AMS691.02: Natural Language Processing – Fall 2024
Assignment 3

Vishesh Kumar
SBU ID: 115972814

Introduction

In this assignment, we utilize BERT features to classify DBPedia articles into categories. We implement a classifier that leverages pre-trained BERT embeddings and fine-tune it for our classification task. The dataset is pre-processed, and we use PyTorch along with the HuggingFace Transformers library for implementation.

Setup

We begin by installing the necessary libraries and importing modules.

```
!pip install tqdm boto3 requests regex sentencepiece sacremoses
!pip install transformers
```

Listing 1: Installing Libraries

```
import collections
import json
import torch
import torch.nn as nn
import tqdm
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer, AutoModel
import numpy as np
import pandas as pd
import random
```

Listing 2: Importing Modules

Data Preparation

We define a custom dataset class for DBPedia and construct data loaders.

```python
SPLITS = ['train', 'dev', 'test']

class DBPediaDataset(Dataset):
    '''DBPedia dataset.'''
    def __init__(self, path):
        with open(path) as fin:
            self._data = [json.loads(l) for l in fin]
        self._n_classes = len(set([datum['label'] for datum in self._data]))

    def __getitem__(self, index):
        return self._data[index]

    def __len__(self):
        return len(self._data)

    @property
    def n_classes(self):
        return self._n_classes

    @staticmethod
    def collate_fn(tokenizer, device, batch):
        '''Collate function for batching.'''
        labels = torch.tensor([datum['label'] for datum in batch]).long().to(device)
        sentences = tokenizer(
            [datum['sentence'] for datum in batch],
            return_tensors='pt',
            padding=True)
        for key in sentences:
            sentences[key] = sentences[key].to(device)
        return labels, sentences

def construct_datasets(prefix, batch_size, tokenizer, device):
    '''Constructs datasets and data loaders.'''
    datasets = collections.defaultdict()
    dataloaders = collections.defaultdict()
    for split in SPLITS:
        datasets[split] = DBPediaDataset(f'{prefix}{split}.json')
        dataloaders[split] = DataLoader(
            datasets[split],
            batch_size=batch_size,
            shuffle=(split == 'train'),
            collate_fn=lambda x:DBPediaDataset.collate_fn(tokenizer,
```

```
                device, x))
    return datasets, dataloaders
```

Listing 3: DBPedia Dataset Class

Problem 1.1: Implementing the Classifier

We implement a simple classifier with one hidden layer.

```
class Classifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Classifier, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Listing 4: Classifier Implementation

Training and Evaluation

We set hyperparameters and initialize the tokenizer and BERT model.

```
batch_size = 32
classifier_hidden_size = 32

tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
bert_model = AutoModel.from_pretrained('bert-base-cased')
if torch.cuda.is_available():
    bert_model = bert_model.cuda()

datasets, dataloaders = construct_datasets(
    prefix='dbpedia_',
    batch_size=batch_size,
    tokenizer=tokenizer,
    device=bert_model.device)
```

Listing 5: Hyperparameters and Initialization

We define functions for training and evaluation.

```python
import random
import numpy as np
from sklearn.metrics import accuracy_score

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
```

Listing 6: Training and Evaluation Functions

We train the classifier using different random seeds and evaluate its performance.

```python
dev_accuracies = []
test_accuracies = []
seed_values = [42, 123, 999, 2021, 7]
best_dev_accuracy = 0.0
best_model_state = None

for seed in seed_values:
    set_seed(seed)
    classifier = Classifier(
        input_size=bert_model.config.hidden_size,
        hidden_size=classifier_hidden_size,
        num_classes=datasets['train'].n_classes).to(bert_model.device)
    optimizer = torch.optim.Adam(classifier.parameters(), lr=5e-4)
    loss_func = nn.CrossEntropyLoss()
    pbar = tqdm.tqdm(dataloaders['train'])
    for labels, sentences in pbar:
        with torch.no_grad():
            unpooled_features = bert_model(**sentences)['last_hidden_state
                ']
        cls_embeddings = unpooled_features[:, 0, :]
        logits = classifier(cls_embeddings)
        loss = loss_func(logits, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
        pbar.set_description(f"Seed: {seed} | Loss: {loss.item():.4f}")
    # Evaluation on development set
    classifier.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for labels, sentences in dataloaders['dev']:
            unpooled_features = bert_model(**sentences)['last_hidden_state
                ']
            cls_embeddings = unpooled_features[:, 0, :]
            logits = classifier(cls_embeddings)
            preds = torch.argmax(logits, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    dev_accuracy = accuracy_score(all_labels, all_preds)
    dev_accuracies.append(dev_accuracy)
    print(f"Seed: {seed} | Dev Accuracy: {dev_accuracy:.4f}")
    if dev_accuracy > best_dev_accuracy:
        best_dev_accuracy = dev_accuracy
        best_model_state = classifier.state_dict()
        best_seed = seed
```

Listing 7: Training Loop

Results

We compute the mean and standard deviation of development accuracies and evaluate the best model on the test set.

```
mean_dev_accuracy = np.mean(dev_accuracies)
std_dev_accuracy = np.std(dev_accuracies)
print(f"\nMean Dev Accuracy: {mean_dev_accuracy:.4f}")
print(f"Std Dev Accuracy: {std_dev_accuracy:.4f}")

# Evaluate on test set
classifier.load_state_dict(best_model_state)
classifier.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for labels, sentences in dataloaders['test']:
        unpooled_features = bert_model(**sentences)['last_hidden_state']
        cls_embeddings = unpooled_features[:, 0, :]
```

```
        logits = classifier(cls_embeddings)
        preds = torch.argmax(logits, dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
test_accuracy = accuracy_score(all_labels, all_preds)
print(f"\nBest Model Seed: {best_seed} | Test Accuracy: {test_accuracy:.4f
    }")
```

Listing 8: Results

Outputs

```
Seed: 42 | Dev Accuracy: 0.9620
Seed: 123 | Dev Accuracy: 0.9620
Seed: 999 | Dev Accuracy: 0.9690
Seed: 2021 | Dev Accuracy: 0.9610
Seed: 7 | Dev Accuracy: 0.9080

Mean Dev Accuracy: 0.9524
Std Dev Accuracy: 0.0224

Best Model Seed: 999 | Test Accuracy: 0.9740
```

Problem 1.2: Mean and Max Pooling

We modify the classifier to use mean and max pooling over content tokens.

```
class Classifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Classifier, self).__init__()
        self.fc1 = nn.Linear(input_size * 2, hidden_size)  # Input size
            doubled
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
```

```
        return x
```

Listing 9: Modified Classifier for Mean and Max Pooling

We adjust the input size since we concatenate mean-pooled and max-pooled vectors.

```
for labels, sentences in pbar:
    with torch.no_grad():
        outputs = bert_model(**sentences)
        unpooled_features = outputs['last_hidden_state']
        attention_mask = sentences['attention_mask']
        mask_expanded = attention_mask.unsqueeze(-1).expand(
            unpooled_features.size()).float()
        masked_embeddings = unpooled_features * mask_expanded
        sum_embeddings = torch.sum(masked_embeddings, dim=1)
        sum_mask = torch.clamp(mask_expanded.sum(dim=1), min=1e-9)
        mean_pooled = sum_embeddings / sum_mask
        masked_embeddings[mask_expanded == 0] = -1e9
        max_pooled = torch.max(masked_embeddings, dim=1)[0]
        pooled_features = torch.cat((mean_pooled, max_pooled), dim=1)
    logits = classifier(pooled_features)
    loss = loss_func(logits, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    pbar.set_description(f"Seed: {seed} | Loss: {loss.item():.4f}")
```

Listing 10: Training with Mean and Max Pooling

Results

```
Seed: 42 | Dev Accuracy: 0.9360
Seed: 123 | Dev Accuracy: 0.9430
Seed: 999 | Dev Accuracy: 0.9140
Seed: 2021 | Dev Accuracy: 0.9310
Seed: 7 | Dev Accuracy: 0.9080

Mean Dev Accuracy: 0.9264
Std Dev Accuracy: 0.0133

Best Model Seed: 123 | Test Accuracy: 0.9320
```

Problem 1.3: Fine-tuning Last Two Layers of BERT

We fine-tune the last two layers of BERT along with the classifier.

```
classifier = Classifier(
    bert_model.config.hidden_size,
    classifier_hidden_size,
    datasets['train'].n_classes).to(bert_model.device)

for name, param in bert_model.named_parameters():
    if name.startswith('encoder.layer.10') or name.startswith('encoder.
       layer.11'):
        param.requires_grad = True
    else:
        param.requires_grad = False

params_to_optimize = list(classifier.parameters()) + [param for param in
    bert_model.parameters() if param.requires_grad]
optimizer = torch.optim.Adam(params_to_optimize, lr=5e-5)
```

Listing 11: Fine-tuning Setup

Training

We train the model without the `torch.no_grad()` context since we are updating BERT parameters.

```
for labels, sentences in pbar:
    outputs = bert_model(**sentences)
    unpooled_features = outputs['last_hidden_state']
    cls_features = unpooled_features[:, 0, :]
    logits = classifier(cls_features)
    loss = loss_func(logits, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    pbar.set_description(f"Loss: {loss.item():.4f}")
```

Listing 12: Training Loop with Fine-tuning

Results

```
Loss: 0.0640: 100%|| 313/313 [00:55<00:00,  5.63it/s]
```

Problem 1.4: Evaluation with Fine-tuning

We evaluate the fine-tuned model using different seeds.

```
Seed: 42 | Dev Accuracy: 0.9960
Seed: 123 | Dev Accuracy: 0.9940
Seed: 999 | Dev Accuracy: 0.9960
Seed: 2021 | Dev Accuracy: 0.9940
Seed: 7 | Dev Accuracy: 0.9950

Mean Dev Accuracy: 0.9950
Std Dev Accuracy: 0.0009

Best Model Seed: 42 | Test Accuracy: 0.9970
```

Problem 1.5: Replacing BERT with GPT-2

We replace BERT with GPT-2 and adjust the code accordingly.

```python
from transformers import GPT2Tokenizer, GPT2Model

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
tokenizer.pad_token = tokenizer.eos_token

gpt2_model = GPT2Model.from_pretrained('gpt2')
if torch.cuda.is_available():
    gpt2_model = gpt2_model.cuda()
```

Listing 13: Using GPT-2 Model

We adjust the feature extraction method since GPT-2 is autoregressive.

```python
with torch.no_grad():
    outputs = gpt2_model(**sentences)
```

```python
    hidden_states = outputs['last_hidden_state']
    attention_mask = sentences['attention_mask'].unsqueeze(-1).expand(
        hidden_states.size()).float()
    masked_hidden_states = hidden_states * attention_mask
    sum_hidden_states = torch.sum(masked_hidden_states, dim=1)
    sum_mask = attention_mask.sum(dim=1).clamp(min=1e-9)
    features = sum_hidden_states / sum_mask
```

Listing 14: Feature Extraction with GPT-2

Results

```
Seed: 42 | Dev Accuracy: 0.8960
Seed: 123 | Dev Accuracy: 0.8300
Seed: 999 | Dev Accuracy: 0.8910
Seed: 2021 | Dev Accuracy: 0.8920
Seed: 7 | Dev Accuracy: 0.8650

Mean Dev Accuracy: 0.8748
Std Dev Accuracy: 0.0249

Best Model Seed: 42 | Test Accuracy: 0.8900
```

Conclusion

We observe that fine-tuning the last two layers of BERT significantly improves the model's performance. Replacing BERT with GPT-2 yields lower accuracy, likely due to the autoregressive nature of GPT-2 and its suitability for generation tasks rather than classification.