In [1]:
```
!pip install sentencepiece
```

```
Requirement already satisfied: sentencepiece in /opt/conda/lib/python3.7/site-packages (0.2.0)
```

In [2]:
```python
import torch
from transformers import T5Tokenizer, T5ForConditionalGeneration
from datasets import Dataset
from torch.utils.data import DataLoader
import json
from torch.cuda.amp import GradScaler, autocast

torch.cuda.empty_cache()  # Clear CUDA cache

class TextCompletionDataset(Dataset):
    def __init__(self, data, tokenizer, max_length=512):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.data_pairs = self.prepare_data(data, tokenizer, max_length)

    def prepare_data(self, data, tokenizer, max_length):
        input_output_pairs = []
        for idx, text in enumerate(data):
            # Split text into chunks of max_length
            chunks = [text[i:i+max_length] for i in range(0, len(text), max_length)]
            for chunk_idx, chunk in enumerate(chunks):
                if chunk_idx < len(chunks) - 1:
                    # For intermediate chunks, the output is the next chunk
                    input_text = chunk
                    output_text = chunks[chunk_idx + 1]
                    output_tokens = tokenizer.encode(output_text, add_special_tokens=False)
                else:
                    # For the last chunk, there's no output
                    continue
                # Tokenize input text
                input_tokens = tokenizer.encode(input_text, add_special_tokens=True)
                input_output_pairs.append((input_tokens, output_tokens))
        return input_output_pairs

    def __getitem__(self, idx):
        input_tokens, output_tokens = self.data_pairs[idx]
        # Handling tensors directly if working with IDs
        input_ids = torch.tensor(input_tokens, dtype=torch.long)
        labels = torch.tensor(output_tokens, dtype=torch.long)
        attention_mask = torch.ones(len(input_ids), dtype=torch.long)  # Create a mask of 1s for attention
        # Ensure all tensors are padded to the max length
        input_ids = torch.cat([input_ids, torch.zeros(self.max_length - len(input_ids), dtype=torch.long)])
        attention_mask = torch.cat([attention_mask, torch.zeros(self.max_length - len(attention_mask), dtype=torch.long)])
        labels = torch.cat([labels, torch.zeros(self.max_length - len(labels), dtype=torch.long)])
        return {
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'labels': labels
        }

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

# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize tokenizer and model
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small").to(device)

# Load and prepare data
file_paths = ["course_data/contexts_fall2023.json", "course_data/contexts_summer2023.json"]
data = []
for file_path in file_paths:
    with open(file_path, 'r', encoding='utf-8') as f:
        data += json.load(f)

dataset = TextCompletionDataset(data, tokenizer, max_length=512)
dataloader = DataLoader(dataset, batch_size=8, shuffle=True)

# Fetch the first data item
first_data_item = dataset[0]

# Decode tokens to see the actual text
input_text = tokenizer.decode(first_data_item['input_ids'], skip_special_tokens=True)
expected_output_text = tokenizer.decode(first_data_item['labels'], skip_special_tokens=True)

print("Input Text:", input_text)
print("Expected Output Text:", expected_output_text)

# Training configurations
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001) # try .001, 2e-3, 1e-3, changed from 2e-5
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.9)
scaler = GradScaler()

# Training loop
```

```python
num_epochs = 10 # try 10-15
model.train()
for epoch in range(num_epochs):
    total_loss = 0
    for batch in dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        optimizer.zero_grad()

        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

    scheduler.step()
    print(f"Epoch {epoch + 1}, Average Loss: {total_loss / len(dataloader):.4f}")

# Clear up memory
torch.cuda.empty_cache()
```

```
Input Text: Homework 1. Question 1: Extracting n-grams from a sentence. Complete the function get_ngrams, which takes a list of strings and an integer n as input, and returns padded n-grams over the list of strings.
The result should be a list of Python tuples. For example: >>> get_ngrams(["natural","language","processing"],1) [('START',), ('natural',), ('language',), ('processing',), ('STOP',)] >>> get_ngrams(["natural","langua
ge","processing"],2) ('START', 'natural'), ('natural', 'language'), ('language', 'processing
Expected Output Text: '), ('processing', 'STOP')] >>> get_ngrams(["natural","language","processing"],3) [('START', 'START', 'natural'), ('START', 'natural', 'language'), ('natural', 'language', 'processing'), ('langu
age', 'processing', 'STOP')]. Question 2: Counting n-grams in a corpus. We will work with two different data sets. The first data set is the Brown corpus, which is a sample of American written English collected in th
e 1950s. The format of the data is a plain text file brown_train.txt, containing one sentence per line
Epoch 1, Average Loss: 4.5059
Epoch 2, Average Loss: 1.2023
Epoch 3, Average Loss: 1.0796
Epoch 4, Average Loss: 0.9980
Epoch 5, Average Loss: 0.9327
Epoch 6, Average Loss: 0.8832
Epoch 7, Average Loss: 0.8385
Epoch 8, Average Loss: 0.7958
Epoch 9, Average Loss: 0.7648
Epoch 10, Average Loss: 0.7327
```

In [3]:

```python
# Another training loop for better results
for epoch in range(num_epochs):
    total_loss = 0
    for batch in dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        optimizer.zero_grad()

        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

    scheduler.step()
    print(f"Epoch {epoch + 1}, Average Loss: {total_loss / len(dataloader):.4f}")
```

```
Epoch 1, Average Loss: 0.7029
Epoch 2, Average Loss: 0.6799
Epoch 3, Average Loss: 0.6577
Epoch 4, Average Loss: 0.6384
Epoch 5, Average Loss: 0.6217
Epoch 6, Average Loss: 0.6010
Epoch 7, Average Loss: 0.5877
Epoch 8, Average Loss: 0.5777
Epoch 9, Average Loss: 0.5641
Epoch 10, Average Loss: 0.5545
```

In [4]:
```python
def test_t5_model(input_text):
    """Generates text completion from a given input using the T5 model."""
    # Encode the input text to tensor of input IDs
    encoded_input = tokenizer(input_text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    input_ids = encoded_input['input_ids'].to(device)

    # Generate outputs using the model
    generated_ids = model.generate(
        input_ids,
        max_length=320,
        num_beams=5,
        no_repeat_ngram_size=4,
        early_stopping=True,
        temperature=0.6,
        top_k=20,
        top_p=0.9
    )

    # Decode generated ids to text
    generated_text = tokenizer.decode(generated_ids[0], skip_special_tokens=True)
    return generated_text

# Test with some input text
input_text = """Homework 1. Question 1: Extracting n-grams from a sentence. Complete the function get_ngrams, which takes a list of strings and an integer n as input, and returns padded n-grams over the list of string

# print(f"Length of input: {len(tokenizer.encode(input_text))}")

generated_text = test_t5_model(input_text)

print("Input:", input_text)
print("Generated Text:", generated_text)
```

```
Input: Homework 1. Question 1: Extracting n-grams from a sentence. Complete the function get_ngrams, which takes a list of strings and an integer n as input, and returns padded n-grams over the list of strings. The r
esult should be a list of Python tuples. For example: >>> get_ngrams(["natural","language","processing"],1) [('START',), ('natural',), ('language',), ('processing',), ('STOP',)] >>> get_ngrams(["natural","languag
e","processing"],2) ('START', 'natural'), ('natural', 'language'), ('language', 'processing
Generated Text: 'language', 'processing'), ('STOP'), 'grading'),'second-to-digital ngrams') n-grams. The result should be a list of integers that result in the tuples. The input will be a string in the filename decod
er, which returns the n-words as input and a vector of lengths. Then, by the filename, the filename is the same as the one that used to be. Iterate over the course, we will be able to do so. ['START'], which is the l
ast word in the class. Now we are working with a string-save strings (which should be based on the type of the type get_ngrams used to compute the unigrams for a string (which is a random number of tokens and the las
t one from a single string. The first entry should be the first one in the list (i.e. the list is a string. Write the constructor. It takes a while. To do this. Write the first, get_n-gram tagged with the tag #, the
output should be slightly different. The output should be the one in the file if the first ever. You need to create a new list of tokens. Question 3: The answer is a bingear, which returns a different number of
```

In [5]:
```python
# Saving model & tokenizer
model.save_pretrained("./trained_completion_model")
tokenizer.save_pretrained("./trained_completion_model")
```

Out[5]:
```
('./trained_completion_model/tokenizer_config.json',
 './trained_completion_model/special_tokens_map.json',
 './trained_completion_model/spiece.model',
 './trained_completion_model/added_tokens.json')
```

In [ ]:
```python
###################
```

In [6]:
```python
!pip install txtinstruct
!pip install transformers[torch]
!pip install accelerate -U
```

```
Requirement already satisfied: txtinstruct in /opt/conda/lib/python3.7/site-packages (0.1.0)
Requirement already satisfied: txtai>=5.5.0 in /opt/conda/lib/python3.7/site-packages (from txtinstruct) (5.5.1)
Requirement already satisfied: datasets>=2.8.0 in /opt/conda/lib/python3.7/site-packages (from txtinstruct) (2.13.2)
Requirement already satisfied: tqdm>=4.48.0 in /opt/conda/lib/python3.7/site-packages (from txtinstruct) (4.62.3)
Requirement already satisfied: xxhash in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (3.4.1)
Requirement already satisfied: aiohttp in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (3.8.1)
Requirement already satisfied: pyyaml>=5.1 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (6.0)
Requirement already satisfied: fsspec[http]>=2021.11.1 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (2022.2.0)
Requirement already satisfied: numpy>=1.17 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (1.19.5)
Requirement already satisfied: importlib-metadata in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (4.11.1)
Requirement already satisfied: dill<0.3.7,>=0.3.0 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (0.3.6)
Requirement already satisfied: packaging in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (21.3)
Requirement already satisfied: pyarrow>=8.0.0 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (12.0.1)
Requirement already satisfied: pandas in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (1.3.5)
Requirement already satisfied: requests>=2.19.0 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (2.27.1)
Requirement already satisfied: multiprocess in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (0.70.14)
Requirement already satisfied: huggingface-hub<1.0.0,>=0.11.0 in /opt/conda/lib/python3.7/site-packages (from datasets>=2.8.0->txtinstruct) (0.16.4)
Requirement already satisfied: torch>=1.6.0 in /opt/conda/lib/python3.7/site-packages (from txtai>=5.5.0->txtinstruct) (1.10.0)
Requirement already satisfied: faiss-cpu>=1.7.1.post2 in /opt/conda/lib/python3.7/site-packages (from txtai>=5.5.0->txtinstruct) (1.7.4)
```

In [7]:
```python
import json
from txtinstruct.models import Instructor
import torch
import os
from txtai.embeddings import Embeddings
```

In [8]:
```python
# Load data
data = []
file_path = 'merged_data.json'  # Load all cleaned edstem_data json files
with open(file_path, encoding="utf-8") as f:
    data += json.load(f)

# Verify that data is loaded correctly and not empty
print(f"Loaded {len(data)} items from {file_path}")

# Initialize the Instructor
instructor = Instructor()

# Load embeddings
embeddings = Embeddings()
embeddings.load(provider="huggingface-hub", container="neuml/txtai-wikipedia")
```

```
Loaded 20 items from merged_data.json
```

Fetching 5 files: 100% 5/5 [00:00<00:00, 268.13it/s]

In [15]:
```python
# Call the Instructor with appropriate arguments
model, tokenizer = instructor(
    output_dir="./trained_model",
    optim="adamw_torch",
    base="./trained_completion_model", # Base model
    data=data, # Instruction-tuning dataset loaded from the JSON file
    task="sequence-sequence", # Model task
    learning_rate=5e-4, # Changed from 1e-3, 2e-4
    per_device_train_batch_size=8, # Changed from 4
    gradient_accumulation_steps=4, # Changed from 128 // 8, 32, 16
    num_train_epochs=80, # Changed from 30
    logging_steps=100,
)
tokenizer.model_max_length = 1024 # Set max input size (default is 512)
```

```
Found cached dataset generator (/home/mz2822/.cache/huggingface/datasets/generator/default-bf2e4fbe7f1e5595/0.0.0)
Loading cached processed dataset at /home/mz2822/.cache/huggingface/datasets/generator/default-bf2e4fbe7f1e5595/0.0.0/cache-4bd07289314bd20c.arrow
You're using a T5TokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster than using a method to encode the text followed by a call to the `pad` method to get a padded en
coding.
```

[2480/2480 59:45, Epoch 79/80]

Step	Training Loss
100	3.764100
200	3.218300
300	2.872200
400	2.603500
500	2.373300
600	2.167800
700	2.005600
800	1.855500
900	1.732500
1000	1.611300
1100	1.497400
1200	1.403900
1300	1.311700
1400	1.229500
1500	1.175900
1600	1.107000
1700	1.054800
1800	1.016100
1900	0.974600
2000	0.929900
2100	0.903900
2200	0.872200
2300	0.861800
2400	0.852800

In [16]:
```
path = "./trained_model"
model.save_pretrained(path)
tokenizer.save_pretrained(path)
```

Out[16]:
```
('./trained_model/tokenizer_config.json',
 './trained_model/special_tokens_map.json',
 './trained_model/spiece.model',
 './trained_model/added_tokens.json',
 './trained_model/tokenizer.json')
```

In [17]:
```
# Testing
from txtai.pipeline import Extractor
from txtai.pipeline import Sequences

# # Load statement generation model
# statements = Sequences((model, tokenizer))

def prompt(query):
    template = ("Answer the following question using only the context below. "
                "Say 'I don't have data on that' when the question can't be answered.\n"
                f"Question: {query}\n"
                "Context: The assignment focuses on n-gram extraction/counting. "
                "For Part 1, `get_ngrams` needs to generate padded n-grams from strings. "
                "Part 2 involves counting n-grams within two datasets, primarily the Brown corpus, "
                "using a lexicon for unseen words, marked as 'UNK'. The `TrigramModel` is initialized "
                "with a corpus file for lexicon collection and n-gram counting. `count_ngrams` updates "
                "frequency dictionaries for unigrams, bigrams, and trigrams. The process accommodates unseen words "
                "and efficient reading, with model testing done via `brown_test.txt` for perplexity evaluation.")
    return template

question = ("Homework 1 Question 6. Do we need to count the word 1 more than each sentence "
            "when computing perplexity? Because I think there will be a hiding STOP "
            "for each sentence. So the total word tokens is the words in document plus "
            "number of sentences. Am I understanding this correctly?")
```

In [18]:
```
# Testing without the model
extractor = Extractor(
    embeddings,
    Sequences("google/flan-t5-small")  # allenai/longformer-base-4096
)

extractor([{
    "query": f"{question}",
    "question": prompt(f"{question}")
}])
```

```
Token indices sequence length is longer than the specified maximum sequence length for this model (586 > 512). Running this sequence through the model will result in indexing errors
```

Out[18]:
```
[{'answer': 'Yes'}]
```

In [19]:
```
# Testing with the model
extractor = Extractor(
    embeddings,
    Sequences((model, tokenizer))
)

extractor([{
    "query": f"{question}",
    "question": prompt(f"{question}")
}])
```

Out[19]:
```
[{'answer': 'You need to include the STOP (or dictionary name) in the sentence. The lexicon is a set of words that appear in the lexicon. The corpus reader automatically selects the most likely "unigram" symbol. So i
t will be counted on the total number of words.'}]
```

In [ ]:
```

```