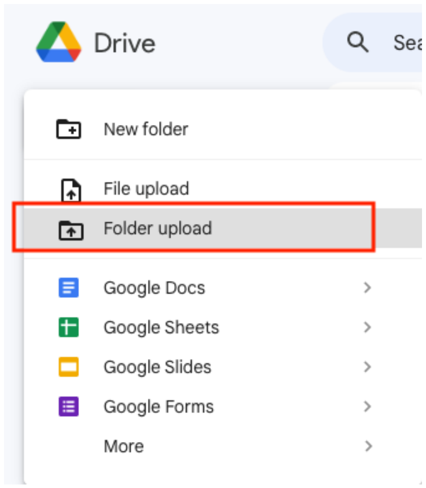
EE148b Assignment 3
Implementing a nanoGPT

May 4, 2023

Getting the Code

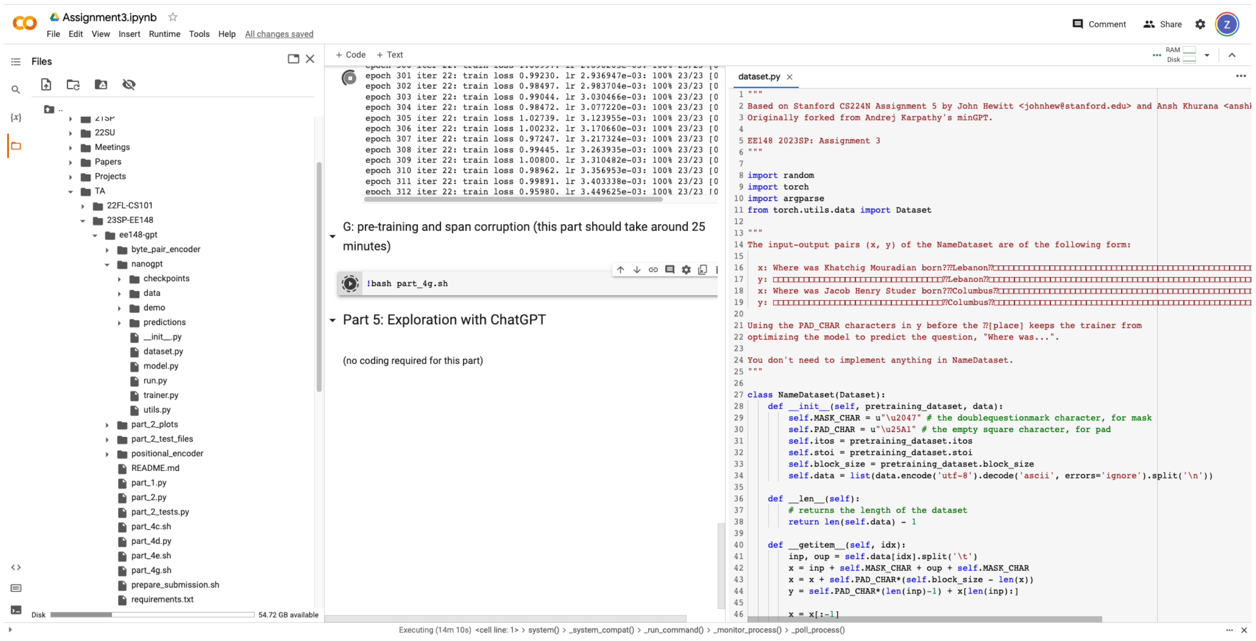
Here is the workflow we suggest you follow:

1. Download the zip file for this assignment.
2. Develop locally on your own computer.
    a. You need to run `pip install -r requirements.txt` to install necessary packages if you don't have them on your local computer.
3. Once you finish:
    a. If your local computer has a GPU, you can run all the scripts locally.
    b. Otherwise, upload the folder (without zipping it) to your personal Google Drive that has Google Colab Pro+ subscription. [Note: if you do not have a personal Google



Drive account, you will need to make one. Caltech Google account cannot be upgraded to Google Colab Pro+]. **Copy** this Colab notebook into your Google Drive account and run your code for each part.

Note: when you upload your code to Google Drive and run in Colab, you can still modify your code *without re-uploading your code or re-start your runtime*. Click on the folder icon on the left, find your file, and double click it. A text editor will show up on the right and you will be able to modify your code. Here is what the UI looks like:



```
epoch 301 iter 22: train loss 0.99230. lr 2.936947e-03: 100% 23/23 [0
epoch 302 iter 22: train loss 0.98497. lr 2.983704e-03: 100% 23/23 [0
epoch 303 iter 22: train loss 0.99044. lr 3.030466e-03: 100% 23/23 [0
epoch 304 iter 22: train loss 0.98472. lr 3.077220e-03: 100% 23/23 [0
epoch 305 iter 22: train loss 1.02739. lr 3.123955e-03: 100% 23/23 [0
epoch 306 iter 22: train loss 1.00232. lr 3.170660e-03: 100% 23/23 [0
epoch 307 iter 22: train loss 0.97247. lr 3.217324e-03: 100% 23/23 [0
epoch 308 iter 22: train loss 0.99445. lr 3.263935e-03: 100% 23/23 [0
epoch 309 iter 22: train loss 1.00800. lr 3.310482e-03: 100% 23/23 [0
epoch 310 iter 22: train loss 0.98962. lr 3.356953e-03: 100% 23/23 [0
epoch 311 iter 22: train loss 0.99891. lr 3.403338e-03: 100% 23/23 [0
epoch 312 iter 22: train loss 0.95980. lr 3.449625e-03: 100% 23/23 [0
```

G: pre-training and span corruption (this part should take around 25 minutes)

```
!bash part_4g.sh
```

Part 5: Exploration with ChatGPT

(no coding required for this part)

```
"""
Based on Stanford CS224N Assignment 5 by John Hewitt <johnhew@stanford.edu> and Ansh Khurana <anshk
Originally forked from Andrej Karpathy's minGPT.

EE148 2023SP: Assignment 3
"""

import random
import torch
import argparse
from torch.utils.data import Dataset

"""
The input-output pairs (x, y) of the NameDataset are of the following form:

  x: Where was Khatchig Mouradian born?⁇Lebanon⁇□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  y: □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□⁇Lebanon⁇□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  x: Where was Jacob Henry Studer born?⁇Columbus⁇□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  y: □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□⁇Columbus⁇□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Using the PAD_CHAR characters in y before the ⁇[place] keeps the trainer from
optimizing the model to predict the question, "Where was...".

You don't need to implement anything in NameDataset.
"""

class NameDataset(Dataset):
    def __init__(self, pretraining_dataset, data):
        self.MASK_CHAR = u"\u2047" # the doublequestionmark character, for mask
        self.PAD_CHAR = u"\u25A1" # the empty square character, for pad
        self.itos = pretraining_dataset.itos
        self.stoi = pretraining_dataset.stoi
        self.block_size = pretraining_dataset.block_size
        self.data = list(data.encode('utf-8').decode('ascii', errors='ignore').split('\n'))

    def __len__(self):
        # returns the length of the dataset
        return len(self.data) - 1

    def __getitem__(self, idx):
        inp, oup = self.data[idx].split('\t')
        x = inp + self.MASK_CHAR + oup + self.MASK_CHAR
        x = x + self.PAD_CHAR*(self.block_size - len(x))
        y = self.PAD_CHAR*(len(inp)-1) + x[len(inp):]

        x = x[:-1]
```

Executing (14m 10s) <cell line: 1> > system() > _system_compat() > _run_command() > _monitor_process() > _poll_process()

Part 1: Tokenization: Byte-Pair Encoding (BPE)

(20pts)

In this part, you will be learning and implementing a tokenization method called Byte Pair Encoding (BPE). BPE is a simple data compression algorithm first introduced in 1994 [1]. It is commonly used in many advanced NLP models, including GPT. BPE is a data-driven tokenization method that first involves a training step. With trained vocabulary, one can use it for encoding and decoding.

A. (1pt) First, let's get a feel of how a tokenizer works. OpenAI has a nice interface that allows you to interact with a tokenizer (https://platform.openai.com/tokenizer). Please give it a try. How many characters does each token roughly have?

B. (2pts) The idea of BPE is to recursively merge pairs of common tokens into a new token and add the new token into the vocabulary. The algorithm start by the vocabulary where each unique character is an individual token. For the rest of this part, we will take the following corpus as a simple example:

Here we go! Ale, ale, ale!
Go, go, go! Ale, ale, ale!

The first step one needs to do is normalization, which is a pre-processing step before tokenization. For simplicity, we convert all letters to lowercase and remove all punctuations:

here we go ale ale ale
go go go ale ale ale

The algorithm usually runs inside words and does not merge across word boundaries. The second step is called pre-tokenization, where the input corpus is first separated by white space to give a set of strings. Each string corresponds to the characters of a word plus a special end-of-word token ("_"). Draw a table of two columns where the first column is the set of characters (i.e. the tokens to start with), separated by comma, and the second column is the corresponding count, ranking in descending order of frequency.
Hint: one of the rows should be " || w, e, _ | 1 ||" because the string "we_" appears only once in the training corpus.

C. (2pts) The next step is to find the most frequent pair of consecutive tokens. Draw another table of two columns where the first column contains all unique pairs of **consecutive** tokens and the second column is the corresponding count, ranking in descending order of frequency.

D. (1pt) The first step of training a BPE is to replace the most frequent pair of tokens with a new token and add it to the vocabulary. Remember that the vocabulary initially contains all the characters in the corpus. What is the updated vocabulary? Update your table in subpart B with the new symbol.

E. (2pts) BPE repeatedly does so until k new tokens are generated, where k is a hyperparameter. If there is a tie among many pairs of tokens, just pick one of them randomly to merge. Assume k=5, what is the updated table (in the same form as in B & D) and vocabulary?

F. (5pts) In Google Colab, implement the training process of BPE.

G. (2pts) Now we consider the encoding steps of BPE. In the encoding stage, BPE recursively merge pairs of bytes that are in the vocabulary. Suppose we have the following sentence to encode

A large whale logo

Note that these words do not appear in the training corpus, but we can still encode them using the trained BPE. Complete the encoding step by step:

a, _, l, a, r, g, e, _, w, h, a, l, e, _, l, o, g, o, _

→ (TODO)

H. (4pts) In Google Colab, implement the encoding process of BPE.

I. (1pt) Compare the learned tokenization with the OpenAI BPE tokenizer (https://platform.openai.com/tokenizer). What difference do you see and why?

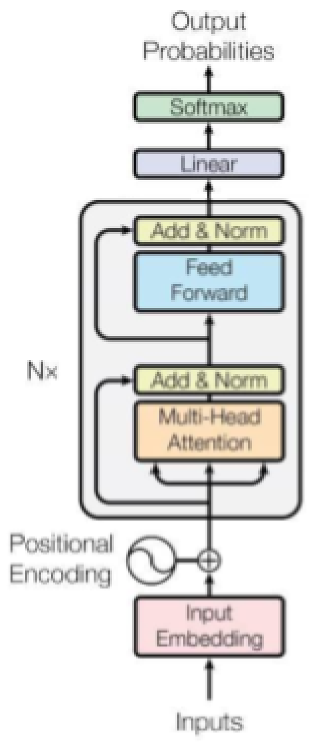
Part 1: Total 20/20 (20/76)

Part 2: Positional Embedding

(15pts)

Unit Tests Warning: The test cases might not work well if your OS is not Linux. If that is your case, run your tests on colab. You do not need to run them on a GPU.

In this part of the homework, we'll focus on understanding how positional embedding works and how to implement it.



Go to the coding part of this assignment. Open the file "positional_encoder/transformer.py". In this file, you will find a Transformer class. This class implements a simple Transformer encoder similar to that in *Attention is All You Need*. However, after the input embedding, positional embedding, and N self-attention and MLP modules, it simply has a linear output. In training, this linear output will be used with a SoftMax to predict the next token in a sequence.

Look at the forward method in the Transformer class. The input is passed through the self.text_encoder module, then the self.pos_encoder module before passing through the remaining layers. The self.text_encoder is a simple text embedder that converts a sequence of token indices into a sequence of arrays, each corresponding to the embedding of that token. The self.pos_encoder module will include the positional information in the token embedding. You will be asked to implement these modules in the next items.

A. **(3pts) Sine Cosine Positional Encoder**
   Open the file "positional_encoder/sin_cos_encoder.py". You will find one class that needs implementation: SinCosPosEncoder. You need to implement the missing part so that the positional embedding works **exactly** like that of *Attention is All You Need* [2]. Look for the details about how they calculate this encoding in the paper. Notice that the parameter d_model that is referenced in the paper is given in the input to the class constructor. Also, notice the max_seq_len parameter. This means that your encoder must work for sequences as large as that parameter.

   Look at the forward method that is already implemented. You will need to create a self.positional_encoding attribute in the SinCosPosEncoder class. This attribute will be added to the token embedding to generate the output of the module, so make sure that the shape of this attribute matches what is required in the forward function. Remember that positional_encoding does not need to have the same shape of the input. If a value will be repeated along some dimension in a tensor, remember that torch supports broadcasting.

   The last detail that you need to think about is whether the positional encoding is optimized during training in *Attention is All You Need*. To make sure your implementation correctly handles this detail, take a look at the difference between these two torch methods: nn.Module.register_parameter and nn.Module.register_buffer.

After implementing this module, your code should pass the 3 tests for part A in the part_2_tests.py file. You can run them with the command “python part_2_tests.py -v”

B. (3pts) Sine Cosine Concatenation Positional Encoder
A question that came up in class and that is very intuitive is why do we add the positional embedding. Won’t it mess with the token embedding? Isn’t it better to just concatenate it?

To implement this positional embedding, open the file positional_encoder/sin_cos_concat_encoder.py. There you will see that changes need to be implemented in both the classes SinCosConcatTextEncoder and SinCosConcatPosEncoder.

Your implementation must create an embedding for a token in the text encoder class and concatenate it with the positional embedding in the positional encoder class. The value of d_model (the number of dimensions in the final embedding with position that is passed to the transformer) should stay the same. Therefore, the text embedding should have a size of d_model/2, and the concatenated positional embedding should have a size of d_model/2, so that the final embedding has size d_model.

A few important implementation details so that your code pass the tests:1

1. Create the token embedding a similar way to the code given to you in SinCosTextEncoder, using the nn.Embedding module but with the right parameters.
2. Concatenate the positional embedding **before** the token embedding. In other words, the first d_model/2 dimensions of the final embedding should be for the positional embedding, and the last d_model/2 dimensions should be for the token embedding
3. The value of the positional embedding at each of the d_model/2 dimensions should be calculated using the same formula as in *Attention is All You Need*.

After implementing this module, your code should pass the 3 tests for part B in the part_2_tests.py file. You can run them with the command “python part_2_tests.py -v”

C. (3pts) Index Positional Encoder
An even more simple idea than concatenating a sinusoidal positional encoding is simply concatenating a one-dimensional array with the index of that token in the sequence.

To implement this positional embedding, open the file positional_encoder/index_encoder.py. There you will see that changes need to be implemented in both the classes IndexTextEncoder and IndextPosEncoder.

Your implementation must create an embedding for a token in the text encoder class and concatenate it with the positional embedding in the positional encoder class. Again, the value of d_model should stay the same. Therefore, the text embedding should have a size of d_model-1, and the concatenated positional embedding should have a size of 1, so that the final embedding has size d_model.

A few important implementation details so that your code pass the tests:

1. Create the token embedding a similar way to the code given to you in SinCosTextEncoder, using the nn.Embedding module but with the right parameters.
2. Concatenate the index **before** the token embedding. In other words, the first dimension of the final embedding should be for the position index, and the remaining d_model-1 dimensions should be for the token embedding
3. Normalize the index with max_seq_len (passed as input to the constructor) so that it falls in the interval [0, 1]. Therefore, instead of using the raw index, use index/max_seq_len.
4.

After implementing this module, your code should pass the 3 tests for part C in the part_2_tests.py file. You can run them with the command “python part_2_tests.py -v”

D. (3pts) Learned Positional Encoder

Instead of trying to figure out what is the best way to create the positional embedding, we could let the model figure it out and learn one!

To implement this positional embedding, open the file positional_encoder/learned_encoder.py. There you will see that changes need to be implemented in the class LearnedPosEncoder.

You must implement a learnable positional embedding that is added to the token embedding, just like the original SinCosPosEncoder. Look at the forward method: just like in part A, you must create an attribute self.positional_encoding that will be simply added to the token embedding. This self.positional_encoding attribute must be randomly initialized with all of its values being drawn from a uniform distribution in the interval [-1, 1]. Make sure that this attribute is a parameter that will be optimized by PyTorch.

Again, remember that torch supports broadcasting. If a value is repeated along some dimension in a tensor, it might be easier to take advantage of it.

After implementing this module, your code should pass the 4 tests for part D in the part_2_tests.py file. You can run them with the command “python part_2_tests.py -v”

E. (3pts) Experimental Results and Discussion

Run your code in Colab using a premium GPU using python part_2.py. It should take less than 10 min. This will load the WikiText-2 Dataset, tokenize it, and train a Transformer in a next-token-prediction task using your different positional embeddings. If you look at the code, you will notice that the Transformer we're using is much smaller than the one used in *Attention is All You Need*, since this would require too much computational power. Results will not approach state of the art, but will give you intuition about the different positional encoders you implemented.

For each kind of positional embedding, the code will generate a plot with train and validation loss over the 15 training epochs, as well as output the final test loss. The plots will be saved in the part_2_plots folder. You will also find a plot called p2_positional_encoding_comparison.png, with train and test losses for all the positional encoders. Include your plot in your written report and answer the following questions:

i) How does the index encoder (part C) compare with the standard SinCos positional embedding? Do your results justify why such a simple approach is not commonly used?(1-2 lines)

ii) The authors of *Attention is All You Need* comment on the results of a learned positional embedding vs sinusoidal and use it to explain why they decide to stick to sinusoidal. What do they say about the difference in performance? Do your results with a learned positional embedding (part D) corroborate their results? (1-2 lines)

iii) How does a concatenated sinusoidal positional embedding compare against the additive sinusoidal positional embedding? If their results are similar, why is one used rather than the other in real, larger models? (max 5 lines)

Looking at the visualization of the sinusoidal embedding might help you. These will be available in p2_sincos_encoding_visualization.png (entire positional embedding) and p2_sincos_encoding_visualization.png (separating even (sin) from odd (cos) indices).

Part 2: Total 15/15 (35/76)

Part 4: nanoGPT

(23pts)

In this part, you will train a nano-scale GPT (Generative Pre-trained Transformer), which is originally proposed by OpenAI in [3], to answer simple Q&A questions. Specifically, you will first pre-train a Transformer on a Wikipedia text corpus that contains knowledge related to the Q&As, and then fine-tune the model on a training set for the Q&A task. You will find that pre-training enables the model to perform substantially better on the test set than one trained from scratch.

The code you're provided with is based on Andrej Karpathy's minGPT and Stanford 224N Assignment 5. The repo is designed for educational purpose so it is relatively simple and transparent. Although we don't ask you to code a GPT from scratch, make sure you read through the code and understand how GPT works. **As in Assignment 2, you should develop locally on an IDE, then run training on Google Colab Pro+.**

A. (0pt) In the `nanogpt/demo` folder is a Jupyter notebook `play_char.ipynb` that trains and samples from a Transformer language model. Take a look at it to get somewhat familiar with how it defines and trains models. Some of the code you're writing will be similar to what you see in this notebook. No code or written answers needed for this part.

B. (0pt) The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. In particular, we consider the following simple form of question answering:

   Q: Where was [person] born?
   A: [place]

   What you will do for this part is to train a GPT model that answers the given questions. From now on, you'll be mainly working in the `nanogpt/` folder. In `nanogpt/dataset.py`, you'll find the the class `NameDataset`, which reads a TSV (tab-separated values) file of name-birthplace pairs and produces examples of the above form that we can feed to our Transformer model. Read through `NameDataset` in `nanogpt/dataset.py`. To get a sense of the examples we'll be working with, if you run the following code, it will load your `NameDataset` on the training set (`nanogpt/data/birth_places_train.txt`) and print out a few examples:

   ```
   python nanogpt/dataset.py namedata
   ```

   No code or written answers needed for this part.

C. (6pts) In `nanogpt/run.py`, you will find some skeleton code for you to pre-train, fine-tune, or evaluate a transformer model. In later subparts, you will first pre-train the model and then fine-tune it on the Q&A task. For now, you will train a Transformer that directly solves the Q&A task **without pre-training**. Focus on the fine-tuning mode (`args.mode == 'finetune'`) and implement the part marked `[part 4c]` in the code. Please use the hyperparameters given to you in the comments. Please also take a look at the evaluation code that is given to you. It samples predictions from the trained model and calculate the percentage of correct place predictions. After implementing, you should be able to run the following command:

```
bash part_4c.sh
```

Training should take around 5 minutes with Google Colab Pro+. Report your model's accuracy on the test set. Don't be surprised if it is below 5%. Take a look at the predictions saved locally in `nanogpt/predictions/p4c_finetune_without_pretrain.txt`, how do the predictions look like? Are they valid places? Do they have any pattern or look completely random?

D. (5pts) To have a better sense of how good (bad) the model is doing, let's implement some baselines. In `part_4d.py`, implement two simple baselines: 1. Make predictions that are uniformly random over the places appeared in the training set and 2. Always predict the most common place in the training set. You should make use of the `evaluate_places` function in `nanogpt/utils.py` and your solution should be a few lines for each baseline. Report the accuracies of the two baselines.

E. (6pts) Implement the pre-train part of `nanogpt/run.py` as indicated by `[part 4e]` in the code. Additionally, modify your fine-tune part to handle fine-tuning in the case with pre-training. In particular, if a path to a pre-trained model is provided as the `reading_params_path` argument, load the model before fine-tuning it on the birthplace prediction task. Pre-train your model on `nanogpt/data/wiki.txt` (which should take around 20 minutes on Google Colab Pro+), fine-tune it on `nanogpt/data/birth_places_train.txt`, and evaluate it on `nanogpt/data/birth_places_test.txt` by running the following command:

```
bash part_4e.sh
```

The loss might appear to plateau or even go up in the middle of pretraining, but don't worry about this and let it be trained for the specified number of epochs. Report the accuracy on the test set.

F. (3pts) In this part, you will implement a span corruption function for pretraining, which is introduced in the T5 paper [4]. It randomly selects spans of text in a document and replaces them with unique tokens (noising). In `nanogpt/dataset.py`, complete the implementation of the `__getitem__()` function for the dataset class `WikiDataset`. Follow the instructions provided in the comments in `dataset.py`. You can run the following command for debugging:

```
python nanogpt/dataset.py charcorruption
```

Include a screenshot of your implementation of the `__getitem__()` function for the dataset class `WikiDataset`.

G. (3pts) Now repeat subpart E with the span corruption dataset by running:

```
bash part_4g.sh
```

Report the accuracy on the test set. How good is it relative to your expectation? Compare the accuracy with those in subparts C and F and analyze your findings.

Part 4: Total: 23/23 (70/76)

References

[1] Gage, Philip (1994). "A New Algorithm for Data Compression". The C User Journal.
[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000–6010, 2017.
[3] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding with unsupervised learning. Technical report, OpenAI (2018)
[4] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research 21, 140 (2020), 1–67.

Assignment designed by Neehar Kondapaneni, Rogério Guimarães, and Zihui (Ray) Wu