1.2.1 Which class does LanguageModelingDataset inherit from?
It is inherited from Dataset class in torch.utils.data library.

1.2.2 What does the function lm_collate_fn do? Explain the structure of the data that results when it is called.
The function check the maximum length x(y is the same length with x) in batch, and pad all x and y in batch with ones to reach to the maximum length. It ensures all x,y in batch share the same length.
When called, it returns padded_x and padded_y in tensor, with shape = (batch size, max length).

1.2.3 What does this tell you about the relationship between the input (X) and output (Y) that is sent the model for training?
X and Y are contain the words(in tokenized indices) in the same sentence. X will remove the punctuation word, Y will remove the first subject.

1.2.4 how many different training examples does it produce?
Given such X,Y pair, there are len(X) = 4 different training examples it produce.

1.2.5 What is the default method for how the generated word is chosen – i.e. based on the model output probabilities?
When default, <do_sample> = False, we choose idx_next using torch.topk(k=1). This means we choose the most likely element purely based on candidate's probability.

1.2.6 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.
There are lm_head and classifier_head.

```
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)
```

They used nn.Linear to build a linear transformation matrix. One in size(n_embd, vocab_size), the other in size(n_embd, n_classification_class)

1.2.7 How are the word embeddings initialized prior to training?
1. based on type_given, (takes "gpt-nano" as an example), setting the embedding dimension (n_embd) to 48.
2. use the embedding dimension to create nn.Embedding

```
wte = nn.Embedding(config.vocab_size, config.n_embd),
```

3. Initial weights on embeddings

```
self.apply(self._init_weights)
```

```
elif isinstance(module, nn.Embedding):
    torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

it is initialized with mean = 0.0 and standard deviation = 0.02

1.2.8 What is the name of the object that contains the positional embeddings?

Positional embeddings in transformer is wrapped in an object called “wpe”

```
pos_emb = self.transformer.wpe(pos) # position embeddings of shape (1, t, n_embd)
```

1.2.9 How are the positional embeddings initialized prior to training?

it is initialized with mean = 0.0 and standard deviation = 0.02, same as word embeddings.

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

In Block class, forward function:

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

It is implemented by adding input directly to the output of attn layer and mlpf layer. In case any optimization layer fails, the original input is preserved.

2.1 Report the value of the loss.

```
iter_dt 0.00ms; iter 0: train loss 10.82358
iter_dt 24.19ms; iter 100: train loss 6.03126
iter_dt 50.66ms; iter 200: train loss 2.49938
iter_dt 26.44ms; iter 300: train loss 1.45750
iter_dt 22.74ms; iter 400: train loss 0.84829
iter_dt 22.42ms; iter 500: train loss 0.83599
iter_dt 21.11ms; iter 600: train loss 0.72285
iter_dt 29.15ms; iter 700: train loss 0.75713
iter_dt 25.79ms; iter 800: train loss 0.66587
iter_dt 26.54ms; iter 900: train loss 0.56146
iter_dt 28.05ms; iter 1000: train loss 0.63020
iter_dt 26.05ms; iter 1100: train loss 0.77099
iter_dt 27.80ms; iter 1200: train loss 0.60608
iter_dt 52.22ms; iter 1300: train loss 0.58718
iter_dt 21.50ms; iter 1400: train loss 0.60112
iter_dt 26.49ms; iter 1500: train loss 0.67451
iter_dt 28.51ms; iter 1600: train loss 0.66049
iter_dt 24.58ms; iter 1700: train loss 0.61786
iter_dt 23.35ms; iter 1800: train loss 0.58953
iter_dt 24.36ms; iter 1900: train loss 0.60130
iter_dt 21.36ms; iter 2000: train loss 0.57404
iter_dt 25.79ms; iter 2100: train loss 0.65719
iter_dt 28.10ms; iter 2200: train loss 0.68743
iter_dt 23.35ms; iter 2300: train loss 0.60095
iter_dt 26.64ms; iter 2400: train loss 0.58640
iter_dt 23.39ms; iter 2500: train loss 0.61831
iter_dt 32.55ms; iter 2600: train loss 0.65961
iter_dt 24.67ms; iter 2700: train loss 0.84658
iter_dt 29.28ms; iter 2800: train loss 0.65053
iter_dt 25.95ms; iter 2900: train loss 0.67797
```

2.2 What is the output for each? Why does the latter parts of the generation not make sense?

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

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

We set the max_new_tokens = 10, it means that we generally ask model to generate 10 more words after the given prompt. When the model already generate the end of the sentence but still have room to reach token length required, it provides some tokens that are commonly used in the end of the sentence. It makes the sentence generated not coherent.

2.3 Show the output along with these probabilities for the two examples, and then one of your own choosing.

```
'He and I can hold the cat. rub the dog and and'
```

tensor([[0.7641],
[0.5231],
[0.5647],
[0.5045],
[0.9956],
[0.3603],
[0.5756],
[0.5402],
[0.9941],
[0.5454]])

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

tensor([[0.3967],
[0.6271],
[0.5313],
[0.9991],
[0.9975],
[0.5801],
[0.9994],
[0.6612],
[0.9949],
[0.9926]])

I chose to generate for “I rub”

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

tensor([[0.5916],
[0.5807],
[0.7259],
[0.9881],
[0.9996],
[0.4601],
[0.3865],
[0.3439],
[0.9989],
[0.9996]])

2.4 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.

Analyze the 10 words generated after “He and I”:

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

Words generated	6 highest probabilities	Corresponding 6 words
can	tensor([7.6410e-01, 1.4153e-01, 8.7845e-02, 4.2865e-03, 1.5088e-03, 2.0270e-04])	can rub hold holds and the
rub	tensor([5.2313e-01, 4.7512e-01, 9.3587e-04, 2.6670e-04, 2.0189e-04, 6.9351e-05])	rub hold can the a cat
a	tensor([5.6471e-01, 4.3368e-01, 9.1597e-04, 2.4229e-04, 1.3242e-04, 1.0456e-04])	a the and dog holds
cat	tensor([5.0453e-01, 4.9500e-01, 1.1102e-04, 1.1012e-04, 6.0891e-05, 4.0465e-05])	cat dog the a rub.
.	tensor([9.9557e-01, 4.1294e-03, 1.5407e-04, 3.9417e-05, 3.1602e-05, 2.6550e-05])	. . and cat dog rub
a	tensor([3.6035e-01, 3.3631e-01, 2.4965e-01, 5.2823e-02, 3.8125e-04, 2.0503e-04])	a the and. dog cat
dog	tensor([5.7561e-01, 4.2391e-01, 1.3059e-04, 1.1968e-04, 4.9949e-05, 3.7253e-05])	dog cat the a hold rub
and	tensor([5.4021e-01, 4.4261e-01, 8.7183e-03, 8.1321e-03, 1.1116e-04, 7.5781e-05])	and. the a . can
dog	tensor([9.9412e-01, 4.9205e-03, 2.1691e-04, 1.6296e-04, 1.4393e-04, 1.0550e-04])	dog cat. and I hold
and	tensor([5.4544e-01, 4.4572e-01, 4.4354e-03, 3.9673e-03, 1.5962e-04, 1.1375e-04])	and. the a can .

The first two generated words are “can” and “rub”. Which makes sense given the input “He and I”, I also observe that these two words have the probabilies that are much higher than other candidates during generation.

3.1 Report which of these two methods you used.
I loaded the saved model.

3.2 Report the examples you used and the generation results, and comment on the quality of the sentences.
Input: “it is”

```
'it is rather than\n    and for some places as the Chief Clerk of'
```

Input: “one of the most”

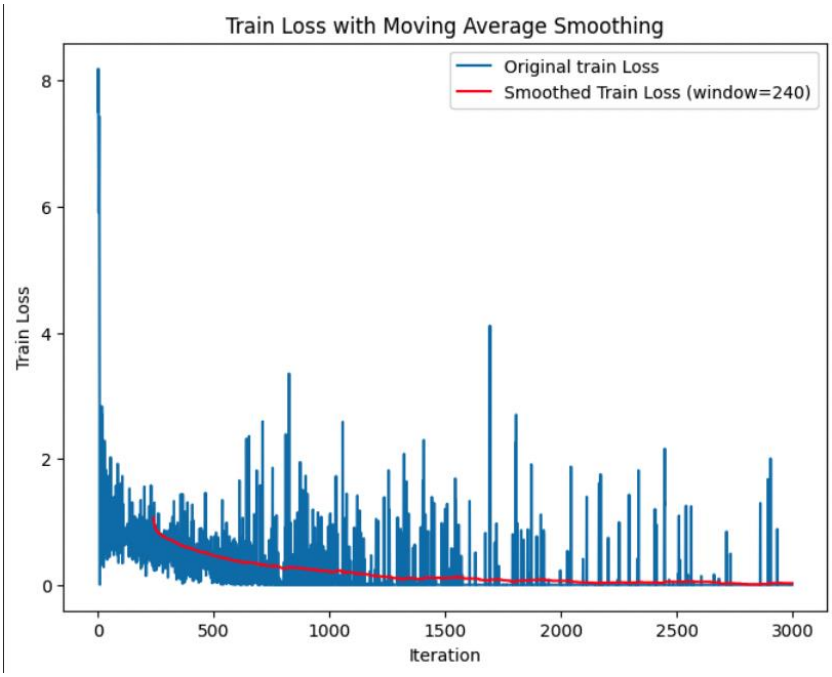
```
'one of the most interesting to the coin of them is to the\ncoins were issued in 17'
```

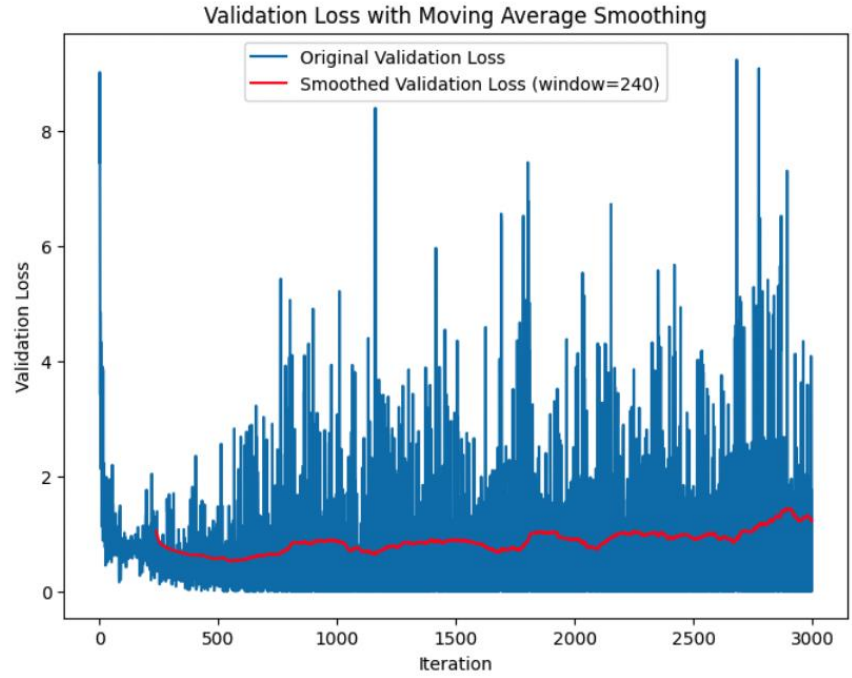
Input: “in the”

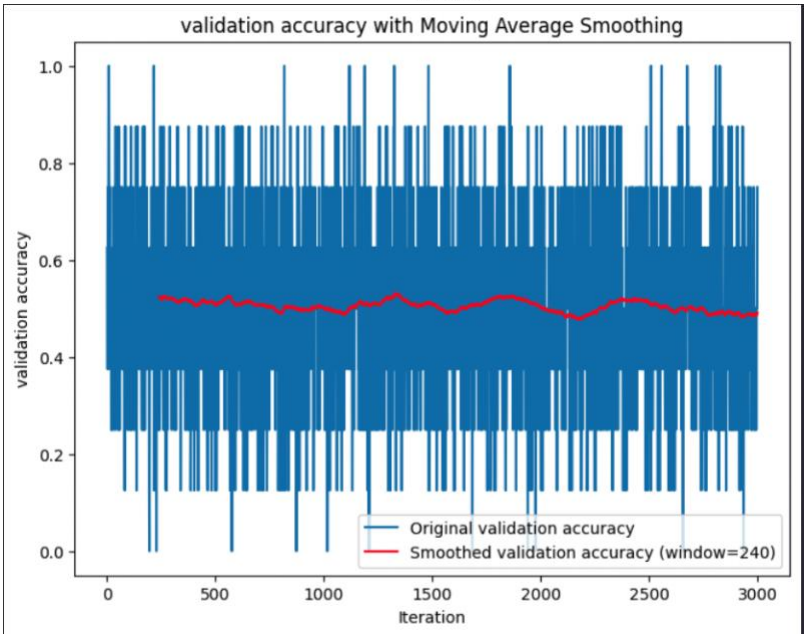
```
'in the financial Mint, and the Cabinet.”\n'
```

I choose some commonly-used phases in the largercorpus texts. I can observe that the model generated the contents based on the training dataset. For example, “coin”, “financial”, “Mint” are the words that are frequently introduced in the dataset.
Besides, the sentence generated obtain the sense of “natural language tone”. For example, “rather than”, “as of”, “one of the most interesting to”, these are commonly used "prepositional phrases" or "conjunctive phrases” in English.
However, the meanings of the sentence generated are unclear. It seems that model was trying to “stick” largercorpurs-based concepts(coins, financial) with frequently used phrases.

3.3 Report the training and validation curves for the fine-tuning, and the accuracy achieved on the validation dataset.







3.3 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 precious section?

Epoch	Training Loss	Validation Loss	Accuracy
1	1.179000	1.753914	0.537500
2	1.696100	0.922638	0.537500
3	1.294000	1.941707	0.679167
4	1.020700	1.145047	0.754167
5	0.629900	1.272566	0.791667
6	0.394100	1.296263	0.808333
7	0.319700	1.518716	0.812500
8	0.148700	1.604863	0.791667
9	0.098800	1.644217	0.829167
10	0.088400	1.665757	0.829167

```
TrainOutput(global_step=9600, training_loss=0.6770278899495801, metrics={'train_runtime': 1618.007,
'train_samples_per_second': 5.933, 'train_steps_per_second': 5.933, 'total_flos': 5016897611366400.0, 'train_loss':
0.6770278899495801, 'epoch': 10.0})
```

The final accuracy after 10 epoch run is around 0.83, which is better than the model in the previous section. GPT-nano pre trained in the previous section is smaller and lighter compared to GPT-2, so it is obvious that GPT-2 would achieve higher validation accuracy and better overall performance compared to GPT-Nano.