llms

November 28, 2023

1 LLM Feature Extraction and Fine-Tuning for Sentence Classification

[ ]:
```
%%capture
! pip install tqdm boto3 requests regex sentencepiece sacremoses
! pip install transformers
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
import collections
import json
import numpy as np
import torch
import torch.nn as nn
import tqdm
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
from transformers import AutoTokenizer, AutoModel
```

1.1 dataset and data loaders

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


class DBPediaDataset(Dataset):

    '''
    DBPedia dataset.
      Args:
        path[str]: path to the original data.
    '''

    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):

        '''
        The collate function that compresses a training batch.
          Args:
            batch[list[dict[str, Any]]]: data in the batch.
          Returns:
            labels[torch.LongTensor]: the labels in the batch.
            sentences[dict[str, torch.Tensor]]: sentences converted by␣
↪tokenizers.
        '''

        # get target labels
        labels = torch.tensor([datum['label'] for datum in batch]).long().
↪to(device)

        # encode sentences with tokenizer
        sentences = tokenizer(
            [datum['sentence'] for datum in batch],
            return_tensors='pt',  # pt = pytorch style tensor
            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.
      Args:
```

```
        prefix[str]: prefix of the dataset (e.g., dbpedia_).
        batch_size[int]: maximum number of examples in a batch.
        tokenizer: model tokenizer that converts sentences to integer tensors.
        device[torch.device]: the device (cpu/gpu) that the tensor should be on.
      Returns:
        datasets[dict[str, Dataset]]: a dict of constructed datasets.
        dataloaders[dict[str, DataLoader]]: a dict of constructed data loaders.
    '''

    datasets = collections.defaultdict()
    dataloaders = collections.defaultdict()

    for split in SPLITS:

        # dataset
        datasets[split] = DBPediaDataset(f'{prefix}{split}.json')

        # dataloader
        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
```

1.2 classifer architecture

[ ]:
```
class Classifier(nn.Module):



    def __init__(self, in_size, layer_sizes:list, layer_acts:list):

        # call parent constructor
        super(Classifier, self).__init__()

        # construct layers (last layer is output layer)
        self.layers = nn.ModuleList()
        for i, layer_size in enumerate(layer_sizes):
            if i == 0:
                # layer = nn.Linear(in_size, layer_size)
                # layer.weight.data.uniform_(-0.01, 0.01)
                # layer.bias.data.zero_()
                # self.layers.append(layer)
                self.layers.append(nn.Linear(in_size, layer_size))
            else:
```

```
                # layer = nn.Linear(layer_sizes[i-1], layer_size)
                # layer.weight.data.uniform_(-0.01, 0.01)
                # layer.bias.data.zero_()
                # self.layers.append(layer)
                self.layers.append(nn.Linear(layer_sizes[i-1], layer_size))

        # set each layer's activation function
        self.layer_acts = layer_acts


    def forward(self, x):

        for i, layer in enumerate(self.layers):
            x = layer(x)
            x = self.layer_acts[i](x)

        return x
```

1.3 1. BERT [CLS] feature extraction for classification

1.3.1 setup

[ ]:
```
# set hyperparameters
batch_size = 32
classifier_hidden_size = 32

# load BERT tokenizer and model
bert_tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
bert_model = AutoModel.from_pretrained('bert-base-cased')   # https://
 ↪huggingface.co/transformers/v3.0.2/model_doc/auto.html#automodel
if torch.cuda.is_available():  # use GPU if available
    bert_model = bert_model.cuda()

# construct datasets with BERT tokenizer
datasets, dataloaders = construct_datasets(
    prefix='/content/drive/MyDrive/LLMs/data/dbpedia_',
    batch_size=batch_size,
    tokenizer=bert_tokenizer,
    device=bert_model.device
)

# # sanity check
# datasets['train'].__getitem__(0)

# # sanity check
# dtrain0 = next(iter(dataloaders['train']))
# # labels
```

```
# dtrain0[0]
# # sentences
# dtrain0[1].keys()   # ['input_ids', 'token_type_ids', 'attention_mask']
# import pandas as pd
# pd.DataFrame(dtrain0[1]['input_ids'])
```

```
tokenizer_config.json:   0%|          | 0.00/29.0 [00:00<?, ?B/s]

config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]

vocab.txt:   0%|          | 0.00/213k [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/436k [00:00<?, ?B/s]

model.safetensors:   0%|          | 0.00/436M [00:00<?, ?B/s]
```

1.3.2 train and eval util funcs

[ ]:
```
# func for extracting frozen BERT token representations for sentences and␣
↪pooling

def extract_bert_rep(pretrained_lm, sentences:dict, pooling:str):


    # extract frozen BERT token representations for sentences
    with torch.no_grad():   # keep BERT params fixed
        unpooled_features = pretrained_lm(**sentences)['last_hidden_state']   #␣
↪(B, L, D): (batch_size, num token in sentence, BERT rep dimension)


    # get pooled_features across tokens for each sentence
    if pooling == 'first':
        pooled_features = unpooled_features[:, 0, :]   # (B, D)

    elif pooling == 'mean':
        # mask padding tokens (where attention_mask is 0) with nan
        unpooled_features_masked = unpooled_features.masked_fill(
            sentences['attention_mask'].unsqueeze(-1)==0,
            float('nan')
        )
        # max-pooling across tokens (L dimension) in each sentence
        pooled_features = unpooled_features_masked.nanmean(dim=1)   # (B, D)

    elif pooling == 'max':
```

```
        # mask padding tokens (where attention_mask is 0) with -inf
        unpooled_features_masked = unpooled_features.masked_fill(
            sentences['attention_mask'].unsqueeze(-1)==0,
            float('-inf')
        )
        # max-pooling across tokens (L dimension) in each sentence
        pooled_features, _ = unpooled_features_masked.max(dim=1)   # (B, D)


    return pooled_features
```

[ ]:
```
# func for running 1 epoch of training

def train1epoch(classifier, train_dataloader, optimizer, criterion,␣
 ↪pretrained_lm, rep_extractor, pooling):

    # progress bar
    pbar = tqdm.tqdm(train_dataloader)

    # turn on training mode
    classifier.train()

    # reset epoch_loss tracker
    epoch_loss = 0

    # iter through mini-batches to train 1 epoch
    for labels, sentences in pbar:  # sentences is a dict (['input_ids',␣
 ↪'token_type_ids', 'attention_mask']) of tensors of shape (B, L)

        # extract frozen BERT token representations for sentences; get [CLS]␣
 ↪token (first token) representation
        cls_features = rep_extractor(pretrained_lm=pretrained_lm,␣
 ↪sentences=sentences, pooling=pooling)   # (B, D)

        # train
        optimizer.zero_grad()   # zero the gradient buffers
        output = classifier(cls_features)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()   # does the update

        epoch_loss += loss.item()

        # print('\n')
        # print(f'  batch loss: {loss.item()}')
        # print(f'  epoch loss: {epoch_loss}')
```

```
    return epoch_loss
```

[ ]:
```
# eval

def eval(classifier, eval_dataloader, pretrained_lm, rep_extractor, pooling):

    # progress bar
    pbar = tqdm.tqdm(eval_dataloader)

    # turn on eval mode
    classifier.eval()

    # set trackers
    ycorrect = 0
    ytotal = 0

    # turn off gradient calc to reduce memory consumption
    with torch.no_grad():

        # iter through mini-batches in eval_dataloader
        for labels, sentences in pbar:

            # extract frozen BERT token representations for sentences; get␣
↪[CLS] token (first token) representation
            cls_features = rep_extractor(pretrained_lm=pretrained_lm,␣
↪sentences=sentences, pooling=pooling)  # (B, D)

            # get classifier predictions on eval_data
            probs = F.softmax(classifier(cls_features), dim=1)
            ypred = torch.argmax(probs, dim=1)

            # count correct predictions
            ycorrect += torch.sum(torch.eq(ypred, labels)).item()

            # total num of obs in eval_data
            ytotal += len(labels)

    # compute accuracy
    yaccu = ycorrect / ytotal

    return yaccu
```

[ ]:
```
# wrapper for train & eval

def main_process(classifier, name, optimizer, criterion, dataloaders,␣
↪pretrained_lm, rep_extractor, pooling, max_epochs=10, early_stopping=3):
```

```python
    # initialize vars: track metrics
    epoch_losses = []
    train_evals = []
    dev_evals = []

    # initialize vars: track best classifier
    best_dev_eval = 0
    best_classifier_epoch = -1


    # train and eval
    for epoch in range(max_epochs):

        print(f'EPOCH {epoch+1}')

        # train
        print(f'----- TRAIN -----')
        epoch_loss = train1epoch(
            classifier=classifier,
            train_dataloader=dataloaders['train'],
            optimizer=optimizer,
            criterion=criterion,
            pretrained_lm=pretrained_lm,
            rep_extractor=rep_extractor,
            pooling=pooling
        )
        print(f'  epoch loss: {epoch_loss}')
        epoch_losses.append(epoch_loss)

        print(f'----- EVAL -----')

        # # eval on training set
        # train_eval = eval(classifier=classifier,␣
↪eval_dataloader=dataloaders['train'], pretrained_lm=pretrained_lm,␣
↪rep_extractor=rep_extractor, pooling=pooling)
        # print(f'  train accuracy: {train_eval}')
        # train_evals.append(train_eval)

        # eval on dev set
        dev_eval = eval(classifier=classifier,␣
↪eval_dataloader=dataloaders['dev'], pretrained_lm=pretrained_lm,␣
↪rep_extractor=rep_extractor, pooling=pooling)
        print(f'  dev accuracy: {dev_eval}')
        dev_evals.append(dev_eval)

        # update best classifier based on dev eval
```

```
        if dev_eval > best_dev_eval:

            # save state_dict of best classifier so far
            torch.save(classifier.state_dict(), '/content/drive/MyDrive/LLMs/
↪classifiers/'+name+'_best.pth.tar')

            # update which epoch best_classifier is from
            best_classifier_epoch = epoch

            # update best_dev_accu
            best_dev_eval = dev_eval

        print(f'  best classifier from epoch {best_classifier_epoch+1}')

        # early stopping based on dev eval
        if early_stopping is not None:
            if epoch - best_classifier_epoch >= early_stopping:
                print('=== EARLY STOPPING ===')
                break
    # end training epochs


    # load state_dict of best classifier (modifies input classifier in place)
    if epoch != best_classifier_epoch:
        print(f'load best classifier...')
        classifier.load_state_dict(torch.load('/content/drive/MyDrive/LLMs/
↪classifiers/'+name+'_best.pth.tar'))

    # eval best classifier on devtest set
    print(f'----- EVAL -----')
    print(f'eval best classifier on devtest...')
    devtest_eval = eval(classifier=classifier,␣
↪eval_dataloader=dataloaders['test'], pretrained_lm=pretrained_lm,␣
↪rep_extractor=rep_extractor, pooling=pooling)
    print(f'devtest accuracy: {devtest_eval}\n')


    return epoch_losses, train_evals, dev_evals, best_dev_eval, devtest_eval
```

1.3.3 run experiments

[ ]:
```
# run 1 epoch of training 5 times with random seeds


# specify seeds for each run
seeds = [42, 645, 234, 534, 56]
```

```python
# for storing dev and devtest accuracy of each run
final_dev_evals_bert_cls_frozen = []
devtest_evals_bert_cls_frozen = []


for i, seed in enumerate(seeds):

    print(f'======== RUN {i+1} ======== ')

    # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # instantiate classifier for current run
    classifier = Classifier(
        in_size=bert_model.config.hidden_size,
        layer_sizes=[classifier_hidden_size, datasets['train'].n_classes],
        layer_acts=[nn.ReLU(), nn.Identity()]
    ).to(bert_model.device)

    # instantiate optimizer for current run
    optimizer = torch.optim.Adam(classifier.parameters(), lr=5e-4)

    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='bert_cls_frozen_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=dataloaders,
        pretrained_lm=bert_model,
        rep_extractor=extract_bert_rep,
        pooling='first',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_bert_cls_frozen.append(final_dev_eval)
    devtest_evals_bert_cls_frozen.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
```

```
----- TRAIN -----

100%|     | 313/313 [00:41<00:00,  7.50it/s]

  epoch loss: 411.33130034804344
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.51it/s]

  dev accuracy: 0.962
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.95it/s]

devtest accuracy: 0.95

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:41<00:00,  7.63it/s]

  epoch loss: 409.298469632864
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.66it/s]

  dev accuracy: 0.958
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.89it/s]

devtest accuracy: 0.955

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:41<00:00,  7.50it/s]

  epoch loss: 387.23794293403625
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.92it/s]

  dev accuracy: 0.968
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:04<00:00,  7.75it/s]
```

```
devtest accuracy: 0.975

======== RUN 4 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:43<00:00,  7.19it/s]

  epoch loss: 390.3639376461506
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.16it/s]

  dev accuracy: 0.972
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.34it/s]

devtest accuracy: 0.969

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:43<00:00,  7.21it/s]

  epoch loss: 502.01690328121185
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.16it/s]

  dev accuracy: 0.899
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.38it/s]

devtest accuracy: 0.889
```

[ ]:
```python
print(final_dev_evals_bert_cls_frozen)
print(devtest_evals_bert_cls_frozen)
```

```
[0.962, 0.958, 0.968, 0.972, 0.899]
[0.95, 0.955, 0.975, 0.969, 0.889]
```

[ ]:
```python
dev_accu_mean_bert_cls_frozen = np.mean(final_dev_evals_bert_cls_frozen)
dev_accu_std_bert_cls_frozen = np.std(final_dev_evals_bert_cls_frozen)
```

```
print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
↪{round(dev_accu_mean_bert_cls_frozen, 4)}, \
with a standard deviation of {round(dev_accu_std_bert_cls_frozen, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_bert_cls_frozen[np.
↪argmax(final_dev_evals_bert_cls_frozen)], 4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.9518, with a standard
deviation of 0.0268.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.969 on the test set.
```

1.4 2. mean-pooling and max-pooling across tokens

1.4.1 mean-pooling

[ ]:
```
# run 1 epoch of training 5 times with random seeds


# specify seeds for each run
seeds = [42, 645, 234, 534, 56]

# for storing dev and devtest accuracy of each run
final_dev_evals_bert_mean_frozen = []
devtest_evals_bert_mean_frozen = []


for i, seed in enumerate(seeds):

    print(f'======== RUN {i+1} ======== ')

    # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # instantiate classifier for current run
    classifier = Classifier(
        in_size=bert_model.config.hidden_size,
        layer_sizes=[classifier_hidden_size, datasets['train'].n_classes],
        layer_acts=[nn.ReLU(), nn.Identity()]
    ).to(bert_model.device)

    # instantiate optimizer for current run
```

```python
    optimizer = torch.optim.Adam(classifier.parameters(), lr=5e-4)

    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='bert_mean_frozen_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=dataloaders,
        pretrained_lm=bert_model,
        rep_extractor=extract_bert_rep,
        pooling='mean',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_bert_mean_frozen.append(final_dev_eval)
    devtest_evals_bert_mean_frozen.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:41<00:00,  7.61it/s]

  epoch loss: 379.73703303933144
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.52it/s]

  dev accuracy: 0.956
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.40it/s]

devtest accuracy: 0.959

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:42<00:00,  7.41it/s]

  epoch loss: 361.24872237443924
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.06it/s]
```

```
  dev accuracy: 0.966
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:04<00:00,  7.88it/s]

devtest accuracy: 0.963

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:44<00:00,  7.09it/s]

  epoch loss: 346.2587603032589
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.53it/s]

  dev accuracy: 0.977
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:04<00:00,  7.81it/s]

devtest accuracy: 0.973

======== RUN 4 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:44<00:00,  7.04it/s]

  epoch loss: 338.6590254753828
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.73it/s]

  dev accuracy: 0.973
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:04<00:00,  7.69it/s]

devtest accuracy: 0.97

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:45<00:00,  6.91it/s]
```

```
  epoch loss: 380.0158703774214
----- EVAL -----

100%|          | 32/32 [00:03<00:00,  8.01it/s]

  dev accuracy: 0.963
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|          | 32/32 [00:03<00:00,  8.25it/s]

devtest accuracy: 0.948
```

[ ]:
```
print(final_dev_evals_bert_mean_frozen)
print(devtest_evals_bert_mean_frozen)
```

```
[0.956, 0.966, 0.977, 0.973, 0.963]
[0.959, 0.963, 0.973, 0.97, 0.948]
```

[ ]:
```
dev_accu_mean_bert_mean_frozen = np.mean(final_dev_evals_bert_mean_frozen)
dev_accu_std_bert_mean_frozen = np.std(final_dev_evals_bert_mean_frozen)

print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
 ↪{round(dev_accu_mean_bert_mean_frozen, 4)}, \
with a standard deviation of {round(dev_accu_std_bert_mean_frozen, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_bert_mean_frozen[np.
 ↪argmax(final_dev_evals_bert_mean_frozen)], 4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.967, with a standard
deviation of 0.0074.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.973 on the test set.
```

1.4.2 max-pooling

[ ]:
```
# run 1 epoch of training 5 times with random seeds



# specify seeds for each run
seeds = [42, 645, 234, 534, 56]

# for storing dev and devtest accuracy of each run
final_dev_evals_bert_max_frozen = []
```

```python
devtest_evals_bert_max_frozen = []


for i, seed in enumerate(seeds):

    print(f'======== RUN {i+1} ======== ')

    # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # instantiate classifier for current run
    classifier = Classifier(
        in_size=bert_model.config.hidden_size,
        layer_sizes=[classifier_hidden_size, datasets['train'].n_classes],
        layer_acts=[nn.ReLU(), nn.Identity()]
    ).to(bert_model.device)

    # instantiate optimizer for current run
    optimizer = torch.optim.Adam(classifier.parameters(), lr=5e-4)

    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='bert_max_frozen_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=dataloaders,
        pretrained_lm=bert_model,
        rep_extractor=extract_bert_rep,
        pooling='max',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_bert_max_frozen.append(final_dev_eval)
    devtest_evals_bert_max_frozen.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:40<00:00,  7.75it/s]
```

```
  epoch loss: 611.1268037557602
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.66it/s]

  dev accuracy: 0.751
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.66it/s]

devtest accuracy: 0.746

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:42<00:00,  7.37it/s]

  epoch loss: 649.1984082460403
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.27it/s]

  dev accuracy: 0.634
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.37it/s]

devtest accuracy: 0.63

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:42<00:00,  7.30it/s]

  epoch loss: 688.3176131248474
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.14it/s]

  dev accuracy: 0.5
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.41it/s]

devtest accuracy: 0.497

======== RUN 4 ========
```

```
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:43<00:00,  7.19it/s]

  epoch loss: 680.7080105543137
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.11it/s]

  dev accuracy: 0.515
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.34it/s]

devtest accuracy: 0.518

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:44<00:00,  7.09it/s]

  epoch loss: 618.6238803863525
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.06it/s]

  dev accuracy: 0.637
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.25it/s]

devtest accuracy: 0.633
```

[ ]:

```python
print(final_dev_evals_bert_max_frozen)
print(devtest_evals_bert_max_frozen)
```

```
[0.751, 0.634, 0.5, 0.515, 0.637]
[0.746, 0.63, 0.497, 0.518, 0.633]
```

[ ]:

```python
dev_accu_mean_bert_max_frozen = np.mean(final_dev_evals_bert_max_frozen)
dev_accu_std_bert_max_frozen = np.std(final_dev_evals_bert_max_frozen)

print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
 ↪{round(dev_accu_mean_bert_max_frozen, 4)}, \
```

```
with a standard deviation of {round(dev_accu_std_bert_max_frozen, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_bert_max_frozen[np.
 ↪argmax(final_dev_evals_bert_max_frozen)], 4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.6074, with a standard
deviation of 0.092.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.746 on the test set.
```

1.5 3. comparing pooling techniques

- As shown in the above results, mean-pooling achieved the highest sentence classification accuracy with frozen BERT features.
- First-token pooling has quite similar performance to mean-pooling, but slightly worse (by only around 1~2%).
- Max-pooling had the lowest accuracy, and its classification accuracy was 20~35% lower than that of mean-pooling and first-token pooling.

1.6 4. BERT fine-tuning with [CLS] features

[ ]:
```
# check BERT last two layers (layers 10 and 11)
for name, param in bert_model.named_parameters():
    if name.startswith('encoder.layer.10') or name.startswith('encoder.layer.
 ↪11'): print(name)
```

```
encoder.layer.10.attention.self.query.weight
encoder.layer.10.attention.self.query.bias
encoder.layer.10.attention.self.key.weight
encoder.layer.10.attention.self.key.bias
encoder.layer.10.attention.self.value.weight
encoder.layer.10.attention.self.value.bias
encoder.layer.10.attention.output.dense.weight
encoder.layer.10.attention.output.dense.bias
encoder.layer.10.attention.output.LayerNorm.weight
encoder.layer.10.attention.output.LayerNorm.bias
encoder.layer.10.intermediate.dense.weight
encoder.layer.10.intermediate.dense.bias
encoder.layer.10.output.dense.weight
encoder.layer.10.output.dense.bias
encoder.layer.10.output.LayerNorm.weight
encoder.layer.10.output.LayerNorm.bias
encoder.layer.11.attention.self.query.weight
encoder.layer.11.attention.self.query.bias
encoder.layer.11.attention.self.key.weight
encoder.layer.11.attention.self.key.bias
encoder.layer.11.attention.self.value.weight
```

```
encoder.layer.11.attention.self.value.bias
encoder.layer.11.attention.output.dense.weight
encoder.layer.11.attention.output.dense.bias
encoder.layer.11.attention.output.LayerNorm.weight
encoder.layer.11.attention.output.LayerNorm.bias
encoder.layer.11.intermediate.dense.weight
encoder.layer.11.intermediate.dense.bias
encoder.layer.11.output.dense.weight
encoder.layer.11.output.dense.bias
encoder.layer.11.output.LayerNorm.weight
encoder.layer.11.output.LayerNorm.bias
```

[ ]:
```
# run 1 epoch of training 5 times with random seeds


# specify seeds for each run
seeds = [42, 645, 234, 534, 56]

# for storing dev and devtest accuracy of each run
final_dev_evals_bert_cls_tune = []
devtest_evals_bert_cls_tune = []


for i, seed in enumerate(seeds):

    print(f'======== RUN {i+1} ======== ')

    # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    ################# instantiate BERT model to be fine-tuned #################
    bert_model_tune = AutoModel.from_pretrained('bert-base-cased')
    if torch.cuda.is_available():  # use GPU if available
        bert_model_tune = bert_model_tune.cuda()
    ###########################################################################

    # instantiate classifier for current run
    classifier = Classifier(
        in_size=bert_model_tune.config.hidden_size,
        layer_sizes=[classifier_hidden_size, datasets['train'].n_classes],
        layer_acts=[nn.ReLU(), nn.Identity()]
    ).to(bert_model_tune.device)
```

```python
    ########## add BERT last two layers' parameters to params to be optimized␣
↪##########
    params = list()
    for name, param in bert_model_tune.named_parameters():
        if name.startswith('encoder.layer.10') or name.startswith('encoder.
↪layer.11'): params.append(param)

    # instantiate optimizer for current run
    optimizer = torch.optim.Adam(params + list(classifier.parameters()),␣
↪lr=5e-4)
   ␣
↪####################################################################################

    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='bert_cls_tune_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=dataloaders,
        pretrained_lm=bert_model_tune,
        rep_extractor=extract_bert_rep,
        pooling='first',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_bert_cls_tune.append(final_dev_eval)
    devtest_evals_bert_cls_tune.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:42<00:00,  7.36it/s]

  epoch loss: 392.02118411660194
----- EVAL -----

100%|      | 32/32 [00:03<00:00,  8.20it/s]

  dev accuracy: 0.967
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.41it/s]
```

```
devtest accuracy: 0.973

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:43<00:00,  7.21it/s]

  epoch loss: 389.5255144238472
----- EVAL -----

100%|     | 32/32 [00:03<00:00,  8.12it/s]

  dev accuracy: 0.966
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.28it/s]

devtest accuracy: 0.964

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:44<00:00,  7.08it/s]

  epoch loss: 428.09788951277733
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.96it/s]

  dev accuracy: 0.973
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.08it/s]

devtest accuracy: 0.972

======== RUN 4 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:44<00:00,  7.10it/s]

  epoch loss: 432.39132446050644
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.95it/s]

  dev accuracy: 0.939
  best classifier from epoch 1
```

```
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.22it/s]

devtest accuracy: 0.931

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:44<00:00,  7.07it/s]

  epoch loss: 396.9234875738621
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.93it/s]

  dev accuracy: 0.969
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.12it/s]

devtest accuracy: 0.972
```

[ ]:
```python
print(final_dev_evals_bert_cls_tune)
print(devtest_evals_bert_cls_tune)
```

```
[0.967, 0.966, 0.973, 0.939, 0.969]
[0.973, 0.964, 0.972, 0.931, 0.972]
```

[ ]:
```python
dev_accu_mean_bert_cls_tune = np.mean(final_dev_evals_bert_cls_tune)
dev_accu_std_bert_cls_tune = np.std(final_dev_evals_bert_cls_tune)

print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
 ↪{round(dev_accu_mean_bert_cls_tune, 4)}, \
with a standard deviation of {round(dev_accu_std_bert_cls_tune, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_bert_cls_tune[np.argmax(final_dev_evals_bert_cls_tune)],␣
 ↪4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.9628, with a standard
deviation of 0.0121.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.972 on the test set.
```

- Fine-tuning BERT with [CLS] features produced slightly better classification accuracy compared to using frozen BERT [CLS] features.
- However, classification accuracy after fine-tuning BERT with [CLS] features did not exceed mean-pooling with frozen BERT features.
- Classification accuracy after fine-tuning BERT with [CLS] features still exceeded max-pooling with frozen BERT features by far.

1.7 5. GPT-2

1.7.1 setup

[ ]:
```
# set hyperparameters
batch_size = 32
classifier_hidden_size = 32

# load GPT-2 tokenizer
gpt2_tokenizer = AutoTokenizer.from_pretrained('gpt2')
gpt2_tokenizer.pad_token = gpt2_tokenizer.eos_token

# load GPT-2 model
gpt2_model = AutoModel.from_pretrained('gpt2')  # https://huggingface.co/
 ↪transformers/v3.0.2/model_doc/auto.html#automodel
if torch.cuda.is_available():  # use GPU if available
    gpt2_model = gpt2_model.cuda()

# construct datasets with GPT2 tokenizer
gpt2_datasets, gpt2_dataloaders = construct_datasets(
    prefix='/content/drive/MyDrive/LLMs/data/dbpedia_',
    batch_size=batch_size,
    tokenizer=gpt2_tokenizer,
    device=gpt2_model.device
)

# # sanity check
# datasets['train'].__getitem__(0)

# # sanity check
# dtrain0 = next(iter(dataloaders['train']))
# # labels
# dtrain0[0]
# # sentences
# dtrain0[1].keys()   # ['input_ids', 'attention_mask']
# import pandas as pd
# pd.DataFrame(dtrain0[1]['input_ids'])
```

[ ]:
```
# func for extracting frozen GPT2 token representations for sentences and␣
 ↪pooling
```

```python
def extract_gpt2_rep(pretrained_lm, sentences:dict, pooling:str):


    # extract frozen GPT2 token representations for sentences
    with torch.no_grad():   # keep GPT2 params fixed
        unpooled_features = pretrained_lm(**sentences).last_hidden_state  # (B,␣
↪L, D): (batch_size, num token in sentence, GPT2 rep dimension)


    # get pooled_features across tokens for each sentence
    if pooling == 'first':
        pooled_features = unpooled_features[:, 0, :]   # (B, D)

    elif pooling == 'last':
        pooled_features = unpooled_features[range(unpooled_features.shape[0]),␣
↪sentences['attention_mask'].sum(dim=1) - 1, :]   # (B, D)

    elif pooling == 'mean':
        # mask padding tokens (where attention_mask is 0) with nan
        unpooled_features_masked = unpooled_features.masked_fill(
            sentences['attention_mask'].unsqueeze(-1)==0,
            float('nan')
        )
        # max-pooling across tokens (L dimension) in each sentence
        pooled_features = unpooled_features_masked.nanmean(dim=1)   # (B, D)

    elif pooling == 'max':
        # mask padding tokens (where attention_mask is 0) with -inf
        unpooled_features_masked = unpooled_features.masked_fill(
            sentences['attention_mask'].unsqueeze(-1)==0,
            float('-inf')
        )
        # max-pooling across tokens (L dimension) in each sentence
        pooled_features, _ = unpooled_features_masked.max(dim=1)   # (B, D)


    return pooled_features
```

1.7.2 frozen representations

Since GPT models are left-hand-side context only, we will use the last content token representation as classifier input.

[ ]:
```python
# run 1 epoch of training 5 times with random seeds


# specify seeds for each run
```

```python
seeds = [42, 645, 234, 534, 56]

# for storing dev and devtest accuracy of each run
final_dev_evals_gpt2_last_frozen = []
devtest_evals_gpt2_last_frozen = []


for i, seed in enumerate(seeds):

    print(f'======== RUN {i+1} ======== ')

    # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # instantiate classifier for current run
    classifier = Classifier(
        in_size=gpt2_model.config.hidden_size,
        layer_sizes=[classifier_hidden_size, gpt2_datasets['train'].n_classes],
        layer_acts=[nn.ReLU(), nn.Identity()]
    ).to(gpt2_model.device)

    # instantiate optimizer for current run
    optimizer = torch.optim.Adam(classifier.parameters(), lr=5e-4)

    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='gpt2_last_frozen_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=gpt2_dataloaders,
        pretrained_lm=gpt2_model,
        rep_extractor=extract_gpt2_rep,
        pooling='last',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_gpt2_last_frozen.append(final_dev_eval)
    devtest_evals_gpt2_last_frozen.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:45<00:00,  6.93it/s]

  epoch loss: 368.22356283664703
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.99it/s]

  dev accuracy: 0.934
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:03<00:00,  8.17it/s]

devtest accuracy: 0.937

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:45<00:00,  6.95it/s]

  epoch loss: 518.4893066287041
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.90it/s]

  dev accuracy: 0.915
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.85it/s]

devtest accuracy: 0.911

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:45<00:00,  6.94it/s]

  epoch loss: 387.91995015740395
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.93it/s]

  dev accuracy: 0.934
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…
```

```
100%|      | 32/32 [00:03<00:00,  8.06it/s]

devtest accuracy: 0.941

======== RUN 4 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:45<00:00,  6.85it/s]

  epoch loss: 361.42160111665726
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.74it/s]

  dev accuracy: 0.933
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.36it/s]

devtest accuracy: 0.948

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:46<00:00,  6.77it/s]

  epoch loss: 438.7018740475178
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.69it/s]

  dev accuracy: 0.908
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.93it/s]

devtest accuracy: 0.906
```

[ ]:

```python
print(final_dev_evals_gpt2_last_frozen)
print(devtest_evals_gpt2_last_frozen)
```

```
[0.934, 0.915, 0.934, 0.933, 0.908]
[0.937, 0.911, 0.941, 0.948, 0.906]
```

```
[ ]: dev_accu_mean_gpt2_last_frozen = np.mean(final_dev_evals_gpt2_last_frozen)
dev_accu_std_gpt2_last_frozen = np.std(final_dev_evals_gpt2_last_frozen)

print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
 ↪{round(dev_accu_mean_gpt2_last_frozen, 4)}, \
with a standard deviation of {round(dev_accu_std_gpt2_last_frozen, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_gpt2_last_frozen[np.
 ↪argmax(final_dev_evals_gpt2_last_frozen)], 4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.9248, with a standard
deviation of 0.0111.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.937 on the test set.
```

1.7.3 fine-tuning

```
[ ]: # check GPT2 last two layers
for name, param in gpt2_model.named_parameters():
    if name.startswith('h.10') or name.startswith('h.11'): print(name)
```

```
h.10.ln_1.weight
h.10.ln_1.bias
h.10.attn.c_attn.weight
h.10.attn.c_attn.bias
h.10.attn.c_proj.weight
h.10.attn.c_proj.bias
h.10.ln_2.weight
h.10.ln_2.bias
h.10.mlp.c_fc.weight
h.10.mlp.c_fc.bias
h.10.mlp.c_proj.weight
h.10.mlp.c_proj.bias
h.11.ln_1.weight
h.11.ln_1.bias
h.11.attn.c_attn.weight
h.11.attn.c_attn.bias
h.11.attn.c_proj.weight
h.11.attn.c_proj.bias
h.11.ln_2.weight
h.11.ln_2.bias
h.11.mlp.c_fc.weight
h.11.mlp.c_fc.bias
h.11.mlp.c_proj.weight
h.11.mlp.c_proj.bias
```

```
[ ]: # run 1 epoch of training 5 times with random seeds


     # specify seeds for each run
     seeds = [42, 645, 234, 534, 56]

     # for storing dev and devtest accuracy of each run
     final_dev_evals_gpt2_last_tune = []
     devtest_evals_gpt2_last_tune = []


     for i, seed in enumerate(seeds):

         print(f'======== RUN {i+1} ======== ')

         # set random seed: https://pytorch.org/docs/stable/notes/randomness.html
         torch.manual_seed(seed)
         torch.cuda.manual_seed_all(seed)
         torch.backends.cudnn.deterministic = True
         torch.backends.cudnn.benchmark = False

         ################# instantiate GPT2 model to be fine-tuned #################
         gpt2_model_tune = AutoModel.from_pretrained('gpt2')
         if torch.cuda.is_available():  # use GPU if available
             gpt2_model_tune = gpt2_model_tune.cuda()
         ###########################################################################

         # instantiate classifier for current run
         classifier = Classifier(
             in_size=gpt2_model_tune.config.hidden_size,
             layer_sizes=[classifier_hidden_size, datasets['train'].n_classes],
             layer_acts=[nn.ReLU(), nn.Identity()]
         ).to(gpt2_model_tune.device)

         ########## add GPT2 last two layers' parameters to params to be optimized␣
      ↪##########
         params = list()
         for name, param in gpt2_model_tune.named_parameters():
             if name.startswith('h.10') or name.startswith('h.11'): params.
      ↪append(param)

         # instantiate optimizer for current run
         optimizer = torch.optim.Adam(params + list(classifier.parameters()),␣
      ↪lr=5e-4)
        ␣
      ↪##################################################################################
```

```python
    # set loss function
    loss_func = nn.CrossEntropyLoss()

    # train and eval current classifier
    _, _, _, final_dev_eval, devtest_eval = main_process(
        classifier=classifier,
        name='gpt2_last_tune_seed'+str(seed),
        optimizer=optimizer,
        criterion=loss_func,
        dataloaders=gpt2_dataloaders,
        pretrained_lm=gpt2_model_tune,
        rep_extractor=extract_gpt2_rep,
        pooling='last',
        max_epochs=1,
        early_stopping=3
    )
    final_dev_evals_gpt2_last_tune.append(final_dev_eval)
    devtest_evals_gpt2_last_tune.append(devtest_eval)
```

```
======== RUN 1 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:46<00:00,  6.71it/s]

  epoch loss: 393.1364585161209
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.98it/s]

  dev accuracy: 0.93
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|     | 32/32 [00:03<00:00,  8.08it/s]

devtest accuracy: 0.923

======== RUN 2 ========
EPOCH 1
----- TRAIN -----

100%|     | 313/313 [00:45<00:00,  6.85it/s]

  epoch loss: 490.42394882440567
----- EVAL -----

100%|     | 32/32 [00:04<00:00,  7.73it/s]

  dev accuracy: 0.912
  best classifier from epoch 1
```

```
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.81it/s]

devtest accuracy: 0.906

======== RUN 3 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:46<00:00,  6.75it/s]

  epoch loss: 349.20694893598557
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.73it/s]

  dev accuracy: 0.922
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.86it/s]

devtest accuracy: 0.91

======== RUN 4 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:47<00:00,  6.66it/s]

  epoch loss: 407.39679250121117
----- EVAL -----

100%|      | 32/32 [00:04<00:00,  7.68it/s]

  dev accuracy: 0.934
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.66it/s]

devtest accuracy: 0.926

======== RUN 5 ========
EPOCH 1
----- TRAIN -----

100%|      | 313/313 [00:46<00:00,  6.77it/s]

  epoch loss: 407.05772137641907
----- EVAL -----
```

```
100%|      | 32/32 [00:04<00:00,  7.61it/s]

  dev accuracy: 0.929
  best classifier from epoch 1
----- EVAL -----
eval best classifier on devtest…

100%|      | 32/32 [00:04<00:00,  7.78it/s]

devtest accuracy: 0.922
```

[ ]:
```python
print(final_dev_evals_gpt2_last_tune)
print(devtest_evals_gpt2_last_tune)
```

```
[0.93, 0.912, 0.922, 0.934, 0.929]
[0.923, 0.906, 0.91, 0.926, 0.922]
```

[ ]:
```python
dev_accu_mean_gpt2_last_tune = np.mean(final_dev_evals_gpt2_last_tune)
dev_accu_std_gpt2_last_tune = np.std(final_dev_evals_gpt2_last_tune)

print(
f'Across the 5 runs, the mean accuracy on the dev set is␣
 ↪{round(dev_accu_mean_gpt2_last_tune, 4)}, \
with a standard deviation of {round(dev_accu_std_gpt2_last_tune, 4)}. \
\nThe best-performing classifier (w.r.t. dev set accuracy) has an accuracy of \
{round(devtest_evals_gpt2_last_tune[np.argmax(final_dev_evals_gpt2_last_tune)],␣
 ↪4)} on the test set.'
)
```

```
Across the 5 runs, the mean accuracy on the dev set is 0.9254, with a standard
deviation of 0.0077.
The best-performing classifier (w.r.t. dev set accuracy) has an accuracy of
0.926 on the test set.
```

[ ]:

[ ]:

[ ]: