Lab8_HuggingFace_Lab

April 1, 2023

0.1 # A Gentle Introduction to HuggingFace (HF)

HuggingFace provides you with a variety of pretrained models and functionalities to train/fine-tune these models and make inferences.

Their datasets library gives you access to many common NLP datasets. You can visualize these datasets on their platform to get a sense of the data you would be working with.

[62]:
```
!pip install datasets transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: datasets in /usr/local/lib/python3.9/dist-
packages (2.11.0)
Requirement already satisfied: transformers in /usr/local/lib/python3.9/dist-
packages (4.27.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.9/dist-packages
(from datasets) (1.4.4)
Requirement already satisfied: dill<0.3.7,>=0.3.0 in
/usr/local/lib/python3.9/dist-packages (from datasets) (0.3.6)
Requirement already satisfied: huggingface-hub<1.0.0,>=0.11.0 in
/usr/local/lib/python3.9/dist-packages (from datasets) (0.13.3)
Requirement already satisfied: xxhash in /usr/local/lib/python3.9/dist-packages
(from datasets) (3.2.0)
Requirement already satisfied: fsspec[http]>=2021.11.1 in
/usr/local/lib/python3.9/dist-packages (from datasets) (2023.3.0)
Requirement already satisfied: requests>=2.19.0 in
/usr/local/lib/python3.9/dist-packages (from datasets) (2.27.1)
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.9/dist-
packages (from datasets) (9.0.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-
packages (from datasets) (23.0)
Requirement already satisfied: responses<0.19 in /usr/local/lib/python3.9/dist-
packages (from datasets) (0.18.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.9/dist-packages
(from datasets) (3.8.4)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-
packages (from datasets) (6.0)
Requirement already satisfied: tqdm>=4.62.1 in /usr/local/lib/python3.9/dist-
packages (from datasets) (4.65.0)
```

```
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from datasets) (1.22.4)
Requirement already satisfied: multiprocess in /usr/local/lib/python3.9/dist-packages (from datasets) (0.70.14)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from transformers) (3.10.7)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.9/dist-packages (from transformers) (2022.10.31)
Requirement already satisfied: tokenizers!=0.11.3,<0.14,>=0.11.1 in
/usr/local/lib/python3.9/dist-packages (from transformers) (0.13.2)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in
/usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (2.0.12)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (1.8.2)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (6.0.4)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in
/usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (4.0.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (22.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.9/dist-packages (from aiohttp->datasets) (1.3.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.9/dist-packages (from huggingface-
hub<1.0.0,>=0.11.0->datasets) (4.5.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests>=2.19.0->datasets) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.9/dist-packages (from requests>=2.19.0->datasets)
(2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.9/dist-packages (from requests>=2.19.0->datasets)
(1.26.15)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.9/dist-packages (from pandas->datasets) (2022.7.1)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.9/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.8.1->pandas->datasets) (1.16.0)
```

0.2 Our Goal

Our goal for this tutorial is to get familiar with the transformers library from HuggingFace and use a pretrained model to fine-tune it on a sequece classification task. More specifically we will fine-tune a BERT model on the Amazon Polarity dataset. > The Amazon reviews dataset consists of reviews from amazon. The data span a period of 18 years, including ~35 million reviews up to

March 2013. Reviews include product and user information, ratings, and a plaintext review.

> The Amazon reviews polarity dataset is constructed by taking review score 1 and 2 as negative, and 4 and 5 as positive. Samples of score 3 is ignored. Each class has 1,800,000 training samples and 200,000 testing samples.

Since the dataset is quite large, we will be working with only a subset of this dataset throughout this tutorial.

0.3 Main Components

The main components we would need to develop to realize our goal are:

1. Load the data and make a dataset object for this task.
2. Write a collate function/class to tokenize/transform/truncate batches of inputs.
3. Make a custom model, which uses a pretrained model as its backbone and it is designed for our current task at hand.
4. Write the training loop and train the model.

These steps constitues the basic building blocks to solve any other problem using HF.

0.4 Loading data

In this stage we will load the data from the `datasets` library. We will only load a small subset of the original dataset here in order to reduce the training time, but feel free to run this code on the full dataset on your own time and experiment with it.

[63]:
```
from datasets import load_dataset

dataset_train = load_dataset("amazon_polarity", split="train[:1000]")
dataset_test = load_dataset("amazon_polarity", split="test[:200]")
```

```
WARNING:datasets.builder:Found cached dataset amazon_polarity (/root/.cache/hugg
ingface/datasets/amazon_polarity/amazon_polarity/3.0.0/a27b32b7e7b88eb274a8fa8ba
0f654f1fe998a87c22547557317793b5d2772dc)
WARNING:datasets.builder:Found cached dataset amazon_polarity (/root/.cache/hugg
ingface/datasets/amazon_polarity/amazon_polarity/3.0.0/a27b32b7e7b88eb274a8fa8ba
0f654f1fe998a87c22547557317793b5d2772dc)
```

[64]:
```
#@title  Quick look at the data { run: "auto" }
#@markdown Lets have quick look at a few samples as well as the label␣
 ↪distributions in our train and test set.
n_samples_to_see = 3 #@param {type: "integer"}
for i in range(n_samples_to_see):
  print("-"*30)
  print("title:", dataset_test[i]["title"])
  print("content:", dataset_test[i]["content"])
  print("label:", dataset_test[i]["label"])
```

```
------------------------------
title: Great CD
```

```
content: My lovely Pat has one of the GREAT voices of her generation. I have
listened to this CD for YEARS and I still LOVE IT. When I'm in a good mood it
makes me feel better. A bad mood just evaporates like sugar in the rain. This CD
just oozes LIFE. Vocals are jusat STUUNNING and lyrics just kill. One of life's
hidden gems. This is a desert isle CD in my book. Why she never made it big is
just beyond me. Everytime I play this, no matter black, white, young, old, male,
female EVERYBODY says one thing "Who was that singing ?"
label: 1
------------------------------
title: One of the best game music soundtracks - for a game I didn't really play
content: Despite the fact that I have only played a small portion of the game,
the music I heard (plus the connection to Chrono Trigger which was great as
well) led me to purchase the soundtrack, and it remains one of my favorite
albums. There is an incredible mix of fun, epic, and emotional songs. Those sad
and beautiful tracks I especially like, as there's not too many of those kinds
of songs in my other video game soundtracks. I must admit that one of the songs
(Life-A Distant Promise) has brought tears to my eyes on many occasions.My one
complaint about this soundtrack is that they use guitar fretting effects in many
of the songs, which I find distracting. But even if those weren't included I
would still consider the collection worth it.
label: 1
------------------------------
title: Batteries died within a year …
content: I bought this charger in Jul 2003 and it worked OK for a while. The
design is nice and convenient. However, after about a year, the batteries would
not hold a charge. Might as well just get alkaline disposables, or look
elsewhere for a charger that comes with batteries that have better staying
power.
label: 0
```

[65]:
```python
def label_stats(ds):
    negative = 0
    positive = 0
    for i in range(ds.num_rows):
        if ds[i]["label"] == 1:
            positive += 1
        else:
            negative += 1
    return positive, negative
```

[66]:
```python
for i, ds in enumerate([dataset_train, dataset_test]):
    positive, negative = label_stats(ds)
    if i == 0:
        str_indicator = "train"
    else:
        str_indicator = "test"
    print("+-" * 15)
```

```
    print("Set:", str_indicator)
    print(f"Positive samples: {positive}\nNegative samples: {negative}")
    print(f"Percentage of overall positive samples: {(positive*100.0)/
 ↪(positive+negative)}%")
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
Set: train
Positive samples: 462
Negative samples: 538
Percentage of overall positive samples: 46.2%
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
Set: test
Positive samples: 109
Negative samples: 91
Percentage of overall positive samples: 54.5%
```

0.5 Collate

Collate is a function that is called on every batch of data prepared by the dataloader. Once we pass our dataset (e.g. `train_set`) to our dataloader, each batch will be a `list` of `dict` items. Therefore, this cannot be directed to the model. We need to perform the followings at this stage:

0.5.1 1 Tokenize the text

Tokenize the `text`portion of each sample (i.e. parsing the text to smaller chuncks). Tokenization can happen in many ways, traditionally this was done based the white spaces. With transformer-based models tokenization is performed based on the frequency of occurance of "chunk of text". This frequence can be learnt in many different ways, however the most common one is the **wordpiece** model. > The wordpiece model is generated using a data-driven approach to maximize the language-model likelihood of the training data, given an evolving word definition. Given a training corpus and a number of desired tokens D, the optimization problem is to select D wordpieces such that the resulting corpus is minimal in the number of wordpieces when segmented according to the chosen wordpiece model.

Under this model: 1. Not all things can be converted to tokens depending on the model. For example, most models have been pretrained without any knowledge of emojis. So their token will be `[UNK]`, which stands for unknown. 2. Some words will be mapped to multiple tokens! 3. Depending on the kind of model, your tokens may or may not respect capitalization!

[67]:
```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

[68]:
```
#@title  Quick look at tokenization { run: "auto", vertical-output: true }
input_sample = "We are very jubilant to demonstrate to you the  Transformers␣
 ↪library." #@param {type: "string"}
tokenizer.tokenize(input_sample)
```

[68]:
```
['we',
 'are',
 'very',
 'ju',
 '##bil',
 '##ant',
 'to',
 'demonstrate',
 'to',
 'you',
 'the',
 '[UNK]',
 'transformers',
 'library',
 '.']
```

0.5.2 2 Encoding

Once we have tokenized the text, we then need to convert these chuncks to numbers so we can feed them to our model. This conversion is basically a look-up in a dictionary **from** `str` → `int`. The tokenizer object can also perform this work. While it does so it will also add the *special* tokens needed by the model to the encodings.

[69]:
```
#@title  Quick look at token encoding { run: "auto"}
input_sample = "We are very jubilant to demonstrate to you the  Transformers␣
↪library." #@param {type: "string"}
print("--> Token Encodings:\n",tokenizer.encode(input_sample))
print("-."*15)
print("--> Token Encodings Decoded:\n",tokenizer.decode(tokenizer.
↪encode(input_sample)))
```

```
--> Token Encodings:
 [101, 2057, 2024, 2200, 18414, 14454, 4630, 2000, 10580, 2000, 2017, 1996, 100,
19081, 3075, 1012, 102]
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
--> Token Encodings Decoded:
 [CLS] we are very jubilant to demonstrate to you the [UNK] transformers
library. [SEP]
```

0.5.3 3 Truncate/Pad samples

Since all the sample in the batch will not have the same sequence length, we would need to truncate the longers (i.e. the ones that exeed a predefined maximum length) and pad the shorter ones so we that we can equal length for all the samples in the batch. Once this is achieved, we would need to convert the result to `torch.Tensors` and return. These tensors will then be retrieved from the dataloader.

[70]:
```python
from typing import List, Dict, Union
import torch


class Collate:
    def __init__(self, tokenizer: str, max_len: int) -> None:
        self.tokenizer_name = tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.tokenizer_name)
        self.max_len = max_len

    def __call__(self, batch: List[Dict[str, Union[str, int]]]) -> Dict[str,␣
 ↪torch.Tensor]:
        texts = list(map(lambda batch_instance: batch_instance["title"], batch))
        tokenized_inputs = self.tokenizer(
            texts,
            padding="longest",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt",
            return_token_type_ids=False,
        )
        labels = list(map(lambda batch_instance: int(batch_instance["label"]),␣
 ↪batch))
        labels = torch.LongTensor(labels)
        return dict(tokenized_inputs, **{"labels": labels})
```

[71]:
```python
#@title   Setting up the collate function { run: "auto" }
tokenizer_name = "distilbert-base-uncased" #@param {type: "string"}
sample_max_length = 64 #@param {type:"slider", min:32, max:512, step:1}
collate = Collate(tokenizer="distilbert-base-uncased",␣
 ↪max_len=sample_max_length)
```

0.6 Model

Our model needs to classify an entire sequence of text. Once we feed an input sequence of length k to a language model, it will output k vectors. Now the question is which of these vectors or combition of these vectors should we use to classify the sequence? We will use the first toke, special token `[cls]` for these purposes. *Refer to the BERT paper for more information.*

Since we have 2 classes (positive, and negative), this means we would need to make a classifier on top of the vector representations of the `[cls]` token. Our custom model will then look like:

[72]:
```python
import torch
from transformers import AutoModel
from typing import Optional, Tuple


class ReviewClassifier(torch.nn.Module):
```

```python
    def __init__(self, backbone: str, backbone_hidden_size: int, nb_classes:
↪int):
        super(ReviewClassifier, self).__init__()
        self.backbone = backbone
        self.backbone_hidden_size = backbone_hidden_size
        self.nb_classes = nb_classes

        self.back_bone = AutoModel.from_pretrained(
            self.backbone,
            output_attentions=False,
            output_hidden_states=False,
        )
        self.classifier = torch.nn.Linear(self.backbone_hidden_size, self.
↪nb_classes)

    def forward(
        self, input_ids: torch.Tensor, attention_mask: torch.Tensor, labels:
↪Optional[torch.Tensor] = None
    ) -> Union[torch.Tensor, Tuple[torch.Tensor, torch.Tensor]]:
        back_bone_output = self.back_bone(input_ids,
↪attention_mask=attention_mask)
        hidden_states = back_bone_output[0]
        pooled_output = hidden_states[:, 0]  # getting the [CLS] token

        logits = self.classifier(pooled_output)
        if labels is not None:
            loss_fn = torch.nn.CrossEntropyLoss()
            loss = loss_fn(
                logits.view(-1, self.nb_classes),
                labels.view(-1),
            )
            return loss, logits
        return logits
```

[73]:

```python
model = ReviewClassifier(backbone="distilbert-base-uncased",
↪backbone_hidden_size=768, nb_classes=2)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_transform.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
```

BertForSequenceClassification model from a BertForSequenceClassification model).

0.7 Training Loop

In this section we will define the training loop to trian our model. Note that these model are sensative wrt the hyperparameters and it usually takes a while to find the right hyperparameters. The default hyperparameters should work fine for our test case.

[74]:
```python
from tqdm.auto import tqdm
from torch.utils.data import DataLoader
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
import numpy as np

print(f"--> Device selected: {device}")
```

```
--> Device selected: cuda
```

[75]:
```python
def train_one_epoch(
    model: torch.nn.Module, training_data_loader: DataLoader, optimizer: torch.
 ↪optim.Optimizer, logging_frequency: int
):
    model.train()
    optimizer.zero_grad()
    epoch_loss = 0
    logging_loss = 0
    for step, batch in enumerate(training_data_loader):
        batch = {key: value.to(device) for key, value in batch.items()}
        outputs = model(**batch)
        loss = outputs[0]
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        logging_loss += loss.item()

        if (step + 1) % logging_frequency == 0:
            print(f"Training loss @ step {step+1}: {logging_loss/
 ↪logging_frequency}")
            logging_loss = 0

    return epoch_loss / len(training_data_loader)


def evaluate(model: torch.nn.Module, test_data_loader: DataLoader, nb_classes:␣
 ↪int):
    model.eval()
    model.to(device)
    eval_loss = 0
```

```python
    correct_predictions = {i: 0 for i in range(nb_classes)}
    total_predictions = {i: 0 for i in range(nb_classes)}

    with torch.no_grad():
        for step, batch in enumerate(test_data_loader):
            batch = {key: value.to(device) for key, value in batch.items()}
            outputs = model(**batch)
            loss = outputs[0]
            eval_loss += loss.item()

            predictions = np.argmax(outputs[1].detach().cpu().numpy(), axis=1)
            for target, prediction in zip(batch["labels"].cpu().numpy(),␣
 ↪predictions):
                if target == prediction:
                    correct_predictions[target] += 1
                total_predictions[target] += 1

    accuracy = (100.0 * sum(correct_predictions.values())) /␣
 ↪sum(total_predictions.values())
    return accuracy, eval_loss / len(test_data_loader)
```

[76]:
```python
#@title   Setting hyperparameters for training { run: "auto" }
nb_epoch = 3 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=collate)
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False,␣
 ↪collate_fn=collate)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
    {
        "params": [p for n, p in model.named_parameters() if not any(nd in n␣
 ↪for nd in no_decay)],
        "weight_decay": 0.0,
    },
    {
        "params": [p for n, p in model.named_parameters() if any(nd in n for nd␣
 ↪in no_decay)],
        "weight_decay": 0.0,
    },
]
```

```
optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate,␣
↪eps=1e-8)
```

[77]:
```
model.to(device)

train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch(model, train_loader, optimizer,␣
↪logging_frequency)
    eval_acc, eval_loss  = evaluate(model, test_loader, 2)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss},␣
↪Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss,␣
↪"Acc/Test": eval_acc})
```

```
Epoch:   0%|          | 0/3 [00:00<?, ?it/s]

Training loss @ step 5: 0.7063361644744873
Training loss @ step 10: 0.70822833776474
Training loss @ step 15: 0.6952305674552918
    Epoch: 1 Loss/Test: 0.6560837924480438, Loss/Test: 0.6994565576314926,
Acc/Test: 67.5
Training loss @ step 5: 0.6405472040176392
Training loss @ step 10: 0.6031859159469605
Training loss @ step 15: 0.5732704758644104
    Epoch: 2 Loss/Test: 0.5556094497442245, Loss/Test: 0.6021909974515438,
Acc/Test: 77.5
Training loss @ step 5: 0.4826825261116028
Training loss @ step 10: 0.4314260184764862
Training loss @ step 15: 0.4189888656139374
    Epoch: 3 Loss/Test: 0.5207142978906631, Loss/Test: 0.4422369506210089,
Acc/Test: 71.0
```

1 Exercises

It is suggested that you have look over the `tokenizer` class and its functionalities before attempting the exercises.

1.1 1 Predict with more context

In the above training we only took advantage of the `title` of each review to predict its polarity. 1. Investigate whether it would be useful to instead use the `content` of each review? 2. Further investigate if it would be usefult to have both the `title` and `content` presented to model during training?

[78]:
```
model = ReviewClassifier(backbone="distilbert-base-uncased",␣
↪backbone_hidden_size=768, nb_classes=2)
class Collate_content:
```

```python
    def __init__(self, tokenizer: str, max_len: int) -> None:
        self.tokenizer_name = tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.tokenizer_name)
        self.max_len = max_len

    def __call__(self, batch: List[Dict[str, Union[str, int]]]) -> Dict[str,␣
 ↪torch.Tensor]:
        texts = list(map(lambda batch_instance: batch_instance["content"],␣
 ↪batch))
        tokenized_inputs = self.tokenizer(
            texts,
            padding="longest",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt",
            return_token_type_ids=False,
        )
        labels = list(map(lambda batch_instance: int(batch_instance["label"]),␣
 ↪batch))
        labels = torch.LongTensor(labels)
        return dict(tokenized_inputs, **{"labels": labels})

current_collate = Collate_content(tokenizer="distilbert-base-uncased",␣
 ↪max_len=sample_max_length)

#@title   Use only Content { run: "auto" }
nb_epoch = 3 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=current_collate)
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False,␣
 ↪collate_fn=current_collate)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
      {
          "params": [p for n, p in model.named_parameters() if not any(nd in n␣
 ↪for nd in no_decay)],
          "weight_decay": 0.0,
      },
      {
```

```python
            "params": [p for n, p in model.named_parameters() if any(nd in n for␣
 ↪nd in no_decay)],
            "weight_decay": 0.0,
        },
    ]

optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate,␣
 ↪eps=1e-8)


model.to(device)

train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch(model, train_loader, optimizer,␣
 ↪logging_frequency)
    eval_acc, eval_loss  = evaluate(model, test_loader, 2)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss},␣
 ↪Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss,␣
 ↪"Acc/Test": eval_acc})
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_transform.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Epoch:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Training loss @ step 5: 0.7042463064193726
Training loss @ step 10: 0.6884832143783569
Training loss @ step 15: 0.6704005122184753
    Epoch: 1 Loss/Test: 0.6573537141084671, Loss/Test: 0.6850832067430019,
Acc/Test: 65.5
Training loss @ step 5: 0.6416653990745544
Training loss @ step 10: 0.5924084663391114
Training loss @ step 15: 0.5557694673538208
    Epoch: 2 Loss/Test: 0.5943339020013809, Loss/Test: 0.5910059176385403,
Acc/Test: 70.5
Training loss @ step 5: 0.6033878207206727
Training loss @ step 10: 0.4367173135280609
```

```
Training loss @ step 15: 0.8465508401393891
    Epoch: 3 Loss/Test: 1.4179713726043701, Loss/Test: 0.6803175583481789,
Acc/Test: 46.5
```

So it is not that useful to use only the content of each review.

[79]:
```python
model = ReviewClassifier(backbone="distilbert-base-uncased",␣
↪backbone_hidden_size=768, nb_classes=2)
class Collate_content_and_title:
    def __init__(self, tokenizer: str, max_len: int) -> None:
        self.tokenizer_name = tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.tokenizer_name)
        self.max_len = max_len

    def __call__(self, batch: List[Dict[str, Union[str, int]]]) -> Dict[str,␣
↪torch.Tensor]:
        titles = list(map(lambda batch_instance: batch_instance["title"],␣
↪batch))
        contents = list(map(lambda batch_instance: batch_instance["content"],␣
↪batch))
        texts = [title + " " + content for title, content in zip(titles,␣
↪contents)]
        tokenized_inputs = self.tokenizer(
            texts,
            padding="longest",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt",
            return_token_type_ids=False,
        )
        labels = list(map(lambda batch_instance: int(batch_instance["label"]),␣
↪batch))
        labels = torch.LongTensor(labels)
        return dict(tokenized_inputs, **{"labels": labels})

current_collate_candt =␣
↪Collate_content_and_title(tokenizer="distilbert-base-uncased",␣
↪max_len=sample_max_length)

#@title   Use Title+ Content { run: "auto" }
nb_epoch = 3 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,␣
↪collate_fn=current_collate_candt)
```

```
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False,␣
↪collate_fn=current_collate_candt)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
      {
          "params": [p for n, p in model.named_parameters() if not any(nd in n␣
↪for nd in no_decay)],
          "weight_decay": 0.0,
      },
      {
          "params": [p for n, p in model.named_parameters() if any(nd in n for␣
↪nd in no_decay)],
          "weight_decay": 0.0,
      },
  ]

optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate,␣
↪eps=1e-8)


model.to(device)

train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch(model, train_loader, optimizer,␣
↪logging_frequency)
    eval_acc, eval_loss  = evaluate(model, test_loader, 2)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss},␣
↪Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss,␣
↪"Acc/Test": eval_acc})
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_transform.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Epoch:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Training loss @ step 5: 0.7020013928413391
Training loss @ step 10: 0.6723769307136536
Training loss @ step 15: 0.6493452787399292
    Epoch: 1 Loss/Test: 0.6319452524185181, Loss/Test: 0.6724485829472542,
Acc/Test: 63.5
Training loss @ step 5: 0.6117583513259888
Training loss @ step 10: 0.545013678073883
Training loss @ step 15: 0.5055689811706543
    Epoch: 2 Loss/Test: 0.6225843876600266, Loss/Test: 0.5558517202734947,
Acc/Test: 62.5
Training loss @ step 5: 0.45966126322746276
Training loss @ step 10: 0.4659955441951752
Training loss @ step 15: 0.8756790161132812
    Epoch: 3 Loss/Test: 0.38495729118585587, Loss/Test: 0.5923217069357634,
Acc/Test: 84.0
```

But if we are using both content and title, the result is indeed better.

1.2 2 Frozen representations

Modify the backbone so that we would only train the classifier layer, and the backbone stays frozen. How does the results compare to the unfrozen version?

[95]:
```
class FrozenReviewClassifier(torch.nn.Module):
    def __init__(self, backbone: str, backbone_hidden_size: int, nb_classes:␣
 ↪int):
        super(FrozenReviewClassifier, self).__init__()
        self.backbone = backbone
        self.backbone_hidden_size = backbone_hidden_size
        self.nb_classes = nb_classes

        self.back_bone = AutoModel.from_pretrained(
            self.backbone,
            output_attentions=False,
            output_hidden_states=False,
        )

        self.classifier = torch.nn.Linear(self.backbone_hidden_size, self.
 ↪nb_classes)

    def forward(
        self, input_ids: torch.Tensor, attention_mask: torch.Tensor, labels:␣
 ↪Optional[torch.Tensor] = None
    ) -> Union[torch.Tensor, Tuple[torch.Tensor, torch.Tensor]]:
        back_bone_output = self.back_bone(input_ids,␣
 ↪attention_mask=attention_mask)
        hidden_states = back_bone_output[0]
        pooled_output = hidden_states[:, 0]  # getting the [CLS] token
```

```
        logits = self.classifier(pooled_output)
        if labels is not None:
            loss_fn = torch.nn.CrossEntropyLoss()
            loss = loss_fn(
                logits.view(-1, self.nb_classes),
                labels.view(-1),
            )
            return loss, logits
        return logits


    def freeze_backbone(self):
        for param in self.back_bone.parameters():
            param.requires_grad = False

    def unfreeze_backbone(self):
        for param in self.back_bone.parameters():
            param.requires_grad = True


model = FrozenReviewClassifier(backbone="distilbert-base-uncased",␣
 ↪backbone_hidden_size=768, nb_classes=2)

model.freeze_backbone()

frozen_collate = Collate(tokenizer="distilbert-base-uncased",␣
 ↪max_len=sample_max_length)

#@title   Use Frozen backbone { run: "auto" }
nb_epoch = 3 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=frozen_collate)
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False,␣
 ↪collate_fn=frozen_collate)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
      {
          "params": [p for n, p in model.named_parameters() if not any(nd in n␣
 ↪for nd in no_decay)],
          "weight_decay": 0.0,
```

```
    },
    {
        "params": [p for n, p in model.named_parameters() if any(nd in n for␣
↪nd in no_decay)],
        "weight_decay": 0.0,
    },
]

optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate,␣
↪eps=1e-8)


model.to(device)

train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch(model, train_loader, optimizer,␣
↪logging_frequency)
    eval_acc, eval_loss  = evaluate(model, test_loader, 2)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss},␣
↪Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss,␣
↪"Acc/Test": eval_acc})
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_transform.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Epoch:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Training loss @ step 5: 0.7004940629005432
Training loss @ step 10: 0.7084245204925537
Training loss @ step 15: 0.7015885233879089
    Epoch: 1 Loss/Test: 0.6878619194030762, Loss/Test: 0.7028164491057396,
Acc/Test: 54.0
Training loss @ step 5: 0.6953300833702087
Training loss @ step 10: 0.6973816514015198
Training loss @ step 15: 0.6932207345962524
    Epoch: 2 Loss/Test: 0.6867895424365997, Loss/Test: 0.6953100487589836,
Acc/Test: 57.0
```

```
Training loss @ step 5: 0.6956471920013427
Training loss @ step 10: 0.6902302026748657
Training loss @ step 15: 0.6860473394393921
    Epoch: 3 Loss/Test: 0.6858240962028503, Loss/Test: 0.6900477260351181,
Acc/Test: 59.5
```

If we freeze the bacbone, the result is worse. This might because of the pre-trained backbone model is not well-suited for the task. We can also see the accuracy doesn't improve too much over epochs.

1.3 3 (Optional) Freeze then unfreeze

It has empirically been shown that freezing the backbone for the first few steps of training and then unfreezing it produces better performing models. Modify the training code to have this option for training.

[100]:
```
model = FrozenReviewClassifier(backbone="distilbert-base-uncased",␣
↪backbone_hidden_size=768, nb_classes=2)

model.freeze_backbone()
frozen_backbone_steps = 5

half_frozen_collate = Collate(tokenizer="distilbert-base-uncased",␣
↪max_len=sample_max_length)

#@title   Use half-Frozen backbone { run: "auto" }
nb_epoch = 3 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,␣
↪collate_fn=half_frozen_collate)
test_loader = DataLoader(dataset_test, batch_size=batch_size, shuffle=False,␣
↪collate_fn=half_frozen_collate)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
      {
          "params": [p for n, p in model.named_parameters() if not any(nd in n␣
↪for nd in no_decay)],
          "weight_decay": 0.0,
      },
      {
          "params": [p for n, p in model.named_parameters() if any(nd in n for␣
↪nd in no_decay)],
          "weight_decay": 0.0,
```

```python
      },
  ]

optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate, eps=1e-8)

def train_one_epoch_half_frozen(
    model: torch.nn.Module, training_data_loader: DataLoader, optimizer: torch.optim.Optimizer, logging_frequency: int, unfrozen_step:int
):
    model.train()
    optimizer.zero_grad()
    epoch_loss = 0
    logging_loss = 0
    for step, batch in enumerate(training_data_loader):
        batch = {key: value.to(device) for key, value in batch.items()}
        outputs = model(**batch)
        loss = outputs[0]
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        logging_loss += loss.item()

        if (step == unfrozen_step):
            model.unfreeze_backbone()

        if (step + 1) % logging_frequency == 0:
            print(f"Training loss @ step {step+1}: {logging_loss/logging_frequency}")
            logging_loss = 0

    return epoch_loss / len(training_data_loader)

model.to(device)
step_count = 0
train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch_half_frozen(model, train_loader, optimizer, logging_frequency,frozen_backbone_steps)
    eval_acc, eval_loss  = evaluate(model, test_loader, 2)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss}, Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss, "Acc/Test": eval_acc})
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_transform.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Epoch:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
Training loss @ step 5: 0.7062148451805115
Training loss @ step 10: 0.705574095249176
Training loss @ step 15: 0.6757287502288818
    Epoch: 1 Loss/Test: 0.6580210328102112, Loss/Test: 0.6936963759362698,
Acc/Test: 69.5
Training loss @ step 5: 0.658850371837616
Training loss @ step 10: 0.6193810224533081
Training loss @ step 15: 0.5822450637817382
    Epoch: 2 Loss/Test: 0.5309212058782578, Loss/Test: 0.6145724877715111,
Acc/Test: 78.5
Training loss @ step 5: 0.4846816956996918
Training loss @ step 10: 0.43681393265724183
Training loss @ step 15: 0.41208499670028687
    Epoch: 3 Loss/Test: 0.3740512579679489, Loss/Test: 0.439826812595129,
Acc/Test: 83.5
```

If we freeze and unfreeze the backbone, the result is indeed better compare to the frozen one above.

1.4 4 (Optional) Build an emotion aware AI

Lets now put everything we learned to the test by building an agent with some emotion detection abilities. Use the emotion dataset to train an ALBERT-based model to detect the six basic emotions in our datasets. (anger, fear, joy, love, sadness, and surprise)

[132]:
```python
from transformers import AlbertModel

class EmotionClassifier(torch.nn.Module):
    def __init__(self, backbone: str, backbone_hidden_size: int, nb_classes: int):
        super(EmotionClassifier, self).__init__()
        self.backbone = backbone
        self.backbone_hidden_size = backbone_hidden_size
        self.nb_classes = nb_classes
```

```
        self.back_bone = AutoModel.from_pretrained(
            self.backbone,
            output_attentions=False,
            output_hidden_states=False,
        )

        self.classifier = torch.nn.Linear(self.backbone_hidden_size, self.
 ↪nb_classes)

    def forward(
        self, input_ids: torch.Tensor, attention_mask: torch.Tensor, labels:␣
 ↪Optional[torch.Tensor] = None
    ) -> Union[torch.Tensor, Tuple[torch.Tensor, torch.Tensor]]:
        back_bone_output = self.back_bone(input_ids,␣
 ↪attention_mask=attention_mask)
        hidden_states = back_bone_output[0]
        pooled_output = hidden_states[:, 0]  # getting the [CLS] token

        logits = self.classifier(pooled_output)
        if labels is not None:
            loss_fn = torch.nn.CrossEntropyLoss()
            loss = loss_fn(
                logits.view(-1, self.nb_classes),
                labels.view(-1),
            )
            return loss, logits
        return logits


    def freeze_backbone(self):
        for param in self.back_bone.parameters():
            param.requires_grad = False

    def unfreeze_backbone(self):
        for param in self.back_bone.parameters():
            param.requires_grad = True

model = EmotionClassifier(backbone="albert-base-v2", backbone_hidden_size=768,␣
 ↪nb_classes=6)

class Collate_emotion:
    def __init__(self, tokenizer: str, max_len: int) -> None:
        self.tokenizer_name = tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.tokenizer_name)
        self.max_len = max_len
```

```python
    def __call__(self, batch: List[Dict[str, Union[str, int]]]) -> Dict[str, torch.Tensor]:
        texts = list(map(lambda batch_instance: batch_instance["text"], batch))
        tokenized_inputs = self.tokenizer(
            texts,
            padding="longest",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt",
            return_token_type_ids=False,
        )
        labels = list(map(lambda batch_instance: int(batch_instance["label"]), batch))
        labels = torch.LongTensor(labels)
        return dict(tokenized_inputs, **{"labels": labels})

emotion_train = load_dataset("dair-ai/emotion", split="train[:1000]")
emotion_test = load_dataset("dair-ai/emotion", split="test[:200]")

emotion_collate = Collate_emotion(tokenizer="albert-base-v2", max_len=128)

#@title   Emotion { run: "auto" }
nb_epoch = 8 #@param {type: "slider", min:1, max:10, step:1}
batch_size = 64 #@param {type: "integer"}
logging_frequency = 5 #@param {type: "integer"}
learning_rate = 1e-5 #@param {type: "number"}

train_loader = DataLoader(emotion_train, batch_size=batch_size, shuffle=True, collate_fn=emotion_collate)
test_loader = DataLoader(emotion_test, batch_size=batch_size, shuffle=False, collate_fn=emotion_collate)

# setting up the optimizer
no_decay = ["bias", "LayerNorm.weight"]
optimizer_grouped_parameters = [
    {
        "params": [p for n, p in model.named_parameters() if not any(nd in n for nd in no_decay)],
        "weight_decay": 0.0,
    },
    {
        "params": [p for n, p in model.named_parameters() if any(nd in n for nd in no_decay)],
        "weight_decay": 0.0,
    },
  ]
```

```
optimizer = torch.optim.AdamW(optimizer_grouped_parameters, lr=learning_rate,␣
 ↪eps=1e-8)

frozen_backbone_steps = 5

model.to(device)
train_bar = tqdm(range(nb_epoch), desc="Epoch")
for e in train_bar:
    train_loss = train_one_epoch_half_frozen(model, train_loader, optimizer,␣
 ↪logging_frequency,frozen_backbone_steps)
    eval_acc, eval_loss  = evaluate(model, test_loader, 6)
    print(f"    Epoch: {e+1} Loss/Test: {eval_loss}, Loss/Test: {train_loss},␣
 ↪Acc/Test: {eval_acc}")
    train_bar.set_postfix({"Loss/Train": train_loss, "Loss/Test": eval_loss,␣
 ↪"Acc/Test": eval_acc})
```

```
Some weights of the model checkpoint at albert-base-v2 were not used when
initializing AlbertModel: ['predictions.bias', 'predictions.LayerNorm.bias',
'predictions.decoder.bias', 'predictions.dense.bias',
'predictions.dense.weight', 'predictions.LayerNorm.weight',
'predictions.decoder.weight']
- This IS expected if you are initializing AlbertModel from the checkpoint of a
model trained on another task or with another architecture (e.g. initializing a
BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing AlbertModel from the checkpoint
of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
WARNING:datasets.builder:No config specified, defaulting to: emotion/split
WARNING:datasets.builder:Found cached dataset emotion
(/root/.cache/huggingface/datasets/dair-ai___emotion/split/1.0.0/cca5efe2dfeb58c
1d098e0f9eeb200e9927d889b5a03c67097275dfb5fe463bd)
WARNING:datasets.builder:No config specified, defaulting to: emotion/split
WARNING:datasets.builder:Found cached dataset emotion
(/root/.cache/huggingface/datasets/dair-ai___emotion/split/1.0.0/cca5efe2dfeb58c
1d098e0f9eeb200e9927d889b5a03c67097275dfb5fe463bd)
```

```
Epoch:   0%|          | 0/6 [00:00<?, ?it/s]
```

```
Training loss @ step 5: 1.698062539100647
Training loss @ step 10: 1.6465108871459961
Training loss @ step 15: 1.6809547424316407
    Epoch: 1 Loss/Test: 1.5663838684558868, Loss/Test: 1.665993593633175,
Acc/Test: 31.0
Training loss @ step 5: 1.5532905101776122
Training loss @ step 10: 1.5964767932891846
Training loss @ step 15: 1.5702354431152343
    Epoch: 2 Loss/Test: 1.5090010166168213, Loss/Test: 1.582436464726925,
```

```
Acc/Test: 44.5
Training loss @ step 5: 1.54567232131958
Training loss @ step 10: 1.46856632232666
Training loss @ step 15: 1.4598652362823485
    Epoch: 3 Loss/Test: 1.4301005899906158, Loss/Test: 1.4890755414962769,
Acc/Test: 52.0
Training loss @ step 5: 1.4560445070266723
Training loss @ step 10: 1.3393847942352295
Training loss @ step 15: 1.6580770492553711
    Epoch: 4 Loss/Test: 1.3566592633724213, Loss/Test: 1.467187874019146,
Acc/Test: 47.0
Training loss @ step 5: 1.2413426876068114
Training loss @ step 10: 1.387000846862793
Training loss @ step 15: 1.3542288303375245
    Epoch: 5 Loss/Test: 1.2181251645088196, Loss/Test: 1.316026285290718,
Acc/Test: 58.5
Training loss @ step 5: 1.2831413984298705
Training loss @ step 10: 1.2567926883697509
Training loss @ step 15: 1.1835227847099303
    Epoch: 6 Loss/Test: 1.044311136007309, Loss/Test: 1.2381494604051113,
Acc/Test: 60.5
```