lesson-04-review

June 18, 2024

[ ]:
```
!pip install kaggle
```

```
Requirement already satisfied: kaggle in /opt/conda/lib/python3.10/site-packages
(1.6.14)
Requirement already satisfied: six>=1.10 in /opt/conda/lib/python3.10/site-
packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in
/opt/conda/lib/python3.10/site-packages (from kaggle) (2024.2.2)
Requirement already satisfied: python-dateutil in
/opt/conda/lib/python3.10/site-packages (from kaggle) (2.9.0.post0)
Requirement already satisfied: requests in /opt/conda/lib/python3.10/site-
packages (from kaggle) (2.32.3)
Requirement already satisfied: tqdm in /opt/conda/lib/python3.10/site-packages
(from kaggle) (4.66.4)
Requirement already satisfied: python-slugify in /opt/conda/lib/python3.10/site-
packages (from kaggle) (8.0.4)
Requirement already satisfied: urllib3 in /opt/conda/lib/python3.10/site-
packages (from kaggle) (1.26.18)
Requirement already satisfied: bleach in /opt/conda/lib/python3.10/site-packages
(from kaggle) (6.1.0)
Requirement already satisfied: webencodings in /opt/conda/lib/python3.10/site-
packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in
/opt/conda/lib/python3.10/site-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.10/site-packages (from requests->kaggle) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.10/site-
packages (from requests->kaggle) (3.6)
```

[ ]:
```
import os
iskaggle = os.environ.get('KAGGLE_KERNEL_RUN_TYPE', '')
```

[ ]:
```
execute_all = False
```

Code Summary

Book Chapter 10

[ ]:
```python
from fastai.text.all import *

if execute_all:
    path = untar_data(URLs.IMDB) # Download data in ~/.fastai/data/imdb

    ###### LM fine-tuning ######
    get_imdb = partial(get_text_files, folders=['train', 'test', 'unsup']) #␣
↪Partial function that sets default arguments for the get_text_files function
    # Create dataloaders with all our movie reviews
    dls_lm = DataBlock(
        blocks=TextBlock.from_folder(path, is_lm=True),
        get_items=get_imdb, splitter=RandomSplitter(0.1)
    ).dataloaders(path, path=path, bs=32, seq_len=80)

    # Create learner and fine-tune the language model for 1 cycle (to learn new␣
↪embeddings)
    learn = language_model_learner(
        dls_lm, AWD_LSTM, drop_mult=0.3,
        metrics=[accuracy, Perplexity()]).to_fp16()
    learn.fit_one_cycle(1, 2e-2)
    learn.save_encoder('finetuned')

    ###### Classifier fine-tuning ######
    # Create a new dataloaders using only labeled data and a category as the␣
↪target block (for classification)
    dls_clas = DataBlock(
        blocks=(TextBlock.from_folder(path, vocab=dls_lm.vocab),CategoryBlock),␣
↪# Passinfg the previously fine-tuned vocabulary: vocab=dls_lm.vocab
        get_y = parent_label,
        get_items=partial(get_text_files, folders=['train', 'test']),
        splitter=GrandparentSplitter(valid_name='test')
    ).dataloaders(path, path=path, bs=32, seq_len=72)

    # Create learner, and load the previously fine-tuned encoder into it
    learn_classifier = text_classifier_learner(dls_clas, AWD_LSTM, drop_mult=0.
↪5,
                                metrics=accuracy).to_fp16()
    learn_classifier = learn_classifier.load_encoder('finetuned')

    # Train with discriminative learning rates and gradual unfreezing
    learn.fit_one_cycle(1, 2e-2) # Most of the layers (except last one) are␣
↪frozen by default by fastai when using a pre-trained model
    learn.freeze_to(-2) # Keep all the layers frozen, except for the last 2
    learn.fit_one_cycle(1, slice(1e-2/(2.6**4),1e-2))
    learn.freeze_to(-3) # Keep all the layers frozen, except for the last 3
    learn.fit_one_cycle(1, slice(5e-3/(2.6**4),5e-3))
    learn.unfreeze() # Unfreeze the whole model
```

```
learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3))
```

NB (Getting started with NLP for absolute beginners)

```
[ ]: if execute_all:
         from pathlib import Path
         import zipfile,kaggle
         import pandas as pd
         from datasets import Dataset,DatasetDict
         from transformers import AutoModelForSequenceClassification,AutoTokenizer
         from transformers import TrainingArguments,Trainer
         import numpy as np
         import datasets
         ###### DATA PREP ######
         # Setup Kaggle and download data
         creds = ''
         cred_path = Path('~/.kaggle/kaggle.json').expanduser()
         if not cred_path.exists():
             cred_path.parent.mkdir(exist_ok=True)
             cred_path.write_text(creds)
             cred_path.chmod(0o600)
         path = Path('us-patent-phrase-to-phrase-matching')
         if not iskaggle and not path.exists():
             kaggle.api.competition_download_cli(str(path))
             zipfile.ZipFile(f'{path}.zip').extractall(path)
         if iskaggle:
             path = Path('../input/us-patent-phrase-to-phrase-matching')
             ! pip install -q datasets
         df = pd.read_csv(path/'train.csv')

         # Create the 'input' col
         df['input'] = 'TEXT1: ' + df.context + '; TEXT2: ' + df.target + '; ANC1: '␣
     ↪+ df.anchor
         ds = Dataset.from_pandas(df)

         # Download model and tokenizer
         model_nm = 'microsoft/deberta-v3-small'
         tokz = AutoTokenizer.from_pretrained(model_nm)

         # Tokenizer/Numericalizer function
         def tok_func(x):
             return tokz(x["input"])

         # Adds input_ids column with the numericalized input
         tok_ds = ds.map(tok_func, batched=True)
         tok_ds = tok_ds.rename_columns({'score':'labels'}) # Rename target column␣
     ↪to label
```

```
# Create DataSetDict by splitting the training data into train/validation␣
↪sets
dds = tok_ds.train_test_split(0.25, seed=42)

# Load and prepare the "test" separate dataset for the submission
eval_df = pd.read_csv(path/'test.csv')
eval_df['input'] = 'TEXT1: ' + eval_df.context + '; TEXT2: ' + eval_df.
↪target + '; ANC1: ' + eval_df.anchor
eval_ds = Dataset.from_pandas(eval_df).map(tok_func, batched=True)


###### DEFINE METRICS/LOSS ######
# Utility function to return correlation coefficient between two variables
def corr(x,y):
    return np.corrcoef(x,y)[0][1]

def corr_d(eval_pred):
    return {'pearson': corr(*eval_pred)}


###### TRAIN MODEL ######

# Define hyperparameters
bs = 16
epochs = 4
lr = 8e-5

# Create a TrainingArguments object for the trainer
args = TrainingArguments('outputs', learning_rate=lr, warmup_ratio=0.1,␣
↪lr_scheduler_type='cosine', fp16=True,
    evaluation_strategy="epoch", per_device_train_batch_size=bs,␣
↪per_device_eval_batch_size=bs*2,
    num_train_epochs=epochs, weight_decay=0.01, report_to='none')

# Create the model
model = AutoModelForSequenceClassification.from_pretrained(model_nm,␣
↪num_labels=1)

# Create the trainer
trainer = Trainer(model, args, train_dataset=dds['train'],␣
↪eval_dataset=dds['test'],
                  tokenizer=tokz, compute_metrics=corr_d)

# Train the model
trainer.train()
```

```
###### SUBMISSION ######

# Make predictions on the eval_ds
preds = trainer.predict(eval_ds).predictions.astype(float)
preds = np.clip(preds, 0, 1) # Clip all predicitons to 0 or 1

submission = datasets.Dataset.from_dict({
    'id': eval_ds['id'],
    'score': preds
})
#submission.to_csv('submission.csv', index=False)
```

Theory Review

Book Chapter 10

[ ]:
```
!pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
from fastbook import *
from IPython.display import display,HTML
```

[ ]:
```
# Download the IMDB movie reviews dataset
from fastai.text.all import *
path = untar_data(URLs.IMDB) # Download data in ~/.fastai/data/imdb
files = get_text_files(path, folders = ['train', 'test', 'unsup']) # Create␣
↪list of lal text files in those 3 folders
txt = files[0].open().read() # Get a sample review from the first file
files[1].open().read()
```

[ ]:
```
'Sheesh! It is amazing how much control the Hollywood establishment has over the
entire spectrum of news media. In the morning paper, I read about some new movie
for the first time ever. At noon, there it is again in a news magazine I get in
the mail. Then I see some "news" story about it at six o\'clock, and later on in
the evening there\'s some story about one of the stars, and later again, an
interview with the director and so on. The next day, the movie opens in a
theater near you… and it turns out to be one mediocre dog doo of a flick
that\'s begging seats in the "dollar theatre" a month later, only to be
forgotten by year\'s end.<br /><br />Then, there are movies like this one. <br
/><br />I\'d never heard of it when I happened by chance to see it at a
friend\'s house. <br /><br />And I\'ll never forget it. What a masterpiece!<br
/><br />If you\'re a musician, and especially if your first instrument was a
hand-me-down, you might appreciate the peculiar tendency of a musical instrument
to absorb and even accumulate the human soul, and find its way into the most
appropriate hands. That\'s what this movie is about. Although if you\'re like
me, you might think it\'s about that one hand-me-down that nobody else wanted,
```

```
that got you started, the one that years later, when you saw some kid admiring
it, you just *knew* it no longer belonged to you…<br /><br />'
```

Language model is model trained to predict the next word, based on past ones. They are trained with self-supervised learning, which means they create the label/targets automatically from the input/training data. Self-supervised learning usually used during the pre-training of the language models (not during transfer learning).

ULMFit

Universal Language Model Fine-tuning approach improves the performance of a model when using transfer learning, by fine-tuning the sequence-based pretrained language model on the corpus that it will actually be used on, before fine-tuning the classification model itself.

Text Processing

The idea behind a (next-word predictor) language model is treat text input as a big categorical variable where: - We make list of all possible levels (all words in our training texts) - Replace each level with its vocab index - Create an embedding matrix associated to the vocab - Use the embedding matrix as the first layer of the NN

The language model fine-tuning/training is done by taking all the input documents/texts and concatenating them all end to end into a single giant document which wil become the input, and the output/target will be that same giant text btu shifted right by one word.

So the language model's final vocab and embedding vectors will consist of the vocabulary learned during its pretraining PLUS the new vocabulary learned during the language-model fine-tuning phase, this wil be language specific to our corpus that the model had not seen before. So the new vocab will combine all those tokens.

Necessary Steps: - Tokenization (convert text into character/substrings/word tokens - Numericalization (create vocab list used for looking up token and their ids) - Create LM dataloader (traning data) - Train LM

Tokenization

FastAI provides consistent interface to range of external (lib) tokenizers.

The WordTokenizer() is always pointing to the current default fastai tokenizer. Ex:

[ ]:
```
spacy = WordTokenizer()
toks = first(spacy([txt]))
print(coll_repr(toks, 30))
first(spacy(['The U.S. dollar 1.00.']))
```

```
(#143) ['Jiang','Xian','uses','the','complex','backstory','of','Ling','Ling','an
d','Mao','Daobing','to','study','Mao',"'s",'"','cultural','revolution','"','(','
1966','-','1976',')','at','the','village','level','.'…]
```

[ ]:
```
(#5) ['The','U.S.','dollar','1.00','.']
```

We can also wrap the WordTokenizer() into a FastAI Tokenizer() object which provides extra functionality. It adds extra special tokens (marked by an xx suffix), like xxbos for begining of text/stream or xxmaj to indicate a capitalized word. These rules are meant to make it easier for the model to recognize important aspects of a sentence and to reduce the total vocabulary size by using special tokens to represent repeated characters or capitalized words (instead of maintaing a vocab entry for multiple repetitions or both the lower and upper case version of the same token).

[ ]:
```
tkn = Tokenizer(spacy)
print(coll_repr(tkn(txt), 31))
```

```
(#158) ['xxbos','xxmaj','jiang','xxmaj','xian','uses','the','complex','backstory
','of','xxmaj','ling','xxmaj','ling','and','xxmaj','mao','xxmaj','daobing','to',
'study','xxmaj','mao',"'s",'"','cultural','revolution','"','(','1966','-'…]
```

Here are the rules used by Tokenizer() object and their function:

- fix_html:: Replaces special HTML characters with a readable version (IMDb reviews have quite a few of these)
- replace_rep:: Replaces any character repeated three times or more with a special token for repetition (xxrep), the number of times it's repeated, then the character
- replace_wrep:: Replaces any word repeated three times or more with a special token for word repetition (xxwrep), the number of times it's repeated, then the word
- spec_add_spaces:: Adds spaces around / and #
- rm_useless_spaces:: Removes all repetitions of the space character
- replace_all_caps:: Lowercases a word written in all caps and adds a special token for all caps (xxup) in front of it
- replace_maj:: Lowercases a capitalized word and adds a special token for capitalized (xxmaj) in front of it
- lowercase:: Lowercases all text and adds a special token at the beginning (xxbos) and/or the end (xxeos)

[ ]:
```
defaults.text_proc_rules
```

[ ]:
```
[<function fastai.text.core.fix_html(x)>,
 <function fastai.text.core.replace_rep(t)>,
 <function fastai.text.core.replace_wrep(t)>,
 <function fastai.text.core.spec_add_spaces(t)>,
 <function fastai.text.core.rm_useless_spaces(t)>,
 <function fastai.text.core.replace_all_caps(t)>,
 <function fastai.text.core.replace_maj(t)>,
 <function fastai.text.core.lowercase(t, add_bos=True, add_eos=False)>]
```

Subword Tokenization

Work tokenization assumes that the language has a concept of words and that they are separated by spaces, which is not always the case (like for Chinese, Japanses, Tourkish, etc). To handle those types of languages, subwords tokenization might be used.

The idea is to analyze the corpus of documents and create a vocab from the group/sequence of letters that occur most frequently. This means we can control the size of the vocab we want: - Smaller Tokens (ex characters) = Smaller Vocabulary (only one entry for each character) ==> Slower training and inference because each input requires one token per character, so more tokens for a given sentence, and more computation for inference but lower memory requirements (smaller embedding matrix/vocab). - Bigger Tokens (ex words/subwords based on frequency) = Bigger vocabulary (there are many ways to combine characters together) ==> Inference is faster since a sentence can be represented with less tokens (because the tokens represent words or subwords and not individual characters) but requires more memory (much bigger matrix embeddings) and much more data for training.

Overall, subword tokenization provides a way to easily scale between character tokenization (i.e., using a small subword vocab) and word tokenization (i.e., using a large subword vocab), and handles every human language without needing language-specific algorithms to be developed. It can even handle other "languages" such as genomic sequences or MIDI music notation! For this reason, in the last year its popularity has soared, and it seems likely to become the most common tokenization approach (it may well already be, by the time you read this!).

Numericalization with fastai

Essentially the same thing as creating a categorical variable; make a list of all the possible unique levels (tokens) and assign an int index to each of them. This list is then used during the forward/inference pass to convert an input from a list of tokens to a list of integers.

[ ]:
```
txts = L(o.open().read() for o in files[:2000]) # Create list of strings where␣
↪each one if a review read from one of the first 2000 files
toks200 = txts[:200].map(tkn) # Use a subset of 200 of those reviews
num = Numericalize() # Initialize numericalizer. Defaults min_freq=3,␣
↪max_vocab=60000
num.setup(toks200) # Call to setup creates the vocab
coll_repr(num.vocab,20), num(toks)[:20], ' '.join(num.vocab[o] for o in␣
↪num(toks)[:20])
```

[ ]:
```
("(#2152) ['xxunk','xxpad','xxbos','xxeos','xxfld','xxrep','xxwrep','xxup','xxma
j','the',',','.','and','a','of','to','is','in','it','i'…]",
 TensorText([   0,    0, 1269,    9, 1270,    0,   14,    0,    0,   12,    0,
0,   15, 1271,    0,   22,   24,    0,  795,   24]),
 'xxunk xxunk uses the complex xxunk of xxunk xxunk and xxunk xxunk to study
xxunk \'s " xxunk revolution "')
```

min_freq=3 means that it will not add to the vocabulary any word that appears less than min_freq times in our whole corpus (training texts) and at the same time max_vocab=60000 it will only add to the vocabulary the max_vocab most frequent tokens. All tokens less than min_freq or not in the first max_vocab are replaced (by fastai) with xxunk

Batching

[ ]:
```
# Input text
stream = "In this chapter, we will go back over the example of classifying␣
↪movie reviews we studied in chapter 1 and dig deeper under the surface.␣
↪First we will look at the processing steps necessary to convert text into␣
↪numbers and how to customize it. By doing this, we'll have another example␣
↪of the PreProcessor used in the data block API.\nThen we will study how we␣
↪build a language model and train it for a while."
tokens = tkn(stream) # Tokenized the text (90 tokens)
```

```
bs,seq_len = 6,15 # Define batch size (number of streams per batch) and seq_len␣
↪(number of tokens per sequence) (6x15=90)
d_tokens = np.array([tokens[i*seq_len:(i+1)*seq_len] for i in range(bs)]) #␣
↪Creates 2D matrix (array of arrays) with 6 rows and 15 columns
df = pd.DataFrame(d_tokens) # Convert it to a pandas dataframe
display(HTML(df.to_html(index=False,header=None)))
```

<IPython.core.display.HTML object>

So we have 6 streams of 15 tokens that we then subdivide in smaller batches, in this case, seq_len = 5

[ ]:
```
#hide_input
bs,seq_len = 6,5
d_tokens = np.array([tokens[i*15:i*15+seq_len] for i in range(bs)])
df = pd.DataFrame(d_tokens)
display(HTML(df.to_html(index=False,header=None)))
```

<IPython.core.display.HTML object>

[ ]:
```
#hide_input
bs,seq_len = 6,5
d_tokens = np.array([tokens[i*15+seq_len:i*15+2*seq_len] for i in range(bs)])
df = pd.DataFrame(d_tokens)
display(HTML(df.to_html(index=False,header=None)))
```

<IPython.core.display.HTML object>

[ ]:
```
#hide_input
bs,seq_len = 6,5
d_tokens = np.array([tokens[i*15+10:i*15+15] for i in range(bs)])
df = pd.DataFrame(d_tokens)
display(HTML(df.to_html(index=False,header=None)))
```

<IPython.core.display.HTML object>

For a larger corpus, like the IMDB movie reviews, at each epoch, we start by shuffling the order of all the text documents (reviews), and then create a mega-stream by concatenating all the reviews together end to end. We divide this stream in a number of fixed-size consecutive mini-streams/batches (called the batch size). We then feed the model mini-batches that contain a part of each of the 10 streams at once, the models keeps an inner state between mini-batches, regardless of the chosen sequence length.

For the IMDB movie reviews, we numericalize our toks200 sample, and pass it to LMDataLoader which takes care of splitting the whole corpus into batches and mini-batches.

[ ]:
```
nums200 = toks200.map(num)
dl = LMDataLoader(nums200)
x,y = first(dl)
x.shape,y.shape
```

[ ]: (torch.Size([64, 72]), torch.Size([64, 72]))

[ ]:
```
len(list(dl))
```

[ ]: 14

- The batch size is 64, so we have 64 mini-streams.
- The sequence length is 72 tokens.
- There are a total of 14 batches, each containing 64 mini-streams of 72 tokens each (each mini-stream is continuous)

[ ]:
```
# Set view_all_batches to print all the rows of all the 14 batches, to
↪visualize how it is split

view_all_batches = False

if view_all_batches:
    batch_index = 0
    for batch in dl:
        print("################## NEW BATCH: ##################")
        print(batch[0].shape)
        for row in range(0,64):
            print(f">>>> NEW ROW <<<<")
            print(f"batch: {batch_index+1} / row: {row+1}")
            print(' '.join(num.vocab[o] for o in batch[0][row]))
        batch_index+=1
        print("\n\n")
```

[ ]:
```
' '.join(num.vocab[o] for o in x[0][:20])
```

[ ]: 'xxbos xxmaj xxunk xxmaj xxunk uses the complex xxunk of xxmaj xxunk xxmaj xxunk and xxmaj xxunk xxmaj xxunk to'

[ ]:
```
' '.join(num.vocab[o] for o in y[0][:20])
```

[ ]: 'xxmaj xxunk xxmaj xxunk uses the complex xxunk of xxmaj xxunk xxmaj xxunk and xxmaj xxunk xxmaj xxunk to study'

Training

Tokenization and numericalization handled automatically by the fastai TextBlock when it is passed to a DataBlock. We can pass the same arguments as we do to Tokenize() and Numericalize() above, to TextBlock itself.

[ ]:
```
get_imdb = partial(get_text_files, folders=['train', 'test', 'unsup']) #
↪Partial function that sets default arguments for the get_text_files function

dls_lm = DataBlock(
    blocks=TextBlock.from_folder(path, is_lm=True),
```

```
    get_items=get_imdb, splitter=RandomSplitter(0.1)
).dataloaders(path, path=path, bs=32, seq_len=80)
```

[ ]:
```
print(f"Training/Validation batch size: {dls_lm.train.bs=}")
print(f"Sequence length: {dls_lm.train.one_batch()[0].shape[1]=}\n")
print(f"Number of training batches: {len(dls_lm.train)=}")
print(f"Number of validation batches: {len(dls_lm.valid)=}\n")
print(f"Shape of one training/validaiton batch (input and output): {dls_lm.
 ↪train.one_batch()[0].shape}")
```

```
Training/Validation batch size: dls_lm.train.bs=32
Sequence length: dls_lm.train.one_batch()[0].shape[1]=80

Number of training batches: len(dls_lm.train)=10530
Number of validation batches: len(dls_lm.valid)=1161

Shape of one training/validaiton batch (input and output): torch.Size([32, 80])
```

[ ]:
```
dls_lm.show_batch(max_n=2)
```

```
<IPython.core.display.HTML object>
```

Fine-tuning the LM

The idea is to now convert each of the numerlicalized integer intputs into learnable embedding vectors that we can pass through an RNN (Recurrent Neural Network).

When we call language_model_learner(), it has a parameter called pretrained with a default value of True, which instructs fastai to create the learner by using a pre-trained model with the architecture AWD_LSTM (fatsai handles the specific model to use in the background).

We also pass our dls_lm dataloaders object with our IMDB movie review corpus. The learner will combine the vocabulary (new words/subwords) it sees in the movie review corpus to the pre-trained model's vocabulary. For new tokens, it will create new random embedding vectors and add them to the combined embedding matrix (from pre-trained and fine-tuning corpus).

[ ]:
```
learn = language_model_learner(
    dls_lm, AWD_LSTM, drop_mult=0.3,
    metrics=[accuracy, Perplexity()]).to_fp16()
```

- Loss function: cross-entropy (by default for classificiation)
- Accuracy metric: how often predicts next word correctly
- Perplexity metric: exponential of cross_entropy (measure of model's confidence in its predictions)

We call fit_one_cycle on the learner, so we can save intermediate results (between epochs, which fine_tune doesn't do). By default, when using a pre-trained model, the fastai learner will freeze the pre-trained parameters and only train the new embeddings (the ones that are in the IMDB movie review corpus but were not in the pre-trained model's vocabulary):

[ ]:
```
learn.fit_one_cycle(1, 2e-2)
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

To save the model, use learn.save('1epoch') which will create a file learn.path/models/1epoch.pth.

We can then load that model file into a learner with learn.load('1epoch').

[ ]:
```
learn.save('1epoch')

learn = learn.load('1epoch')
```

Once the new embeddings have been trained, we can unfreeze the rest of the pretrained model and fine-tune all of its parameters, with a lower learning rate:

[ ]:
```
if execute_all:
    learn.unfreeze()
    learn.fit_one_cycle(1, 2e-3)
```

Once we have finished fine-tuning the (next-word predictor) language model (from the pre-trained one) using our specific corpus (IMDB movie reviews), we can save the encoder of this final fine-tuned model. The encoder is essentially the model without the last layer which is task specific. In this case the last layer has a probability distribution over the entire vocabulary in order to predict the most likely next-word. For a classifier, we want to replace that last layer with one suited for our specific classification task.

[ ]:
```
learn.save_encoder('finetuned')
```

[ ]:
```
# How to use a next-word predictor model to generate text:

# We provide a seed text (beginning of sentence)
TEXT = "I liked this movie because"

# Specify when to stop generating
N_WORDS = 40

# Number of generated samples we want
N_SENTENCES = 2

# Predictions, aka generated texts
preds = [learn.predict(TEXT, N_WORDS, temperature=0.75)
         for _ in range(N_SENTENCES)]
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Classifier DataLoaders

Need to create a new DataLoader with only the labeled data (we leave out the 'unsup' folder from the IMDB movie review). The validation set is provided as a separate folder, so no need to split up the training data. This dataloader is meant for the classifier model fine-tuning, as opposed to the language model fine-tuning. Some differences:

- TextBlock.from_folder doesn't have the is_lm=True parameter, which indicates the dataloader is made of regular labeled data
- TextBlock gets passed the vocabulary created previously, so that the vocab and embeddings match

[ ]:
```
dls_clas = DataBlock(
    blocks=(TextBlock.from_folder(path, vocab=dls_lm.vocab),CategoryBlock),
    get_y = parent_label,
    get_items=partial(get_text_files, folders=['train', 'test']),
    splitter=GrandparentSplitter(valid_name='test')
).dataloaders(path, path=path, bs=32, seq_len=72)
```

[ ]:
```
dls_clas.show_batch(max_n=3)
```

```
<IPython.core.display.HTML object>
```

Create a learner with the new data block, then we load the encoder we fine-tuned in the previous section, into the learner object.

[ ]:
```
learn = text_classifier_learner(dls_clas, AWD_LSTM, drop_mult=0.5,
                                metrics=accuracy).to_fp16()

learn = learn.load_encoder('finetuned')
```

Fine-tuning the classifier

Unlike computer vision models where we train all the layers at once (the model is fully unfrozen for all the training), for NLP, we get better results by using:

- Discriminative learning rates (later layers like the classifier use a higher learning rate than early one)
- Gradual unfreezing (fine-tune with most layers frozen, and gradually unfreeze more and more layers)

[ ]:
```
if execute_all:
    learn.fit_one_cycle(1, 2e-2) # Most of the layers (except last one) are␣
 ↪frozen by default by fastai when using a pre-trained model
    learn.freeze_to(-2) # Keep all the layers frozen, except for the last 2
    learn.fit_one_cycle(1, slice(1e-2/(2.6**4),1e-2))
    learn.freeze_to(-3) # Keep all the layers frozen, except for the last 3
    learn.fit_one_cycle(1, slice(5e-3/(2.6**4),5e-3))
    learn.unfreeze() # Unfreeze the whole model
    learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3))
```

Questinnaire

- What is "self-supervised learning"?

It's a technique for training language models where the target/label is automatically derived from the input data (text) by shifting it.

- What is a "language model"?

  A LM is a model trained to predict the next word, based on past words (seed phrase)

- Why is a language model considered self-supervised?

  Because it does not require labeled data for training, it creates the labels automatically from the input text

- What are self-supervised models usually used for?

  They are mostly used as pre-trained model to be fine-tuned for other specific tasks

- Why do we fine-tune language models?

  Because they are often trained on a general corpus of text. By fine-tuning it to our specific corpus, it allows the LM to learn additional words/embeddings that were not present in the original texts

- What are the three steps to create a state-of-the-art text classifier?

  Use or train a language model on a huge data set of english documents. Then fine-tune the language model to our specific corpus. Finaly, replace that LM's last layer with our classification specific layer(s) and fine-tune the classifier.

- How do the 50,000 unlabeled movie reviews help us create a better text classifier for the IMDb dataset?

  They can be used for fine-tuning the language model (with self-supervised learning)

- What are the three steps to prepare your data for a language model?

  Tokenization, numericalization and batching of the text

- What is "tokenization"? Why do we need it?

  It's the process where we convert English words into tokens, that can be either words, sub-words or characters, that will eventually make up the model's vocabulary

- Name three different approaches to tokenization.

  Word based, sub-word based and character based

- What is xxbos?

  Indicates the beginning of a text document (a review)

- List four rules that fastai applies to text during tokenization.

  Replaces repeated characters with special tokens, replaces capitalized words/letters with special tokens, lowercases capitalized words, lowercases all caps words

- Why are repeated characters replaced with a token showing the number of repetitions and the character that's repeated?

  To reduce the vocabulary's size, while still maintaing the information of the repetition

- What is “numericalization”?

  The process of mapping tokens to integers (ids)

- Why might there be words that are replaced with the “unknown word” token?

  Those are for words that did not get added to the vocab (based on the min_freq and max_vocab parameters)

- With a batch size of 64, the first row of the tensor representing the first batch contains the first 64 tokens for the dataset. What does the second row of that tensor contain? What does the first row of the second batch contain? (Careful—students often get this one wrong! Be sure to check your answer on the book’s website.)

  With a batch-size of 64, it means each batch has 64 ministreams. Depending on the sequence length, and the length of the actual documents, the second row of that first batch would either contain part of the first review, or parts of the second review (text). The first row of the second batch, would contain thet next 64 tokens following the ones in the first row of the frist batch.

- Why do we need padding for text classification? Why don’t we need it for language modeling?

  For languaqge modeling, we concatenate all our texts together and then split them in equal sized batches. For classification, we need to associate a variable length input to an output, so we batch inputs with similar lengths together, and padd the smaller ones to match the length of the biggest input in that specific batch.

- What does an embedding matrix for NLP contain? What is its shape?

  Its a matrix of shape VOCABxEMBEDDING_SIZE where each row index corresponds to a token in the vocabulary, and contains an learnable embedding vector (often of size 512) that represents the meaning of a given token

- What is “perplexity”?

  The exponential of the cross_entropy

- Why do we have to pass the vocabulary of the language model to the classifier data block?

  To make sure we use that same token indexes that were used/learned for the LM fine-tuning

- What is “gradual unfreezing”?

  To train a model by starting with most of the layers frozen (untrainable) and gradually unfreezing more and more layers at each epoch.

- Why is text generation always likely to be ahead of automatic identification of machine-generated texts?

  Because the models used for automatic indentification of machine-generated texts can also be used to fine-tune those models further and make them harder to detect.

NB (Getting started with NLP for absolute beginners)

Data

[ ]:
```
creds = ''
cred_path = Path('~/.kaggle/kaggle.json').expanduser()
if not cred_path.exists():
    cred_path.parent.mkdir(exist_ok=True)
    cred_path.write_text(creds)
    cred_path.chmod(0o600)
```

[ ]:
```
path = Path('us-patent-phrase-to-phrase-matching')
if not iskaggle and not path.exists():
    kaggle.api.competition_download_cli(str(path))
    zipfile.ZipFile(f'{path}.zip').extractall(path)
if iskaggle:
    path = Path('../input/us-patent-phrase-to-phrase-matching')
    ! pip install -q datasets
```

[ ]:
```
!ls {path}
```

```
sample_submission.csv  test.csv  train.csv
```

[ ]:
```
df = pd.read_csv(path/'train.csv')
df
```

[ ]:
```
                     id        anchor                  target context  score
0      37d61fd2272659b1     abatement  abatement of pollution     A47   0.50
1      7b9652b17b68b7a4     abatement          act of abating     A47   0.75
2      36d72442aefd8232     abatement         active catalyst     A47   0.25
3      5296b0c19e1ce60e     abatement      eliminating process     A47   0.50
4      54c1e3b9184cb5b6     abatement           forest region     A47   0.00
…                   …           …                       …       …     …
36468  8e1386cbefd7f245  wood article          wooden article     B44   1.00
36469  42d9e032d1cd3242  wood article              wooden box     B44   0.50
36470  208654ccb9e14fa3  wood article           wooden handle     B44   0.50
36471  756ec035e694722b  wood article         wooden material     B44   0.75
36472  8d135da0b55b8c88  wood article        wooden substrate     B44   0.50

[36473 rows x 5 columns]
```

[ ]:
```
df.describe(include='object')
```

[ ]:
```
                      id                       anchor       target context
count              36473                        36473        36473   36473
unique             36473                          733        29340     106
top     8d135da0b55b8c88  component composite coating  composition     H01
freq                   1                          152           24    2186
```

Create 'input' column by combining multiple columns:

[ ]:
```
df['input'] = 'TEXT1: ' + df.context + '; TEXT2: ' + df.target + '; ANC1: ' +␣
↪df.anchor
```

[ ]:
```
df.input.head()
```

[ ]:
```
0    TEXT1: A47; TEXT2: abatement of pollution; ANC1: abatement
1            TEXT1: A47; TEXT2: act of abating; ANC1: abatement
2           TEXT1: A47; TEXT2: active catalyst; ANC1: abatement
3       TEXT1: A47; TEXT2: eliminating process; ANC1: abatement
4             TEXT1: A47; TEXT2: forest region; ANC1: abatement
Name: input, dtype: object
```

Tokenization

[ ]:
```
from datasets import Dataset,DatasetDict
ds = Dataset.from_pandas(df)
ds
```

[ ]:
```
Dataset({
    features: ['id', 'anchor', 'target', 'context', 'score', 'input'],
    num_rows: 36473
})
```

Need to download a model to use its tokenizer

[ ]:
```
from transformers import AutoModelForSequenceClassification,AutoTokenizer

model_nm = 'microsoft/deberta-v3-small'
tokz = AutoTokenizer.from_pretrained(model_nm)
tokz.tokenize("A platypus is an ornithorhynchus anatinus.")
```

```
tokenizer_config.json:   0%|          | 0.00/52.0 [00:00<?, ?B/s]
```

```
/opt/conda/lib/python3.10/site-packages/huggingface_hub/file_download.py:1132:
FutureWarning: `resume_download` is deprecated and will be removed in version
1.0.0. Downloads always resume when possible. If you want to force a new
download, use `force_download=True`.
  warnings.warn(
```

```
config.json:   0%|          | 0.00/578 [00:00<?, ?B/s]
```

```
spm.model:   0%|          | 0.00/2.46M [00:00<?, ?B/s]
```

```
/opt/conda/lib/python3.10/site-
packages/transformers/convert_slow_tokenizer.py:560: UserWarning: The
sentencepiece tokenizer that you are converting to a fast tokenizer uses the
byte fallback option which is not implemented in the fast tokenizers. In
practice this means that the fast version of the tokenizer can produce unknown
tokens whereas the sentencepiece version would have converted these unknown
tokens into a sequence of byte tokens matching the original piece of text.
  warnings.warn(
```

```
[ ]: [' A',
      ' platypus',
      ' is',
      ' an',
      ' or',
      'ni',
      'tho',
      'rhynch',
      'us',
      ' an',
      'at',
      'inus',
      '.']
```

Define a function that tokenizes the 'input' column for each data sample:

```
[ ]: def tok_func(x):
         return tokz(x["input"])
```

Apply the tok_func to all the rows in our dataset (ds), creates a new column, input_ids, which is the tokenized and numericalized version of 'input'. The tokenizer contains an indexed list of all string tokens in tokz.vocab, which is used to get the numerical ID of each token:

```
[ ]: tok_ds = ds.map(tok_func, batched=True)
```

```
Map:   0%|          | 0/36473 [00:00<?, ? examples/s]
```

```
[ ]: tok_ds[0]
```

```
[ ]: {'id': '37d61fd2272659b1',
      'anchor': 'abatement',
      'target': 'abatement of pollution',
      'context': 'A47',
      'score': 0.5,
      'input': 'TEXT1: A47; TEXT2: abatement of pollution; ANC1: abatement',
      'input_ids': [1,
       54453,
       435,
       294,
       336,
       5753,
       346,
       54453,
       445,
       294,
       47284,
       265,
       6435,
       346,
```

```
  23702,
  435,
  294,
  47284,
  2],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

[ ]:
```
tok_ds = tok_ds.rename_columns({'score':'labels'}) #To conform with␣
↪Transformers lib expected 'label' target column
```

Test and Validation sets

For the validation, set, we can define it in a DatasetDict (object that contains multiple DataSet objects) by splitting

[ ]:
```
dds = tok_ds.train_test_split(0.25, seed=42)
dds
```

[ ]:
```
DatasetDict({
    train: Dataset({
        features: ['id', 'anchor', 'target', 'context', 'labels', 'input',
'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 27354
    })
    test: Dataset({
        features: ['id', 'anchor', 'target', 'context', 'labels', 'input',
'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 9119
    })
})
```

Test set provided as a separate file and it is to be used at the very end, after trying multiple models and settling on a final one:

[ ]:
```
eval_df = pd.read_csv(path/'test.csv')
eval_df['input'] = 'TEXT1: ' + eval_df.context + '; TEXT2: ' + eval_df.target +␣
↪'; ANC1: ' + eval_df.anchor
eval_ds = Dataset.from_pandas(eval_df).map(tok_func, batched=True)
eval_df.describe()
```

```
Map:   0%|          | 0/36 [00:00<?, ? examples/s]
```

[ ]:
```
                      id          anchor                           target  \
count                 36              36                               36
unique                36              34                               36
top     4112d61851461f60  hybrid bearing  inorganic photoconductor drum
freq                   1               2                                1
```

```
       context  \
count       36
unique      29
top        G02
freq         3

                                                                   input
count                                                                 36
unique                                                                36
top     TEXT1: G02; TEXT2: inorganic photoconductor drum; ANC1: opc drum
freq                                                                   1
```

Metrics and correlation

The competition was evaluated on the Pearson correlation coefficient, r, which has a range of -1 to 1 (perfect positive correlation).

We define a corr function, which returns the correlation coefficient between two variables (it is returned as a 2x2 matrix). Then the corr_d utility function that simply wraps the returned result in a dictionary:

[ ]:
```
def corr(x,y):
    return np.corrcoef(x,y)[0][1]


def corr_d(eval_pred):
    return {'pearson': corr(*eval_pred)}
```

Training model

[ ]:
```
from transformers import TrainingArguments,Trainer
```

```
2024-06-18 14:59:05.653993: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-06-18 14:59:05.654138: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-06-18 14:59:05.833985: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
```

Define hyperparameters and create a TrainingArguments object (required for transformers):

[ ]:
```
bs = 16
epochs = 4
lr = 8e-5
```

```
args = TrainingArguments('outputs', learning_rate=lr, warmup_ratio=0.1,␣
↪lr_scheduler_type='cosine', fp16=True,
    evaluation_strategy="epoch", per_device_train_batch_size=bs,␣
↪per_device_eval_batch_size=bs*2,
    num_train_epochs=epochs, weight_decay=0.01, report_to='none')
```

```
/opt/conda/lib/python3.10/site-packages/transformers/training_args.py:1474:
FutureWarning: `evaluation_strategy` is deprecated and will be removed in
version 4.46 of  Transformers. Use `eval_strategy` instead
  warnings.warn(
```

Then we create and train the classification model:

[ ]:
```
model = AutoModelForSequenceClassification.from_pretrained(model_nm,␣
↪num_labels=1)
trainer = Trainer(model, args, train_dataset=dds['train'],␣
↪eval_dataset=dds['test'],
                  tokenizer=tokz, compute_metrics=corr_d)

trainer.train();
```

```
pytorch_model.bin:   0%|          | 0.00/286M [00:00<?, ?B/s]
```

```
/opt/conda/lib/python3.10/site-packages/torch/_utils.py:831: UserWarning:
TypedStorage is deprecated. It will be removed in the future and UntypedStorage
will be the only storage class. This should only matter to you if you are using
storages directly.  To access UntypedStorage directly, use
tensor.untyped_storage() instead of tensor.storage()
  return self.fget.__get__(instance, owner)()
Some weights of DebertaV2ForSequenceClassification were not initialized from the
model checkpoint at microsoft/deberta-v3-small and are newly initialized:
['classifier.bias', 'classifier.weight', 'pooler.dense.bias',
'pooler.dense.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
```

```
  warnings.warn('Was asked to gather along dimension 0, but all '
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/opt/conda/lib/python3.10/site-packages/torch/nn/parallel/_functions.py:68:
UserWarning: Was asked to gather along dimension 0, but all input tensors were
scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

Predictions and submission

Make predictions on the eval_ds (the test.csv file) to use for the submission. We use the clip() function to set all values greater than 1 to 1 and all negative values to 0:

[ ]:
```
preds = trainer.predict(eval_ds).predictions.astype(float)
preds = np.clip(preds, 0, 1)
preds
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
array([[0.46401793],
       [0.67237478],
       [0.57941985],
       [0.38659182],
       [0.        ],
       [0.53017342],
       [0.51792258],
       [0.        ],
       [0.26737645],
       [1.        ],
       [0.19829026],
       [0.2516216 ],
       [0.69299537],
       [0.99349993],
       [0.77207237],
       [0.41444772],
       [0.252572  ],
       [0.        ],
       [0.57144505],
       [0.35935143],
```

```
        [0.45219156],
        [0.21763106],
        [0.03066588],
        [0.2434327 ],
        [0.55008698],
        [0.        ],
        [0.        ],
        [0.        ],
        [0.        ],
        [0.60079515],
        [0.33296514],
        [0.        ],
        [0.74083984],
        [0.56434649],
        [0.38907242],
        [0.24090996]])
```

[ ]:
```
import datasets

submission = datasets.Dataset.from_dict({
    'id': eval_ds['id'],
    'score': preds
})

submission.to_csv('submission.csv', index=False)
submission[0]
```

```
Creating CSV from Arrow format:   0%|          | 0/1 [00:00<?, ?ba/s]
```

[ ]:
```
{'id': '4112d61851461f60', 'score': [0.46401792764663696]}
```