Fine-tuning with custom datasets

Note

The datasets used in this tutorial are available and can be more easily accessed using the 🤗 Datasets library. We do not use this library to access the datasets here since this tutorial meant to illustrate how to work with your own data. A brief of introduction can be found at the end of the tutorial in the section "Using the 🤗 Datasets & Metrics library".

This tutorial will take you through several examples of using 🤗 Transformers models with your own datasets. The guide shows one of many valid workflows for using these models and is meant to be illustrative rather than definitive. We show examples of reading in several data formats, preprocessing the data for several types of tasks, and then preparing the data into PyTorch/TensorFlow `Dataset` objects which can easily be used either with `Trainer` / `TFTrainer` or with native PyTorch/TensorFlow.

We include several examples, each of which demonstrates a different type of common downstream task:

- Sequence Classification with IMDb Reviews
- Token Classification with W-NUT Emerging Entities
- Question Answering with SQuAD 2.0
- Additional Resources

Sequence Classification with IMDb Reviews

> **Note**
>
> This dataset can be explored in the Hugging Face model hub (IMDb), and can be alternatively downloaded with the 🤗 Datasets library with `load_dataset("imdb")`.

In this example, we'll show how to download, tokenize, and train a model on the IMDb reviews dataset. This task takes the text of a review and requires the model to predict whether the sentiment of the review is positive or negative. Let's start by downloading the dataset from the Large Movie Review Dataset webpage.

```
wget http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
tar -xf aclImdb_v1.tar.gz
```

This data is organized into `pos` and `neg` folders with one text file per example. Let's write a function that can read this in.

```
from pathlib import Path

def read_imdb_split(split_dir):
    split_dir = Path(split_dir)
    texts = []
    labels = []
    for label_dir in ["pos", "neg"]:
        for text_file in (split_dir/label_dir).iterdir():
            texts.append(text_file.read_text())
            labels.append(0 if label_dir is "neg" else 1)

    return texts, labels

train_texts, train_labels = read_imdb_split('aclImdb/train')
test_texts, test_labels = read_imdb_split('aclImdb/test')
```

We now have a train and test dataset, but let's also also create a validation set which we can use for for evaluation and tuning without tainting our test set results. Sklearn has a convenient utility for creating such splits:

```
from sklearn.model_selection import train_test_split
train_texts, val_texts, train_labels, val_labels = train_test_split(train_texts, train_labels, test_size=.2)
```

Alright, we've read in our dataset. Now let's tackle tokenization. We'll eventually train a classifier using pre-trained DistilBert, so let's use the DistilBert tokenizer.

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
```

Now we can simply pass our texts to the tokenizer. We'll pass `truncation=True` and `padding=True`, which will ensure that all of our sequences are padded to the same length and are truncated to be no longer model's maximum input length. This will allow us to feed batches of sequences into the model at the same time.

```
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)
```

Now, let's turn our labels and encodings into a Dataset object. In PyTorch, this is done by subclassing a `torch.utils.data.Dataset` object and implementing `__len__` and `__getitem__`. In TensorFlow, we pass our input encodings and labels to the `from_tensor_slices` constructor method. We put the data in this format so that the data can be easily batched such that each key in the batch encoding corresponds to a named parameter of the `forward()` method of the model we will train.

PyTorch TensorFlow

```
import torch

class IMDbDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

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

train_dataset = IMDbDataset(train_encodings, train_labels)
val_dataset = IMDbDataset(val_encodings, val_labels)
test_dataset = IMDbDataset(test_encodings, test_labels)
```

Now that our datasets our ready, we can fine-tune a model either with the 🤗 `Trainer`/ `TFTrainer` or with native PyTorch/TensorFlow. See training.

Fine-tuning with Trainer

The steps above prepared the datasets in the way that the trainer is expected. Now all we need to do is create a model to fine-tune, define the `TrainingArguments`/ `TFTrainingArguments` and instantiate a `Trainer`/ `TFTrainer`.

PyTorch TensorFlow

```python
from transformers import DistilBertForSequenceClassification, Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    logging_steps=10,
)

model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")

trainer = Trainer(
    model=model,                         # the instantiated 🤗 Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=val_dataset             # evaluation dataset
)

trainer.train()
```

Fine-tuning with native PyTorch/TensorFlow

We can also train use native PyTorch or TensorFlow:

PyTorch TensorFlow

```python
from torch.utils.data import DataLoader
from transformers import DistilBertForSequenceClassification, AdamW
```

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased')
model.to(device)
model.train()

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)

optim = AdamW(model.parameters(), lr=5e-5)

for epoch in range(3):
    for batch in train_loader:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs[0]
        loss.backward()
        optim.step()

model.eval()
```

Token Classification with W-NUT Emerging Entities

Note

This dataset can be explored in the Hugging Face model hub (WNUT-17), and can be alternatively downloaded with the 🤗 Datasets library with `load_dataset("wnut_17")`.

Next we will look at token classification. Rather than classifying an entire sequence, this task classifies token by token. We'll demonstrate how to do this with Named Entity Recognition, which involves identifying tokens which correspond to a predefined set of "entities". Specifically, we'll use the W-NUT Emerging and Rare entities corpus. The data is given as a collection of pre-tokenized documents where each token is assigned a tag.

Let's start by downloading the data.

```
wget http://noisy-text.github.io/2017/files/wnut17train.conll
```

In this case, we'll just download the train set, which is a single text file. Each line of the file contains either (1) a word and tag separated by a tab, or (2) a blank line indicating the end of a document. Let's write a function to read this in. We'll take in the file path and return `token_docs` which is a list of lists of token strings, and `token_tags` which is a list of lists of tag strings.

```
from pathlib import Path
import re

def read_wnut(file_path):
    file_path = Path(file_path)

    raw_text = file_path.read_text().strip()
    raw_docs = re.split(r'\n\t?\n', raw_text)
    token_docs = []
    tag_docs = []
    for doc in raw_docs:
        tokens = []
        tags = []
        for line in doc.split('\n'):
            token, tag = line.split('\t')
            tokens.append(token)
            tags.append(tag)
        token_docs.append(tokens)
        tag_docs.append(tags)

    return token_docs, tag_docs

texts, tags = read_wnut('wnut17train.conll')
```

Just to see what this data looks like, let's take a look at a segment of the first document.

```
>>> print(texts[0][10:17], tags[0][10:17], sep='\n')
['for', 'two', 'weeks', '.', 'Empire', 'State', 'Building']
['O', 'O', 'O', 'O', 'B-location', 'I-location', 'I-location']
```

`location` is an entity type, `B-` indicates the beginning of an entity, and `I-` indicates consecutive positions of the same entity ("Empire State Building" is considered one entity). `O` indicates the token does not correspond to any entity.

Now that we've read the data in, let's create a train/validation split:

```
from sklearn.model_selection import train_test_split
train_texts, val_texts, train_tags, val_tags = train_test_split(texts, tags, test_size=.2)
```

Next, let's create encodings for our tokens and tags. For the tags, we can start by just create a simple mapping which we'll use in a moment:

```
unique_tags = set(tag for doc in tags for tag in doc)
tag2id = {tag: id for id, tag in enumerate(unique_tags)}
id2tag = {id: tag for tag, id in tag2id.items()}
```

To encode the tokens, we'll use a pre-trained DistilBert tokenizer. We can tell the tokenizer that we're dealing with ready-split tokens rather than full sentence strings by passing `is_split_into_words=True`. We'll also pass `padding=True` and `truncation=True` to pad the sequences to be the same length. Lastly, we can tell the model to return information about the tokens which are split by the wordpiece tokenization process, which we will need in a moment.

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-cased')
train_encodings = tokenizer(train_texts, is_split_into_words=True, return_offsets_mapping=True, padding=True,
val_encodings = tokenizer(val_texts, is_split_into_words=True, return_offsets_mapping=True, padding=True, trun
```

Great, so now our tokens are nicely encoded in the format that they need to be in to feed them into our DistilBert model below.

Now we arrive at a common obstacle with using pre-trained models for token-level classification: many of the tokens in the W-NUT corpus are not in DistilBert's vocabulary. Bert and many models like it use a method called WordPiece Tokenization, meaning that single words are split into multiple tokens such that each token is likely to be in the vocabulary. For example, DistilBert's tokenizer would split the Twitter handle `@huggingface` into the tokens `['@', 'hugging', '##face']`. This is a problem for us because we have exactly one tag per token. If the tokenizer splits a token into multiple sub-tokens, then we will end up with a mismatch between our tokens and our labels.

One way to handle this is to only train on the tag labels for the first subtoken of a split token. We can do this in 🤗 Transformers by setting the labels we wish to ignore to `-100`. In the example above, if the label for `@HuggingFace` is `3` (indexing `B-corporation`), we would set the labels of `['@', 'hugging', '##face']` to `[3, -100, -100]`.

Let's write a function to do this. This is where we will use the `offset_mapping` from the tokenizer as mentioned above. For each sub-token returned by the tokenizer, the offset mapping gives us a tuple indicating the sub-token's start position and end position relative to the original token it was split from. That means that if the first position in the tuple is anything other than `0`, we will set its corresponding label to `-100`. While we're at it, we can also set labels to `-100` if the second position of the offset mapping is `0`, since this means it must be a special token like `[PAD]` or `[CLS]`.

Note

Due to a recently fixed bug, -1 must be used instead of -100 when using TensorFlow in 🤗 Transformers <= 3.02.

```
import numpy as np

def encode_tags(tags, encodings):
    labels = [[tag2id[tag] for tag in doc] for doc in tags]
    encoded_labels = []
    for doc_labels, doc_offset in zip(labels, encodings.offset_mapping):
        # create an empty array of -100
        doc_enc_labels = np.ones(len(doc_offset),dtype=int) * -100
        arr_offset = np.array(doc_offset)

        # set labels whose first offset position is 0 and the second is not 0
        doc_enc_labels[(arr_offset[:,0] == 0) & (arr_offset[:,1] != 0)] = doc_labels
        encoded_labels.append(doc_enc_labels.tolist())

    return encoded_labels

train_labels = encode_tags(train_tags, train_encodings)
val_labels = encode_tags(val_tags, val_encodings)
```

The hard part is now done. Just as in the sequence classification example above, we can create a dataset object:

PyTorch TensorFlow

```
import torch

class WNUTDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)
```

```
train_encodings.pop("offset_mapping") # we don't want to pass this to the model
val_encodings.pop("offset_mapping")
train_dataset = WNUTDataset(train_encodings, train_labels)
val_dataset = WNUTDataset(val_encodings, val_labels)
```

Now load in a token classification model and specify the number of labels:

PyTorch TensorFlow

```
from transformers import DistilBertForTokenClassification
model = DistilBertForTokenClassification.from_pretrained('distilbert-base-cased', num_labels=len(unique_tags))
```

The data and model are both ready to go. You can train the model either with `Trainer` / `TFTrainer` or with native PyTorch/TensorFlow, exactly as in the sequence classification example above.

- Fine-tuning with Trainer
- Fine-tuning with native PyTorch/TensorFlow

Question Answering with SQuAD 2.0

Note

This dataset can be explored in the Hugging Face model hub (SQuAD V2), and can be alternatively downloaded with the 🤗 Datasets library with `load_dataset("squad_v2")`.

Question answering comes in many forms. In this example, we'll look at the particular type of extractive QA that involves answering a question about a passage by highlighting the segment of the passage that answers the question. This involves fine-tuning a model which predicts a start position and an end position in the passage. We will use the Stanford Question Answering Dataset (SQuAD) 2.0.

We will start by downloading the data:

```
mkdir squad
wget https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v2.0.json -O squad/train-v2.0.json
wget https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v2.0.json -O squad/dev-v2.0.json
```

Each split is in a structured json file with a number of questions and answers for each passage (or context). We'll take this apart into parallel lists of contexts, questions, and answers (note that the contexts here are repeated since there are multiple questions per context):

```
import json
from pathlib import Path

def read_squad(path):
    path = Path(path)
    with open(path, 'rb') as f:
        squad_dict = json.load(f)

    contexts = []
    questions = []
    answers = []
    for group in squad_dict['data']:
        for passage in group['paragraphs']:
            context = passage['context']
            for qa in passage['qas']:
                question = qa['question']
                for answer in qa['answers']:
                    contexts.append(context)
                    questions.append(question)
                    answers.append(answer)

    return contexts, questions, answers

train_contexts, train_questions, train_answers = read_squad('squad/train-v2.0.json')
val_contexts, val_questions, val_answers = read_squad('squad/dev-v2.0.json')
```

The contexts and questions are just strings. The answers are dicts containing the subsequence of the passage with the correct answer as well as an integer indicating the character at which the answer begins. In order to train a model on this data we need (1) the tokenized context/question pairs, and (2) integers indicating at which *token* positions the answer begins and ends.

First, let's get the *character* position at which the answer ends in the passage (we are given the starting position). Sometimes SQuAD answers are off by one or two characters, so we will also adjust for that.

```
def add_end_idx(answers, contexts):
    for answer, context in zip(answers, contexts):
        gold_text = answer['text']
        start_idx = answer['answer_start']
        end_idx = start_idx + len(gold_text)
```

```
        # sometimes squad answers are off by a character or two – fix this
        if context[start_idx:end_idx] == gold_text:
            answer['answer_end'] = end_idx
        elif context[start_idx-1:end_idx-1] == gold_text:
            answer['answer_start'] = start_idx - 1
            answer['answer_end'] = end_idx - 1     # When the gold label is off by one character
        elif context[start_idx-2:end_idx-2] == gold_text:
            answer['answer_start'] = start_idx - 2
            answer['answer_end'] = end_idx - 2     # When the gold label is off by two characters

add_end_idx(train_answers, train_contexts)
add_end_idx(val_answers, val_contexts)
```

Now `train_answers` and `val_answers` include the character end positions and the corrected start positions. Next, let's tokenize our context/question pairs. 🤗 Tokenizers can accept parallel lists of sequences and encode them together as sequence pairs.

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

train_encodings = tokenizer(train_contexts, train_questions, truncation=True, padding=True)
val_encodings = tokenizer(val_contexts, val_questions, truncation=True, padding=True)
```

Next we need to convert our character start/end positions to token start/end positions. When using 🤗 Fast Tokenizers, we can use the built in `char_to_token()` method.

```
def add_token_positions(encodings, answers):
    start_positions = []
    end_positions = []
    for i in range(len(answers)):
        start_positions.append(encodings.char_to_token(i, answers[i]['answer_start']))
        end_positions.append(encodings.char_to_token(i, answers[i]['answer_end'] - 1))

        # if start position is None, the answer passage has been truncated
        if start_positions[-1] is None:
            start_positions[-1] = tokenizer.model_max_length
        if end_positions[-1] is None:
            end_positions[-1] = tokenizer.model_max_length

    encodings.update({'start_positions': start_positions, 'end_positions': end_positions})

add_token_positions(train_encodings, train_answers)
add_token_positions(val_encodings, val_answers)
```

Our data is ready. Let's just put it in a PyTorch/TensorFlow dataset so that we can easily use it for training. In PyTorch, we define a custom `Dataset` class. In TensorFlow, we pass a tuple of `(inputs_dict, labels_dict)` to the `from_tensor_slices` method.

PyTorch TensorFlow

```
import torch

class SquadDataset(torch.utils.data.Dataset):
    def __init__(self, encodings):
        self.encodings = encodings

    def __getitem__(self, idx):
        return {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

    def __len__(self):
        return len(self.encodings.input_ids)

train_dataset = SquadDataset(train_encodings)
val_dataset = SquadDataset(val_encodings)
```

Now we can use a DistilBert model with a QA head for training:

PyTorch TensorFlow

```
from transformers import DistilBertForQuestionAnswering
model = DistilBertForQuestionAnswering.from_pretrained("distilbert-base-uncased")
```

The data and model are both ready to go. You can train the model with `Trainer`/`TFTrainer` exactly as in the sequence classification example above. If using native PyTorch, replace `labels` with `start_positions` and `end_positions` in the training example. If using Keras's `fit`, we need to make a minor modification to handle this example since it involves multiple model outputs.

- Fine-tuning with Trainer

PyTorch TensorFlow

```
from torch.utils.data import DataLoader
from transformers import AdamW
```

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model.to(device)
model.train()

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)

optim = AdamW(model.parameters(), lr=5e-5)

for epoch in range(3):
    for batch in train_loader:
        optim.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        start_positions = batch['start_positions'].to(device)
        end_positions = batch['end_positions'].to(device)
        outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positi
        loss = outputs[0]
        loss.backward()
        optim.step()

model.eval()
```

Additional Resources

- How to train a new language model from scratch using Transformers and Tokenizers. Blog post showing the steps to load in Esperanto data and train a masked language model from scratch.
- Preprocessing. Docs page on data preprocessing.
- Training. Docs page on training and fine-tuning.

Using the 🤗 Datasets & Metrics library

This tutorial demonstrates how to read in datasets from various raw text formats and prepare them for training with 🤗 Transformers so that you can do the same thing with your own custom datasets. However, we recommend users use the 🤗 Datasets library for working with the 150+ datasets included in the hub, including the three datasets used in this tutorial. As a very brief overview, we will show how to use the Datasets library to download and prepare the IMDb dataset from the first example, Sequence Classification with IMDb Reviews.

Start by downloading the dataset:

```python
from datasets import load_dataset
train = load_dataset("imdb", split="train")
```

Each dataset has multiple columns corresponding to different features. Let's see what our columns are.

```python
>>> print(train.column_names)
['label', 'text']
```

Great. Now let's tokenize the text. We can do this using the `map` method. We'll also rename the `label` column to `labels` to match the model's input arguments.

```python
train = train.map(lambda batch: tokenizer(batch["text"], truncation=True, padding=True), batched=True)
train.rename_column_("label", "labels")
```

Lastly, we can use the `set_format` method to determine which columns and in what data format we want to access dataset elements.

PyTorch TensorFlow

```python
>>> train.set_format("torch", columns=["input_ids", "attention_mask", "labels"])
>>> {key: val.shape for key, val in train[0].items()})
{'labels': torch.Size([]), 'input_ids': torch.Size([512]), 'attention_mask': torch.Size([512])}
```

We now have a fully-prepared dataset. Check out the 🤗 Datasets docs for a more thorough introduction.

Previous

Next