=True, truncation=True,
return_tensors="pt", max_length=14)

test_input_ids = test_inputs['input_ids'].squeeze().to(device)
test_attention_mask = test_inputs['attention_mask'].squeeze().to(device)

test_loader = DataLoader(torch.utils.data.TensorDataset(test_input_ids,
test_attention_mask), batch_size=512, shuffle=True)
```

معماری شبکه

حال به بخش پیاده‌سازی و کد اصلی مدل می‌رسیم.

برای توضیح معماری و پیاده‌سازی انجام شده از بخش Encoder شروع می‌کنیم که برای آن از ساختار Bi LSTM استفاده شده است.

همچنین از مدل پارس برت برای Embedding ورودی در این ساختار استفاده شده.

```
class Encoder(nn.Module):  # Bi-LSTM
    def __init__(self, bert_model, hidden_size, num_layers):
        super(Encoder, self).__init__()
        self.bert = bert_model
        self.bi_lstm = nn.LSTM(bert_model.config.hidden_size, hidden_size,
num_layers, batch_first=True, bidirectional=True)
```

مراحل انجام کار به این صورت است که ابتدا ورودی به مدل ParsBERT داده می‌شود تا Embedding ها دریافت شوند. بعد از دریافت Embedding ها، این تعبیه‌ها به عنوان ورودی به Bi-LSTM داده می‌شوند.

```
    def forward(self, input_ids, attention_mask):
        with torch.no_grad():
            bert_embedding_outputs = self.bert(input_ids=input_ids,
attention_mask=attention_mask)[0]
        outputs, (hidden, cell) = self.bi_lstm(bert_embedding_outputs)
        return outputs, hidden, cell
```

در گام دوم، به ساختار Attention می‌پردازیم. از این ساختار در Decoder برای پیدا کردن و تاثیر دادن Hidden state های موجود در Encoder استفاده شده است.

مکانیزیم توجه، به دو ورودی احتیاج دارد. یکی Hidden state قبلی Decoder، و دیگری Hidden state های بدست آمده از Encoder. با در اختیار داشتن این دو و براساس ارتباط پیدا شده بین آنان، وزن مناسب برای تاثیر دادن بخش‌های مختلف ورودی در تولید خروجی استفاده شود. از آن‌جا که در ساختار Encoder از BiLSTM استفاده شده و Hidden state های بدست آمده از Left-to-right LSTM و Right-to-Left LSTM در انکدر با هم concate شدند، سایز Hidden state انکدر دو برابر دیکدر است.

```
class Attention(nn.Module):
    def __init__(self, encoder_hidden_size, decoder_hidden_size, method='general'):
        super(Attention, self).__init__()
        self.method = method
        self.encoder_hidden_size = encoder_hidden_size * 2  # Bi-directional
        self.decoder_hidden_size = decoder_hidden_size
```

حال دو روش برای پیاده‌سازی مکانیزیم Attention در نظر گرفته شده است. روش رایج‌تر که ابتدا با استفاده از یک لایه fully connected قابل یادگیری بازنمایی‌های Hidden state از Encoder را به سایز بازنمایی‌های Decoder hidden state انتقال می‌دهد. سپس مقدار Hidden state های Encoder را در Decoder ضرب می‌کند. با اینکار لایه fully connected یاد می‌گیرد که به هر بخش چه وزنی را اختصاص دهد.

روش دوم که در برخی پیاده‌سازی های موجود مشاهده کردم، بازنمایی‌های Encoder hidden state را با بازنمایی Decoder hidden state در کنار هم قرار می‌دهد و یک بردار بزرگتر می‌سازد. سپس سعی می‌کند با یک لایه fully connectedقابل یادگیری، این بردار را به برداری با سایز Decoder hidden state تبدیل کند. به عبارتی در این لایه که مانند یک ماتریس ضرایب عمل می‌کند، وزن تاثیر هر عنصر در ساخت بردار با سایز مناسب دیکدر پیدا می‌شود.

```
        if self.method == 'general':
            self.attn = nn.Linear(self.encoder_hidden_size, decoder_hidden_size)
        elif self.method == 'concat':
            self.attn = nn.Linear(self.encoder_hidden_size + decoder_hidden_size,
decoder_hidden_size)
            self.v = nn.Parameter(torch.rand(decoder_hidden_size))
```

در هنگام forward کردن با استفاده از این ساختارها، ابتدا آخرین Decoder hidden state در نظر گرفته می‌شود و سپس از attention تعریف شده در قسمت بالا استفاده می‌شود.

```
    def forward(self, hidden, encoder_outputs):
        if self.method == 'general':
            hidden = hidden[-1].unsqueeze(1)
            logits = torch.bmm(self.attn(encoder_outputs), hidden.transpose(1, 2)).squeeze(2)
        elif self.method == 'concat':
            hidden = hidden[-1].expand(encoder_outputs.shape[0], -1, -1)
            logits = torch.sum(self.v * torch.tanh(self.attn(torch.cat((hidden,
encoder_outputs), 2))), dim=2)
        return F.softmax(logits, dim=1)
```

حال به بخش Decoder می‌رسیم. در این بخش، نیاز داریم تا در ابتدا vocab_size را بدانیم. دانستن vocab size خروجی (نه ورودی)، برای تصویر کردن ابعاد embed_size استفاده شده در Decoder به سایز مجموعه لغات خروجی مورد نیاز است. با انجام این کار، می‌توان از Softmax برای بدست آوردن پیش‌بینی شبکه استفاده نمود.

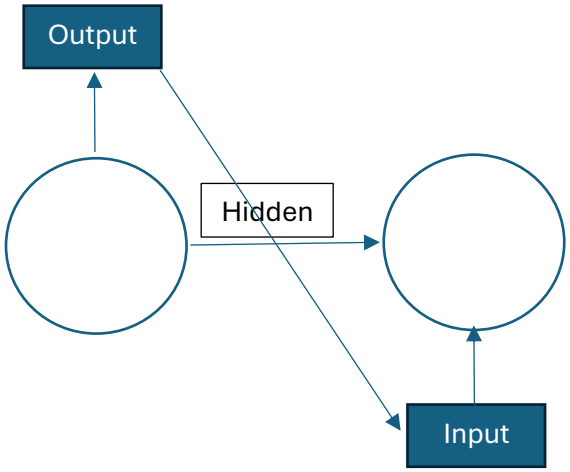
از سمت دیگر، به ابعادی که Decoder hidden state, Encoder hidden state, Embed_dim دارند نیز نیاز داریم. برای توضیح دقیق‌تر پیاده‌سازی انجام شده برای Deocder بهتر است آن را با شیوه استفاده شده در هنگام train و Eval توضیح دهم.

در ابتدا اشاره می‌کنم که در Decoder به یک لایه Embedding برای تعبیه توکن‌های خروجی، یک لایه LSTM، عملیات Attention و در نهایت یک fully connected برای تبدیل بازنمایی شبکه به تعداد vocab size نیاز داریم.

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, encoder_hidden_size, decoder_hidden_size,
num_layers, attention_method='general'):
        super(Decoder, self).__init__()
```

```
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, decoder_hidden_size, num_layers, batch_first=True)
        self.w = nn.Linear(decoder_hidden_size + encoder_hidden_size * 2, decoder_hidden_size)
        self.attention = Attention(encoder_hidden_size, decoder_hidden_size, attention_method)
        self.fc = nn.Linear(decoder_hidden_size, vocab_size)
```

در هنگام آموزش و استفاده از این ساختار برای هر batch، هر بار به آن توکن قبلی دنباله خروجی (که قبلا شبکه آن را تولید کرده یا واقعا برچسب درست است) داده می‌شود. این ورودی به لایه Embedding داخل Decoder داده می‌شود تا بازنمایی مناسب آن را یادگرفته و پیدا کند. از سمت دیگر بازنمایی hidden پیدا شده از مرحله قبلی استفاده می‌شود تا بازنمایی پنهان جدید بدست آید.



با انجام این کار، بازنمایی بدست آمده در دیکدر را به همراه خروجی موجود در انکدر به مکانیزیم توجه که پیش‌تر آن را توضیح دادیم ورودی می‌دهیم تا وزن‌های تاثیر را دریافت کنیم.

```
    def forward(self, inputs, hidden, cell, encoder_outputs):
        inputs = inputs.unsqueeze(1)
        embedding = self.embedding(inputs)
        outputs, (hidden, cell) = self.lstm(embedding, (hidden, cell))
        attention_weights = self.attention(hidden, encoder_outputs)
```

بعد از دریافت وزن‌های توجه، نیاز است تا با استفاده از آنان و بازنمایی‌های بدست آمده از انکدر، context vector را بدست آوریم. برای این کار encoder output را در وزن‌های توجه ضرب می‌کنیم و مقادیر context vector را بدست می‌آوریم. در نهای از خروجی بدست آمده از LSTM و این context بدست آمده استفاده می‌کنیم و آنان را با هم ترکیب می‌کنیم.

```
        context_vec = torch.bmm(attention_weights.unsqueeze(1), encoder_outputs)
        concat_input = torch.cat((outputs, context_vec), dim=2)
        cats = torch.tanh(self.w(concat_input))
```

سپس مقدار بدست آمده را بعد از عبور از یک تابع فعالسازی غیر خطی به یک لایه fully connected می‌دهیم تا آن را به سایز vocab تبدیل کند.

```
        pred = self.fc(cats.squeeze(1))
        return pred, hidden, cell
```

با جست و جویی که داشتم، رویکرد بالا یکی از رویکردهای استفاده از مکانیزیم attention است که به طور خاص به آن Luong گفته می‌شود که از hidden بدست آمده در همین cell برای محاسبه مقادیر وزن های توجه استفاده می‌کند. رویکرد دیگر Bahdanaou است که از از hidden state سلول پیشین برای بدست آوردن مقادیر وزن‌های توجه استفاده می‌کند.

Intuitive Introduction to Neural Machine Translation with Bahdanau and Luong Attention

فرآیند آموزش

حال با استفاده از معماری پیاده‌سازی شده برای شبکه، به کد بخش آموزش و ارزیابی می‌پردازیم. در ابتدا encoder , decoder و همین طور شیوه optimize شدن و بروزرسانی شدن وزن‌هایشان را تنظیم می‌کنیم. تابع زیان نیز برابر CrossEntropyLoss است.

```
encoder = Encoder(bert_model, hidden_size, num_layers_encoder).to(device)
decoder = Decoder(output_dim, embed_dim, hidden_size, hidden_size,
num_layers_decoder).to(device)

encoder_optimizer = optim.Adam(encoder.parameters(), lr=0.001, weight_decay=0.01)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=0.001, weight_decay=0.01)

criterion = nn.CrossEntropyLoss()
```

در تابع آموزش مدل، ابتدا مدل‌ها را در حالت آموزش قرار می‌دهیم. برای هر batchدر training data ابتدا ورودی، توکن‌های ماسک شده و خروجی درست را دریافت می‌کنیم. گردایان را برای انکدر و دیکدر در ابتدا روی صفر قرار می‌دهیم. سپس از انکدر برای بدست آوردن hidden state ها و خروجی انکدر استفاده می‌کنیم.

```
def train_epoch(encoder, decoder, dataloader, val_loader, encoder_optimizer,
decoder_optimizer, criterion, teacher_forcing_ratio = 0.95):
    encoder.train()
    decoder.train()
    epoch_loss = 0
    best_val_loss = float('inf')
    patience_counter = 0
    patience = 3

    for i, batch in enumerate(dataloader):
        input_ids, attention_mask, labels = [x.to(device) for x in batch]

        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()
```

```
        encoder_outputs, hidden, cell = encoder(input_ids, attention_mask)
```

در ادامه هر بار با در اختیار قرار دادن توکن t ام از خروجی مدنظر به مدل یا استفاده از خروجی قبلی مدل، از دیکدر می‌خواهیم خروجی بعدی را پیش‌بینی کند.

```
        decoder_input = torch.zeros(labels.size(0), dtype=torch.long, device=device)  # Start
token
        loss = 0

        for t in range(0, labels.size(1)):
            output, hidden, cell = decoder(decoder_input, hidden, cell, encoder_outputs)
            loss += criterion(output, labels[:, t])
            # teacher forcing ....
            if random.random() < teacher_forcing_ratio:
                decoder_input = labels[:, t]
            else:
                decoder_input = output.argmax(1)
```

برای این کار از teacher forcing استفاده می‌کنیم. رویکرد teacher forcing با scheduler در این پیاده‌سازی استفاده شده که به شکل زیر عمل می‌کند:

در ورودی به تابع یک مقدار اولیه برای teacher forcing می‌دهیم که بین ۰ تا ۱ است. در حین آموزش به صورت رندوم یک عدد از بین ۰ تا ۱ تولید می‌کنیم. در صورتی که این عدد کمتر از teacher forcing ratio بود، از برچسب درست برای ورودی دادن به decoder برای پیش‌بینی توکن بعدی استفاده می‌کنیم. در صورتی که بیشتر بود، از خروجی قبلی مدل به عنوان ورودی جدید استفاده می‌کنیم.

مقدار teacher forcing ratio نیز به تدریج کاهش می‌یابد. با این کاهش، در iteration های آخر مدل کاملا بر اساس پیش‌بینی‌های خودش پیش می‌رود.

```
teacher_forcing_ratio = max(0.03, teacher_forcing_ratio - 0.001)
```

حال loss بدست آمده را حساب کرده و backward می‌کنیم و وزن‌ها را بروزرسانی می‌کنیم.

```
        loss.backward()
        encoder_optimizer.step()
        decoder_optimizer.step()

        epoch_loss += loss.item() / labels.size(1)
        print(f'batch {i}/{len(dataloader)} ,loss: {loss:.4f}')
```

بعد از تعداد مشخصی iteration و در زمانی که به آخرین batch از training رسیدیم، یکبار loss را روی داده‌های ارزیابی validation حساب می‌کنیم. این کار دقیقا مطابق همان مراحل برای داده‌های آموزشی است.

```
        if i == (len(dataloader) - 1):
            total_val_loss = 0
```

```
            num_val_batches = 0

            encoder.eval()
            decoder.eval()
            with torch.no_grad():
                for val_batch in val_loader:
                    input_ids, attention_mask, labels = [x.to(device) for x in val_batch]
                    encoder_outputs, hidden, cell = encoder(input_ids, attention_mask)
                    decoder_input = torch.zeros(labels.size(0), dtype=torch.long,
device=device)
                    val_loss = 0

                    for t in range(0, labels.size(1)):
                        output, hidden, cell = decoder(decoder_input, hidden, cell,
encoder_outputs)
                        val_loss += criterion(output, labels[:, t])
                        # decoder_input = labels[:, t]
                        decoder_input = output.argmax(1)

                    total_val_loss += val_loss.item() / labels.size(1)
                    num_val_batches += 1

            avg_val_loss = total_val_loss / num_val_batches if num_val_batches > 0 else
float('inf')
            print(f'Batch {i + 1}, Validation Loss: {avg_val_loss:.4f}')
```

برای پیش‌گیری از طولانی شدن آموزش و overfitting نیز از مکانیزیم early stopping استفاده می‌کنیم. در این رویکرد بعد از هر بار محاسبه شدن خطا برای داده‌های ارزیابی اگر خطا از بهترین خطای دفعات پیش نبود، یک عدد به شمارنده patience_counter اضافه می‌کنیم. اگر بیشتر از عدد مشخصی شد، یعنی به تعداد دفعات مشخصی بهبودی در نتایج مشاهده نکردیم، آموزش را متوقف می‌کنیم.

```
            if avg_val_loss < best_val_loss:
                best_val_loss = avg_val_loss
                patience_counter = 0
                torch.save({'encoder': encoder.state_dict(), 'decoder': decoder.state_dict()},
'best_model.pth')
            else:
                patience_counter += 1

            if patience_counter >= patience:
                print(f'Early stopping triggered during training at batch {i + 1} in epoch.')
                return epoch_loss / len(dataloader)

            encoder.train()
            decoder.train()
```

حال با استفاده از این تابع، مدل را آموزش می‌دهیم.

```
n_epochs = 20
for epoch in range(n_epochs):
    train_loss = train_epoch(encoder, decoder, train_loader, val_loader, encoder_optimizer,
decoder_optimizer, criterion)
```

```
    print(f'Epoch {epoch+1}, Training Loss: {train_loss:.4f}')
```

ارزیابی

برای ارزیابی مدل و بدست آوردن `metric` های مختلف، تابعی نوشته‌ام که دقیقا مشابه با حلقه `training` یا `evaluation` را اجرا می‌کند. با این تفاوت که خروجی آن `loss` نیست بلکه `preds, true_labels` است.

بدین شکل که، برای هر `batch` از `validation data` ابتدا ورودی، خروجی درست و ماسک‌ها را دریافت کرده. ورودی را به `Encoder` می‌دهد تا بازنمایی آن را بگیرد. سپس این بازنمایی را به `Decoder` می‌دهد. هر بار از خروجی دفعه قبلی `decoder` به عنوان ورودی بعدی استفاده می‌کند و پیش‌بینی‌ها را ذخیره می‌کند.

```
def evaluation(encoder, decoder, dataloader):
    preds = []
    true_labels = []

    encoder.eval()
    decoder.eval()
    with torch.no_grad():
        for j, val_batch in enumerate(dataloader):
            print(f'batch: {j} / {len(dataloader)}')
            input_ids, attention_mask, labels = [x.to(device) for x in val_batch]
            encoder_outputs, hidden, cell = encoder(input_ids, attention_mask)

            batch_size = input_ids.size(0)
            seq_length = labels.size(1)

            # Initialize decoder inputs for the entire batch
            decoder_input = torch.zeros(batch_size, dtype=torch.long, device=device)
            batch_preds = [[] for _ in range(batch_size)]

            hidden = hidden.contiguous()
            cell = cell.contiguous()

            # Iterate through the sequence length
            for t in range(0, seq_length):
                output, hidden, cell = decoder(decoder_input, hidden, cell, encoder_outputs)
                decoder_input = output.argmax(1)

                # Collect predictions for each sequence in the batch
                for i in range(batch_size):
                    batch_preds[i].append(decoder_input[i].item())

            preds.extend(batch_preds)
            true_labels.extend(labels[:, 0:].tolist())
    return preds, true_labels
```

با صدا زدن این تابع برچسب‌های درست و پیش‌بینی مدل را برای داده‌های اعتبارسنجی بدست می‌آوریم.

```
val_preds, val_true_labels = evaluation(encoder, decoder, val_loader)
```

آنان را به بردار np تبدیل کرده و Decode می‌کنیم تا از فرمت عددی id به دنباله کلمات تبدیل شوند و قابل خواندن و بررسی باشند.

```
        val_preds = np.array(val_preds)

val_true_labels = np.array(val_true_labels)
```

```
val_pred_decoded = [label_tokenizer.decode(pred) for pred in val_preds]
val_true_labels_decoded = [label_tokenizer.decode(label) for label in val_true_labels]
```

با استفاده از حلقه زیر برای تعدادی از نمونه‌ها، خروجی درست و خروجی مدل را بدست می‌آوریم.

```
for i in range(0,20):
    print(f'val_pred: {val_pred_decoded[i]}')
    print(f'val_true_label: {val_true_labels_decoded[i]}')
    print('-------------------------------------------')
```

خروجی‌ها به شکل زیر هستند.

```
val_pred: فاعلاتن فاعلاتن فاعلن
val_true_label: فعلاتن مفاعلن فعلن
-------------------------------------------
val_pred: فعولن فعولن فعولن فعل
val_true_label: فعولن فعولن فعولن فعل
-------------------------------------------
val_pred: مفاعیلن مفاعیلن فعولن
val_true_label: مفاعیلن مفاعیلن فعولن
-------------------------------------------
val_pred: فعلاتن فعلاتن فعلاتن فعلن
val_true_label: مفاعلن فعلاتن مفاعلن فعلن
```

حال برای محاسبه معیارها از کتابخانه torchmetrics استفاده می‌کنیم.

در این کتابخانه، دو معیار مطرح برای ارزیابی تبدیل دنباله به دنباله وجود دارد: Rouge, Bleu

در معیار bleu تمرکز بر precision است. این معیار بیان می‌کند که چه تعداد از n-gram های موجود در متن پیش‌بینی شده توسط مدل، در متن برچسب درست نیز وجود دارد.

در معیار rouge تمرکز بر recall , f1 است. این معیار بیان می‌کند که چه تعداد از n-gram های موجود در متن درست، در بین پیش‌بینی‌های مدل نیز وجود دارد. در این معیار البته precision نیز با منطق مشابه (یعنی در نظر گرفتن تعداد n-gram های موجود از متن پیش‌بینی شده در متن اصلی) محاسبه می‌شود. سپس معیار f1 مطابق زیر با در نظر گرفتن بتا برابر ۱ محاسبه می‌شود.

$$F_\beta = \frac{(1+\beta^2) \times P \times R}{\beta^2 \times P + R}$$

در کد زیر این معیارها محاسبه شده‌اند.

```
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

nltk.download('punkt_tab')

from torchmetrics.text import BLEUScore, ROUGEScore

bleu = BLEUScore()
rouge = ROUGEScore()

# NLTK not good for persian tokenization ...
val_pred_str = [' '.join(map(str, pred)) for pred in val_preds]
val_true_str = [' '.join(map(str, true)) for true in val_true_labels]

print(f'BLEU Score: {bleu(val_pred_str, [[true] for true in val_true_str])}')
print(f'ROUGE Score: {rouge(val_pred_str, val_true_str)}')
```

معیار Rouge1 در شمارش خود، بر اساس uni-gram ها کار می‌کند. یعنی توکن به توکن بررسی می‌کند. اما معیار Rouge2 بر اساس bigram کار می‌کند. یعنی جفت توکن‌های داخل دنباله‌ها را شمارش می‌کند. در نهایت RougeL بلندترین دنباله مشترک بین دو دنباله را مدنظر قرار می‌دهد.

```
BLEU Score: 0.6196612119674683
```

```
ROUGE Score: {'rouge1_fmeasure': tensor(0.7640), 'rouge1_precision':
tensor(0.7640), 'rouge1_recall': tensor(0.7640), 'rouge2_fmeasure':
tensor(0.6680), 'rouge2_precision': tensor(0.6680), 'rouge2_recall':
tensor(0.6680), 'rougeL_fmeasure': tensor(0.7639), 'rougeL_precision':
tensor(0.7639), 'rougeL_recall': tensor(0.7639), 'rougeLsum_fmeasure':
tensor(0.7639), 'rougeLsum_precision': tensor(0.7639), 'rougeLsum_recall':
tensor(0.7639)}
```

همان‌طور که قابل مشاهده می‌باشد، مقادیر در حدود ۷۶ برای f1 در Unigram و حدود ۶۶ برای bigram است.

البته می‌توان f1 , precision, recall را با در نظر گرفتن توکن‌های خروجی به صورت تکی نیز به دست آورد. یعنی به ازای هر توکن بررسی کنیم که این توکن پیش‌بینی شده با توکن موجود در برچسب منطبق هست یا خیر.

در این حالت می‌توان از کد زیر استفاده نمود که البته رویکرد قبلی و معیارهای قبلی برای بررسی نتایج seq2seq استفاده می‌شوند.

```
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score
import numpy as np

val_preds = np.array(val_preds)
val_true_labels = np.array(val_true_labels)

val_preds_flat = val_preds.ravel()
val_true_labels_flat = val_true_labels.ravel()

accuracy = accuracy_score(val_true_labels_flat, val_preds_flat)
```

```
f1 = f1_score(val_true_labels_flat, val_preds_flat, average='macro', zero_division=1)
recall = recall_score(val_true_labels_flat, val_preds_flat, average='macro', zero_division=1)
precision = precision_score(val_true_labels_flat, val_preds_flat, average='macro',
zero_division=1)

print(f"Accuracy: {accuracy:.4f}")
print(f"F1 Score: {f1:.4f}")
print(f"Recall: {recall:.4f}")
print(f"Precision: {precision:.4f}")
```

با این شیوه داریم:

```
Accuracy: 0.7590

F1 Score: 0.4108

Recall: 0.4101

Precision: 0.6879
```

ولی همان‌طور که بیان شد در ارزیابی Seq2seq از همان معیارهای `f1, recall, precision` بدست آمده از طریق `ROUGE, BLEU` استفاده می‌شود.

پیش‌بینی داده‌های آزمون

در نهایت نتایج روی داده‌های تست با استفاده از کد مشابه پیدا و در فایل `test_samples_seq_to_seq_results.csv` ذخیره شده‌اند.

```
def predict(encoder, decoder, dataloader):
    encoder.eval()
    decoder.eval()

    predicted_metres = []

    with torch.no_grad():
        for batch in dataloader:
            input_ids, attention_mask = batch
            encoder_outputs, hidden, cell = encoder(input_ids, attention_mask)
            decoder_input = torch.zeros(input_ids.size(0), dtype=torch.long).to(device)

            batch_predictions = []
            for t in range(14):
                output, hidden, cell = decoder(decoder_input, hidden, cell, encoder_outputs)
                decoder_input = output.argmax(1)
                batch_predictions.append(decoder_input)

            batch_predictions = torch.stack(batch_predictions, dim=1).cpu().numpy()
            predicted_metres.extend(batch_predictions)

    return predicted_metres
```

```
test_predictions = predict(encoder, decoder, test_loader)
```

```
test_data['predicted_metre'] = test_prediction_decoded
test_data.to_csv('test_samples_seq_to_seq_results.csv', index=False)
```

رویکرد Beam Search

در پیاده‌سازی پیشین برای Evaluation از Greedy استفاده کردیم و هر بار توکنی را که بیشترین احتمال را داشت به عنوان توکن درست بعدی استفاده کردیم. اما رویکرد دیگری به عنوان Beam Search نیز مطرح است. این رویکرد هر بار n تا توکن با بیشترین احتمال را در نظر می‌گیرد و سعی می‌کند برای هر کدام از آن حالات، پیش‌بینی‌های بعدی را نیز انجام دهد و هر بار از بین تمام شاخه‌هایی که ایجاد می‌شود n شاخه‌ای که بیشترین احتمال را دارند در نظر گیرد و ادامه دهد.

در این پیاده‌سازی شیوه آموزش مدل را برای این حالت تغییری نداده‌ام. بلکه فقط در هنگام evaluation تغییراتی را ایجاد کرده‌ام. برای این کار بعد از قرار دادن Encoder, decoder در حالت Eval، برای هر batch در validation، به ازای هر sample یک آرایه beams تعریف کرده‌ام که قرار است حالات مختلف را در خود نگه دارد.

```
    with torch.no_grad():
        for j, val_batch in enumerate(dataloader):
            print(f'batch: {j} / {len(dataloader)}')
            input_ids, attention_mask, labels = [x.to(device) for x in val_batch]
            encoder_outputs, hidden, cell = encoder(input_ids, attention_mask)

            batch_size = input_ids.size(0)
            seq_length = labels.size(1)

            hidden = hidden.contiguous()
            cell = cell.contiguous()

            batch_preds = [[] for _ in range(batch_size)]
            for i in range(batch_size):
                beams = [(torch.zeros(1, dtype=torch.long, device=device), 0.0, hidden[:,
i:i+1, :].contiguous(), cell[:, i:i+1, :].contiguous(), 1)]
                completed_sequences = []
```

برای هر توکن در هر دنباله، دیکدر پیش‌بینی را انجام می‌دهد. سپس softmax اعمال می‌شود و k تا دنباله با بیشترین احتمال انتخاب می‌شوند. این k توسط مقدار beam width تعیین می‌شود. برای اینکه k تای اول softmax را برداریم و سپس احتمال بدست آمده را در احتمال دنباله تا قبل از این توکن ضرب کنیم، بار محاسباتی زیادی صرف می‌شود. از طرفی هدف ما نهایتا برداشتن k تا دنباله با بیشترین احتمال است. از این روی به جای استفاده از خود Softmax از log softmax استفاده شده و با مقادیر قبلی دنباله جمع شده است.

```
                for seq, score, hidden_i, cell_i, length in beams:
                    output, hidden_i, cell_i = decoder(seq[-1:], hidden_i, cell_i,
encoder_outputs[i:i+1])
                    topk_logits, topk_indices = torch.topk(output, beam_width, dim=-1)
                    topk_log_probs = F.log_softmax(topk_logits, dim=-1)
```

```
                        for k in range(beam_width):
                            new_seq = torch.cat([seq, topk_indices[:, k]], dim=-1)
                            new_score = score + topk_log_probs[0, k].item()
                            normalized_score = new_score / ((5 + length + 1) / 6) **
length_penalty_alpha

                            new_beams.append((new_seq, normalized_score, hidden_i, cell_i,
length + 1))
                    beams = sorted(new_beams, key=lambda x: x[1], reverse=True)[:beam_width]
```

سپس دنباله‌های تکمیل شده و به انتها رسیده (که به اندیس ۰ می‌رسند)، را انتخاب کرده و در نهایت آن دنباله کاملی که بیشترین احتمال را دارد به عنوان جواب نهایی انتخاب می‌کنیم.

```
                    completed_sequences.extend([b for b in beams if b[0][-1].item() == 0])
                    beams = [b for b in beams if b[0][-1].item() != 0]

                    if len(beams) == 0:
                        break
```

```
                if len(completed_sequences) == 0:
                    completed_sequences = beams

                best_seq = max(completed_sequences, key=lambda x: x[1])[0]
                batch_preds[i] = best_seq[1:].tolist()
```

حال با استفاده از این تابع، شیوه Beam search را روی مجموعه داده‌های اعتبارسنجی استفاده می‌کنیم.

```
beam_val_preds, beam_val_true_labels = beam_search_eval(encoder, decoder, val_loader)
```

خروجی‌های بدست آمده مطابق زیر است.

```
beam search val_pred: فعلاتن مفاعلن فعولن
beam search val_true_label: فعلاتن مفاعلن فعلن
-----------------------------------------
beam search val_pred: مفاعیلن مفاعیلن فعولن
beam search val_true_label: مفاعیلن مفاعیلن فعولن
-----------------------------------------
beam search val_pred: فعلاتن مفاعلن فعلن
beam search val_true_label: فاعلاتن فاعلاتن فاعلن
-----------------------------------------
beam search val_pred: مفعول مفاعلن فعولن
beam search val_true_label: فعلاتن مفاعلن فعلن
```

اگر f1 score و precision , recall بدست آمده توسط Rouge را بررسی کنیم، معیارهای ارزیابی به شکل زیر می‌شوند.

BLEU Score: 0.6199867129325867

ROUGE Score: {'rouge1_fmeasure': tensor(0.7631), 'rouge1_recall': (tensor(0.7631

نتایج بسیار به حالت Greedy نزدیک است. با این وجود زمان اجرای مورد نیاز برای این حالت حدودا ۳ یا ۴ برابر Greedy است.

اگر معیارها را به ازای تک تک توکن‌ها محاسبه کنیم و نه به شکل دنباله n-gram نتیجه به شکل زیر می‌شود.

Beam search eval:

Accuracy: 0.7582

F1 Score: 0.4108

Recall: 0.4125

Precision: 0.6864

در نهایت تابعی نوشته شده که دقیقا مشابه با beam search eval برای داده‌های test مقادیر را پیش‌بینی می‌کند. در نهایت در مسیر test_samples_seq_to_seq_results_beam_search.csvنتایج مربوط به این حالت ذخیره سازی شده‌اند.

```
beam_test_predictions = beam_prediction(encoder, decoder, test_loader)
beam_test_prediction_decoded = [label_tokenizer.decode(pred) for pred in
beam_test_predictions]


test_data['predicted_metre'] = beam_test_prediction_decoded
test_data.to_csv('test_samples_seq_to_seq_results_beam_search.csv', index=False)
```

ج) رویکرد دنباله به دنباله ترنسفورمری

پیش‌پردازش و آماده‌سازی

در این بخش سعی شده است تا با استفاده از ساختار Transformer based encoder, decoder برای حل این مسئله Seq2Seq استفاده شود.

برای این بخش نیز برای قسمت tokenization مشابه قسمت قبلی از ParsBERT برای توکنایز کردن دنباله ورودی و از SimpleTokenizer پیاده‌سازی شده برای توکنایز کردن دنباله خروجی استفاده شده است. برای جلوگیری از تکرار، توضیحات مرتبط با این دو در این قسمت آورده نشده ولی در قسمت ب به طور مفصل توضیح داده شده است.

همچنین به شکل یکسان و مطابق با شیوه لود و پردازش مطرح شده در قسمت ب، در این قسمت نیز داده‌های آموزش، ارزیابی و تست را لود و آماده می‌کنیم. توضیحات این موارد نیز به طور مفصل در قسمت ب آورده شده است.

معماری شبکه

حال به بخش معماری شبکه می‌رسیم که در واقع تفاوت اصلی این قسمت با قسمت ب می‌باشد.

در ساختار `Transformer` قبل از ورودی دادن دنباله به شبکه نیازمند `Positional encoding` هستیم. وظیفه `Positional encoding` این است که به ترتیب و جایگاه توکن‌ها حساسیت ایجاد کند. به عبارتی، ترتیب توکن‌ها در نظر مدل `Transformer` اهمیت داشته باشد و ورودی مدل به ازای ترتیب‌های مختلف متفاوت شود. از این جهت `Positional encoding` برای هر یک از جایگاه‌های موجود در دنباله یک تعبیه یکتا ایجاد می‌کند.

برای اینکار از فرمول زیر استفاده می‌کند:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

در این رابطه `Pos` نشان‌دهنده موقعیت توکن در دنباله است. مثلا توکن سوم از دنباله. مقدار `i` نشان می‌دهد که اندیس بعد از بازنمایی است. به عنوان مثال اگر بردار تعبیه شبکه ۲۵۶ تایی است. مقدار `i` می‌تواند از ۰ تا ۲۵۵ باشد. همان‌طور که مشخص است، مقدار تولیدی رابطه به ازای `i` زوج و فرد متفاوت است.

به این ترتیب خروجی این بخش، به ازای هر موقعیت، یک بردار به اندازه `embedding size dim` شبکه است.

در رابطه زیر ابتدا مخرج `10000^(2i/d_model)` به شکل زیر محاسبه شده است.

```
import math
class PositionalEncoding(nn.Module):
    def __init__(self, embed_size, max_len=512):
        super(PositionalEncoding, self).__init__()
        pos_encoding = torch.zeros(max_len, embed_size)
        position_list = torch.arange(0, max_len, dtype=torch.float).view(-1,1)
        div_term = torch.exp(torch.arange(0, embed_size, 2).float() * (-math.log(10000.0) /
embed_size))
```

که معادل عبارت زیر است.

$$\text{div\_term} = \exp\left(\frac{-\log(10000)}{d_{\text{model}}} \times i\right)$$

سپس برای جایگاه‌های زوج و فرد به ترتیب از رابطه sin و cos استفاده شده است.

```
        pos_encoding[:, 0::2] = torch.sin(position_list * div_term)
        pos_encoding[:, 1::2] = torch.cos(position_list * div_term)
 pos_encoding = pos_encoding.unsqueeze(0).transpose(0, 1)
```

در تابع forward این قسمت، مقدار embedding یا x ورودی گرفته می‌شود و سپس pos_encoding آن بهش اضافه می‌شود.

```
    def forward(self, x):
        x + self.pos_encoding[:x.size(0), :]
        return x
```

در ادامه به ساختار Transformer می‌رسیم.

برای پیاده‌سازی ساختار Transformer در این مسئله نیاز به تعریف چند متغیر داریم. در گام اول، باز هم از pars bert برای embedding ورودی استفاده می‌کنیم. در ادامه باید خروجی این لایه را به سایز embedding مورد استفاده در شبکه transformer تبدیل کنیم. این کار توسط یک لایه fully connected انجام می‌گیرد. همچنین به positional encoding برای تاثیر گذاشتن موقعیت در بازنمایی ورودی نیاز داریم. در انتها به Encoder, Decoder شبکه Transformer برای تولید خروجی نیاز است.

```
class Transformer(nn.Module):
    def __init__(self, bert_model, vocab_size, hidden_size, num_encoder_layers=2,
num_decoder_layers=2, nhead=2, dim_feedforward=1024, dropout=0.1):
        super(Transformer, self).__init__()
        self.bert = bert_model
        self.linear_transformation = nn.Linear(bert_model.config.hidden_size,
hidden_size)
        self.positional_encoding = PositionalEncoding(hidden_size)
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.transformer = nn.Transformer(d_model=hidden_size, nhead=nhead,
num_encoder_layers=num_encoder_layers,
                                          num_decoder_layers=num_decoder_layers,
dim_feedforward=dim_feedforward, dropout=dropout)
        self.fc_out = nn.Linear(hidden_size, vocab_size)
```

در تابع forward این کلاس، بعد از بدست آوردن Embedding توکن‌های دنباله ورودی، بازنمایی بدست آمده را با لایه fully connected به اندازه مناسب برای شبکه تصویر می‌کنیم. بعد از این انتقال، از positional encoding استفاده می‌شود. از طرف دیگر توکن مدنظر برای پیش‌بینی (در دنباله خروجی) نیز به یک لایه Embedding جدید داده می‌شود تا بازنمایی مناسب آن برای شبکه یادگرفته و پیدا شود. روی این بازنمایی نیز positional encoding انجام می‌شود. سپس با استفاده از transformer خروجی تولید می‌شود و به vocab size تصویر می‌شود.

```
    def forward(self, src_input_ids, src_attention_mask, tgt_input_ids):
        with torch.no_grad():
            bert_embedding_outputs = self.bert(input_ids=src_input_ids,
attention_mask=src_attention_mask)[0]
        encoder_input = self.linear_transformation(bert_embedding_outputs)
        encoder_input = self.positional_encoding(encoder_input)
        encoder_input = encoder_input.transpose(0, 1)

        tgt_embeddings = self.embedding(tgt_input_ids)
        tgt_embeddings = self.positional_encoding(tgt_embeddings)
        tgt_embeddings = tgt_embeddings.transpose(0, 1)

        tgt_mask =
nn.Transformer.generate_square_subsequent_mask(tgt_embeddings.size(0)).to(tgt_input_i
ds.device)
        output = self.transformer(src=encoder_input, tgt=tgt_embeddings,
tgt_mask=tgt_mask)

        output = output.transpose(0, 1)
        output = self.fc_out(output)

        return output
```

بعد از انجام این مراحل، به تابع آموزش مدل می‌رسیم. برای آموزش مدل از `Adam optimizer` استفاده شده و همچنین برای تابع زیان نیز `cross entropy` در نظر گرفته شده. همچنین با توجه به اینکه در هنگام `padding` به توکن‌هایی که اهمیت معنایی در جمله نداشتند ۰ دادیم، در اینجا لاس مرتبط با آنان را در نظر نمی‌گیریم.

```
def train_transformer_model(transformer, train_loader, vocab_size, num_epochs=10,
learning_rate=1e-4, device='cuda'):
    transformer = transformer.to(device)

    optimizer = optim.Adam(transformer.parameters(), lr=learning_rate,
weight_decay=0.01)
    criterion = nn.CrossEntropyLoss(ignore_index=0)  # Ignore padding token with
index 0
```

برای آموزش `transformer` ابتدا آن را روی حالت `training` قرار می‌دهیم. سپس در یک حلقه برای هر اپاک، به صورت `batch` به `batch` پیش می‌رویم. برای اولین توکن، که توکنی قبل از آن تولید نشده به عنوان ورودی به مدل یک بردار توکن ۰ می‌دهیم. در یک حلقه به طول دنباله هر بار پیش‌بینی مدل را دریافت کرده و برای دور بعدی در حلقه، پیش‌بینی دفعه قبلی خودش (محتمل‌ترین خروجی از نظر

مدل) را ورودی می‌دهیم. در واقع در پیاده‌سازی انجام شده برای transformer از teacher forcing بهره برده نشده. (البته در ابتدا از teacher forcing استفاده شد، اما در رویکردی که از teacher forcing استفاده شد، بعد از کاهش یافتن و نزدیک شدن teacher forcing ratio به صفر، مدل با استفاده از خروجی خودش بهبودهای بسیار کندی داشت و از این روی قادر به ادامه دادن فرآیند آموزش در این حالت نبودم؛ اما در حالتی که از ابتدا teacher forcing را در نظر نگرفتم، با اینکه در شروع سرعت بهبود کندتری داشت ولی با سرعت بهتری بهبود می‌یافت.)

```
    for epoch in range(num_epochs):
        total_loss = 0

        for batch_idx, (input_ids, attention_mask, target_ids) in
enumerate(train_loader):
            input_ids, attention_mask, target_ids = input_ids.to(device),
attention_mask.to(device), target_ids.to(device)

            # Prepare decoder input and target
            decoder_input = torch.zeros(target_ids.size(0), 1, dtype=torch.long,
device=device)  # Start token (assuming index 0 is <sos>)
            decoder_target = target_ids

            # Reset gradients
            optimizer.zero_grad()

            # Forward pass through transformer with teacher forcing
            seq_length = target_ids.size(1)
            outputs = []
            for t in range(seq_length):
                output = transformer(input_ids, attention_mask, decoder_input)
                outputs.append(output[:, -1:, :])
                top1 = output[:, -1, :].argmax(1, keepdim=True)
                decoder_input = torch.cat([decoder_input, top1], dim=1)
```

در انتها مقادیر پیش‌بینی شده توسط مدل با مقادیر درست مقایسه می‌شوند. خطا حساب شده و backward می‌شوند و در نهایت وزن‌ها بروزرسانی می‌شوند.

```
            outputs = torch.cat(outputs, dim=1)
            outputs = outputs.view(-1, vocab_size)
            decoder_target = decoder_target.view(-1)

            # Compute loss
            loss = criterion(outputs, decoder_target)
            total_loss += loss.item()

            # Backward pass and optimization
            loss.backward()
```

```
            optimizer.step()

            if batch_idx % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step
[{batch_idx}/{len(train_loader)}], Loss: {loss.item():.4f}, Teacher Forcing Ratio:
{teacher_forcing_ratio:.4f}')

        # Print epoch loss
        avg_loss = total_loss / len(train_loader)
        print(f'Epoch [{epoch+1}/{num_epochs}], Average Loss: {avg_loss:.4f}')
```

برای آموزش مدل، اینبار از تعداد اپاک های کمتری استفاده شد. علت این موضوع، زمان‌برتر بودن آموزش این مدل و همچنین روند آموزش کندتر آن بود. از این روی احتمالا این پیاده‌سازی با مدت زمان آموزش بیشتر (بیشتر از ۶ اپاک)، نتایج بهتری دریافت خواهد کرد.

ارزیابی

در ادامه به همین شکل تابع `evaluation` ولی با تفاوت در خروجی ایجاد شد. در این تابع، خروجی `preds, true_labels` می‌باشد تا بتوان با استفاده از آنان، مقادیر `precision, recall, f1 score` را حساب کرد.

با استفاده از تابع `evaluation` مقادیر پیش‌بینی شده توسط مدل و مقادیر درست را می‌گیریم.

```
val_preds_transformer, val_true_labels_transformer =
evaluate_transformer_model(transformer, val_loader,vocab_size=output_dim)
```

```
val_preds_transformer = np.array(val_preds_transformer)
val_true_labels_transformer = np.array(val_true_labels_transformer)
```

با دیکد کردن و تبدیل کردن اعداد به توکن‌های متنی در کد زیر:

```
 val_pred_decoded_transformer = [label_tokenizer.decode(pred) for pred in
val_preds_transformer]

val_true_labels_decoded_transformer = [label_tokenizer.decode(label) for label in
val_true_labels_transformer]

# Print a few decoded samples
for i in range(0, 20):
    print(f'val_pred_transformer: {val_pred_decoded_transformer[i]}')
    print(f'val_true_label_transformer: {val_true_labels_decoded_transformer[i]}')
    print('-----------------------------------------')
```

به خروجی زیر می‌رسیم.

```
val_pred_transformer: مفاعیلن مفاعیلن فعولن
val_true_label_transformer: فعلاتن مفاعلن فعلن
-----------------------------------------
val_pred_transformer: فعلاتن فعلاتن مفاعیل
val_true_label_transformer: فاعلاتن فاعلاتن فاعلن
-----------------------------------------
val_pred_transformer: فعولن فعولن فعولن فعل
val_true_label_transformer: فعلات فاعلاتن فعلات فاعلاتن
```

حال به قسمت ارزیابی و محاسبه معیارها می‌رسیم. در این بخش ابتدا به صورت توکن به توکن ارزیابی‌ها را انجام دادم. برای اینکار از کتابخانه sklearn استفاده کردم و به صورت توکن به توکن مقادیر f1score, precision, recall و غیره را محاسبه کردم. برای محاسبه از حالت macro استفاده شده که با استفاده از این رویکرد، تاثیر کلاس‌های با sampleکم کاهش نمی‌یابد چرا که هر معیار در سطح کلاس حساب شده و در نهایت میانگین گرفته می‌شود.

```python
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score
import numpy as np

# Calculate metrics
val_preds_flat_transformer = val_preds_transformer.ravel()
val_true_labels_flat_transformer = val_true_labels_transformer.ravel()

accuracy_transformer = accuracy_score(val_true_labels_flat_transformer,
val_preds_flat_transformer)
f1_transformer = f1_score(val_true_labels_flat_transformer,
val_preds_flat_transformer, average='macro', zero_division=1)
recall_transformer = recall_score(val_true_labels_flat_transformer,
val_preds_flat_transformer, average='macro', zero_division=1)
precision_transformer = precision_score(val_true_labels_flat_transformer,
val_preds_flat_transformer, average='macro', zero_division=1)

print(f"Accuracy: {accuracy_transformer:.4f}")
print(f"F1 Score: {f1_transformer:.4f}")
print(f"Recall: {recall_transformer:.4f}")
print(f"Precision: {precision_transformer:.4f}")
```

خروجی به صورت زیر می‌باشد.

```
Accuracy: 0.4270
F1 Score: 0.2397
Recall: 0.2624
Precision: 0.6441
```

اما این شیوه برای ارزیابی تسک دنباله به دنباله چندان مناسب نیست و برای classification مناسب است. از این روی، در ادامه به بررسی همین معیارها اما اب استفاده از حالت n-gram آنان می‌پردازیم. یعنی به صورت دنباله n-gram ای چک می‌کنیم که چه تعداد از دنباله‌های درست در دنباله پیش‌بینی شده هستند (برای recall) و چه تعداد از دنباله‌های پیش‌بینی شده در بین دنباله‌های درست هستند (برای precision).

برای این کار از کتابخانه torchmetrics استفاده می‌کنیم. معیار BLEU و ROUGE این مقادیر را به ما خروجی می‌دهند.

```
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

nltk.download('punkt_tab')

import torchmetrics
from torchmetrics.text import BLEUScore, ROUGEScore

# BLEU and ROUGE scores
bleu = BLEUScore()
rouge = ROUGEScore()

val_pred_str_transformer = [' '.join(map(str, pred)) for pred in
val_preds_transformer]
val_true_str_transformer = [' '.join(map(str, true)) for true in
val_true_labels_transformer]

print(f'BLEU Score: {bleu(val_pred_str_transformer, [[true] for true in
val_true_str_transformer])}')
print(f'ROUGE Score: {rouge(val_pred_str_transformer, val_true_str_transformer)}')
```

خروجی معیارها به صورت زیر می‌باشند. معیارهای rouge1 نشان‌دهنده معیار بر اساس uni-gram های دو مشترک دو دنباله است. در حالی که rouge2 براساس bigram های مشترک دو دنباله است و rougeL بلندترین اشتراک دو دنباله است.

```
BLEU Score: 0.24592259526252747

ROUGE Score: {'rouge1_fmeasure': tensor(0.4468), 'rouge1_precision':
tensor(0.4468), 'rouge1_recall': tensor(0.4468), 'rouge2_fmeasure':
tensor(0.2844), 'rouge2_precision': tensor(0.2844), 'rouge2_recall':
tensor(0.2844), 'rougeL_fmeasure': tensor(0.4467), 'rougeL_precision':
tensor(0.4467), 'rougeL_recall': tensor(0.4467), 'rougeLsum_fmeasure':
tensor(0.4467), 'rougeLsum_precision': tensor(0.4467), 'rougeLsum_recall':
tensor(0.4467)}
```

همان‌طور که ملاحظه می‌کنیم بنظر روند آموزش کندتر باعث شده است که نتیجه این قسمت چندان مناسب نباشد.

پیش‌بینی داده‌های آزمون

در نهایت برای این بخش نیز تابع transformer_prediction پیاده‌سازی شده که مانند evaluation، ابتدا مدل را در حالت eval قرار داده و سپس در هر batch ورودی، توکن به توکن پیش‌بینی را انجام و ذخیره می‌کند.

```
def transformer_prediction(transformer, dataloader, device='cuda'):
    transformer = transformer.to(device)
    transformer.eval()
    predicted_metres = []

    with torch.no_grad():
        for batch in dataloader:
            input_ids, attention_mask = batch
            input_ids, attention_mask = input_ids.to(device),
attention_mask.to(device)
            decoder_input = torch.zeros(input_ids.size(0), 1,
dtype=torch.long, device=device)

            batch_predictions = []
            for t in range(10):
                output = transformer(input_ids, attention_mask,
decoder_input)
                top1 = output[:, -1, :].argmax(1, keepdim=True)
                decoder_input = torch.cat([decoder_input, top1], dim=1)
                batch_predictions.append(top1.squeeze(1).cpu().numpy())

            batch_predictions = list(map(list, zip(*batch_predictions)))

    return predicted_metres
```

در نهایت به صورت خروجی‌ها برای مجموعه داده آزمون در فایل
test_samples_seq_to_seq_transformer_results.csv ذخیره شده اند.

```
test_predictions_transformer = transformer_prediction(transformer,
test_loader)
test_prediction_decoded_transformer = [label_tokenizer.decode(pred) for
pred in test_predictions_transformer]

test_data['predicted_metre'] = test_prediction_decoded_transformer
test_data.to_csv('test_samples_seq_to_seq_transformer_results.csv',
index=False)
```

د) رویکرد fine-tuning روی mt۵

در این بخش با استفاده از یک شبکه از پیش‌آموزش دیده سعی کردم آزمایش دیگری انجام دهم تا ببینم آیا به نتیجه قوی‌تر و بهتری نسبت به رویکردهای پیشین می‌رسم یا خیر.

در این راستا، شبکه از پیش‌آموزش دیده mt5 را مدنظر قرار دادم. این شبکه، همان مدل t5 است که روی تسک‌های بسیاری در زبان‌های مختلف آموزش دیده است. قابل ذکر است که mt5 روی متون فارسی نیز آموزش دیده و از این جهت این گزینه را برای fine-tuning انتخاب کردم.

نتایج نهایی این رویکرد از رویکردهای پیشین تا حد خوبی بهتر است که در ادامه به پیاده‌سازی و نتایج آن می‌پردازیم.

پیش‌پردازش و آماده‌سازی

ابتدا مدل از پیش‌آموزش دیده mt5 را دریافت و لود کرده‌ام. از آن‌جا که این مدل روی داده‌های فارسی یز آموزش دیده است، توکنایزر آن نیز برای توکنایز کردن دنباله ورودی مناسب است.

```
mt5_tokenizer = MT5Tokenizer.from_pretrained("google/mt5-small")

mt5_model = MT5ForConditionalGeneration.from_pretrained("google/mt5-small").to(device)
```

اما برای دنباله خروجی، به دلیل اینکه مجموعه توکن‌های ممکن بسیار کمتر هستند، بنظر چندان مناسب نیست. از این روی از همان توکنایزری که در بخش‌های پیشین نیز مطرح کردم و توضیح دادم استفاده شده است. در این توکنایزر ابتدا توکن‌های منحصر به فرد متن پیدا می‌شوند. سپس به آن توکن‌ها EOSنیز اضافه می‌شود. در ادامه یک نگاشت از توکن‌هها به آیدی و برعکس محاسبه می‌شود.

با ورودی دادن متن، به ازای هر توکن در دنباله، عدد مروبطه به آن خروجی دادن می‌شود. برای اینکه طول دنباله‌ها یکسان باشند، تا max_length مشخص شده به آن آیدی ۰ اضافه شده که اصطلاحا برای padding است.

```
class SimpleSpaceTokenizer:
    def __init__(self):
        self.token2id = {}
        self.id2token = {}
        self.vocab_size = 0
        self.eos_token_id = None

    def fit_on_texts(self, texts):
        unique_tokens = set()
        for text in texts:
            tokens = text.split(" ")
            unique_tokens.update(tokens)

        self.token2id = {token: idx for idx, token in enumerate(unique_tokens, start=1)}
        self.eos_token_id = len(self.token2id) + 1  # Assign a unique ID to the EOS token
        self.token2id["<EOS>"] = self.eos_token_id

        self.id2token = {idx: token for token, idx in self.token2id.items()}
        self.vocab_size = len(self.token2id) + 1  # Adding 1 for padding token

    def tokenize(self, texts, max_length=48):
```

```
        tokenized_texts = []
        for text in texts:
            tokens = text.split(" ")
            token_ids = [self.token2id.get(token, 0) for token in tokens]
            token_ids = token_ids[:max_length - 1]  # Reserve space for EOS token
            token_ids.append(self.eos_token_id)  # Add EOS token

            padding_length = max_length - len(token_ids)
            token_ids += [0] * padding_length  # Add padding tokens
            tokenized_texts.append(token_ids)
        return torch.tensor(tokenized_texts)

    def decode(self, token_ids):
        return " ".join([self.id2token.get(token_id, "") for token_id in token_ids if token_id
!= 0 and token_id != self.eos_token_id])
```

در ادامه با استفاده از این دو توکنایزر، مجموعه داده‌های `training` مان را لود کرده‌ام. در این قسمت و از آن‌جا که مدل `mt5` روی داده‌های متنی فارسی آموزش دیده است، به قبل و بعد از آن یک تیکه متن اضافه می‌شود که مطابق زیر است.

```
train_data = pd.read_csv('/content/drive/MyDrive/LanguageUnderstanding/HW1/train_samples.csv')
poem_text = " وزن مصرع داده شده" + train_data['poem_text'] + " برابر است با "
```

در ادامه این دنباله به توکنایزر `mt5` و دنباله خروجی به توکنایزر `SimpleSpaceTokenizer` داده می‌شود. از توکنایزر `mt5` آن مقادیری که برای `padding` اضافه شده بودند، با `mask` مشخص شده اند.

```
inputs = mt5_tokenizer(poem_text.tolist(), padding=True, truncation=True, return_tensors="pt",
max_length=18)
input_ids = inputs['input_ids'].squeeze().to(device)
attention_mask = inputs['attention_mask'].squeeze().to(device)

# Process labels
metre = train_data['metre'].astype(str)
label_tokenizer = SimpleSpaceTokenizer()
label_tokenizer.fit_on_texts(metre.tolist())
labels = label_tokenizer.tokenize(metre.tolist(), max_length=7).to(device)
```

حال مجموعه داده‌های آموزشی را داخل یک `DataLoader` می‌ریزیم و آماده ورودی داده شدن به مدل می‌کنیم.

```
train_loader = DataLoader(torch.utils.data.TensorDataset(input_ids, attention_mask, labels),
batch_size=320, shuffle=True)
```

برای مجموعه داده‌های اعتبارسنجی و آزمون نیز به شکل مشابه عمل می‌کنیم.

فرآیند fine-tuning

بعد از آماده‌سازی داده‌ها به مرحله `fine-tune` کردن مدل می‌رسیم. برای این مرحله الگوریتم بهینه سازی `AdamW` و خطای `crossentropy` استفاده شده. همچنین با `scheduler` تنظیم شده که نرخ یادگیری به مرور کاهش بیابد.

```
def train_mt5(model, train_loader, val_loader, epochs=3, lr=1e-3):
    model.train()
    optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.90)
    loss_fn = nn.CrossEntropyLoss()

    teacher_forcing_ratio = 1.0
```

در ادامه و در هر اپاک، داده‌ها را دسته دسته لود می‌کنیم.

```
    for epoch in range(epochs):
        model.train()
        total_loss = 0.0
        for i, batch in enumerate(train_loader):
            input_ids, attention_mask, labels = batch
            input_ids, attention_mask, labels = input_ids.to(device),
attention_mask.to(device), labels.to(device)
```

سپس آن را توکن به توکن به مدل می‌دهیم تا توکن بعدی را پیش‌بینی کند. ورودی اول یک توکن با مقدار ۰ است. در ادامه یا خروجی قبلی مدل را بدان ورودی می‌دهیم و یا برچسب درست را بدان ورودی می‌دهیم (`teacher forcing`).

```
            decoder_input_ids = torch.full(
                (input_ids.size(0), 1), model.config.decoder_start_token_id, dtype=torch.long,
device=device
            )

            loss = torch.tensor(0.0, device=device)
            for t in range(labels.size(1) - 1):
                # Forward pass
                outputs = model(input_ids=input_ids, attention_mask=attention_mask,
decoder_input_ids=decoder_input_ids)
                logits = outputs.logits[:, -1, :]  # Get logits for the last generated token

                target_token = labels[:, t].view(-1)  # Ensure correct shape
                loss += loss_fn(logits, target_token)

                use_teacher_forcing = random.random() < teacher_forcing_ratio

                if use_teacher_forcing:
                    next_token = target_token.unsqueeze(1)
                else:
                    next_token = torch.argmax(logits, dim=-1).unsqueeze(1)
```

```
                decoder_input_ids = torch.cat([decoder_input_ids, next_token], dim=1)
```

در ادامه خطا محاسبه و بازگردانده می‌شود.

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item() / (labels.size(1) - 1)  # Average loss per token
            print(f'batch {i}/ {len(train_loader)}, loss training: {loss:.4f}')
```

همچنین در انتهای هر اپاک، یکبار روی مجموعه داده اعتبار سنجی این مقدادیر پیش‌بینی می‌شوند. برای این کار مدل روی حالت `eval` قرار داده می‌شود. سپس داده‌های اعتبارسنجی به صورت دسته‌ای به مدل ورودی داده می‌شوند و روی این مجموعه داده‌ها توکن به توکن پیش‌بینی صورت می‌گیرد و خطا حساب می‌شود. در این بخش از `early stopping` استفاده نشده و تنها خطا روی این مجوعه داده‌ها محاسبه و گزارش شده است.

```
        # Validation
        model.eval()
        val_loss = torch.tensor(0.0, device=device)
        with torch.no_grad():
            for batch in val_loader:
                input_ids, attention_mask, labels = batch
                input_ids, attention_mask, labels = input_ids.to(device),
attention_mask.to(device), labels.to(device)


                # Start token
                decoder_input_ids = torch.full(
                    (input_ids.size(0), 1), model.config.decoder_start_token_id,
dtype=torch.long, device=device
                )

                loss = torch.tensor(0.0, device=device)
                for t in range(labels.size(1) - 1):
                    outputs = model(input_ids=input_ids, attention_mask=attention_mask,
decoder_input_ids=decoder_input_ids)
                    logits = outputs.logits[:, -1, :]

                    target_token = labels[:, t]
                    loss += loss_fn(logits, target_token)

                    next_token = target_token.unsqueeze(1)
                    decoder_input_ids = torch.cat([decoder_input_ids, next_token], dim=1)

                val_loss += loss.item() / (labels.size(1) - 1)
        avg_val_loss = val_loss.item() / len(val_loader)
        print(f"Epoch {epoch + 1}/{epochs}, Validation Loss: {avg_val_loss:.4f}")
```

در انتها با استفاده از کد زیر، مدل را روی این مجموعه داده fine-tune کرده‌ام.

```
train_mt5(mt5_model, train_loader, val_loader, epochs=1, lr=1e-3)
```

ارزیابی

برای اینکه نتایج خروجی روی مجموعه داده‌های اعتبارسنجی بدست بیاید، از تابع evaluate_mt5 استفاده شده. این تابع مانند حلقه آموزش است و با قرار دادن مدل در حالت eval ، دنباله خروجی را توکن به توکن تولید می‌کند و سپس آرایه‌ای از پیش‌بینی‌های دنباله‌ها و آرایه‌ای از برچسب‌های درست را خروجی می‌دهد.

```
def evaluate_mt5(model, dataloader, device='cuda'):
    model = model.to(device)
    model.eval()

    preds = []
    true_labels = []

    with torch.no_grad():
        for batch_idx, (input_ids, attention_mask, target_ids) in enumerate(dataloader):
            input_ids, attention_mask, target_ids = input_ids.to(device),
attention_mask.to(device), target_ids.to(device)

            # Prepare decoder input
            decoder_input = torch.zeros(target_ids.size(0), 1, dtype=torch.long,
device=device)  # Start token (assuming index 0 is <sos>)

            batch_preds = []
            seq_length = target_ids.size(1)

            for t in range(seq_length):
                outputs = model(input_ids=input_ids, attention_mask=attention_mask,
decoder_input_ids=decoder_input)
                top1 = outputs.logits[:, -1, :].argmax(1, keepdim=True)  # Greedy decoding to
use as next input
                decoder_input = torch.cat([decoder_input, top1], dim=1)
                batch_preds.append(top1.squeeze(1).cpu().tolist())

            # Collect predictions and true labels
            batch_preds = list(map(list, zip(*batch_preds)))  # Transpose to match batch-wise
structure
            preds.extend(batch_preds)
            true_labels.extend(target_ids.cpu().tolist())

    return preds, true_labels
```

حال با گرفتن خروجی درست و خروجی پیش‌بینی شده برای چند نمونه آنان را پرینت کرده‌ام.

```
val_pred_mt5: مفعول فاعلاتن مفعول فاعلاتن
val_true_label_mt5: مفعول فاعلاتن مفعول فاعلاتن
----------------------------------------
val_pred_mt5: مفاعیلن مفاعیلن فعولن
val_true_label_mt5: مفاعیلن مفاعیلن فعولن
----------------------------------------
val_pred_mt5: فعولن فعولن فعولن فعل
val_true_label_mt5: فعولن فعولن فعولن فعل
----------------------------------------
val_pred_mt5: مفاعیلن مفاعیلن فعولن
val_true_label_mt5: مفاعیلن مفاعیلن فعولن
```

با توجه به دنباله خروجی پیش‌بینی شده توسط مدل و دنباله درست، بنظر می‌رسد که نتیجه این قسمت از سایرین بهتر است. حال برای بررسی کمی این نتایج، از معیارهای ارزیابی استفاده می‌کنیم. برای این کار ابتدا بررسی توکن به توکن را انجام می‌دهیم. برای اینکار از `sklearn` استفاده می‌کنیم. البته این رویکرد (بررسی توکن به توکن) برای یک تسک دنباله به دنباله درست نیست ولی در اینجا با استفاده از کد زیر محاسبه و گزارش شده است.

```python
import torchmetrics
from torchmetrics.text import BLEUScore, ROUGEScore
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score
import numpy as np

val_preds_flat_mt5 = val_preds_mt5.ravel()
val_true_labels_flat_mt5 = val_true_labels_mt5.ravel()

accuracy_mt5 = accuracy_score(val_true_labels_flat_mt5, val_preds_flat_mt5)
f1_mt5 = f1_score(val_true_labels_flat_mt5, val_preds_flat_mt5, average='macro',
zero_division=1)
recall_mt5 = recall_score(val_true_labels_flat_mt5, val_preds_flat_mt5, average='macro',
zero_division=1)
precision_mt5 = precision_score(val_true_labels_flat_mt5, val_preds_flat_mt5, average='macro',
zero_division=1)

print(f"Accuracy: {accuracy_mt5:.4f}")
print(f"F1 Score: {f1_mt5:.4f}")
print(f"Recall: {recall_mt5:.4f}")
print(f"Precision: {precision_mt5:.4f}")
```

نتایج این قسمت توکن به توکن به صورت زیر می‌باشد.

```
Accuracy: 0.8860
F1 Score: 0.7356
Recall: 0.7277
Precision: 0.8036
```

حال به معیارهای ارزیابی دنباله به دنباله می‌رسیم. برای اینکار از معیارهای ارزیابی BLEU, ROUGE استفاده می‌کنیم. این معیارها برخلاف رویکرد قبلی به صورت توکن به توکن ارزیابی نمی‌کنند و همان‌طور که در بخش‌های قبلی توضیح داده شده به صورت n-gram ای ارزیابی را انجام می‌دهند. به عنوان مثال برای recall rouge بررسی می‌شود که از دنباله‌های n-gram ای موجود در دنباله درست چند تایش در n-gram های دنباله پیش‌بینی شده می‌باشد. بر این اساس در کد زیر از این دو معیار برای ارزیابی نتایج استفاده شده است.

```
# BLEU and ROUGE scores
bleu = BLEUScore()
rouge = ROUGEScore()

val_pred_str_mt5 = [' '.join(map(str, pred)) for pred in val_preds_mt5]
val_true_str_mt5 = [' '.join(map(str, true)) for true in val_true_labels_mt5]

print(f'BLEU Score: {bleu(val_pred_str_mt5, [[true] for true in val_true_str_mt5])}')
print(f'ROUGE Score: {rouge(val_pred_str_mt5, val_true_str_mt5)}')
```

نتایج به صورت زیر هستند. در معیار rouge1 بر اساس unigram، در معیار rouge2 براساس bigram و در معیار RougeL براساس بلندترین اشتراک بین دو دنباله خروجی محاسبه شده است.

```
BLEU Score: 0.8573588728904724

ROUGE Score: BLEU Score: 0.8573588728904724

ROUGE Score: {'rouge1_fmeasure': tensor(0.9123), 'rouge1_precision':
tensor(0.9123), 'rouge1_recall': tensor(0.9123), 'rouge2_fmeasure':
tensor(0.8744), 'rouge2_precision': tensor(0.8744), 'rouge2_recall':
tensor(0.8744), 'rougeL_fmeasure': tensor(0.9123), 'rougeL_precision':
tensor(0.9123), 'rougeL_recall': tensor(0.9123), 'rougeLsum_fmeasure':
tensor(0.9123), 'rougeLsum_precision': tensor(0.9123), 'rougeLsum_recall':
(tensor(0.9123)}
```

با توجه به نتایج بدست آمده در هر دو شیوه ارزیابی، بنظر نتایج این رویکرد تا حد بسیارخوبی از نتایج بخش‌های دیگر بهتر است و به دقت بیشتر از ۹۰ درصد رسیده.

پیش‌بینی داده‌های آزمون

در انتها با استفاده از تابعی مشابه با evaluation برای مجموعه داده‌های تست پیش‌بینی‌ها را انجام می‌دهیم. در این تابع نیز ابتدا مدل را روی حالت eval قرار می‌دهیم تا گرادیان‌ها حساب نشوند و بروزرسانی صورت نگیرد. سپس برای هر دسته داده در داده‌های آزمون، به صورت توکن به توکن پیش‌بینی را انجام می‌دهیم. هر بار آن خروجی که بیشترین احتمال را دارد به عنوان خروجی در نظر گرفته و به عنوان ورودی گام بعدی می‌دهیم. در انتها تمامی این مقادیر پیش‌بینی شده را ذخیره کرده و خروجی می‌دهیم.

```
def mt5_prediction(model, dataloader, device='cuda'):
    model = model.to(device)
    model.eval()
    predicted_metres = []

    with torch.no_grad():
        for batch in dataloader:
            input_ids, attention_mask = batch
            input_ids, attention_mask = input_ids.to(device), attention_mask.to(device)

            decoder_input = torch.zeros(input_ids.size(0), 1, dtype=torch.long,
device=device)  # Start token

            batch_predictions = []
            for t in range(10):
                outputs = model(input_ids=input_ids, attention_mask=attention_mask,
decoder_input_ids=decoder_input)
                top1 = outputs.logits[:, -1, :].argmax(1, keepdim=True)  # Greedy
                decoder_input = torch.cat([decoder_input, top1], dim=1)
                batch_predictions.append(top1.squeeze(1).cpu().numpy())

            batch_predictions = list(map(list, zip(*batch_predictions)))
            predicted_metres.extend(batch_predictions)

    return predicted_metres
```

با اجرای این کد، خروجی آن را در فایل `test_samples_seq_to_seq_transformer_mt5_results.csv` ذخیره می‌کنیم.

```
test_predictions_mt5 = mt5_prediction(mt5_model, test_loader, device=device)

test_prediction_decoded_mt5 = [label_tokenizer.decode(pred) for pred in test_predictions_mt5]

test_data['predicted_metre'] = test_prediction_decoded_mt5
test_data.to_csv('test_samples_seq_to_seq_transformer_mt5_results.csv', index=False)
```

نتیجه گیری

با توجه به نتایج بدست آمده در قسمت‌های قبل مشاهده می‌شود که در حالتی که از fine-tune کردن مدل mt5 استفاده کنیم، بنظر به نتایج بهتری می‌رسیم. بعد از این رویکرد، بنظر رویکرد classification و سپس با اختلاف کمی rnn seq-to-seq مناسب هستند.

استفاده از رویکرد classification برای این مسئله چند نکته مثبت دارد:

۱- نیازمند داده‌های کمتری برای آموزش دیدن است؛ چرا که تسک ساده‌تری را یاد می‌گیرد. البته قطعا اگر خروجی ما فقط به ۴۸ حالت خلاصه نمی‌شد رویکرد دسته‌بندی اصلا مناسب نبود. ولی در این مسئله و با توجه به اینکه خروجی ما تنها ۴۸ حالت مختلف می‌تواند داشته باشد، رویکرد دسته‌بندی می‌تواند مناسب باشد.
۲- رویکرد دسته‌بندی به ارور کمتری می‌خورد و ساده‌تر است؛ چرا که در حالتی که به صورت دنباله به دنباله به این مسئله نگاه کنیم، ممکن است بعضی از خروجی‌های تولید شده اصلا در مجموعه حالات ممکن خروجی نباشند.
۳- از نظر آموزش سبک‌تر، راحت‌تر و سریع‌تر است.

اما نکته منفی دسته‌بندی، در این است که ممکن است ارتباط بین توکن‌های موجود در دنباله خروجی را واقعا به شکل درستی لحاظ نکند. امکان دارد که این مورد باعث گردد که از حدی نتواند بهتر عمل کند.

رویکرد sequence-to-sequence در مقابل، ارتباط بین توکن‌ها در خروجی را فرا می‌گیرد و این نکته مثبتی محسوب می‌گردد. اما آموزش آن دشوارتر و هزنیه‌برتر است. فرآیند آموزش آن طولانی‌تر است و به داده‌های بیشتری نیاز دارد چون تسک سخت‌تری دارد. در این حالت مدل دارد تولید دنباله خروجی را به شکل کامل یاد می‌گیرد و نه صرفا پیدا کردن مرز بین کلاس‌ها را.

در انتها بنظر می‌رسد که مدلی که روی تسک‌های دیگر آموزش دیده باشد و generalization بهتری داشته باشد، از رویکردهای دیگر تا حدی بهتر است. در انتها و در بخش د مشاهده کردیم که fine-tune کردن mt5 بهترین نتیجه را می‌دهد.