Assignment 3: Training a Transformer Language Model and using it for Classification

1 Karpathy's minGPT

1.1 Structure of Provided minGPT Code

In the zip file associated with this assignment, you will find a notebook LM.ipynb which imports several python files from the folder mingpt. Below we explain the role of each of this notebook and python code files. You may choose to convert your code into only python files, or perhaps one very large notebook.

1.2 Questions

1. Which class does LanguageModelingDataset inherit from? (1 point)

The LanguageModelingDataset class inherits from the Dataset class, a PyTorch class used for handling datasets.

2. What does the function lm_collate_fn do? Explain the structure of the data that results when it is called. (2 points)

The function is responsible for preparing and batching data in the corpus. Specifically, it handles the collation of each sample (pairs of input x and target y sequences) into a single batch, so sequences are of equal length by padding them.

The resulting output returns two tensors:

- Padded input tensor (padded_x): A 2D tensor of size (batch_size, maxlen) where each row is an input sequence, padded with ones where necessary.

- Padded target tensor (padded_y): A 2D tensor of size (batch_size, maxlen) where each row is the corresponding target sequence, also padded with ones.

3. 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? (1 point)

The relationship between X and Y can be summarized as a shifted target sequence, the output (Y) is essentially the input (X) shifted by one position to the right, with an additional token at the end. This indicates that the model is being trained to predict the next word in a sequence given the previous words.

The model is trained to predict the next token given the current one. For each position in X, the corresponding position in Y represents the token that the model is expected to predict.

For example, given the token "I" in X, the model should predict "rub" in Y, and so on for the rest of the sequence.

4. Given one such X,Y pair, how many different training examples does it produce? (1 point)

For this X, Y pair - I rub the dog, there are 4 training examples, one for each token in X, with the corresponding token in Y as the target.

Each token in X forms a new training example with the corresponding in Y, resulting in as many examples as there are tokens in the input sequence minus one.

5. 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? (1 point)

The default method for choosing is to select the most likely word based on the model's output probabilities. This occurs when do_sample=False set in the definition.

The function computes the model's output logits and applies a softmax function to convert these logits into probabilities. If do_sample=False (the default), then the function uses torch.topk to select the word with the highest probability by setting k=1.

The generated word is chosen deterministically by selecting the word with the highest probability (greedy search), not by random sampling.

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. (2 points)

Language Modeling (LM) head: This is used for generating logits for predicting the next token.

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

Classifier head: This is used when the model is fine-tuned for a classification task.

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

Both heads are initialized in the `__init__` method of the `GPT` class. They are also used in the `forward` method of depending on the value of the `finetune_classify` flag.

```
if not finetune_classify:
            # LM forward procedure
            logits = self.lm_head(x)
        else:
            # Finetune classify procedure
            logits = self.classifier_head(x[:, -1, :])
```

7. How are the word embeddings initialized prior to training? (1 point)

The word token embeddings are created using nn.Embedding in the GPT class constructor: wte is the embedding layer for the tokens, where config.vocab_size specifies the size of the vocabulary, and config.n_embd specifies the embedding dimension.

The weights of the embeddings are initialized using a normal distribution with a mean of 0.0 and a standard deviation of 0.02

8. What is the name of the object that contains the positional embeddings? (1 point)

```
wpe = nn.Embedding(config.block_size, config.n_embd)
```

wpe is the positional embedding.

9. How are the positional embeddings initialized prior to training? (1 point)

The weights of the wpe embedding layer are initialized from a normal distribution with a mean of 0.0 and a standard deviation of 0.02.

10. Which module and method implement the skip connections in the transformer block? Give the line(s) of code that implement this code. (1 point)

The skip connections in the transformer block are implemented in the Block module, specifically within the forward method of the Block class.

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

The first line adds the output of the causal self-attention layer to the input x, creating the first skip connection.

The second line adds the output of the feedforward layer to the result of the previous operation, creating the second skip connection.

2 Training and using a language model on a Small Corpus

After your review of the code, you're ready to train the model on the file smallsimplecorpus.txt from Assignment 1. The code as given in the notebook will train the model using that file as input.

1. Run the code up to the line trainer.run() and make sure it functions. Report the value of the loss. (1 point)

```
iter_dt 0.00ms; iter 0: train loss 10.82099
iter_dt 123.29ms; iter 100: train loss 5.97739
iter_dt 84.31ms; iter 200: train loss 2.52468
iter_dt 70.39ms; iter 300: train loss 1.45734
iter_dt 136.42ms; iter 400: train loss 0.82555
iter_dt 97.86ms; iter 500: train loss 0.81646
iter_dt 99.35ms; iter 600: train loss 0.79090
iter_dt 89.28ms; iter 700: train loss 0.67038
iter_dt 127.29ms; iter 800: train loss 0.66822
iter_dt 106.59ms; iter 900: train loss 0.56715
iter_dt 85.47ms; iter 1000: train loss 0.59438
iter_dt 142.45ms; iter 1100: train loss 0.76046
iter_dt 83.78ms; iter 1200: train loss 0.58739
iter_dt 71.13ms; iter 1300: train loss 0.59170
iter_dt 119.47ms; iter 1400: train loss 0.62839
iter_dt 84.17ms; iter 1500: train loss 0.66044
iter_dt 86.87ms; iter 1600: train loss 0.70982
iter_dt 153.18ms; iter 1700: train loss 0.75451
iter_dt 132.08ms; iter 1800: train loss 0.59662
iter_dt 97.70ms; iter 1900: train loss 0.59755
iter_dt 79.32ms; iter 2000: train loss 0.58447
iter_dt 123.53ms; iter 2100: train loss 0.58511
iter_dt 102.33ms; iter 2200: train loss 0.71029
iter_dt 105.86ms; iter 2300: train loss 0.58380
iter_dt 77.12ms; iter 2400: train loss 0.62196
iter_dt 102.27ms; iter 2500: train loss 0.62588
iter_dt 186.35ms; iter 2600: train loss 0.64252
iter_dt 90.81ms; iter 2700: train loss 0.74261
iter_dt 97.77ms; iter 2800: train loss 0.65661
iter_dt 159.49ms; iter 2900: train loss 0.68115
```

The final loss value is 0.68115 at 2900 iterations.

2. 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? (2 points)

```
[13] # 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 = trainer.model.generate(encoded_prompt, trainer.device, temperature=0.8, max_new_tokens=10)
     train_dataset.tokenizer.decode(generated_sequence[0])

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

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

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

Above are the output of the two generation functions.

The nonsensical generation observed in both sequences can be attributed to several factors:

Repetition of Words: Both outputs include repeated terms, particularly "cat" and "dog." This is likely due to the training corpus primarily containing very short sentences that often conclude with these words.

Limited Training Data: The model was trained on a restricted dataset with brief sentences averaging only 4 to 5 words. This limited variety leads the model to overfit to these common patterns, resulting in recursive repetition when attempting to generate longer sequences.

Temperature Setting: The temperature parameter influences the randomness of the word sampling. In the second prompt, with a lower temperature of 0.6, the model becomes more deterministic in its predictions. This can cause it to repeatedly select familiar words rather than exploring more diverse options, leading to recursive output like "cat. cat. cat."

Sentence Ending Patterns: The model has learned phrases like "cat." and "dog." often signal the end of sentences in the training data. Consequently, when generating text, it may default to these patterns, creating nonsensical responses when it runs out of context.

3. 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. (1 point)

Two examples from the notebook with corresponding sampling probabilities

```
He and I can (0.3781) hold (0.6837) a (0.5504) dog (0.5622) . (0.9983) cat
(0.5166) . (0.9028) cat (0.6259) and (0.7123) dog (0.8785)

She rubs a (0.4137) dog (0.5550) and (0.7097) cat (0.9945) . (0.9959) cat
(0.5336) . (0.9912) cat (0.6628) . (0.9850) cat (0.5598)
```

One new example using he holds with the following parameters

```
encoded_prompt = train_dataset.tokenizer("He holds").to(trainer.device)
```

```
generated_sequence, probabilities =
trainer.model.generate_2(encoded_prompt, trainer.device, temperature=0.8,
max_new_tokens=10)
```

```
He holds a (0.4355) dog (0.5347) and (0.6349) cat (0.9945) . (0.9897) .
(0.9737) cat (0.5151) . (0.9975) cat (0.6115) . (0.9946)
```

4. 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. (5 points)

Tokens	**Word 1**	**Word 2**	**Word 3**	**Word 4**	**Word 5**	**Word 6**
Token 1	a (0.4124)	the (0.3169)	and (0.2700)	. (0.0003)	can (0.0002)	cat (0.0002)
Token 2	dog (0.5181)	cat (0.4814)	and (0.0002)	the (0.0001)	a (0.0001)	holds (0.0001)
Token 3	and (0.7187)	. (0.2787)	rub (0.0010)	can (0.0009)	holds (0.0004)	a (0.0003)
Token 4	cat (0.9973)	dog (0.0013)	hold (0.0009)	and (0.0002)	the (0.0002)	a (0.0001)
Token 5	. (0.9983)	. (0.0013)	and (0.0003)	cat (0.0000)	the (0.0000)	dog (0.0000)
Token 6	. (0.8970)	. (0.0541)	dog (0.0330)	cat (0.0120)	I (0.0020)	the (0.0018)
Token 7	dog (0.8024)	cats (0.1970)	rub (0.0004)	. (0.0001)	hold (0.0001)	
Token 8	. (0.9969)	and (0.0023)	. (0.0007)	rub (0.0000)	cat (0.0000)	holds (0.0000)
Token 9	cat (0.8109)	dog (0.1782)	and (0.0060)	the (0.0018)	a (0.0018)	. (0.0012)
Token 10	. (0.9410)	and (0.0576)	I (0.0005)	. (0.0005)	rub (0.0003)	holds (0.0002)

Output table of “He Holds” as a starting phrase.

Tokens	**Word 1**	**Word 2**	**Word 3**	**Word 4**	**Word 5**	**Word 6**
Token 1	a (0.4571)	the (0.3330)	and (0.2098)	. (0.0001)	can (0.0000)	cat (0.0000)
Token 2	cat (0.5667)	dog (0.4333)	and (0.0000)	the (0.0000)	a (0.0000)	holds (0.0000)
Token 3	and (0.7279)	. (0.2719)	a (0.0001)	the (0.0000)	can (0.0000)	rub (0.0000)
Token 4	dog (0.9996)	cat (0.0003)	the (0.0000)	a (0.0000)	I (0.0000)	. (0.0000)
Token 5	. (0.9997)	. (0.0002)	and (0.0001)	rub (0.0000)	holds (0.0000)	can (0.0000)
Token 6	dog (0.7057)	cats (0.2942)	. (0.0000)	hold (0.0000)	rub (0.0000)	
Token 7	. (0.9446)	and (0.0553)	. (0.0000)	holds (0.0000)	rub (0.0000)	can (0.0000)
Token 8	cat (0.8837)	dog (0.1162)	. (0.0000)	the (0.0000)	and (0.0000)	a (0.0000)
Token 9	. (0.9983)	. (0.0016)	and (0.0000)	I (0.0000)	cat (0.0000)	hold (0.0000)
Token 10	cat (0.6967)	dog (0.3033)	the (0.0000)	a (0.0000)	and (0.0000)	hold (0.0000)

Output table of “She rubs” as a starting phrase.

The first token: Given the corpus, it is logical to expect an article to follow the phrase "he holds," as demonstrated by the first two options, "a" and "the." This indicates that after the verb "holds," we typically anticipate an article, a pattern evident in our limited corpus.

The second token: It follows this from the structure above. After the article, we expect either "dog" or "cat," the two primary object nouns in the corpus. This highlights that the corpus predominantly follows a simple structure of "person + action + article + dog/cat," which characterizes most of the text.

Similarly, the phrase “she rubs” adheres to a comparable pattern, where we anticipate a preposition followed by either "cat" or "dog," as indicated by the highest probability words. This suggests that the model is heavily influenced by the relatively small corpus and has learned to recognize and predict the sequence of words with a deterministic approach, effectively grasping the overall sentence structure.

Furthermore, this pattern reveals the model’s ability to generalize from the limited examples it has encountered. By identifying consistent grammatical structures, the model demonstrates a foundational understanding of syntax, which enables it to generate coherent and contextually appropriate responses. This capability, while impressive given the constraints of the dataset, underscores the significance of a well-curated corpus in training language models to enhance their predictive performance.

5. Submit your code for generate in the file A3_2.5.py. (3 points)

Code submitted with generate_2() and generate_3(), as well as the print helpers used to format the tables.

3 Training on the Bigger Corpus and Fine-Tuning Classification

In this section you will train the same model on a larger corpus, and then take the trained model and turn it into a classifier. You will then fine-tune the classifier to do sentiment analysis.

1. Modify the notebook to train on the larger corpus, by un-commenting the other two dataset lines as described in the comment at the end of the block [4] of LM.ipynb. This dataset needs significantly more training, and as suggested in the notebook, set the number of iterations (train config.max iters) to be 100,000 and the batch size to 16. Run the training; it will take significantly longer. If you're using google colab, you should be able to make the training run faster by switching the runtime type to be a GPU. You will have access to faster GPUs if you use Google Colab Pro. This code will automatically use the GPU.

If you are short on time, you can instead just load a model that has already been trained from the saved model file here. Report which of these two methods you used - trained yourself, or loaded the saved model.

I ended up using the pre-trained model, I do not think it's feasible to do this with my laptop.

2. Check that this model can generate words by seeding the generate function with a few examples different from the ones given. Report the examples you used and the generation results, and comment on the quality of the sentences. If you trained your own model, be sure to save it for re-use in the fine-tuning process below. [2 points]

```
encoded_prompt = train_dataset.tokenizer("Money can be").to(trainer.device)
generated_sequence = trainer.model.generate(encoded_prompt, trainer.device, temperature=0.6, max_new_tokens=8)
train_dataset.tokenizer.decode(generated_sequence[0])
```

```
'Money can be of Rome,\nwas in the corner'
```

```
[49] encoded_prompt = train_dataset.tokenizer("Gold is the").to(trainer.device)
     generated_sequence = trainer.model.generate(encoded_prompt, trainer.device, temperature=0.6, max_new_tokens=8)
     train_dataset.tokenizer.decode(generated_sequence[0])
```

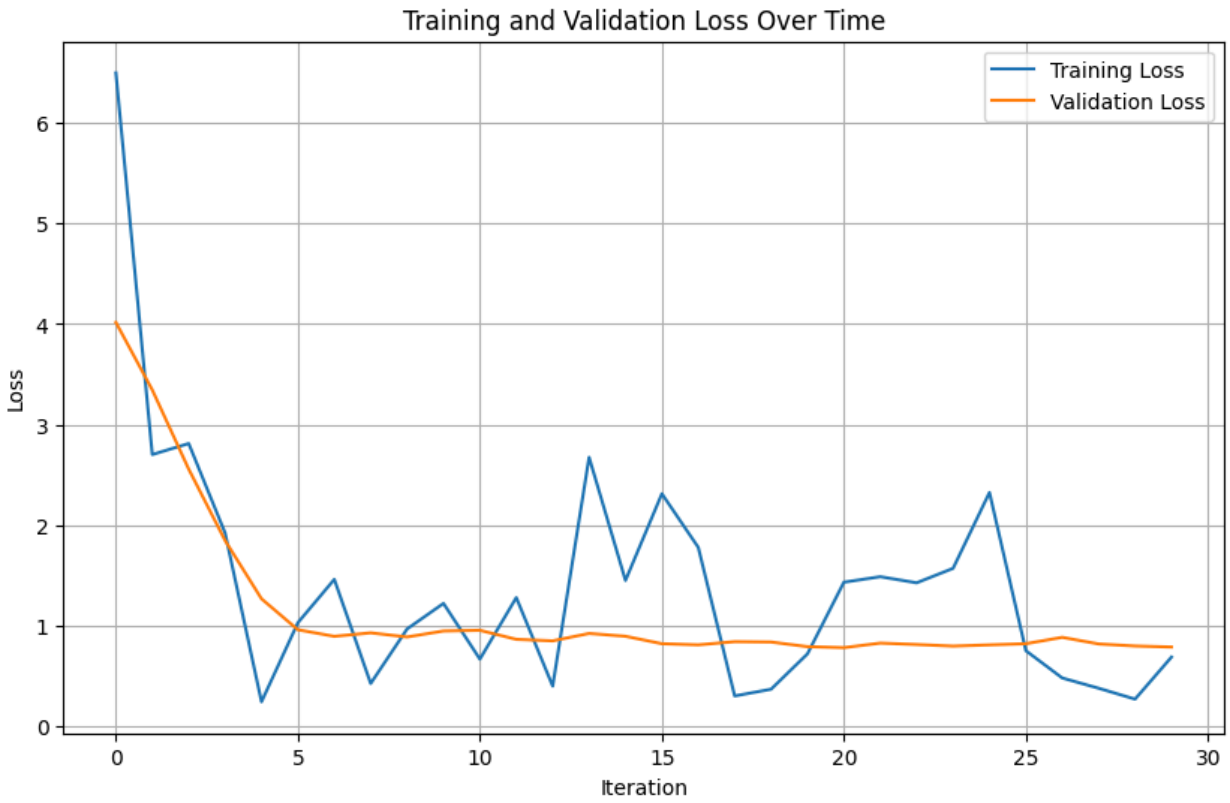
```
'Gold is the during the first term of the Mint and'
```

Overall, while the endings of the sentences still exhibit nonsensical responses similar to the prior model, it's clear that with a wider vocabulary, adjusting the temperature has a more noticeable impact on sentence generation. Increasing or decreasing the temperature significantly improved or worsened the quality of the sentences. Although the sentences still lack coherence, they do reflect the source material, which discusses various currencies and monetary usage. Common terms included references to the United States, Rome, minting, and other major topics present in the larger corpus.

3. Next, the goal is to convert this trained model into a classifier, and fine-tune it on another dataset to perform a new task. The classifier will be trained to determine the sentiment of a sequence of words predicting whether the sequence is positive sentiment (label 1) or negative (label 0). We will use the Stanford Sentiment Treebank dataset (sst2), which is part of the General Language Understanding Dataset (referred to as glue) provided by Huggingface. You'll need to install the datasets library from Huggingface using: pip install datasets. The command datasets.load dataset("glue", "sst2") loads the Stanford Sentiment Tree- bank (binary classification) dataset from the Huggingface server. Use the first 1200 samples from the train split, and then use the train test split method in scikit-learn to split them into train and validation sets. Be sure to stratify the splits.

Report the training and validation curves for the fine-tuning, and the accuracy achieved on the validation dataset. Submit your new code - the new notebook and all the .py files in a zip file named A3_3_2.zip. [6 points]

For ease of loading into Google Collab, the modified model and training code were put directly into 1 notebook. This is uploaded instead of the zip file. This will run in the original environment, no source code was changed.




```
Final Validation Accuracy: 45.00%
```

The final validation accuracy was not ideal; however, with more tuning, it could potentially improve, as the model appears to be underfitting. I mostly chose to leave it as is because the model was already running very slowly in collab and I couldn't run it locally.

4 Fine-Tuning Using Huggingface Models and Library

The fine-tuning work of in the previous section involved some complex code that needed to be customized. Fortunately, there exists a library of models and code that make this process much easier.

1. Read (and watch) the tutorial from Huggingface on how to use their model hub and their datasets, here. This tutorial shows you how to both download a pre-trained model and to fine-tune it, using Huggingface's AutoModelForSequenceClassification class. (It also shows how to do a more detailed-level training with lower-level pytorch).

2. Modify that tutorial to implement a fine-tuned sentiment classifier using the same dataset that was used in Section 3. Rather than the one used in the tutorial, use the smallest version of a pre-trained GPT2 you can find. 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? Submit your code either as a notebook named A3 4 2.ipynb or a python file with a similar name. [5 points]

This model performed significantly better than the model I ran on my own (which was not optimized at all due to time constraints)

The final classification accuracy was 0.9058.