Assignment 3

Question 1.2.1

Question: Which class does LanguageModelingDataset inherit from?

Answer: Inherits from PyTorch's 'Dataset' class, and implements the necessary methods including __init__, __len__, __getitem__ etc.

Question 1.2.2

Question: What does the function lm_collate_fn do? Explain the structure of the data that results when it is called.

Answer: This function processes batches of data for language modeling tasks. Its purpose is to take a batch of variable-length sequences, pad them to make them equal in length, and convert them into PyTorch tensors that can be easily fed into a neural network. The function returns a tuple containing two tensors: torch.stack(padded_x) and torch.stack(padded_y). These tensors represent the input and target sequences for the batch, with each row of the tensors corresponding to a padded sequence of the same length.

Question 1.2.3

Question: Looking the notebook block [6], (with comment "Print out an example of the data") what does this tell you about the relationship between the input (X) and output (Y) that is sent the model for training?

Answer: The code demonstrates how the input-output pairs (X, Y) are structured for training a language model. X represents the input context, and Y represents the expected output or continuation, and the model is trained to predict Y based on the given X. The model learns to generate Y (the next token) based on the information provided by X (the preceding context).

Question 1.2.4

Question: Given one such X,Y pair, how many different training examples does it produce?

Answer: For a token ID sequence of length n, it generates n—1 X,Y pair training samples. But for a specified X,Y pair, it generate only one training sample.

Question 1.2.5

Question: In the generate function in the file model.py what is the default method for how the generated word is chosen – i.e. based on the model output probabilities?

Answer: In default method, which is do_sample=False, the indices are sampled from the multinomial probability distribution located in the corresponding row of output probabilities using torch.multinomial method.

Question 1.2.6

Question: What are the two kinds of heads that model.py can put on to the transformer model? Show (reproduce) all the lines of code that implement this functionality and indicate which method(s) they come from.

Answer: Language modeling head and Classification head in the __init__() method of class GPT()

```
self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
self.classifier_head = nn.Linear(config.n_embd, config.n_classification_class, bias=True)
```

Question 1.2.7

Question: How are the word embeddings initialized prior to training?

Answer: In class GPT(nn.Module), by calling function _init_weights(*self*, *module*), the embedding layers are initialized with mean weight = 0.0 and standard deviation = 0.02.

Question 1.2.8

Question: What is the name of the object that contains the positional embeddings?

Answer: GPT.transformer.wpe

Question 1.2.9

Question: How are the positional embeddings initialized prior to training?

Answer: While the embedding layer different in sizes, the positional embeddings initialized prior to training

Question 1.2.10

Question: Which module and method implement the skip connections in the transformer block? Give the line(s) of code that implement this code.

```
    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlpf(self.ln_2(x))
        return x
```

Answer: method forward(x) in module Block(nn.module)

Question 2.1

Question: Run the code up to the line trainer.run() and make sure it functions. Report the value of the loss.

```
iter_dt 0.00ms; iter 0: train loss 10.82358
iter_dt 18.54ms; iter 100: train loss 6.03126
iter_dt 18.45ms; iter 200: train loss 2.49938
iter_dt 18.06ms; iter 300: train loss 1.45750
iter_dt 19.77ms; iter 400: train loss 0.84829
iter_dt 19.21ms; iter 500: train loss 0.83599
iter_dt 18.90ms; iter 600: train loss 0.72285
iter_dt 18.72ms; iter 700: train loss 0.75713
iter_dt 19.75ms; iter 800: train loss 0.66587
iter_dt 18.22ms; iter 900: train loss 0.56146
...
iter_dt 18.61ms; iter 2600: train loss 0.65961
iter_dt 17.90ms; iter 2700: train loss 0.84658
iter_dt 17.65ms; iter 2800: train loss 0.65053
iter_dt 18.28ms; iter 2900: train loss 0.67797
```

Question 2.2

Question: Run the two code snippets following the training that calls the generate function. What is the output for each? Why does the the latter parts of the generation not make sense?

```
'He and I can hold a dog. cat. cat and dog'

'She rubs a cat and dog. dog. cat. cat'
```

Answer:

I think the reason is that when the generated tokens are already the end of a sentence and there still some words to reach max length, so the model keep providing the most common last few words of a sentence.

Question 2.3

Question: Modify the generate function so that it outputs the probability of each generated word. Show the output along with these probabilities for the two examples, and then one of your own choosing.

```python
# Use the trained language model to predict a sequence of words following a few words
encoded_prompt = train_dataset.tokenizer("He and I").to(trainer.device)
generated_sequence, probs = trainer.model.modified_generate_1(encoded_prompt, trainer.device, temperature=0.8, max_new_tokens=10)
train_dataset.tokenizer.decode(generated_sequence[0])
```

[15] ✓ 0.0s Python

```
'He and I can hold a dog. dog. cat. cat'
```

```python
torch.round(probs, decimals=3)
```

[16] ✓ 0.0s Python

```
tensor([[1.0000, 1.0000, 1.0000, 0.3650, 0.5280, 0.5440, 0.5910, 0.9980, 0.4950,
         0.9410, 0.5670, 0.5420, 0.7700]])
```

```python
# Another example
encoded_prompt = train_dataset.tokenizer("She rubs").to(trainer.device)
generated_sequence, probs = trainer.model.modified_generate_1(encoded_prompt, trainer.device, temperature=0.6, max_new_tokens=10)
train_dataset.tokenizer.decode(generated_sequence[0])
```

[17] ✓ 0.0s Python

```
'She rubs a dog. cat. cat. dog and dog'
```

```python
torch.round(probs, decimals=3)
```

[18] ✓ 0.0s Python

```
tensor([[1.0000, 1.0000, 1.0000, 0.4050, 0.6510, 0.8100, 0.9640, 0.9990, 0.6730,
         0.5090, 0.5240, 0.6770, 1.0000]])
```

My own example:

```python
# My example
encoded_prompt = train_dataset.tokenizer("I rub").to(trainer.device)
generated_sequence, probs = trainer.model.modified_generate_1(encoded_prompt, trainer.device, temperature=0.6, max_new_tokens=10)
train_dataset.tokenizer.decode(generated_sequence[0])
```

[20] ✓ 0.0s Python

```
'I rub a dog. cat.. cat and dog.'
```

```python
torch.round(probs, decimals=3)
```

[21] ✓ 0.0s Python

```
tensor([[1.0000, 1.0000, 0.5100, 0.5960, 0.5500, 0.8380, 0.9710, 0.9350, 0.7110,
         0.5680, 0.9980, 0.9160]])
```

Question 2.4

Question: Modify the generate function, again, so that it outputs, along with each word, the words that were the 6–most probable (the 6 highest probabilities) at each word output. Show the result in a table that gives all six words, along with their probabilities, in each column of the table. The number of columns in the table is the total number of generated words. For the first two words generated, explain if the probabilities in the table make sense, given the input corpus.

Answer:

```python
# Use the trained language model to predict a sequence of words following a few words
encoded_prompt = train_dataset.tokenizer("He and I").to(trainer.device)
generated_sequence, idx, probs = trainer.model.modified_generate_2(encoded_prompt, trainer.device, temperature=0.8, max_new_tokens=1
train_dataset.tokenizer.decode(generated_sequence[0])
```

[11] ✓ 0.0s Python

```
'He and I can hold a dog. cat. cat and dog'
```

```python
word = []
for i, x in enumerate(idx):
    col = []
    for j, y in enumerate(x):
        col.append(train_dataset.tokenizer.decode(idx[i][j].reshape(1)) + f"  {probs[i][j]:.3f}")
    word.append(col)

import pandas as pd
result = pd.DataFrame(data=word)
result.T
```

[12] ✓ 0.1s Python

	0	1	2	3	4	5	6	7	8	9
0	can 0.555	hold 0.679	a 0.531	dog 0.614	. 0.998	cat 0.667	. 0.986	cat 0.645	and 0.676	dog 0.991
1	hold 0.288	rub 0.319	the 0.465	cat 0.386	. 0.002	dog 0.331	and 0.012	dog 0.353	. 0.321	cat 0.005
2	rub 0.152	can 0.001	and 0.004	a 0.000	and 0.000	a 0.000	. 0.002	a 0.001	a 0.001	can 0.001
3	holds 0.003	the 0.000	hold 0.000	the 0.000	rub 0.000	the 0.000	a 0.000	the 0.001	the 0.001	rub 0.001
4	and 0.000	a 0.000	cat 0.000	and 0.000	dog 0.000	and 0.000	the 0.000	and 0.001	can 0.000	holds 0.001
5	dog 0.000	dog 0.000	holds 0.000	rub 0.000	cat 0.000	. 0.000	cat 0.000	. 0.000	. 0.000	and 0.000

This part make sense, since the first and second words are four kinds of verbs and the third one is ‘a’, ‘the’, ‘and’ which are article words.

Input corpus: “She rubs”

```python
# Another example
encoded_prompt = train_dataset.tokenizer("She rubs").to(trainer.device)
generated_sequence, idx, probs = trainer.model.modified_generate_2(encoded_prompt, trainer.device, temperature=0.6, max_new_tokens=1(
train_dataset.tokenizer.decode(generated_sequence[0])
```

[16] ✓ 0.0s Python

```
'She rubs a cat and dog. cat. cat. dog'
```

```python
word = []
for i, x in enumerate(idx):
    col = []
    for j, y in enumerate(x):
        col.append(train_dataset.tokenizer.decode(idx[i][j].reshape(1)) + f"  {probs[i][j]:.3f}")
    word.append(col)

import pandas as pd
result = pd.DataFrame(data=word)
result.T
```

[17] ✓ 0.0s Python

	0	1	2	3	4	5	6	7	8	9
0	a 0.493	cat 0.569	and 0.679	dog 0.998	. 0.999	cat 0.508	. 0.998	cat 0.724	. 0.998	dog 0.776
1	the 0.387	dog 0.431	. 0.321	cat 0.002	. 0.001	dog 0.489	and 0.001	dog 0.275	. 0.001	cat 0.224
2	and 0.120	and 0.000	a 0.000	a 0.000	and 0.000	a 0.001	. 0.000	and 0.001	rub 0.000	a 0.000
3	. 0.000	. 0.000	the 0.000	the 0.000	cat 0.000	and 0.001	rub 0.000	a 0.000	I 0.000	the 0.000
4	holds 0.000	holds 0.000	. 0.000	rub 0.000	dog 0.000	the 0.001	can 0.000	the 0.000	and 0.000	and 0.000
5	rub 0.000	rub 0.000	can 0.000	and 0.000	rub 0.000	. 0.000	holds 0.000	. 0.000	hold 0.000	holds 0.000

make sense. The first word in column 0 has choices among 2 article words and ‘and’, and the second word has choices between ‘cat’ and ‘dog’, which are following the article words.

Question 3.1

Question: Report which of these two methods you used – trained yourself, or loaded the saved model.

Answer: I used the saved model.

Question 3.2

Question: Report the examples you used and the generation results, and comment on the quality of the sentences.

Answer:
1. I used “She flips” as example, the output is:

```
number of parameters: 2.52M
running on device cpu
She flips in the coins is always room. were
```

There are some grammar errors. But mostly make sense, she flips the coins, make sense.
2. I used “I like” as example, the output is:

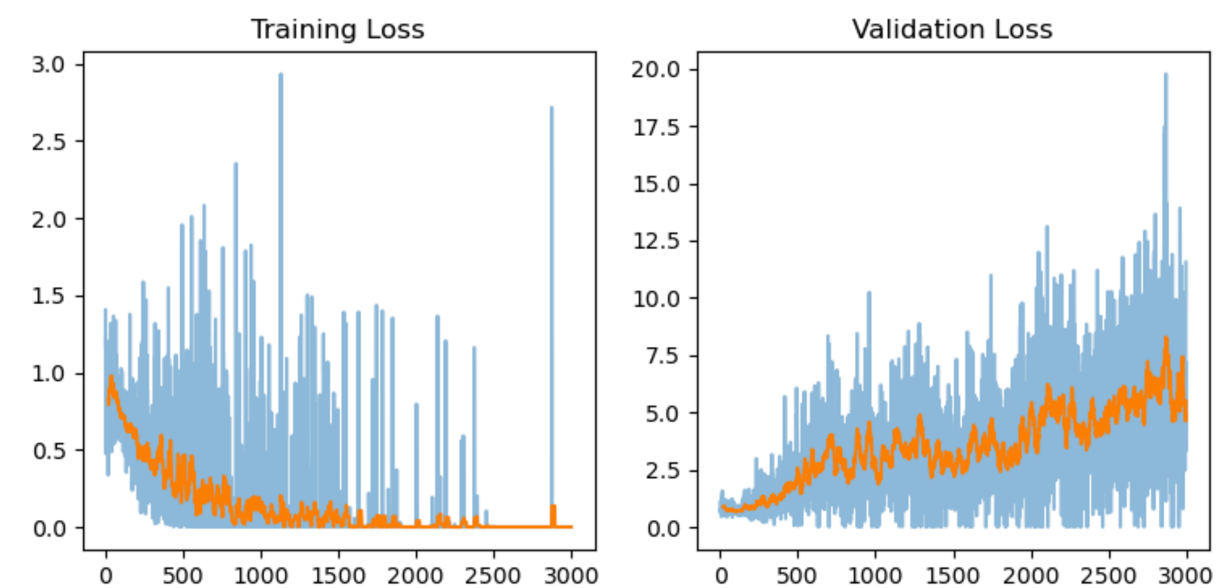
```
I like that time he was passed by Mr.
```

Make sense, both of the grammar and meaning.

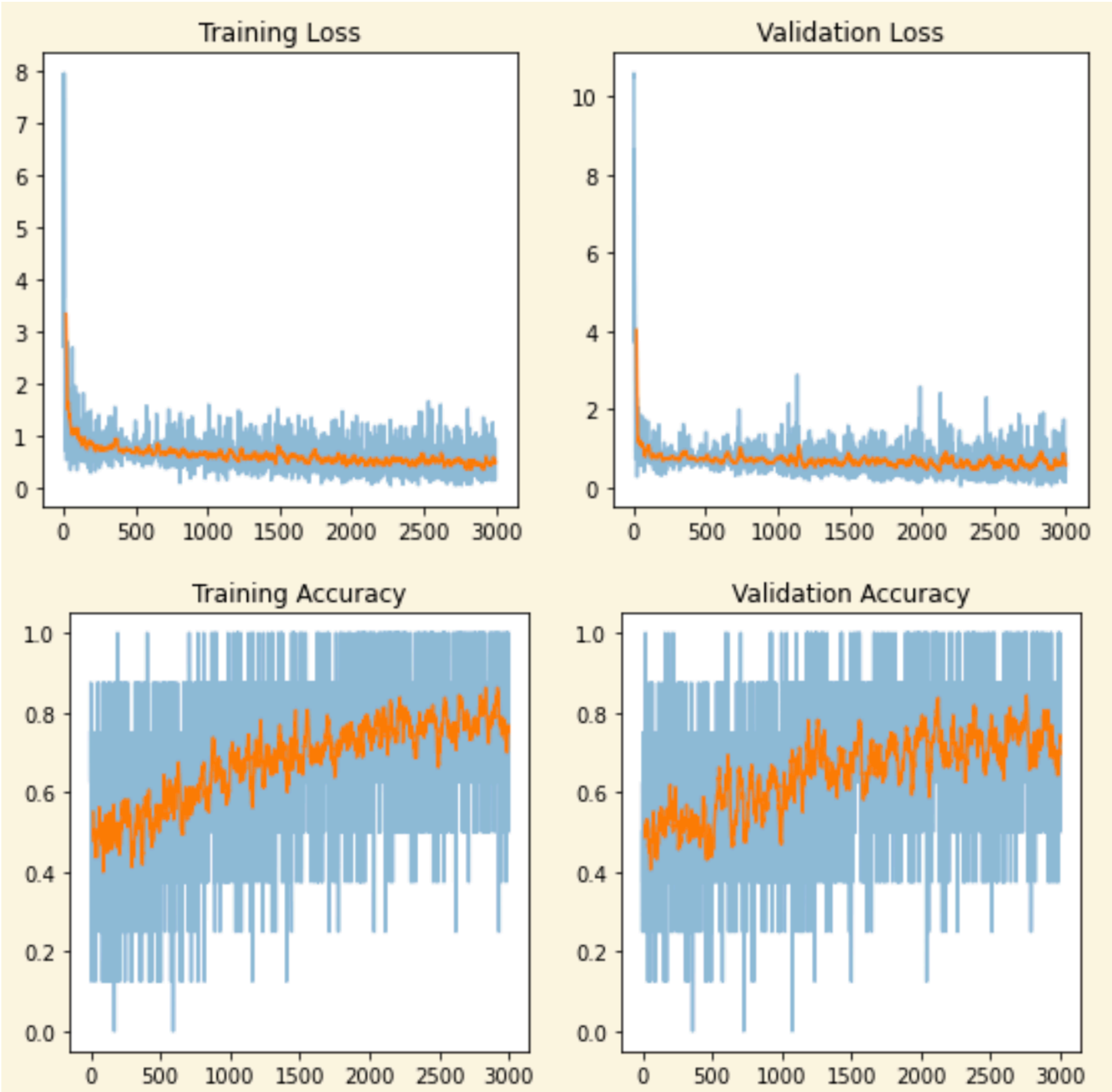
Question 3.3

Question: Next, the goal is to convert this trained model into a classifier, and fine-tune it on another dataset to perform a new task.

Answer:
I don’t know why the overfit appeared like this:



So I choose to expand the train and validation dataset, then re-trained the model, and I found that the curves are far more better



Question 4.2

Question: Report the classification accuracy on the validation set. Comment on the performance of this model: is it better than the model you fine-tuned in the previous section?

Answer:

```
100% 25257/25257 [2:21:03<00:00, 1.47s/it]

You're using a GPT2TokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster than using a meth
{'loss': 0.474, 'learning_rate': 4.9010175396919665e-05, 'epoch': 0.06}
{'loss': 0.3764, 'learning_rate': 4.8020350793839334e-05, 'epoch': 0.12}
{'loss': 0.3757, 'learning_rate': 4.7030526190759e-05, 'epoch': 0.18}
{'loss': 0.345, 'learning_rate': 4.6040701587678666e-05, 'epoch': 0.24}
{'loss': 0.3394, 'learning_rate': 4.5050876984598335e-05, 'epoch': 0.3}
{'loss': 0.3392, 'learning_rate': 4.4061052381518e-05, 'epoch': 0.36}
{'loss': 0.3207, 'learning_rate': 4.307122777843766e-05, 'epoch': 0.42}
{'loss': 0.307, 'learning_rate': 4.208140317535733e-05, 'epoch': 0.48}
{'loss': 0.3005, 'learning_rate': 4.109157857227699e-05, 'epoch': 0.53}
{'loss': 0.2921, 'learning_rate': 4.010175396919666e-05, 'epoch': 0.59}
...
{'loss': 0.1286, 'learning_rate': 2.4884190521439603e-06, 'epoch': 2.85}
{'loss': 0.1099, 'learning_rate': 1.4985944490636262e-06, 'epoch': 2.91}
{'loss': 0.1243, 'learning_rate': 5.087698459832917e-07, 'epoch': 2.97}
{'train_runtime': 8463.8209, 'train_samples_per_second': 23.872, 'train_steps_per_second': 2.984, 'train_loss': 0.20911685609664582, 'epoch
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

100% 109/109 [00:30<00:00, 3.54it/s]

Validation accuracy:  0.908256880733945
```

The final validation accuracy is 91%, it is obvious better than last section.